Previously

Code injection

Mitigations

Counter-mitigation strategies

~~Counter-counter-mitigation mitigations~~

Code injection

1. Inject code

- writable buffers
- user-driver memory allocation

2. Hijack control flow

- targets: return addresses, function pointers, conditions...
- approaches: buffers, integers, format strings, application logic...

Mitigations

How can we prevent/reduce stack smashing?

- stack canaries: `-fstack-protector`
- non-executable stacks (we needed `-z execstack` to demo!)
- no stack access

A *stack canary*, like a canary in a coal mine (fun picture here), is something that can be checked to see if conditions are too dangerous to continue normal operations. In the case of a canary, it would faint from carbon dioxide before humans would, sending a signal that the mine wasn't safe. In the case of a stack, _____ can be written to the stack in between functions' allocations. Code is inserted to check this "canary" value _____ _____ to ensure that _____.

Counter-mitigation strategies

nop sleds

Heap spraying

Disguised shellcode

Counter-counter-mitigation mitigations

Modern MMUs

W^X policy

Your computer's memory management unit (MMU) is the thing that translates virtual addresses to physical addresses. Along the way, there is an opportunity to check _____ _____. Specific mappings can be marked as read-only, or as inaccessible to user code, and on modern machines, as _____. Marking memory as *non-executable* is something that wasn't possible on 32-bit x86 computers, but *is* possible on _____. This allows us to prevent the execution of bytes in specific regions like _____.

However, it's more general than that! In general, we would like to have memory be writable XOR executable. If it's possible for an attacker to write in to the memory (whether directly, like providing a buffer of shellcode, or indirectly, by tricking a program into writing some data in a particular place), it should *not* be possible to execute that code. There are some exceptions (a JIT engine, by definition, needs to be able to write out executable code), but normally we would like to enforce a W^X policy that will completely prevent some of the attacks described in the previous slides.

So . . .

Stages of code injection

1. Inject code

2. Hijack control flow

But step 1 is getting harder!

What if...

~~0. Inject code~~

1. Hijack control flow

What code do we execute?

If the attacker can't inject any code, the only code that can be run is _____ _____. But what can an attacker do with that?

What is a program?

Where does a program come from?

- programmer intent
- source code
- object code
- executable binary + linked libraries

Final result: bytes

At the end of this long process of compilation and execution, we end up with _____ _____. Those bytes were generated via a long and complex process that started with the intentions of a programmer, but now they are just _____ that represent _____ for the computer.

So what if we use those bytes to reflect the intent of a _____?

What is a program?

How does a program work?

- CPU executes instructions linearly (mostly)
- can *branch* to other instructions
- can *call* and *return*

How can an attacker control return?

We've seen how an attacker can control the return from a function by modifying the return address on the stack. This is helpful when redirecting control flow to code that an attacker has injected, but what can they do when they can't put executable code into the process' memory?

Return to libc

Uses existing code from `libc`

e.g., return to `system()`

Especially easy on 32b x86

If you can't add code to memory, you'll just have to use what's already there! This kind of "living off the land" is possible because there is already quite a lot of code lying around in memory. For example, there is *lots* of code in the standard C library, which gets loaded into just about every process running on your system.

One common thing we'd like to be able to do when we attack a program is... anything! We'd like a general-purpose tool for letting us execute arbitrary commands once we've broken into a process, and `libc` provides us with just such a tool: the `system(2)` system call. This will allow us to execute any program we like, and if that program is a shell program, we can execute *more* arbitrary actions.

ROP

*Return-oriented programming**

Generalization of return-to-libc attack

Relies on existing "gadgets" (instruction + `ret`)

Can be automated (e.g., ROPC, Ropper)

---

* See, e.g., Roemer et al, "Return-Oriented Programming: Systems, Languages, and Applications", ACM TISSEC 15(1), 2012. DOI: https://doi.org/10.1145/2133375.2133377

But we can go even further than this!

Instead of just trying to "return" to function in `libc` that do interesting things like run other programs, we can _____ using little "gadgets" that are already lying around in memory.

What is a gadget?

Suppose the attacker would like to write a little program that pops a few values off the stack and then calls a function. They can't inject this malicious code themselves, but they can probably find quite a few instances of functions that *end* in interesting instructions like:

```
pop %rbp
ret
```

If you find enough of these gadgets, you can construct a whole program by pushing their return addresses on the stack, causing them to be executed _____.

Now, building programs from whatever instrutions you have lying around is a very challenging compilation problem, but people have built tools that use heuristics to automate it. For fun, try out the tutorials at https://ropemporium.com; we'll also see some ROP in our third lab.

ASLR

Address Space Layout Randomization

Not super-helpful on 32b platforms

Increases "work factor"

But maybe not by as much as you think!*

* "ASLR on the Line: Practical Cache Attacks on the MMU", Gras, Razavi, Bosmen, Box an Giuffrida, *Proceedings of the 2017 Networked and Distributed Systems Security Symposium*, 2017. DOI: https://dx.doi.org/10.14722/ndss.2017.23271.

Defenders can make the attacker's life harder by ensuring that `libc` (and other code) isn't loaded at the same location every time.

On a 32b machine, however, we might only have 16b or even 8b available for randomization. A lack of randomness *seems* bad in a defensive technique called "randomization", but why? What would more randomness give us?

ASLR _____. Unlike other security techniques, it won't always say "no" to an attack. What it will do is make an attacker have to do _____ _____. For example, on a 32b system, an attacker might have to _____ _____ in order to succeed.

Practical attacks exist that use low-level properties of things like memory management units (MMUs) to break ASLR, even from JavaScript code!

Code reuse attacks

~~0. Inject code~~

1. Hijack control flow

How do we stop the hijacking?

Stopping hijacking

Stack protection

Non-executable memory
Stack canaries (`-fstack-protector`)

CFI: control flow integrity

Static analysis, dynamic enforcement

Full *memory safety* (next time)