Introduction to **Binary Exploitation**

Presented by
Aaron (Esau|Jobé)

Prepared for
BSidesPDX 2019

About

Authors:

Aaron E. and Aaron J. are both seniors at Tigard High School who are interested in security (especially binary exploitation!) and compete with their school's CTF team, Red Tuxedo (https://tuxedo.red/).

Booklet Contents:

I. Preface
II. Lab 1: Fundamentals
III. Lab 2: Buffer Overflows
IV. Lab 3: Shellcode
V. Lab 4: NX and ret2libc
VI. Lab 5: NX and ASLR with ret2libc
VII. Post-Workshop

Preface

Objectives:

- Understand low level concepts including
  - stacks
  - registers
  - pointers
  - buffers
  - address randomization
  - memory page permissions
- Be able to exploit a buffer overflow vulnerability with or without ASLR by
  - redirecting code execution to shellcode
  - using a ret2libc attack
- Be able to identify basic vulnerabilities in C source code

Recommended Background:

- Linux and Python 3 experience is highly recommended
- C programming or Assembly (x86_64) experience is helpful but not required

Lab 1: Fundamentals

Working Directory: lab1/
ASLR: off

1. ASLR should be disabled for this challenge. We'll talk about what this means later. In the meantime, run the script: **./aslr-disable.sh**.

2. From the source code (**vuln.c**), you can see that the binary **./vuln** in **chal1/** is using the **gets** function and blocking until you give it input.

3. Open the binary in gdb with **gdb ./vuln**, type out **run**, press the enter key, then press Ctrl+C; **gets** will block until it receives input, and Ctrl+C will pause code execution in gdb.

4. While people are setting up their VMs, input things into **./vuln** and try to cause it to crash (or cause a segmentation fault).

   If you can cause the program to crash, in gdb, try using the gdb commands below to investigate the crash and figure out what is causing it and why:

- **run**: (re)start a program
- **continue**: continue executing code after a breakpoint
- **step**: execute a single instruction then pause again
- **break <address/function>**: set a breakpoint to pause execution when that point is reached
- **reg**: list all registers
- **vmmap**: list the process' virtual memory mappings
- **x/gx <address>**: display a 64-bit hex value at an address
- **x/s <address>**: display a string at an address (up until a null byte)

Lab 2: Buffer Overflow

Working Directory: lab1/
ASLR: off

1. Look at the source code for **chal1/** in **vuln.c**. How large is the buffer?

   On the stack, the stack frame is right after the buffer. The size of the base pointer (between the buffer and the return pointer on the stack) on 64-bit systems is 8 bytes.

   So, the offset of the return pointer is the size of the buffer + 8 bytes. If the goal is to overwrite up until the return pointer, how many bytes of padding are required?

   Note: For challenges that you may solve in the future where the source is not provided, the easiest way to determine the padding size is to simply provide a

series of bytes in like **AAAAAAAABBBBBBBBCCCCCCCCDDDDDDDDEEEEEEEE** etc, check at the value of **rip** after it segfaults, and determine the offset of the characters.

2. Open gdb on the binary with **gdb ./vuln**. Type out **run** then press Ctrl+C. What is the address of the win function (run **x win**)?

The **win** function is our goal. How do we get there? We are able to write up to the return pointer with the "payload padding". Then, we just write the **win** function in little endian.

What is the address of the **win** function in little endian?

---

3. Try creating a python script to write the padding out then write the new return pointer.

print("A"*padding_size + return_pointer)

...where **padding_size** is the distance from the start of the buffer to the return pointer (we figured out in step 1) and **return_pointer** is a string to containing 8-byte return pointer (address of **win**) in little endian.

Let's execute it: **python exploit.py | ./vuln**

At this point, you should see the "congratulations" messages because it executed the **win** function. If not, here are a few things to double check:

- Are you sure that you converted the address to little endian correctly?
- Are you sure that the padding length was correct? Try using the address **BBBBBBBB** instead and write the payload to a file, **payload.txt**. Open up gdb and execute **run <payload.txt**. It should cause a segmentation fault. Does **x/gx $rbp** output **0x4242424242424242** (hex of **BBBBBBBB**)? If not, you may not have the right padding length.
- Are there null bytes or newline characters in the payload? Remember, **gets** terminates at the first null byte (**\x00**), first newline character (**\x0a**), or EOF.

Lab 3: Shellcode

Working Directory: lab1/
ASLR: off

1. Read the source code of this challenge. Uh oh, there's no **win** function now. What will we do?

   Now, instead of redirecting code execution to the **win** function, we want to execute our own code. To execute our own code, we need to:
   - have our code (the shellcode) somewhere in memory
   - know the address of the shellcode (we need to know what to overwrite the return pointer with)

   Think… Where in memory are we able to put our own arbitrary values? The stack!

   We can put the shellcode in the buffer before or after the return pointer and overwrite the return pointer with a pointer back to the shellcode in the buffer.

   To do this, we need to know where the shellcode is.

Open up this lab's challenge in gdb. Set a breakpoint on **puts** with **break puts**. Run the program with **run**. When it asks for input, give it **AAAAAAAA**.

Now, let's examine the stack and look for the start address of the buffer that we control in memory.

x/20gx $rsp – show **160 bytes** (20 * 64 bits) starting at **rsp**.

Look for the **0x4141414141414141** that we put in memory. On the left hand side, it lists the addresses of the values. Where does the buffer start?

---

2. Now we know that when we input shellcode, that address is the start of the shellcode. Our payload will look like:

   shellcode + **padding** + **return pointer**

   Remember how we calculated the length of padding in lab 2 (buffer size + 8 bytes)? The return pointer is at a fixed position relative to the buffer, so we need

to adjust the padding length to work with the shellcode.

To calculate the new length of the padding:

padding = original padding - shellcode length

Simple!

3. Choose the shellcode. There's a file called **shellcode.txt** in your home directory in the VM that contains some pre-made 64-bit shellcode.

   The exploit should look something like:

   print(shellcode + "A"*(padding_size - len(shellcode)) + return_pointer)

   Unfortunately, now, it isn't as simple as executing the exploit script and piping it into the program; after the exploit script writes the payload to **vuln**'s **stdin**, the python script will terminate. Then, the pipe will close and **vuln** will terminate.

   Remember that our shellcode will give us a shell with **/bin/sh**.

After executing the exploit script, we need to continue taking input from **stdin** and writing to **stdout** to be able to use the shell. There's a cool command for that: **cat**.

We can do it two ways:

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

Or, if you want to use the exploit in gdb:

```
python exploit.py > exploit.txt
...
(gdb) run <exploit.txt
```

Nice! That should give us a shell.

Lab 4: NX and ret2libc

Working Directory: lab1/
ASLR: off

1. For the first challenge file, run **checksec --file ./vuln**. As you can see from the output, NX is enabled on the stack (i.e. shellcode on the stack cannot be executed).

   If we can't execute code on the stack, where can we execute code from?

   The **text** (code) or library segments of the binary. Those memory pages must be marked as executable, of course, otherwise the program cannot execute (as there would be no executable sections!).

   We just need to find existing code in the **text** section or the libraries.

2. In the libc library, there are multiple functions that you can call to pop a shell (**/bin/sh**). Let's use one of those.

There is a tool called **one_gadget** that takes in the path to libc and outputs the offsets from the "libc base" (the first address of libc in memory).

First, we need the path to the libc ELF file and the position of libc in memory.

Open gdb, **run** the program, then press Ctrl+C to pause the program execution. Now run **vmmap** to get the process memory map.

What is the lowest address (the first listing in the process mapping) of libc (the "libc base")?

_____

What is the path to libc, listed in the process mapping?

_____

3. Great. Now let's get the offset of the functions that pop a shell in libc. Run **one_gadget /path/to/libc** where **/path/to/libc** is the path you just obtained.

Now you have:

- the address in memory of libc (“libc base”)
- the offset in libc of a function to pop a shell

To calculate the offset in memory of the function to pop a shell, simply add the two together.

4. That address is the “win function”, like what you had in lab 2.

   Simply write an exploit to overwrite the return pointer with the address of the win function in libc.

   Remember to to execute the exploit script like this:

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

Lab 5: NX and ret2libc with ASLR

Working Directory: lab1/
ASLR: on ← **./aslr-enable.sh**

1. In lab 4, the libc base address was fixed. Now that you've enabled ASLR, libc (and the stack and other libraries) are placed at a randomized offset.

   To be able to use the **win** function, now you must leak the offset of libc in memory at runtime.

   You must be able to read **./vuln**'s stdout and write to **./vuln**'s stdin to be able to obtain a leak and exploit the binary. A library called **pwntools** makes this process simple.

   Consider the following Python 3 code:

   ```
   from pwn import *

   p = process('./vuln')

   p.sendline('%x') # writes '%x\n' to stdin
   ```

```
output = p.clean() # returns the contents of stdout up to this point

integer_val = u64('\x00\x00\x00\x00\x00\x00\x00\x01')
# unpacks 64-bit little endian
string_val = p64(0x1) # packs 64-bit little endian

p.interactive() # connect the process' stdin to stdin and the process' stdout to stdout
```

Those functions should be fairly self-explanatory. Try messing around and printing the output of some of those functions to see what they do.

If you would like to read more, the documentation is accessible at http://docs.pwntools.com/en/stable/

2. There are many ways that you could obtain a memory leak, but for the purposes of this lab, we will use another vulnerability type: a format string injection vulnerability.

   In C, the **printf** function takes format specifiers (like Python's **%x**, **%s**, **%d**, etc). When calling **printf** like this where the **user_input** variable is, of course, your user input, the vulnerability exists:

printf(user_input, arg1, arg2, arg3, ...);

When more format specifiers are used in the user input than arguments are provided in the function, **printf** will pop values off of the stack that were not intended to be provided to the function.

As **printf** calls internal functions to parse the format specifiers, there is a return pointer to somewhere in the **printf** function (in libc) on the stack.

Functions in libraries are always at a fixed position relative to the base of the library, so if you can leak a pointer inside of the library, you can calculate the address of the base of the library.

This challenge is vulnerable to format string injection.

3. Run **gdb ./vuln** and start the program with **run**. Input **%016x %016x %016x %016x** (**%016x** is the 64-bit version of **%x**), press enter, then press Ctrl+C.

   The program should output some hex values. Run **vmmap** in gdb to output the process memory mapping and look

for which of the hex values appears to be a pointer inside of the range of addresses that libc uses.

If you subtract the libc base address from the leaked pointer, you will get the offset from libc base of the pointer. This value (the offset) is always constant for that libc binary.

What value do you get for the offset of the pointer from the libc base address?

The offset of the hex value of the pointer (i.e. the number of **%016x** required to get the hex value of the pointer) is always the same.

4. Write a Python script with pwntools to:
    a. Send enough **%016x**'s to leak a libc pointer (**p.sendline(payload)**)

    b. Read the response from the binary (**p.clean()**)

    c. Extract the hex value from the response

   d. Hex decode the pointer (**int(value, 16)**)

   e. Subtact the offset you calculated before from the pointer (to calculate the libc base)

   f. Add the offset of the win function that you got from **one_gadget**

   g. “Pack” the result (i.e. convert the integer representing the address of the win function to a little endian string address) using **p64(value)**

   h. Send a buffer overflow payload to fill the buffer, write past the base pointer, and overwrite the return pointer with the packed value of the win function.

5. You should get a shell. Congratulations, you’ve bypassed both ASLR and NX!

   At this point, you are free to move on to the next section of this booklet and work on your own.

   If you see someone who needs help and you have gotten this far, please feel free to help them!

Post-Workshop

As you may have noticed, each of the labs had 2-3 challenges and we only had you solve the first challenge in each. If you want to get more practice, we would encourage you to take a shot at solving the other challenges (if you haven't already).

Remember, **gdb** is your friend.

Here are our tips for you at and after this conference:

- Go play the CTF. Really. We've coordinated our workshop with BSidesPDX CTF so that you are able to solve at least the first binary exploitation challenge this year.

  BSidesPDX CTF offers a great opportunity to test out your binary exploitation skills and continue learning during the conference.

  The CTF is available at https://bsidespdxctf.party.

- If you have any further questions, ask Google, DuckDuckGo, or us. We'll be hanging out in the CTF

room a lot of the time during the conference and would love to talk.

- Keep practicing! There is a lot more to learn and there are plenty of CTFs to help you get good. Check out picoCTF 2019, it is known to be very beginner friendly.

There is a lot more to learn. Here are some search terms to Google:

- **ELF Reverse Engineering:** We taught you how to exploit binaries but not how to reverse engineer them to find vulnerabilities. Often in CTFs, the source code is not provided, and you must reverse engineer it yourself. You can really help yourself out by learning how to use tools like Ghidra and radare2.

- **NOP Sleds:** If you cannot obtain a leak when ASLR is enabled, you can use a "NOP sled" (a series of nop instructions, **\x90**, followed by the shellcode) to increase your chances of "guessing" an address that executes your shellcode successfully.

- **Stack Canaries:** A protection that attempts to prevent buffer overflow attacks by placing a random value

("canary") before the base pointer and return pointer that is checked before returning from a function.

- **Position-Independent Executable:** This protection, like ASLR, randomizes the position of the code in memory to make it difficult to return to the program (similar to ret2libc attacks, but to a win function in the program).

- **Return-Oriented Programming (ROP):** Although we did indirectly talk about this, we called it ret2libc to simplify things. Really, the technique is called Return-Oriented Programming, and there is a lot more to learn about it.

- **Format String Injection:** We talked about leaking values from the stack using format string injection, but it is also possible to directly obtain remote code execution using format string injection attacks.

- **Global Offset Table (GOT) Hijacking and RELRO:** After obtaining an arbitrary write in memory, it is common to hijack entries in the GOT. There is also a protection to prevent GOT hijacking called RELRO.

- **Execution Hooks:** These are hooks that are executed when certain events occur (e.g. a chunk is malloc'd). These can also be used to obtain remote code execution given an arbitrary write.

- **Heap Exploitation:** Once you have learned how to do basic pwn, try learning about heap corruption-related attacks. They're more complicated but are very relevant.

- **Kernel Exploitation:** This is also more complicated, but is also very interesting and very relevant.

And finally, below is our contact information. Feel free to reach out if you have questions or would like to let us know how the workshop was:

- Aaron Esau (@arinerron): me@arinerron.com
- Aaron Jobé (@dirtyc0wsay): me@aaronjo.be

Slides are available at https://tuxedo.red/bsidespdx/.