#### CS 240 Laboratory 10 Introduction to Buffer Overflow

Create 4 exploit strings to use as input to a vulnerable executable called laptop.bin

Each exploit is a **buffer overrun** attack (intentionally overfilling the stack with specific values, causing the program to execute differently than it normally does)

The exploits increase in sophistication from exploit 1 to exploit 4

## Repository

**laptop.c**: you are given important parts of the C code that you can read to help you understand how the compiled version works.

This is similar to how you were given the **main.c** code for the x86 assignment to help you understand how **adventure** worked.

**laptop.bin**: compiled/executable version of laptop.c You will run **laptop.bin** using a file containing input to the program.

This is similar to how you ran **adventure.bin** using **inputs.txt** in the x86 assignment.

You will design the input to the program to take advantage of how the stack is used to get the program to do something different than what was originally intended (this is called an *exploit*).

exploit1.hex, exploit2.hex, exploit3.hex, exploit4.hex: input files you will create for each of the 4 exploits you will design.

These correspond to **inputs.txt** for the x86 assignment, except that you use a separate file for each exploit.

Each exploit file must contain the correct number of bytes with the correct hexadecimal value for each byte to represent your exploit.

hex2raw.bin: tool/executable file you are given which helps you prepare to run your exploits

This executable file will take the input you design for an exploit (for example, **exploit1.hex)** and turn it into the proper raw byte form (**exploit1.bytes**) needed to run the program.

responses.txt: file for English descriptions of your exploits

This is similar to **descriptions.txt** in the x86 assignment.

# **Designing Stack Exploits**

laptop uses a function Gets () that is poorly designed

Gets () is called from function getbuf()

```
unsigned long long getbuf() {
    char buf[36];
    // other statements
    unsigned long long val = (unsigned long long)Gets(buf);
    // other statements
}
```

buf, an array of 36 bytes, is a local variable, so will by stored on the stack

Gets() takes buf as a parameter

Gets(char\* buf)

- **Gets()** accepts a string from standard input and stores the bytes from the string in **buf** (on the stack)
- Gets() does NOT check to see if the string accepted is longer than the size of the array!
- If the string is too long, it may overwrite important values on the stack (such as return addresses).

X86 code and stack diagram for the execution of getbuf function

ump of assembler code for function getbuf:						
	0x0000000000400df0		push	%rbp	%rsp <sub>0</sub> ->	ret addr
	0x0000000000400df1	<+1>:	mov	%rsp,%rbp		
	0x0000000000400df4	<+4>:	sub	\$0x30,%rsp		
	0x0000000000400df8	<+8>:	lea	-0x30(%rbp),%rdi		
	0x0000000000400dfc	<+12>:	callq	0x400cf0 <gets></gets>		
	0x0000000000400e01	<+17>:	movabs	\$0xcccccccccccd,%rdx		
	0x0000000000400e0b	<+27>:	mov	%rax,%rcx		
	0x0000000000400e0e	<+30>:	mul	%rdx		
	0x0000000000400e11	<+33>:	shr	\$0x5,%rdx		
	0x0000000000400e15	<+37>:	lea	(%rdx,%rdx,4),%rax		
	0x000000000400e19	<+41>:	mov	%rcx,%rdx		
	0x000000000400e1c	<+44>:	shl	\$0x3,%rax		
	0x000000000400e20	<+48>:	sub	%rax,%rdx		
	0x0000000000400e23	<+51>:	mov	\$0x24,%eax		
	0x000000000400e28	<+56>:	cmp	\$0x24,%rdx		
	0x000000000400e2c	<+60>:	cmovae	%rdx,%rax		
	0x000000000400e30	<+64>:	xor	%ecx,%ecx		
	0x0000000000400e32	<+66>:	add	\$0x1e,%rax		
	0x000000000400e36	<+70>:	and	\$0xffffffffffffff,%rax		
	0x0000000000400e3a	<+74>:	sub	%rax,%rsp		
	0x0000000000400e3d	<+77>:	lea	0xf(%rsp),%r8		
	0x0000000000400e42	<+82>:	and	\$0xffffffffffffff0,%r8		
	0x0000000000400e4	6 <+86>:	nopw	%cs:0x0(%rax,%rax,1)		
	0x0000000000400e5	0 <+96>:	movzb.	1 -0x30(%rbp,%rcx,1),%edi		
	0x0000000000400e5	5 <+101>:	lea	(%r8,%rcx,1),%rsi		
	0x0000000000400e5	9 <+105>:	add	\$0x1,%rcx		
	0x0000000000400e5	d <+109>:	cmp	\$0x24,%rcx		
	0x0000000000400e6	1 <+113>:	mov	%dil,(%rsi)		
	0x0000000000400e64	4 <+116>:	jne	0x400e50 <getbuf+96></getbuf+96>		
	0x0000000000400e6	6 <+118>:	mov	%rdx,%rax		
	0x0000000000400e6	9 <+121>:	leave	9		
	0x0000000000400e6	a <+122>:	retq	-		

# **Formatting Exploit Strings**

Use a string to represent the values of the bytes in the exploit

## Given the string

"01 02 03 04*"* 

The string is represented by the ASCII values for each character:

If the raw bytes we want are actually:

0x01 0x02 0x03 0x04

We use a tool called hex2raw.bin to convert "01 02 03 04" to 0x01 0x02 0x03 0x04

- In the string, each byte is written as pair of hexadecimal digits
- Successive bytes may be separated by spaces.
- The output of hex2raw.bin is a raw byte sequence, where each byte has the hexadecimal value described by the corresponding pair of characters in the input.

## NOTE: do not use **OA** in your exploit strings!

Your exploit string must not contain 0A, since this is the ASCII code for newline ( ' n' ).

When Gets() encounters this byte, it will assume you intended to terminate the string input, and will ignore the rest of your values.

hex2raw.bin will warn you if it encounters this byte value.

## **Running and Testing Exploits**

- 1. Write the exploit string in a file, for example exploit1.hex
- 2. Translate it to raw bytes with hex2raw

\$ ./hex2raw.bin < exploit1.hex > exploit1.bytes

3. Run it directly (possible for Exploits 1 and 2):

\$ ./laptop.bin -u your cs username < exploit1.bytes</pre>

or run it under gdb (required for Exploits 3 and 4):

\$ gdb ./laptop.bin

(gdb) run -u your\_cs\_username < exploit1.bytes

4. If you change your exploit string in exploit1.hex, you must always run hex2raw.bin to create a new version of exploit1.bytes before running again.

#### Why do exploit 3 and 4 need to be run from gdb?

Exploits 3 and 4 require putting code on the stack, and executing it from there.

Running program stored on the stack is not normally allowed (for security reasons).

However, from within the debugger, it is safe to do so (so, we must test those exploits using **gdb**).