15-213 / 15-513, Fall 2023
Attack Lab: Understanding Buffer Overflow Bugs

Assigned:	Thursday, September 21, 11:59PM ET
Due:	Thursday, September 28, 11:59PM ET
Last possible time to turn in:	Sunday, October 1, 11:59PM ET
Maximum number of grace days:	1

1 Introduction

This assignment involves generating a total of five attacks on two programs which have different security vulnerabilities.

In this lab, you will:

- Learn different ways that attackers can exploit security vulnerabilities when programs do not safeguard themselves well enough against buffer overflows.
- Gain a better understanding of how to write programs that are more secure, as well as some of the features provided by compilers and operating systems to make programs less vulnerable.
- Gain a deeper understanding of the stack and parameter-passing mechanisms of x86-64 machine code.
- Gain a deeper understanding of how x86-64 instructions are encoded.
- Gain more experience with debugging tools such as GDB and OBJDUMP.

Note: In this lab, you will gain firsthand experience with methods used to exploit security weaknesses in operating systems and network servers. Our purpose is to help you learn about the runtime operation of programs and to understand the nature of these security weaknesses so that you can avoid them when you write system code. We do not condone the use of any other form of attack to gain unauthorized access to any system resources.

You will want to study Sections 3.10.3 and 3.10.4 of the book as reference material for this lab.

Table 1: The five phases of attack lab, with scoring

Phase	Program	Level	Method	Function	Points
1	CTARGET	1	CI	`touch1`	10
2	CTARGET	2	CI	`touch2`	25
3	CTARGET	3	CI	`touch3`	25
4	RTARGET	2	ROP	`touch2`	35
5	RTARGET	3	ROP	`touch3`	5

Table 1 summarizes the five phases of this lab. The first three involve code-injection (CI) attacks on CTARGET, while the last two involve return-oriented-programming (ROP) attacks on RTARGET.

Whenever you successfully solve a phase locally, the target program will transmit your solution to Autolab, which will test it to make sure it works reliably. You should always check your score on Autolab, after a few minutes, to ensure that you have really gotten credit for the phase.

Each phase is graded independently; you do not need to do them in the order they are listed above. Unlike the Bomb Lab, there is no penalty for making mistakes in this lab. Feel free to fire away at CTARGET and RTARGET with any strings you like.

Good luck and have fun!

2 Logistics

As usual, this is an individual project. You will generate attacks for target programs that are custom generated for you.

As usual, you must do the assignment on one of the class shark machines. A full list of these machines can be found on the course Web site at `http://www.cs.cmu.edu/~213/labmachines.html`.

2.1 Getting Files

You can obtain your target programs from the Autolab site, `https://autolab.andrew.cmu.edu`.

After logging in to Autolab, select `Attacklab -> Download handout`. The Autolab server will generate a `tar` file called `target`*k*`.tar`, where *k* is the unique number of your target programs. Upload the `tar` file to the (protected) Andrew directory in which you plan to do your work. Then login to a shark machine and give the command:

```
tar -xvf targetk.tar
```

This will extract a directory `target`*k* containing the files described below. (As with previous labs, you can also use the `autolab download` command to download `target`*k*`.tar` directly to the sharks.)

Warning: If you extract the contents of `target`*k*`.tar` on your personal computer and then upload the extracted files to the sharks, you might not upload all the necessary files, and the executable programs may lose their "executable bits." It is much more reliable to copy the `tar` file to the sharks and unpack it there.

The files in `target`*k* include:

`README.txt`: A file describing the contents of the directory.

`ctarget`: An executable program vulnerable to *code-injection* attacks.

`rtarget`: An executable program vulnerable to *return-oriented-programming* attacks.

`cookie.txt`: Contains an 8-digit hex code that you will need for level 2 and 3 attacks.

`handin`: A directory where `ctarget` and `rtarget` will keep a record of your successful attacks. Take care not to modify or delete any files in this directory unless instructed to do so by course staff.

In the following instructions, we will assume that you have already extracted your `target`*k*`.tar` on the sharks and you are working in the `target`*k* directory.

2.2 Important Points

Here is a summary of some important things to know about this lab. These may not make much sense when you read this document for the first time. They are presented here as a reference for once you get started.

- You are encouraged to research the subject of code injection and ROP attacks in as much detail as you like. The "Extra Reading" and "References" sections at the end of this document have some starting points. They also include a pointer to an x86 machine language reference, which may be very useful to you.
- You are allowed to use standard debugging tools for this assignment, such as `objdump`, `gdb`, `lldb`, `strace`, and `valgrind`. You are also allowed to write your own tools, from scratch, to assist with the assignment.
- You are **not** allowed to use any existing tools or plugins that are designed to assist in implementing code injection or ROP attacks.
- All of your attacks will destroy some of the information that `gdb` uses to keep track of what the program is doing. You should expect source-level debugging commands (e.g. `step`, `next`, `until`, printing the values of local variables) and stack-walking commands (e.g. `backtrace`, `up`, `down`, `frame`) to malfunction once your exploit starts running. Assembly-level commands, however, should still work correctly (e.g. `stepi`, `disas`, printing the values of registers).
- It is common for attacks to malfunction because they are not *quite* in the place, or the order, that you expected them to be. You should use `gdb`'s memory inspection commands to verify that each attack is loaded onto the stack correctly, before allowing it to run.
- CTARGET and RTARGET start up in an unusual manner. We recommend setting a breakpoint on `decode_line` and letting the program run until it reaches that breakpoint, rather than trying to single-step from the beginning of `main`.
- Much as you did not need to understand the code inside `__isoc99_sscanf` in order to complete bomb lab, you will not need to understand the code inside `get_hex_byte` and `print_line` to complete this lab.
- Your attacks may not bypass the functions `touch1`, `touch2`, and `touch3`. Attacks that violate this rule may appear to pass on the sharks but will fail on Autolab.
- In phases 4 and 5, you should only use gadgets whose starting addresses are in between the symbols named `start_farm` and `end_farm`. Attacks that violate this rule may or may not pass on Autolab (even if they do pass on the sharks).
- However, you can use any gadgets you discover in the farm, not just those for which we give the byte codes in Table 2.

3 Target Programs

CTARGET and RTARGET are almost the same program (we talk about the differences in Section 5). What they *do* is this: they read a sequence of byte values in hexadecimal and print the corresponding ASCII characters. For example:

```
$ ./ctarget
Cookie: 0x599051eb
Type hex bytes: 68 65 6c 6c 6f 20
Keep going: 77 6f 72 6c 64 0a
hello world
$
```

In this listing, text in **boldface** was printed by CTARGET or by the shell (it will not be in boldface if you try this yourself). I entered `68 65 6c 6c 6f 20` and then `77 6f 72 6c 64 0a`. These are the ASCII codes for "hello world" followed by a newline, and so CTARGET printed "hello world".

You can input as many lines of hex as you like. All of the characters will be stored in memory until CTARGET encounters an encoded byte with value `0a`, which is the ASCII code for newline (`'\n'`), and then they are printed all at once, after which CTARGET exits. RTARGET does exactly the same thing.

(Note: If you try this yourself, you will see a different number on the `Cookie:` line. The importance of the cookie number will be explained later.)

Both CTARGET and RTARGET read their input with this function, which has a bug:

```
/* decode_line - Read and decode hex bytes from INFILE.
   This function has a buffer overflow vulnerability.  */
void decode_line(FILE *infile) {
    unsigned char buf[BUFFER_SIZE];
    unsigned char *sp = buf;
    int c;
    while ((c = get_hex_byte(infile)) != EOF && c != '\n') {
        *sp++ = c;
    }
    *sp = '\0';
    print_line(buf);
}
```

You can see that `decode_line` stores decoded characters in a local variable `buf`, which is an array of fixed size, `BUFFER_SIZE` bytes. (`BUFFER_SIZE` is a compile-time constant, specific to your version of CTARGET and RTARGET.) Notice that the `while` loop does not stop when `BUFFER_SIZE` bytes have been read. This is the same bug that's found in the C library function `gets`: it keeps reading data until the *input* ends, possibly overrunning the bounds of the storage allocated for the data.

Because of this bug, if you enter too many encoded bytes, `ctarget` will crash:

```
$ ./ctarget
Cookie: 0x599051eb
Type hex bytes: 54 68 69 73 20 69 73 20 6e 6f 74 20 61 20 76 65
Keep going: 72 79 20 69 6e 74 65 72 65 73 74 69 6e 67 20 73
Keep going: 74 72 69 6e 67 2c 20 62 75 74 20 69 74 20 69 73
Keep going: 20 71 75 69 74 65 20 6c 6f 6e 67 0a
This is not a very interesting string, but it is quite long
Segmentation fault (core dumped)
$
```

Your task is to be more clever with the overlength byte sequences you feed CTARGET and RTARGET, so that they do something interesting, instead of crashing. These clever sequences are called *exploit strings*. You can use *any* byte value in an exploit string, not just the values that correspond to printable ASCII or Unicode—*except*, you cannot use `0a`, because the targets will stop reading input when they find that byte.

When you have correctly solved one of the levels of this lab, the target program will automatically send your exploit string to Autolab to be verified.

3.1 Reading Exploits from Files

You will probably not want to retype your exploit strings over and over while testing them out, so CTARGET and RTARGET can take input from a file, instead of reading it from the terminal. You can specify the file to use on the command line:

```
$ cat boring.txt
54 68 69 73 20 69 73 20 6e 6f 74 20 61 20 76 65
72 79 20 69 6e 74 65 72 65 73 74 69 6e 67 20 73
74 72 69 6e 67 2c 20 62 75 74 20 69 74 20 69 73
20 71 75 69 74 65 20 6c 6f 6e 67 0a
$ ./ctarget boring.txt
This is not a very interesting string, but it is quite long
Segmentation fault (core dumped)
$
```

When you run them this way, CTARGET and RTARGET do not print any prompts, or the cookie.

3.2 Formatting and Commentary

As you may have guessed, CTARGET and RTARGET ignore all white space in their input. You can break up your exploit string over as many lines as you like. In addition, you can write comments: if a # character appears anywhere in the input, everything from that character to the end of the line is ignored.

```
$ ./ctarget
Cookie: 0x599051eb
Type hex bytes: 68 65 6c  6c 6f 20  # hello SPC
Keep going:     77 6f 72  6c 64 0a  # world RET
hello world
$
```

4 Part I: Code Injection Attacks

For the first three phases, your exploit strings will attack CTARGET. This program's stack is always in the same location in memory, and data on its stack can be treated as executable code. These properties make it vulnerable to attacks where the exploit strings contain the byte encodings of executable code.

4.1 Phase 1 (CTARGET, attack level 1)

In Phase 1, you will not inject new code. Instead, your exploit string will redirect the program to execute an existing procedure.

CTARGET contains a function named `touch1`, which is normally never called. This is the source code for `touch1`:

```
void touch1(void) {
    vlevel = 1; /* Part of validation protocol */
    puts("Touch1!: You called touch1()");
    validate(1, true);
    exit(0);
}
```

The function `decode_line` is called by a function named `test`. This is the source code for `test`:

```
void test(FILE *infile) {
    decode_line(infile);
    exit(0);
}
```

When `decode_line` returns, CTARGET ordinarily resumes execution within `test`, and proceeds to call `exit`, which terminates the program. Your task is to get CTARGET to execute the code for `touch1` when `decode_line` executes its return statement, rather than returning to `test`. Notice that `touch1`'s last action is also to call `exit`. This means it is OK if your exploit corrupts the stack and makes it impossible for `touch1` to return normally, because `touch1` *won't* return.

Some Advice

- All the information you need to devise your exploit string for this level can be determined by examining a disassembled version of CTARGET. Use `objdump -d` to get this dissembled version.
- The idea is to position a byte representation of the starting address for `touch1` so that the `ret` instruction at the end of the code for `decode_line` will transfer control to `touch1`.
- Be careful about byte ordering. Remember, arrays (such as strings) are saved in index order, but values like integers are evaluated in little-endian.
- You might want to use GDB to step the program through the last few instructions of `decode_line` to make sure it is doing the right thing.
- The placement of `buf` within the stack frame for `decode_line` depends on the value of the compile-time constant `BUFFER_SIZE`, as well the allocation strategy used by GCC. You will need to examine the disassembled code to determine its position.

4.2 Phase 2 (CTARGET, attack level 2)

Phase 2 involves injecting a small amount of code as part of your exploit string.

CTARGET contains another function that is normally never called, named `touch2`. This is the source code for `touch2`:

```
void touch2(unsigned val) {
    vlevel = 2; /* Part of validation protocol */
    if (val == cookie) {
        printf("Touch2!: You called touch2(0x%.8x)\n", val);
        validate(2, true);
    } else {
        printf("Misfire: You called touch2(0x%.8x)", val);
        validate(2, false);
    }
    exit(0);
}
```

Your task in Phase 2 is to get CTARGET to execute the code for `touch2` rather than returning to `test`. But also, you must make it appear to `touch2` that you have passed a *cookie* value as its argument. The value to pass is the value printed after `Cookie:` when you run CTARGET without a file of input. You can also find the cookie value in the file `cookie.txt` that was included in `target`*k*`.tar`.

Some Advice

- You will want to position a byte representation of the address of your injected code in such a way that `ret` instruction at the end of the code for `decode_line` will transfer control to it.
- Recall that the first argument to a function is passed in register `%rdi`.

- Your injected code should set the register to your cookie, and then use a `ret` instruction to transfer control to the first instruction in `touch2`.
- It will be easiest to use `ret` instructions for all transfers of control, even when you are not returning from a call. The `jmp` and `call` instructions contain 32-bit *relative* destination addresses. Relative addresses are difficult to calculate, and 32 bits may not be enough.
- Appendix B explains how to use tools like `gcc` and `objdump` to generate the byte-level representations of instruction sequences. You can also refer to Table 2.

4.3 Phase 3 (CTARGET, attack level 3)

Phase 3 also involves injecting code to make it appear that you have called a function inside CTARGET and passed an argument. However, this time the argument must be a string.

A third function inside CTARGET that is normally never called is named `touch3`. This is the source code for `touch3` and its subroutine `hexmatch`:

```
/* Compare string to hex represention of unsigned value */
static int hexmatch(unsigned val, char *sval) {
    char *endp;
    unsigned long cval = strtoul(sval, &endp, 16);
    return (cval == (unsigned long)val && endp != sval && *endp == '\0');
}

void touch3(char *sval) {
    vlevel = 3; /* Part of validation protocol */
    if (hexmatch(cookie, sval)) {
        report_touch3("Touch3!", sval);
        validate(3, true);
    } else {
        report_touch3("Misfire", sval);
        validate(3, false);
    }
    exit(0);
}
```

Your task is to get CTARGET to execute the code for `touch3` rather than returning to `test`. You must make it appear to `touch3` as if you have passed a string representation of your cookie as its argument.

Some Advice

- You will need to include a string representation of your cookie in your exploit string. The string should consist of the eight hexadecimal digits of the cookie, ordered from most to least significant, without a leading "`0x`."
- Recall that a string is represented in C as a sequence of bytes followed by a byte with value 0. Type "`man ascii`" on any Linux machine to see the byte representations of the characters you need.
- Your injected code should set register `%rdi` to the *address* of this string.
- When `hexmatch` and `strncmp` are called, they push data onto the stack, overwriting portions of memory that held the buffer used by `decode_line`. You will need to place the string representation of your cookie somewhere that won't be damaged by this. (Hint: The attack string can be as long as you want.)

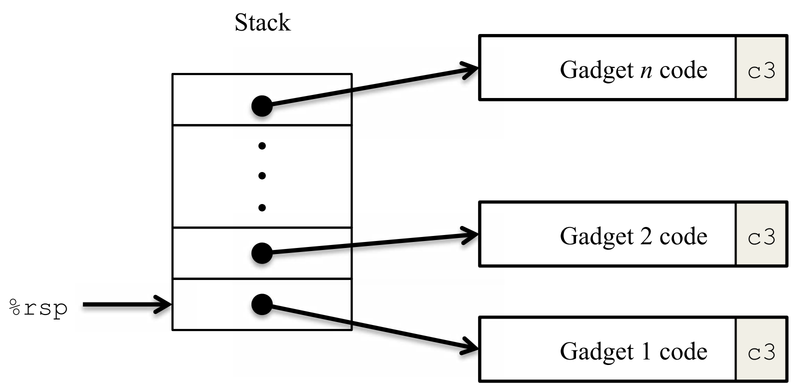

Figure 1: Setting up sequence of gadgets for execution. Byte value `0xc3` encodes the `ret` instruction.

5 Part II: Return-Oriented Programming

In the last two phases, your exploit strings must attack RTARGET instead of CTARGET. The main difference between the two is that RTARGET includes defenses against code injection:

- The location of the stack is *randomized*. Every time you run RTARGET, the stack pointer will start out at a different address. This means you cannot overwrite the return address from `decode_line` with the address of your injected code, because you don't know what address to use.
- The memory area holding the stack is marked as *non-executable*. (You'll learn how this is possible when we discuss virtual memory.) This means, if you somehow discovered the address of your injected code and jumped to it, the CPU would refuse to execute it. Instead RTARGET would crash with a segmentation fault.

Fortunately, clever people have devised strategies for getting useful things done in a program by executing existing code, rather than injecting new code. The most general form of this is referred to as *return-oriented programming* (ROP) [1, 2]. The strategy with ROP is to identify byte sequences within an existing program that consist of one or more instructions followed by the instruction `ret`. Such a segment is referred to as a *gadget*. Figure 1 illustrates how the stack can be set up to execute a sequence of *n* gadgets. In this figure, the stack contains a sequence of gadget addresses. Each gadget consists of a series of instruction bytes, with the final one being `0xc3`, encoding the `ret` instruction. When the program executes a `ret` instruction starting with this configuration, it will initiate a chain of gadget executions, with the `ret` instruction at the end of each gadget causing the program to jump to the beginning of the next.

A gadget can make use of code corresponding to assembly-language statements generated by the compiler, especially ones at the ends of functions. However, it is unlikely for *all* the gadgets you need to be available via code actually generated by the compiler. For example, it is unlikely that a compiled function would have `popq %rdi` as its last instruction before `ret`, because `%rdi` is a call-clobbered register, so the compiler has no reason to restore its value before returning.

Fortunately, the x86 uses a byte-oriented instruction set, which means you may be able to find more gadgets that start in the *middle* of an instruction. For example, `rtarget`'s gadget farm might contains the following C function:

```
void setval_210(unsigned *p)
{
    *p = 3347663060U;
}
```

The disassembled machine code for this function shows an interesting byte sequence:

```
0000000000400f15 <setval_210>:
  400f15:       c7 07 d4 48 89 c7       movl   $0xc78948d4,(%rdi)
  400f1b:       c3                      retq
```

The first instruction is six bytes long, and its last three bytes are `48 89 c7`. Considered by itself, this byte sequence encodes the instruction `movq %rax, %rdi`. (See Table 2(a) for the encodings of useful `movq` instructions, and Appendix B for how to generate and read disassembly listings like this.) The next byte after that sequence is `c3`, which encodes the `ret` instruction. Therefore, if we use `0x400f18`—three bytes after the beginning of the function—as the starting address of a gadget, that gadget will copy the 64-bit value in register `%rax` to register `%rdi`.

RTARGET contains a number of functions similar to the `setval_210` function shown above, in a region we refer to as the *gadget farm*. Your job will be to identify useful gadgets in the gadget farm and use these to perform attacks similar to those you did in Phases 2 and 3. You can see the gadget farm by using `objdump -d` on RTARGET.

Important: The gadget farm is in between the symbols `start_farm` and `end_farm` in your copy of RTARGET. If you try to use gadgets from other portions of RTARGET's code, your exploit string may not work when validated on Autolab.

5.1 Phase 4 (RTARGET, attack level 2)

The challenge of phase 4 is to repeat your level-2 attack on CTARGET—calling `touch2` with the cookie value as an integer argument—against RTARGET. Since you cannot write code into the stack, you must instead use gadgets from the gadget farm in RTARGET.

It is possible to do this using gadgets consisting of the following instruction types, and using only the first eight x86-64 registers (`%rax`–`%rdi`).

`movq`: The codes for these are shown in Table 2(a).

`popq`: The codes for these are shown in Table 2(b).

`ret`: This instruction is encoded by the single byte `0xc3`.

`nop`: This instruction (pronounced "no op," which is short for "no operation") is encoded by the single byte `0x90`. Its only effect is to cause the program counter to be incremented by 1.

Some Advice

- All the gadgets you will need can be found in the region of the code for `rtarget` in between the symbols `start_farm` and `mid_farm`.
- You can do this attack with just two gadgets.
- When a gadget uses a `popq` instruction, it will pop data from the stack. As a result, your exploit string will contain a combination of gadget addresses and data.
- Autolab will test Phase 4 exploit strings repeatedly. They must work *every time* in order to earn full credit.

5.2 Phase 5 (RTARGET, attack level 3)

Before you take on Phase 5, pause to consider what you have accomplished so far. In Phases 2 and 3, you caused a program to execute machine code of your own design. If CTARGET had been a network server, you could have injected your own code into a distant machine. In Phase 4, you circumvented two of the main devices modern systems use to thwart buffer overflow attacks. Although you did not inject your own code, you were able inject a type of program that operates by stitching together sequences of existing code. You have also gotten 95/100 points for the lab. That's a good score. If you have other pressing obligations, consider stopping right now.

The challenge of phase 5 is to repeat your level-3 attack on CTARGET—calling `touch3` with the cookie value as a string argument—against RTARGET, using gadgets from the gadget farm. Because the gadgets available to you are not really geared for string operations, phase 5 will be significantly more difficult than phase 4, and it only counts for five points. Think of it as an extra credit problem, for those who want to go beyond the normal expectations for the course.

To solve Phase 5, you will need gadgets from the region of the code for `rtarget` in between the symbols `mid_farm` and `end_farm`, as well as the code in between `start_farm` and `mid_farm`. Remember that using gadgets from outside the farm may make your exploit string not work when validated by Autolab.

In addition to the gadgets used in Phase 4, this expanded farm includes the encodings of different `movl` instructions, as shown in Table 2(c). The byte sequences in this part of the farm also contain 2-byte instructions that serve as *functional nops*, i.e., they do not change any register or memory values. Examples of these instructions are shown in 2(d).

Some Advice

- You'll want to review the effect a `movl` instruction has on the upper 4 bytes of a register, as is described on page 183 of the text.
- It is *possible* to solve Phase 5 using a sequence of eight gadgets. Depending on the contents of your farm, you may be able find a shorter solution.
- Remember that your exploit string cannot contain the newline character (byte value 0x0a) at any intermediate position, because `decode_line` will stop reading data when it finds a newline character.

A Encodings of Some x86 Instructions

Table 2: Byte encodings of instructions. All values are shown in hexadecimal.

(a) Encodings of `movq` instructions

`movq` *S* , *D*

Source	Destination *D*							
S	`%rax`	`%rcx`	`%rdx`	`%rbx`	`%rsp`	`%rbp`	`%rsi`	`%rdi`
`%rax`	48 89 c0	48 89 c1	48 89 c2	48 89 c3	48 89 c4	48 89 c5	48 89 c6	48 89 c7
`%rcx`	48 89 c8	48 89 c9	48 89 ca	48 89 cb	48 89 cc	48 89 cd	48 89 ce	48 89 cf
`%rdx`	48 89 d0	48 89 d1	48 89 d2	48 89 d3	48 89 d4	48 89 d5	48 89 d6	48 89 d7
`%rbx`	48 89 d8	48 89 d9	48 89 da	48 89 db	48 89 dc	48 89 dd	48 89 de	48 89 df
`%rsp`	48 89 e0	48 89 e1	48 89 e2	48 89 e3	48 89 e4	48 89 e5	48 89 e6	48 89 e7
`%rbp`	48 89 e8	48 89 e9	48 89 ea	48 89 eb	48 89 ec	48 89 ed	48 89 ee	48 89 ef
`%rsi`	48 89 f0	48 89 f1	48 89 f2	48 89 f3	48 89 f4	48 89 f5	48 89 f6	48 89 f7
`%rdi`	48 89 f8	48 89 f9	48 89 fa	48 89 fb	48 89 fc	48 89 fd	48 89 fe	48 89 ff

(b) Encodings of `popq` instructions

Operation	Register *R*							
	`%rax`	`%rcx`	`%rdx`	`%rbx`	`%rsp`	`%rbp`	`%rsi`	`%rdi`
`popq` *R*	58	59	5a	5b	5c	5d	5e	5f

(c) Encodings of `movl` instructions

`movl` *S* , *D*

Source	Destination *D*							
S	`%eax`	`%ecx`	`%edx`	`%ebx`	`%esp`	`%ebp`	`%esi`	`%edi`
`%eax`	89 c0	89 c1	89 c2	89 c3	89 c4	89 c5	89 c6	89 c7
`%ecx`	89 c8	89 c9	89 ca	89 cb	89 cc	89 cd	89 ce	89 cf
`%edx`	89 d0	89 d1	89 d2	89 d3	89 d4	89 d5	89 d6	89 d7
`%ebx`	89 d8	89 d9	89 da	89 db	89 dc	89 dd	89 de	89 df
`%esp`	89 e0	89 e1	89 e2	89 e3	89 e4	89 e5	89 e6	89 e7
`%ebp`	89 e8	89 e9	89 ea	89 eb	89 ec	89 ed	89 ee	89 ef
`%esi`	89 f0	89 f1	89 f2	89 f3	89 f4	89 f5	89 f6	89 f7
`%edi`	89 f8	89 f9	89 fa	89 fb	89 fc	89 fd	89 fe	89 ff

(d) Encodings of 2-byte functional nop instructions

Operation	Register *R*			
	`%al`	`%cl`	`%dl`	`%bl`
`andb` *R*, *R*	20 c0	20 c9	20 d2	20 db
`orb` *R*, *R*	08 c0	08 c9	08 d2	08 db
`cmpb` *R*, *R*	38 c0	38 c9	38 d2	38 db
`testb` *R*, *R*	84 c0	84 c9	84 d2	84 db

B Generating Byte Codes

Using GCC as an assembler and OBJDUMP as a disassembler makes it convenient to generate the byte codes for instruction sequences. For example, suppose you write a file `example.s` containing the following assembly code:

```
# Example of hand-written assembly code
        pushq   $0xabcdef           # Push value onto stack
        addq    $17,%rax            # Add 17 to %rax
        movl    %eax,%edx           # Copy lower 32 bits to %edx
```

The code can contain a mixture of instructions and data. Anything to the right of a '`#`' character is a comment.

You can now assemble and disassemble this file:

```
    $ gcc -c example.s
    $ objdump -d example.o > example.d
```

The generated file `example.d` contains the following:

```
example.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0: 68 ef cd ab 00        push   $0xabcdef
   5: 48 83 c0 11           add    $0x11,%rax
   9: 89 c2                 mov    %eax,%edx
```

The lines at the bottom show the machine code generated from the assembly language instructions. Each line has a hexadecimal number on the left indicating the instruction's starting address (starting with 0), while the hex digits after the '`:`' character indicate the byte codes for the instruction. Thus, we can see that the instruction `push $0xABCDEF` has hex-formatted byte code `68 ef cd ab 00`.

From this file, you can get the byte sequence for the code:

```
68 ef cd ab 00 48 83 c0 11 89 c2
```

This sequence can be fed to the target programs. Alternatively, you can edit `example.d` down to

```
   68 ef cd ab 00   # pushq  $0xabcdef
   48 83 c0 11      # add    $0x11,%rax
   89 c2            # mov    %eax,%edx
```

and this is also a valid input to either of the target programs.

C Extra Reading

Buffer overflow exploits have a long history. You may find these additional readings helpful. **Caution**: Examples in these readings use *32-bit* x86 assembly language.

- http://phrack.org/issues/49/14.html
- https://blog.skullsecurity.org/2013/ropasaurusrex-a-primer-on-return-oriented-programming

If you want more information on the encoding of x86 machine instructions than was shown in Table 2, you can find it in these documents:

- http://ref.x86asm.net/
- https://www.sandpile.org/
- https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

References

[1] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information System Security*, 15(1):2:1–2:34, March 2012.

[2] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *USENIX Security Symposium*, 2011.