Last time

Code injection

1. Inject code (e.g., copying payload into buffers)
2. Hijack control flow (e.g., stack smashing)

Today

Mitigations

Counter-mitigation attacks

Counter-counter-mitigation mitigations

Mitigations

How can we prevent/reduce stack smashing?

- stack canaries: `-fstack-protector`
- non-executable stacks (we needed `-z execstack` to demo!)
- `W^X`: memory regions writable **or** executable (limitations?)
- ASLR: address space layout randomization (more later)

... and more to follow

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 _____.

Marking memory as *non-executable* is something that wasn't possible on 32-bit x86 computers, but *is* possible on _____. This functionality can be used to prevent executable stacks (always a good idea!) and/or a full `W^X` policy.

The attacker strikes back

Guessing precise addresses is hard

`nop` sleds, relative addressing

Shellcode authors avoid zeroes (why?)

Is shellcode easy to spot? See: English shellcode*

* "English Shellcode", Mason, Small, Monrose and MacManus, in *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, 2009. DOI: 10.1145/1653662.1653725

For this demo to work, I had to embed the stack address that would store the program counter. How did I do this? By _____ when I ran the program! That is not a very reproducible solution. In reality, it may not be possible to guess where a piece of data will land in a program's memory.

To deal with this difficulty, shellcode can include "nop sleds", which are long chains of `nop` instructions with shellcode at the end. If the program counter lands anywhere within the `nop` sled, it will "slide" all the way to the end and then execute the payload that's found there.

These kinds of payloads are often delivered via code that expects to read strings from somewhere (a file, the network, the user, etc.). When the target code receives a string and passes to around to functions, etc., it's very likely to need to run functions like `strlen` to figure out how much data to pass, etc. So, a shellcode author needs to avoid zeros in their strings: if not, `strlen` will think that _____ and then the code will _____ _____!

You might think that shellcode would be easy to spot, but you can hide all kinds of things inside innocuous-looking content. Remember how, when we examined our malicious file with tools like `hexyl` or `xxd`, many of the characters in the shellcode were displayed like ordinary ASCII characters? A lot of instructions' opcodes _____!

Higher-level languages?

One mitigation: no stack access

Alternative technique: *heap spraying*

- Create lots of shellcode strings (how much? try me!)
- Even further: *Heap Feng Shui**
- Just need *one* control-flow hack to trigger

---

* Alexander Sotirov, "Heap Feng Shui in JavaScript", Black Hat Europe, 2007.

One way to stop stack smashing is to avoid letting user code _____: if they can't write to _____, they can't *overwrite* it. However, that's not the only memory of interest. In fact, with non-executable stacks, the stack isn't really the most interesting any more!

Even very high-level languages running under bytecode interpreters will allow user code to create strings on the heap. These strings can contain things like `nop` sleds that lead to shellcode... lots of strings. How much heap data can you create? Check `window.performance.memory.jsHeapSizeLimit` in your browser, for one. It's no big deal to create hundreds of MiB of `nop` sleds all around a browser's memory, just waiting to be exploited by a control-flow hijack.

It's important to note that heap spraying (in all of its flavours) doesn't actually *execute* an attack: you still need to _____. We'll talk about some more ways this can be done in a few minutes.

Stages of code injection

1. Inject code

2. Hijack control flow

Code injection

Writable buffers

- any executable memory region

User-driven memory allocation

- user is *supposed* to be able to request allocation
- e.g., untrusted JavaScript allocates strings

In general, code injection can occur anywhere that code can be executed. That's typically not the stack anymore, and we'll soon see that the executable places an attacker can write to are getting scarcer over time.

However, we can't stop the attacker from allocating *any* memory: allocating memory is a pretty important and legitimate function of every programming language environment!

Control-flow hijacking

Targets:

return addresses, function pointers (inc. vtables), conditions...

Approaches:

buffer overflows, integer under/over-flows, format string vulnerabilities, application-level errors...

If code injection is the first step of a software attack, the second step is to make the injected code actually _____. This is done by subverting the regular control of the victim program. Anything that can be used for legitimate control flow can also be subverted for _____ control flow.

Targets:

_____ As we saw last time, call-and-return is a critical form of control flow for most programs, and it hinges on a detail of stack layout. If we can overwrite return addresses on the stack, we can cause all sorts of mischief (even if we can no longer do the classic stack smashing attack due to the default _____).

_____ There are lots of reasons to use function pointers in real code. One of the most prominent is in *vtables,* which support _____ in object-oriented systems (whether or not the languages themselves are object-oriented!).

_____ Sometimes all an attacker wants to do is to make your program decide one thing incorrectly. Should I let this user access that thing? Should I let the player into the game without a license?

Approaches:

_____ We saw these last time!

_____ We'll look at these on the next slide.

_____ We'll talk about these in just a few minutes.

_____ We'll talk a *lot* about these when we get to Web security (SQLi, XSS, CSRF...)

Integer overflow

Q: What is an integer? How about on a computer?

See demo code

Lesson: the details matter!

- don't assume that integers behave like, well, integers
- don't trust user input
- use safe integer arithmetic (US-CERT, Microsoft)

An integer is a whole number that can be positive, negative or zero. What is the maximum value of an integer? _____.

On a computer, however, an integer *in a register* is not exactly the same as an integer in mathematical terms. They are *almost* identical, but the small differences can matter a lot.

Integer overflow... still???

Over 3,000 reported CVEs, including dozens in 2024!

- Firefox: CVE-2024-2608
- LLaMA: CVE-2024-21836
- TP-Link router: CVE-2024-25139
- Windows Defender: CVE-2024-21420
- Probably Cellebrite (older)

Integer overflow is *still* very much a going concern!

Cellebrite is a system for digital forensics relied on by police services around the world, and apparently their own security practices were... not good. Sadly, this is all too common in the security world: people not practicing what they preach. We'll talk more about Cellebrite later in the course, but for now you may enjoy the following (genuinely amusing) read:

https://cyberlaw.stanford.edu/blog/2021/05/i-have-lot-say-about-signal's-cellebrite-hack

Format string vulnerabilities

See demo code

Lesson: the details matter!

- don't trust user input
  - put user strings in *values*, sure
  - do **not** put user strings in *format*
- also important for higher-level languages (e.g., Ruby)

Notes about code injection

Modern MMUs and DEP

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 _____. 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.

Stages of code injection

1. Inject code

2. Hijack control flow

But step 1 is getting harder!

What if...

Policies such as `W^X` make it much tougher to inject attacker-controlled code into memory that can actually be executed. However, that doesn't mean that attackers just gave up! Instead, they did what attackers do: they thought creatively, out of the box, not limited by the constraints that defenders impose on them.

What if...

~~0. Inject code~~

1. Hijack control flow

What code do we execute?

Is it possible to attack running software *without* injecting code? If we could still hijack the control flow of a program (which seems to often be the case!) and put non-executable data in memory (e.g., on the stack), how could we still have a viable attack?

What code would we even excute?

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

For fun, try out the tutorials at https://ropemporium.com !

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)