CS105 – Computer Systems Fall 2024

Assignment 4: Attack Lab

Due: October 1, 2024 at 11:59pm

In this lab, you will gain firsthand experience with methods used to exploit security weaknesses in operating systems and network servers. You will do this by generating a series of four attacks on two programs with different security vulnerabilities. The purpose is to help you learn about the runtime operation of programs—in particular about function calls, parameter passing, and the stack—and to understand the nature of these security weaknesses so that you can avoid them when you write system code. *I do not condone the use of any other form of attack to gain unauthorized access to any system.*

Logistics

As usual, this is a pair project. But this week you may choose who to work with. Each pair will be generating attacks for target programs that are custom generated for you.

Getting Files

You can obtain your files by directing your browser to

`http://itbdcv-lnx04p.campus.pomona.edu:1052`

The server will build your files and return them to your browser in a `tar` file called `target`*k*`.tar`, where *k* is the unique number of your target programs. It takes a few seconds to build and download your target, so please be patient.

You should only download one set of files. If for some reason you download multiple targets, choose one target to work on and delete the rest.

If the server is not working, wait a couple minutes and then try again. If you are completely unable to download starter files, let me know.

Copy the `target`*k*`.tar` file *from your local machine* to a (protected) Linux directory on the server, e.g.,

```
% scp ~/Downloads/target47.tar user1234@itbdcv-lnx04p.campus.pomona.edu:~
```

Except using your target number and username.

Then connect to the server over `ssh` and extract the files using the following command with your number:

```
% tar -xvf target47.tar
```

This will extract a directory containing the following files:

- `README.txt`: A file describing the contents of the directory
- `ctarget`: An executable program vulnerable to *code-injection* attacks (Phases 1-3)
- `rtarget`: An executable program vulnerable to *return-oriented-programming* attacks (Phase 4)
- `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. **IMPORTANT: if you ever decide to compile this file from scratch, make sure you are using the optimization flag `-Og`!**
- `hex2raw`: A utility to generate attack strings.

Rules for the Assignment

Here are some rules regarding valid solutions for this lab. They may not make much sense when you read this document for the first time. They are presented here as a central reference of rules once you get started.

- You must do the assignment on the course VM (`itbdcv-lnx04p.campus.pomona.edu`).
- 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`.

Target Programs

There are two programs you will be attacking in this assignment: CTARGET and RTARGET. Both programs read strings from standard input using the helper 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 this code, you can see that the destination is an array `buf`, declared as having `BUFFER_SIZE` bytes. When your target was generated, `BUFFER_SIZE` was a compile-time constant specific to your 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:

```
% ./ctarget
Cookie: 0x59b997fa
Type string: short string
No exploit.  Getbuf returned 0x1
Normal return
```

Typically an error occurs if you type a long string:

```
% ./ctarget
Cookie: 0x59b997fa
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 and user id shown will differ from yours.) Program RTARGET will have the same behavior. As the error message indicates, overrunning the buffer typically causes the program to attempt to jump to an invalid address, 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 are called *exploit* strings.

Important Details

- 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.
- Your exploit strings will typically contain byte values that do not correspond to the ASCII values for typeable characters. In class demos, I've solved this problem by editing the raw binary. But I'm not going to make you do that. Instead, I've provided a program HEX2RAW that will take a string that *looks* like a sequence of hex bytes, e.g.: `47 47 47 47 47 47 47 47 a4 f2 01 0c 3c 5f 54 2a` and will generate the corresponding sequence of *actual binary bytes*.
  There are lots of ways to use the program HEX2RAW (see Appendix A), but there are two that I recommend. If you think you have the right answer and want to test it out quickly, you can save your sequence of byte-like characters in a textfile called `phase1.txt`, redirect the program HEX2RAW to read from that file, and then pipe the output directly into the program you are attacking. For example,
  ```
  % ./hex2raw < phase1.txt | ./ctarget
  ```
  Note that you can include both linebreaks and comments in the form `/* comment */` in your textfile; I recommend using both to make your file more readable for you.
  If you want to test a string multiple times, you can save the output from HEX2RAW as a raw binary file and then re-use it as often as you want, using redirects to read and write from files instead of reading from the keyboard and printing to the screen. For example,
  ```
  % ./hex2raw < phase1.txt > phase1.bin
  % ./ctarget < phase1.bin
  ```
  Once you have the raw sequence of binary bytes saved as a file, you can also use that while running your program with `gdb`:
  ```
  % gdb ctarget
  (gdb) r < phase1.bin
  ```
- Remember to work on the VM! This code generally won't run correctly on other machines.

The following table summarizes the phases of the lab. As can be seen, the first three involve code-injection (CI) attacks on CTARGET, while the last one requires return-oriented-programming (ROP) to circumvent the additional defenses on RTARGET.

Phase	Program	Level	Method	Function	Points
1	CTARGET	1	CI	`touch1`	10
2	CTARGET	2	CI	`touch2`	20
3	CTARGET	3	CI	`touch3`	20
4	RTARGET	2	ROP	`touch2`	20

CI:	Code injection
ROP:	Return-oriented programming

Part I: 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.

Phase 1

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

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

```
1 void test()
2 {
3     int val;
4     val = getbuf();
5     printf("No exploit.  Getbuf returned 0x%x\n", val);
6 }
```

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 phase can be determined by examining a disassembled version of CTARGET. You can use `objdump -d` to get the dissembled version of the full program (along with the assembled binary encoding of each instruction), but I prefer to just disassemble individual functions in gdb.
- Be careful about byte ordering. Machines are little endian!
- You might find it helpful 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.

Phase 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 (an unsigned integer value) as its argument.

Some Advice

- Recall that the first argument to a function is passed in register `%rdi`.
- Your injected code will need to set up the argument to `touch2` before jumping to the start of `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.
- To solve this phase, you will need to know the binary encoding of the sequence of instructions you want to inject. I recommend saving the sequence of assembly instructions to a file `assembly.s`, e.g.

  ```
  addq %rax, %rbx
  ret
  ```

  and then using the `gcc` compiler to compile your hand-written assembly instructions into a binary file:

  ```
  % gcc -c assembly.s
  ```

  You can then use `objdump` to generate a file that displays the binary encodings in a readable way:

  ```
  % objdump -d assembly.o > assembly.d
  ```

  and read `assembly.d` with your preferred text editor.
  Appendix B has additional detailed information on how to generate the byte-level representations of instructions.

Phase 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. For example, if your cookie is `0x59b997fa` you should call `touch3` with the argument `"59b997fa"`.

Some Advice

- You will need to include a string representation of your cookie in your exploit string. The string should consist of the eight ascii characters corresponding to the eight hex digits of your cookie (the hex 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. Or I prefer to use `https://www.asciitable.com`
- Recall that strings are really type `char*` (aka a pointer). Your injected code should set register `%rdi` to the address of the first character of your 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.

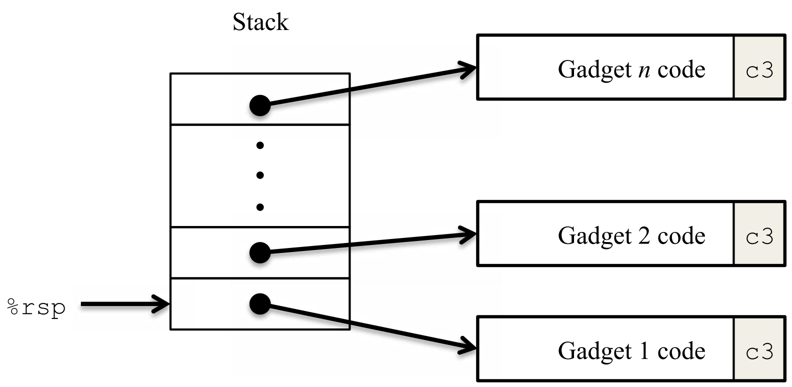

Figure 1: Setting up sequence of gadgets for execution. Byte value `0xc3` encodes the `ret` instruction.

Part II: 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, you can use *return-oriented programming* (ROP). Recall from class that 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. 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 2A 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.

Phase 4

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 2a.
- `popq`**:** The codes for these are shown in Figure 2b.
- `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.

`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

(a) Encodings of `movq` instructions

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

(b) Encodings of `popq` instructions

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

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`.
- At this point, `objdump -d rtarget` may be useful. The printout might be long, so I recommend redirecting it to a file (e.g., `objdump -d rtarget > target.d`).
- Pop instructions are useful :-)

Part III: Feedback

Please upload to submit.cs a file called `feedback.txt` that answers the following questions:

1. How long did each of you spend on this assignment?
2. Any comments on this assignment?

As always, how you answer these questions **will not affect your grade**, but whether you answer them will.

Submission

For this assignment, you should submit eight (8) files: textfiles named `phase1.txt`, `phase2.txt`, `phase3.txt`, and `phase4.txt` that contain your solutions to the four phases (the textfiles that you input to `hex2raw`, not the binary ones!), along with your targets `ctarget` and `rtarget`, your `feedback.txt`, and a file named `targetid.txt` that contains the id of the target you were working on (e.g., `target1`). Submit these eight files as one submission on Gradescope.

Good luck and have fun!

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 `0x3`x, 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 are 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.

   ```
   % cat exploit.txt | ./hex2raw | ./ctarget -q
   ```

2. You can use an I/O redirect to direct the text file into HEX2RAW and then pipe the result into the target

   ```
   % ./hex2raw < exploit.txt | ./ctarget -q
   ```

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

   ```
   % ./hex2raw < exploit.txt > exploit-raw.txt
   % ./ctarget -q < exploit-raw.txt
   ```

   This approach can also be used when running from within GDB:

   ```
   % gdb ctarget
   (gdb) run -q < exploit-raw.txt
   ```

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

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

   This approach also can be used when running from within GDB.

Appendix 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:

```
% 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        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.