

Project #1 – Reverse Engineering

In this project, you will acquire hands-on experience in practical reverse engineering, developing a comprehensive understanding of ELF binary programs, their execution, and their interaction with the underlying software system.

You will be given several ELF binary files, which require a user to enter a password. Your primary task is to crack their passwords by performing reverse engineering on each binary.

Instructions

- Given that all binaries were built on a Thayer's Linux server, it is highly recommended to use one of these machines for this project (e.g., *babylon1.thayer.dartmouth.edu*). While alternative preferences are allowed, grading will be based on results obtained from Thayer's Linux machines.
- You will be given four binary files, *bin0*, *bin0-strip*, *bin1* and *bin2*, along with *bin0.c*. Each of these files requires a user to enter a correct password, but each binary file may have a different password and a distinct mechanism to validate it.
- You are NOT allowed to modify any of the files provided.
- For reverse engineering tools, it is recommended to use GNU debugging tools, such as *readelf*, *gdb*, and *objdump* if you do not have any preference.
- You might consider writing a python (or shell) script to automate your testing.
- Regardless of its difficulty, it may take some time. Start early!

(50 pts) Task 1

For the first task, you will examine a binary program, *bin0*, which was compiled from the source code *bin0.c*. Please answer the following questions.

1. (3 pts) Is the binary statically or dynamically linked? If it is statically linked, what is the size of the binary file? If dynamically linked, please list all externally used libraries.

- Using the ``file`` command followed by ``bin0``, we know that the binary file, `bin0`, is dynamically linked.
- Using the ``ldd`` command followed by ``bin0``, we know that all the externally used libraries in `bin0` are
 - `linux-vdso.so.1 (0x00007fff983b9000)`
 - `libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f6b13314000)`
- And the dynamic linker/loader is
 - `/lib64/ld-linux-x86-64.so.2 (0x00007f6b13570000)`

2. (3 pts) List all the sections that are merged into a code segment (having an “execute” permission).

- Using the ``objdump -h`` command followed by ``bin0``, it lists out all the different sections in the binary file with their properties. We are interested in the sections that have the ``CODE`` property because these are the ones that are merged into a code segment (i.e. having an “execute” permission). These sections are
 - `.init`
 - `.plt`
 - `.text`
 - `.fini`

3. (4 pts) What are the addresses of the program’s entry point *before* and *after* execution?

- The address of the program’s entry point before execution is `0x401060` and it’s the same address after execution as well.
- I used the ``info files`` command in `gdb` to inspect the entry point of the program before and after execution.
- I run `gdb` on `bin0`, and before running the program, I use the ``info files`` command, and it states that the address of the entry point is `0x401060`.

- I then run the program, and after the end of its execution, I use the `info files` command again, and the address of the entry point is still 0x401060.

4. (15 pts) Draw the layout of the stack frame corresponding to the function *check_password()* after its function prologue is done. For each element (e.g., local variables, function return address) on the stack, provide its address and size.

- In gdb, I set a breakpoint at the address of `check_passwd`. Then, after running the program, it stops at the `check_passwd` breakpoint.
- Afterwards, I used the `info frame` command to inspect some frame information. I got the following information:
 - The stack frame for `check_passwd` is at 0x7fffffffce80
 - `rip = 0x40114a` in `check_passwd`; saved `rip = 0x4011ee`
 - called by frame at 0x7fffffffcea0
 - Arglist at 0x7fffffffce70, args:
 - Locals at 0x7fffffffce70, Previous frame's `sp` is 0x7fffffffce80
 - Saved registers:
 - Saved `rbp` at 0x7fffffffce70, saved `rip` at 0x7fffffffce78
- Now, I will disassemble the `check_passwd` frame to examine the stack frame further. The assembly code for `check_passwd` verifies our instinct, given the source code, that there are no local variables in the `check_passwd` stack frame because no values are being stored within a negative offset from `%ebp`.
- Putting all the information we acquired together, we get the following stack frame layout for `check_passwd`:

addresses	values
⋮	
0x f f f f f f f f f f	
⋮	
	previous frames
rbp → 0x 7 f f f f f f f c e 8 0	return address = 0x 4 0 1 1 e e
0x 7 f f f f f f f c e 7 8	saved rip = 0x 4 0 1 1 e e
rsp → 0x 7 f f f f f f f c e 7 0	saved rbp = 0x 7 f f f f f f f c e a 0
⋮	
0x 0 0 0 0 0 0 0 0 0 0	

% rbp = 0x 7 f f f f f f f c e 8 0
 % rsp = 0x 7 f f f f f f f c e 7 0
 % rip = 0x 4 0 1 1 4 a

- Now, the size of the function return address is 12 bytes long, the saved rip is also 12 bytes long, and the saved rbp is 24 bytes long.

5. (15 pts) In which segment(s) and address(es) would the password be located during program execution? (segment: heap, code, data, etc)

- It's stored in the code segment at the following addresses
 - 0x0000000000401183
 - This address has the character `T` stored in.
 - 0x000000000040118e
 - This address has the character `e` stored in.
 - 0x0000000000401199
 - This address has the character `S` stored in.
 - 0x00000000004011a4
 - This address has the character `t` stored in.

6. (10 pts) *bin0-strip* is a binary program compiled from the same source code, but with an additional `gcc -s` option. What's the difference between compiling with and without this option? Please describe as detail as possible using screenshot(s).

- Compiling a source code without the `-s` flag will produce a non-stripped ELF binary file. On the other hand, compiling a source code with the `-s` flag will produce a stripped ELF binary file.
- The main difference between a stripped ELF binary file and a non-stripped one is that stripped binaries don't preserve debugging symbols when compiled because the `-s` flag will tell the compiler when compiling to discard these debugging symbols, which are not needed for program execution. Unlike stripped binaries, non-stripped binaries preserve these debugging symbols when compiling. Some examples of debugging symbols include function names, variable names, line numbers, type information, stack frames, function parameters, and other pieces of information.
- Here are a few main consequences for this difference in terms of debugging symbols when it comes to ELF binary files.
 - a. Stripping a binary reduces its size on the disk compared to a non-stripped one.

- b. Stripping a binary file makes it a little more difficult to debug and reverse engineer compared to non-stripped ones because of the lack of the debugging symbols that gdb uses for example.
- c. Stripping a binary file makes it harder, while not impossible, to read and analyze the binary file at hand and understand the logic behind it because of the lack of the debugging symbols.

(50 pts) Task 2

In this task, you will reverse engineer the two binary programs, *bin1*, and *bin2*, with no source code provided. Similar to Task 1, your goal is to crack the password for each binary. Note that the passwords can be located in any of the memory segments at runtime, such as stack segment, and the size of the password in each binary is variable.

1. (10 pts) What is the correct password for *bin1*?

- The correct password for bin1 is `PO_y0u_l1Qe_Kr4bbY_pAt71ez?`.

2. (15 pts) Describe your strategy to crack the password of *bin1*. To receive full credit, explain how you determined the password. Explain all the steps. (screenshots may help.)

- Since bin1 is a stripped ELF binary, gdb has no idea where the address of the `main` function is when asked to disassemble the binary.

The screenshot shows a VS Code editor with a C file named `bin0.c` open. The code is a simple program that prompts for a password. The GDB terminal at the bottom displays the output of the `(gdb) info file` command, showing the ELF section table for the executable `bin1`. The table lists various sections such as `.interp`, `.note.gnu.property`, `.note.gnu.build-id`, `.note.ABI-tag`, `.gnu.hash`, `.dynsym`, `.dynstr`, `.gnu.version`, `.gnu.version_r`, `.rela.dyn`, `.rela.plt`, `.init`, `.plt`, `.text`, `.fini`, `.rodata`, `.eh_frame_hdr`, `.eh_frame`, `.init_array`, `.fini_array`, `.dynamic`, `.got`, `.got.plt`, `.data`, and `.bss`.

```
project-1 > C bin0.c > ...
1 #include <stdio.h>
2 #include <string.h>
3
4 char inp[20];
5
6 void check_passwd()
7 {
8     printf("Enter the password: ");
9 }

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
gdb - project-1 + - - - - -

(No debugging symbols found in bin1)
(gdb) info file
Symbols from "/thayerfs/home/f004s8n/cs69.17/project-1/bin1".
Local exec file:
  /thayerfs/home/f004s8n/cs69.17/project-1/bin1', file type elf64-x86-64.
Entry point: 0x4010f0
0x0000000000400318 - 0x0000000000400334 is .interp
0x0000000000400338 - 0x0000000000400358 is .note.gnu.property
0x0000000000400358 - 0x000000000040037c is .note.gnu.build-id
0x000000000040037c - 0x000000000040039c is .note.ABI-tag
0x00000000004003a0 - 0x00000000004003bc is .gnu.hash
0x00000000004003c0 - 0x0000000000400468 is .dynsym
0x0000000000400468 - 0x00000000004004e9 is .dynstr
0x00000000004004ea - 0x00000000004004f8 is .gnu.version
0x00000000004004f8 - 0x0000000000400548 is .gnu.version_r
0x0000000000400548 - 0x0000000000400578 is .rela.dyn
0x0000000000400578 - 0x00000000004005d8 is .rela.plt
0x0000000000401000 - 0x000000000040101b is .init
0x0000000000401020 - 0x0000000000401070 is .plt
0x0000000000401070 - 0x00000000004011d6 is .text
0x00000000004011d8 - 0x00000000004011e5 is .fini
0x0000000000402000 - 0x0000000000402045 is .rodata
0x0000000000402048 - 0x0000000000402074 is .eh_frame_hdr
0x0000000000402078 - 0x0000000000402104 is .eh_frame
0x0000000000403e10 - 0x0000000000403e18 is .init_array
0x0000000000403e18 - 0x0000000000403e20 is .fini_array
0x0000000000403e20 - 0x0000000000403ff0 is .dynamic
0x0000000000403ff0 - 0x0000000000404000 is .got
0x0000000000404000 - 0x0000000000404038 is .got.plt
0x0000000000404040 - 0x00000000004040de is .data
0x00000000004040e0 - 0x0000000000404180 is .bss

(gdb) disass
No frame selected.
(gdb) disass main
No symbol table is loaded. Use the "file" command.
(gdb)
```

- Therefore, we have to figure out ourselves where the code starts and set a breakpoint at that address.
- To do that, I use the command ``info files`` in gdb to browse over some high level, but important, information about bin1 and the different sections in it.

The screenshot shows a VS Code editor with a C file named `bin0.c` open. The code is a simple password checker. The terminal window shows the output of the `gdb -info` command, displaying the ELF symbol table for the executable `bin1`. The symbol table lists various sections of the executable, including `.text`, which is the section containing the main function. The address of the start of the `.text` section is `0x00000000401070`.

```
project-1 > C bin0.c > ...
1 #include <stdio.h>
2 #include <string.h>
3
4 char inp[20];
5
6 void check_passwd()
7 {
8     printf("Enter the password: ");
9 }

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
gdb - project-1 + - - - - -

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from bin1...
(gdb) info file
Symbols from "/thayerfs/home/f004s8n/cs69.17/project-1/bin1".
Local exec file:
'/thayerfs/home/f004s8n/cs69.17/project-1/bin1', file type elf64-x86-64.
Entry point: 0x4010f0
0x0000000000400318 - 0x0000000000400334 is .interp
0x0000000000400338 - 0x0000000000400358 is .note.gnu.property
0x0000000000400358 - 0x000000000040037c is .note.gnu.build-id
0x000000000040037c - 0x000000000040039c is .note.ABI-tag
0x00000000004003a0 - 0x00000000004003bc is .gnu.hash
0x00000000004003c0 - 0x0000000000400468 is .dynsym
0x0000000000400468 - 0x00000000004004e9 is .dynstr
0x00000000004004ea - 0x00000000004004f8 is .gnu.version
0x00000000004004f8 - 0x0000000000400548 is .gnu.version_r
0x0000000000400548 - 0x0000000000400578 is .rela.dyn
0x0000000000400578 - 0x00000000004005d8 is .rela.plt
0x0000000000401000 - 0x000000000040101b is .init
0x0000000000401020 - 0x0000000000401070 is .plt
0x0000000000401070 - 0x00000000004011d6 is .text
0x00000000004011d8 - 0x00000000004011e5 is .finl
0x0000000000402000 - 0x0000000000402045 is .rodata
0x0000000000402048 - 0x0000000000402074 is .eh_frame_hdr
0x0000000000402078 - 0x0000000000402164 is .eh_frame
0x0000000000403e10 - 0x0000000000403e18 is .init_array
0x0000000000403e18 - 0x0000000000403e20 is .fini_array
0x0000000000403e20 - 0x0000000000403ff0 is .dynamic
0x0000000000403ff0 - 0x0000000000404000 is .got
0x0000000000404000 - 0x0000000000404038 is .got.plt
0x0000000000404040 - 0x00000000004040de is .data
0x00000000004040e0 - 0x0000000000404180 is .bss
```

- When the `info files` command is run in `gdb`, it shows where the `.text` section starts and ends, that is, the addresses of which the executable code starts and ends. The beginning of this `.text` section is of interest to us because this is where the `main` function lies — or maybe within a few higher addresses.
- We then set a breakpoint at the beginning of the `.text` section, in our case, that would be the address `0x0000000000401070`.

The screenshot shows a VS Code editor with a C file named `bin0.c` open. The code is as follows:

```
1 #include <stdio.h>
2 #include <string.h>
3
4 char inp[20];
5
6 void check_passwd()
7 {
8     printf("Enter the password: ");
```

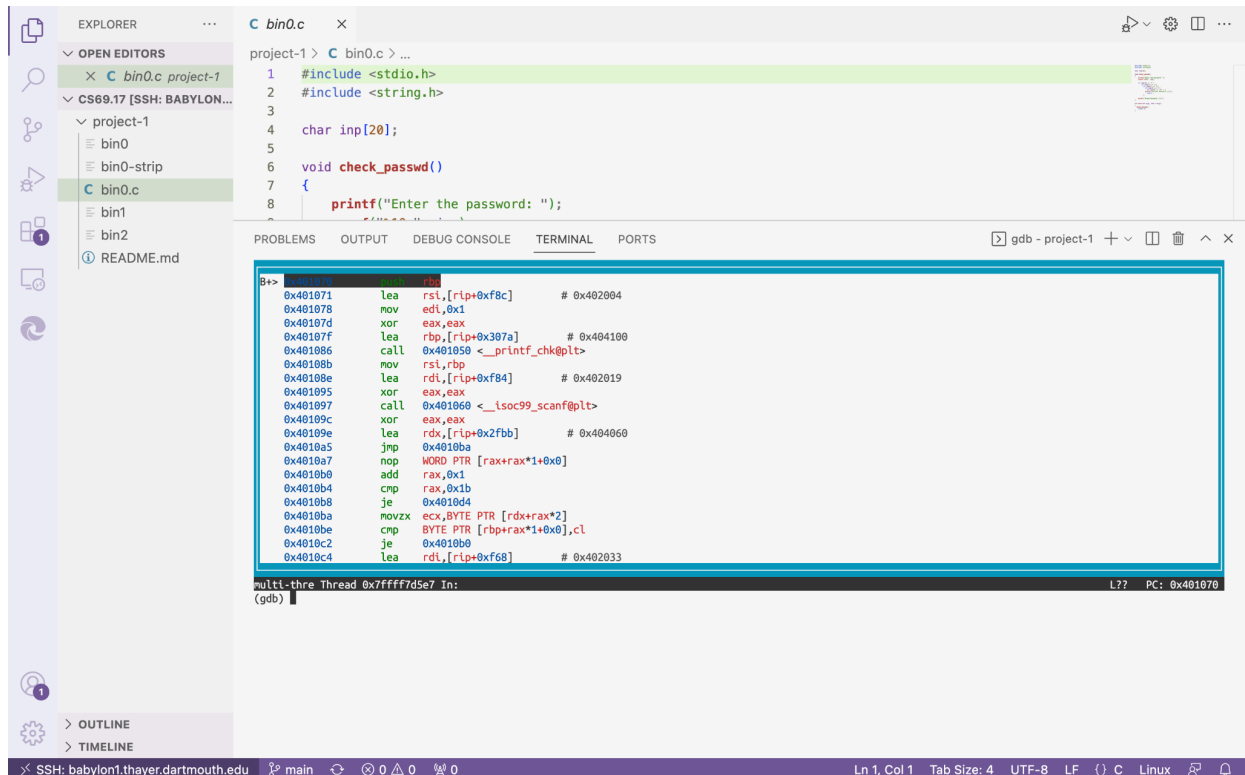
The terminal window displays the memory layout of the program, showing various sections and their addresses:

```
Entry point: 0x4010f0
0x0000000000400318 - 0x0000000000400334 is .interp
0x0000000000400338 - 0x0000000000400358 is .note.gnu.property
0x0000000000400358 - 0x000000000040037c is .note.gnu.build-id
0x000000000040037c - 0x000000000040039c is .note.ABI-tag
0x00000000004003a0 - 0x00000000004003bc is .gnu.hash
0x00000000004003c0 - 0x0000000000400468 is .dynsym
0x0000000000400468 - 0x00000000004004e9 is .dynstr
0x00000000004004ea - 0x00000000004004f8 is .gnu.version
0x00000000004004f8 - 0x0000000000400548 is .gnu.version_r
0x0000000000400548 - 0x0000000000400578 is .rela.dyn
0x0000000000400578 - 0x00000000004005d8 is .rela.plt
0x0000000000401000 - 0x000000000040101b is .init
0x0000000000401020 - 0x0000000000401070 is .plt
0x0000000000401070 - 0x00000000004011d6 is .text
0x00000000004011d8 - 0x00000000004011e5 is .fini
0x0000000000402000 - 0x0000000000402045 is .rodata
0x0000000000402048 - 0x0000000000402074 is .eh_frame_hdr
0x0000000000402078 - 0x0000000000402104 is .eh_frame
0x0000000000403e10 - 0x0000000000403e18 is .init_array
0x0000000000403e18 - 0x0000000000403e20 is .fini_array
0x0000000000403e20 - 0x0000000000403ff0 is .dynamic
0x0000000000403ff0 - 0x0000000000404000 is .got
0x0000000000404000 - 0x0000000000404038 is .got.plt
0x0000000000404040 - 0x00000000004040de is .data
0x00000000004040e0 - 0x0000000000404180 is .bss
```

The terminal also shows the following commands and output:

```
(gdb) disass
No frame selected.
(gdb) disass main
No symbol table is loaded. Use the "file" command.
(gdb) b *0x0000000000401070
Breakpoint 1 at 0x401070
(gdb)
```

- Next, we run the program.
- After running the program, I will use the instruction `'layout asm'` to examine the assembly code as the program is running.



- After accessing the assembly, I try entering a few random passwords to try to understand the logic behind the assembly code. I keep doing this for a couple of few examples.
- I noticed that there is a counter running throughout the program that is being incremented after each character check. The counter is stored in `%rax`, and it's being compared with the decimal value of 27 after each increment.
- While playing with different random examples at first, a few commands are of use to us to understand what is going on in the assembly code. One command is ``info reg`` where it lays out the different values of the registers at hand at the point of the program where the command was called. This is useful to observe how register values change when stepping through different assembly instructions in the program.

The screenshot shows a Visual Studio Code editor with a C program 'bin0.c' open. The program includes `<stdio.h>` and `<string.h>`, and contains a `check_passwd()` function that prints 'Enter the password: '.

The assembly output in the debugger shows the following instructions:

```

0x401070  push    rbp
0x401071  lea     rsi,[rip+0xf8c] # 0x402004
0x401078  mov     edi,0x1
0x40107d  xor     eax,eax
0x40107f  lea     rbp,[rip+0x307a] # 0x404100
0x401086  call    0x401050 <_printf_chk@plt>
0x40108b  mov     rsi,rbp
0x40108e  lea     rdi,[rip+0xf84] # 0x402019
0x401095  xor     eax,eax
0x401097  call    0x401060 <__isoc99_scanf@plt>
0x40109e  lea     rdx,[rip+0x2fbb] # 0x404060
0x4010a5  jmp     0x4010ba
0x4010a7  nop     WORD PTR [rax+rax*1+0x0]
0x4010b0  add     rax,0x1
0x4010b4  cmp     rax,0x1b
0x4010b8  je      0x4010d4
0x4010ba  movzx   ecx, BYTE PTR [rdx+rax*2]
0x4010be  cmp     BYTE PTR [rbp+rax*1+0x0],cl
0x4010c2  je      0x4010b0
0x4010c4  lea     rdi,[rip+0xf68] # 0x402033

```

The register window shows the following values:

Register	Value
rax	0x1
rbx	0x0
rcx	0x7ffff7f7baa0
rdx	0x0
rsi	0xa
rdi	0x7ffffc950
rbp	0x404100
rsp	0x7ffffc90
r8	0x0
r9	0x4056b0
r10	0xfffffffffffb0

- Another useful command I used was ``print $cl``. This command is of use to us in the case of bin1 because in the assembly code, in each loop the code makes while checking the user-input password, it compares each character from what the user entered with whatever value is stored in `%cl`, that is, the lower byte value of `%rcx`. This is an indicator that the correct password is actually being stored at each round in `%cl` since this is the register we use to verify whatever the user entered as a password.

```

project-1 > C bin0.c > ...
1 #include <stdio.h>
2 #include <string.h>
3
4 char inp[20];
5
6 void check_passwd()
7 {
8     printf("Enter the password: ");
9 }

```

```

0x401070 push rbp, +0xf68 # 0x402033
0x401071 lea rsi, [rip+0xf8c] # 0x402004
0x401072 mov edi, 0x1
0x401073 xor eax, eax
0x401074 lea rbp, [rip+0x307a] # 0x404100
0x401075 call 0x401050 <_printf_chk@plt>
0x401076 mov rsi, rbp
0x401077 lea rdi, [rip+0xf84] # 0x402019
0x401078 xor eax, eax
0x401079 call 0x401060 <_lsc99_scanf@plt>
0x40107a call 0x401060 <_lsc99_scanf@plt>
0x40107b xor eax, eax
0x40107c lea rdx, [rip+0x2fbb] # 0x404060
0x40107d jmp 0x4010ba [rax+rax*1+0x0]
0x40107e add rax, 0x1
0x40107f cmp rax, 0x1b
0x401080 je 0x4010d4
0x401081 movzx ecx, BYTE PTR [rdx+rax*2], cl
0x401082 movzx ecx, BYTE PTR [rdx+rax*2], cl
0x401083 lea rdi, [rip+0xf68] # 0x402033

```

```

(gdb) print $cl
$3 = 80
(gdb)

```

- In total, after trying out different password examples and following the assembly code, we can understand the general logic behind the program. Essentially, the program compares the entered password character by character with the correct password it has already stored. If one comparison fails, i.e. the two compared characters are not the same, the loop breaks and then prints the string `Wrong Password :(`. Otherwise, it would just keep comparing the characters, character by character, until it reaches the end at 27-long characters, and then it prints `Correct Password :)`.
- My strategy to crack the correct password is the following. In the first round, I enter an arbitrary one-character long password. Then, as soon as the program reaches the compare instruction with %cl, i.e. the `cmp %cl, 0x0(%rbp, %rax, 1)` instruction, I print the value of %cl using the `print \$cl` command in gdb, and then I write that value down because that is the first character of the correct password. Note that the printed value of %cl is in decimal value, that is, I have to look up the equivalent character value in ASCII of that decimal value. In our case, the first character of the correct password is `P`.

- In the next round, I enter an arbitrary two-character long password starting with P. I repeat the same strategy I used to crack the first character of the correct password to crack the second character of the correct password. In our case, that would be `0`.
- I keep repeating the same steps for 27 rounds to crack the 27 characters of the correct password.
- Overall, here are the characters stored in %cl that I got after each round.
 - First character is decimal 80 which is equivalent to `P` in ASCII
 - 48 = 0
 - 95 = _
 - 121 = y
 - 48 = 0
 - 117 = u
 - 95 = _
 - 108 = l (lowercase L)
 - 49 = 1 (number one)
 - 81 = Q
 - 101 = e
 - 95 = _
 - 75 = K
 - 114 = r
 - 52 = 4
 - 98 = b
 - 98 = b
 - 89 = Y
 - 95 = _
 - 112 = p
 - 65 = A
 - 116 = t
 - 55 = 7
 - 49 = 1 (number one)
 - 101 = e
 - 122 = z
 - Last character 63 = ?

- At the end, I concatenate all the characters I got into one single 27-character long password, and that would be precisely `PO_y0u_l1Qe_Kr4bbY_pAt71ez?``.

3. (10 pts) What is the correct password for *bin2*?

- The correct password for bin2 is `K_<3_mY_sOF7W4re_5EcUR1tY_cl4sZ``.

4. (15 pts) Describe your strategy to crack the password of *bin2*. To receive full credit, explain how you determined the password. Explain all the steps. (screenshots may help.)

- Since bin2 is a stripped ELF binary, gdb has no idea where the address of the ``main`` function is when asked to disassemble the binary.
- An alternative to access the assembly code is by using the ``objdump -d`` command instruction. This instruction spits out the entire assembly code of bin2.

```

project-1 > C bin0.c > ...
1 #include <stdio.h>
2 #include <string.h>
3
4 char inp[20];
5
6 void check_passwd()
7 {
8     printf("Enter the password: ");
9     // ...
10 }

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
f004s8n@babylon1:~/cs69.17/project-1$ objdump -d bin2
bin2:      file format elf64-x86-64

Disassembly of section .init:
0000000000401000 <.init>:
401000: f3 0f 1e fa      endbr64
401004: 48 83 ec 08      sub    $0x8,%rsp
401008: 48 8b 05 e9 2f 00 mov    0x2fe9(%rip),%rax      # 403ff8 <__isoc99_scanf@plt+0x2fa8>
40100f: 48 85 c0      test   %rax,%rax
401012: 74 02      je     401016 <puts@plt-0x1a>
401014: ff 00      call   *%rax
401016: 48 83 c4 08      add    $0x8,%rsp
40101a: c3      ret

Disassembly of section .plt:
0000000000401020 <puts@plt-0x10>:
401020: ff 35 e2 2f 00 00 push   0x2fe2(%rip)      # 404008 <__isoc99_scanf@plt+0x2fb8>
401026: ff 25 e4 2f 00 00 jmp     *0x2fe4(%rip)      # 404010 <__isoc99_scanf@plt+0x2fc0>
40102c: 0f 1f 40 00      nopl   0x0(%rax)

0000000000401030 <puts@plt>:
401030: ff 25 e2 2f 00 00 jmp     *0x2fe2(%rip)      # 404018 <__isoc99_scanf@plt+0x2fc8>
401036: 68 00 00 00 00 00 push   $0x0
40103b: e9 e0 ff ff      jmp     401020 <puts@plt-0x10>

0000000000401040 <__printf_chk@plt>:
401040: ff 25 da 2f 00 00 jmp     *0x2fda(%rip)      # 404020 <__isoc99_scanf@plt+0x2fd0>
401046: 68 01 00 00 00 00 push   $0x1
40104b: e9 d0 ff ff      jmp     401020 <puts@plt-0x10>

0000000000401050 <__isoc99_scanf@plt>:
401050: ff 25 d2 2f 00 00 jmp     *0x2fd2(%rip)      # 404028 <__isoc99_scanf@plt+0x2fd8>

```

- After examining the assembly code we got from the object dump, we notice that there are a ton of ``cmpb`` instructions in the assembly. After a

closer look, it appears that the program compares whatever the user entered as a password with all these hardcoded values in the different `cmpb` instructions.

```

1 #include <stdio.h>
2 #include <string.h>
3
4 char inp[20];
5
6 void check_passwd()
7 {
8     printf("Enter the password: ");
9 }

```

ADDRESS	HEX	ASSEMBLY	COMMENT
40109f:	e8 8c ff ff	call	401030 <puts@plt>
4010a4:	31 c0	xor	%eax,%eax
4010a6:	48 81 c4 88 00 00 00	add	\$0x88,%rsp
4010a8:	c3	ret	
4010aa:	80 7c 24 02 3c	cmpb	\$0x3c,0x2(%rsp)
4010b3:	75 e3	jne	401098 <__isoc99_scanf@plt+0x48>
4010b5:	80 7c 24 03 33	cmpb	\$0x33,0x3(%rsp)
4010ba:	75 dc	jne	401098 <__isoc99_scanf@plt+0x48>
4010bc:	80 7c 24 0d 34	cmpb	\$0x34,0xd(%rsp)
4010c1:	75 d5	jne	401098 <__isoc99_scanf@plt+0x48>
4010c3:	80 7c 24 0b 37	cmpb	\$0x37,0xb(%rsp)
4010c8:	75 ca	jne	401098 <__isoc99_scanf@plt+0x48>
4010ca:	80 7c 24 11 35	cmpb	\$0x35,0x11(%rsp)
4010cf:	75 c7	jne	401098 <__isoc99_scanf@plt+0x48>
4010d1:	80 7c 24 05 6d	cmpb	\$0x6d,0x5(%rsp)
4010d6:	75 c0	jne	401098 <__isoc99_scanf@plt+0x48>
4010d8:	80 7c 24 06 59	cmpb	\$0x59,0x6(%rsp)
4010dd:	75 b9	jne	401098 <__isoc99_scanf@plt+0x48>
4010df:	80 7c 24 07 5f	cmpb	\$0x5f,0x7(%rsp)
4010e4:	75 b2	jne	401098 <__isoc99_scanf@plt+0x48>
4010e6:	80 7c 24 12 45	cmpb	\$0x45,0x12(%rsp)
4010eb:	75 ab	jne	401098 <__isoc99_scanf@plt+0x48>
4010ed:	80 7c 24 14 55	cmpb	\$0x55,0x14(%rsp)
4010f2:	75 a4	jne	401098 <__isoc99_scanf@plt+0x48>
4010f4:	80 7c 24 1a 63	cmpb	\$0x63,0x1a(%rsp)
4010f9:	75 9d	jne	401098 <__isoc99_scanf@plt+0x48>
4010fb:	80 7c 24 1f 00	cmpb	\$0x0,0x1f(%rsp)
401100:	75 96	jne	401098 <__isoc99_scanf@plt+0x48>
401102:	80 7c 24 0c 57	cmpb	\$0x57,0xc(%rsp)
401107:	75 8f	jne	401098 <__isoc99_scanf@plt+0x48>
401109:	80 7c 24 10 5f	cmpb	\$0x5f,0x10(%rsp)
40110e:	75 88	jne	401098 <__isoc99_scanf@plt+0x48>
401110:	80 7c 24 1b 6c	cmpb	\$0x6c,0x1b(%rsp)
401115:	75 81	jne	401098 <__isoc99_scanf@plt+0x48>
401117:	80 7c 24 1c 34	cmpb	\$0x34,0x1c(%rsp)
40111c:	0f 85 76 ff ff	jne	401098 <__isoc99_scanf@plt+0x48>

- However, it is important to note that the comparisons are not being done in order. That is, the program compares the 13th character before it compares the 5th character for example, and the way to find out which character we are comparing precisely is by looking at the offset from %rsp in the `cmpb` instructions.
- Another thing to note is that the values we are comparing are in hex, so to find the character values we have to convert these hex values into characters by looking them up in an ASCII table.
- Overall, my strategy to crack the correct password is the following. Look into all the `cmpb` instructions where we compare a hard coded hex value with a value within a positive offset from the stack pointer. This hard coded hex value we are comparing it with a value being retrieved from the stack is one character from the correct password we are trying to hack. However, to find out which character exactly this value is from the correct

password, i.e. is it the 5th character vs the 23th character, we just keep note of the offset from the stack we are comparing its value with.

- Here are all the characters we compare with the password that the user inputs. I had written them in order for easier concatenation of the correct password.

- Password[0] = K
- Password[1] = _
- Password[2] = <
- Password[3] = 3
- Password[4] = _
- Password[5] = m
- Password[6] = Y
- Password[7] = _
- Password[8] = s
- Password[9] = 0 (zero)
- Password[10] = F
- Password[11] = 7
- Password[12] = W
- Password[13] = 4
- Password[14] = r
- Password[15] = e
- Password[16] = _
- Password[17] = 5
- Password[18] = E
- Password[19] = c
- Password[20] = U
- Password[21] = R
- Password[22] = 1 (number one)
- Password[23] = t
- Password[24] = Y
- Password[25] = _
- Password[26] = c
- Password[27] = l (lower case L)
- Password[28] = 4

- Password[29] = s
- Password[30] = Z
- Password[31] = Null
- At the end, I concatenate all the characters I got into one single 30-character long password, and that would be precisely`
K_<3_mY_s0F7W4re_5EcUR1tY_cl4sZ`.