Module:
Return Oriented Programming

Complications

Yan Shoshitaishvili
Arizona State University

Issue 1: Limited Control

Sometimes, your control of the stack is limited.

1. Limited overflow size.
2. Inability to input NULL bytes. Often, we can still trigger one gadget!

Can we win with one gadget?

The Magic Gadget

Yes we can!

Consider `system()`...

- it has to set up a call to `execve("/bin/sh", {"/bin/sh", "-c", command}, env);`
- what if we jump partway through?

This actually works! If you get lucky with register values and stack setup, you can often trigger `/bin/sh` by jumping partway into `system()`. This location is called the *magic gadget*.

More useful for you: trigger `execve(some_garbage);` and create a `some_garbage` file that reads the flag.

Issue 2: Address Space Layout Randomization

Our old friend...

Workaround 1: Remember babymem: partial return pointer overwrite. Often, you can restart the whole program by jumping back to main or some other high-level function.

Why is this useful? Think about the REPEAT backdoor in babymem: having another shot at the exploit, but with potentially *more data* is invaluable!

Psst! If you're returning from main (into libc), you might be able to hit the magic gadget!

Workaround 2: Situation information disclosure (i.e., null-termination issues as in babymem).

Issue 3: Stack Canaries

Need to either leak or bypass in some other way, babymem-style.

Issue 4: Exotic Academic Solutions

ROP is part of a long-term cat and mouse game between attackers and defenders.

Anti-ROP approaches:

- removing ROP gadgets (too onerous):
  G-Free: Defeating Return-Oriented Programming through Gadget-less Binaries
- detecting ROP in progress (deployed, but bypassable):
  kBouncer: Efficient and Transparent ROP Mitigation
  ROPecker: A Generic and Practical Approach for Defending Against ROP Attacks
- control flow integrity

Control Flow Integrity

Proposed in 2009 by Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti in *Control-Flow Integrity: Principles, Implementations, and Applications*.

Core idea: whenever a hijackable control flow transfer occurs, make sure its target is something it's *supposed* to be able to return to!

This spawned a whole arms race.

Counter-CFI techniques:

B(lock)OP: ROP on a block (or multi-block) level by carefully compensating for side-effects.
J(ump)OP: instead of returns, use indirect jumps to control execution flow
C(all)OP: instead of returns, use indirect calls to control execution flow
S(ignreturn)ROP: instead of returns, use the `sigreturn` system call
D(ata)OP: instead of hijacking control flow, carefully overwrite the program's data to puppet it

Control Flow Integrity: Intel Edition!

Intel recently (like, September 2020) released processors with *Control-flow Enforcement Technology* (CET).

Among other things, CET adds the `endbr64` instruction.

On CET-enabled CPUs, indirect jumps (including `ret`, `jmp rax`, `call rdx`, etc) MUST end up at an `endbr64` instruction or the program will terminate.

This is still bypassable by some advanced ROP techniques (Block Oriented Programming, SROP, etc), but it will significantly complicate exploitation.

Issue: Hacking Blind?

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

Intuition: things are only randomized at *program start time*.

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