Assignment #4: Attack Lab (due on Sun, Mar. 17, 2019 at 11:59pm)

Contents

Introduction

This assignment asks you to run buffer overflow attacks using two strategies: (1) loading your binary code on the stack and starting its execution by overwriting the return address, or (2) a return-oriented attack, where return addresses are used to jump to one or more "gadgets" (short sequences of instructions ending with `ret`).

Through this assignment:

- You will get a better understanding of how to write programs that are more secure (i.e., using explicit checks on buffer sizes, or through features provided by compilers and operating systems).
- You will gain a deeper understanding of the stack and parameter-passing mechanisms of x86-64 machine code.
- You will 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 CS:APP3e book as reference material for this lab.

Instructions

A new directory will be created in your GitHub repository including the following files:

- `ctarget`: a program vulnerable to *code injection* attacks;
- `rtarget`: a program vulnerable to *return-oriented programming* attacks;
- `farm.c`: the source code of the "gadget farm" present inside `rtarget`;
- `check_grade.sh`: a script to check your current grade;
- `hex2raw`: a program to convert text files with hex sequences into their raw binary values (e.g., convert the text `48 65 6c 6c 6f` to the binary sequence encoding the ASCII string `Hello`).

Both `ctarget` and `rtarget` read a string from `stdin` and store it inside the buffer array `buf`. They do so using the vulnerable `getbuf` function:

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

The function `Gets` is similar to the standard library function `gets`: it reads bytes from `stdin` until it finds `\n` or `EOF` and stores them inside the input array `buf`, followed by a null terminator `\0`.

If the string is short, nothing interesting happens:

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

When the string typed by the user (or sourced from a text file with `ctarget < attack.raw` or `ctarget -i attack.raw`) is longer than the space allocated on the stack by the compiler, `Gets` will overwrite the return address of `getbuf`. Most likely, this will cause a segmentation fault:

```
$ ./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 magic cookie shown will differ from yours. This value is not given to you directly: you must inspect assembly to figure out what it is. Hint: Keep endianness in mind when injecting it!)

Your goal is to craft attack strings that trigger the execution of functions `touch1`/`touch2`/`touch3` inside `ctarget`, and `touch2`/`touch3` inside `rtarget`, by "properly" overwriting return addresses.

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.

- `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 4-byte word `0xdeadbeef` (an `int`) you should pass `ef be ad de` to HEX2RAW (note the reversal required for little-endian byte ordering).
- Your exploit string must not contain byte value `0x0a` at any intermediate position, since this is the ASCII code for newline `\n`. If `Gets` encounters this byte, it will assume you intended to terminate the string.

Evaluation

You must complete the assignment using the class VM. *Your virtual machine must be connected to the internet,* as the program will connect to our server when you complete an attack.

Every attempt you make will be logged by the automated grading server. As in the Bomb Lab, run `./check_grade.sh` to view your current progress. *Unlike the previous project, there is no penalty for making mistakes in this lab.* Attempts to break or overload the server, however, are not allowed, and will be considered cheating.

Your solutions may not use attacks to circumvent the validation code in the programs. Specifically, any address you incorporate into an attack string for use by a `ret` instruction should be to one of the following destinations:

- The addresses for functions `touch1`, `touch2`, or `touch3`.
- The address of your injected code.
- The address of one of your gadgets from the gadget farm.

You may only construct gadgets from file `rtarget` with addresses ranging between those for functions `start_farm` and `end_farm`.

There are 5 attacks to complete:

- `touch1` in `ctarget` (15 points).
- `touch2` in `ctarget` (35 points).
- `touch3` in `ctarget` (35 points).
- `touch2` in `rtarget` (10 points).
- `touch3` in `rtarget` (5 points).

The first three involve code-injection (CI) attacks on `ctarget`, while the last two involve return-oriented-programming (ROP) attacks on `rtarget`.

Logistics

This is an individual project. All handins are electronic and automatically logged by the software. Clarifications and corrections will be posted on the course Piazza.

Handout Instructions

There is no explicit handin. The target functions will notify our server automatically each time you complete an attack.

We have included the ability for you to check your score and last completed attack through the script `check_grade.sh` provided in your repository.

Since our server logs your successful attack completions, we will handle grace day usage by just looking at the timestamps. You do not need to do anything special to use grace days. However, don't attempt to complete any more attacks after the due date as we will count those attempts as a grace day usage.

Attack Instructions: Code Injection

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.

Level 1: `touch1` in `ctarget` (15 points)

In the first attack, 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 the function `test` having the following C code:

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

When `getbuf` executes its `return` statement, the program ordinarily resumes execution within function `test` (with a call to `validate`). 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, 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 (`layout asm` inside `gdb` also works).
- 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 (Intel CPUs are little-endian).
- 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.*

Level 2: `touch2` in `ctarget` (35 points)

Level 2 involves injecting a small amount of code as part of your exploit string (see the section Generating Binary Instructions on how to generate the code to inject). Within the file `ctarget` there is code for a function `touch2` having the following C representation:

```
void touch2(unsigned val, unsigned val2) {
  printf("%d\n", last_five);
  printf("%d\n", user_id);
  vlevel = 2;  /* Part of validation protocol */

  if (val == last_five && val2 == magic) {
    printf("Touch2!: You called touch2(%d, 0x%.8x)\n", val, val2);
    validate(2, 1);
  } else {
    printf("Misfire: You called touch2(%d, 0x%.8x)\n", val, val2);
    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:

- The last five digits of your USC ID as the first argument;
- Your magic cookie number as the second 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 and second arguments to a function are passed in registers `%rdi` and `rsi`.
- Your injected code should set these registers to your USC ID and magic cookie number, 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.
- See the discussion at the end of this page on how to use tools to generate the byte-level representations of instructions.

Level 3: `touch3` in `ctarget` (35 points)

Level 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() % 80;
  sprintf(s, "%.8x", val);
  return strncmp(sval, s, 9) == 0;
}

void touch3(char *sval, char *sval2) {
  char idStr[6];
  sprintf(idStr, "%d", last_five);
  vlevel = 3; /* Part of validation protocol */

  if (hexmatch(magic, sval) && (strcmp(idStr, sval2) == 0 )) {
    printf("Touch3!: You called touch3(\"%s\", \"%s\")\n", sval, sval2);
    validate(3, 1);
  } else {
    printf("Misfire: You called touch3(\"%s\", \"%s\")\n", sval, sval2);
    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 *magic value as its first argument*, and a string representation of the *last five digits of your USC ID as the second argument.*

Important note: You must drop all leading `0`'s from your USC ID for Level 3. So if your USC ID ends with `01234`, you should only pass `1234`. If your USC ID ends with `00001`, you should only pass in `1`. If your USC ID ends with `10000`, you should pass in `10000`, as you only drop the leading `0`'s.

Some advice:

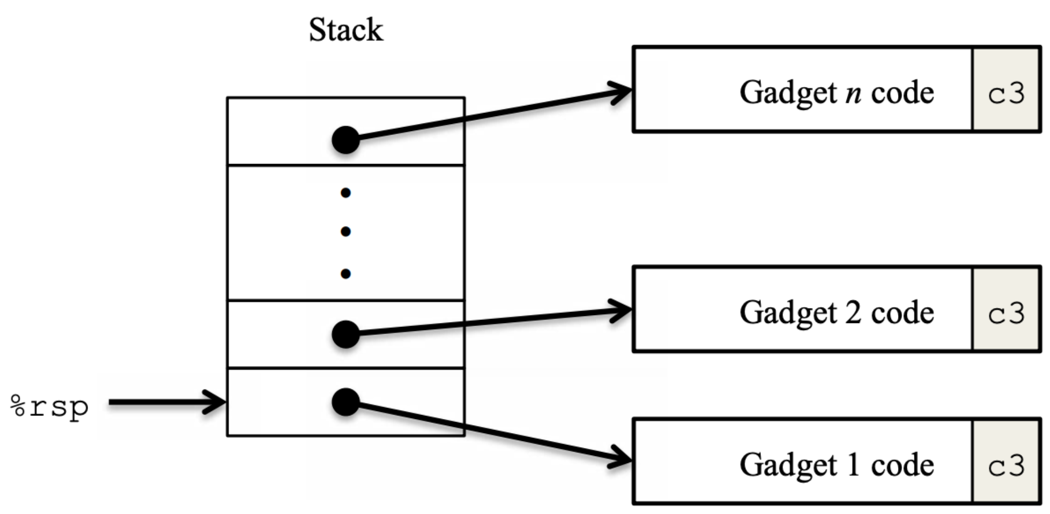
- You will need to include a string representation of your cookie in your exploit string. The string should consist of the eight hexadecimal digits (ordered from most to least significant) *without* a leading `0x` and *lowercase* (e.g., if your cookie value is `0x1A7DD803` in hexadecimal, the string should be "`1a7dd803`").
- 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 representation of your magic number, and `%rsi` to the address of the string representation of the last five digits of your USC ID.
- 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 about the placement of the string representations of your magic cookie and USC ID digits.

Attack Instructions: 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). The strategy of 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 called a *gadget*. The following figure illustrates how the stack can be set up to execute a sequence of n gadgets.



- 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. 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` (at the end of the binary encoding of `movl $0xc78948d4,(%rdi)`) encodes the instruction `movq %rax,%rdi`.

This sequence is followed by the 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 Levels 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.

Level 4: `touch2` in `rtarget` (10 points)

For Level 4, you will repeat a simplified version of Level 2's `touch2` in the program `rtarget` using gadgets from your gadget farm. `touch2` now only takes one argument as seen in the code below:

```
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);
}
```

You can construct your solution using gadgets consisting of the following instruction types, and using only the first eight x86-64 registers (`%rax` through `%rdi`).

- `movq`: The codes for these are shown below.

`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

- `popq`: The codes for these are shown below.

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

- `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 complete this attack with just two gadgets.*

Level 5: `touch3` in `rtarget` (5 points)

Before you take on the Level 5, pause to consider what you have accomplished so far! In Levels 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 Level 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 also collected 95/100 points for the lab. That’s a good score. *If you have other pressing obligations consider stopping right now.*

Level 5 requires you to do an ROP attack on `rtarget` to invoke a simplified version of the function `touch3` with a pointer to a string representation of your magic cookie number. It is shown below.

```
/* 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);
}
```

That may not seem significantly more difficult than using an ROP attack to invoke `touch2`, except that we have made it so through address space randomization. Moreover, Level 5 counts for only 5 points, which is not a true measure of the effort it will require. Think of it as an extra credit problem for those who want to go beyond the normal expectations for the course.

- To solve Level 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 Level 4, this expanded farm includes encodings of `movl` instructions shown below.

`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

- 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 instructions, shown below, operate on the low-order bytes of some of the registers but do not change their values.

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

Some advice:

- You'll want to review the effect of `movl` on the upper 4 bytes of a register (page 183 of the textbook).
- The official solution requires eight gadgets (not all of which are unique).
- Remember address space randomization means you probably can't hard-code a pointer.
- There are some gadgets that might be useful to you as is (i.e., no hidden instructions... just call that function).

Using `hex2raw`

The program `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 attack string with newlines while you're working on it. `hex2raw` supports C-style block comments, so you can mark off sections of your attack 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 delimiters (`/*` and `*/`), so that the comments are ignored.

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

- You can set up a series of pipes to pass the string through `hex2raw`:
  ```
  $ cat attack.txt | ./hex2raw | ./ctarget
  ```
- You can store the raw string in a file and use I/O redirection:
  ```
  $ ./hex2raw < attack.txt > attack.raw
  $ ./ctarget < attack.raw
  ```
  This approach can also be used when running from within GDB:
  ```
  $ gdb ctarget
  (gdb) run < attack.raw
  ```
- You can store the raw string in a file and provide the file name as a command-line argument:
  ```
  $ ./hex2raw < attack.txt > attack.raw
  $ ./ctarget -i attack.raw
  ```
  This approach also can also be used when running from within GDB (`run -i attack.raw`).

Generating Binary Instructions

For code-injection attacks, you need to save binary instructions on the stack. But how can you figure out the binary encoding of your attack instructions?

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:

```
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:

```
$ objdump -d example.o
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.

Acknowledgements. This lab was developed by the authors of the course textbook and their staff. It has been customized for use by this course.

CS356