Raluca Popa
Spring 2018

CS 161
Computer Security

Discussion 1

Week of January 22, 2018: GDB and x86 assembly

Objective: Studying memory vulnerabilities requires being able to read assembly and step through it with a debugger. In this class, we'll be using 32-bit x86 and GDB.

Note: **Feel free to come by office hours held by any of the staff.** Don't hesitate to ask for help! Our office hours exist to help you. Please visit us if you have any questions or doubts about the material.

A few useful GDB commands

For OS X users: `lldb` uses different commands. You will be expected to know `gdb`.

- run (r)
- break (b) ⟨ `func` | **addr* | `line` ⟩: add a breakpoint at the specified spot
- step (s): continue to next line, next (n): next line, skip function calls
- stepi (si), nexti (ni): same, but at the instruction level
- continue (c): until next breakpoint
- ⟨enter⟩: repeat previous command
- print (p) [/*f*] ⟨ `var` | `$register` ⟩: print the specified value (in format *f*)
- list (l) [`line`]: show source code around the current line or `line`
- layout split: splits the GDB interface into source, assembly, and commands sections.
- disassemble (disas) [`func`]: show the assembly for the current context, or `func`
- x/*n*x[b|w] *addr*: print *n* bytes (b) or 4-byte words (w) of memory as hex (x)
  (If displaying bytes, keep in mind that x86 is *little-endian*!)

Intro to x86 assembly

32-bit x86 prefixes its registers with *e-* (eax, ebp, esp...). x86–64 uses *r-* (rax, rbp, rsp...).

In AT&T syntax, the suffixes *-b*, *-i*, *-l*, and *-q* clarify if the instruction operates on bytes, 16-bit words, 32-bit words, or 64-bit words. Source is on the left, destination on the right.

There are 8 general-purpose registers: EAX, EBX, ECX, EDX, ESI, EDI, ESP, and EBP. The registers EBP (base pointer) and ESP (stack pointer) are usually used to delimit the current function's *stack frame*.

The stack grows down (towards lower addresses), by decrementing ESP (`subl $0x18, %esp`) or using the shortcut `push`: `pushl %ebp` (decrement ESP by 4 and copy EBP there).

Correspondingly, `popl %ebp` puts the memory (ESP,ESP+4) into EBP and increments ESP.

The usual *function prologue* is

```
push %ebp       // save the top of the previous frame
mov %esp %ebp   // start new frame by moving EBP down to ESP
sub X %esp      // X = size of local variables
```

And the corresponding exit is

```
add X %esp      // * (sometimes `mov %ebp %esp`)
pop %ebp        // *
ret             // pops return address from stack, goes there
```

* sometimes these two lines are replaced with just `leave`.

Conversely to `ret`, `call` ***addr*** pushes EIP (the instruction pointer, that is, the address of the *next* instruction) onto the stack as a saved return address before jumping to *addr*.

A more thorough overview of 32-bit x86 can be found at https://www.cs.virginia.edu/~evans/cs216/guides/x86.html

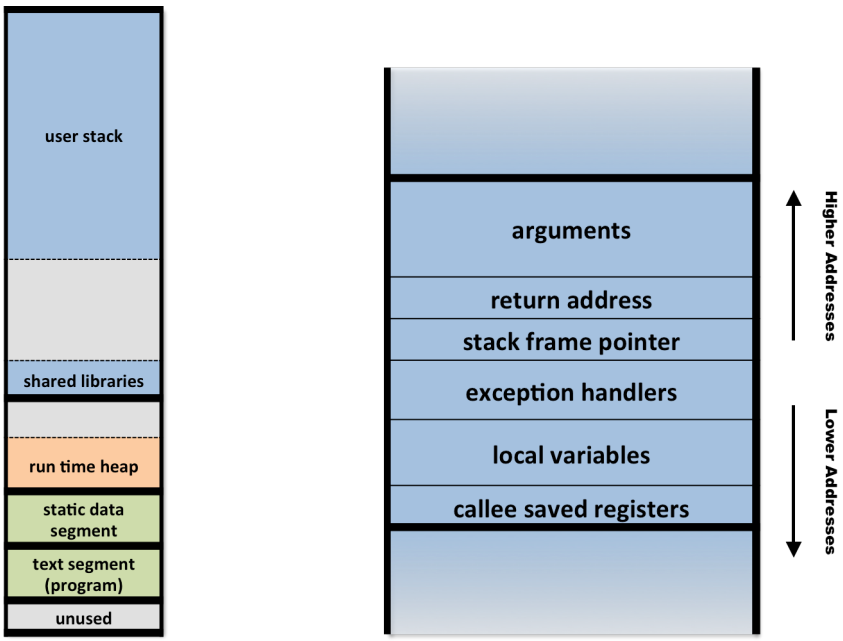
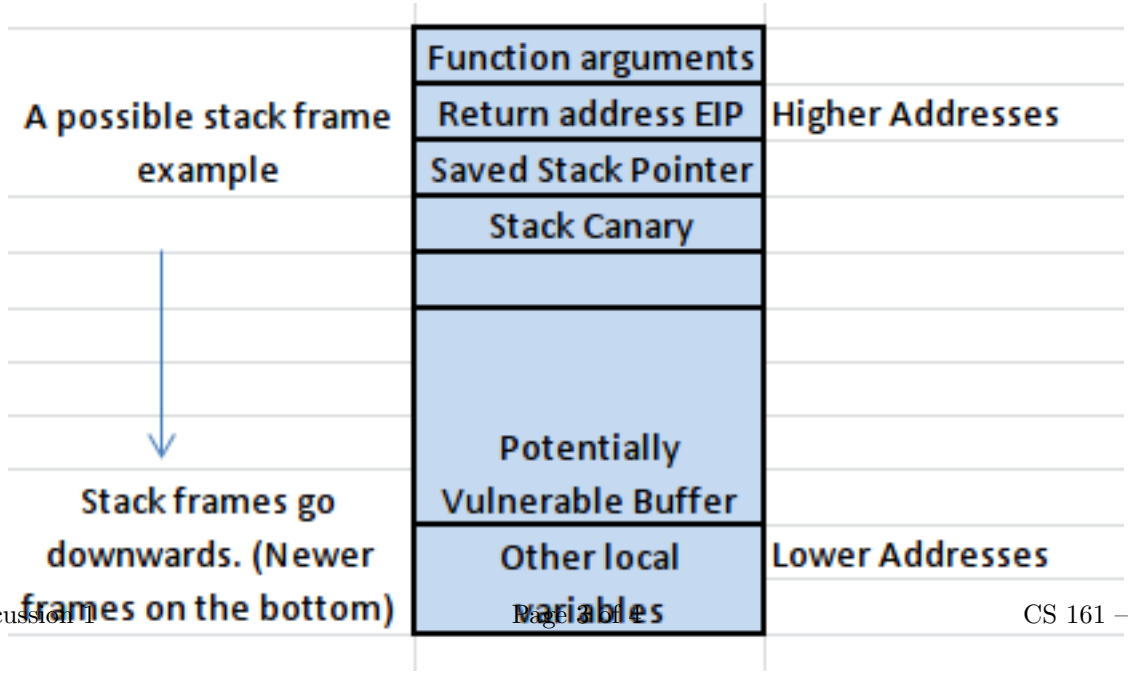


Figure 1: Left: memory layout for 32-bit Linux. The stack (left, at top) grows downward. Right: the contents of one frame on the stack (exercise: match the entries up with the instructions in the function prologue and exit).

Question 1 ***A Short Discussion on Canaries*** **(10 min)**



Question 2 ***C Memory Defenses*** **(30 min)**

In the Thursday Lecture, Professor Raluca described some C memory vulnerabilities and defenses for those insisting on writing C or C++ code.

Mark the following statements as True or False and justify your solution. Please feel free to discuss with students around you.

1. Stack canaries can not protect against all buffer overflow attacks in the stack.

   ---

2. A format-string vulnerability can allow an attacker to overwrite a saved return address even when stack canaries are enabled.

   ---

3. If you have data execution prevention/executable space protection/NX bit, an attacker can write code into memory to execute.

   ---

4. If you have a non-executable stack and heap, buffer overflows are no longer exploitable.

   ---

5. If you have a non-executable stack and heap, an attacker can use Return Oriented Programming.

   ---

6. If you use a memory-safe language, buffer overflow attacks are impossible.

   ---

7. ALSR, stack canaries, and NX all combined are insufficient to prevent exploitation of all buffer overflow attacks.

Solution:

1. True, stack canaries defeated if they are revealed by information leakage, or if there is not sufficient entropy, an attacker can guess the value. Remember, the attack just needs to work once in the real world.
2. True, with format string vulnerabilities, the attacker can learn the contents of the stack frame, other parts of memory, and write to other addresses in memory. Stack canaries won't save you here.
3. Many attacks rely on writing malicious code to memory and then executing them. If we make writable parts of memory non-executable, these attacks cannot succeed.

4. False, an attacker can still use techniques like Return Oriented Programming.
5. True, Return oriented programming is a technique that uses existing instructions already in memory to change the original program flow.
6. True, buffer overflow attacks do not work with memory safe languages.
7. True, all of these protections can be overcome.

Short answer!

1. What would happen if the stack canary was between the return address and the saved frame/base pointer?

   ---

2. What if the canary was above the return address?

Solution:

1. An attacker can overwrite the saved frame pointer so when the program tries to return, it uses the wrong address as the return address.
2. It doesn't stop an attacker from overwriting the return address. Although if an attacker had absolutely no idea where the return address, it could potentially detect stack smashing.