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:

```c
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 AAAAAAAAAAAApass.

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$ ./oflow_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$ ./oflow_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$ ./oflow_partA AAAAAAAAAAAAAAAAAAAA

Buffer: AAAAAAAAAAAAA
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 AAAAAAAAAAAApass

Buffer: AAAAAAAAAAAAA
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 AAAAAAAAAAAAAAAAAAAA

Buffer: AAAAAAAAAAAA
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 AAAAAAAAAAAApass

Buffer: AAAAAAAAAAAA
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 "AAAAAAAAAAAAAAAAAAAA". What happens if we append "hacker" to the input that got us the key?

```
user@host$ ./oflow_partB AAAAAAAAAAAApasshacker

Buffer: AAAAAAAAAAAA
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 AAAAAAAAAAAAAAAAAAAA

Buffer: AAAAAAAAAAAA
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: AAAAAAAAAAAA
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: AAAAAAAAAAAA
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 AAAAAAAAAAAAAAAAAAAA
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 AAAAAAAAAAAAAAAA
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 AAAAAAAAAAAAAAAAAAAA
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 AAAAAAAAAAAAAAAKCAH
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 AAAAAAAAAAAAAAAAHACK
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 AAAAAAAAAAAAAAAAKCAH
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.

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

```
/* 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 the there are more than 3 arguments.

Suppose we have a very simple C function:

```c
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.

```c
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 they value is corrupted the 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

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

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

```
 */
.syntax unified
.arm
.cpu cortex-a7


.section .text
.global main
.equ buffer_size, 12
.equ target_size, 4

main:
  push {r0-r7,fp,lr}

  // Make sure there is an argument
  mov r2, r0
  cmp r2, #2
  beq _get_argument

  // Print usage statement
  ldr r0, = _usage_str
  ldr r1, = _usage_str_len
  bl write_to_stdout
  b exit

  _get_argument:
  ldr r8, [r1, #4] // r8 <-- (char *) argv[1]

  // ## SETUP LOCAL VARIABLES ##
  sub sp, sp, target_size
  mov r5, sp         // Space for target
  sub sp, sp, buffer_size
  mov r6, sp         // space for overflow buffer

  // Set up target: "fail"
  ldr r7, =#0x6C696166
  str r7, [r5]

  mov r0, r6
  mov r1, buffer_size
  bl memzero

  // Copy commandline argument to buffer.
  mov r0, r8
  mov r1, r6
  bl copy_str

  mov r0, r5
  mov r1, r6
  bl review_stack

  mov r8, 0
  // Test target == "pass"
  ldr r1, [r5]
  ldr r2, =#0x73736170
  cmp r1, r2
  addeq r8, r8, #1
  blne target_fail
```

```
  cmp r8, #1
  bleq print_key


  exit:
  pop {r0-r7,fp,lr}
  mov r7, #1
  svc #0


/* target_fail:
 *  Informs user that the target was overwritten with incorrect value.
 * Input: None
 * Output: None
 * Side Effects:
 *  Writes failure message to stdout.
 */
target_fail:
  push {r7,fp}
  mov r0, #1
  ldr r1, =_target_fail
  ldr r2, =_target_fail_len
  mov r7, #4
  svc #0

  pop {r7,fp}
  bx lr


/* print_key:
 *  Shows the secret key with the target buffer is overwritten
    correctly.
 * Input: None
 * Output: None
 * Side Effects:
 *  Writes secret key to stdout.
 */
print_key:
  push {r4-r10, fp, lr}

  ldr r0, =_success_str
  ldr r1, =_success_str_len
  bl write_to_stdout

  ldr r0, =_key_str
  ldr r1, =_key_str_len
  bl write_to_stdout

  pop {r4-r10, fp, lr}
  bx lr


/* review_stack:
 *  function to print out formatted local variable contents.
 * Input:
 *  r0: target
 *  r1: overflow buffer
```

```
 * Output: None
 * Side effects:
 *  Prints the contents of the canary, target, and overflow buffers.
 */
review_stack:
  push {r4,r5,r6,fp,lr}
  mov r5, r0
  mov r6, r1

  ldr r0, =_newline_str
  ldr r1, =_newline_str_len
  bl write_to_stdout

  // Print out buffer //
  ldr r0, =_buffer_str
  ldr r1, =_buffer_str_len
  bl write_to_stdout

  mov r0, r6
  ldr r1, =buffer_size
  bl write_by_len

  ldr r0, =_newline_str
  ldr r1, =_newline_str_len
  bl write_to_stdout

  // Print out target //
  ldr r0, =_target_str
  ldr r1, =_target_str_len
  bl write_to_stdout

  mov r0, r5
  ldr r1, =target_size
  bl write_by_len

  ldr r0, =_newline_str
  ldr r1, =_newline_str_len
  bl write_to_stdout

  ldr r0, =_newline_str
  ldr r1, =_newline_str_len
  bl write_to_stdout

  pop {r4,r5,r6,fp,lr}
  bx lr


/* copy_str:
 *  Copy string from one buffer to another..
 * Input:
 *  r0: char* src
 *  r1: char* dst
 * Output: None
 * Side Effects: N/a
 */
copy_str:
  push {lr}
  _copy_str_while_loop:
```

```
    ldr r2, [r0]
    and r2, r2, #0xFF

    cmp r2, #0
    beq _end_copy_str_while_loop
    strb r2, [r1]
    add r0, r0, #1
    add r1, r1, #1
    b _copy_str_while_loop

  _end_copy_str_while_loop:

  pop {lr}
  bx lr


/* memzero:
 *  Fill vuffer with zeros.
 * Input:
 *  r0: char* buffer
 *  r1: lenth of buffer
 * Output: None
 * Side Effects: N/A
 */
memzero:
  push {r4, lr}
  cmp r1, #0
  add r1, #1
  beq _end_memzero
  _memzero:
  mov r2, #0x0
    _memzero_while_loop:
    sub r1, #1
    cmp r1, #0
    beq _end_memzero_while_loop
    strb r2, [r0]
    add r0, #1
    b _memzero_while_loop
    _end_memzero_while_loop:
  _end_memzero:

  pop {r4, lr}
  bx lr


/* write_by_len:
 *  print output using given length
 * Input:
 *  r0: char* buffer
 *  r1: lenth to print out
 * Output: None
 * Side Effects: Prints r1 chars from r0 to stdout.
 */
write_by_len:
  push {r4-r9, fp, lr}

  mov r4, #0
  mov r5, r1
```

```
  sub r5, r5, #1
  mov r1, r0
  for_loop_one_write_by_len:
    ldr r6, [r1]
    and r6, r6, #0xff
    cmp r6, #0
    bne wbl_flowbl_write
    ldr r6, [r1]
    add r6, r6, #0x20
    str r6, [r1]

    wbl_flowbl_write:
    mov r0, #1
    mov r2, #1
    mov r7, #4
    svc #0

    cmp r4, r5
    beq end_for_loop_one_write_by_len
    add r4, r4, #1
    add r1, r1, #1
    b for_loop_one_write_by_len

  end_for_loop_one_write_by_len:

  pop {r4-r9, fp, lr}
  bx lr


/* write_to_stdout:
 *  Wrapper for Syscall::write. Prints null-terminated string.
 * Input:
 *  r0: char* buffer
 *  r1: lenth to print out
 * Output: None
 * Side Effects: Prints r1 chars from r0 to stdout.
 */
write_to_stdout:
  push {fp}
  mov r2, r1
  mov r1, r0
  mov r0, #1  // fd: stdout
  mov r7, #4  // syscall: write
  svc #0
  pop {fp}
  bx lr


.section .data
.balign 4
  _usage_str:      .asciz "./binary <string>\nThis program takes one argument.\n"
  .set _usage_str_len, .-_usage_str
  _canary_str:     .asciz "Canary: "
  .set _canary_str_len, .-_canary_str
  _canary_fail:     .asciz "You overwrote the stack canary.\n"
  .set _canary_fail_len, .-_canary_fail
  _target_str:     .asciz "Target: "
```

```
.set _target_str_len, .-_target_str
_target_fail:    .asciz "Target not set to \"pass\".\n"
.set _target_fail_len, .-_target_fail
_buffer_str:     .asciz "Buffer: "
.set _buffer_str_len, .-_buffer_str
_newline_str:        .asciz "\n"
.set _newline_str_len, .-_newline_str
_hacker_str:     .asciz "hacker"
.set _hacker_str_len, .-_hacker_str
_fail_str:       .asciz "fail"
.set _fail_str_len, .-_fail_str
_pass_str:       .asciz "pass"
.set _pass_str_len, .-_pass_str
_success_str: .asciz "Congrats! You overwrote the target!\n"
.set _success_str_len, .-_success_str
_key_str:     .asciz "The key is: \"smash the stack\"\n"
.set _key_str_len, .-_key_str
```

oflow_partB.s

```
/* Author: Chuck L. Norris
 * Association: Johns Hopkins University
 * Date: 24 March 2021
 * Purpose: This provides an exploitable buffer as a vehicle for exposition of stack/buffer
 *  overflows.
 * Theory of Operation:
 *  Local variables are declared on the stack. User input is copied to the stack
 *  using a vulnerable strcopy-like function that does no length checking. Using this
 *  vulnerable function the user can overwrite one of the local variables. There is a
 *  check on the value of two local variables. The first is the target which is check for
 *  a particular value: "pass" (non-zero terminated). The other is a canary value that is
 *  to ensure the user does not overwrite further in the stack than necessary.If the canary
 *  value is unaltered and the target value is correct the program will print a secret key.
 * Input:
 *  string: Single command line argument. Input is interpretted as ASCII.
 * Output:
 *  Write strings to standard out depending on the state of the stack and local
 *  variables.
 *
 * Local Variables: The way this binary allocates space on the stack and then
 * keeps track of those variables is not consistent with typical compiled
 * programs. In this program, a point to each variable on the stack is
 * stored in a register. This can be done because we know exactly how many
 * local variables we have and the number of local variables is small.
```

```
 * Typically in compiled programs an offset from the stack pointer is stored.
 * To access a local variable we would add the offset to the stack pointer.
 *
 * An additional step taken in this binary is to zero out memory on the stack. This is
 * typically done by a compiler during program initialization.
 *
 * Assembly Information:
 * System Call Information:
 *   man 2 syscall                                    // shows the ARM ABI for syscalls
 *   /usr/include/arm-linux-gnueabihf/asm/unistd-common.h // header for syscalls
 * Assemble and Link:
 *   To assemble and link the .s file into a binary use the following shell command
 *     as -mcpu=cortex-a72 -g --warn  <name>.s -o <name>.o; ld <name>.o -o <name>   // doesnt allow static linking
 *     gcc -Wall -fpic -fno-stack-protector -z execstack <name>.s -o <name>
 *
 * ARM Convention:
 *  r0-r3: Arguments, additional agruments taken from stack
 *  r4-r11: Local variables
 *  r0: return values
 */
.syntax unified
.arm
.cpu cortex-a7


.section .text
.global main
.equ buffer_size, 12
.equ target_size, 4
.equ canary_size, 6

main:
  push {r0-r7,fp,lr}

  // Make sure there is an argument
  mov r2, r0
  cmp r2, #2
  beq _get_argument

  // Print usage statement
  ldr r0, = _usage_str
  ldr r1, = _usage_str_len
  bl write_to_stdout
  b exit

  _get_argument:
  ldr r8, [r1, #4] // r8 <-- (char *) argv[1]

  // ## SETUP LOCAL VARIABLES ##
  sub sp, sp, canary_size
```

```
mov r4, sp          // Space for canary
sub sp, sp, target_size
mov r5, sp          // Space for target
sub sp, sp, buffer_size
mov r6, sp          // space for overflow buffer

// Set up target: "fail"
ldr r7, =#0x6C696166
str r7, [r5]

// Set up Canary: "hacker"
ldr r7, =#0x6B636168
str r7, [r4]
ldr r7, =#0x7265
str r7, [r4, #4]

mov r0, r6
mov r1, buffer_size
bl memzero

// Copy commandline argument to buffer.
mov r0, r8
mov r1, r6
bl copy_str

mov r0, r4
mov r1, r5
mov r2, r6
bl review_stack

mov r8, 0
// Test target == "pass"
ldr r1, [r5]
ldr r2, =#0x73736170
cmp r1, r2
addeq r8, r8, #1
blne target_fail

// Test canary == "hacker"
ldr r1, =#0x6B636168
ldr r2, [r4]
eor r3, r1, r2
ldr r1, =#0x7265
ldr r2, [r4, #4]
eor r2, r1, r2
add r1, r2, r3
cmp r1, #0
addeq r8, r8, #1
blne canary_fail

cmp r8, #2
bleq print_key


exit:
pop {r0-r7,fp,lr}
mov r7, #1
svc #0
```

```


/* target_fail:
 *  Informs user that the target was overwritten with incorrect value.
 * Input: None
 * Output: None
 * Side Effects:
 *  Writes failure message to stdout.
 */
target_fail:
  push {r7,fp}
  mov r0, #1
  ldr r1, =_target_fail
  ldr r2, =_target_fail_len
  mov r7, #4
  svc #0

  pop {r7,fp}
  bx lr


/* canary_fail:
 *  Informs user that the canary was overwritten.
 * Input: None
 * Output: None
 * Side Effects:
 *  Writes failure message to stdout.
 */
canary_fail:
  push {r7, fp}
  mov r0, #1
  ldr r1, =_canary_fail
  ldr r2, =_canary_fail_len
  mov r7, #4
  svc #0

  pop {r7,fp}
  bx lr


/* print_key:
 *  Shows the secret key with the target buffer is overwritten
    correctly.
 * Input: None
 * Output: None
 * Side Effects:
 *  Writes secret key to stdout.
 */
print_key:
  push {r4-r10, fp, lr}

  ldr r0, =_success_str
  ldr r1, =_success_str_len
  bl write_to_stdout

  ldr r0, =_key_str
  ldr r1, =_key_str_len
  bl write_to_stdout
```

```
  pop {r4-r10, fp, lr}
  bx lr


/* review_stack:
 *  function to print out formatted local variable contents.
 * Input:
 *  r0: canary
 *  r1: target
 *  r2: overflow buffer
 * Output: None
 * Side effects:
 *  Prints the contents of the canary, target, and overflow buffers.
 */
review_stack:
  push {r4,r5,r6,fp,lr}
  mov r4, r0
  mov r5, r1
  mov r6, r2

  ldr r0, =_newline_str
  ldr r1, =_newline_str_len
  bl write_to_stdout

  // Print out buffer //
  ldr r0, =_buffer_str
  ldr r1, =_buffer_str_len
  bl write_to_stdout

  mov r0, r6
  ldr r1, =buffer_size
  bl write_by_len

  ldr r0, =_newline_str
  ldr r1, =_newline_str_len
  bl write_to_stdout

  // Print out target //
  ldr r0, =_target_str
  ldr r1, =_target_str_len
  bl write_to_stdout

  mov r0, r5
  ldr r1, =target_size
  bl write_by_len

  ldr r0, =_newline_str
  ldr r1, =_newline_str_len
  bl write_to_stdout

  // Print out canary //
  ldr r0, =_canary_str
  ldr r1, =_canary_str_len
  bl write_to_stdout

  mov r0, r4
  ldr r1, =canary_size
```

```
  bl write_by_len

  ldr r0, =_newline_str
  ldr r1, =_newline_str_len
  bl write_to_stdout



  ldr r0, =_newline_str
  ldr r1, =_newline_str_len
  bl write_to_stdout

  pop {r4,r5,r6,fp,lr}
  bx lr


/* copy_str:
 *  Copy string from one buffer to another..
 * Input:
 *  r0: char* src
 *  r1: char* dst
 * Output: None
 * Side Effects: N/a
 */
copy_str:
  push {lr}
  _copy_str_while_loop:
    ldr r2, [r0]
    and r2, r2, #0xFF

    cmp r2, #0
    beq _end_copy_str_while_loop
    strb r2, [r1]
    add r0, r0, #1
    add r1, r1, #1
    b _copy_str_while_loop

  _end_copy_str_while_loop:

  pop {lr}
  bx lr


/* memzero:
 *  Fill vuffer with zeros.
 * Input:
 *  r0: char* buffer
 *  r1: lenth of buffer
 * Output: None
 * Side Effects: N/A
 */
memzero:
  push {r4, lr}
  cmp r1, #0
  add r1, #1
  beq _end_memzero
  _memzero:
  mov r2, #0x0
```

```
    _memzero_while_loop:
    sub r1, #1
    cmp r1, #0
    beq _end_memzero_while_loop
    strb r2, [r0]
    add r0, #1
    b _memzero_while_loop
    _end_memzero_while_loop:
  _end_memzero:

  pop {r4, lr}
  bx lr


/* write_by_len:
 *  print output using given length
 * Input:
 *  r0: char* buffer
 *  r1: lenth to print out
 * Output: None
 * Side Effects: Prints r1 chars from r0 to stdout.
 */
write_by_len:
  push {r4-r9, fp, lr}

  mov r4, #0
  mov r5, r1
  sub r5, r5, #1
  mov r1, r0
  for_loop_one_write_by_len:
    ldr r6, [r1]
    and r6, r6, #0xff
    cmp r6, #0
    bne wbl_flowbl_write
    ldr r6, [r1]
    add r6, r6, #0x20
    str r6, [r1]

    wbl_flowbl_write:
    mov r0, #1
    mov r2, #1
    mov r7, #4
    svc #0

    cmp r4, r5
    beq end_for_loop_one_write_by_len
    add r4, r4, #1
    add r1, r1, #1
    b for_loop_one_write_by_len

  end_for_loop_one_write_by_len:

  pop {r4-r9, fp, lr}
  bx lr


/* write_to_stdout:
 *  Wrapper for Syscall::write. Prints null-terminated string.
```

```
 * Input:
 *  r0: char* buffer
 *  r1: lenth to print out
 * Output: None
 * Side Effects: Prints r1 chars from r0 to stdout.
 */
write_to_stdout:
  push {fp}
  mov r2, r1
  mov r1, r0
  mov r0, #1  // fd: stdout
  mov r7, #4  // syscall: write
  svc #0
  pop {fp}
  bx lr


.section .data
.balign 4
  _usage_str:      .asciz "./binary <string>\nThis program takes one argument.\n"
  .set _usage_str_len, .-_usage_str
  _canary_str:     .asciz "Canary: "
  .set _canary_str_len, .-_canary_str
  _canary_fail:     .asciz "You overwrote the stack canary.\n"
  .set _canary_fail_len, .-_canary_fail
  _canary_success: .asciz "You didnt overwrite the stack canary.\n"
  .set _canary_success_len, .-_canary_success
  _target_str:     .asciz "Target: "
  .set _target_str_len, .-_target_str
  _target_fail:    .asciz "Target not set to \"pass\".\n"
  .set _target_fail_len, .-_target_fail
  _buffer_str:     .asciz "Buffer: "
  .set _buffer_str_len, .-_buffer_str
  _newline_str:        .asciz "\n"
  .set _newline_str_len, .-_newline_str
  _hacker_str:     .asciz "hacker"
  .set _hacker_str_len, .-_hacker_str
  _fail_str:       .asciz "fail"
  .set _fail_str_len, .-_fail_str
  _pass_str:       .asciz "pass"
  .set _pass_str_len, .-_pass_str
  _success_str: .asciz "Congrats! You overwrote the target without touching the canary!\n"
  .set _success_str_len, .-_success_str
  _key_str:     .asciz "The key is: \"halt and catch fire\"\n"
  .set _key_str_len, .-_key_str
```

oflow_partC.s

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

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

```
 * ARM Convention:
 *  r0-r3: Arguments, additional agruments taken from stack
 *  r4-r11: Local variables
 *  r0: return values
 */
.syntax unified
.arm
.cpu cortex-a7


.section .text
.global main
.equ buffer_size, 12
.equ target_size, 4
.equ canary_size, 6

main:
  push {r0-r7,fp,lr}

  // Make sure there is an argument
  mov r2, r0
  cmp r2, #2
  beq _get_argument

  // Print usage statement
  ldr r0, = _usage_str
  ldr r1, = _usage_str_len
  bl write_to_stdout
  b exit

  _get_argument:
  ldr r8, [r1, #4] // r8 <-- (char *) argv[1]

  // ## SETUP LOCAL VARIABLES ##
  sub sp, sp, target_size
  mov r4, sp         // Space for target
  sub sp, sp, canary_size
  mov r5, sp         // Space for canary
  sub sp, sp, buffer_size
  mov r6, sp         // space for overflow buffer

  // Set up Canary: "hacker"
  ldr r7, =#0x6B636168
  str r7, [r5]
  ldr r7, =#0x7265
  str r7, [r5, #4]

  // Set up target: "fail"
  ldr r7, =#0x6C696166
  str r7, [r4]

  mov r0, r6
  mov r1, buffer_size
  bl memzero

  // Copy commandline argument to buffer.
  mov r0, r8
  mov r1, r6
```

```
  bl copy_str

  mov r0, r5
  mov r1, r4
  mov r2, r6
  bl review_stack

  mov r8, 0
  // Test target == "pass"
  ldr r1, [r4]
  ldr r2, =#0x73736170    //ssap
  cmp r1, r2
  addeq r8, r8, #1
  blne target_fail

  // Test canary == "hacker"
  ldr r1, =#0x6B636168  // kcah
  ldr r2, [r5]
  eor r3, r1, r2
  ldr r1, =#0x7265      // "re"
  ldr r2, [r5, #4]
  mov r0, #0xFFFF
  and r2, r2, r0
  eor r2, r1, r2
  add r1, r2, r3
  cmp r1, #0
  addeq r8, r8, #1
  blne canary_fail

  cmp r8, #2
  bleq print_key


  exit:
  pop {r0-r7,fp,lr}
  mov r7, #1
  svc #0


/* target_fail:
 *  Informs user that the target was overwritten with incorrect value.
 * Input: None
 * Output: None
 * Side Effects:
 *  Writes failure message to stdout.
 */
target_fail:
  push {r7,fp}
  mov r0, #1
  ldr r1, =_target_fail
  ldr r2, =_target_fail_len
  mov r7, #4
  svc #0

  pop {r7,fp}
  bx lr


```

```
/* canary_fail:
 *  Informs user that the canary was overwritten.
 * Input: None
 * Output: None
 * Side Effects:
 *  Writes failure message to stdout.
 */
canary_fail:
  push {r7, fp}
  mov r0, #1
  ldr r1, =_canary_fail
  ldr r2, =_canary_fail_len
  mov r7, #4
  svc #0

  pop {r7,fp}
  bx lr


/* print_key:
 *  Shows the secret key with the target buffer is overwritten
    correctly.
 * Input: None
 * Output: None
 * Side Effects:
 *  Writes secret key to stdout.
 */
print_key:
  push {r4-r10, fp, lr}

  ldr r0, =_success_str
  ldr r1, =_success_str_len
  bl write_to_stdout

  ldr r0, =_key_str
  ldr r1, =_key_str_len
  bl write_to_stdout

  pop {r4-r10, fp, lr}
  bx lr


/* review_stack:
 *  function to print out formatted local variable contents.
 * Input:
 *  r0: canary
 *  r1: target
 *  r2: overflow buffer
 * Output: None
 * Side effects:
 *  Prints the contents of the canary, target, and overflow buffers.
 */
review_stack:
  push {r4,r5,r6,fp,lr}
  mov r4, r0
  mov r5, r1
  mov r6, r2

```

```
  ldr r0, =_newline_str
  ldr r1, =_newline_str_len
  bl write_to_stdout

  // Print out buffer //
  ldr r0, =_buffer_str
  ldr r1, =_buffer_str_len
  bl write_to_stdout

  mov r0, r6
  ldr r1, =buffer_size
  bl write_by_len

  ldr r0, =_newline_str
  ldr r1, =_newline_str_len
  bl write_to_stdout

  // Print out canary //
  ldr r0, =_canary_str
  ldr r1, =_canary_str_len
  bl write_to_stdout

  mov r0, r4
  ldr r1, =canary_size
  bl write_by_len

  ldr r0, =_newline_str
  ldr r1, =_newline_str_len
  bl write_to_stdout
  // Print out target //
  ldr r0, =_target_str
  ldr r1, =_target_str_len
  bl write_to_stdout

  mov r0, r5
  ldr r1, =target_size
  bl write_by_len

  ldr r0, =_newline_str
  ldr r1, =_newline_str_len
  bl write_to_stdout

  ldr r0, =_newline_str
  ldr r1, =_newline_str_len
  bl write_to_stdout

  pop {r4,r5,r6,fp,lr}
  bx lr


/* copy_str:
 *  Copy string from one buffer to another..
 * Input:
 *  r0: char* src
 *  r1: char* dst
 * Output: None
 * Side Effects: N/a
 */
```

```
copy_str:
  push {lr}
  _a:
  _copy_str_while_loop:
    ldr r2, [r0]
    and r2, r2, #0xFF

    cmp r2, #0
    beq _end_copy_str_while_loop
    strb r2, [r1]
    add r0, r0, #1
    add r1, r1, #1
    b _copy_str_while_loop

  _end_copy_str_while_loop:

  pop {lr}
  bx lr


/* memzero:
 *  Fill vuffer with zeros.
 * Input:
 *  r0: char* buffer
 *  r1: lenth of buffer
 * Output: None
 * Side Effects: N/A
 */
memzero:
  push {r4, lr}
  cmp r1, #0
  add r1, #1
  beq _end_memzero
  _memzero:
  mov r2, #0x0
    _memzero_while_loop:
    sub r1, #1
    cmp r1, #0
    beq _end_memzero_while_loop
    strb r2, [r0]
    add r0, #1
    b _memzero_while_loop
    _end_memzero_while_loop:
  _end_memzero:

  pop {r4, lr}
  bx lr


/* write_by_len:
 *  print output using given length
 * Input:
 *  r0: char* buffer
 *  r1: lenth to print out
 * Output: None
 * Side Effects: Prints r1 chars from r0 to stdout.
 */
write_by_len:
```

```
  push {r4-r9, fp, lr}

  mov r4, #0
  mov r5, r1
  sub r5, r5, #1
  mov r1, r0
  for_loop_one_write_by_len:
    ldr r6, [r1]
    and r6, r6, #0xff
    cmp r6, #0
    bne wbl_flowbl_write
    ldr r6, [r1]
    add r6, r6, #0x20
    str r6, [r1]

    wbl_flowbl_write:
    mov r0, #1
    mov r2, #1
    mov r7, #4
    svc #0

    cmp r4, r5
    beq end_for_loop_one_write_by_len
    add r4, r4, #1
    add r1, r1, #1
    b for_loop_one_write_by_len

  end_for_loop_one_write_by_len:

  pop {r4-r9, fp, lr}
  bx lr


/* write_to_stdout:
 *  Wrapper for Syscall::write. Prints null-terminated string.
 * Input:
 *  r0: char* buffer
 *  r1: lenth to print out
 * Output: None
 * Side Effects: Prints r1 chars from r0 to stdout.
 */
write_to_stdout:
  push {fp}
  mov r2, r1
  mov r1, r0
  mov r0, #1  // fd: stdout
  mov r7, #4  // syscall: write
  svc #0
  pop {fp}
  bx lr


.section .data
.balign 4
  _usage_str:      .asciz "./<binary> <string>\nThis program takes one argument.\n"
  .set _usage_str_len, .-_usage_str
  _canary_str:     .asciz "Canary: "
```

```
  .set _canary_str_len, .-_canary_str
  _canary_fail:     .asciz "You overwrote the stack canary.\n"
  .set _canary_fail_len, .-_canary_fail
  _canary_success: .asciz "You didnt overwrite the stack canary.\n"
  .set _canary_success_len, .-_canary_success
  _target_str:     .asciz "Target: "
  .set _target_str_len, .-_target_str
  _target_fail:    .asciz "Target not set to \"pass\".\n"
  .set _target_fail_len, .-_target_fail
  _buffer_str:     .asciz "Buffer: "
  .set _buffer_str_len, .-_buffer_str
  _newline_str:        .asciz "\n"
  .set _newline_str_len, .-_newline_str
  _hacker_str:     .asciz "hacker"
  .set _hacker_str_len, .-_hacker_str
  _fail_str:       .asciz "fail"
  .set _fail_str_len, .-_fail_str
  _pass_str:       .asciz "pass"
  .set _pass_str_len, .-_pass_str
  _success_str: .asciz "Congrats! You overwrote the target while perserving the canary!\n"
  .set _success_str_len, .-_success_str
  _key_str:     .asciz "The key is: \"hack the planet\"\n"
  .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.

```
#! /bin/sh

case $1 in
  "A" | "a")
    gcc -Wall -fpic -fno-stack-protector -z execstack oflow_partA.s -o oflow_partA
    ;;
  "B" | "b")
    gcc -Wall -fpic -fno-stack-protector -z execstack oflow_partB.s -o oflow_partB
    ;;
  "C" | "c")
    gcc -Wall -fpic -fno-stack-protector -z execstack oflow_partC.s -o oflow_partC
esac

if [ $? -eq 0 ]
then
  echo "Built."
else
  echo "Failed."
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

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ucontext.h>
#include <ucontext.h>
#include <signal.h>
#include <unistd.h>
#include <stdbool.h>

void fault_handler(int signo, siginfo_t *info, void *extra){
        printf("------------------------ERROR-------------------------\n");
    printf("Goodbye cruel world! I was a young program. And I have
        died too soon!\n");
    printf("You can avenge my death! I received Signal Number %d\n",
        signo);
    printf("Looks like I was near address %p at my untimely
        demise.\n", info->si_addr);
    abort();
}

void set_up_handlers(){
    struct sigaction act;
    act.sa_flags = SA_SIGINFO;
    act.sa_sigaction = fault_handler;
        if (sigaction(SIGFPE, &act, NULL) == -1) {
                perror("sigfpe: sigaction");
                exit(1);
        }
        if (sigaction(SIGSEGV, &act, NULL) == -1) {
                perror("sigsegv: sigaction");
                exit(1);
        }
        if (sigaction(SIGILL, &act, NULL) == -1) {
                perror("sigill: sigaction");
                exit(1);
        }
        if (sigaction(SIGBUS, &act, NULL) == -1) {
                perror("sigbus: sigaction");
                exit(1);
        }
}


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

```
        set_up_handlers();
    printf("Hello to the string consumer 3000!\n");
    char buf[10];
    if (argc > 1){
        printf("You provided me a string! Yum!\n");
        strcpy(buf,argv[1]);
    } else{
        printf("Provide your name as the first argument\n");
        printf("./demo jimbob\n");
    }
}
```

bet_you_cant.S

```
    .section .win_sec, "ax"
_bet_you_cant_get_here:
    adr r0, winstr
    bl puts
    movs r0, #0
    b exit
winstr:
    .ascii "  _____\n /      \\\n|  ()  ()  |\n \\   "
           "^   /\n  |||||\n  |||||\n\nOh no! The"
           "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).