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)

Slicing Assembly

What do we slice?

- **Assembly**
  Starts and ends with assembly. Can be tricky!
- **Semantics**
  Might not end with assembly, or might have to invent new assembly.

```
inc rax
lea rcx, [rax*8]
push rcx
push rax
mov rdi, 21
call _optc
pop rcx
pop rax
; want to know rax here
```

Slicing Assembly

What do we slice?

- **Semantics**
  Need to represent the functional effect of every instruction. There are many ways to do this.

```
inc rax
lea rcx, [rax*8]
push rcx
push rax
mov rdi, 21
call _optc
pop rcx
pop rax
; want to know rax here
```

A Simple Semantics*

`inc rax`	`rax := rax + 1 ; of := ...`
`lea rcx, [rax*8]`	`rcx := rax * 8`
`push rcx`	`rsp := rsp - 8 ; M[rsp] := rcx`
`push rax`	`rsp := rsp - 8 ; M[rsp] := rax`
`mov rdi, 21`	`rdi := 21`
`call _optc`	`...do whatever _optc does...`
`pop rcx`	`rcx := M[rsp] ; rsp := rsp + 8`
`pop rax`	`rax := M[rsp] ; rsp := rsp + 8`

```
; want to know rax here
```

* All math takes place in a finite-length bit field, so $a+b$ is really $(a+b) \bmod 2^{64}$, etc.

Aside: Ghidra P-Code

Ghidra is a reverse engineering tool developed by the NSA and made available as open source software.

https://ghidra-sre.org/

It can disassemble, do a passable job of decompilation, and has a semantics for many processors, including X86-64.

Aside: Ghidra P-Code

Opening the code in Ghidra displays the usual disassembly.

But… you can click on the "jenga" button above the code, then switch to the "instruction" tab, and right-click and enable PCode...

```
00401000 48 ff c0         INC      RAX
00401003 48 8d 0c         LEA      RCX,[RAX*0x8]
         c5 00 00
         00 00
0040100b 51               PUSH     RCX
0040100c 50               PUSH     RAX
0040100d bf 15 00         MOV      EDI,0x15
         00 00
00401012 e8 0f 00         CALL     _optc
         00 00
00401017 59               POP      RCX
00401018 58               POP      RAX
```

Aside: Ghidra P-Code

...and the listing is populated with P-Code semantic information!

Find the P-Code reference manual in the Ghidra distribution, or online:

ghidra.re/courses/languages/html/pcoderef.html

```
00401000 48 ff c0         INC        RAX
                                              OF = INT_SCARRY RAX, 1:8
                                              RAX = INT_ADD RAX, 1:8
                                              SF = INT_SLESS RAX, 0:8
                                              ZF = INT_EQUAL RAX, 0:8
00401003 48 8d 0c         LEA        RCX,[RAX*0x8]
         c5 00 00
         00 00
                                              $U6d0:8 = INT_MULT RAX, 8:8
                                              RCX = COPY $U6d0
0040100b 51               PUSH       RCX
                                              $U2510:8 = COPY RCX
                                              RSP = INT_SUB RSP, 8:8
                                              STORE ram(RSP), $U2510
0040100c 50               PUSH       RAX
                                              $U2510:8 = COPY RAX
                                              RSP = INT_SUB RSP, 8:8
                                              STORE ram(RSP), $U2510
0040100d bf 15 00         MOV        EDI,0x15
         00 00
                                              RDI = COPY 21:8
00401012 e8 0f 00         CALL       _optc
         00 00
                                              RSP = INT_SUB RSP, 8:8
                                              STORE ram(RSP), 0x401017:8
                                              CALL *[ram]0x401026:8
00401017 59               POP        RCX
                                              RCX = LOAD ram(RSP)
                                              RSP = INT_ADD RSP, 8:8
00401018 58               POP        RAX
                                              RAX = LOAD ram(RSP)
                                              RSP = INT_ADD RSP, 8:8
```

Aside: Ghidra P-Code

...and the listing is populated with P-Code semantic information!

Find the P-Code reference manual in the Ghidra distribution, or online:

ghidra.re/courses/languages/html/pcoderef.html

```
00401000 48 ff c0        INC      RAX
                                            OF = INT_SCARRY RAX, 1:8
                                            RAX = INT_ADD RAX, 1:8
                                            SF = INT_SLESS RAX, 0:8
                                            ZF = INT_EQUAL RAX, 0:8
00401003 48 8d 0c        LEA      RCX,[RAX*0x8]
         c5 00 00
         00 00
                                            $U6d0:8 = INT_MULT RAX, 8:8
                                            RCX = COPY $U6d0
0040100b 51              PUSH     RCX
                                            $U2510:8 = COPY RCX
                                            RSP = INT_SUB RSP, 8:8
                                            STORE ram(RSP), $U2510
0040100c 50              PUSH     RAX
                                            $U2510:8 = COPY RAX
                                            RSP = INT_SUB RSP, 8:8
                                            STORE ram(RSP), $U2510
0040100d bf 15 00        MOV      EDI,0x15
         00 00
                                            RDI = COPY 21:8
00401012 e8 0f 00        CALL     _optc
         00 00
                                            RSP = INT_SUB RSP, 8:8
                                            STORE ram(RSP), 0x401017:8
                                            CALL *[ram]0x401026:8
00401017 59              POP      RCX
                                            RCX = LOAD ram(RSP)
                                            RSP = INT_ADD RSP, 8:8
00401018 58              POP      RAX
                                            RAX = LOAD ram(RSP)
                                            RSP = INT_ADD RSP, 8:8
```

Aside: Ghidra P-Code

```
OF = INT_SCARRY RAX, 1:8
RAX = INT_ADD RAX, 1:8
SF = INT_SLESS RAX, 0:8
ZF = INT_EQUAL RAX, 0:8
```

After each instruction we see the P-Code representation of the semantics.

Aside: Ghidra P-Code

```
OF = INT_SCARRY RAX, 1:8
RAX = INT_ADD RAX, 1:8
SF = INT_SLESS RAX, 0:8
ZF = INT_EQUAL RAX, 0:8
```

After each instruction we see the P-Code representation of the semantics.

Aside: Ghidra P-Code

```
OF = INT_SCARRY RAX, 1:8
RAX = INT_ADD RAX, 1:8
SF = INT_SLESS RAX, 0:8
ZF = INT_EQUAL RAX, 0:8
```

After each instruction we see the P-Code representation of the semantics.

INT_SCARRY

Parameters	Description
input0	First varnode to add.
input1	Second varnode to add.
output	Boolean result containing signed overflow condition.

Semantic statement

```
output = scarry(input0,input1);
```

This operation checks for signed addition overflow or carry conditions. If the result of adding input0 and input1 as signed integers overflows the size of the varnodes, output is assigned *true*. Both inputs must be the same size, and output must be size 1.

Aside: Ghidra P-Code

```
OF = INT_SCARRY RAX, 1:8
RAX = INT_ADD RAX, 1:8
SF = INT_SLESS RAX, 0:8
ZF = INT_EQUAL RAX, 0:8
```

The two comma-separated items after `INT_SCARRY` are the arguments. The first is `RAX`, which we recognize, and the second is the value 1, represented as an eight-byte integer.

INT_SCARRY

Parameters	Description
input0	First varnode to add.
input1	Second varnode to add.
output	Boolean result containing signed overflow condition.

Semantic statement

```
output = scarry(input0,input1);
```

This operation checks for signed addition overflow or carry conditions. If the result of adding input0 and input1 as signed integers overflows the size of the varnodes, output is assigned *true*. Both inputs must be the same size, and output must be size 1.

Aside: Ghidra P-Code

```
OF = INT_SCARRY RAX, 1:8
RAX = INT_ADD RAX, 1:8
SF = INT_SLESS RAX, 0:8
ZF = INT_EQUAL RAX, 0:8
```

Note that we specify `ZF` by checking to see if `RAX` is zero. This only works if `RAX` is *already* set to the incremented value… so these semantics are *sequential* assignments, and order matters.

Our simple semantics are *concurrent*, so the order does not matter.

A Simple Semantics*

`inc rax`	`rax := rax + 1 ; of := ...`
`lea rcx, [rax*8]`	`rcx := rax * 8`
`push rcx`	`rsp := rsp - 8 ; M[rsp] := rcx`
`push rax`	`rsp := rsp - 8 ; M[rsp] := rax`
`mov rdi, 21`	`rdi := 21`
`call _optc`	`...do whatever _optc does...`
`pop rcx`	`rcx := M[rsp] ; rsp := rsp + 8`
`pop rax`	`rax := M[rsp] ; rsp := rsp + 8`

```
; want to know rax here
```

* All math takes place in a finite-length bit field, so $a+b$ is really $(a+b) \bmod 2^{64}$, etc.

Slicing on Semantics

	Depends set
`rax := rax + 1 ; of := ...`	
`rcx := rax * 8`	
`rsp := rsp - 8 ; M[rsp] := rcx`	
`rsp := rsp - 8 ; M[rsp] := rax`	
`rdi := 21`	
`...do whatever _optc does...`	
`rcx := M[rsp] ; rsp := rsp + 8`	
`rax := M[rsp] ; rsp := rsp + 8`	

`rax`

This is what we want to know; we are slicing on the value at this point.

A Simple Semantics*

	Depends set
`rax := rax + 1 ; of := ...`	
`rcx := rax * 8`	
`rsp := rsp - 8 ; M[rsp] := rcx`	
`rsp := rsp - 8 ; M[rsp] := rax`	
`rdi := 21`	
`...do whatever _optc does...`	
`rcx := M[rsp] ; rsp := rsp + 8`	
`rax := M[rsp] ; rsp := rsp + 8`	`M[rsp]`

`rax`

The value of `RAX` is determined by `M[RSP]`. Yes, `RSP` is modified, but that only affects its value *after* the instruction.

A Simple Semantics*

	Depends set
`rax := rax + 1 ; of := ...`	
`rcx := rax * 8`	
`rsp := rsp - 8 ; M[rsp] := rcx`	
`rsp := rsp - 8 ; M[rsp] := rax`	
`rdi := 21`	
`...do whatever _optc does...`	
`rcx := M[rsp] ; rsp := rsp + 8`	`M[rsp - 8]`
`rax := M[rsp] ; rsp := rsp + 8`	`M[rsp]`
	`rax`

The next line up modifies `RSP`, which we care about. It adds 8 to the value, so `M[RSP]` becomes `M[RSP - 8]` (to get the original value).

A Simple Semantics*

	Depends set
`rax := rax + 1 ; of := ...`	
`rcx := rax * 8`	
`rsp := rsp - 8 ; M[rsp] := rcx`	
`rsp := rsp - 8 ; M[rsp] := rax`	
`rdi := 21`	`M[rsp - 8]`
`...do whatever _optc does...`	`M[rsp - 8]`
`rcx := M[rsp] ; rsp := rsp + 8`	`M[rsp - 8]`
`rax := M[rsp] ; rsp := rsp + 8`	`M[rsp]`

`rax`

The next few lines don't mention `RSP`. We make some assumptions about `_optc`.

A Simple Semantics*

	Depends set (values just before instruction)
`rax := rax + 1 ; of := ...`	
`rcx := rax * 8`	
`rsp := rsp - 8 ; M[rsp] := rcx`	
`rsp := rsp - 8 ; M[rsp] := rax`	`M[rsp]`
`rdi := 21`	`M[rsp - 8]`
`...do whatever _optc does...`	`M[rsp - 8]`
`rcx := M[rsp] ; rsp := rsp + 8`	`M[rsp - 8]`
`rax := M[rsp] ; rsp := rsp + 8`	`M[rsp]`
	`rax`

The next instruction modifies `M[RSP]`, which we don't care about (we only care about `M[RSP - 8]`. It also modifies `RSP`, which we *do* care about. It subtracts 8, so now we need to watch `M[RSP - 8 + 8]` = `M[RSP]`.

A Simple Semantics*

	Depends set (values just before instruction)
`rax := rax + 1 ; of := ...`	
`rcx := rax * 8`	
`rsp := rsp - 8 ; M[rsp] := rcx`	`rcx`
`rsp := rsp - 8 ; M[rsp] := rax`	`M[rsp]`
`rdi := 21`	`M[rsp - 8]`
`...do whatever _optc does...`	`M[rsp - 8]`
`rcx := M[rsp] ; rsp := rsp + 8`	`M[rsp - 8]`
`rax := M[rsp] ; rsp := rsp + 8`	`M[rsp]`
	`rax`

Now `M[RSP]` is an lvalue, and it is overwritten by `RCX`.

A Simple Semantics*

	Depends set (values just before instruction)
`rax := rax + 1 ; of := ...`	`rax`
`rcx := rax * 8`	`rax`
`rsp := rsp - 8 ; M[rsp] := rcx`	`rcx`
`rsp := rsp - 8 ; M[rsp] := rax`	`M[rsp]`
`rdi := 21`	`M[rsp - 8]`
`...do whatever _optc does...`	`M[rsp - 8]`
`rcx := M[rsp] ; rsp := rsp + 8`	`M[rsp - 8]`
`rax := M[rsp] ; rsp := rsp + 8`	`M[rsp]`
	`rax`

We can complete this and we determine that the final value of RAX depends only on the initial value of RAX. We can also remove code that is irrelevant.

A Simple Semantics*

	Depends set (values just before instruction)
rax := rax + 1 ~~; of := ...~~	rax
rcx := rax * 8	rax
~~rsp := rsp - 8 ;~~ M[rsp] := rcx	rcx
rsp := rsp - 8 ~~; M[rsp] := rax~~	M[rsp]
~~rdi := 21~~	M[rsp - 8]
~~...do whatever _optc does...~~	M[rsp - 8]
~~rcx := M[rsp] ;~~ rsp := rsp + 8	M[rsp - 8]
rax := M[rsp] ; ~~rsp := rsp + 8~~	M[rsp]
	rax

Code that does not modify a value we care about can be discarded.

A Simple Semantics*

	Depends set (values just before instruction)
`rax := rax + 1`	`rax`
`rcx := rax * 8`	`rax`
`M[rsp] := rcx`	`rcx`
`rsp := rsp - 8`	`M[rsp]`
	`M[rsp - 8]`
	`M[rsp - 8]`
`rsp := rsp + 8`	`M[rsp - 8]`
`rax := M[rsp]`	`M[rsp]`

`rax`

At this point slicing is done.
We can simplify. Note `RSP := RSP - 8 + 8` = `RSP`

A Simple Semantics*

	Depends set (values just before instruction)
`rax := rax + 1`	`rax`
`rcx := rax * 8`	`rax`
`M[rsp] := rcx`	`rcx`
	`M[rsp]`
	`M[rsp - 8]`
	`M[rsp - 8]`
	`M[rsp - 8]`
`rax := M[rsp]`	`M[rsp]`

`rax`

We can simplify. Note `M[RSP] := RCX` and then `RAX := M[RSP]`, so really `RAX := RCX`.

A Simple Semantics*

Depends set (values just before instruction)

`rax := rax + 1`	`rax`
`rcx := rax * 8`	`rax`
	`rcx`
	`M[rsp]`
	`M[rsp - 8]`
	`M[rsp - 8]`
	`M[rsp - 8]`
`rax := rcx`	`M[rsp]`

`rax`

We can simplify. Likewise substitute `RAX * 8` for `RCX`.

A Simple Semantics*

	Depends set (values just before instruction)
rax := rax + 1	rax
	rax
	rcx
	M[rsp]
	M[rsp - 8]
	M[rsp - 8]
	M[rsp - 8]
rax := rax * 8	M[rsp]
	rax

We can simplify. Likewise substitute RAX + 1 for RAX.

A Simple Semantics*

	Depends set (values just before instruction)
	`rax`
	`rax`
	`rcx`
	`M[rsp]`
	`M[rsp - 8]`
	`M[rsp - 8]`
	`M[rsp - 8]`
`rax := (rax + 1) * 8`	`M[rsp]`
	`rax`

We have computed the *semantics* of the entire program. We could reduce it to `lea rax, [rax*8 + 8]`.

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

Instruction	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