Notes from Exploring Stack Overflows

Instructions

Make a copy of this document, rename it to “exploring-stack-overflow-notes” and move it to your CSE 523 Google Docs collection. If at any point in this exercise you feel stuck, raise your hand and get some guidance. When you reach each GATE below, switch over to the Tracking Progress document and update your position. Try to be efficient with your time.

Overview

Today we will explore stack buffer overflows. Keep detailed notes below (place your comments in between the provided horizontal lines); you will be referring to these in the future to do your work. Note that the material below is largely inspired by content from the book for the course (Hacking: the Art of Exploitation, by Jon Erickson).

We will be working in your CSE 523 Ubuntu virtual machine, so start that now and open a terminal window.

GATE 1

Make a folder called “stack_overflow” and enter the new directory. Using nano or the text editor of your choice, create a file stack_overflow.c and fill it with the following:

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {

  int value = 5;
  char buffer_one[8], buffer_two[8];

  strcpy(buffer_one, "one");
  strcpy(buffer_two, "two");

  printf("[BEFORE] buffer_two is at %p and contains \'%s\'\n",
buffer_two, buffer_two);
  printf("[BEFORE] buffer_one is at %p and contains \'%s\'\n",
buffer_one, buffer_one);
```

```
  printf("[BEFORE] value is at %p and is %d (0x%08x)\n\n", &value,
value, value);

  printf("[STRCPY] copying %d bytes into buffer_two\n\n",
strlen(argv[1]));
  strcpy(buffer_two, argv[1]);

  printf("[AFTER] buffer_two is at %p and contains \'%s\'\n",
buffer_two, buffer_two);
  printf("[AFTER] buffer_one is at %p and contains \'%s\'\n",
buffer_one, buffer_one);
  printf("[AFTER] value is at %p and is %d (0x%08x)\n", &value, value,
value);

}
```

Take a moment to read through the code and anticipate what it will do. Next, compile the C file with debug options enabled. If you get a warning about expecting type 'int' but getting an argument of type 'size_t' it is safe to ignore.


```
gcc -g -o so -fstack-protector stack_overflow.c
```


Run the program on the command line as follows, and copy the output between the lines below.

```
./so hi
```


Now, run the program with larger input, as follows, and copy the output below.

```
./so hiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii
```

Notice the messages towards the end of your output. There is a compiler feature that is preventing us from "smashing the stack", i.e. performing a stack buffer overflow. To disable this feature, recompile as follows.

```
gcc -fno-stack-protector -g -o so stack_overflow.c
```

Now, run the program with larger input, as follows, and copy the output below.

```
./so hiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii
```

GATE 2

Using a text editor, create the file ans_check.c and fill it with the following text:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int check_answer(char *ans) {

  int ans_flag = 0;
  char ans_buf[16];

  strcpy(ans_buf, ans);

  if (strcmp(ans_buf, "forty-two") == 0)
    ans_flag = 1;

  return ans_flag;

}

int main(int argc, char *argv[]) {

  if (argc < 2) {
    printf("Usage: %s <answer>\n", argv[0]);
    exit(0);
  }
  if (check_answer(argv[1])) {
    printf("Right answer!\n");
  } else {
    printf("Wrong answer!\n");
  }

}
```


Take a moment to read and understand this code. Between the lines below, briefly explain what it does.

Next, compile the program as follows.

```
gcc -g -fno-stack-protector -o ans_check ans_check.c
```

Use the following transcript to execute the program several times. Capture your console transcript in the space following (There are exactly 29 “1”s).

```
./ans_check
./ans_check yes
./ans_check forty-two
./ans_check 11111111111111111111111111111
```

Take no more than 90 seconds to try to explain the last result.

GATE 3

To see what's happening, we will examine execution within gdb. Use the following transcript, and capture the output between the lines below.


```
gdb ans_check -q
list 1
list
list
break 10
break 15
run 11111111111111111111111111111
```


As you may know, debug breakpoints trigger before the designated instruction/line has executed. So, in this case, we are on the strcpy line but it has not yet executed. Examine the two key values, as follows, and copy the output below.

```
x/s ans_buf
x/xw &ans_flag
```

Continue execution by entering `c <enter>` on the gdb command line.

At this breakpoint, the strcpy has completed, so once again examine the variables and copy the output below.

```
x/s ans_buf
x/xw &ans_flag
```

So this stack buffer overflow spilled over into the condition variable. In C, any non-zero value evaluates to "true" so the conditional check on line 25 passes . You may have noticed that if the static variables had been laid out in a different order (ans_buf above ans_flag), then it would have been safe from the overflow. So, this type of vulnerability may exist in a program, but it will not always be present. We next consider the vulnerability that will always be present.

Exit gdb with `quit`.

GATE 4

Enter gdb again with the following command.

```
gdb ans_check -q
```


Set a breakpoint at line 25. Record below the address associated with this breakpoint.

Now, disassemble the main function with the following command, and copy the output below.

```
disass main
```


In the main disassembly, find the address of the breakpoint. The assembly instructions starting here and ending with the call to `check_answer` implement line 25. Record in the following space the address of the instruction after the call. This is the return address, and we'll be looking for it shortly.

Next, set breakpoints for lines 10 and 15. That makes a total of 3 breakpoints set: line 25 (pre-call), line 10 (pre-strcpy), and line 15 (pre-return).

Now execute the program with the following gdb command.

```
run 11111111111111111111111111
```

Next, examine the stack register and the stack itself with the following commands. Record the output below. (You may find it easier to read if you shrink the font of your captured output so that lines are not broken.)

```
i r rsp
x/32xw $rsp
```

The output above displays the stack location and the stack contents prior to the call to `check_answer`.

Enter `c <enter>` to move to the next breakpoint.

Once again, examine the stack information and record the output below.

```
i r rsp
x/32xw $rsp
```


Examine the address and contents of the two static variables, as follows.

```
x/s ans_buf
x/xw &ans_flag
```


Find them in the stack output above, and change the font of stack locations to be bold. Similarly, find the word in the stack that corresponds to the return address that you recorded above. Change that word's font to bold as well.

Finally, continue execution with `c <enter>`. Capture the stack information below, and change the text corresponding to `ans_buf`, `ans_flag` and the return address to bold.

```
i r rsp
x/32xw $rsp
```

As you can see, ans_flag has been overwritten on the stack but the return address has not been modified.

GATE 5

Repeat the steps between gates 4 and 5, but this time use the smallest input string needed to overwrite the return address on the stack. Copy your gdb input and output below, beginning with your "run" command that includes the input string.

COMPLETE