CS-3943-G / CS-9223-H

Introduction to ROP

Pwning so far

- So far all the binaries we've pwned have either:
  - Had NX off, allowing us to write shellcode
  - Had a "give_shell" function
  - Had "system" imported

Pwning so far

- Most real programs don’t have any of these properties
  - So they’re secure, right? We’re all out of a job!
  - Time to retire…
  - Just kidding, we can still win against the bugs!

Return Oriented Programming

- ROP (Return Oriented Programming) to the rescue!
  - ROP is a Code-Reuse attack, which means we don’t write our own code to insert into the program
  - Instead, we use the bytes already in the program, which are marked executable
  - Of course, the clever programmers have gotten rid of all the useful functions, like “system”, so we’ll need to be clever

Some observations about x86

- Functions aren’t really a “thing” in x86…
  - You can jump to the middle of a function, and that’s all fine!
- Instructions are variable-length in x86…
  - You can jump into the middle of an instruction, and that’s fine!
  - Jumping at different points yields different opcodes, which means we have a lot more instructions to use than the programmer intended
- If we can smash the callstack, we can control return pointers
  - That means we can do evil things like return, then return again, etc...

Functions in x86

- x86 *kinda* understands functions
- What it really understands is the call stack
  - *call* and *ret* instructions push and pop to the callstack, for instance
- What does the *ret* instruction really do?
  - *pop rip*
  - If we control the stack, we control what it pops
    - That means we can control multiple returns, each one making small changes

Instructions in x86

- Instructions are multiple bytes long, and they're variable length
  - Many instructions, like *ret*, are 1 byte (opcode = c3)
  - That means, if you look for all instances of c3, you can find sequences of instructions that lead up to a ret
- You can just jump to any byte, and the processor will happily decode it and run it…
  - So you can jump into the *middle* of multi-byte instructions

Example

- mov rax, 0xc30f05 ; ret == 48 c7 c0 05 0f c3 00 c3
  - If we offset a few bytes into this instruction sequence…
    - 05 0f c3 == “syscall ; ret”
- That’s a pretty powerful example. Maybe a little contrived…
  - In a large enough program you’ll have tons of these little “gadgets” to do fun things with
- There are programs that find these gadgets for you:
  - I like rp++, some people swear by ROPGadget

pop ; pop ; pop ; ret

- Remember, arguments are passed in registers!
  - If we want to pass arguments to functions, we need to set registers
- Luckily, all the *pop <register>* instructions are 1-byte long
  - Normal programs might even have them leading up to a *ret*
- If we want to set the first argument, we might find a *pop rdi ; ret* gadget, and then have our next return go to the function.
  - [pop_rdi_ret, hello_world, puts]

Figuring out where libc is

- On modern systems, we put shared libraries in a different place every time the program starts
  - That way, we can’t just predict where “system” will be loaded and jump to it
- Some part of our program needs to know where it is, to call functions…
  - The GOT holds pointers to libc once they get resolved!
  - If we can read some bytes of out the GOT… we can know where libc is!
  - And since libc offsets from the base are always the same…
    - We can calculate any address in libc from a single GOT leak!

Imagine this exploit...

- We have full control of the stack
  - The program did gets(buf), or something similarly vulnerable
- We start our ROP…
  - puts(puts_got_address) to leak it
  - This prints out the address of puts in libc
  - We subtract the offset of puts in libc from that address to get the base address of libc
  - Then we add the offset of system in libc to get the real address of system
  - We jump there, with an argument of “/bin/sh”, and get a shell
  - Cool!