Project Buffer Overflow

Due: October 8, 2014

1 Introduction

You have just stolen RoboLabs, Inc.'s latest invention: the Ultimate RoboBaby™. Unfortunately, after getting it back to your hideout high in the mountains, you realize that all of its awesome capabilities are locked! Fortunately, you know x86-64 assembly and architecture, and you can use this knowledge to unlock the power of the Ultimate RoboBaby™!

2 Assignment

This assignment will help you develop a detailed understanding of x86-64 calling conventions and stack organization. It involves applying a series of *buffer overflow attacks* on an executable file called `buffer`.

In this project, you will gain firsthand experience with one of the methods commonly 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 this form of security weakness so that you can avoid it when you write system code. We do not condone the use of this or any other form of attack to gain unauthorized access to any system resources.

This assignment contains the following files:

- *buffer*: The buffer program you will attack.
- *makecookie*: Generates a "cookie" based on your userid.
- *hex2raw*: A utility to help convert between string formats.
- *buffer.pdf*: This document.

Your task during this assignment is to use buffer overflow attacks to cause the `buffer` program to behave in unexpected ways.

3 Userids and Cookies

Phases of this project will require a slightly different solution from each student. The correct solution will be based on your userid.

A *cookie* or *hash* is a string of eight hexadecimal digits generated from your userid in such a way that distinct userids will (with high probability) produce distinct cookies. You can generate your cookie with the `makecookie` program giving your userid as the argument. For example:

```
./makecookie jcarberr
0x44e22c05
```

In two of your three buffer attacks, your objective will be to make your cookie show up in places where it ordinarily would not. In three of those four attacks, you will accomplish this by supplying machine code instructions to the `buffer` program.

A problem with doing so is that Linux does not allow data on the program stack to be executed as machine instructions. However, your accomplice inside RoboLabs, Inc. anticipated this safeguard, and tricked the programmers that wrote Ultimate RoboBaby^{TM}'s software into moving the stack to a different, executable memory location. This means that the instructions that you place on the stack can indeed be executed.

4 The `buffer` Program

The `buffer` program reads a string from standard input. It does so with the function `getbuf` defined below:

```
/* Buffer size for getbuf */
#define NORMAL_BUFFER_SIZE 32

int getbuf() {
     char buf[NORMAL_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` having sufficient space for 32 characters.

`Gets()` (and `gets()`) grabs a string off the input stream and stores it into its destination address (in this case `buf`). However, `Gets()` has no way of determining whether `buf` is large enough to store the whole input. It simply copies the entire input string, possibly overrunning the bounds of the storage allocated at the destination.

If the string typed by the user to `getbuf()` is no more than 31 characters long, `getbuf()` will correctly return 1, as shown by the following execution example:

```
./buffer -u jcarberr
Type string: I love CS 33.
Oops: getbuf returned 0x1
```

Typically an error occurs if a longer string is entered:

```
./buffer -u jcarberr
Type string: It is easier to love this class when you are a TA.
Ouch!: You caused a segmentation fault!
```

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 `buffer` so that it does more interesting things. These are called *exploit* strings.

`buffer` takes several different command line arguments:

`-u userid:` Operate the program for the indicated userid (required argument).

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

`-n:` Operate in "Nitro" mode, as is used in Level 4 below.

At this point, you should think about the x86-64 stack structure a bit and figure out what entries of the stack you will be targeting. You may also want to think about *exactly* why the last example created a segmentation fault, although this is less clear. Be aware that the buffer starts at the top of the stack (the lowest memory address in the stack) and grows towards the bottom (towards higher addresses).

Your exploit strings will typically contain byte values that do not correspond to the ASCII values for printing characters. The program `hex2raw` can help you generate these *raw* strings. It 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`." (Recall that the ASCII code for decimal digit x is `0x3`x.)

The hex characters you pass `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` also supports C-style block comments, so you can mark off sections of your exploit string. For example:

```
bf 66 7b 32 78 /* mov    $0x78327b66,%edi */
```

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

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

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

   ```
   cat exploit.txt | ./hex2raw | ./buffer -u jcarberr
   ```

2. You can store the raw string in a file and use I/O redirection to supply it to `buffer`:

   ```
   ./hex2raw < exploit.txt > exploit-raw.txt
   ./buffer -u jcarberr < exploit-raw.txt
   ```

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

   ```
   gdb buffer
   (gdb) run -u jcarberr < exploit-raw.txt
   ```

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 a whitespace. So if you want to create a byte with a hex value of 0, you need to specify 00. To create the word `0xDEADBEEF` you should pass DE AD BE EF to `hex2raw`.
- The CS department machines are *little-endian*, which means that the least-significant byte of a word is read first. This means that you should enter addresses into your hex string in reverse order, e.g. `17 42 04 08` for address `0x08044217`.
- It is not acceptable to jump directly to the validate() function calls. The goal of this assignment is to understand the x86-64 program stack by manipulating and exploiting it - jumping directly to validate() in each phase circumvents the need for any such understanding, and you will not receive credit for solutions that do this.
- Your exploit strings should not cause segmentation faults. The following output of the buffer program is not valid:

```
Userid: jcarberr
Cookie: 0x12345678
Type string: <string>
BEEP BEEP!: getbuf returned 0x12345678
VALID
NICE JOB!
Ouch!: You caused a segmentation fault!
Better luck next time
```

  More generally, it is not sufficient to receive the VALID NICE JOB! confirmation. If your goal is to get the program to set or return a particular value, it must actually do this!

5 Phases

This project consists of three phases of buffer overflow attacks. The manner of attack will be slightly different in each phase.

5.1 Level 1: Talking (10 pts)

The function `getbuf()` is called within `buffer` by a function `test()` having the following C code:

```
void test() {
     int val;
     /* Put canary on stack to detect possible corruption */
     volatile int local = uniqueval();

     val = getbuf();

     /* Check for corrupted stack */
     if (local != uniqueval()) {
          printf("Sabotaged!: the stack has been corrupted\n");
     }
     else if (val == cookie) {
          printf("BEEP BEEP!: getbuf returned 0x%x\n", val);
          validate(4);
     } else {
          printf("Oops: 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 7 of that function). We want to change this behavior.

Within the file *buffer*, there is a function `talking()` having the following C code:

```
void talking()
{
  printf("\"Mama?\": You called talking!\n");
  validate(1);
  exit(0);
}
```

Your task is to get `buffer` to execute the code for `talking()` 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 `talking()` 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 `buffer`. `objdump -d buffer > obj.txt` will disassemble the contents of the buffer executable to `obj.txt`. This file will then contain each function's name, with all of its instructions and the addresses of those instructions.
- 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 which version of `gcc` was used to compile `buffer`, so you will have to read some assembly to figure out its true location.

- `si` is a gdb command which you can use to step over a single x86-64 instruction. `step` and `next` won't do this sometimes, but `si` never fails to do so. Use this command well.

5.2 Level 2: Walking (10 pts)

Now that your Ultimate RoboBaby™ can talk, you need to calibrate its movement servos so that it can get to your enemies! In order to be sure that it can walk, you need to provide four balance values that sum to 32, to ensure that, no matter the terrain, your Ultimate RoboBaby™ can make it across.

Within the file *buffer* there is a function `walking()` having the following C code:

```
struct calibration {
  int baby_id;
  int balances[4];
};

void walking(struct calibration cal)
{
  int balance = sum_balances(cal.balances);
  if (cal.baby_id == cookie && balance == 32) {
    printf("SUCCESS!: You called walking({0x%x, %d})\n",
           cal.baby_id, balance);
    validate(2);
  } else {
    printf("Baby stumbled and fell: You called walking({0x%x, %d})\n",
           cal.baby_id, balance);
  }
  exit(0);
}
```

Similar to Level 1, your task is to get `buffer` to execute the code for `walking()` rather than returning to `test()`. In this example, though, you must make it look like your data is the argument to `walking()`. How might this work?

Note that the program won't really call `walking()`—it will simply execute its code. This has important implications for where on the stack you want to place your cookie and other data.

5.3 Level 3: Flying (20 pts)

Now it is time for the final basic functionality of your Ultimate RoboBaby™: flight! However, flight is very tricky; you must be sure that flight can start and stop without causing any crashes or premature exits, otherwise your baby will fall out of the sky!

Our preceding attacks have all caused the program to jump to the code for some other function, which then causes the program to exit. As a result, it was acceptable to use exploit strings that corrupt the stack.

The most sophisticated form of buffer overflow attack causes the program to execute some exploit code that changes the program's register/memory state, but makes the program return to the original calling function (`test()` in this case). The calling function is oblivious to the attack. This style of attack is tricky, though, since you must: 1) get machine code onto the stack, 2) set the return pointer to the start of this code, and 3) undo any corruptions made to the stack state.

Your job for this level is to supply an exploit string that will cause `getbuf()` to return your cookie back to `test()`, rather than the value 1. You can see in the code for `test()` that this will cause the program to go `"BEEP BEEP!"`. Your exploit code should set your cookie as the return value, restore any corrupted state, push the correct return location on the stack, and execute a `ret` instruction to really return to `test()`.

Some advice:

- You can use `gdb` to get the information you need to construct your exploit string. Set a breakpoint within `getbuf()` and run to this breakpoint. Determine parameters such as the saved return address.
- Determining the byte encoding of instruction sequences by hand is tedious and prone to errors. You can let tools do all of the work by writing an assembly code file containing the instructions and data you want to put on the stack. Assemble this file with `gcc` and disassemble it with `objdump`. You should be able to get the exact byte sequence that you will type at the prompt. An example of doing this is contained in section 6.
- Keep in mind that your exploit string depends on your machine, your compiler, and even your userid's cookie. Do all of your work on the machines assigned by your instructor, and make sure you include the proper userid on the command line to `buffer`.

5.4 Level 4: Fire-Breathing (Extra credit!)

Once you've completed each of the other level, you've finished the project! Congratulations! This level is extra credit. You are welcome and encouraged to complete it, but you need not do so if you don't want to.

For this phase you'll need to run the `buffer` program in "nitro mode" by using the `-n` command-line flag.

From one run to another, especially by different users, the exact stack positions used by a given procedure will vary. One reason for this variation is that the values of all environment variables are placed near the base of the stack when a program starts executing. Environment variables are stored as strings, requiring different amounts of storage depending on their values. Thus, the stack space allocated for a given user depends on the settings of his or her environment variables. Stack positions also differ when running a program under `gdb`, since `gdb` uses stack space for some of its own state.

In the code that calls `getbuf()`, we have incorporated features that stabilize the stack, so that the position of `getbuf()`'s stack frame will be consistent between runs. This made it possible for you to write an exploit string knowing the exact starting address of `buf`. If you tried to use such an exploit on a normal program, you would find that it works some times, but it causes segmentation faults at other times.

For this level, we have gone the opposite direction, making the stack positions even less stable than they normally are. This extra difficulty was added to protect people from the responsibility that comes with fire-breathing.

When you run `buffer` with the command line flag "`-n`," it will run in "Nitro" mode. Rather than calling the function `getbuf()`, the program calls a slightly different function `getbufn()`:

```
int getbufn() {
     char buf[FIRE_BUFFER_SIZE];
     Gets(buf);
     return 1;
}
```

This function is similar to `getbuf()`, except that it has a buffer of 512 characters. You will need this additional space to create a reliable exploit. The code that calls `testn()` (which calls `getbufn()`) first allocates a random amount of storage on the stack, such that if you sample the value of `%rsp` during two successive executions of `testn()` or `getbufn()`, you would find they differ by as much as ±240. As a result, the addresses you used to solve previous phases may not work in this phase.

In addition, when run in Nitro mode, `buffer` requires you to supply your string 5 times, and it will execute `getbufn()` 5 times, each with a different stack offset. Your exploit string must make it return your cookie each of these times.

Your task is identical to the task for the Flying level. Once again, your job for this level is to supply an exploit string that will cause `getbufn()` to return your cookie back to test, rather than the value 1. You can see in the code for test that this will cause the program to go "`HOT HOT!`." Your exploit code should set your cookie as the return value, restore any corrupted state, push the correct return location on the stack, and execute a `ret` instruction to really return to `testn()`.

Some Advice:

- You can use the program `hex2raw` to send multiple copies of your exploit string by providing it with the command-line argument `-n`. If you have a single copy in the file *exploit.txt*, then you can use the following command:

  ```
  cat exploit.txt | ./hex2raw -n | ./buffer -n -u wario
  ```

  You must use the same string for all 5 executions of `getbufn()`.

- The trick for this phase is to make use of the `nop` instruction. It is encoded with a single byte (code `0x90`). By including a sequence of `nop` instructions before your exploit code, if the program jumps to any point in the sequence, it will "slide" along until it reaches the exploit code. Such a sequence of `nop`s is known as a *nop sled*. More information about `nop` sleds can be found on page 262 of the CS:APP textbook.

6 Generating Machine Code

Using `gcc` as an assembler and `objdump` as a disassembler makes it convenient to generate the bytes for instruction sequences. For example, suppose we write a file *example.s* containing the following assembly code:

```
# Example of hand-generated assembly code

     push $0xabcdef          # Push value onto stack
     add   $17,%eax          # Add 17 to %eax
     .align 4                # Following will be aligned on multiple of 4
     .long   0xfedcba98      # A 4-byte constant
```

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

We can now assemble and disassemble this file:

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

The generated file *example.d* contains the following lines

```
0:        68 ef cd ab 00              push   $0xabcdef
5:        83 c0 11                    add    $0x11,%eax
8:        98                          cwtl
9:        ba                          .byte 0xba
a:        dc fe                       fdivr  %st,%st(6)
```

Each line shows a single instruction. The number on the left indicates the 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`.

Starting at address 8, the disassembler gets confused. It tries to interpret the bytes in the file *example.o* as instructions, but these bytes actually correspond to data. Note, however, that if we read off the 4 bytes starting at address 8 we get: `98 ba dc fe`. This is a byte-reversed version of the data word `0xFEDCBA98`. This byte reversal represents the proper way to supply the bytes as a string, since a little endian machine lists the least significant byte first.

Finally, we can read off the byte sequence for our code as:

```
68 ef cd ab 00 83 c0 11 98 ba dc fe
```

This string can then be passed through `hex2raw` to generate a proper input string we can give to `buffer`. Alternatively, we can edit *example.d* to look like this:

```
68 ef cd ab 00 /* push   $0xabcdef */
83 c0 11 /* add    $0x11,%eax */
98
ba dc fe
```

which is also a valid input we can pass through `hex2raw` before sending it to `buffer`.

You can check the output of the `hex2raw` program by running its output through the `hexdump` program. This is a built-in Linux utility that outputs a human-readable hexadecimal representation of a file. For more information, run `man hexdump`.

7 gdb

Here are some `gdb` commands that you may find helpful for this assignment:

- `x/i $pc` prints the current instruction.
- `disassemble <function>` prints each instruction (and its address) of `<function>`.
- `info r` prints the value contained in each register.
- `x/48b` prints 48 bytes of memory. `x/48b <address>` prints the 48 bytes of memory after `<address>`.

For more `gdb` commands, consult the `gdb` Cheatsheet (available on the website and the assignment stencil).

Set breakpoints frequently and use these commands if you get stuck. However, *do not set a breakpoint on instructions which you have placed on the stack*. Doing so may cause a null byte, `0x00`, to replace one of your instruction bytes, which will very likely ruin that instruction (and the ones following it).

8 README

You should have a detailed explanation of each of your exploits in one of two places. Your first option is to write a README file (which is required to hand in anyway) in which you explain how each of your exploits works. Your second option is to use block comments (i.e. `/*...*/`) within each of your exploit text files to explain the exploit, and then hand in a README saying that you have done so.

9 Grading

You will receive points for successfully exploiting each level.

- Level 1: Talking is worth 10 points.
- Level 2: Walking is worth 10 points.
- Level 3: Flying is worth 20 points.
- Level 4: Fire-breathing is worth extra credit!

This assignment is worth 40 points total.

10 Handing In

Your handin for this assignment should include one file for each phase containing your input string for that phase, plus a README:

- *talking.txt*
- *walking.txt*
- *flying.txt*
- *fire-breathing.txt*, if you have completed this level.
- *README*

That is, the command

```
cat <phase>.txt | ./hex2raw | ./buffer -u <your login>
```

should solve the indicated phase. Be sure to create these files after you solve each phase so that you don't have to re-solve any of them.

To hand in your solutions, run

```
cs033_handin buffer
```

from your project working directory.

If you wish to change your handin, you can do so by re-running the handin script. Only your most recent handin will be graded.