Module:
Return Oriented Programming

Techniques

Yan Shoshitaishvili
Arizona State University

ROP Concept: resources on the stack

If you know where your ropchain is on the stack, you can include resources (such as a "/flag" string) in your chain.

	gadget 1 address	address of "/flag" (2 words to the right)	gadget 3 address	"/flag\0\0\0"	gadget 4 address	gadget 5 address	"/bin/cat"	gadget 6 address	gadget 7 address (syscall)	

But how do we prevent "/flag" and "/bin/cat" from being interpreted as gadget addresses (obviously, that would crash the program)?

ROP Concept: janitorial gadgets

Some of your gadgets will be there simply to unbreak your ropchain, and that is okay!

Example: stack fix-up gadgets!

```
pop r12; pop rdi; pop rsi; ret
add rsp, 0x40; ret
```

This lets you skip data on your stack.

ROP Concept: storing values into registers

You can store values into registers using register-popping gadgets. For example:

`0x400400: pop rax; ret`

	gadget 1 address (0x400400)	desired rax value	gadget 2 address	"/flag"	gadget 3 address	gadget 4 address	"/bin/cat"	gadget 5 address	gadget 6 address (syscall)	

This gadget will pop rax off the stack, then return to gadget 2. Now you've set rax!

ROP Concept: rare and common gadgets

Some gadgets are rarer than others. Relatively common ones:

`ret` (at the end of every function)
`leave; ret` (at the end of many functions)
`pop REG; ret` (restoring callee-saved registers before returning)
`mov rax, REG; ret` (setting the return value before returning)

Because you can jump into the middle of an instruction, instructions don't have to be common to appear in common gadgets!

Example: every `add rsp, 0x08; ret` also contains a `add esp, 0x08;` if you jump past the REX ("H") prefix.

ROP Concept: storing addresses into registers

This one is trickier... `lea` gadgets that do exactly what you want are rare (long instruction, and mostly used in the beginning or middle of functions, not near a ret).

1. Alternative #1: `push rsp; pop rax; ret` (equivalent to `mov rax, rsp`) will get the stack address into rax.
2. Alternative #2: `add rax, rsp; ret` (not perfect, but will conceptually get rsp into rax)
3. Alternative #3: `xchg rax, rsp; ret` (swap rax and rsp. DANGEROUS, be careful)

Once you have the stack address, later gadgets can dynamically compute necessary addresses on the stack instead of having them hardcoded. Now you (might not) need a stack leak!

ROP Concept: stack pivot

Don't like your stack? Too limiting? Try a stack pivot!

```
xchg rax, rsp; ret
pop rsp; ...; ret
```

This will *pivot* your stack to point elsewhere. That **`ret`** will read the return address off of your new stack!

ROP Concept: data transfer

Shellcode needs to move data around. So do ropchains. One common gadget:

```
add byte [rcx], al ; pop rbp ; ret
```

Obviously, this would require a gadget to set rcx (surprisingly rare) and rax (less rare).

ROP Concept: syscalls are rare

In shellcode, you use `syscall` to invoke system calls. This instruction is quite rare in normal programs (even as a part of other instructions).

You might have to call library functions, instead!

Advice: Keep it Simple.

ROP Concept: KNOW YOUR ENVIRONMENT

Your ropchain doesn't run in a vacuum! When it starts, there are useful addresses all over the place.

- code, stack, heap addresses in registers
- code, stack, heap addresses all over the stack

USE THEM.

ROP Concept: finding the rop gadgets

Many tools available (https://github.com/zardus/ctf-tools has installers for 3!). For example, rp++:

```
# rp++ --unique -r2 -f /bin/bash
```

Can also try greater values than 2, but long gadgets become increasingly unstable (side-effects!).

From here, *regular expressions are your friends*.

```
# rp++ --unique -r2 -f /bin/bash | grep -P "(add|sub|mov) rax, r.."
```