CSC 6580
Spring 2020

Instructor: Stacy Prowell

Homework:
Basic Blocks

Midterm

Midterm Topics

- Is it risky to disclose vulnerabilities (1/21)?
- Does Rice's theorem preclude analysis (1/21)?
- How might optimization break a program (1/28)?
- How do memory references work (1/28)?
- What are some simple assembly idioms (1/28-30)?
- How does two's complement arithmetic work (2/4)?
- How do you write inline assembly (2/4)?
- How does `RIP` work (2/6)?
- What do little endian and big endian mean (2/11)?
- How do the stack and `EBP` work (2/11)?
- How do you perform a system call (2/11)?
- What is a proper program (2/11)?
- What is a structured program (2/11)?
- How do you structure a program (2/13)?
- What are sections and the entry point (2/13)?
- What are basic blocks (2/25)?
- What is position-independent code (2/27)?
- What is `RIP`-relative addressing (2/27)?
- What is the PLT (3/5)?
- How do you determine if a variable is live (3/5)?
- How do you construct a simple trace table (3/5)?
- How do you do backward static slicing (3/10)?
- How to apply liveness and slicing to assembly (3/12)?

Basic Knowledge

Expect you to:

- understand binary and hexadecimal numbers;
- understand bitwise operators (and, inclusive or, exclusive or, not, shifting and rotating);
- be proficient in the C programming language, including pointers;
- have a basic familiarity with Python;
- know simple program analysis;
- know how to apply first order logic and algebra; and
- write clearly in complete sentences when explaining.

Don't memorize; apply

- Instructions used will be defined
  ...but you need to know how to understand and create programs using them.
- Theorems used will be stated
  ...but you need to understand how to apply them.
- Any calling convention used will be given
  ...but you need to know how to use it.
- Unusual code idioms will be explained
  ...but you should know how to read and write programs.

Linux System Calls

Where are these documented... really?

There are man pages. Section 2 of the man pages is devoted to the Linux system calls.
(I still like using Chapman's quick reference: https://bit.ly/2W3IXBF.)

Introduction to section 2:

```
$ man 2 intro
```

Get a list of *all* the Linux system calls:

```
$ man 2 syscalls
```

Get information on the `sys_exit` system call:

```
$ man 2 exit
```

Anti-Sandboxing

Is your code being debugged? Sandboxed?

Maybe you don't want your code to be sandboxed! Maybe you are worried about loss of trade secrets, or someone capturing a decryption key, or even someone stealing your intellectual property!

- https://www.shadesandbox.com/
- https://www.sandboxie.com/
- https://solebit.io/

The Trap Flag

The processor has a trap flag (`TF`) that causes an interrupt after a single instruction executes (`SIGTRAP`). Install a signal handler with the `ra_sigaction` system call (not that easy), set the trap flag, and then jump to the code you want to run.

Am I being debugged?

Check the trap flag, but do it stealthy. So stealthy!

```
mov ss, dx
mov edx, ss
pushf
pop edx
and edx, 0x100
rol edx, 0x18
ror edx, 0x1a
pushf
and DWORD [esp], 0xffffffbf
or [esp], edx
popf
jz tf_set
```

`pushf`	Push the `FLAGS` onto the stack	`FLAGS = 0b ... ODIT SZXA XPXC` `[ESP] = 0b ... ODIT SZXA XPXC`
`pop edx`	Pop the flags into `EDX`	`EDX = 0b ... ODIT SZXA XPXC`

`and edx, 0x100`	Mask the bit 8 (the trap flag `TF`)	`EDX = 0b ... ODIT SZXA XPXC`
		`& 0b ... 0001 0000 0000`
		`= 0b ... 000T 0000 0000`

`rol edx, 0x18`	Rotate left by 16+8 = 24 bits. It is a 32-bit register, so bit 8 ends up at position 8+24 = 32, which wraps around through the carry and ends up at bit 32-32 = 0	`EDX = 0b ... 0000 0000 000T`
`ror edx, 0x1a`	Rotate right by 16+10 = 26 bits. It is a 32-bit register, so bit 0 ends up at position 0-26 = -26, which wraps around through the carry and ends up at bit 32-26 = 6	`EDX = 0b ... 0000 0T00 0000`

`pushf`	Push the `FLAGS` onto the stack	`FLAGS = 0b ... ODIT SZXA XPXC`
		`[ESP] = 0b ... ODIT SZXA XPXC`
		`EDX = 0b ... 0000 0T00 0000`

`and DWORD [esp], 0xffffffbf`	And the 32 bit value at the top of the stack to zero out bit 6	`[ESP] = 0b ... ODIT SZXA XPXC`
		`& 0b ... 1111 1011 1111`
		`= 0b ... ODIT S0XA XPXC`
		`EDX = 0b ... 0000 0T00 0000`

`or [esp], edx`	Or the value on top of the stack with the shifted trap flag so the value is now in the `ZF` position	`[ESP] = 0b ... ODIT S0XA XPXC`
		`| 0b ... 0000 0T00 0000`
		`= 0b ... ODIT STXA XPXC`

`popf`	Pop the `FLAGS` off the stack	`FLAGS = 0b ... ODIT STXA XPXC`
`jz debugging`	Now branch if `ZF` (really the original `TF`) is set	

Paranoid Fish

"Pafish is a demonstration tool that employs several techniques to detect sandboxes and analysis environments in the same way as malware families do."

https://github.com/a0rtega/pafish

Back to Slicing (on Semantics)

Example: Slicing Semantics

Instruction	Depends
`[rax := (rax+1)%2^64, cf …]`	`rax`
`[rcx := (rax*8)%2^64]`	`rax`
`[rsp := (rsp-8)%2^64, M[rsp] := rcx]`	`rcx`
`[rsp := (rsp-8)%2^64, M[rsp] := rax]`	`stack(0)`
`[rdi := 21]`	`stack(8)`
`[_optc]`	`stack(8)`
`[rcx := M[rsp+8], rsp := (rsp+8)%2^64]`	`stack(8)`
`[rax := M[rsp+8], rsp := (rsp+8)%2^64]`	`stack(0)`
	`rax`

Example: Slicing Semantics

Instruction	Depends
[rax := (rax+1)%2^64]	rax
[rcx := (rax*8)%2^64]	rax
[M[rsp] := rcx]	rcx
[rsp := (rsp-8)%2^64]	stack(0)
	stack(8)
	stack(8)
[rsp := (rsp+8)%2^64]	stack(8)
[rax := M[rsp+8]]	stack(0)
	rax

Example: Slicing Semantics (Fixed)

Instruction	Depends
[rax := (rax+1)%2^64, cf …]	rax
[rcx := (rax*8)%2^64]	rax
[rsp := (rsp-8)%2^64, M[rsp] := rcx]	rax
[rsp := (rsp-8)%2^64, M[rsp] := rax]	rax
[rdi := 21]	stack(0)
[_optc]	stack(0)
[rax := M[rsp+8], rsp := (rsp+8)%2^64]	stack(0)
[rcx := M[rsp+8], rsp := (rsp+8)%2^64]	rax
	rax

Example: Slicing Semantics (Fixed)

Instruction	Depends
`[rax := (rax+1)%2^64]`	`rax`
	`rax`
	`rax`
`[rsp := (rsp-8)%2^64, M[rsp] := rax]`	`rax`
	`stack(0)`
	`stack(0)`
`[rsp := (rsp+8)%2^64]`	`stack(0)`
`[rcx := M[rsp+8]]`	`rax`
	`rax`

Naïve Slicing in Assembly

1. LET $D[n+1] = \{v\}$
2. FOR $i = n$ TO 1:
   a. LET w = written(inst[i]) intersect $D[i+1]$
   b. LET $D[i] = D[i+1] - w$
   c. IF w is not empty THEN LET $D[i] = D[i]$ + read(inst[i])
   d. IF $D[i]$ intersect written(inst[i]) is not empty THEN mark i as needed

Liveness Analysis and Slicing in Assembly

Liveness

Consider the block from the Python 3.7 executable.

Where does the jump go?

At each line we ask "What do the variables in the live set depend on?"

- If a variable in the live set is an lvalue, then first remove it from the set and then add all corresponding rvalues to the set.

```
block at: 0x47e0f1
  mov   r10, qword ptr [rbp + 8]
  mov   rdi, rbp
  mov   r11, qword ptr [r10 + 0x30]
  pop   rdx
  pop   rbp
  pop   r12
  jmp   r11
next: unknown
```

Liveness

block at: 0x47e0f1	Live Set (Before Line)
`mov    r10, qword ptr [rbp + 8]`	
`mov    rdi, rbp`	
`mov    r11, qword ptr [r10 + 0x30]`	
`pop    rdx`	
`pop    rbp`	
`pop    r12`	
`jmp    r11`	`r11`
next: unknown	

At the end we need to know the value of `R11`

Liveness

block at: 0x47e0f1

	Live Set (Before Line)
`mov r10, qword ptr [rbp + 8]`	
`mov rdi, rbp`	
`mov r11, qword ptr [r10 + 0x30]`	
`pop rdx`	r11
`pop rbp`	r11
`pop r12`	r11
`jmp r11`	r11

next: unknown

`R11` is not an lvalue; nothing is done to the set

Liveness

`block at: 0x47e0f1`

	Live Set (Before Line)
`mov    r10, qword ptr [rbp + 8]`	
`mov    rdi, rbp`	
`mov    r11, qword ptr [r10 + 0x30]`	`r10`
`pop    rdx`	`r11`
`pop    rbp`	`r11`
`pop    r12`	`r11`
`jmp    r11`	`r11`

`next: unknown`

`R11` is an lvalue; remove it from the set, leaving {}

Add the lvalue `R10` to the set

Liveness

`block at: 0x47e0f1`

	Live Set (Before Line)
`mov    r10, qword ptr [rbp + 8]`	
`mov    rdi, rbp`	`r10`
`mov    r11, qword ptr [r10 + 0x30]`	`r10`
`pop    rdx`	`r11`
`pop    rbp`	`r11`
`pop    r12`	`r11`
`jmp    r11`	`r11`

`next: unknown`

`R10` is not an lvalue; the set is unchanged

Liveness

block at: 0x47e0f1

	Live Set (Before Line)
`mov    r10, qword ptr [rbp + 8]`	rbp
`mov    rdi, rbp`	r10
`mov    r11, qword ptr [r10 + 0x30]`	r10
`pop    rdx`	r11
`pop    rbp`	r11
`pop    r12`	r11
`jmp    r11`	r11

next: unknown

`R10` is an lvalue; remove it from the set leaving {}

`RBP` is an rvalue and is added

Liveness

block at: 0x47e0f1

	Live Set (Before Line)
`mov    r10, qword ptr [rbp + 8]`	rbp
~~mov rdi, rbp~~	r10
`mov    r11, qword ptr [r10 + 0x30]`	r10
~~pop rdx~~	r11
~~pop rbp~~	r11
~~pop r12~~	r11
`jmp    r11`	r11

next: unknown

If a line does not modify anything in the live set, discard the line

Liveness

block at: 0x47e0f1	Live Set (Before Line)
`mov    r10, qword ptr [rbp + 8]`	rbp
`mov    r11, qword ptr [r10 + 0x30]`	r10
`jmp    r11`	r11

next: unknown

We obtain the reduced program

Next Time (after Spring Break): Midterm