CSC 6580
Spring 2020

Instructor: Stacy Prowell

Assembly: Strings and Loops

Aside: jrcxz / jecxz / jcxz

`RCX` is special, and it even has its own conditional branch instruction to test whether the register content is zero. These instructions *do not test* `ZF`, but instead test the content of `RCX`, or `ECX`, or `CX`, and take the branch if it is zero.

Searched all executables in `/usr/bin`*, and found it used in `busybox` and nothing else.

Its use in `/usr/bin/busybox` appears to be *signaling whether to take a branch in a subroutine*. That is, it is an argument that specifies which of two options to take.

* `for file in $( find /usr/bin -exec file {} \; | grep ELF | cut -d: -f1 ) ; do ( objdump -Mintel -d $file | grep -q 'jrcx' && echo $file ) ; done`
NB: Not a pristine Ubuntu 19.10 installation. Your counts will likely vary (slightly).

loop

The `loop LABEL` instruction is *almost* equivalent to the sequence `dec rcx; jnz LABEL`.

The difference is that the latter instructions set flags… and the `loop` instruction does *not*.

So: Use your assembly reading skillz. What will be the exit value of this program? (The exit value will be the final value in `RDI` when `sys_exit` is called.)

```
                      section .text
                      global _start

_start:     mov rcx, 0xffff
                      cmp rcx, 76
                      mov rdi, 100
                      mov rbx, 50
.top:                 loop .top
                      mov rsi, 0
                      cmovz rdi, rbx
                      mov rax, 60
                      syscall
                      hlt
```

loope / loopne
loopz / loopnz

There are *two* variants with *four* mnemonics. These terminate when RCX reaches zero, but they can also terminate early by testing ZF.

In the code at the right, the loop terminates early because when RCX reaches 7, the ZF flag is set by the compare, and loopne terminates the loop.

Note that the *compare* happens and *then* the loopne instruction runs. This *decrements* RCX, so the return value is 6.

```
                    section .text
                    global _start

_start:   mov rcx, 10
.top:               cmp rcx, 7
                    loopne .top
                    mov rdi, rcx
                    mov rax, 60
                    syscall
                    htl
```

loope / loopne
loopz / loopnz

Here RCX reaches zero before the condition can match, to the loop exits and the return value is zero.

Remember: These forms end the loop if *either* the condition is met *or* RCX is zero.

```
                    section .text
                    global _start

_start:   mov rcx, 10
.top:               cmp rcx, -7
                    loopne .top
                    mov rdi, rcx
                    mov rax, 60
                    syscall
                    htl
```

How Common? Not very.

Instruction	Count in /usr/bin (1,190 ELF files)
`loope`	3 (`sntp`, `sudo`, `sudoreplay`)
`loopz`	0
`loopne`	1 (`dash`)
`loopnz`	0
`loop`	88

String Instructions*

Instruction	Meaning
movsb / movsw / movsd / movsq	**Move String**: Copy a byte, word, double-word, or quad-word from [DS:RSI] to [ES:RDI] and increment (or decrement) both RSI and RDI by the correct amount
cmpsb / cmpsw / cmpsd / cmpsq	**Compare String**: Compare a byte, word, double-word, or quad-word from [DS:RSI] to [ES:RDI] and increment (or decrement) both RSI and RDI by the correct amount
scasb / scasw / scasd / scasq	**Scan String**: Compare a byte, word, double-word, or quad-word from [ES:RDI] to AL, AX, EAX, or RAX and then increment (or decrement) both RDI
lodsb / lodsw / lodsd / lodsq	**Load from String**: Move the byte, word, double-word, or quad-word from [DS:RSI] to AL, AX, EAX, or RAX and then increment (or decrement) RSI
stosb / stosw / stowd / stosq	**Store to String**: Move the byte, word, double-word, or quad-word from AL, AX, EAX, or RAX to [ES:RDI] and then increment (or decrement) RDI

* NB: This table ignores two other families of string instructions: insb / insw / insd / insq / ins and outsb / outsw / outsd / outsq / outs. These input from and output to an I/O port.

The Direction Flag

The `DI` flag controls the *direction* of the string instructions. All string instructions increment (or decrement) either `RSI` or `RDI` or both.

- If `DI` is clear (0) then *increment*.
- If `DI` is set (1) then *decrement*.

Control the `DI` flag with `STD` (set direction flag) and `CLD` (clear direction flag). The flag is *commonly* clear, but it is a good idea to explicitly set or clear the flag before you use it.

* NB: It is rare, but the string instructions can be written `movs`, `cmps`, `scas`, `lods`, and `stos` (without a letter specifying size). In this case the instruction takes a memory argument. This argument's *size* determines the size of the data... but is otherwise *ignored*. Thus `movs QWORD [rdi + 12]` is the same as `movsq`.

How Common? Pretty common.

Instruction	Count in /usr/bin (1,190 ELF files)
`movs?`	1,116
`cmps?`	612
`scas?`	306
`lods?`	0
`stos?`	441
`cld?`	62
`std?`	1,176

rep / repe / repne

Repeat a string instruction some number of times.

This is a *prefix*, and not an instruction.

These work similarly to `loop` / `loope` / `loopne`. These repeat the subsequent string instruction the number of times specified in the count register RCX or until the indicated condition of the ZF flag is no longer met.

… But how does ZF get set?

```
; Copy a string from str to pad.
lea rsi, [str]
lea rdi, [pad]
mov rcx, str.len
rep movsb

; Find the first exclamation mark and
; replace it with a period.
lea rdi, [pad]
mov al, '!'
mov rcx, str.len
repne scasb
mov [rdi-1], BYTE '.'
```

rep / repe / repne

The `cmps` and `scas` instructions set (or clear) `ZF`.

```
; Copy a string from str to pad.
lea rsi, [str]
lea rdi, [pad]
mov rcx, str.len
rep movsb

; Find the first exclamation mark and
; replace it with a period.
lea rdi, [pad]
mov al, '!'
mov rcx, str.len
repne scasb
mov [rdi-1], BYTE '.'
```

rep / repe / repne

The string instructions *always* increment (or decrement) RCX and RSI and/or RDI.

```
; Copy a string from str to pad.
lea rsi, [str]
lea rdi, [pad]
mov rcx, str.len
rep movsb

; Find the first exclamation mark and
; replace it with a period.
lea rdi, [pad]
mov al, '!'
mov rcx, str.len
repne scasb
mov [rdi-1], BYTE '.'
```

Aside: Ambiguity

You must specify the data width here because it is *ambiguous*. The instruction `mov al, '!'` is not ambiguous, since `AL` is a byte value.

The instruction `mov BYTE [rdi-1], '.'` would have also worked.

```
; Copy a string from str to pad.
lea rsi, [str]
lea rdi, [pad]
mov rcx, str.len
rep movsb

; Find the first exclamation mark and
; replace it with a period.
lea rdi, [pad]
mov al, '!'
mov rcx, str.len
repne scasb
mov [rdi-1], BYTE '.'
```

PIC, PIE, PLT, GOT, and RDI-relative

Position Independent Executable (PIE)

Some instructions require information about the location of resources, including libraries, strings, and instructions.

```
call puts                          mov rsi, [str]                          jmp build
```

This makes it hard to use techniques like address space layout randomization (ASLR), a defensive programming technique. It also makes it hard to write shared libraries that are dynamically loaded, and which might end up anywhere in memory. It's *annoying*. You can't load two programs into the same address space because they might not be address compatible.

Ideally you would write position independent code (PIC).

-no-pie and -static

We've been telling GCC that we are not writing position independent code with `-no-pie` (do not try to create a position independent executable) and `-static` (include libraries we need in the executable so they have fixed addresses).

This is *not practical* in general, because now *every* executable has to have a copy of the libraries. We really want to have the libraries loaded in memory once, and then let everyone reference them. To make that work we need (among other things) to make them *location independent*.

```
$ gcc -o copy copy.o         ; ls -l copy         #  16,640 bytes
$ gcc -o copy copy.o -static ; ls -l copy         # 863,064 bytes
```

The Global Offset Table (GOT)

GOT is a section (`.got`) created during assembly that holds addresses of data and routines. It is "fixed up" during load by computing the memory offsets based on the actual addresses of the code, data, etc. Instead of referencing data or instruction addresses directly, you reference them via the GOT.

Now the only problem you have is finding the GOT. To do that you use a special symbol (`_GLOBAL_OFFSET_TABLE_`) and a hack to get its address into a register (typically `RBX`… which is now no longer useful since you have to use it to reference stuff: `lea [rbx+str]`). 32-bit code is full of this stuff, and 64-bit code still has it (``objdump -s -j .got `which python3` ``) and you *can* use it.

If you want to read about it you can… but 64-bit code introduced something really useful: `RIP`-relative addressing!

RIP-Relative Addressing

At load time the code is scanned and offsets to locations are "fixed up" by the loader to reference data and instruction addresses relative to `RIP`. This makes the program position independent!

You write: `lea rsi, [rel str]`, the assembler gives you: `lea rsi,[rip+0x0]`, and the linker converts this into `lea rsi,[rip+0xbc465]` once it has laid out the sections, segments, etc. Now our code is position independent!

This is so useful you can just write `default rel` at the top of your NASM file and then the instruction `lea rsi, [str]` is implicitly converted to `lea rsi, [rel str]`. If you don't want that for some reason, write `lea rsi, [abs str]`.

So far, so good

Some instructions require information about the location of resources, including libraries, strings, and instructions.

`call puts` `mov rsi, [str]` `jmp build`

We've handled the last two cases.

`mov rsi, [rel str]` `lea rax, [rel build] ; jmp rax`

The jump is overkill. If you are jumping to an address within a section, you can just use `jmp build` directly.

Next Time:
PLT, Liveness, and Trace Tables