Computer Systems Security

CSE 466
Fall 2018
Yan Shoshitaishvili

http://pwn.college
http://groups.google.com/group/cse-466
https://goo.gl/yorGNc

Homework 6 Status

Statistics

Total in course: 73 graded students (+1 auditing)

HW6 thus far:

62 attempted
24 A+
14 have more than the example shellcode

Difficulties

- Redirecting control flow.
- Garbage bytes in shellcode.

Master Hackers

Full solves (thus far):

dasher
plutoisaplanet
presitator_will
pwnish

CSE 466 Week 7

Return Oriented Programming

The story so far...

Our offensive capabilities:

- With HW3/4, we explored reverse-engineering of software to understand how it works.
- With HW4/5, we explored redirecting the control flow of applications.
- With HW6, we explored the injection of code (to which control flow will be redirected).

This requires:

- access to the binary code
- normally, knowledge of where your hijack target (shellcode) is in memory
- ability to execute your shellcode

The story so far...

Our offensive capabilities:

- With HW3/4, we explored reverse-engineering of software to understand how it works.
- With HW4/5, we explored redirecting the control flow of applications.
- With HW6, we explored the injection of code (to which control flow will be redirected).

This requires:

- access to the binary code
- knowledge of where various code is in memory
- **ability to execute your shellcode**

The "No-eXecute" bit

Modern architectures support memory permissions:

- **PROT_READ** allows the process to read memory
- **PROT_WRITE** allows the process to write memory
- **PROT_EXEC** allows the process to execute memory

Intuition: *normally*, all code is located in .text segments of the loaded ELF files. There is no need to execute code located on the stack or in the heap.

By default in modern systems, the stack and the heap are *not* executable.

YOUR SHELLCODE NEEDS TO EXECUTE.

Game over?

Workarounds?

Let's assume we *can't* inject executable code. What now?

Last class, we discussed fooling the program into calling mprotect to make our shellcode executable.

Let's take a deeper look...

Demo:

- **simple mprotect**

What if the program isn't so nice?

We can't rely on the program purposefully fixing up our shellcode and calling it. Can we do some other shenanigans?

What if the program isn't so nice?

We can't rely on the program purposefully fixing up our shellcode and calling it. Can we do some other shenanigans?

Enter: *Return Oriented Programming*

You already touched on this in HW4 and HW5!

HW3: dos_*.c

```
#include <assert.h>

int main()
{

     char input[128];
     read(0, input, 0x1000);
}

void win(int win_token)
{
     assert(win_token == 0x1337);
     puts("You win! Here is your flag:");
     register int flag_fd = open("/flag", 0);
     sendfile(1, flag_fd, 0, 1024);
}
```

What's the solution there?

Demo reminder!

What is the implication of this?

You are not restricted to jumping to the beginning of functions!

Let's apply this!

Demo time!

... but what if the program is even less nice?

Demo time!

You are not restricted to jumping to the beginning of functions, but...

... are you restricted to jumping to the beginning of *instructions*?

Return Oriented Programming

Demo time!

Return Oriented Programming

Conceptually, shellcode chains together instructions that you choose to carry out your goal.

With ROP, you can do the same with *ROP Gadgets*.

ROP Gadget: a small snippet of code that performs something simple and performs a *controllable control flow transfer*.

ROP gadgets by induction

Step 0: overflow the stack

Step n: by controlling the return address, you trigger the following *gadget*:

```
0x004005f3: pop rdi ; ret
```

Step n+1: when the gadget returns, *it returns to an address you control* (i.e., the next gadget)

By chaining these gadgets, you can perform arbitrary actions!

What's the point?

ROP allows you to achieve arbitrary control over a program *without executing any shellcode*, but just by reusing existing code in the binary.

This is a very relevant modern technique (mitigations remain spotty despite extensive research).

Complications

As with everything, there are complications:

- limits to your control of the stack
- limits to the bytes you can input (on 64-bit systems, if you cannot input null bytes, you are most likely limited to one gadget)
    - why is this?

However, often, the ability to control a single return address leads to immediate "victory".

The story so far...

Our offensive capabilities:

- With HW3/4, we explored reverse-engineering of software to understand how it works.
- With HW4/5, we explored redirecting the control flow of applications.
- With HW6, we explored the injection of code (to which control flow will be redirected).

This requires:

- access to the binary code
- **knowledge of where various code is in memory**
- ability to execute your shellcode

Address Space Layout Randomization

On modern systems, the location of *anything* is unknown to the user.

Can we cope with this?

1. Like in HW6 level 3, or some levels in HW4 and HW5, you generally first have to disclose memory.
2. On the other hand, if you are attacking a *forking network service* or are on Android (where every process forks off of a common parent), you can brute-force memory locations.

Demo time!

Address Space Layout Randomization

In general, ASLR is a **very good** mitigation.

Usually, NX just requires a workaround: ROP. Stack smashing vulnerabilities remain quite dangerous.

ASLR, on the other hand, normally necessitates a *second vulnerability*: an information disclosure.

There is a similar concept here to a stack canary. What is the concept?

The story so far...

Our offensive capabilities:

- With HW3/4, we explored reverse-engineering of software to understand how it works.
- With HW4/5, we explored redirecting the control flow of applications.
- With HW6, we explored the injection of code (to which control flow will be redirected).

This requires:

- **access to the binary code**
- knowledge of where various code is in memory
- ability to execute your shellcode

Hacking Blind?

Can you exploit a program that you *do not have*? In certain situations, yes!

The standard blind attack requires a forking service.

- Break ASLR and the canary byte-by-byte. Now we can redirect memory semi-controllably.
- Redirect memory until we have a *survival signal* (i.e., an address that doesn't crash).
- Use the survival signal to find non-crashing ROP gadgets.
- Find functionality to produce output.
- Leak the program.
- Hack it.

Hacking Blind!

Proposed by Andrea Bittau at the 2014 IEEE Symposium on Security & Privacy.

http://www.scs.stanford.edu/brop/bittau-brop.pdf

Homework 6

Keep hacking away at HW6.

I will also upload a few challenges that will feature in HW7, if you want to get an early start.