Week4-RE Write-ups

Name: **Xinsheng Zhu**
UnivID: **N10273832**
NetID: **xz4344**

!!! 500/300 pts solved !!!

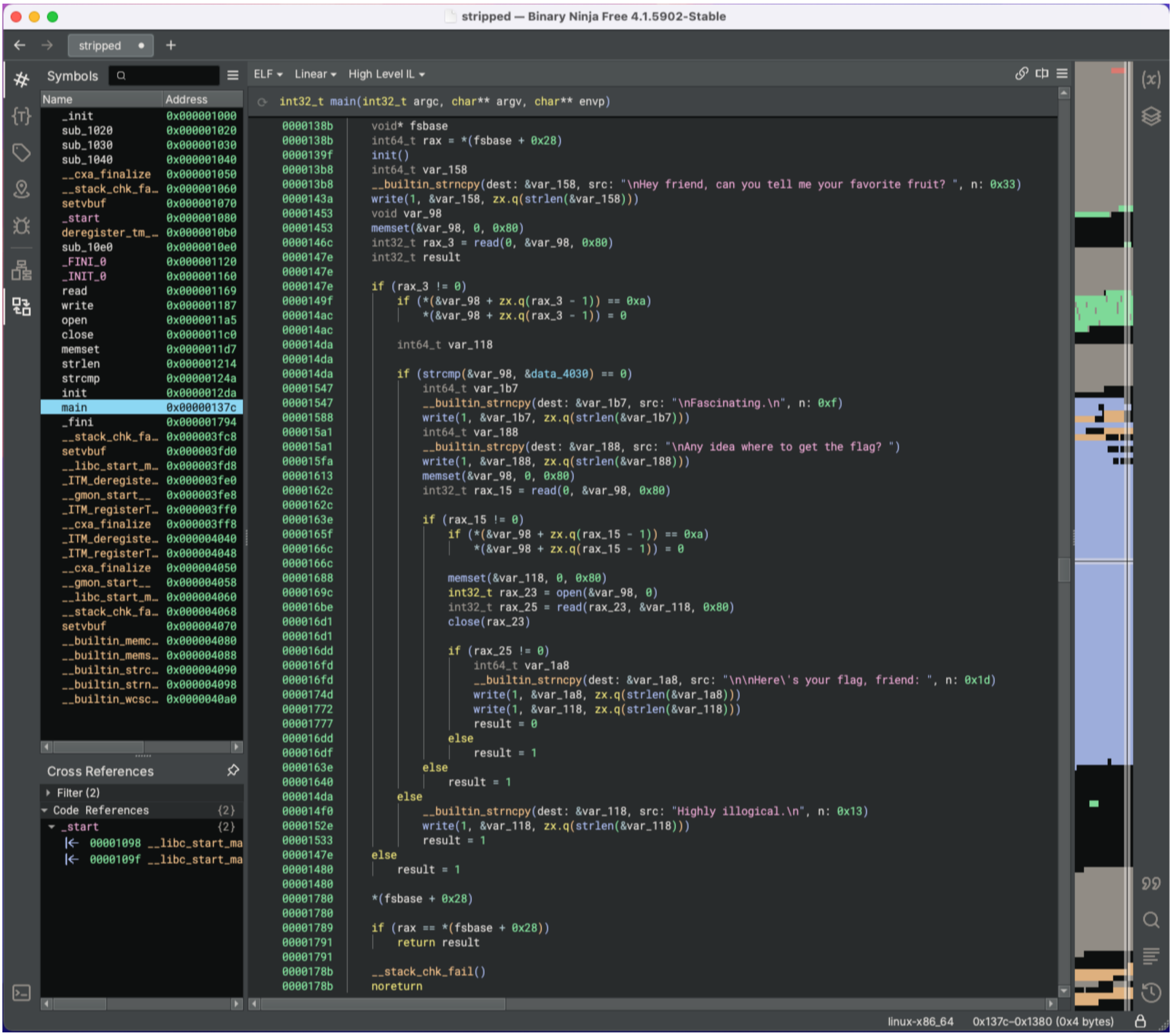
Stripped (50 pts)

In this CTF challenge, we must tell the program what the "favorite fruit" is; nothing further useful is provided.

As shown below, since the `objdump` command outputs nothing for the binary file `stripped`, we can clearly learn that it is a stripped binary. Stripping effectively removes all user-defined variables and function names in the compiled output.

```
root@17b95fe8a8e6:~/wk4/stripped# objdump -M intel --disassemble=main stripped

stripped:     file format elf64-x86-64


Disassembly of section .init:

Disassembly of section .plt:

Disassembly of section .plt.got:

Disassembly of section .plt.sec:

Disassembly of section .text:

Disassembly of section .fini:
```

Thus, while opening the binary file `stripped` with Binary Ninja to find helpful information, we have to rename unreadable symbols and variables, which makes it conveniently understandable. The following is the High-Level IL of the `main` function.



In short, the `main` function asks for two pieces of input. If the first input matches a pre-set string, aka the "favorite fruit", stored in `data_4030`, the function treats the second input as a filename and performs file operations on it to retrieve and display a flag.

By calling the `init` function (screenshot omitted) in the `main` function, `data_4030` is set to `Golana Melon` as a string. Besides, based on the common sense of former challenges, the flag is always stored in a file called `flag.txt`.

Therefore, a script using Pwntools is written to send these two answers to the server. Part of the script is shown below.

```
from pwn import *
......
p = remote(URL, PORT)
......
print(p.recvuntil(b"\nHey friend, can you tell me your favorite fruit? ").decode())
fruit = "Golana Melon"
p.sendline(fruit.encode())
log.info(f"Sending fruit: {fruit}")
print(p.recvuntil(b"\nAny idea where to get the flag? ").decode())
filename = "flag.txt"
p.sendline(filename.encode())
log.info(f"Sending filename: {filename}")
print(p.recvall().decode())

p.interactive()
```

The console output of the script execution is shown below.

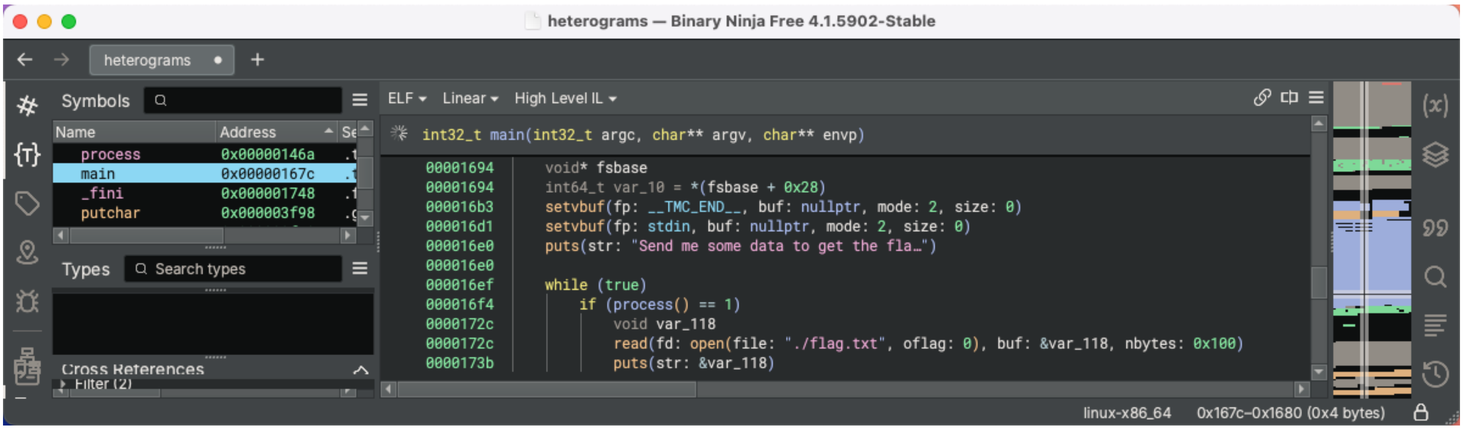
```
root@17b95fe8a8e6:~/wk4/stripped# python3 stripped.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1270: Done
hello, xz4344. Please wait a moment...

Hey friend, can you tell me your favorite fruit?
[*] Sending fruit: Golana Melon

Fascinating.

Any idea where to get the flag?
[*] Sending filename: flag.txt
[+] Receiving all data: Done (96B)
[*] Closed connection to offsec-chalbroker.osiris.cyber.nyu.edu port 1270


Here's your flag, friend: flag{4ll_w3_n33d_1s_kn0wl3dg3_0f_th3_sysc4ll_API!_e47a584bb03631ee}

[*] Switching to interactive mode
[*] Got EOF while reading in interactive
$
[*] Interrupted
```

The captured flag is `flag{4ll_w3_n33d_1s_kn0wl3dg3_0f_th3_sysc4ll_API!_e47a584bb03631ee}`.

Heterograms (200 pts)

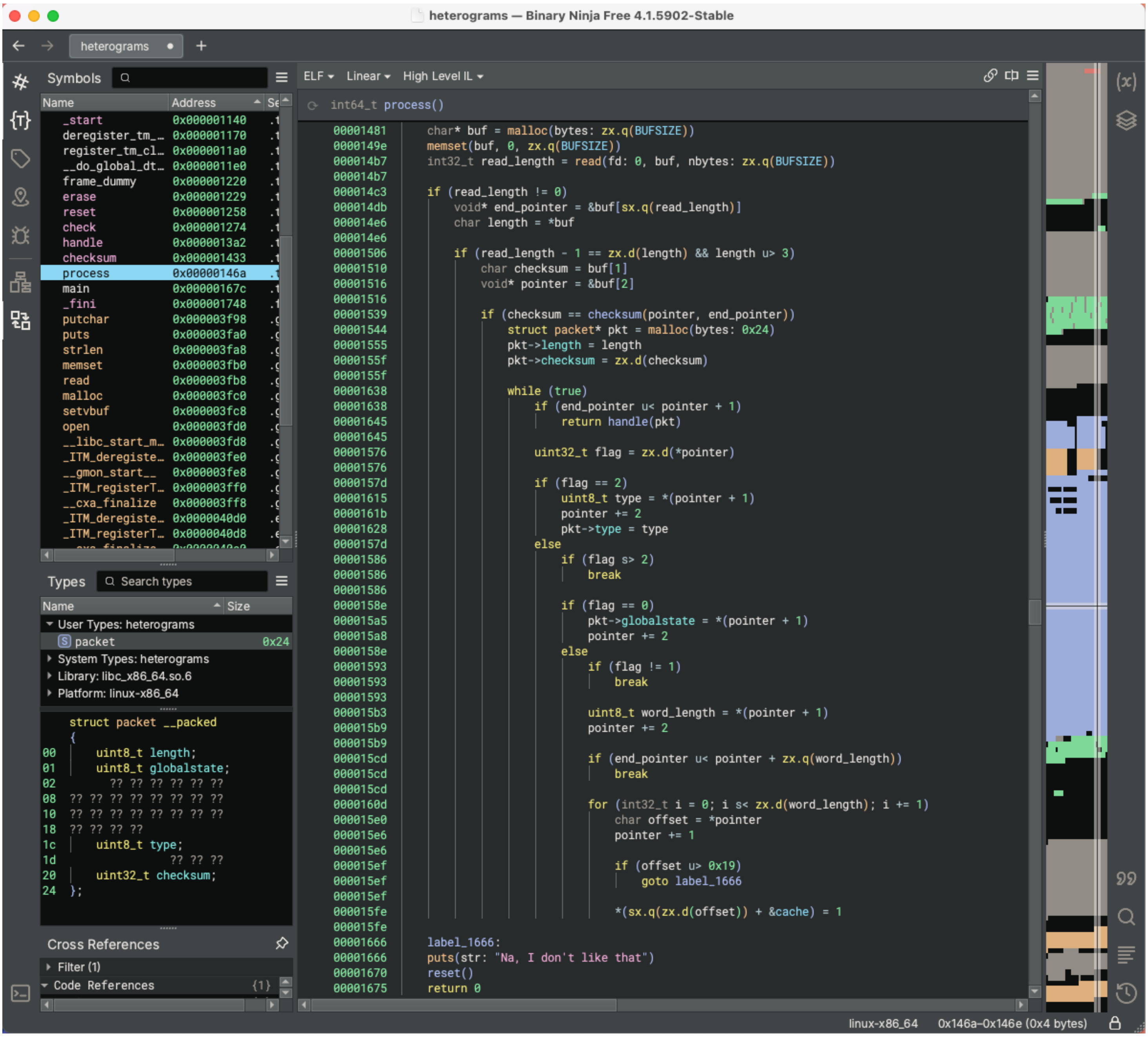
In this CTF challenge, we must send some data to get the flag; nothing further useful is provided.

We directly open the binary file `heterograms` with Binary Ninja to inspect the High-Level IL of the `main` function.



From the code of the `main` function, the program processes a series of messages in an infinite loop, in which the `process` function is called. If the `process` function returns 1, the flag will be displayed.

To figure out what operation the `process` function does, we continuously inspect its High-Level IL to see if we can find something.



It should be noted that we have already renamed unreadable symbols and variables, and defined a structure called "packet" to make the code more easily understandable. The packet-type structure format seems to be:

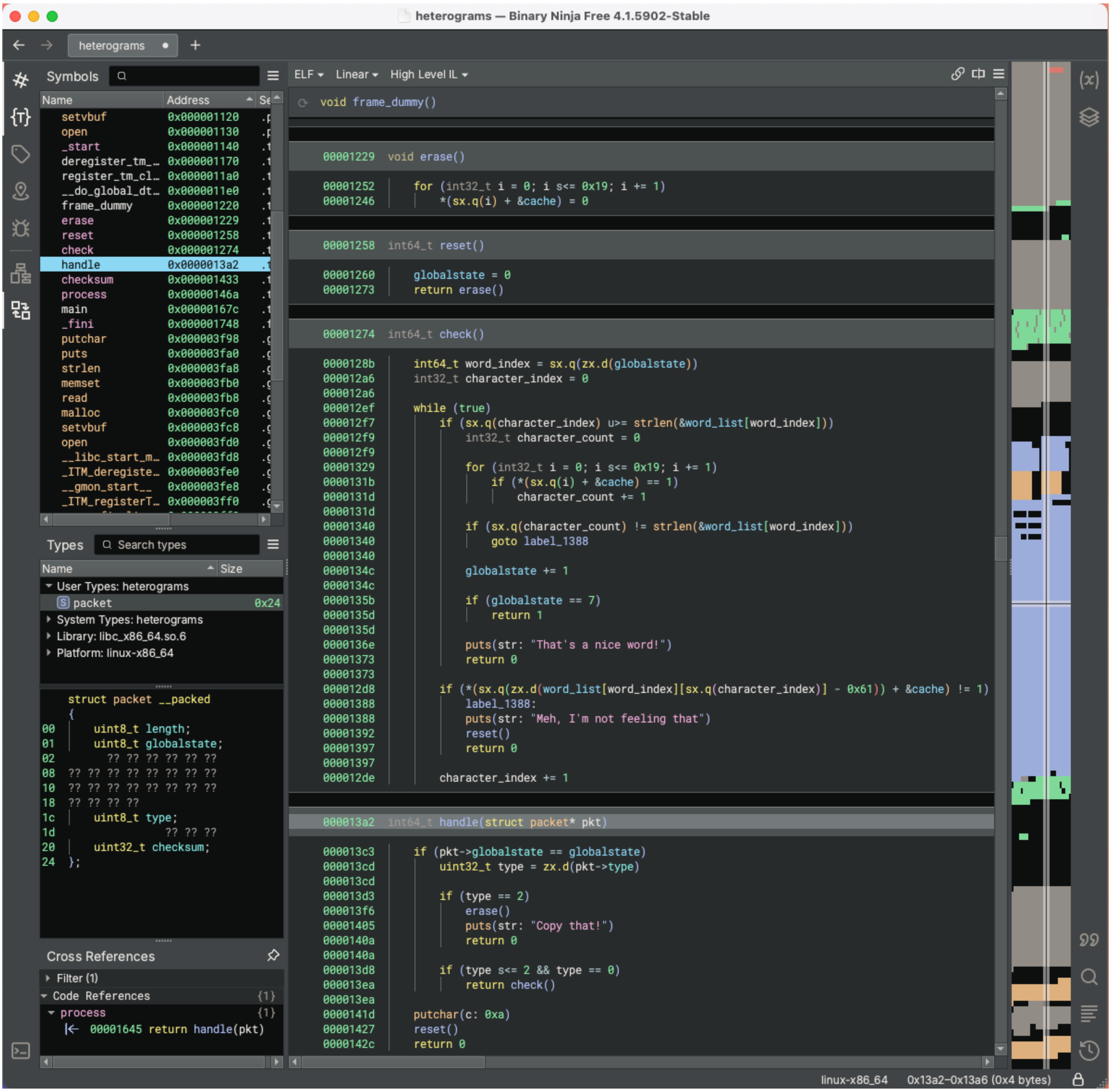
```
struct packet __packed
{
    uint8_t length; # 0x1
    uint8_t globalstate;    # 0x1
    __padding char _2[0x1a];
    uint8_t type;    # 0x1
    __padding char _1d[3];
    uint32_t checksum;  # 0x4
};  # 0x24
```

From the code of the `process` function, we can tell that:

1. It allocates a buffer and reads an input message into it.
2. It checks if the read length minus 1 matches the first byte of the buffer and if the length is greater than 3.
3. It checks if the second byte of the buffer matches the checksum of the rest of the buffer.
4. If the first two bytes of the buffer are valid, it allocates a packet-type structure of size 0x24 and starts parsing the buffer.
5. The first byte of the buffer is treated as the packet's `length` byte. The second byte of the buffer is treated as the packet's `checksum` data.

6. It iterates through the rest of the buffer, processing different flags:
   - Flag 0: Set the packet's `globalstate` byte as the byte after the buffer's flag byte. This part of the buffer is like `{flag = 0 | globalstate}`.
   - Flag 1: Process a word, updating a `cache` based on the word length number of offsets, setting each `offset` byte of the `cache` to 1. This part of the buffer is like `{flag = 1 | word_length | offsets}`. The `offsets` part has `word_length` number of bytes.
   - Flag 2: Set the packet's `type` byte as the byte after the buffer's flag byte. This part of the buffer is like `{flag = 2 | type}`.
7. If parsing completes successfully, it calls the `handle` function and returns its value.

Now, let's take a look at the High-Level IL of the `handle` function.



From the code of the `handle` function, we can tell that:

1. It checks if the packet's `globalstate` byte matches the current global variable `globalstate`.
2. If so, it processes based on the packet's `type` byte:
   - Type 0: Call the `check` function and return its value.

- Type 2: Call the `erase` function, print "Copy that!", and return 0.

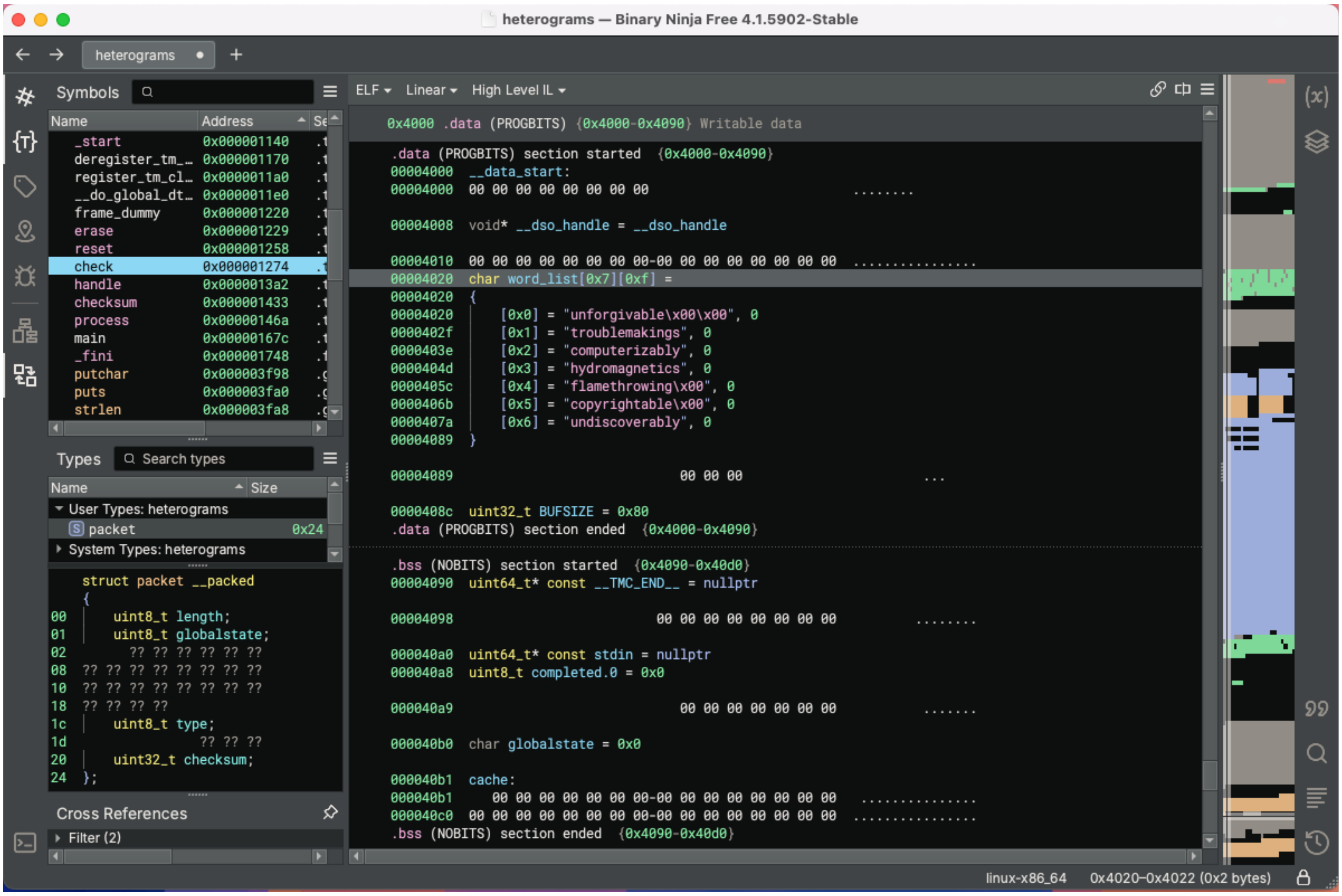
3. If the check on `globalstate` doesn't succeed, it calls the `reset` function and returns 0.

From the code of the `check` function, we can tell that:

1. `word_index` is set based on the current global variable `globalstate`.
2. It iterates through characters of a word from `word_list[word_index]`
3. For each character `word_list[word_index][character_index]`, it checks if the offset byte by the character's ASCII value minus 0x61 from the `cache` is set to 1.
4. If all characters pass the check, and the `cache` has exactly the same counts of bytes set as the word length:
   - It increments the global variable `globalstate`.
   - If the global variable `globalstate` reaches 7, it returns 1 (success condition); Otherwise, it prints "That's a nice word!" and returns 0.
5. If any character is not found or the count doesn't match, it prints "Meh, I'm not feeling that", calls the `reset` function, and returns 0

From the code of the `erase` function, we can tell that it sets each element of the `cache` to 0, which effectively clears the `cache`. From the code of the `reset` function, we can tell that it sets the global variable `globalstate` to 0 and calls the `erase` function to clear the `cache`.

The `word_list` contains 7 words, each of which has 15 (0xf) characters. The content of the `word_list` is shown below, helping to understand better.



In summary, to capture the flag of this challenge, which is to let the `process` function return 1, we need to send 7 check messages for seven words and 6 erase messages for the first six words in the `word_list`.

For each word `word_list[i]`:

- Three-part format of the check message:
  - `{flag = 0 | globalstate = i}`

- `{flag = 1 | word_length = len(word_list[i]) | offsets = (word_list[i][j] - 0x61) for j in word_length}`
    - `{flag = 2 | type = 0}`
- Two-part format of the erase message:
    - `{flag = 0 | globalstate = i + 1}`
    - `{flag = 2 | type = 2}`

It should be noted that for each word, its erase message is sent just after its check message.

Therefore, a script using Pwntools is written to act like a client for this protocol-like challenge, which sends this series of messages to the server. Part of the script is shown below.

```
from pwn import *
......
    p = process(CHALLENGE)
......
for word in word_list:

    word_payload0 = b"\x00" + globalstate.to_bytes(1, byteorder='big')
    word_payload1 = b"\x01" + len(word).to_bytes(1, byteorder='big') + bytes([c - 0x61 for c in
word.encode()])
    word_payload2 = b"\x02\x00"
    word_payload = word_payload0 + word_payload1 + word_payload2
    word_checksum = (~ sum(word_payload) & 0xff).to_bytes(1, byteorder='big')
    word_data = word_checksum + word_payload
    word_length = (len(word_data)).to_bytes(1, byteorder='big')
    word_message = word_length + word_data

    p.send(word_message)
    log.info(f"Sending message for checking word '{word}': {word_message}")

    globalstate += 1
    if globalstate == 7:
        print(p.recvuntil(b"\n").decode())
        break
    print(p.recvuntil(b"That's a nice word!\n").decode())

    erase_payload0 = b"\x00" + globalstate.to_bytes(1, byteorder='big')
    erase_payload2 = b"\x02\x02"
    erase_payload = erase_payload0 + erase_payload2
    erase_checksum = (~sum(erase_payload) & 0xff).to_bytes(1, byteorder='big')
    erase_data = erase_checksum + erase_payload
    erase_length = (len(erase_data)).to_bytes(1, byteorder='big')
    erase_message = erase_length + erase_data

    p.send(erase_message)
    log.info(f"Sending message for erasing word '{word}': {erase_message}")

    print(p.recvuntil(b"Copy that!\n").decode())

p.interactive()
```

The console output of the script execution is shown below.

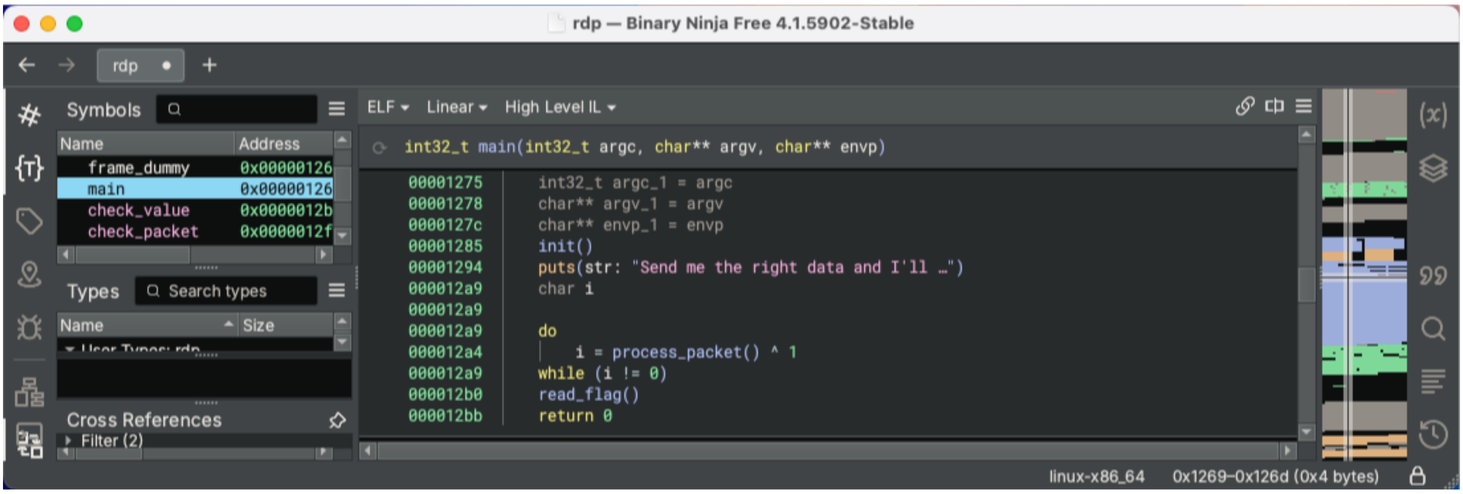
```
root@17b95fe8a8e6:~/wk4/heterograms# python3 heterograms.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1271: Done
hello, xz4344. Please wait a moment...
Send me some data to get the flag!

[*] Sending message for checking word 'unforgivable': b'\x13x\x00\x00\x01\x0c\x14\r\x05\x0e\x11\x06\x08\x15\x00\x01\x0b\x04\x02\x00'
That's a nice word!

[*] Sending message for erasing word 'unforgivable': b'\x05\xfa\x00\x01\x02\x02'
Copy that!

[*] Sending message for checking word 'troublemakings': b'\x15T\x00\x01\x01\x0e\x13\x11\x0e\x14\x01\x0b\x04\x0c\x00\n\x08\r\x06\x12\x02\x00'
That's a nice word!

[*] Sending message for erasing word 'troublemakings': b'\x05\xf9\x00\x02\x02\x02'
Copy that!

[*] Sending message for checking word 'computerizably': b'\x15@\x00\x02\x01\x0e\x02\x0e\x0c\x0f\x14\x13\x04\x11\x08\x19\x00\x01\x0b\x18\x02\x00'
That's a nice word!

[*] Sending message for erasing word 'computerizably': b'\x05\xf8\x00\x03\x02\x02'
Copy that!

[*] Sending message for checking word 'hydromagnetics': b'\x15X\x00\x03\x01\x0e\x07\x18\x03\x11\x0e\x0c\x00\x06\r\x04\x13\x08\x02\x12\x02\x00'
That's a nice word!

[*] Sending message for erasing word 'hydromagnetics': b'\x05\xf7\x00\x04\x02\x02'
Copy that!

[*] Sending message for checking word 'flamethrowing': b'\x14a\x00\x04\x01\r\x05\x0b\x00\x0c\x04\x13\x07\x11\x0e\x16\x08\r\x06\x02\x00'
That's a nice word!

[*] Sending message for erasing word 'flamethrowing': b'\x05\xf6\x00\x05\x02\x02'
Copy that!

[*] Sending message for checking word 'copyrightable': b'\x14j\x00\x05\x01\r\x02\x0e\x0f\x18\x11\x08\x06\x07\x13\x00\x01\x0b\x04\x02\x00'
That's a nice word!

[*] Sending message for erasing word 'copyrightable': b'\x05\xf5\x00\x06\x02\x02'
Copy that!

[*] Sending message for checking word 'undiscoverably': b'\x15L\x00\x06\x01\x0e\x14\r\x03\x08\x12\x02\x0e\x15\x04\x11\x00\x01\x0b\x18\x02\x00'
flag{s3r1aL1z3d_d4t4_and_ST4T3_m4ch1n3s_e1287c2b087ac0f2}

[*] Switching to interactive mode

$
[*] Interrupted
[*] Closed connection to offsec-chalbroker.osiris.cyber.nyu.edu port 1271
```

The captured flag is `flag{s3r1aL1z3d_d4t4_and_ST4T3_m4ch1n3s_e1287c2b087ac0f2}`.

Rudimentary Data Protocol (100 pts)

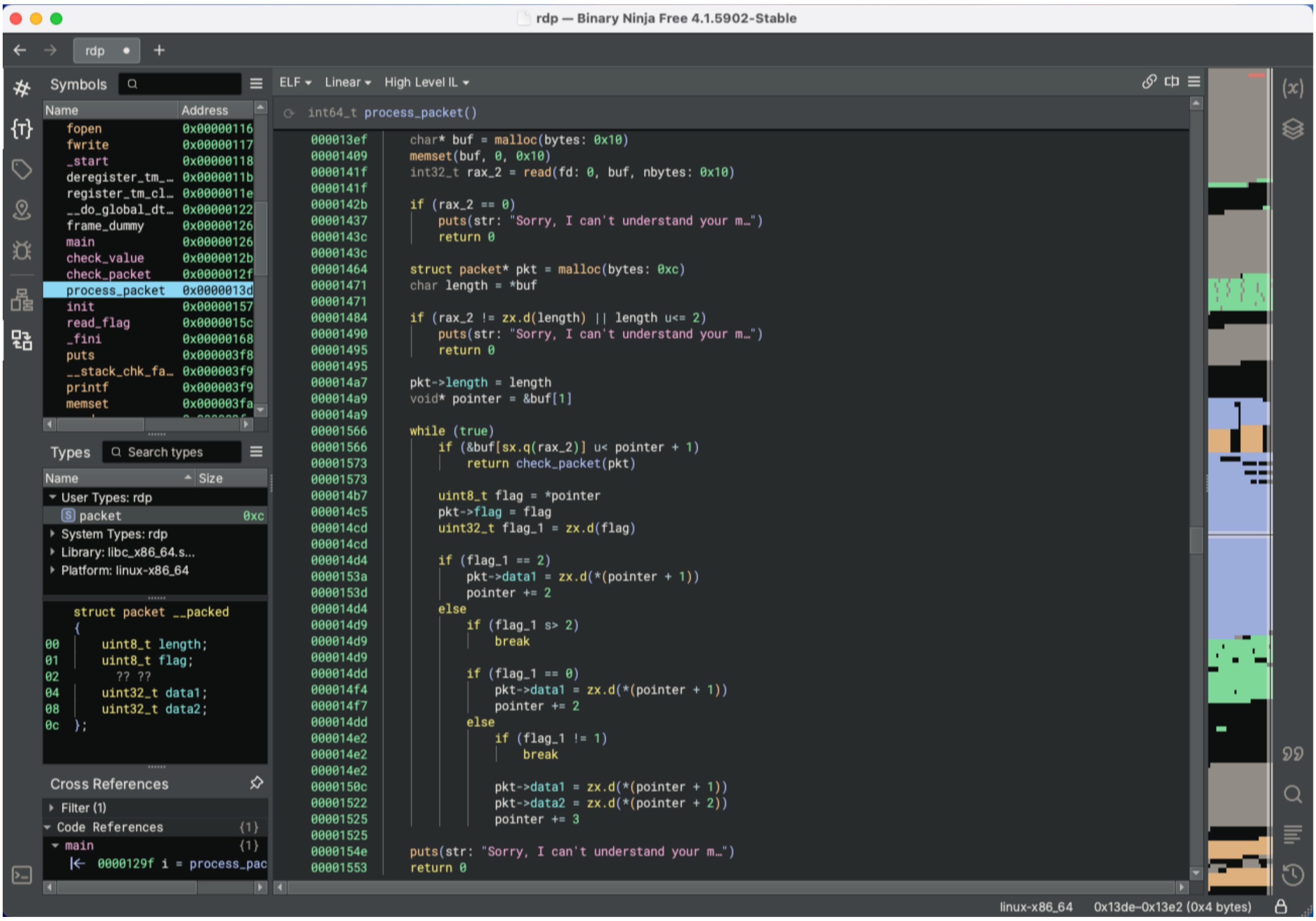
In this CTF challenge, we must send the right data to get the flag; nothing further useful is provided.

We directly open the binary file `rdp` with Binary Ninja to inspect the High-Level IL of the `main` function.



From the code of the `main` function, the program processes a series of messages in a loop until a certain condition is met, then reads a flag. Inside the loop, it calls a function `process_packet` and XORs the result with 1. The loop continues until the result of this operation is 0. In other words, the loop will exit only if the `process_packet` function returns a value of 1.

To figure out what operation the `process_packet` function does, we continuously inspect its High-Level IL to see if we can find something.



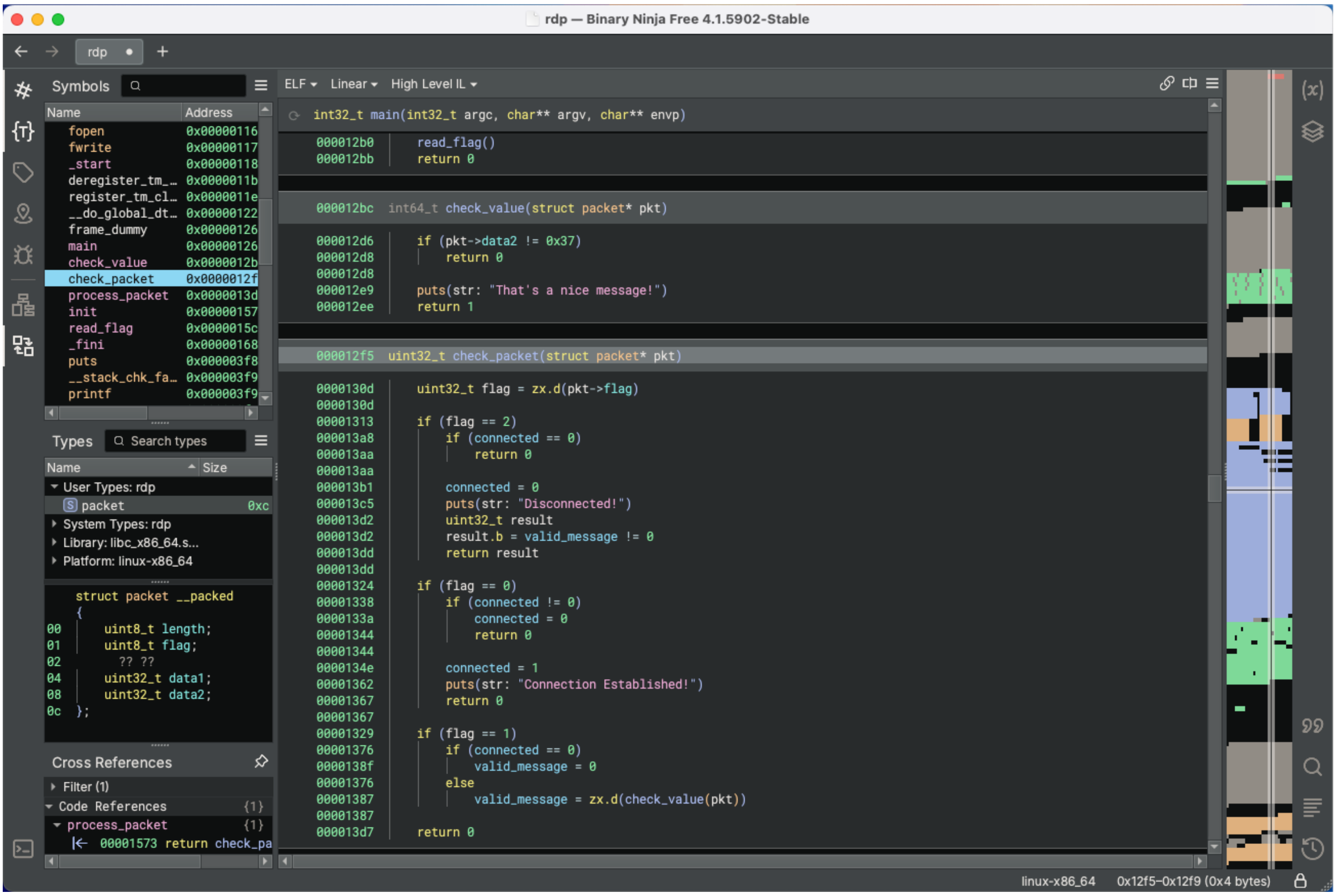
It should be noted that we have already renamed unreadable symbols and variables, and defined a structure called "packet" to make the code more easily understandable. The packet-type structure format seems to be:

```
struct packet __packed
{
    uint8_t length; # 0x1
    uint8_t flag;  # 0x1
    __padding char _2[2];
    uint32_t data1;  # 0x4
    uint32_t data2; # 0x4 optional
};  # 0xc
```

From the code of the `process_packet` function, we can tell that:

1. It allocates a 16-byte buffer and reads an input message up to 16 bytes from the standard input.
2. It allocates a packet-type structure of 12 bytes.
3. It checks if the read length matches the first byte of the buffer and if the length is greater than 2. If not, it prints an error and returns 0.
4. The first byte of the buffer is treated as the packet's `length` byte.
5. It enters a loop to process the packet data:
   - The second byte of the buffer is treated as the packet's `flag` byte.
   - For flag 0 or 2: It reads one additional byte into the packet-type structure as `data1`.
   - For flag 1: It reads two additional bytes into the packet-type structure as `data1` and `data2`.
   - If the flag is not 0, 1, or 2, it breaks the loop and returns an error.
6. If it successfully processes all the data, it calls the `check_packet` function and returns its result.
7. If it runs out of the buffer or encounters an invalid flag, it prints an error message and returns 0.

Now, let's take a look at the High-Level IL of the `check_packet` function.



From the code of the `check_packet` function, we can tell that:

1. It takes a packet-type structure as input.
2. It handles three different packet types based on the `flag` value:
    - Flag 0 (establish a connection): If already connected, set `connected` to 0 and return 0; Otherwise, set `connected` to 1, print "Connection Established!" and return 0.
    - Flag 1 (communicate information): If not connected, set `valid_message` to 0; If connected, call the `check_value` function and store the result in `valid_message`.
    - Flag 2 (Disconnect): If not connected, return 0; Otherwise, set `connected` to 0, print "Disconnected!", and return 1 if `valid_message` is non-zero, 0 otherwise.
    - For any other value, it returns 0.

From the code of the `check_value` function, we can tell that:

1. It takes a packet-type structure as input.
2. It checks if the `data2` of the packet is equal to 0x37.
3. If it matches, print "That's a nice message!" and return 1; Otherwise, return 0.

In summary, to capture the flag of this challenge, we need to send three messages to let the `process_packet` function return 1.

1. The first message is to establish a connection, whose first byte is the `length` 3, second byte is the `flag` 0, and third byte is arbitrary for `data1`.
2. The second message is to communicate information, whose first byte is the `length` 4, second byte is the `flag` 1, third byte is arbitrary for `data1`, and fourth byte is 0x37 for `data2`.
3. The third message is to disconnect, whose first byte is the `length` 3, second byte is the `flag` 2, and third byte is arbitrary for `data1`.

Therefore, a script using Pwntools is written to act like a client for this protocol-like challenge, which sends these three messages to the server. Part of the script is shown below.

```python
from pwn import *
......
p = process(CHALLENGE)
......
print(p.recvuntil(b"Send me the right data and I'll give you the flag!\n").decode())
msg_1 = b"\x03\x00\xff"
p.send(msg_1)
log.info(f"Sending a message to establish a connection: {msg_1}")
print(p.recvuntil(b"Connection Established!\n").decode())
msg_2 = b"\x04\x01\xff\x37"
p.send(msg_2)
log.info(f"Sending a message to communicate information: {msg_2}")
print(p.recvuntil(b"That's a nice message!\n").decode())
msg_3 = b"\x03\x02\xff"
p.send(msg_3)
log.info(f"Sending a message to disconnect: {msg_3}")
print(p.recvuntil(b"Disconnected!\n").decode())
print(p.recvall().decode())

p.interactive()
```

The console output of the script execution is shown below.

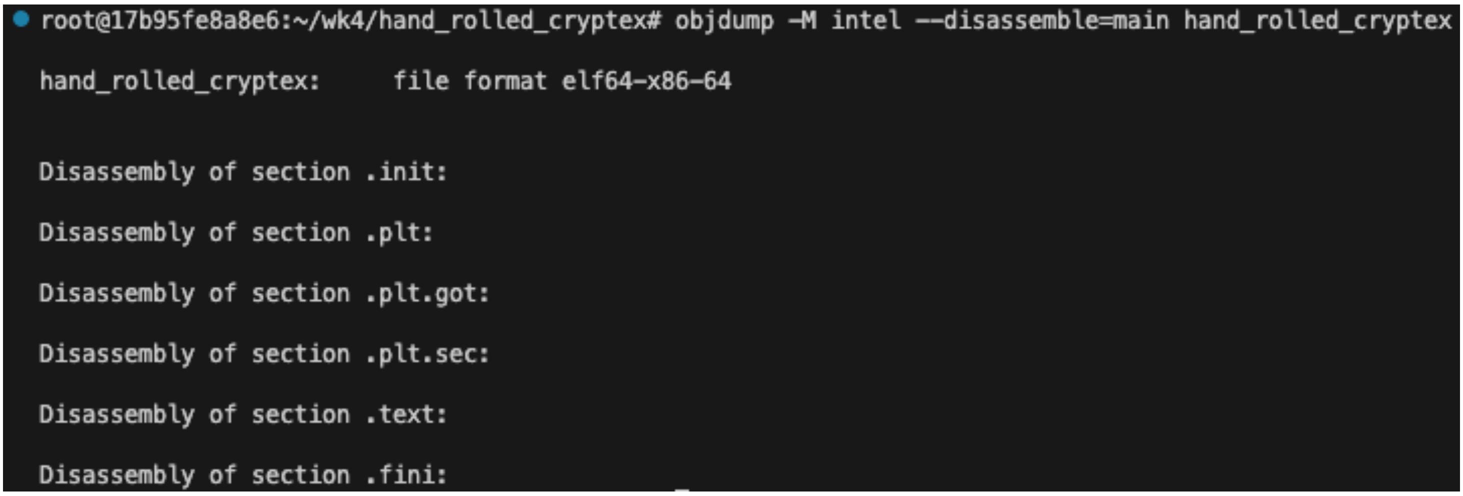
```
root@17b95fe8a8e6:~/wk4/rdp# python3 rdp.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1272: Done
hello, xz4344. Please wait a moment...
Send me the right data and I'll give you the flag!

[*] Sending a message to establish a connection: b'\x03\x00\xff'
Connection Established!

[*] Sending a message to communicate information: b'\x04\x01\xff7'
That's a nice message!

[*] Sending a message to disconnect: b'\x03\x02\xff'
Disconnected!

[+] Receiving all data: Done (85B)
[*] Closed connection to offsec-chalbroker.osiris.cyber.nyu.edu port 1272

        Here's your flag, friend: flag{w3_r34lly_l1k3_s3r14l1z3d_d4t4!_cba709f17019650c}



[*] Switching to interactive mode
[*] Got EOF while reading in interactive
$
[*] Interrupted
```

The captured flag is `flag{w3_r34lly_l1k3_s3r14l1z3d_d4t4!_cba709f17019650c}`.

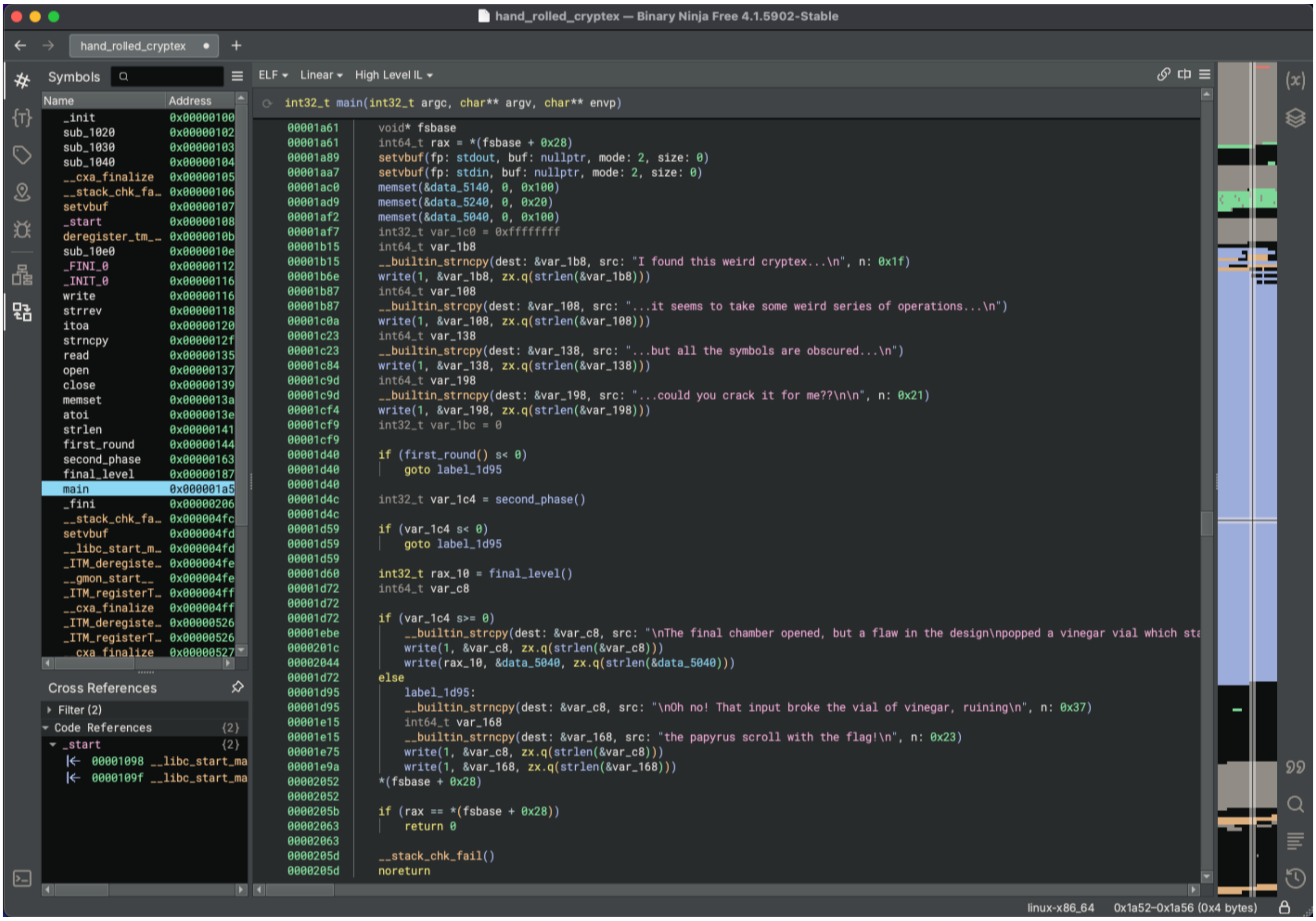
Hand Rolled Cryptex (150 pts)

In this CTF challenge, with symbols obscured, we must complete a series of operations to get the flag; nothing further useful is provided.

As shown below, since the `objdump` command outputs nothing for the binary file `hand_rolled_cryptex`, we can clearly learn that it is a stripped binary. Stripping effectively removes all user-defined variables and function names in the compiled output.



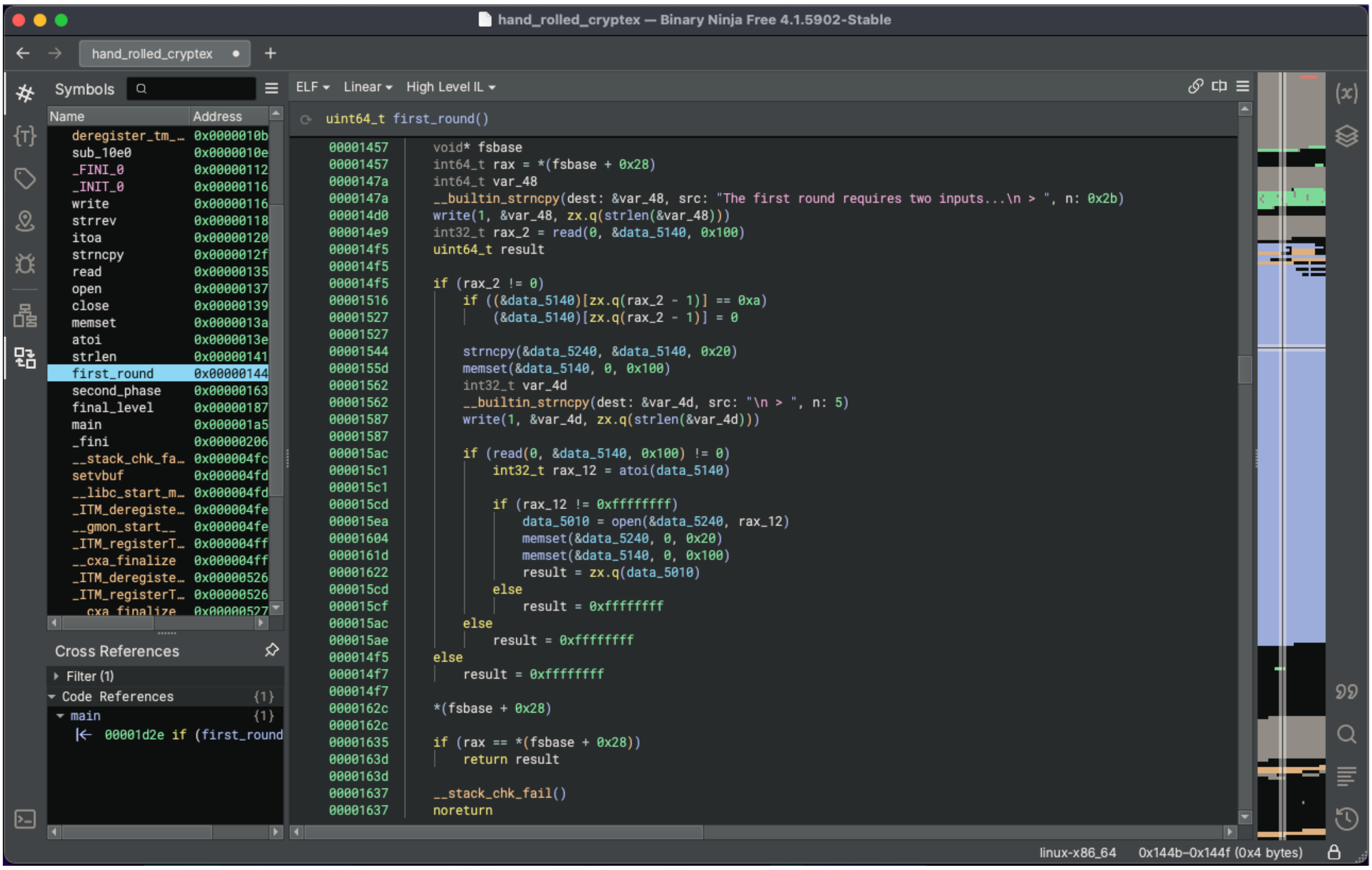
Thus, while opening the binary file `hand_rolled_cryptex` with Binary Ninja to find helpful information, we have to rename unreadable symbols and variables, which makes it conveniently understandable. The following is the High-Level IL of the `main` function.



In short, the `main` function is structured like a puzzle-solving game with three stages (via the `first_round`, `second_phase`, and `final_level` functions).

1. It prints narrative messages about a "cryptex".
2. If the `first_round` function returns a negative value, it jumps to a failure label `label_1d95`. The outcome of the `first_round` function influences what happens next.
3. The `second_phase` function is called, whose result is stored in `var_1c4`. If this returns a negative value, it also jumps to the failure label `label_1d95`. The outcome of the `second_phase` function influences what happens next.
4. The `final_level` function is called, and its result is stored in `rax_10`. The outcome of the `final_level` function influences what happens next.
5. If both `first_round` and `second_phase` succeed, it prints a success message and writes the data stored in `data_5040` (possibly contains important information like the flag) to the place referred to by `rax_10` (obviously equals to integer `1` as the standard output, aka the console), using the `write` system call.

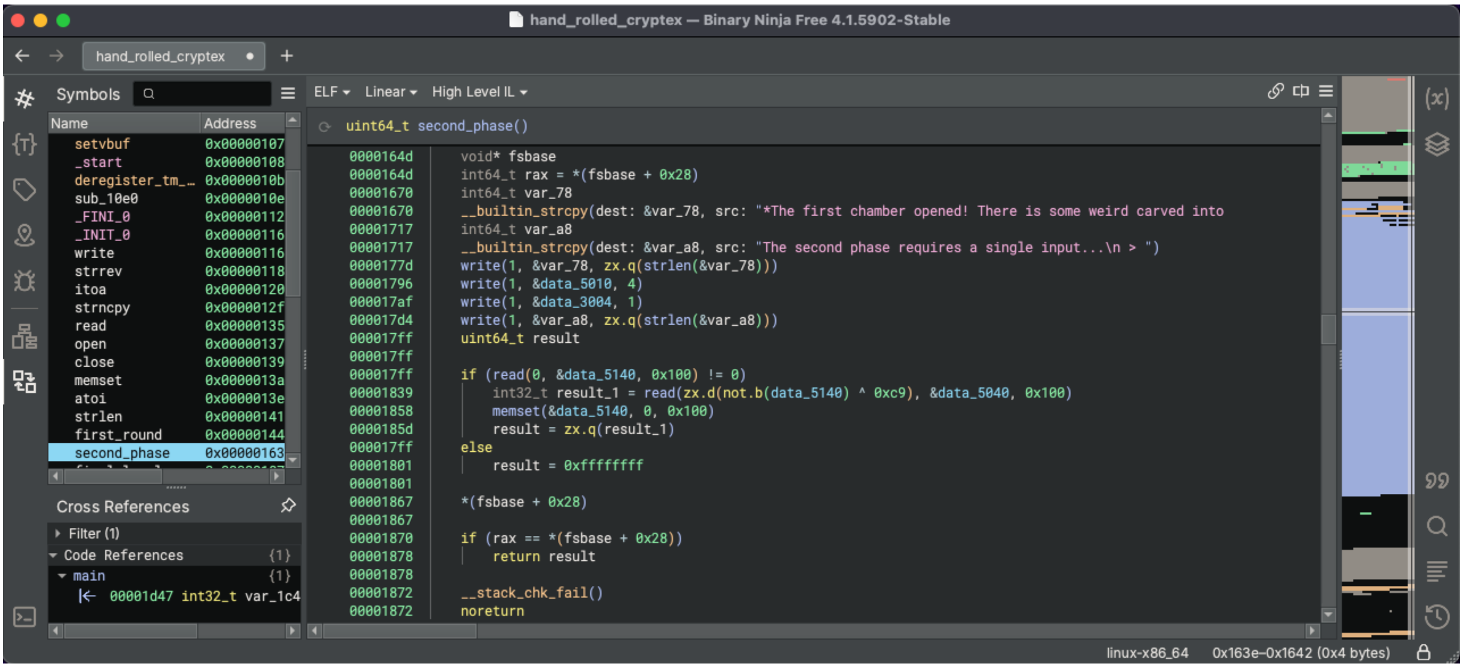
Now, let's inspect the High-Level IL of the function of the three stages.



From the High-Level IL of the `first_round` function shown above, we can see the following main operations:

1. It asks the user for two inputs: a filename (a string) and an access mode (a one-digit integer).
2. It reads the inputs, processes them, and attempts to open the file with the provided mode, using the `open` system call: `data_5010 = open(&data_5240, rax_12)`, where `data_5240` is the filename, `rax_12` is the access mode, and `data_5010` stores the file descriptor.
3. If the inputs are valid, it returns the file descriptor based on the `open` system call.

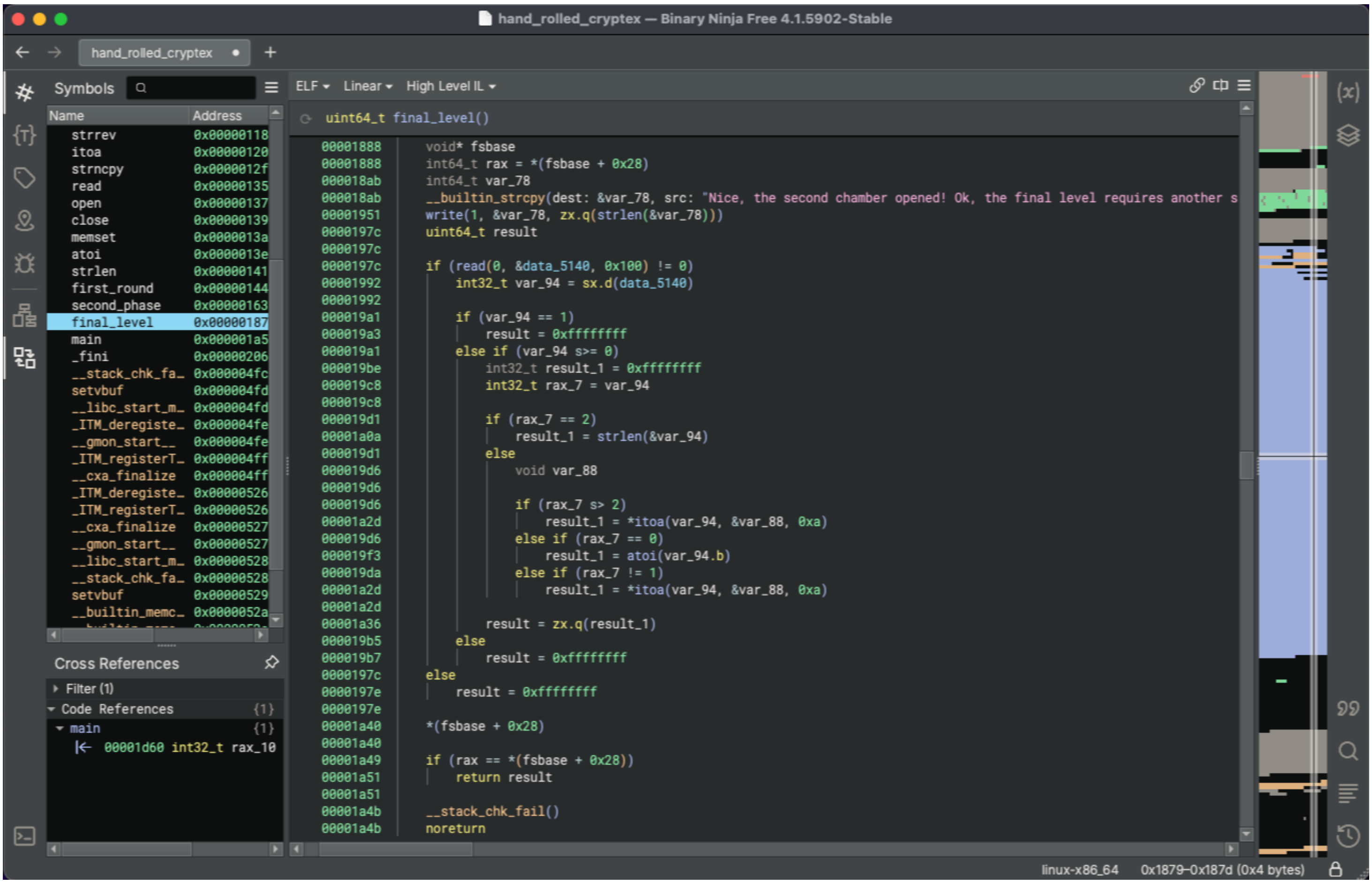
Thus, as for the first stage: Based on the common sense of former challenges, the flag is always stored in a file called `flag.txt`; Besides, to read the file content later, we can choose `O_RDONLY` as the access mode for the `open` system call, which is integer `0`.



From the High-Level IL of the `second_phase` function shown above, we can see the following main operations:

1. It prints the file descriptor returned by the `open` system call in the `first_round` function, which is stored in `data_5010`.
2. It asks the user for a single input (a character).
3. The user's input is processed, and a `read` system call is performed based on the processed input: `result_1 = read(zx.d(not.b(data_5140) ^ 0xc9), &data_5040, 0x100)`, where `data_5140` is the input, `data_5040` stores the read content, which is the flag, and `result_1` is the number of bytes read.
4. If successful, the result of the `read` system call, which is the number of bytes read, is returned.

Thus, as for the second stage: We can first acquire the printed file descriptor from the console as `fd`, and then, according to how the input is processed, calculate `input_2` by `input_2 = ~ (fd ^ 0xc9) & 0xff`, which is the ASCII value of the input character.



From the High-Level IL of the `final_level` function shown above, we can see the following main operations:

1. It asks the user for a single input (a character or a string).
2. It forces the input character or string to be converted to an integer.
3. Depend on the value of this integer:
   - If the integer is equal to 0, the function calls `atoi` on the least significant byte of the integer and stores its return value as the result.
   - If the integer is equal to 1, the function stores the error value `0xffffffff` as the result.
   - If the integer is equal to 2, the function calls `strlen` on the integer and stores its return value as the result.
   - If the integer is greater than 2, the function calls `itoa` on the integer and stores the first byte of its return value as the result.
4. The function returns the result if the input is valid and successfully processed.

Thus, as for the third stage: Since the return value of the `final_level` function has to be `1` as the standard output for the `write` system call in the `main` function, the `final_level` function must return `1`, taking `input_3 = 2`, which is the input character's unpacked version from raw bytes.

To sum up, this challenge is a process of correctly calling the `open`, `read`, and `write` system calls sequentially to retrieve the flag from the file and display it in the console. Our answers should satisfy the consistency of these three system calls.

Therefore, a script using Pwntools is written to send three stages of answers to the server. Part of the script is shown below.

```
from pwn import *
......
p = process(CHALLENGE)
......
print(p.recvuntil(b"The first round requires two inputs...\n > ").decode())
filename = "flag.txt"
p.sendline(filename.encode())
print(p.recvuntil(b"> ").decode())
```

```python
mode = 0
p.send(str(mode).encode())
log.info(f"Sending filename {filename} and access mode {mode} (O_RDONLY) for the open system call")

print(p.recvuntil(b"*The first chamber opened! There is some weird carved into                    the interior...\n").decode())
fd = p.recv(4)
print(fd)
fd = u32(fd)
log.info(f"Receiving returned fd {fd} from the previous open system call")
print(p.recvuntil(b"The second phase requires a single input...\n > ").decode())
input_2 = ~ (fd ^ 0xc9) & 0xff
p.send(p8(input_2))
log.info(f"Sending character '{chr(input_2)}' with ASCII value {hex(input_2)} to satisfy the condition input_2 = ~ (fd ^ 0xc9) & 0xff")

print(p.recvuntil(b"Nice, the second chamber opened! Ok, the final level requires another single input...\n > ").decode())
input_3 = 2
p.send(p8(input_3))
log.info(f"Sending character '{chr(input_3)}' to get 1 as standard output for the later write system call")
print(p.recvall().decode())

p.interactive()
```

The console output of the script execution is shown below.

```
root@17b95fe8a8e6:~/wk4/hand_rolled_cryptex# python3 hand_rolled_cryptex.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1273: Done
hello, xz4344. Please wait a moment...
I found this weird cryptex...
...it seems to take some weird series of operations...
...but all the symbols are obscured...
...could you crack it for me??

The first round requires two inputs...
 >

 >
[*] Sending filename flag.txt and access mode 0 (O_RDONLY) for the open system call
*The first chamber opened! There is some weird carved into                    the interior...

b'\x06\x00\x00\x00'
[*] Receiving returned fd 6 from the previous open system call

The second phase requires a single input...
 >
[*] Sending character '0' with ASCII value 0x30 to satisfy the condition input_2 = ~ (fd ^ 0xc9) & 0xff
Nice, the second chamber opened! Ok, the final level requires another single input...
 >
[*] Sending character '\x02' to get 1 as standard output for the later write system call
[+] Receiving all data: Done (237B)
[*] Closed connection to offsec-chalbroker.osiris.cyber.nyu.edu port 1273

The final chamber opened, but a flaw in the design
popped a vinegar vial which started to eat away at the papyrus
scroll inside. You hold it up, trying to decipher the text... flag{str1PP3d_B1N4R135_4r3_S0o0_much_FUN!_d13dce2c4a67a206}

[*] Switching to interactive mode
[*] Got EOF while reading in interactive
$
[*] Interrupted
```

The captured flag is `flag{str1PP3d_B1N4R135_4r3_S0o0_much_FUN!_d13dce2c4a67a206}`.