Reverse Engineering and Buffer Overflow

Table of Contents

TASK 1 – Extracting Strings

The binary in TASK 1 requires you enter a password to print the flag.

```
~/CompetentAggressiveEnvironment/TASK1$ ./crackme1
Enter password: aaaaa
Incorrect password.
```

Use `rabin2 -z crackme1` to extract the strings from a binary file.

```
~/CompetentAggressiveEnvironment/TASK1$ rabin2 -z crackme1
[Strings]
nth paddr      vaddr      len size section type  string
―――――――――――――――――――――――――――――――――――――――――――――――――
0   0x00002004 0x00402004 16  17   .rodata ascii Enter password:
1   0x00002015 0x00402015 19  20   .rodata ascii SuperSecretPassword
2   0x0000202b 0x0040202b 8   9    .rodata ascii flag.txt
3   0x00002034 0x00402034 25  26   .rodata ascii Error: Could not open %s\n
4   0x00002051 0x00402051 19  20   .rodata ascii Incorrect password.
5   0x00002068 0x00402068 34  35   .rodata ascii Access granted. Here is your flag:
```

Run the binary again and enter the extracted password.

```
~/CompetentAggressiveEnvironment/TASK1$ ./crackme1
Enter password: SuperSecretPassword
Access granted. Here is your flag:
FCIL{NiC3_0n3}
```

TASK 2 – It's HASHED

Same as before, this binary requires you to enter a password to print the flag.

```
~/CompetentAggressiveEnvironment/TASK2$ ./crackme2
Enter password: aaaaa
Incorrect password.
```

Let's try `rabin2 -z crackme2` to extract strings again.

```
~/CompetentAggressiveEnvironment/TASK2$ rabin2 -z crackme2
[Strings]
nth paddr      vaddr      len size section type  string
―――――――――――――――――――――――――――――――――――――――――――――――――――――――――
0   0x00002008 0x00402008 35  36   .rodata ascii echo -n "%s" | openssl dgst -sha256
1   0x00002030 0x00402030 34  35   .rodata ascii Access granted. Here is your flag:
2   0x00002058 0x00402058 64  65   .rodata ascii
059a00192592d5444bc0caad7203f98b506332e2cf7abb35d684ea9bf7c18f08
3   0x0000209b 0x0040209b 29  30   .rodata ascii Failed to run openssl command
4   0x000020b9 0x004020b9 16  17   .rodata ascii Enter password:
5   0x000020ca 0x004020ca 17  18   .rodata ascii Error in OpenSSL\n
6   0x000020dc 0x004020dc 8   9    .rodata ascii flag.txt
7   0x000020e5 0x004020e5 25  26   .rodata ascii Error: Could not open %s\n
8   0x00002102 0x00402102 19  20   .rodata ascii Incorrect password.
```

It looks like the password itself is not there, but its hash. It is easy to see that it is a sha256 hash.

Use John the ripper to crack it.

1. First, save the hash into a file: `echo -n "059a00192592d5444bc0caad7203f98b506332e2cf7abb35d684ea9bf7c18f08" > hashfile`

```
~/CompetentAggressiveEnvironment/TASK2$ echo -n
"059a00192592d5444bc0caad7203f98b506332e2cf7abb35d684ea9bf7c18f08" > hashfile
```

2. Run JtR on the hash file: `john hashfile`

```
~/CompetentAggressiveEnvironment/TASK2$ john hashfile
Warning: detected hash type "gost", but the string is also recognized as "HAVAL-256-
3"
Use the "--format=HAVAL-256-3" option to force loading these as that type instead
Warning: detected hash type "gost", but the string is also recognized as "Panama"
Use the "--format=Panama" option to force loading these as that type instead
Warning: detected hash type "gost", but the string is also recognized as "po"
Use the "--format=po" option to force loading these as that type instead
Warning: detected hash type "gost", but the string is also recognized as "Raw-
Keccak-256"
Use the "--format=Raw-Keccak-256" option to force loading these as that type instead
Warning: detected hash type "gost", but the string is also recognized as "Raw-
SHA256"
Use the "--format=Raw-SHA256" option to force loading these as that type instead
Warning: detected hash type "gost", but the string is also recognized as "skein-256"
Use the "--format=skein-256" option to force loading these as that type instead
Warning: detected hash type "gost", but the string is also recognized as "Snefru-
256"
Use the "--format=Snefru-256" option to force loading these as that type instead
Using default input encoding: UTF-8
Loaded 1 password hash (gost, GOST R 34.11-94 [64/64])
Will run 6 OpenMP threads
Proceeding with single, rules:Single
Press 'q' or Ctrl-C to abort, almost any other key for status
Almost done: Processing the remaining buffered candidate passwords, if any.
Proceeding with wordlist:/nix/store/yfyqnlplj6y28c1x9n3wf3av12b1dsqy-john-1.9.0-
jumbo-1/share/john/password.lst, rules:Wordlist
Proceeding with incremental:ASCII
```

Notice that JtR attempts to detect what type of hash the input is. It fails to detect that it is sha256. So, we will have to enforce it to decode sha256: `john --format=raw-sha256 hashfile`

```
~/CompetentAggressiveEnvironment/TASK2$ john --format=raw-sha256 hashfile
Created directory: /home/runner/.john
Using default input encoding: UTF-8
Loaded 1 password hash (Raw-SHA256 [SHA256 128/128 SSE2 4x])
Warning: poor OpenMP scalability for this hash type, consider --fork=6
Will run 6 OpenMP threads
Proceeding with single, rules:Single
Press 'q' or Ctrl-C to abort, almost any other key for status
Almost done: Processing the remaining buffered candidate passwords, if any.
Proceeding with wordlist:/nix/store/yfyqnlplj6y28c1x9n3wf3av12b1dsqy-john-1.9.0-
jumbo-1/share/john/password.lst, rules:Wordlist
1qaz2wsx         (?)
1g 0:00:00:00 DONE 2/3 (2024-11-27 11:48) 50.00g/s 2457Kp/s 2457Kc/s 2457KC/s
123456..ship4
Use the "--show --format=Raw-SHA256" options to display all of the cracked passwords
reliably
Session completed
```

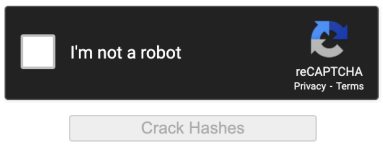
3. Run the binary again and enter the unhashed password

```
~/CompetentAggressiveEnvironment/TASK2$ ./crackme2
Enter password: 1qaz2wsx
Access granted. Here is your flag:
FCIL{Y0U_unh7sh3d_m3}
```

If you cannot use JtR, you can unhash the password using https://crackstation.net/.

Enter up to 20 non-salted hashes, one per line:

```
059a00192592d5444bc0caad7203f98b506332e2cf7abb35d684ea9bf7c18f08
```



Crack Hashes

Supports: LM, NTLM, md2, md4, md5, md5(md5_hex), md5-half, sha1, sha224, sha256, sha384, sha512, ripeMD160, whirlpool, MySQL 4.1+ (sha1(sha1_bin)), QubesV3.1BackupDefaults

Hash	Type	Result
059a00192592d5444bc0caad7203f98b506332e2cf7abb35d684ea9bf7c18f08	sha256	1qaz2wsx

TASK 3 – Meet Assembly

This challenge is copied from prof. Andrew Novocin at UD.

As before, the binary in TASK 3 requires you enter a password.

```
~/CompetentAggressiveEnvironment/TASK3$ ./crackme3
What is the password?:
asdf
nope!
```

Let's use `rabin2 -z crackme3` again

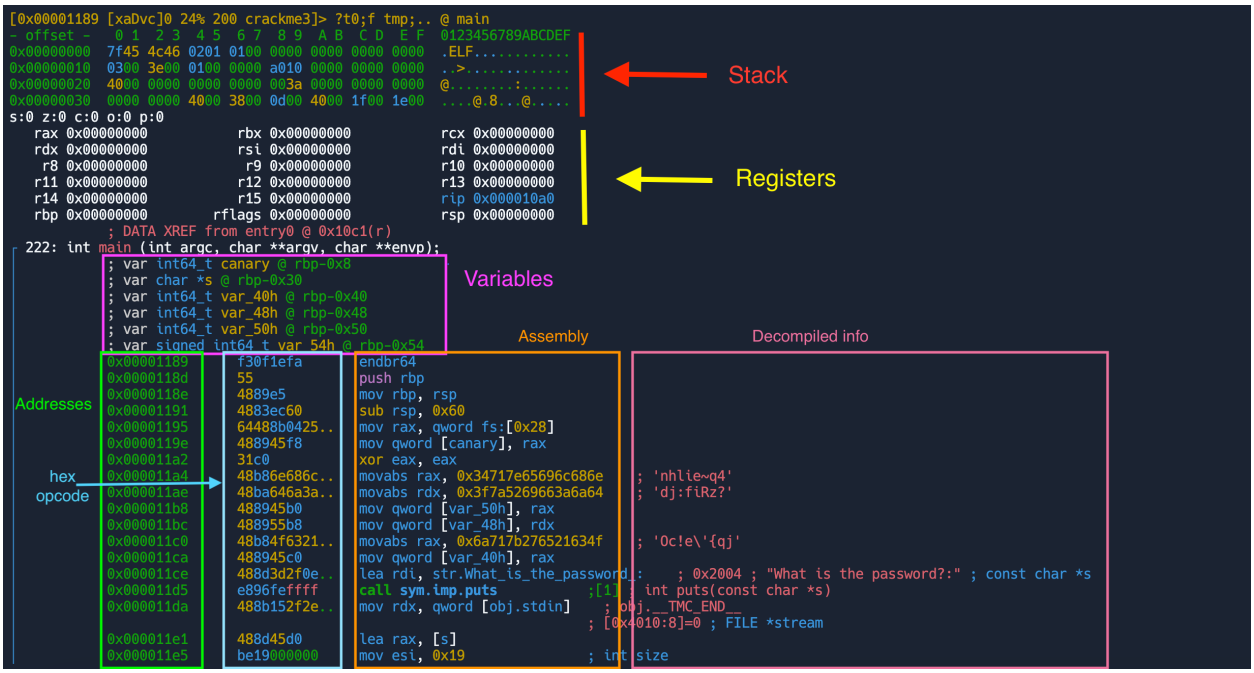
```
~/CompetentAggressiveEnvironment/TASK3$ rabin2 -z crackme3
[Strings]
nth paddr      vaddr      len size section type  string
―――――――――――――――――――――――――――――――――――――――――――――――――
0   0x00002004 0x00002004 22  23   .rodata ascii What is the password?:
1   0x0000201b 0x0000201b 5   6    .rodata ascii nope!
```

Nothing here! We will have to reverse engineer it. Run `r2 -A crackme3`

```
~/CompetentAggressiveEnvironment/TASK3$ r2 -Ad crackme3
WARN: Relocs has not been applied. Please use `-e bin.relocs.apply=true` or `-e
bin.cache=true` next time
INFO: Analyze all flags starting with sym. and entry0 (aa)
INFO: Analyze imports (af@@@i)
INFO: Analyze entrypoint (af@ entry0)
INFO: Analyze symbols (af@@@s)
INFO: Recovering variables
INFO: Analyze all functions arguments/locals (afva@@@F)
INFO: Analyze function calls (aac)
INFO: Analyze len bytes of instructions for references (aar)
INFO: Finding and parsing C++ vtables (avrr)
INFO: Analyzing methods
INFO: Recovering local variables (afva)
INFO: Skipping type matching analysis in debugger mode (aaft)
INFO: Propagate noreturn information (aanr)
INFO: Use -AA or aaaa to perform additional experimental analysis
[0x7f852cd61100]>
```

Seek to the main function: `s main`

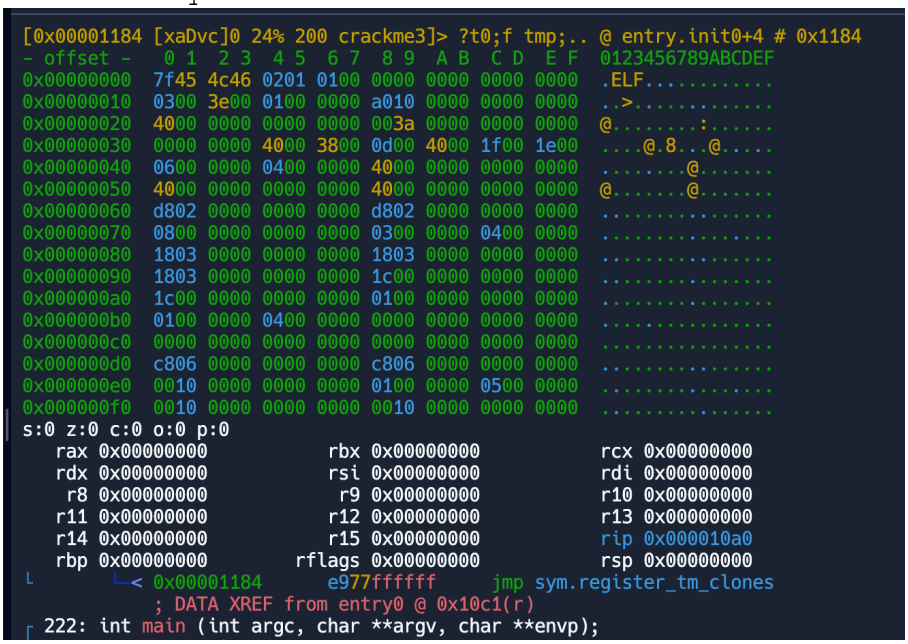
```
~/CompetentAggressiveEnvironment/TASK3$ r2 -Ad crackme3
WARN: Relocs has not been applied. Please use `-e bin.relocs.apply=true` or `-e
bin.cache=true` next time
INFO: Analyze all flags starting with sym. and entry0 (aa)
INFO: Analyze imports (af@@@i)
INFO: Analyze entrypoint (af@ entry0)
INFO: Analyze symbols (af@@@s)
INFO: Recovering variables
INFO: Analyze all functions arguments/locals (afva@@@F)
INFO: Analyze function calls (aac)
INFO: Analyze len bytes of instructions for references (aar)
INFO: Finding and parsing C++ vtables (avrr)
INFO: Analyzing methods
INFO: Recovering local variables (afva)
INFO: Skipping type matching analysis in debugger mode (aaft)
INFO: Propagate noreturn information (aanr)
INFO: Use -AA or aaaa to perform additional experimental analysis
[0x7f852cd61100]> s main
[0x556165dc1189]>
```



You can change the views by pressing `p`

Increase the stack view.

1. Press `shift+:` to enter command mode.
2. Enter `e stack.size=256`
3. Enter `q` to exit



You will see these lines in every function. This is the called the function prolog. It initiates the required stack space in memory for the main function.

```
222: int main (int argc, char **argv, char **envp);
          ; var int64_t var_8h @ rbp-0x8
          ; var int64_t var_30h @ rbp-0x30
          ; var int64_t var_40h @ rbp-0x40
          ; var int64_t var_48h @ rbp-0x48
          ; var int64_t var_50h @ rbp-0x50
          ; var int64_t var_54h @ rbp-0x54
          0x556165dc1189      f30f1efa        endbr64
          0x556165dc118d      55              push rbp
          0x556165dc118e      4889e5          mov rbp, rsp
          0x556165dc1191      4883ec60        sub rsp, 0x60
```

At the end of the function, you will see function epilog. Which clears the function from the stack.

```
          0x556165dc1255      6448330c25..    xor rcx, qword fs:[0x28]
          0x556165dc125e      7405            je 0x556165dc1265
          0x556165dc1260      e81bfeffff      call sym.imp.__stack_chk_fail ;[2] ; void __stack_chk_fail(void)
          0x556165dc1265      c9              leave
          0x556165dc1266      c3              ret
          0x556165dc1267      660f1f8400..    nop word [rax + rax]
```

Let's start the debugger to see what the binary does.

1. Press ":" to enter command mode.

```
          0x556165dc11e5      be19000000      mov esi, 0x19
          0x556165dc11ea      4889c7          mov rdi, rax
          0x556165dc11ed      e89efeffff      call sym.imp.fgets
          0x556165dc11f2      c745ac0000..    mov dword [var_54h], 0
          0x556165dc11f9      c745ac0000..    mov dword [var_54h], 0
          0x556165dc1200      eb38            jmp 0x556165dc123a
          0x556165dc1202      8b45ac          mov eax, dword [var_54h
> 
```

2. Enter $db\ main$ to set a breakpoint at the main function.
3. Enter dc to run the program until hitting a breakpoint. Then press enter.
    a. You may need to press $F9$ to continue running to the main function.

```
          ; var int64_t var_54h @ rbp-0x54
          0x55bb4ea85189      f30f1efa        endbr64
          0x55bb4ea8518d      55              push rbp
          0x55bb4ea8518e      4889e5          mov rbp, rsp
          0x55bb4ea85191      4883ec60        sub rsp, 0x60
> db main
> dc
INFO: hit breakpoint at: 0x55bb4ea85189
> 
```

4. Press “s” to do single step.
5. The following lines with decoded string will process and store some hex values in the rax and rdx registers.

```
        ;-- rip:
        0x55bb4ea851a4      48b86e686c..    movabs rax, 0x34717e65696c686e    ; 'nhlie~q4'
        0x55bb4ea851ae      48ba646a3a..    movabs rdx, 0x3f7a5269663a6a64    ; 'dj:fiRz?'
        0x55bb4ea851b8      488945b0        mov qword [var_50h], rax
        0x55bb4ea851bc      488955b8        mov qword [var_48h], rdx
        0x55bb4ea851c0      48b84f6321..    movabs rax, 0x6a717b276521634f    ; 'Oc!e\'{qj'
        0x55bb4ea851ca      488945c0        mov qword [var_40h], rax
```

6. The next two lines will move the values in rax and rdx into the variables located at $rbp - 50$ and $rbp - 48$, respectively.

```
step at 0x55bb4ea851c0
- offset -      F0F1 F2F3 F4F5 F6F7 F8F9 FAFB FCFD FEFF  0123456789ABCDEF
0x7fffb9c0ddf0  4040 a84e bb55 0000 0d00 0000 0000 0000  @@.N.U..........
0x7fffb9c0de00  6e68 6c69 657e 7134 646a 3a66 6952 7a3f  nhlie~q4dj:fiRz?
0x7fffb9c0de10  e015 4f6e 937f 0000 bd52 a84e bb55 0000  ..On.....R.N.U..
0x7fffb9c0de20  c88f 4e6e 937f 0000 7052 a84e bb55 0000  ..Nn....pR.N.U..
0x7fffb9c0de30  0000 0000 0000 0000 a050 a84e bb55 0000  .........P.N.U..
0x7fffb9c0de40  40df c0b9 ff7f 0000 0094 9851 9e08 30e3  @..........Q..0.
0x7fffb9c0de50  0000 0000 0000 0000 b3f0 316e 937f 0000  ..........1n....
0x7fffb9c0de60  0100 0000 0000 0000 48df c0b9 ff7f 0000  ........H.......
0x7fffb9c0de70  1806 4e6e 0100 0000 8951 a84e bb55 0000  ..Nn.....Q.N.U..
0x7fffb9c0de80  7052 a84e bb55 0000 cf18 6e98 5107 3c48  pR.N.U....n.Q.<H
0x7fffb9c0de90  a050 a84e bb55 0000 40df c0b9 ff7f 0000  .P.N.U..@.......
0x7fffb9c0dea0  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7fffb9c0deb0  cf18 ae24 d074 c3b7 cf18 a078 32db 1ab7  ...$.t.....x2...
0x7fffb9c0dec0  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7fffb9c0ded0  0000 0000 0000 0000 0100 0000 0000 0000  ................
0x7fffb9c0dee0  48df c0b9 ff7f 0000 58df c0b9 ff7f 0000  H.......X.......
s:0 z:1 c:0 o:0 p:1
   rax 0x34717e65696c686e   rbx 0x55bb4ea85270       rcx 0x55bb4ea85270
   rdx 0x3f7a5269663a6a64    r8 0x00000000           r9 0x7f936e500d50
   r10 0x7f936e51cf68       r11 0x00000202          r12 0x55bb4ea850a0
   r13 0x7fffb9c0df40       r14 0x00000000          r15 0x00000000
   rsi 0x7fffb9c0df48       rdi 0x00000001          rsp 0x7fffb9c0ddf0
   rbp 0x7fffb9c0de50       rip 0x55bb4ea851c0    rflags 0x00000246
  orax 0xffffffffffffffff
            0x55bb4ea8518d      55              push rbp
            0x55bb4ea8518e      4889e5          mov rbp, rsp
            0x55bb4ea85191      4883ec60        sub rsp, 0x60
            0x55bb4ea85195      64488b0425..    mov rax, qword fs:[0x28]
            0x55bb4ea8519e      488945f8        mov qword [var_8h], rax
            0x55bb4ea851a2      31c0            xor eax, eax
            0x55bb4ea851a4      48b86e686c..    movabs rax, 0x34717e65696c686e
            0x55bb4ea851ae      48ba646a3a..    movabs rdx, 0x3f7a5269663a6a64
            0x55bb4ea851b8      488945b0        mov qword [var_50h], rax
            0x55bb4ea851bc      488955b8        mov qword [var_48h], rdx
            ;-- rip:
```

7. Until this point, the binary has stored the strings in stack. The next lines show that the binary will print the message to enter a password and will be waiting for our input.

```
;-- rip:
0x55bb4ea851ce      488d3d2f0e..    lea rdi, str.What_is_the_password_:     ; 0x55bb4ea86004 ; "What is the password?:"
0x55bb4ea851d5      e896feffff      call sym.imp.puts       ;[1] ; int puts(const char *s)
0x55bb4ea851da      488b152f2e..    mov rdx, qword [reloc.stdin]    ; [0x55bb4ea88010:8]=0x7f936e4e3980
0x55bb4ea851e1      488d45d0        lea rax, [var_30h]
0x55bb4ea851e5      be19000000      mov esi, 0x19           ; 25
0x55bb4ea851ea      4889c7          mov rdi, rax
0x55bb4ea851ed      e89efeffff      call sym.imp.fgets      ;[2] ; char *fgets(char *s, int size, FILE *stream)
0x55bb4ea851f2      c745ac0000..    mov dword [var_54h], 0
0x55bb4ea851f9      c745ac0000..    mov dword [var_54h], 0
0x55bb4ea85200      eb38            jmp 0x55bb4ea8523a
```

8. Press ":" to go to the command mode. We will set a breakpoint after the $fgets$ line.

```
            0x5558937201ed      e89efeffff      call sym.imp.fgets      ;[2] ; char *fgets(char *s,
            0x5558937201f2      c745ac0000..    mov dword [var_54h], 0
            0x5558937201f9      c745ac0000..    mov dword [var_54h], 0
        ┌─< 0x555893720200      eb38            jmp 0x55589372023a
        ┌─> 0x555893720202      8b45ac          mov eax, dword [var_54h]
        ¦   0x555893720205      4898            cdqe
        ¦   0x555893720207      0fb64405d0      movzx eax, byte [rbp + rax - 0x30]
        ¦   0x55589372020c      0fbed0          movsx edx, al
        ¦   0x55589372020f      8b45ac          mov eax, dword [var_54h]
        ¦   0x555893720212      4898            cdqe
        ¦   0x555893720214      0fb64405b0      movzx eax, byte [rbp + rax - 0x50]
        ¦   0x555893720219      0fbec0          movsx eax, al
        ¦   0x55589372021c      3345ac          xor eax, dword [var_54h]
        ¦   0x55589372021f      39c2            cmp edx, eax
> db 0x5558937201f2
> dc
What is the password?:
```

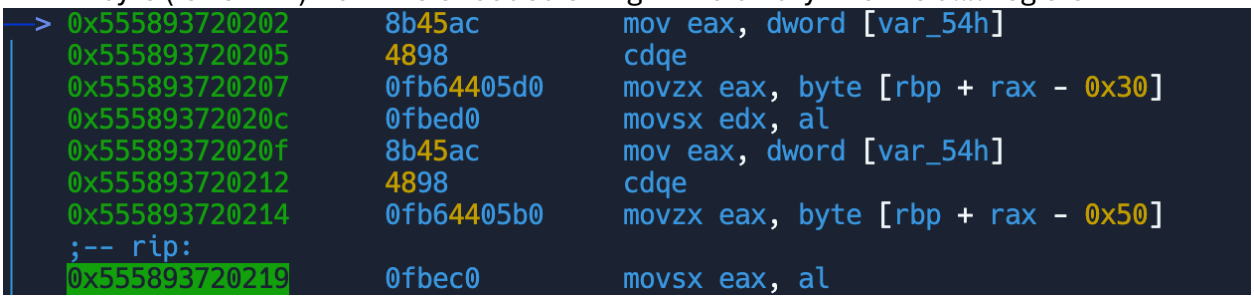
9. Enter a dummy string: aaaaaaaaaaaa. Then press enter two times. Notice how my input is stored in the stack.

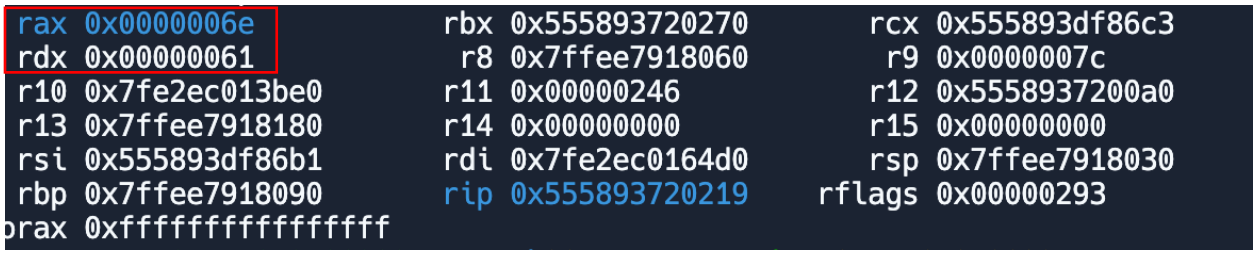
```
step at 0x555893720200
- offset -      3031 3233 3435 3637 3839 3A3B 3C3D 3E3F  0123456789ABCDEF
0x7ffee7918030  40f0 7193 5855 0000 0d00 0000 0000 0000  @.q.XU..........
0x7ffee7918040  6e68 6c69 657e 7134 646a 3a66 6952 7a3f  nhlie~q4dj:fiRz?
0x7ffee7918050  4f63 2165 277b 716a bd02 7293 5855 0000  Oc!e'{qj..r.XU..
0x7ffee7918060  6161 6161 6161 6161 6161 6161 6161 6161  aaaaaaaaaaaaaaaa
0x7ffee7918070  6161 0a00 0000 0000 a000 7293 5855 0000  aa........r.XU..
```

10. Now the binary will process my input to check if it is equal to the obfuscated password or not.
11. Apparently, these two lines initialize two variables to 0. These are possibly used in a loop, as we can see in the reversed code.

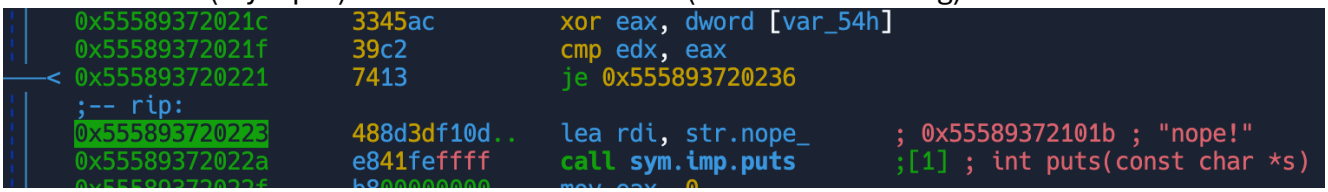
```
;-- rip:
0x5558937201f2 b    c745ac0000..    mov dword [var_54h], 0
0x5558937201f9      c745ac0000..    mov dword [var_54h], 0
```

12. These lines have copied a byte (letter “a”) from my input into the rdx register and a byte (letter “n”) from the encoded string in the binary into the eax register.





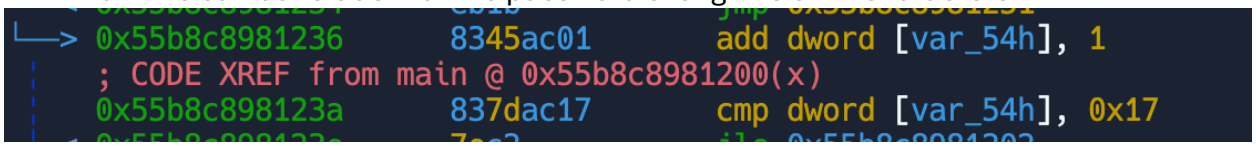
13. These two lines were doing two things: XORing the value in the eax with the value at $rbp - 54$ (which is the counter set 0) and doing a comparison between the value in the edx (my input) and the value in the eax (the encoded string).



14. The comparison between my input in the edx (letter “a”) and the encoded string in the eax (letter “n”) make the comparison fails and the program will output “nope!” and exit.

15. We will go over all the previous debugging steps again and this time we enter the same string that is encoded in the binary. Also, make sure to set a breakpoint directly at the line after $fgets$.

16. This time the binary compares my input (n) to the encoded character (n) and finds that both are equal. So, it continues in the loop. Notice that the counter ($ebp - 54$) is increased by one and checks whether it's 0x17 or not.
    a. We can conclude that the password's length is 0x17 characters.



17. In the second iteration, we see that the binary XORs its encoded string with the loop counter and compares it with my input.
    a. The second letter in the binary is "h".
    b. After XOR with the counter it becomes, "i"
    c. The comparison fails because my input's second letter is "h".
    d. The program prints "nope!" and exits.

Now, we understand how the encoded string is obfuscated: it gets every character and XORs it with a counter starting at 0x0 and ends at 0x17.

We can get that string and XOR it with the same counter sequence.

Use python:

```
s = "nhlie~q4dj:fiRz?Oc!e'{qj"
for i in range(len(s)):
  print(chr(ord(s[i])^i),end='')
```

TASK 4 – Reverse Again

Solve TASK4 by yourself.

TASK 5 – Let's Overflow It

When you extract the strings in this binary, you will see nothing about a password or a flag!

```
~/CompetentAggressiveEnvironment/TASK5$ rabin2 -z crackme5
[Strings]
nth paddr      vaddr      len size section type  string
―――――――――――――――――――――――――――――――――――――――――――――――――
0   0x0000200a 0x0040200a 8   9    .rodata ascii flag.txt
1   0x00002013 0x00402013 25  26   .rodata ascii Error: Could not open %s\n
2   0x00002030 0x00402030 34  35   .rodata ascii Access granted. Here is your flag:
3   0x00002056 0x00402056 16  17   .rodata ascii Get out of here!
4   0x00002067 0x00402067 12  13   .rodata ascii Who are you:
```

Let's analyze it using radare2: `radare2 -Ad crackme5` → `s main` → `Vpp`

The main function is simple:

```
69: int main (int argc, char **argv, char **envp);
          ; var int64_t var_1h @ rbp-0x1
          ; var int64_t var_70h @ rbp-0x70
          0x00401287      55              push rbp
          0x00401288      4889e5          mov rbp, rsp
          0x0040128b      4883ec70        sub rsp, 0x70
          0x0040128f      c645ff5a    1   mov byte [var_1h], 0x5a     ; 'Z' ; 90
          0x00401293      488d05cd0d..    lea rax, str.Who_are_you:    ; 0x402067 ; "Who are you:"
          0x0040129a      4889c7          mov rdi, rax
          0x0040129d      e88efdffff      call sym.imp.puts            ;[1] ; int puts(const char *s)
          0x004012a2      488d4590        lea rax, [var_70h]
          0x004012a6      4889c7          mov rdi, rax
          0x004012a9      b800000000      mov eax, 0
          0x004012ae      e8cdfdffff  2   call sym.imp.gets            ;[2] ; char *gets(char *s)
          0x004012b3      0fbe45ff        movsx eax, byte [var_1h]
          0x004012b7      488d5590        lea rdx, [var_70h]
          0x004012bb      4889d6          mov rsi, rdx
          0x004012be      89c7            mov edi, eax
          0x004012c0      e8c1feffff  3   call sym.print_flag          ;[3]
          0x004012c5      b800000000      mov eax, 0
          0x004012ca      c9              leave
          0x004012cb      c3              ret
```

- At 1: it stores the letter 0x5a (Z) at location $ebp - 1$
- At 2: it reads input from the user. The input is stored at the address found in the rdi register, which is the address of the array starting at $ebp - 70$.
    - Notice that $gets()$ function expects a character pointer as an argument. That is why it treats the value found in the rdi as the start address of the array at which it will store the user's input.
- At 3: it calls a function $print\_flag$.

Add a break point at the line that follows the $gets()$ function and enter any input when prompted.

```
> db 0x004012b3
> dc
Who are you:
AAAAAAAAAAAAAAAAAAAAAAAAAA
INFO: hit breakpoint at: 0x4012b3
```

Look at the stack. Can you see where your input "AAAA…" is stored and where is the "Z" is stored?

```
- offset -      E8E9 EAEB ECED EEEF F0F1 F2F3 F4F5 F6F7  89ABCDEF01234567
0x7ffe3f7e9fe8  c512 4000 0000 0000 4141 4141 4141 4141  ..@.....AAAAAAAA
0x7ffe3f7e9ff8  4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
0x7ffe3f7ea008  4141 4100 0000 0000 0000 0000 0000 0000  AAA.............
0x7ffe3f7ea018  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffe3f7ea028  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffe3f7ea038  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffe3f7ea048  6010 f6bc a37f 0000 30a1 7e3f fe7f 0000  `.......0.~?....
0x7ffe3f7ea058  0000 0000 0000 005a 0100 0000 0000 0000  .......Z........
0x7ffe3f7ea068  0ef1 d7bc a37f 0000 60a1 7e3f fe7f 0000  ........`.~?....
0x7ffe3f7ea078  8712 4000 0000 0000 4000 4000 0100 0000  ..@.....@.@.....
0x7ffe3f7ea088  78a1 7e3f fe7f 0000 78a1 7e3f fe7f 0000  x.~?....x.~?....
0x7ffe3f7ea098  6799 acd5 6fd6 3ddb 0000 0000 0000 0000  g...o.=.........
0x7ffe3f7ea0a8  88a1 7e3f fe7f 0000 0090 f7bc a37f 0000  ..~?............
0x7ffe3f7ea0b8  803d 4000 0000 0000 6799 4e95 92a8 c124  .=@.....g.N....$
0x7ffe3f7ea0c8  6799 2a34 c0af 7a24 0000 0000 0000 0000  g.*4..z$........
0x7ffe3f7ea0d8  0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Notice that the pointer to the array is copied from the rdx into the rsi register and the letter Z is copied from rax into the edi register.

```
0x004012bb      4889d6          mov rsi, rdx
0x004012be      89c7            mov edi, eax
;-- rip:
```

```
:0 z:0 c:0 o:0 p:0
 rax 0x0000005a      rbx 0x7ffe3f7ea178      rcx 0x7fa3bcf338e0
 rdx 0x7ffe3f7e9ff0   r8 0x0070b6cc           r9 0x00000000
 r10 0x00000004      r11 0x00000246          r12 0x00000000
 r13 0x7ffe3f7ea188  r14 0x7fa3bcf79000      r15 0x00403d80
 rsi 0x7ffe3f7e9ff0  rdi 0x0000005a          rsp 0x7ffe3f7e9ff0
 rbp 0x7ffe3f7ea060  rip 0x004012c0       rflags 0x00000202
orax 0xffffffffffffffff
```

Step into (pressing s) the $print\_flag$ function when reached.

- The function reads its arguments $arg1$ from the edi register, which is the letter "Z" and $arg2$ from the rsi register, which is the array pointer.
- At 0x004011a0, the binary compares the value at $ebp - 414$ (the letter "Z") to the hex value 0x5a (also the letter "Z").
- If they are equal, it will jump to another address.
- If they are not equal, it will continue executing the following lines.

We can see from the decompiled code segments that the lines after the comparison instruction read the flag from the file.

We want the program to execute them. But HOW?
We need to make comparison results in False, so it does not jump. HOW?
We need to change the value "Z" that is passed to the function. HOW?
Overwrite it. HOW?

The vulnerability here is that the $gets()$ function is insecure; it reads the user's input and stores it in the array without checking its length. Thus, if the array's size is 100 bytes and a user enters 400 bytes, they will be stored in the array.

Keep in mind that an array is nothing but continuous words in the memory. So, whatever input you enter, it will be stored in the stack. The vulnerability here manifests in this idea; if the $gets()$ function gets input with unlimited size, we can overwrite other values in the memory.

Look at the stack again, you will see that the start of the array is far from the letter Z by 112 bytes. Thus, if we enter 112 bytes of "A", we will overwrite the value "Z". Hence, we will pass the comparison.

Before reading the user's input:

```
step at 0x004012a9
- offset -       C0C1 C2C3 C4C5 C6C7 C8C9 CACB CCCD CECF  0123456789ABCDEF
0x7ffe0f85f8c0  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffe0f85f8d0  1900 0000 0100 0000 0000 0000 0000 0000  ................
0x7ffe0f85f8e0  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffe0f85f8f0  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffe0f85f900  0000 0000 0000 0000 0000 0000 0000 0000  ................
0x7ffe0f85f910  0000 0000 0000 0000 60f0 8b1b ed7f 0000  ........`.......
0x7ffe0f85f920  00fa 850f fe7f 0000 0000 0000 0000 005a  ...............Z
```

Entering the payload

```
> db 0x004012b3
> dc
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
INFO: hit breakpoint at: 0x4012b3
> 
```

After reading the input:

```
- offset -       C0C1 C2C3 C4C5 C6C7 C8C9 CACB CCCD CECF  0123456789ABCDEF
0x7ffe0f85f8c0  4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
0x7ffe0f85f8d0  4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
0x7ffe0f85f8e0  4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
0x7ffe0f85f8f0  4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
0x7ffe0f85f900  4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
0x7ffe0f85f910  4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
0x7ffe0f85f920  4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
```

Now go back to the binary, execute it, and enter the 112 As and it will print the flag.