Last time

Code injection

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

Mitigations

Mitigations

How can we prevent/reduce stack smashing?

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

... and more to follow

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

Today

Mitigation details

Counter-mitigation attacks

Counter-counter-mitigation mitigations

Higher-level languages?

One mitigation: no stack access

Alternative technique: *heap spraying*

- Create lots of shellcode strings
- Just need *one* control-flow hack to trigger

Stages of code injection

1. Inject code

2. Hijack control flow

Code injection

Writable buffers

- any memory region: heap, stack or BSS

User-driven memory allocation

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

Control-flow hijacking

Targets

Buffer overflow

- as demonstrated last class!

Integer under/over-flow

Format string vulnerabilities

Return addresses (last class), function pointers, vtables, conditions...

Integer overflow

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)

Integer overflow... still???

- OpenSSL: https://nvd.nist.gov/vuln/detail/CVE-2021-23840
- Linux: https://nvd.nist.gov/vuln/detail/CVE-2021-3490
- Windows: https://www.fortinet.com/blog/threat-research/microsoft-kernel-integer-overflow-vulnerability.html
- probably: https://arstechnica.com/information-technology/2021/04/in-epic-hack-signal-developer-turns-the-tables-on-forensics-firm-cellebrite

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

Another great read about this hack: 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)

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!)