Ch1_Readelf

The objective is to find the **static** password stored in code. The “readelf” command will show us all of the sections of this binary.

Run:

```
readelf -a Ch1_Readelf
```

Knowing that our password is statically stored, we can inspect the “.rodata” section.

```
  [16] .rodata           PROGBITS         00000000004007c0  000007c0
       000000000000003e  0000000000000000   A       0     0     4
```

The main thing needed is the section number of .rodata so that it can be displayed in the terminal. There is a different “readelf” command to perform a hex dump of a specific section.

Run:

```
readelf -x 16 Ch1_Readelf
```

```
Hex dump of section '.rodata':
0x004007c0 01000200 25730045 6e746572 20746865 ....%s.Enter the
0x004007d0 20706173 73776f72 643a2000 25387300  password: .%8s.
0x004007e0 32503535 34384f42 00547279 20616761 2P5548OB.Try aga
0x004007f0 696e2e00 476f6f64 204a6f62 2e00     in..Good Job..
```

Just as suspected, the password is stored in ASCII within the program.

Result:

```
Enter the password: 2P5548OB
Good Job.
```

Ch1_Ltrace

The objective is to display library calls performed by the executable directly to the terminal, specifically the <strcmp> function which compares against the correct password.

Run:

```
ltrace ./Ch1_Ltrace
```

When the program encounters <scanf> function to take in an argument, enter any string and hit enter. Because we are running “ltrace” we will see the what the <strcmp> function is comparing our string to in real-time.

```
)                              = 311
printf("Enter the password: ")                                   = 20
__isoc99_scanf(0x40081c, 0x7ffece7a35f0, 0, 0Enter the password: test
)                                   = 1
strcmp("test", "MBaVeZMf")                                       = 39
puts("Try again."Try again.
)                                                          = 11
+++ exited (status 0) +++
```

Result:

```
Enter the password: MBaVeZMf
Good Job.
```

Ch2_01_Endian

After running the program:

```
char password[9]="xxxxxxxx";
unsigned int * ip;
ip = (unsigned int *) &password;
printf("%08x : %08x\n", *ip, *(ip+1));

Output of above C code
42627735 : 69576941
```

You are given the code that the program is using, as well as the output of that code. The important thing to notice is that the password is 8 characters long. You can either look at the first line (password[n] = size n-1) or count the two-digit representation of ASCII characters in the "output" portion.

The easiest way to figure out what gets outputted is to enter a string like "ABCDEFGH" which we can easily decipher by looking at an ASCII table (capitalization matters!).

Output of "ABCDEFGH":

```
Enter the password: ABCDEFGH
Your input represented as consecutive 4-byte integers:
44434241 : 48474645
```

We can reference what comes out against what we know about the string we entered.

Input:

```
41(A) 42(B) 43(C) 44(D) 45(E) 46(F) 46(G) 48(H)
```

Output:

```
44(D) 43(C) 42(B) 41(A) : 48(H) 47(G) 46(F) 45(E)
```

We can now reference the "output of above C code" section to convert the hex numbers provided to ASCII (use google) and then enter them in the correct order which we just figured out above.

Convert:

"42627735 : 69576941" becomes "Bbw5 : iWiA"

Rearrange:

"Bbw5 : iWiA" becomes "5wbB : AiWi"

Result:

```
Enter the password: 5wbBAiWi
Your input represented as consecutive 4-byte integers:
42627735 : 69576941

Good Job.
```

Ch2_01_Showkey

The password will be entered in three parts: hexadecimal, decimal, and octal. Before trying to decipher the string, perform the “showkey -a” command on the example provided (ab%).

    For example, a password string of ab% would yield a password
    of 616225 097098037 141142045

Run:

    showkey -a

Once you type the command hit <return> and manually type in ‘a’, ‘b’, and ‘%’. Press <ctrl d> when finished. You will get the following:

```
a       97 0141 0x61
b       98 0142 0x62
%       37 0045 0x25
```

The decimal values print first (green) and must be padded with 0’s. The octal values (blue) are next, use 3 digits for the octal values. Last is the hex values (yellow) for the inputs.

The password string is provided. What’s left is to perform the same procedure on this string and then enter the password as hex, decimal and octal inputs.

    The password as a string is mYQ5

Run:

    showkey -a

Enter the string characters individually and press <return>.

```
m      109 0155 0x6d
Y       89 0131 0x59
Q       81 0121 0x51
5       53 0065 0x35
```

Use the provided values to put it into the format specified by the binary and enter as the password. The password will be: 6d595135 109089081053 155131121065

Result:

```
Enter the password: 6d595135 109089081053 155131121065
Good Job.
```

Ch2_03_IntOverflow

You are given an unsigned char, unsigned short, and unsigned int:

```
unsigned char addc, c=0x9c;
  unsigned short adds, s=0xb3c1;
  unsigned int addi, i=0xc9e2acae;
```

The goal is to find the positive values for each of these to make them add up to zero. First step is to figure out what the unsigned values represent numerically. For this, they need to be converted to the two’s complement value. You can do this manually or google a “unsigned hex to binary” converter.

Calculating Two’s complement (negative numbers):

X = [-2^(w-1) + 2^(w-2) … + 2^0] where w represents word size

char 0x9c
binary = 10011100
unsigned decimal = -(2^7) +2^4 +2^3 + 2^2 = -100

Short 0xb3c1
binary = 10110011 11000001
Unsigned decimal = -(2^15) + 2^13 + ... = -19519

Unsigned int 0xc9e2acae
Binary = 11001001 11100010 10101100 10101110
Unsigned decimal = -(2^31) + 2^30 + ... = -907891538

Now run the program again and enter the positive versions of these decimal values separated by spaces as the password.

Result:

```
Enter the password: 100 19519 907891538
Good Job.
```

Ch2_03_TwosComplement

We are told the password is the value represented by a 12-bit two's complement number. The first step is to find the value of the most significant bit. We can set size to 12 and work our way through until size equals 0.

Most significant bit ([1xxxxxxxxxxx]):

```
size = s = 12

-(2^s-1) = -(2^11) = -2048
```

Positive values at the resulting 11 bit's ([x11100000101]):

```
Size = s = 12

[2^s-2 + 2^s-3 + 2^s-4 + 2^s-5 + 2^s-6 + 2^s-7 + 2^s-8 + 2^s-9 +
 2^s-10 + 2^s-11 + 2^s-12] = 1797
```

Total:

```
-2048 + 1797 = -251
```

Result:

```
Enter the password: -251
Good Job.
```

Ch2_03_XorInt

The program asks for a 4-byte hexadecimal number (e.g. 4f91853a) that will be xor'd against a key. Recall what xor does:

```
  1001
^ 0001
  1000
```

The xor command returns a "1" only if there is a single "1" at a position.

If the key were "1001" and we entered "0001" then we would in effect return the first three characters exactly the same as the key. But, since both the key and our input contains a "1" at the last position, it returns a "0".

Since binary is just another representation of hexadecimal values, we can do the same thing even though the program asks for hex values.

```
0x44 is the same as 01000100
```

Recall that when our input contained a "0", we returned the same binary digit as the key in our example. That means that if our 4-byte hexadecimal input was all "0" we could return the key back.

Enter a 4-byte hexadecimal representation of all 0's as the input:

```
Enter the password: 00000000
I have XOR encrypted the value you entered (0).
The encrypted result is: 9ada3af5
Try again.
```

Now run the program again and enter the password.

Result:

```
Enter the password: 9ada3af5
I have XOR encrypted the value you entered (9ada3af5).
The encrypted result is: 0
Good Job.
```

Ch2_05_FloatConvert

We are asked to convert floating point numbers between hex and decimal values.

> Give the machine representation of the floating point number 24790.0 (in hex) followed by the floating point value represented by the hex pattern 47646800

Let's start with the floating point number 24790.0 which is the same as the decimal (integral)number 24790 because the ".0" (fractional number) doesn't represent any significant figures.

Convert integral to binary. Divide successively by 2 getting a "0" for whole numbers and "1" fon non-whole numbers. Only the whole number is carried on to the next operation.

24790/2	= 12395	= 0
12395/2	= 6197.5	= 1
6197/2	= 3098.5	= 1
3098/2	= 1549	= 0
1549/2	= 774.5	= 1
774/2	= 387	= 0
387/2	= 193.5	= 1
193/2	= 96.5	= 1
96/2	= 48	= 0
48/2	= 24	= 0
24/2	= 12	= 0
12/2	= 6	= 0
6/2	= 3	= 0
3/2	= 1.5	= 1
½	= 0.5	= 1

Reversing the order and placing the "0's" and "1's" into binary we get:

    110000011010110.0000

Next the exponent needs to get normalized by counting how many times the decimal point can be moved to the left. Giving us:

    1.10000011010110 (moved 14 positions)

Since the number is signed, we must add 127 to 14 giving us 141. Which will be turned into binary:

    141 = 10001101

Adding a leading "0" for the sign bit, then placing the exponent, then the rest of the value gives us:

0100 0110 1100 0001 1010 1100

Turning this into hexadecimal we get:

0x46C1AC00

The last two “0’s” are to fill the last two empty positions since the program expects 8 character hex values.

The last part left is to convert the hex value 0x47646800 to a float. Begin by converting to binary beginning with a 0 for the sign bit.

0100 0111 0110 0100 0110 1000 0000 0000

Since the first 8 characters are for the sign bit and exponent, we can convert the following figures to its decimal value.

Exponent = 1000 1110 = 142
Subtract 127 (signed number) = 15
Mantissa = (2^0 + 2^-1 2^-2 + 2^-5 + 2^-9 + 2^-10 +2^-12) = 1.7844238281
Float = 2^15 * 1.7844238281 = 58472.0

We now have the hex value (0x46C1AC00) and float value (58472.0) which we can enter as the password separated by a space.

Result:

Enter the password: 0x46C1AC00 58472.0
Good Job.

Ch3_00_GdbIntro:

Run:

```
gdb ./Ch3_00_GdbIntro
```

Enter into assembly mode by typing “layout asm”

Search for the string compare functions in the main portion of the assembly code. There will be a total of 3.

```
0x4007db <main+170>     callq  0x4005f0 <strcmp@plt>
```

Set a breakpoint at each call:

```
Breakpoint *0x4007db or (breakpoint *main+170)
```

Step through the program and single out the breakpoints that get called multiple times. Those breakpoints can be deleted, leaving the working “strcmp” function call.

Tip: Enter “ctrl l” to clear the screen in gdb

Run the program again, but this time print the values contained in the %RDI and %RSI registers. These are typically the first two registers used to hold values, followed by (%rdx, %rcx, %r8, %r9).

```
x /s $rdi <return>
x /s $rsi <return>
```

You should see that the registers contain the string you entered, and the password that it’s testing against.

Run the program and enter the string obtained from %RSI.

The program will print:

```
Good job
```

Ch3_04_FnPointer:

Run:

```
gdb ./Ch3_04_FnPointer
```

Since we want to see all of the possible functions, display the assembly code by typing “layout asm”.

After having read the instructions, think about which function we want to call to get the “good job” message.

```
0x364c744b <print_good>         push   %rbp
```

The first line of the <print_good> function contains the function address.

Run the program and enter the address of the <print_good> function.

```
Enter the password: 0x364c744b
Good Job.
```

Ch3_04_LinkedList:

After reading the program instructions it looks like we need to pay attention to the registers during the traversal process of the linear linked list.

Run:

```
Gdb ./Ch3_04_LinkedList
```

Run the program to find the source of the segmentation fault. Type the “where” command to display the sources.

```
#0  0x00000000004007e7 in cats ()
#1  0x000000000040079c in ferrets_before ()
#2  0x0000000000400855 in try_command ()
#3  0x000000000040089f in main ()
```

We’re interested in the first one. Set a breakpoint at the address for cats (). You can similarly set breakpoints for the other functions if the first one doesn’t lead anywhere.

```
b *0x00000000004007e7
```

Now we’re going to follow through the traversal. Enter “layout asm” to display the assembly, and type “layout regs” to display the registers. Enter any string as a password when prompted.

Enter the “si” (step into) command and hit <enter> to call it repeatedly. While stepping through the traversal, pay attention to the registers. In particular, we are interested in %rax and %rdx.

```
rax       0x6016a0 6297248                     rbx            0x0      0
rcx       0x7972546563696e 34184178985822574   rdx            0x6016c0 6297280
```

Tip: print the %rax register by typing “x /s 0x6016a0”, looks like we’re close

On the assembly side, we’re interested in two lines:

```
0x4007e7 <cats+72>      mov    0x18(%rax),%rax
0x4007c0 <cats+33>      cmp    %rax,%rdx
```

The first line lets us know that %rax is important, and the fact that we’re comparing %rax to %rdx is also interesting. Notice that the entire time we’re traversing, %rdx never changes. Perhaps that is where the password is stored, since we keep checking to see if %rax is equal to %rdx.

As suggested by the program, lets try %rdx as the input for the password.

```
Enter the password: 0x6016c0
```

Congratulations! You're one step away. Try using eff0cbc7 as the password.

Ch3_05_XorStr:

The important parts of solving this program are finding the original password, and the xor value that will then alter the password to the working version.

Run:

```
objdump -d Ch3_05_XorStr | less
```

Tip:

The command “| less” will display the assembly in a more readable format.

We must locate the main function and the location where the xor is performed. After some searching you should come across:

```
  40067d:       0f b6 80 80 11 60 00    movzbl 0x601180(%rax),%eax
  400684:       83 f0 11                xor    $0x11,%eax
```

You can check what the value holds, or other values that may look interesting by entering the following in gdb:

```
x /s 0x601180 <return>
```

We now have the password before it is altered (qdCcBevE) and the xor scheme (0x11). You can now perform the xor operation on the string to get the resulting password. The easiest way to do this would be by looking up a xor calculator on google.

There will be 2 inputs:

```
1st input (ASCII base 256)        : qdCcBevE
2nd input (hexadecimal base 16)   : 1111111111111111
Result                            : `uRrStgT
```

There are 8 values in the original value. Each of those values will be xor’d with 0x11, giving us 16 11’s (8 * 2). For example, ‘q’ will be xor’d with “0x” ‘11’ and ‘d’ will be xor’d with “0x” ‘11’ and so on for each character.

You are now ready to enter the password!

```
Enter the password: `uRrStgT
Good Job.
```

Ch3_06_SwitchTable:

Let's begin this program by looking at the Ch3_06_SwitchTable.s file which contains all of the assembly code for this program.

Type:

```
Vim Ch3_06_SwitchTable.s
```

Our goal for this program is to get to the function .LC4:

```
.LC4:
  .string "Good Job."
  .text
  .globl  main
  .type   main, @function
```

The main section will give us an understanding of what the program is doing.

```
main:
.LFB3:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movq    %fs:40, %rax
        movq    %rax, -8(%rbp)
        xorl    %eax, %eax
        movl    $0, %eax
        call    print_msg
        movl    $.LC1, %edi
        movl    $0, %eax
        etall   printf
        leaq    -12(%rbp), %rax
        movq    %rax, %rsi
        movl    $.LC2, %edi
        movl    $0, %eax
        call    __isoc99_scanf
        movl    -12(%rbp), %eax
        subl    $29996, %eax
        cmpl    $4, %eax
        ja      .L3
        movl    %eax, %eax
```

```
movq    .L5(,%rax,8), %rax
jmp     *%rax
.section        .rodata
.align 8
.align 4
```

It looks like right before the compare, there is a subtraction of a literal value of (29996). This number represents where the “menu” starts. For example entering ‘29997’ will be the equivalent of entering ‘1’ (29997-29996 = 1), entering 29997 will be equivalent to entering ‘2’ and so on.

Working backwards, we know that we need to get to .LC4, which is accessed only by .L7 (option in switch table), which in turn is accessed through .L5 (switch table).

```
.L5:                * beginning of switch table
  .quad   .L4       * think of as option ‘0’
  .quad   .L6       * option ‘1’
  .quad   .L7       * option ‘2’
  .quad   .L6       * option ‘3’
  .quad   .L4       * default (anything greater than or less than 29996, or ‘0’)
  .text
```

Within .L5, we want .L7 which contains:

```
.L7:
  movl    $.LC4, %edi
  call    puts
  jmp     .L8
```

Numerically, entering ‘29996’ as the value would give us a ‘0’ which takes us to .L4 (default). Similarly, ‘29997’ gives us ‘1’ which takes us to .L6, and finally, ‘29998’ gives us ‘2’ which takes us to .L7 like we want. Since we know that our choice must be “2”, we have to enter ‘29997’ as the password.

Go back to the program and enter:

```
Enter the password: 29998
Good Job.
```

Ch3_07_ParamsRegs:

After reading the instructions from the program, it looks like we need to look for a function that holds 6 parameters. Gdb will allow us to see what’s going on behind the scenes.

Run:

```
gdb ./Ch3_07_ParamsRegs
```

Type “layout asm” to see the assembly code for this program.

```
0x40075c <main+26>      movabs $0x7463654d78756549,%rax
0x400774 <main+50>      movabs $0x5467587a6c373273,%rax
0x40078c <main+74>      movabs $0x796b376550323648,%rax
0x4007a4 <main+98>      movabs $0x586a4c534d307242,%rax
0x4007bc <main+122>     movabs $0x51386f6177374f6b,%rax
0x4007d4 <main+146>     movabs $0x4f436c794f733373,%rax
```

These six lines look promising. Now we need to figure out the function that takes these 6 parameters.
Scrolling down a little bit further, we come across these two lines:

```
0x400891 <main+335>     callq  0x4006a1 <foo>
0x400896 <main+340>     test   %eax,%eax
```

The function call previous to this one was:

```
0x40086a <main+296>     callq  0x400570 <__isoc99_scanf@plt>
```

It looks like we are at a spot in the program where the function <foo> is called after some input is accepted. We should investigate further into the <foo> function.

Set a breakpoint at the beginning of the <foo> function.

```
break *0x4006a1
```

Then run the program and enter a test string. Hit <ctrl L> to clear the screen. And type “layout regs” so we can take a look at the registers (remember Diane’s Silk Dress cost 89 Dollars). We can print out what the registers contain by typing the following command for each register’s hex address.

```
x /s *(hex address)

(gdb) x /s 0x40098c
0x40098c:       "TYdfExPX"
(gdb) x /s 0x400983
0x400983:       "Ys1QXVxq"
```

```
(gdb) x /s 0x40097a
0x40097a:       "uyQMBeoD"
(gdb) x /s 0x400971
0x400971:       "Kr3Pp6kr"
(gdb)
(gdb) x /s 0x7fffffffdd90
0x7fffffffdd90: "MjgyNzNh"
(gdb) x /s 0x7fffffffdda0
0x7fffffffdda0: "test"
(gdb)
```

We can now try all of these strings which look like passwords to see which it may be. Upon closer inspection though, it looks like the registers %r9 and %r8 contain the string typed in earlier at the prompt (test) and the string (MjgyNzNh), respectively. Let’s try that one first because it looks like they are being compared against each other.

Result:

```
Enter the password: MjgyNzNh
Good Job.
```

Ch3_07_SegvBacktrace

Run:

```
gdb Ch3_07_SegvBacktrace
```

Enter “layout asm” and hit <return> so the assembly becomes visible and then run the program by entering “run”. You can then enter any test string and allow the program to return a segmentation fault error:

```
Program received signal SIGSEGV, Segmentation fault.
0x00000000004007e3 in blackberry ()
```

Now that we have received the segmentation fault, we can take a step back and look to see what happened. We are told that the seg fault happened at “0x00000000004007e3 in blackberry ()“. Enter “bt” and hit <return> to display the stack trace listing the function call path:

```
#0  0x00000000004007e3 in blackberry ()
#1  0x00000000004007cd in watermelon ()
#2  0x0000000000400747 in grapefruit ()
#3  0x00000000004006b7 in pineapple ()
#4  0x0000000000400989 in main ()
```

We want to set a breakpoint at the blackberry () function so that we can re-run the program and get some clues as to what is happening.

Enter:

```
b *0x00000000004007e3
```

Now that we have set the breakpoint, we can run the program again (enter y when prompted) and view the registers to see what they hold as parameters. Enter “lay regs” to view the registers more easily at this point as well.

It looks like register %rdi and register %r9 hold some parameters, we can print what they contain:

Run:

```
x /s (address of register) to display what it contains
```

Result:

```
(gdb) x /s 0x400a88
0x400a88:       "OTk4OTQx"
(gdb) x /s 0x400a68
0x400a68:       "grape"
```

The hint is “grape”, if you recall, there is a function called grapefruit(). We also got what looks like a password, you can go ahead and see if it works or if we need to keep going.

```
0x400731 <grapefruit>          push   %rbp
0x400732 <grapefruit+1>        mov    %rsp,%rbp
0x400735 <grapefruit+4>        sub    $0x10,%rsp
0x400739 <grapefruit+8>        mov    %rdi,-0x8(%rbp)
0x40073d <grapefruit+12>       mov    $0x400a4f,%edi
0x400742 <grapefruit+17>       callq  0x4007b7 <watermelon>
0x400747 <grapefruit+22>       leaveq
0x400748 <grapefruit+23>       retq
```

We can print the parameter highlighted in yellow to see what it holds as it is passed to %edi.

Enter:

```
(gdb) x /s 0x400a4f
0x400a4f:       "Y2EwZDgz"
```

We can try this new password and see if it works, or keep going down the trail of function calls.

Result:

```
Enter the password: Y2EwZDgz
Good Job.
```

Ch3_07_HijackPLT

It looks like we need to hijack the <sleep> function, but first we can get some hints as to what to search for by looking at the source file (*PLT.c).

```
printf("Enter the password: ");
scanf("%lx %lx",(unsigned long int *) &ip,&i);
if (ip > (unsigned long int *) 0xff000000) {
printf("Address too high.  Try again.\n");
exit(0);
}
*ip = i;
printf("The address: %lx will now contain %lx\n",(unsigned long int) ip,i);
sleep(1);
```

We need to hijack the <sleep> function's address, and overwrite it with the <print_good> function instead. The first step is to figure out where in the PLT table the <sleep> function is located. We will do an object dump to see the relevant information.

Display program headers:

```
objdump -h ./*PLT | less

23 .got.plt      00000050  0000000059906000  0000000059906000  00106000  2**3
                 CONTENTS, ALLOC, LOAD, DATA
```

We can get the address of the <print_good> function by looking at the assembly code.

Display <print_good> address:

```
Objdump -d ./*PLT | less

0000000059705376 <print_good>:
59705376:   55                    push   %rbp
59705377:   48 89 e5              mov    %rsp,%rbp
5970537a:   bf 58 55 70 59        mov    $0x59705558,%edi
5970537f:   e8 ec b1 cf a6        callq  400570 <puts@plt>
59705384:   bf 00 00 00 00        mov    $0x0,%edi
59705389:   e8 32 b2 cf a6        callq  4005c0 <exit@plt>
```

Scroll down further to see the disassembly of the PLT table so we can figure out the offset needed to access the <sleep> function.

```
00000000004005d0 <sleep@plt>:
4005d0:       ff 25 72 5a 50 59       jmpq   *0x59505a72(%rip)        # 59906048
<_GLOBAL_OFFSET_TABLE_+0x48>
4005d6:       68 06 00 00 00          pushq  $0x6
```

```
4005db:       e9 80 ff ff ff          jmpq   400560 <_init+0x20>
```

The program expects two arguments in hex; the address of the <sleep> function in the PLT table, and the address of the <print_good> function that we want to be called instead. The first hex address will be 0x59906000 + 0x48 giving us 0x5990648 (address of global PLT table + offset for <sleep>). The second hex value is the address of the <print_good> function.

Result:

```
Enter the password: 0x59906048 0x59705376
The address: 59906048 will now contain 59705376
Good Job.
```

Ch3_07_StackSmash

In order to see what's happening as we try to smash the stack, we can go into gdb.

Run:

```
Gdb Ch3_07_StackSmash
```

Enter "layout asm" and hit <return> and "layout regs" and hit return. Since we are going to overflow the buffer, it looks like the function <unsafe_input >to be called:

```
0x6f636b1a <main+34>    callq  0x6f636aad <unsafe_input>
```

We can scroll above <main> and find <unsafe_input> in the assembly code and set a breakpoint at the return line of <unsafe_input> and examine the registers after running the program.

```
 0x6f636ad5 <unsafe_input+40>    callq  0x400580 <__isoc99_scanf@plt>
│0x6f636ada <unsafe_input+45>    nop
│0x6f636adb <unsafe_input+46>    leaveq
│0x6f636adc <unsafe_input+47>    retq
```

Enter:

```
b *0x6f636adc
```

Run the program and enter a long string that will go out of bounds and then work backwards to figure out the size of the buffer by examining the %rbp pointer. Enter "layout regs" to see the registers in gdb for this part.

Test string:

AABBCCDDEEFFGGHHII (in ASCII)

41414242434344444545464647474848 4949 (in hex)

When we examine %rbp (base pointer) we get:

```
rbp            0x7fffff004949   0x7fffff004949
```

It looks like we went two characters past the buffer size. We now know that the buffer size is 16 bits.

We can find the address of <print_good> in the assembly code:

```
0x6f636a76 <print_good>         push   %rbp
│0x6f636a77 <print_good+1>       mov    %rsp,%rbp
│0x6f636a7a <print_good+4>       mov    $0x6f636bb8,%edi
│0x6f636a7f <print_good+9>       callq  0x400540 <puts@plt>
│0x6f636a84 <print_good+14>      mov    $0x0,%edi
```

Because we are working in 'little endian' we need to enter the address of print good in the following order with the least significant bit first up through the most significant bit:

```
76 6a 63 6f
```

As explained in the program instructions, thie ASCII representation of the <print_good> function (google hex to ascii converter) must be entered as part of the password.

```
766a636f =>  vjco
```

We now have:

```
AABBCCDDEEFFGGHHvjco
```

Highlighted in blue is the portion that fills the buffer, and in green the address (in ascii) of <print_good>.

We are very close to being done, but take a look at <unsafe_input> in gdb again:

```
0x6f636ad5 <unsafe_input+40>    callq  0x400580 <__isoc99_scanf@plt>
0x6f636ada <unsafe_input+45>    nop
0x6f636adb <unsafe_input+46>    leaveq
0x6f636adc <unsafe_input+47>    retq
```

We need to return the address of <print_good> (highlighted in green) but we encounter the command 'leaveq' before we are able to. You can do some searching on the man page or google and see that it pops an address off of the stack frame. To bypass this, we need to give it data that we don't care about (like filling the buffer) and then finally concatenating the address to <print_good> in ascii.

We now have:

```
AABBCCDDEEFFGGHHaaaaaaaavjco
```

The blue portion takes care of the buffer, the orange takes care of the command 'leaveq', and the green portion is the address we want returned.

Verify in gdb:

```
(gdb) x/8xg $rsp
0x7fffffffead0: 0x4444434342424141      0x4848474746464545
```

```
0x7fffffffeae0: 0x6161616161616161      0x000000006f636a76
0x7fffffffeaf0: 0x000000006f636b30      0x00007ffff7a2d830
0x7fffffffeb00: 0x0000000000000000      0x00007fffffffebd8
```

```
It looks like the stack pointer (%rsp) is set in the correct order.
```

```
Run the program and enter the string:
```

```
Enter the password: AABBCCDDEEFFGGHHaaaaaaaavjco
Good Job.
```

Ch3_07_CanaryBypass

Take a look at the C source file to get an idea of what will happen as the program runs. The two functions are highlighted in orange, and the code that concerns us in yellow.

```
void prompt_user() {
    char buffer[40];
    int offset;
    char *user_addr;
    char **over_addr;
    printf("Enter the password: ");
    scanf("%d %lx", &offset, (unsigned long *) &user_addr);
    over_addr = (char **) (buffer + offset);
    *over_addr = user_addr;
}

int main(int argc, char *argv[]) {
    print_msg();
    prompt_user();
    printf("Try again.\n");
    return 0;
}
```

Run:

```
Gdb Ch3_07_CanaryBypass
```

Enter “layout asm” and hit <return> so we can look at the assembly. We need to figure out the offset so that we can supply the address of the <print_good> function instead. As well as the address of the <print_good> function.

```
 0x4006f4 <prompt_user+59>      callq  0x400560 <__isoc99_scanf@plt>
│0x4006f9 <prompt_user+64>      mov    -0x44(%rbp),%eax
│0x4006fc <prompt_user+67>      cltq
│0x4006fe <prompt_user+69>      lea    -0x30(%rbp),%rdx
│0x400702 <prompt_user+73>      add    %rdx,%rax
│0x400705 <prompt_user+76>      mov    %rax,-0x38(%rbp)
│0x400709 <prompt_user+80>      mov    -0x40(%rbp),%rdx
│0x40070d <prompt_user+84>      mov    -0x38(%rbp),%rax
│0x400711 <prompt_user+88>      mov    %rdx,(%rax)
```

```
 0x400686 <print_good>          push   %rbp
│0x400687 <print_good+1>        mov    %rsp,%rbp
│0x40068a <print_good+4>        mov    $0x4007e4,%edi
│0x40068f <print_good+9>        callq  0x400520 <puts@plt>
│0x400694 <print_good+14>       mov    $0x0,%edi
```

```
│0x400699 <print_good+19>        callq  0x400570 <exit@plt>
```

There are several offsets and each can be tried as a decimal number (trial and error) along with the address of the <print_good> function. A good way to check if the end of the stack has been reached is to set a breakpoint at the return of the <prompt_user> function (found in gdb or objectdump) and displaying the stack pointer.

Let’s use 0x38 (convert to decimal) as the offset since it goes into rax which seems appropriate. Run the program and enter “56 0x400686” as the password. Now we can see where the stack pointer is pointing.

Run:

```
b *0x40072a
x/8xw $rsp
```

```
0x7fffffffeac8: 0x0040072a      0x00000000      0xffffebc8      0x00007fff
0x7fffffffead8: 0x00000000      0x00000001      0x00400760      0x00000000
```

It looks like we are at the end of the stack which is exactly what we want. Looks like we have the correct offset so we can now call <print_good>.

Result:

```
Enter the password: 56 0x400686
Good Job.
```

Ch3_07_StaticStrcmp

We are told that the password is stored statically, meaning it is held in a register until it is tested against the string typed in at the prompt.

Run:

```
gdb Ch3_07_StaticStrcmp
```

Enter “layout asm” and hit <return> and enter “layout regs” and hit <return> so that we can see the assembly code and see the contents of the registers. We know that a ‘strcmp’ function will be called with a ‘callq’ command. Pressing down on the keyboard will allow us to scroll down the contents of main while searching for the ‘strcmp’ call.

```
0x400a54 <main+139>     callq  0x40f510 <__isoc99_scanf>
0x400a59 <main+144>     lea    -0x20(%rbp),%rdx
0x400a5d <main+148>     lea    -0x40(%rbp),%rax
0x400a61 <main+152>     mov    %rdx,%rsi
0x400a64 <main+155>     mov    %rax,%rdi
0x400a67 <main+158>     callq  0x400360
```

It looks like strcmp isn’t called by name anywhere, since the name of the function <name> usually follows the call. The next best thing is highlighted in yellow. It looks like a call to a function is made but the name isn’t supplied. This function looks promising because a ‘scanf’ function is called just before it, taking in user input. We can set a breakpoint at this address, and follow the code when we run the program.

Set a breakpoint:

```
b *0x400360
```

Now let’s run the program by typing “run” and hitting <return>. Since we are inside the unnamed function which we believe to be the ‘strcmp’ function, we can print the contents of the registers most often used for temp values. They are (rdi, rsi, rdx, rcx, r8, r9) and the contents of the registers can be displayed with ‘x /s’ as before.

Display contents of registers:

```
(gdb) x /s 0x7fffffffea50
0x7fffffffea50: "test"
(gdb) x /s 0x7fffffffea70
0x7fffffffea70: "TEAndIUe"
```

After displaying the first two registers, we can already see the ‘test’ string entered at the prompt, as well as the string it is being compared to. Try entering this string as the answer.

Result:

```
Enter the password: TEAndIUe
Good Job.
```

Ch3_08_Matrix

You are given an array:

```
2D Array:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
 15  16  17  18  19  20  21  22  23  24  25  26  27  28  29
 30  31  32  33  34  35  36  37  38  39  40  41  42  43  44
 45  46  47  48  49  50  51  52  53  54  55  56  57  58  59
             ...
             ...
705 706 707 708 709 710 711 712 713 714 715 716 717 718 719
720 721 722 723 724 725 726 727 728 729 730 731 732 733 734
735 736 737 738 739 740 741 742 743 744 745 746 747 748 749
```

The program provides a location where the password is stored. In this case:

```
Password = [11][54]
```

If you look at the array above, you can see that the first element starts at zero, then each successive row increases by 15 at the 1st position. Each column then, is 15 positions long.

The password is at the 11th row, and 54th position in the column. Since we know the maximum size of the rows (15 column positions), the final position would be 11 full rows plus an extra 54 positions.

Compute:

```
(11 * 15) + 54 = 219
```

Run the program and enter the value:

```
Enter the password: 219
Good Job.
```

Ch5_08_LoopUnroll

Run:

```
objdump -d *./Unroll | less
```

Tip:

“ | less” pipes the display of the file into an easier format for us to view

As instructed in the program description, search the “main” portion of the file for three function calls. They will look something like:

```
400920:    e8 5f fe ff ff        callq  400784 <unroll3>
400925:    89 c3                 mov    %eax,%ebx
400927:    48 8d 7c 24 10        lea    0x10(%rsp),%rdi
40092c:    e8 03 ff ff ff        callq  400834 <unroll7>
400931:    89 c5                 mov    %eax,%ebp
400933:    48 8d 7c 24 10        lea    0x10(%rsp),%rdi
400938:    e8 2d ff ff ff        callq  40086a <unroll8>
```

Knowing the names and addresses of these three functions, we can unroll each of their loops.

We will examine <unroll3> to figure out the process, which can then be repeated for <unroll7> and <unroll8>.

```
0000000000400784 <unroll3>:
400784:    b8 00 00 00 00        mov    $0x0,%eax
400789:    b9 00 00 00 00        mov    $0x0,%ecx
40078e:    eb 13                 jmp    4007a3 <unroll3+0x1f>
400790:    48 63 f1              movslq %ecx,%rsi
400793:    8b 54 b7 04           mov    0x4(%rdi,%rsi,4),%edx
400797:    03 14 b7              add    (%rdi,%rsi,4),%edx
40079a:    03 54 b7 08           add    0x8(%rdi,%rsi,4),%edx
40079e:    01 d0                 add    %edx,%eax
4007a0:    83 c1 03              add    $0x3,%ecx
4007a3:    83 f9 17              cmp    $0x17,%ecx
4007a6:    7e e8                 jle    400790 <unroll3+0xc>
4007a8:    f3 c3                 repz retq
```

Since we are unrolling a loop, we must know the stopping condition for said loop. The stopping condition is always tested by **comparing** the current condition/value to the stopping condition/value. The cmp call highlited in red will stop the loop. The “$” tells us that this is an integer value represented in hex, which in decimal equals to 23. This tells us that we will stop when equal or greater to the index 23 (nums[23]).

The add call highlighted in yellow tells us that the value ‘3’ is added to %ecx on each pass. %ecx starts at ‘0’ and represents the index (set at green highlight)

Apply Gauss sum theororem:

```
Sum = 1+2+3+4+5...n = n(n+1)/2
```

For <unroll3>:

```
Sum = 23/2 * (24) = 276
```

We can apply the same procedure for the final two functions and then enter the sums separated by spaces as the password.

Result:

```
Enter the password: 276 595 2556
Good Job.
```

Ch7_13_LdPreloadGetUID

Run the program and read the instructions carefully. It looks like we can input any string and get a hint as to how to proceed after program failure.

```
Enter the password: test
If you run this program with the UID of 270954, the password will be on the
nextline.

Error: UID 10657 ran us. We expected UID 270954.
Try again.
```

In this exercise, we will write our own ‘getuid()’ function and preload it dynamically (at run-time) to provide the UID that will give us the password. Create a C source file so we can write our version of the ‘getuid()’ function and enforce the program to use our version.

Create C source file (I called it uid.c, you can call it anything you want):

```
vim uid.c
```

Since we want to provide the program with the UID of 270954, create a simple function to return that number. Make sure to call it “getuid”:

```
int getuid()
{
    return 270954;
}
```

Now “uid.c” must be compiled into a shared object file (.so):

```
META$ gcc -shared -fPIC uid.c -o uid.so
```

And we must specify that we want to preload this program dynamically.

```
META$ LD_PRELOAD=$PWD/uid.so ./Ch7_13_LdPreloadGetUID
```

Now that we know our version of ‘getuid()’ will be run, returning the correct UID number, run the program again and enter any string.

```
Enter the password: test
If you run this program with the UID of 270954, the password will be on the
nextline.

Hint: ngYkUyZTp
```

```
Try again.
```

We are one step closer because we have been provided the password as a hint. Run the program again and enter the password:

```
Enter the password: ngYkUyZTp
If you run this program with the UID of 270954, the password will be on the
nextline.

Error: UID 10657 ran us. We expected UID 270954.
Good Job.
```

Ch8_05_Signals

Since the password is not known, we need to send a “SIGSTOP” command to the terminal.

Press the following keys on the keyboard:

```
<ctrl z>
```

The program is now running in the background. Bring it back to the foreground by running the following command:

```
fg
```

The program is now back in the foreground where we can communicate with it again.

Open a new terminal and run it side by side with the current terminal which is running the executable “Ch5_08_Signals”. On the new terminal type the following command:

```
ps -al
```

```
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
1 R 10657  3023     1 99  80   0 -  1088 -      pts/10   00:10:12 Ch8_05_Signals
0 S 12777  3131 31487  0  80   0 - 77073 -      pts/11   00:00:00 vim
0 S 10657  3680 28448  0  80   0 -  1088 wait_w pts/10   00:00:00 Ch8_05_Signals
1 R 10657  3681  3680 96  80   0 -  1088 -      pts/10   00:00:06 Ch8_05_Signals
0 R 10657  3686  2341  0  80   0 -  7302 -      pts/7    00:00:00 ps
0 S 10410 11167 11150  0  80   0 - 60317 -      pts/36   00:04:50 weechat
0 S 12574 15635 13497  0  80   0 - 14582 -      pts/29   00:00:00 ssh
0 S 11442 19020 15710  0  80   0 - 77053 -      pts/3    00:00:00 vim
0 S 12790 20264 20237  0  80   0 - 77056 -      pts/0    00:00:03 vim
```

The child (red) PID can be distinguished from the parent (green) PID because the child PID will be higher numerically. Run the following command:

```
kill 3681
```

In the terminal not running the program, run the command:

```
man kill
```

```
OPTIONS
       <pid> [...]
              Send signal to every <pid> listed.

       -<signal>
       -s <signal>
       --signal <signal>
```

```
              Specify the signal to be sent.  The signal can be specified by using
              name  or  number.  The behavior of signals is explained in signal(7)
              manual page.

       -l, --list [signal]
              List signal names.  This option has optional  argument,  which  will
              convert signal number to signal name, or other way round.
```

As you saw in the man page, a signal can be sent numerically or spelled out. You can find the numeric representations of different kill signals by running the “kill -l” command. We need to send a ‘SIGWINCH command so run the following:

```
ps -al

F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
1 R 10657  3023     1 99  80   0 -  1088 -      pts/10   00:24:52 Ch8_05_Signals
0 S 10657  3680 28448  0  80   0 -  1088 wait_w pts/10   00:00:00 Ch8_05_Signals
0 S 12777  4369 31487  0  80   0 - 77071 -      pts/11   00:00:01 vim
0 R 10657  4613  2341  0  80   0 -  7302 -      pts/7    00:00:00 ps
0 S 10410 11167 11150  0  80   0 - 60317 -      pts/36   00:04:50 weechat
0 S 12574 15635 13497  0  80   0 - 14582 -      pts/29   00:00:00 ssh
0 S 12790 20264 20237  0  80   0 - 77056 -      pts/0    00:00:03 vim

kill -SIGWINCH 3680
```

The SIGPWR signal can be sent in a similar manner;

```
kill -SIGPWR 3680
```

Finally enter <ctrl c> on your keyboard to finish the program and get the password:

```
^CTry YmRkNjZ
```

Run the program again and enter the password to finish.

```
Enter the password: YmRkNjZ
Good Job.
```

Ch8_05_PsSignals

Run the program, then open a new terminal as the instructions recommend. You will be the terminal you just opened to look up codes and send 'kill' signals.

Tip:

Make sure you are on the same host. So, if you're doing these on quizor both terminals must be logged in to quizor (or wherever you are running the terminal).

We are interested in finding the child using the most memory. Use the "man ps" command to see the modifiers available to us so we can use the 'ps' command to find what we need.

On the second terminal (the one not running Ch5_08_PsSignals) run the following command to get the PID's with the '-al' modifier to display a wider range of information for all running processes:

```
ps -al
```

```
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
  1 R 10657  3023     1 98  80   0 -  1088 -      pts/10   00:55:41 Ch8_05_Signals
     0 S 10657  7757 28448  0  80   0 -  1089 wait_w pts/10   00:00:00
Ch8_05_PsSignal
1 S 10657  7758  7757  0  80   0 -  1089 pause  pts/10   00:00:00 Ch8_05_PsSignal
1 S 10657  7759  7757  0  80   0 -  1089 pause  pts/10   00:00:00 Ch8_05_PsSignal
1 S 10657  7760  7757  0  80   0 -  1089 pause  pts/10   00:00:00 Ch8_05_PsSignal
1 S 10657  7761  7757  0  80   0 -  1089 pause  pts/10   00:00:00 Ch8_05_PsSignal
1 S 10657  7762  7757  0  80   0 -  1089 pause  pts/10   00:00:00 Ch8_05_PsSignal
1 S 10657  7763  7757  0  80   0 -  1089 pause  pts/10   00:00:00 Ch8_05_PsSignal
1 S 10657  7764  7757  0  80   0 -  1089 pause  pts/10   00:00:00 Ch8_05_PsSignal
1 S 10657  7765  7757  0  80   0 -  1089 pause  pts/10   00:00:00 Ch8_05_PsSignal
1 S 10657  7766  7757  0  80   0 -  1089 pause  pts/10   00:00:00 Ch8_05_PsSignal
1 S 10657  7767  7757  0  80   0 -  1089 pause  pts/10   00:00:00 Ch8_05_PsSignal
1 S 10657  7768  7757  0  99  19 -  1089 pause  pts/10   00:00:00 Ch8_05_PsSignal
1 R 10657  7769  7757 98  80   0 -  1089 -      pts/10   00:07:22 Ch8_05_PsSignal
1 S 10657  7770  7757  0  80   0 -  3043 pause  pts/10   00:00:00 Ch8_05_PsSignal
1 T 10657  7771  7757  0  80   0 -  1089 signal pts/10   00:00:00 Ch8_05_PsSignal
```

It looks like PID 7770 is taking up the most memory (SZ) category. Send the kill signal:

```
kill -SIGALRM 7770
```

Now we need to send the kill signal to the lowest nice (NI) level

```
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
1 R 10657  3023     1 98  80   0 -  1088 -      pts/10   01:02:18 Ch8_05_Signals
0 S 10657  7757 28448  0  80   0 -  1089 wait_w pts/10   00:00:00 Ch8_05_PsSignal
1 S 10657  7758  7757  0  80   0 -  1089 pause  pts/10   00:00:00 Ch8_05_PsSignal
```

```
1 S 10657  7759  7757  0  80   0 -  1089 pause  pts/10   00:00:00 Ch8_05_PsSignal
1 S 10657  7760  7757  0  80   0 -  1089 pause  pts/10   00:00:00 Ch8_05_PsSignal
1 S 10657  7761  7757  0  80   0 -  1089 pause  pts/10   00:00:00 Ch8_05_PsSignal
1 S 10657  7762  7757  0  80   0 -  1089 pause  pts/10   00:00:00 Ch8_05_PsSignal
1 S 10657  7763  7757  0  80   0 -  1089 pause  pts/10   00:00:00 Ch8_05_PsSignal
1 S 10657  7764  7757  0  80   0 -  1089 pause  pts/10   00:00:00 Ch8_05_PsSignal
1 S 10657  7765  7757  0  80   0 -  1089 pause  pts/10   00:00:00 Ch8_05_PsSignal
1 S 10657  7766  7757  0  80   0 -  1089 pause  pts/10   00:00:00 Ch8_05_PsSignal
1 S 10657  7767  7757  0  80   0 -  1089 pause  pts/10   00:00:00 Ch8_05_PsSignal
1 S 10657  7768  7757  0  99  19 -  1089 pause  pts/10   00:00:00 Ch8_05_PsSignal
1 R 10657  7769  7757 98  80   0 -  1089 -      pts/10   00:13:59 Ch8_05_PsSignal
1 T 10657  7771  7757  0  80   0 -  1089 signal pts/10   00:00:00 Ch8_05_PsSignal
```

```
kill -SIGALRM 7768
```

The next signal needs to be sent to the child using the most cpu cycles. To look up ps modifiers run the "man ps" command. You can pick out modifiers that will suit different needs.

Run:

```
ps -ux
```

```
     PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
    2340  0.0  0.0 140276  4312 ?        S    Jun07   0:02 sshd: yuriy@pts/7
    2341  0.0  0.0  34020  4020 pts/7    Ss   Jun07   0:00 -bash
    3023 98.3  0.0   4352    80 pts/10   R    Jun07  78:35 ./Ch8_05_Signals
    7757  0.0  0.0   4356   660 pts/10   S+   Jun07   0:00 ./Ch8_05_PsSignals
    7758  0.0  0.0   4356    80 pts/10   S+   Jun07   0:00 ./Ch8_05_PsSignals
    7759  0.0  0.0   4356    80 pts/10   S+   Jun07   0:00 ./Ch8_05_PsSignals
    7760  0.0  0.0   4356    80 pts/10   S+   Jun07   0:00 ./Ch8_05_PsSignals
    7761  0.0  0.0   4356    80 pts/10   S+   Jun07   0:00 ./Ch8_05_PsSignals
    7762  0.0  0.0   4356    80 pts/10   S+   Jun07   0:00 ./Ch8_05_PsSignals
    7763  0.0  0.0   4356    80 pts/10   S+   Jun07   0:00 ./Ch8_05_PsSignals
    7764  0.0  0.0   4356    80 pts/10   S+   Jun07   0:00 ./Ch8_05_PsSignals
    7765  0.0  0.0   4356    80 pts/10   S+   Jun07   0:00 ./Ch8_05_PsSignals
    7766  0.0  0.0   4356    80 pts/10   S+   Jun07   0:00 ./Ch8_05_PsSignals
    7767  0.0  0.0   4356    80 pts/10   S+   Jun07   0:00 ./Ch8_05_PsSignals
    7769 98.8  0.0   4356    80 pts/10   R+   Jun07  30:15 ./Ch8_05_PsSignals
    7771  0.0  0.0   4356    80 pts/10   T+   Jun07   0:00 ./Ch8_05_PsSignals
```

```
kill -SIGALRM 7769
```

Lastly, we need to send a kill signal to the sleeping child process. You can search the man page under the 'process state codes' to determine that 'T' represents a stopped process. The 'ps -ux' command will provide the necessary information once again (highlighted in green above).

Run:

```
kill -SIGCONT 7771

The password is: M2U4NDg2
```

Run the program again and enter the password.

```
Enter the password: M2U4NDg2
Good Job.
```