Introduction to Binary Exploitation

Aaron (Esau|Jobé)

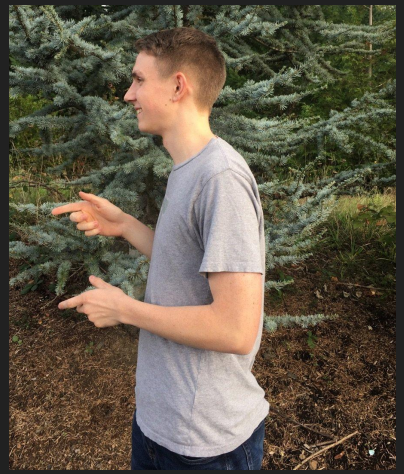
Setup

- Network
  - SSID: `pwn5` (5 GHz), `pwn` (2.4 GHz)
  - Password: `bsidespdx`
- Files
  - VM (.ova, 1.1 GB)
    - sha256sum: `3d6f5e5e16a3d9e356537c1e745c1bd69311315f96d5e31ea4ea3ffcf8430544`
    - https://tuxedo.red/bsidespdx/bsidespdx.ova
  - Slides (.pdf)
    - sha256sum: `f008d44197ae8ac5d43a29df3cd858e5d748759131ec08edcd1fe6a19d9d767d`
    - https://tuxedo.red/bsidespdx/slides.pdf
  - Booklet (.pdf)
    - sha256sum: `1d13bf77abe06e68f5c4cee4caba1dcba49eba259fd4aa7a834361135ce05898`
    - https://tuxedo.red/bsidespdx/booklet.pdf

Who are we?

Aaron Jobé (I don’t really use twitter)

Aaron Esau (@arinerron)



- We use Arch btw
- Tigard High School (Red Tuxedo CTF Team)

Structure

- Fundamentals
- Buffer Overflow
- Shellcode
- ret2libc
- Memory Leak

- Background
- The Attack
- Lab Time

- Feel free to work ahead!
- Follow along at https://tuxedo.red/bsidespdx/ or http://192.168.3.2

What is Binary Exploitation (binex/pwn/rev)?

- Deals with binaries
  - Programs that run on your computer
  - C programs (in this talk)
- Binex
  - Exploit some binary
- Rev
  - Figure out what a binary does
- Pwn
  - A combination of the two

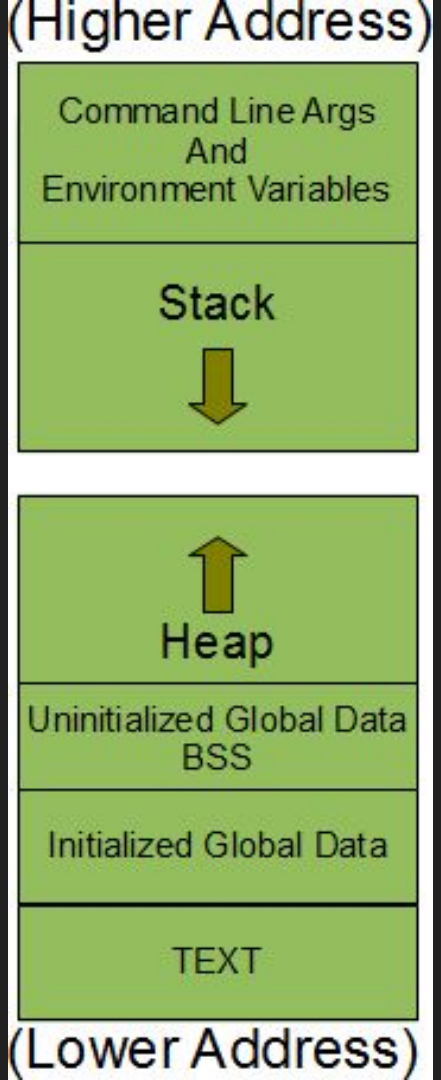
Fundamentals

Registers

- Specific-use registers
  - Instruction pointer (`rip`)
    - CPU isn’t intelligent, and it executes whatever you tell it to
  - Stack pointer (`rsp`)
  - Base pointer (`rbp`)
- General-use registers
  - All are writable
  - Used for arguments for function calls on x86_64 systems

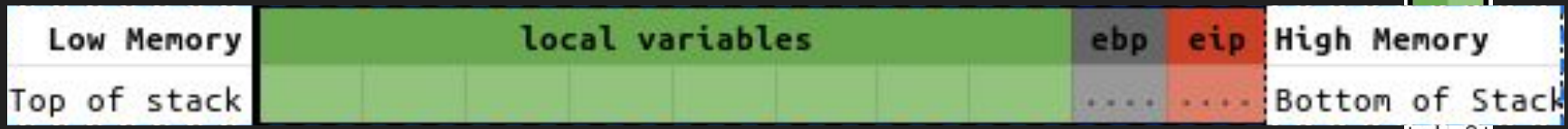
Memory

- Memory
  - Everything's in memory
  - Code
  - Variables
  - Arguments
- Buffers
  - A place to store data
  - (Usually temporary)
  - Useful for data manipulation



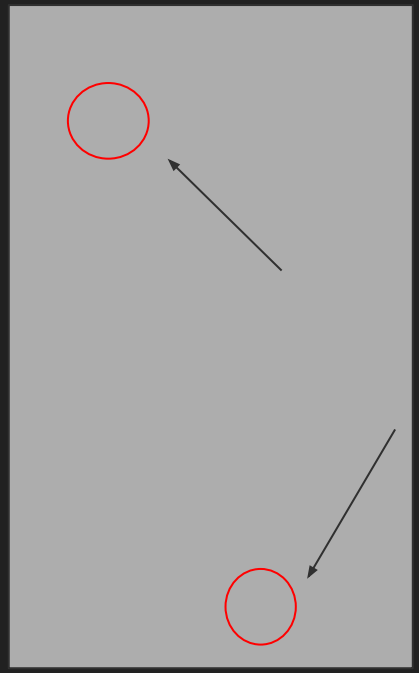
The Stack

- More volatile
  - Read/write
- Grows downwards
  - Higher value addresses “lower” on the stack
- Great place for temporary data
  - Buffers
  - Arguments
  - Local variables



Return Pointers

- When we call a function…
  - We start executing code outside of the original function
- Where do we go when we're done?
  - Hopefully back to the function that called it!
- How do we do that?
  - Store the address we wanna go back to
  - The stack!
- Before the function
  - Store our address
- After the function
  - Read our address



Lab Time (#1) — Fundamentals

- Set up the VM
  - Open VirtualBox
  - Configure Host-only Adapter for networking
  - Start up the VM
- Log in
  - Username: `root`
  - Password: `toor`
- Play around for a while
  - Look in `/root/lab1`
  - Try running the programs
  - Take a look at the source code
- Questions? Ask a friend or one of the Aarons!

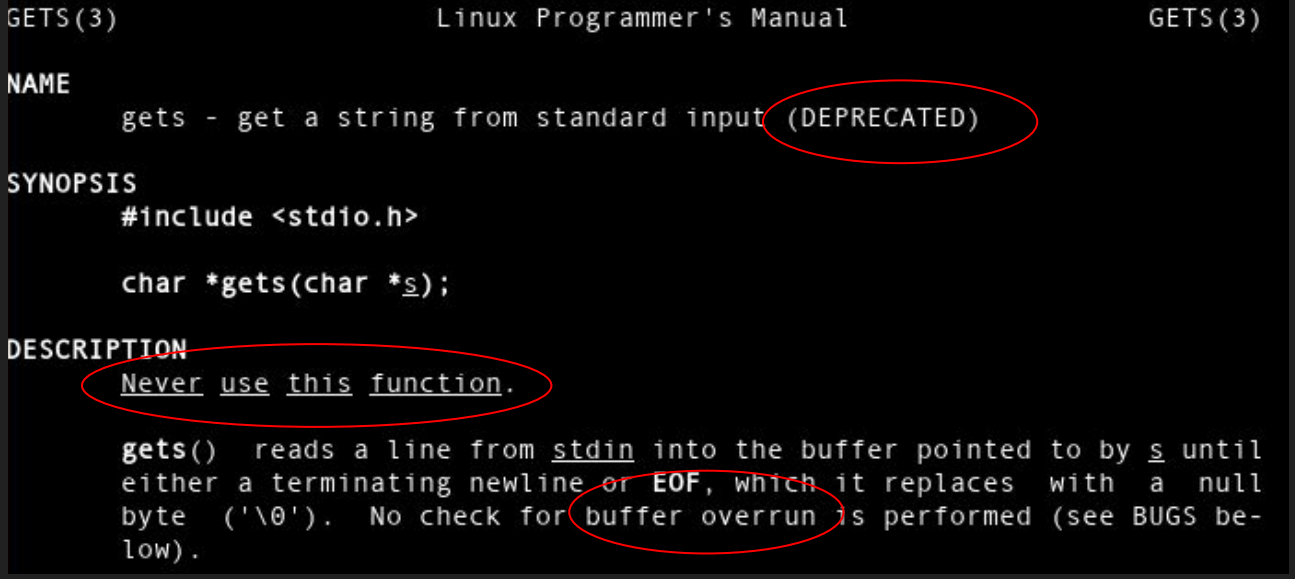
Buffer Overflow

Stack Frame

```
call function      # push %rip; jmp function
...
push %rbp
mov %rsp, %rbp
sub $VARS, %rsp
...
mov %rbp, %rsp
pop %rbp
ret                # pop %rip
```

The Attack

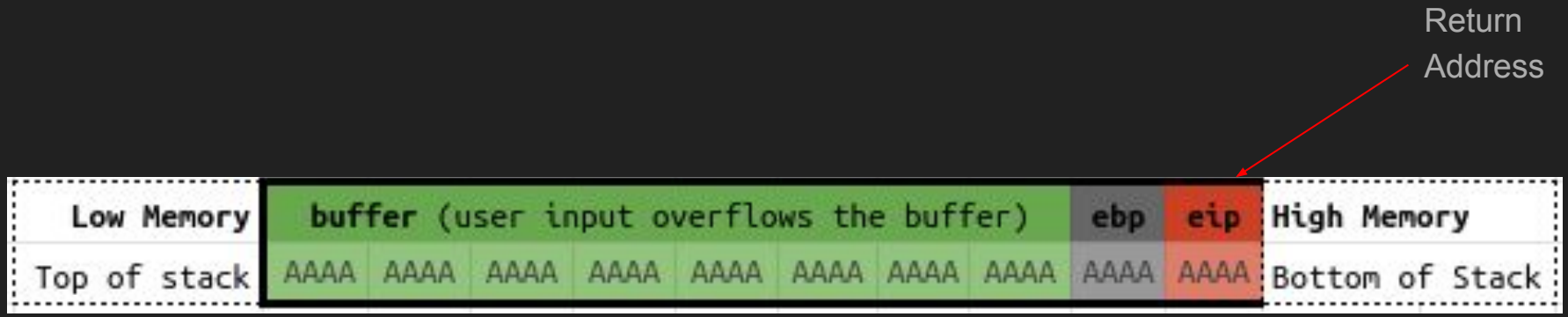
```
char array[20];
gets(array);
puts(array);
```



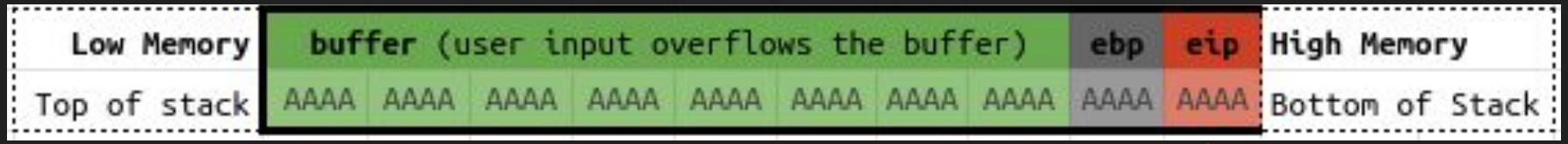
```
GETS(3)                    Linux Programmer's Manual                    GETS(3)

NAME
       gets - get a string from standard input (DEPRECATED)

SYNOPSIS
       #include <stdio.h>

       char *gets(char *s);

DESCRIPTION
       Never use this function.

       gets()  reads a line from stdin into the buffer pointed to by s until
       either a terminating newline or EOF, which it replaces  with  a  null
       byte  ('\0').  No check for buffer overrun is performed (see BUGS be-
       low).
```


```
sh-5.0$ ./vuln
AAAA
AAAA
```

```
sh-5.0$ ./vuln
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
```



Controlling the Return Pointer



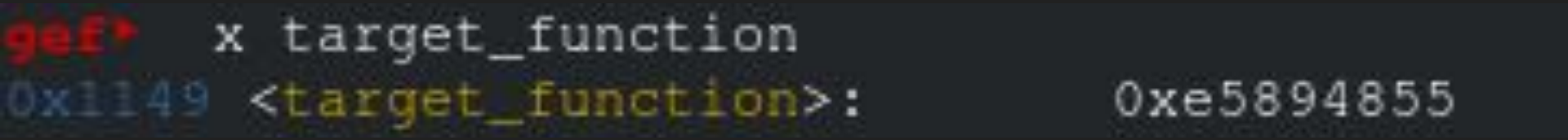
Tries to return to
0x41414141

Overwrite the return pointer with a valid address...

Which Address?

We want to return to a specific function

```
$ gdb ./vuln
(gdb) x target_function
```



What You’re Overwriting

- Not `rip`!
- The return pointer
  - Data on the stack
  - Sets `rip` at end of function
- Controlling the return pointer…
  - We decide where the function returns to
  - Instead of original function
  - Can be anywhere
    - Invalid address
    - Another function
    - The stack

Segmentation Faults

- When the program tries to access restricted memory, it will “segfault”
  - Could be memory that hasn’t been allocated
  - `0x4141414141414141`

Endianness

- Little Endian
  - A way to store an integer value in bytes (in memory)
  - Most significant to least significant bytes
  - **Example:**
    - 0x**1020304050607080** => **\x80\x70\x60\x50\x40\x30\x20\x10**
    - 0x00000000**44332211** => **\x11\x22\x33\x44**
    - 0x**4433221100000000** => **\x00\x00\x00\x00\x11\x22\x33\x44**

Lab Time (#2) — Buffer Overflow — ASLR off!

- Find the “padding” before the return pointer
  - Calculate from the source code or find experimentally
- Find the target function (the “win function”)
- Convert the address of the target function to little endian (“packing”)
  - 0x**1020304050607080** => `\x80\x70\x60\x50\x40\x30\x20\x10`
  - 0x00000000**44332211** => `\x11\x22\x33\x44`
  - 0x**4433221100000000** => `\x00\x00\x00\x00\x11\x22\x33\x44`
- Send the `padding + address` to the program’s stdin

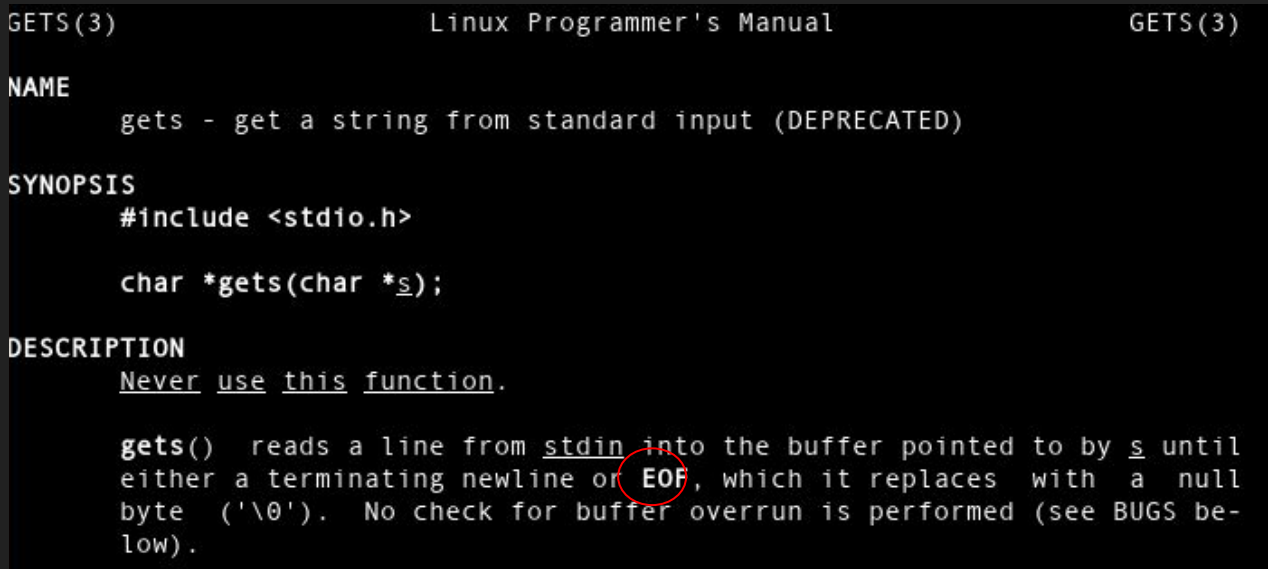
Shellcode

What is Shellcode?

- Machine code is stored as a series of bytes in text section
- We can write anything to a buffer on the stack
  - What if we put machine code in the buffer?
- We control where the function returns to
  - Stack addresses are valid — no segfault!
- Put machine code on the stack
  - We can set `rip` to beginning of code
  - CPU will execute our code!

How to Write Shellcode

- Write the code
- Compile or assemble it
- Extract the bytes from binary (encode with hex)
- No null bytes or newlines



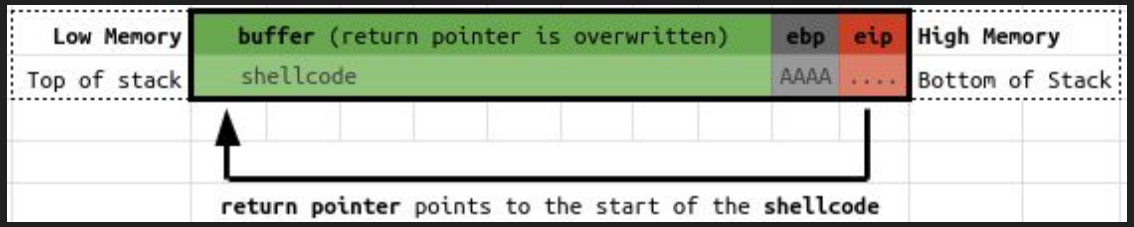
Okay, no shellcode writing

- Prior programming experience required
- Restrictions on shell code (e.g. null bytes)
- Too much debugging

We made some for you!

(`shellcode.txt`)

The Attack


Low Memory
buffer (return pointer is overwritten)
ebp
eip
High Memory
Top of stack
shellcode
AAAA
....
Bottom of Stack
return pointer points to the start of the shellcode

Lab Time (#3) — Shellcode — ASLR off!

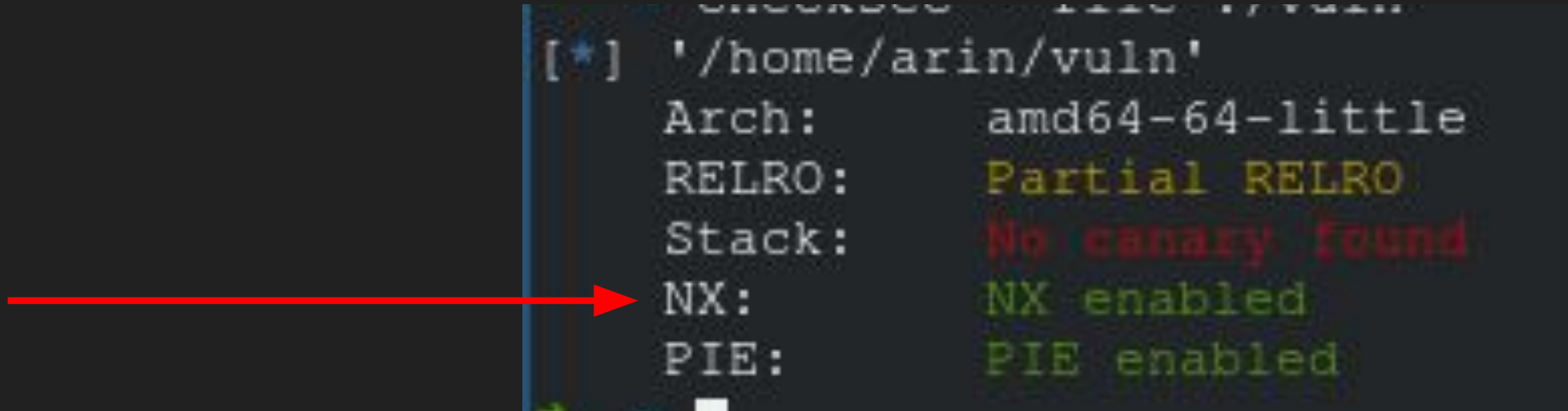
- Figure out where the shellcode is on the stack (hint: it's in the buffer!)
- Write a payload with shellcode and the return pointer

```
(python exploit.py; cat) | ./vuln
```

ret2libc

NX

- “Not executable”
- A bit that is set on the memory page
- You can’t execute code on the stack
- Our exploits don’t work anymore!



libc

- Static Linking
  - Golang
  - All your code is in the binary
- Dynamic Linking
  - C!
  - Some of the code is in a shared library
  - Smaller source files
  - Easier to roll out bug fixes
- libc is one of these shared libraries
  - The entire library is loaded in memory
  - Lots of functions
  - gets, printf, execv, system, etc...

The Attack

- With NX enabled…
  - We can’t just jump to the stack
  - We can only execute code that is meant to be executed
- Where can we find such code?
  - In the program code
  - In the dynamic libraries!
- Overwrite the return pointer...
  - With the location of a target function

Tools

- one_gadget
  - functions in libc that call /bin/sh
  - offset (i.e. “distance” from the base address of libc in memory)

Lab Time (#4) — ret2libc — ASLR off!

- Find the “win function” in libc to pop a shell
  - Find the path of libc using `vmmap` in gdb
  - `one_gadget` the libc binary to find the win function
- Craft a buffer overflow payload as before

NX and ASLR

Address Stack Layout Randomization (ASLR)

- You have to know where your shellcode is to be able to execute it
- Let’s randomize the stack and libraries!
- A “base” address is chosen at runtime and the library is placed there
    - Addresses in libraries are always at a fixed position relative to the base address

Memory Leak

- We need to know where our win function (in libc) is
- If can leak a libc address…
  - We can calculate offset from libc base
  - We can go to our function now!

pwntools

- What is it?
  - A CTF framework and exploit development library written in Python
  - Simplifies exploit development
- Why would we use it?
  - Calculating addresses on the fly (thanks ASLR!)
  - Simplifies exploit development (!)

Format String

```
char buf[20];
fgets(buf, 20, stdin);
printf(buf);
```

```
aaron :: ~ » ./asdf
%x %x %x %x %x
707677d0 2a48d0 20782520 9f8e9270 4af4c0
```

Code such as **printf**(foo); often indicates a bug, since foo may contain a % character. If foo comes from untrusted user input, it may contain **%n**, causing the **printf**() call to write to memory and creating a security hole.

What’s happening?

- Example function call:

```
printf(“Hello %s!\n”, name);
```

- printf blindly prints the format string you give it:

```
printf(“Hello %s!\n %x %x %x %x %x %x %x %x”, name);
```

- It’ll just print what it thinks are arguments
  - They’ll come from the stack!
- What happens when we control the format string?
  - We can control what “arguments” we leak

The Attack

- Protections
  - NX: you can’t execute shellcode on the stack anymore!
  - ASLR: you don’t know where your win function in libc is anymore!
- Approach
  - We need to find a win function is in libc
    - Use the `one_gadget` tool to find the offset of the function
  - We need to know where libc is at runtime (because of ASLR)
    - Obtain a leak
    - Subtract the offset to find libc base
    - Add the offset of the win function to the libc base
  - Simple buffer overflow!

Lab Time (#5) — NX and ASLR — ASLR on!

- Leak a pointer in libc using format string injection
  - Subtract the offset from libc, the offset is always the same!
  - Add the offset of the win function (`one_gadget`)
  - Use pwntools' `p64` function to pack the win function address (little endian)
- Exploit the buffer overflow vulnerability as before

Conclusion

Review

- Approach low level concepts through binex
  - Don’t just try to break something!
  - Try to stop and think
- Software isn’t perfect…
  - It can have flaws
- There’s a lot more out there
  - Look at the back of the workbook

Contact

- Aaron Esau
  - @arinerron
  - me@aaronesau.com
- Aaron Jobé
  - Doesn’t use Twitter
  - me@aaronjo.be

Come find us in the CTF room!

https://tuxedo.red/contact

BSidesPDX CTF

- BSidesPDX CTF: https://bsidespdxctf.party
- Do the CTF!