New York University
Computer Science Department
Courant Institute of Mathematical Sciences

Course Title: Computer Systems Organization
Course Number: CSCI-UA.0201-005
Instructor: Jean-Claude Franchitti
Session: 9

Lab #2

Machine Level Programming

I. Due

April 18, 2022 at the beginning of class.

II. Objectives

The purpose of this programming lab is for you to become more familiar with assembly language, memory layout, security vulnerabilities, and machine level programs debugging. You will achieve this by working on a couple of problems that require extensive use of gdb.

III. References

1. Slides and handouts posted on the course Web site for sessions 10-19
2. Textbook Part I Chapter 3
3. Lab2 material downloadable from the Cloud.

IV. Software Required

1. Microsoft Word.
2. WinZip as necessary.
3. Linux utilities and programs as needed (e.g., tar, gcc, make, gdb)

Important: You should do all your work on Linux machines to which you have access within the CIMS cluster (please follow the instructions/PDF you received at the beginning of the semester, you should have access to the access or linserv1 server to work on the labs).

V. Assignment Steps and Questions

1. Binary Bomb Defusal:

A binary bomb is a program that consists of a sequence of phases. Each phase expects you to type a particular string on `stdin`. If you type the correct string, then the phase is *defused* and the bomb proceeds to the next phase. Otherwise, the bomb *explodes* and terminates. The bomb is defused when every phase has been defused. Your mission is to defuse one bomb before the due date.

You can obtain your bomb by navigating to a Microsoft Azure Cloud server located at the following URL: http://40.124.53.8:15213/. Enter your username (i.e., your NYU CIMS ID) and email address (i.e., your nyu.edu email) and hit the "Submit" button on the form. The server will build a bomb and return it to your browser in a `tar` file called `bombk.tar`, where `k` is the unique number of your bomb. Save the `bombk.tar` file to a (protected) directory in which you plan to do your work. Then give the command: `tar -xvf bombk.tar`. This will create a directory called `./bombk` containing the following files: `README` (identifies the bomb and its owner), `bomb` (the executable binary bomb), `bomb.c` (source file with the bomb's main routine and a friendly greeting). If for some reason you request multiple bombs, this is not a problem. Choose one bomb to work on and delete the rest.

You must perform this assignment on the `access2.cims.nyu.edu` server. The bomb will always explode if run elsewhere and there are several other tamper-proofing devices built into the bomb. You can use many tools to help you defuse your bomb and the best way is to use your favorite debugger to step through the disassembled binary. Each time your bomb explodes it notifies the Azure Cloud server, and you lose points in the final score for this problem. Phases 5 and 6 are a little more difficult than the first four phases and are worth more points. Although phases get progressively harder to defuse, the expertise you gain as you move from phase to phase should offset this difficulty. The bomb ignores blank input lines. If you run your bomb with a command line argument (e.g., `./bomb mysol.txt`), it will read the input lines from `mysol.txt` until it reaches EOF (end of file), and then switch over to `stdin` so you do not have to keep retyping the solutions to phases you have already defused. To avoid accidentally detonating the bomb, you will need to single-step through the assembly code and set breakpoints. You will also need to inspect both the registers and the memory states.

You need to provide screenshots and explanations in your report for this problem. Clarifications and corrections may be posted via the course NYU Classes Google Group. Please note that the bomb will notify your instructor automatically about your progress as you work on this problem. You can keep track of how you are doing by looking at the scoreboard at: http://40.124.53.8:15213/scoreboard. This web page is updated continuously to show your progress defusing each bomb. Your final grade for this problem is a combination of the corrected and quality of the report you submit and your performance in defusing the bomb as per the self-grading scoreboard.

There are many ways of defusing your bomb. You can examine the program in great detail before running it and figure out exactly what it does. You can also run the program using a debugger, watch what it does step by step, and use this information to defuse the bomb. Using a debugger is probably the fastest way of defusing the bomb. Please do not try and use a "brute force" approach by writing a program that will try every possible key to find the right one. This will not work as every time you guess wrong a message will be sent to the Azure Cloud server and you will lose points when the bomb explodes. You could also very quickly saturate the network with messages using a brute force approach and cause the system administrators to revoke your server access. Finally, a brute force approach will not work because you do not know how long the strings are and what characters are in them. Note that even if you made the (incorrect) assumptions that all the strings are less than 80 characters long and only contain

letters, you would have 26^{80} guesses for each phase. Therefore, a brute approach force would take a very long time to run, and you would not get the answer before the midterm submission deadline.

The best way to defuse your bomb is to use tools that are designed to help you figure out both how programs work, and what is wrong when they don't work. In particular, the GNU debugger `gdb` can trace through a program line by line, examine memory and registers, look at both the source code and assembly code (note that you are not given the source code for all the functions in your program), set breakpoints, set memory watch points, and write scripts. Please leverage the gdb resources provided on the course website. To keep the bomb from exploding every time you type in a wrong input, you will need to set breakpoints. For online documentation, type "`help`" at the gdb command prompt. You can also type "`man gdb`" or "`info gdb`" at the Linux prompt. You may also run `gdb` under `gdb-mode` in emacs if you are familiar with that editor. Another useful tool is `objdump` and you can print out the bomb's symbol table via "`objdump -t`". The symbol table includes the names of all the functions and global variables within the bomb. It also includes the names of all the functions that are called by the bomb as well as their addresses. You may learn something by looking at the function names. Also, "`objdump -d`" can be used to disassemble all of the code within the bomb. You can then look at individual functions and reading the assembly language code can tell you how the bomb works. Note that, although "`objdump -d`" gives you a lot of information, it does not tell you everything you need to know as calls to system-level functions are displayed in a cryptic form. For example, a call to `sscanf` might appear as "`8048c36: e8 99 fc ff ff call 80488d4 <_init+0x1a0>`". Therefore, to determine that the call was to `sscanf`, you would need to disassemble the program within `gdb`. Finally, another useful utility is `strings`, which will display the printable strings in your bomb. Do not hesitate to look through tools' documentation using the commands `apropos`, `man`, and `info`. In particular, "`man ascii`" might be handy. "`info gas`" will give you more information than you ever wanted to know about the GNU assembler.

2. Security Exploits Programming:

There are several ways for attackers to exploit security vulnerabilities when programs do not safeguard themselves well enough against buffer overflows. It is therefore important to understand how to write programs that are more secure and use some of the features provided by compilers and operating systems to make programs less vulnerable to attacks at runtime. As part of this problem, you need to generate a total of five attacks on two programs that are custom generated for you and have different security vulnerabilities. This problem tests your understanding of the stack and parameter-passing mechanisms of x86-64 machine code, x86-64 instructions encodings, and debugging tools such as `gdb` and `objdump`.You may also want to review sections 3.10.3 and 3.10.4 of the course textbook.

You can obtain your custom generated files by navigating to a Microsoft Azure Cloud server located at the following URL: `http://40.124.53.8:15515`. The server will create your files and return them to your browser in a `tar` file called `targetk.tar`, where `k` is the unique number of your target programs (note that it takes a few seconds to build and download your

target, so please be patient). Save the `targetk.tar` file to a (protected) directory in which you plan to do your work. Then give the command: `tar -xvf targetk.tar` (do not use WinZip or let your browser extract the files as you will risk resetting permission bits on the executable files). This will create a directory called `./targetk` containing the following files: `README.txt` (describes 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` (an 8-digit hex code that you will use as a unique identifier in your attacks), `farm.c` (the source code of your target's "gadget farm", which you will use in generating return-oriented programming attacks), and `hex2raw` (a utility to generate attack strings). If for some reason you download multiple targets, this is not a problem. Choose one target to work on and delete the rest. The following assumes that you have copied the files to a protected local directory on the `access2.cims.nyu.edu` server, and that you are executing the programs in that local directory as it is necessary to use a machine that is similar to the one that generated your targets.

Target Programs Information:

Both `ctarget` and `rtarget` read strings from standard input. They do so with the function `getbuf` defined below:

```
unsigned getbuf()
{
   char buf[BUFFER_SIZE];
   Gets(buf);
   return 1;
}
```

The function `Gets` is similar to the standard library function `gets`, it reads a string from standard input (terminated by '`\n`' or end-of-file) and stores it (along with a null terminator) at the specified destination. In the code shown above, you can see that the destination is an array `buf`, declared as having `BUFFER_SIZE` bytes. At the time your targets were generated, `BUFFER_SIZE` was a compile-time constant specific to your version of the programs. Functions `Gets()` and `gets()` have no way to determine whether their destination buffers are large enough to store the string they read. They simply copy sequences of bytes, possibly overrunning the bounds of the storage allocated at the destinations. If the string typed by the user and read by `getbuf` is sufficiently short, it is clear that `getbuf` will return 1, as shown by the following execution examples:

```
access2> ./ctarget
```

```
Cookie: 0x1a7dd803
Type string: Keep it short!
No exploit. Getbuf returned 0x1 Normal return
```

Typically an error occurs if you type a long string:

```
access2> ./ctarget
Cookie: 0x1a7dd803
Type string: This is not a very interesting string, but it has the
property ...
Ouch!: You caused a segmentation fault!
Better luck next time
```

Note that the value of the cookie shown will differ from yours. Program `rtarget` will have the same behavior. As the error message indicates, overrunning the buffer typically causes the program state to be corrupted, leading to a memory access error. Your task is to be more clever with the strings you feed `ctarget` and `rtarget` so that they do more interesting things. These "smart" strings are called exploit strings.

Both `ctarget` and `rtarget` take several different command line arguments:

`-h`**:** Print list of possible command line arguments

`-q`**:** Don't send results to the Microsoft Azure server

`-i FILE`**:** Supply input from a file, rather than from standard input

Your exploit strings will typically contain byte values that do not correspond to the ASCII values for printing characters. The program `hex2raw` will enable you to generate these *raw* strings. See Section 2.3 (part A) for more information on how to use `hex2raw`.

Important points:

- Your exploit string must not contain byte value `0x0a` at any intermediate position, since this is the ASCII code for newline ('`\n`'). When `Gets` encounters this byte, it will assume you intended to terminate the string.
- `hex2raw` expects two-digit hex values separated by one or more white spaces. So if you want to create a byte with a hex value of 0, you need to write it as `00`. To create the word `0xdeadbeef` you should pass "`ef be ad de`" to `hex2raw` (please note the reversal required for little-endian byte ordering).

When you have correctly solved one of the levels, your target program will automatically send a notification to the Microsoft Azure server. For example:

```
access2> ./hex2raw < ctarget.l2.txt | ./ctarget
Cookie: 0x1a7dd803
Type string:Touch2!: You called touch2(0x1a7dd803) Valid solution
for level 2 with target ctarget
PASSED: Sent exploit string to server to be validated. NICE JOB!
```

You need to provide screenshots and explanations in your report as part of your solution to this problem. Clarifications and corrections may be posted via the course NYU Classes Google Group. The Microsoft Azure server will test your exploit string to make sure it really works, and it will update the attacklab scoreboard page indicating that your userid (listed by your target number for anonymity) has completed this phase. You can view the scoreboard at `http://40.124.53.8:15515/scoreboard`. This web page is updated continuously to show your progress. Different from the previous problem, there is no penalty for making mistakes in this problem, therefore feel free to fire away at `ctarget` and `rtarget` with any strings you like. You can work on your solution on any Linux machine, but in order to submit your solution, you will need to be running it on `access2.cims.nyu.edu`. Your final grade for this problem is a combination of the quality and correctness of the report you submit and the progress recorded via the self-grading scoreboard.

Figure 1 below summarizes the five phases to follow to solve this problem. As can be seen, the first three involve code-injection (CI) attacks on `ctarget`, while the last two involve return-oriented-programming (ROP) attacks on `rtarget`.

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

Figure 1: Summary of attack lab phases

2.1. Code Injection Attacks

For the first three phases, your exploit strings will attack `ctarget`. This program is set up in a way that the stack positions will be consistent from one run to the next and so that data on the stack can be treated as executable code. These features make the program vulnerable to attacks where the exploit strings contain the byte encodings of executable code.

2.1.1. Level 1

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

Function `getbuf` is called within `ctarget` by a function `test` having the following C code:

```
void test()
{
   int val;
   val = getbuf();
   printf("No exploit. Getbuf returned 0x%x\n", val);
```

```
}
```

When `getbuf` executes its return statement (line 5 of `getbuf`), the program ordinarily resumes execution within function `test` (at line 5 of this function). We want to change this behavior. Within the file `ctarget`, there is code for a function `touch1` having the following C representation:

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

Your task is to get `ctarget` to execute the code for `touch1` when `getbuf` executes its return statement, rather than returning to `test`. Note that your exploit string may also corrupt parts of the stack not directly related to this stage, but this will not cause a problem, since `touch1` causes the program to exit directly.

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 getbuf will transfer control to `touch1`.
- Be careful about byte ordering.
- You might want to use `gdb` to step the program through the last few instructions of `getbuf` to make sure it is doing the right thing.
- The placement of buf within the stack frame for getbuf depends on the value of 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.

2.1.2. Level 2

Phase 2 involves injecting a small amount of code as part of your exploit string. Within the file `ctarget` there is code for a function `touch2` having the following C representation:

```
void touch2(unsigned val)
```

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

Your task is to get `ctarget` to execute the code for `touch2` rather than returning to `test`. In this case, however, you must make it appear to `touch2` as if you have passed your cookie as its argument.

Some Advice:

- You will want to position a byte representation of the address of your injected code in such a way that the `ret` instruction at the end of the code for `getbuf` 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`.
- Do not attempt to use `jmp` or `call` instructions in your exploit code. The encodings of destination addresses for these instructions are difficult to formulate. Use `ret` instructions for all transfers of control, even when you are not returning from a call.
- See the discussion in Section 2.3 (part B) on how to use tools to generate the byte-level representations of instruction sequences.

2.1.3. Level 3

Phase 3 also involves a code injection attack but passing a string as argument. Within the file `ctarget` there is code for functions `hexmatch` and `touch3` having the following C representations:

```
/* Compare string to hex represention of unsigned value */
int hexmatch(unsigned val, char *sval)
{
   char cbuf[110];
   /* Make position of check string unpredictable */
   char *s = cbuf + random() % 100;
   sprintf(s, "%.8x", val);
   return strncmp(sval, s, 9) == 0;
}
```

```
void touch3(char *sval)
{
   vlevel = 3;    /* Part of validation protocol */
   if (hexmatch(cookie, sval)) {
      printf("Touch3!: You called touch3(\"%s\")\n", sval);
      validate(3);
   } else {
      printf("Misfire: You called touch3(\"%s\")\n", sval);
      fail(3);
   }
   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 (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 functions `hexmatch` and `strncmp` are called, they push data onto the stack, overwriting portions of memory that held the buffer used by getbuf. As a result, you will need to be careful where you place the string representation of your cookie.

2.2. Return-Oriented Programming

Performing code-injection attacks on program `rtarget` is much more difficult than it is for `ctarget`, because it uses two techniques to thwart such attacks:

- It uses randomization so that the stack positions differ from one run to another. This makes it impossible to determine where your injected code will be located.
- It marks the section of memory holding the stack as nonexecutable, so even if you could set the program counter to the start of your injected code, the program would fail 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 *gadget*. Figure 2 below 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.

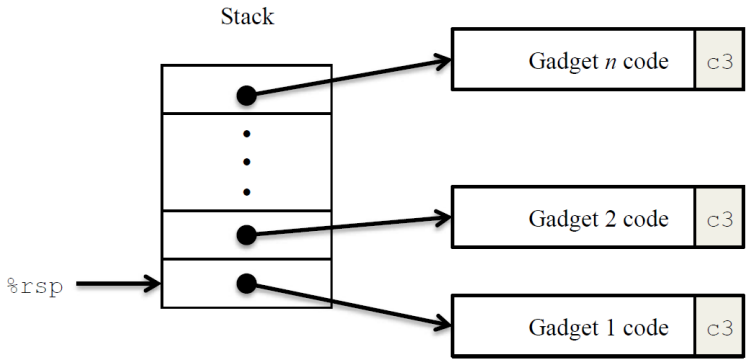

Figure 2: Setting up sequence of gadgets for execution (byte value `0xc3` encodes the `ret` instruction).

A gadget can make use of code corresponding to assembly-language statements generated by the compiler, especially ones at the ends of functions. In practice, there may be some useful gadgets of this form, but not enough to implement many important operations. For example, it is highly unlikely that a compiled function would have `popq %rdi` as its last instruction before `ret`. Fortunately, with a byte-oriented instruction set, such as x86-64, a gadget can often be found by extracting patterns from other parts of the instruction byte sequence.

For example, one version of `rtarget` contains code generated for the following C function:

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

The chances of this function being useful for attacking a system seem pretty slim. But, 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 byte sequence `48 89 c7` encodes the instruction `movq %rax, %rdi`. (See Figure 3A for the encodings of useful `movq` instructions.) This sequence is followed by byte value `c3`, which encodes the `ret` instruction. The function starts at address `0x400f15`, and the sequence starts on the fourth byte of the function. Thus, this code contains a gadget, having a starting address of `0x400f18`, that will copy the 64-bit value in register `%rax` to register `%rdi`.

Your code for `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.

Important: The gadget farm is demarcated by functions `start_farm` and `end_farm` in your copy of `rtarget`. Do not attempt to construct gadgets from other portions of the program code.

2.2.1. Level 2

For Phase 4, you will repeat the attack of Phase 2, but do so on program `rtarget` using gadgets from your gadget farm. You can construct your solution 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 Figure 3A.

`popq`: The codes for these are shown in Figure 3B.

`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 need can be found in the region of the code for `rtarget` demarcated by the functions `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.

2.2.2. Level 3

Before you take on the 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 this problem. That is a good score.

Phase 5 requires you to do an ROP attack on `RTARGET` to invoke function `touch3` with a pointer to a string representation of your cookie. That may not seem significantly more difficult than using an ROP attack to invoke `touch2`, except that we have made it so. Moreover, Phase 5 counts for only 5 points, which is not a true measure of the effort it will require. Think of it as more an extra credit problem for those who want to go beyond the normal expectations for the midterm examination.

To solve Phase 5, you can use gadgets in the region of the code in `rtarget` demarcated by functions `start_farm` and `end_farm`. In addition to the gadgets used in Phase 4, this expanded farm includes the encodings of different `movl` instructions, as shown in Figure 3C below. 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. These include instructions, shown in Figure 3D, such as `andb %al,%al`, that operate on the low-order bytes of some of the registers but do not change their values.

`movq` *S*, *D*

Source *S*	Destination *D*							
	%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

Figure 3A. Encodings of movq instructions
(all values are shown in hexadecimal)

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

Figure 3B. Encodings of popq instructions
(all values are shown in hexadecimal)

`movl` *S*, *D*

Source *S*	Destination *D*							
	%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

Figure 3C. Encodings of movl instructions
(all values are shown in hexadecimal)

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

Figure 3D. Encodings of 2-byte functional nop instructions
(all values are shown in hexadecimal)

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 course textbook.
- The official solution requires eight gadgets (not all of which are unique).

2.3. Appendix

A. Using Hex2raw

`Hex2raw` takes as input a *hex-formatted* string. In this format, each byte value is represented by two hex digits. For example, the string "`012345`" could be entered in hex format as "`30 31 32 33 34 35 00`." (Recall that the ASCII code for decimal digit x is `0x3x`, and that the end of a string is indicated by a null byte.)

The hex characters you pass to `hex2raw` should be separated by whitespace (blanks or newlines). We recommend separating different parts of your exploit string with newlines while you're working on it. `Hex2raw` supports C-style block comments, so you can mark off sections of your exploit string. For example:

```
48 c7 c1 f0 11 40 00 /* mov $0x40011f0,%rcx */
```

Be sure to leave space around both the starting and ending comment strings ("`/*`", "`*/`"), so that the comments will be properly ignored.

If you generate a hex-formatted exploit string in the file `exploit.txt`, you can apply the raw string to `ctarget` or `rtarget` in several different ways:

1. You can set up a series of pipes to pass the string through `hex2raw`:

```
access2> cat exploit.txt | ./hex2raw | ./ctarget
```

2. You can store the raw string in a file and use I/O redirection:

```
access2> ./hex2raw < exploit.txt > exploit-raw.txt
access2> ./ctarget < exploit-raw.txt
```

This approach can also be used when running from within `gdb`:

```
access2> gdb ctarget
(gdb) run < exploit-raw.txt
```

3. You can store the raw string in a file and provide the file name as a command-line argument:

```
access2> ./hex2raw < exploit.txt > exploit-raw.txt
access2> ./ctarget -i exploit-raw.txt
```

This approach also can be used when running from within `gdb`.

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-generated 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:

```
access2> gcc -c example.s
access2> 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       pushq $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 string can then be passed through `hex2raw` to generate an input string for the target programs. Alternatively, you can edit `example.d` to omit extraneous values and to contain C-style comments for readability, yielding:

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

This is also a valid input you can pass through `hex2raw` before sending to one of the target programs.

2.4. References

[1] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, lan- guages, 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.

VI. Deliverables

1. Report Submission Format:

   a) Save your work as a named archive as follows (lab-2 being a work directory that contains your file):

      a. `tar -cvf firstname_lastname_lab2.tar lab-2`

      Since source code is submitted, include your name as a comment at the top of each file (note: all files submitted should include your name).

   b) Provide details (in short report form) related to your completion of the two problems from part V.

      a. Save your report as a Word document.

      b. Name the file "**firstname_lastname_lab_2.docx**" (e.g., "john_doe_lab_2.docx").

2. Electronic Submission:

   Your report and lab assignment files must be submitted via NYU Brightspace (including

your report word document and your named archive). The files must be created and sent by the beginning of class. After the class period, the lab assignment is late. The email clock is the official clock. You may submit your work as often as you like until the due date of the lab assignment. At any point in time, your most recently uploaded files constitute your official submission. You may (re-)submit as often as you like.

Note: Pre-approved delayed submissions must be emailed directly to the professor. Please use the following naming convention in the subject line of the eMail for delayed submissions:

"**CC - firstname lastname - lab 2"**
(e.g.: "CC – John Doe - lab 2").

3. Cover page and other formatting requirements (to be included in the report document):

   The cover page supplied on the next page must be the first page of your report.

   Fill in the blank area for each field.

   NOTE:

   The sequence of the electronic submission is:

   1. **Cover sheet**
   2. **Report sheet(s)**

4. Grading guidelines (grade normalized to 100):

   Report Layout (5%)
   - o Report is neatly assembled on 8 1/2 by 11 layout.
   - o Report cover page with your name (last name first followed by a comma then first name), NYU ID, and section number with a signed statement of independent effort is included.
   - o File name is correct.

   Answers to Individual Questions (95%):
   - o Solutions to problems specified in part V are correct and your name is included at the beginning of each file.
   - o Assumptions provided as required.

   (100 points total, all questions weighted equally)

VIII. Sample Cover Sheet:

Name _____ Date: _____
(last name, first name)

NYU ID: _____

Section: _____

Lab 2

Total in points (100 points total): _____

Professor's Comments: