Weeks 6 and 7. Midterm, The Stack, Loops, Scanf(), and Printf()

The Stack

- The stack is one of the most fundamental data structures in computer science. - Wikipedia

- What is it?

- The stack is the memory set aside as scratch space for a thread of execution.
  - When a function is called, a block is reserved on the top of the stack for local variables and some bookkeeping data.
    - When that function returns, the block becomes unused and can be used the next time a function is called.
- The stack is always reserved in a LIFO (last in first out) order; the most recently reserved block is always the next block to be freed.
  - This makes it really simple to keep track of the stack; freeing a block from the stack is nothing more than adjusting one pointer.
- A block of memory along with the ESP or RSP register in x86 or x64 as a pointer within that block.
- PUSH subtracts 4 from the ESP in 32-bit mode or 8 from the RSP in 64-bit mode and then writes the contents of its sole operand to the memory address pointed by ESP/RSP.
- POP is the reverse operation: retrieve the data from the memory location that SP points to, load it into the instruction operand (often a register) and then add 4 (or 8) to the stack pointer.
- After stack allocation, the stack pointer points at the bottom of the stack. PUSH decreases the stack pointer and POP increases it. The bottom of the stack is actually at the beginning of the memory allocated for the stack block.

- **Stack Layout example**

...	...
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
...	...

- **What is its purpose?**
  - Save the function's return address
    - Briefly touched on in Week 4 Assembly
      - 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.
      - The `CALL` instruction is equivalent to a `PUSH address_after_call / JMP` operand instruction pair.
      - `RET` fetches a value from the stack and jumps to it —that is equivalent to a `POP tmp / JMP tmp` instruction pair.
    - Show an example of this by looking at stack in GDB while stepping through a program that calls a simple method such as printf() and a complex method (user created)
  - 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
...	...

- The callee function does not have any information about how many arguments were passed. C functions with a variable number of arguments (like printf()) determine their number using format string specifiers (which begin with the % symbol).
- Ex:
  - `printf("%d %d %d", 1234);`
- printf() will print 1234, and then two random numbers which were lying next to it in the stack. That's why it is not very important how we declare the main() function:
  - `main()`
  - `main(int argc, char *argv[])`
  - `main(int argc, char *argv[], char *envp[])`
- C Runtime Library-code is calling main() roughly as:

```
push envp
push argv
push argc
call main
...
```

- If you declare main() as main() without arguments, they are, nevertheless, still present in the stack, but are not used. If you declare main() as main(int argc, char *argv[]), you will be able to use first two arguments, and the third will remain "invisible" for your function. Even more, it is possible to declare main(int argc), and it will work.
- **NOTE:**
  - This is not the *only* way to pass parameters to functions. Just a service that the stack provides. To learn more about other ways to pass parameters, pg 33 and 34 of your book should be helpful, as well as the Theory of Programming Languages course.

- 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.
- It's very fast, no matter how many local variables are defined.
  - It is not a requirement to store local variables in the stack. You could store local variables wherever you like, but traditionally this is how it's done.
- 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.

- **What is a buffer overflow?**
  - When a program, while writing data to a buffer (stack), overruns the buffer's (stack's) boundary and overwrites adjacent memory locations.
  - 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.
  - Ex.

```
#include <string.h>

void foo (char *bar)
{
   char  c[5];
   strcpy(c, bar);  // no bounds checking
}

int main (int argc, char **argv)
{
   foo(argv[1])
  return 0;
}
```

- Any words that are 5 characters or greater will corrupt or overflow the stack.
  - The maximum number of characters that is safe is one less than the size of the buffer here because in the C programming language strings are terminated by a null byte character.
    - A 5-character input requires 6 bytes to store.
    - If you manage to overflow the stack to the point where the return address is overwritten, specifically with little/big endian (more on that later) like: "`\x08\x35\xC0\x80`" the function (in our case foo()) will pop that return address off of the stack and jumps to it (starts executing from there).
      - Overwriting the return address with a pointer to the stack buffer char c[5] that contains attacker-supplied data (typically shell code suitable to the platform and desired function).
      - If this program had special privileges running as root or a sudoer, then the attacker could use this vulnerability to gain superuser privileges on the affected machine.
- **How to protect against it?**
  - **The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities**
    - **by Mark Dowd (Author), John McDonald (Author), Justin Schuh (Author)**
  - **Bounds checking!**
  - **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 (W^X, "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.
- NOTE:
  - While these approaches does fix the approach of a stack buffer overflow, it does not stop all approaches arbitrary shellcode execution.

Printf() with arguments

- Let's try and use printf with some arguments this time.
  - How do arguments work in printf? How can we find out?
    - `man printf`

```
#include <stdio.h>
int main() {
    printf("a=%d; b=%d; c=%d", 1, 2, 3);
    return 0;
};
```

- Compiling this with gcc and taking a look with radare2
  - We see that the program is using multiple MOV instructions, this is directly affecting the stack (instead of PUSH POP that MSVC uses).
  - Set a breakpoint on printf and run the debugger
    - `db sys.printf`
    - `dc`
  - Look at the stack

  - `pxr @ esp/rsp`
- Step into/over
  - `s/S`
- First element is return address
- Second element is the format string address
  - content: “a=%d; b=%d; c=%d”
- Next 3 elements (element 3, element 4, element 5) are the printf arguments
  - content: 1, 2, 3

Scanf()

- What does scanf do?
  - Manpage it!
- Consider this code:

```
#include <stdio.h>
int main() {
    int x;
    printf ("Enter X:\n");
    scanf ("%d", &x);
    printf ("You entered %d...\n", x);
    return 0;
```

};

- Over the span of the function’s execution, EBP is pointing to the current stack frame making it possible to access local variables and function arguments via EBP+offset.

- It is also possible to use ESP for the same purpose, although that is not very convenient since it changes frequently. The value of the EBP could be perceived as a frozen state of the value in ESP at the start of the function’s execution.
- MOV is used for all argument passing, same as in printf. You can step through this program as well if you like to demonstrate.
- This simple example is a demonstration of the fact that compiler translates list of expressions in C/C++- block into sequential list of instructions. There are nothing between expressions in C/C++, and so in resulting machine code, there are nothing between, control flow slips from one expression to the next one.

Loops

- **For**
  - Take this code block:

```c
// For loop example.
#include <stdio.h>

void printingFunction(int i) {
        printf ("f(%d)\n", i);
}

int main() {
    int i;
    for (i=2; i<10; i++){
        printingFunction(i);
    }
}
```

  - And input it into godbolt.org compiler. Go over how gcc 8.2 looks like, compare it to gcc 4.4.7 (closest to book representation), and then with 4.4.7 add the argument `-O3` and compare those with the class.
    - Huh, GCC just unwound our loop.
  - Loop unwinding has an advantage in the cases when there aren't much iterations and we could cut some execution time by removing all loop support instructions. On the other side, the resulting code is obviously larger.
  - Big unrolled loops are not recommended in modern times, because bigger functions may require bigger cache footprint.
    - As you can see it doesn't fully unwind the loop with gcc 8.2 (it has learned!)
  - However, if you change the loop to go from 10 to 100, even 4.4.7 will choose not to unwind that loop.
  - Have the class take note of this.
  - Compile this and open in radare2
    - Look at it in visual mode, see anything that denotes a loop? (visual mode: `VV`)

- Lastly, take note of the placement of the `i` initialization and see how it is outside of loop execution, if the compare does not match the loop will never execute.

- **While**
  - Repeat the above test/demo with a while. What changes?
    - Nothing!