Reverse Engineering

Stack Review

Instructor:	G Leaden	**Due:**	See Syllabus (Friday)
Email:	g.leaden1@marist.edu	**Place:**	Hancock 2023

Goals:

The stack is one of the most fundamental data structures in computer science. - Wikipedia
This is to ensure you know the basics of how stacks operate at the machine level.

Instructions:

1. Draw a sample stack layout. x86 or x64.
2. What purpose does a stack serve? Give three examples.
3. What is a buffer overflow?
4. How do you protect against buffer overflow attacks? Give three examples.

Submitting:

Print out the answers, and hand it in on the due date.

Grading Rubric:

1	10%
2	30%
3	30%
4	30%

ANSWERS

1. Draw a sample stack layout. x86 or x64.

...	...
ESP-0xC	local variable#2, marked in IDA as var_8
ESP-8	local variable#1, marked in IDA as var_4
ESP-4	saved value ofEBP
ESP	Return Address
ESP+4	argument#1, marked in IDA as arg_0
ESP+8	argument#2, marked in IDA as arg_4
ESP+0xC	argument#3, marked in IDA as arg_8
...	...

2. What purpose does a stack serve? Give three examples.

- Save the functions return address.
  When calling another function with a CALL instruction, the address of the point exactly after the CALL instruction is saved to the stack and then an unconditional jump to the address in the CALL operand is executed.
- Passing function arguments.

```
push arg3
push arg2
push arg1
call f
add esp, 12 ; 4*3=12
```

  Functions get their arguments from the stack pointer (ESP in x86).

ESP	return address
ESP+4	argument#1, marked in IDA as arg_0
ESP+8	argument#2, marked in IDA as arg_4
ESP+0xC	argument#3, marked in IDA as arg_8
...	...

- Local variable storage. A function could allocate space in the stack for its local variables just by decreasing the stack pointer towards the stack bottom.
  Its very fast, no matter how many local variables are defined.
- Automatic deallocation. One of the reasons we store local variables and SEH records in the stack is that they are freed automatically upon function exit, using just one instruction to correct the stack pointer (often ADD).
  Function arguments, are also deallocated automatically at the end of function.
  In contrast, everything stored in the heap must be deallocated explicitly.

3. What is a buffer overflow?
   A buffer overflow occurs when data written to a buffer also corrupts data values in memory addresses adjacent to the destination buffer due to insufficient bounds checking. This can occur when copying data from one buffer to another without first checking that the data fits within the destination buffer.

4. How do you protect against buffer overflow attacks? Give three examples. Explain.

   - Bounds checking!
     Self explanatory
   - Stack Canary
     Detect that a stack buffer overflow has occurred and thus prevent redirection of the instruction pointer to malicious code.
     Like a canary in a coal mine, this method works by placing a small integer, the value of which is randomly chosen at program start, in memory just before the stack return pointer. Most buffer overflows overwrite memory from lower to higher memory addresses, so in order to overwrite the return pointer (and thus take control of the process) the canary value must also be overwritten. This value is checked to make sure it has not changed before a routine uses the return pointer on the stack.
   - Nonexecutable Stack
     Prevent the execution of malicious code from the stack without directly detecting the stack buffer overflow.
     Enforce a memory policy on the stack memory region that disallows execution from the stack (WˆX, "Write XOR Execute").
   - Randomization
     Randomize the memory space such that finding executable code becomes unreliable.
     Randomizing the memory layout will, as a concept, prevent the attacker from knowing where any code is. However, implementations typically will not randomize everything; usually the executable itself is loaded at a fixed address and hence even when ASLR (address space layout randomization) is combined with a non-executable stack the attacker can use this fixed region of memory.