Lab1

Task1

Brief introduction:

- Compile `simple.c`;
- Run the program and output the original string and its hexadecimal representation.

Experimental Objectives

- Get familiar with the compilation process of C projects.
- Learn how to print and debug hexadecimal data in the program.
- Prepare the message processing tool for subsequent encryption operations.

Code:

Lab1 > Task1 > C simple.c > ...

```c
#include <stdio.h>

// Hard-coded message.
char input[] = "Hello world!\n";
// Compute size of message (including any terminating null byte).
int input_size = sizeof(input)/sizeof(input[0]);

// Print out hex encoding of a buffer.
void dump_hex(char* buf, int s) {
    for (int n = 0; n < s; n++) {
        printf("0x%02hhx, ", buf[n]);
    }
}

// Print out hex encoding of a buffer as a C variable definition.
void dump_hex_var(char* name, char* buf, int s) {
    printf("char %s[] = {", name);
    dump_hex(buf, s);
    printf("};\n");
}

int main() {
    printf("Printing output:\n");
    fwrite(input, 1, input_size, stdout);
    printf("Dumping hex:\n");
    dump_hex_var("output", input, input_size);
}
```

Result:

```
PS D:\CWORKSPACE\Lab1\Task1> gcc -o simple simple.c
PS D:\CWORKSPACE\Lab1\Task1> ./simple
Printing output:
Hello world!
Dumping hex:
char output[] = {0x48, 0x65, 0x6c, 0x6c, 0x6f, 0x20, 0x77, 0x6f, 0x72, 0x6c, 0x64, 0x21, 0x0a, 0x00, };
PS D:\CWORKSPACE\Lab1\Task1>
```

Task2

Brief introduction

- Decrypt the encrypted message (including the key, nonce, MAC, and ciphertext) in comic.h;
- Output the original plaintext.

Experimental Objectives

- Learn the basic structure of symmetric encryption;
- Practice using the crypto_unlock() function;
- Understand the roles of nonce and MAC (ensuring uniqueness and authentication).

Code

Lab1 > Task2 > C task2.c > ...

```c
#include <stdio.h>
#include "comic.h"
#include "monocypher.h"

// Hard-coded message.
char input[] = "Hello world!\n";
// Compute size of message (including any terminating null byte).
int input_size = sizeof(input)/sizeof(input[0]);

// Print out hex encoding of a buffer.
void dump_hex(char* buf, int s) {
    for (int n = 0; n < s; n++) {
        printf("0x%02hhx, ", buf[n]);
    }
}

// Print out hex encoding of a buffer as a C variable definition.
void dump_hex_var(char* name, char* buf, int s) {
    printf("char %s[] = {", name);
    dump_hex(buf, s);
    printf("};\n");
}

int main()
{
int len = sizeof(cipher_text);
char p_text[sizeof(cipher_text)];
crypto_unlock(p_text, key, nonce, mac, cipher_text, len);
printf("p_text is: %s", p_text);
printf("\nLength of cipher_text is: %d", len);
return 0;
}

```

Result:

```
PS D:\CWORKSPACE\Lab1\Task2> gcc -o task2 task2.c monocypher.c
PS D:\CWORKSPACE\Lab1\Task2> ./task2
p_text is: A crypto nerd's                |  What would
imagination                               |  actually happen:
------------------------------------------+------------------------------
His laptop's encrypted.                   |  His laptop's encrypted.
Let's build a million-dollar              |  Drug him and hit him with
cluster to crack it.                      |  this $5 wrench until
    |                                     |  he tells us the password.
    |        No good! It's                |      |
    |        4096-bit RSA!                |      |              Got it.
    |                   |                 |      |                  |
 Blast! Our             |                 |      |                  |
 evil plan              |                 |      |                  |
 is foiled!             |                 |      |                  |
                        |                 |
   /-\                 /-\                |    /-\                /-\
   \-/                 \-/                |    \-/                \-/
    |                   |                 |     |                  |
   /|\   .---.         /|\                |    /|\      c         /|\
  / | \  |   |        / | \               |   / | \    /         / | \
    |    /===\          |                 |     |     /            |
   / \                 / \                |    / \   o            / \
  /   \               /   \               |   /   \              /   \
-------------------------------------------------------------------------
Actual Actual Reality: Nobody really cares about his secrets.
(Also, I would be hard pressed to find that wrench for $5.

Length of cipher_text is: 1437
PS D:\CWORKSPACE\Lab1\Task2>
```

Task3

Brief introduction:

- Decrypt the ciphertexts in `message1.h`, `message2.h`, and `message3.h`.
- Generate a shared key using `crypto_key_exchange()`.
- Decrypt each message and check if the decryption is successful; Identify which message has been tampered with (by a failed MAC check).

Experimental Objectives

- Understand the basic principle of public-key encryption communication;
- Practice the key exchange process to generate a shared symmetric key;
- Understand the role of the Message Authentication Code (MAC) in verifying integrity.

Code

Lab1 > Task3 > C task3.c > ...

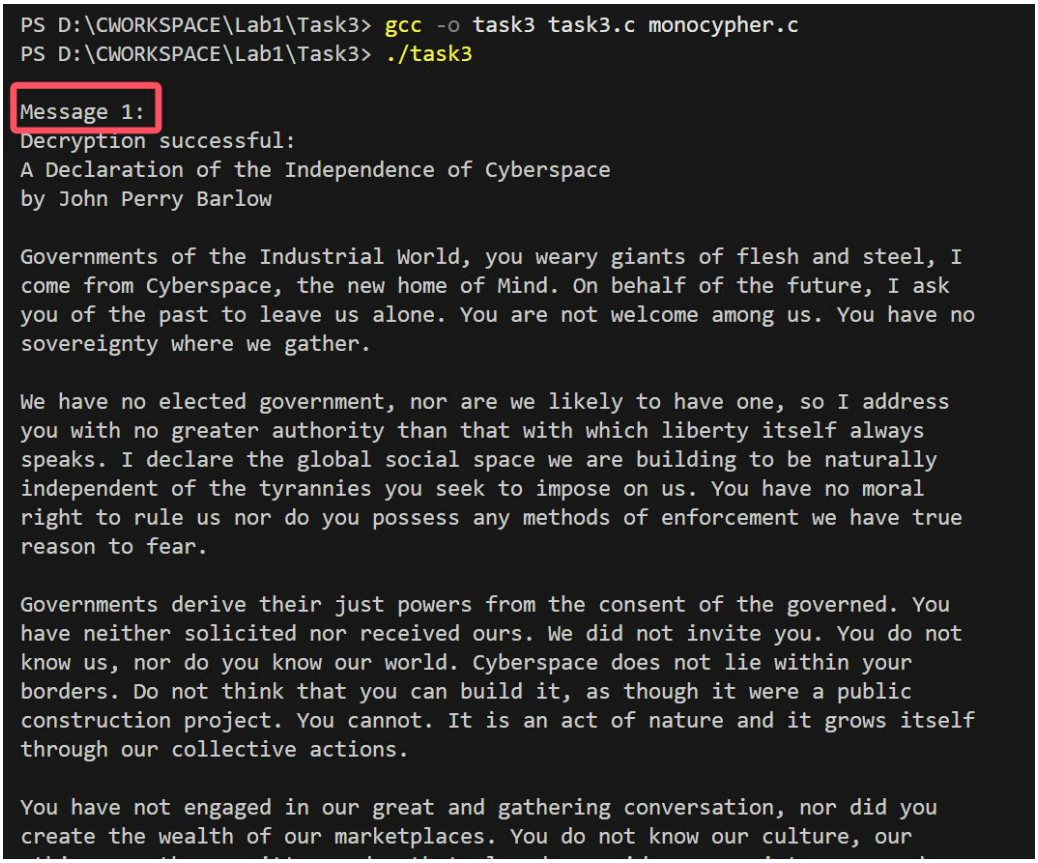
```c
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include "monocypher.h"
#include "keys.h"
#include "message1.h"
#include "message2.h"
#include "message3.h"

void try_decrypt(const char* label, const uint8_t* sender_pk, const uint8_t* rx_sk,
                 const uint8_t* nonce, const uint8_t* mac, const uint8_t* ciphertext, size_t len) {
    // Buffer to store the shared symmetric key calculated from the key exchange
    uint8_t shared_key[32];
    // Perform key exchange to get the shared symmetric key
    crypto_key_exchange(shared_key, rx_sk, sender_pk);
    // Buffer to store the decrypted message
    uint8_t decrypted[len];
    // Decrypt the ciphertext using the shared key, nonce and MAC
    int ok = crypto_unlock(decrypted, shared_key, nonce, mac, ciphertext, len);
    // Print the label of the message
    printf("\n%s:\n", label);
    if (ok == 0) {
        printf("Decryption successful:\n%.*s\n", (int)len, decrypted);
    } else {
        printf("Decryption failed (MAC invalid or corrupted message).\n");
    }
}

int main() {
    // Decrypt the message
    try_decrypt("Message 1", tx1_pk, rx_sk, nonce1, mac1, cipher_text1, sizeof(cipher_text1));
    try_decrypt("Message 2", tx2_pk, rx_sk, nonce2, mac2, cipher_text2, sizeof(cipher_text2));
    try_decrypt("Message 3", tx3_pk, rx_sk, nonce3, mac3, cipher_text3, sizeof(cipher_text3));
    return 0;
}
```

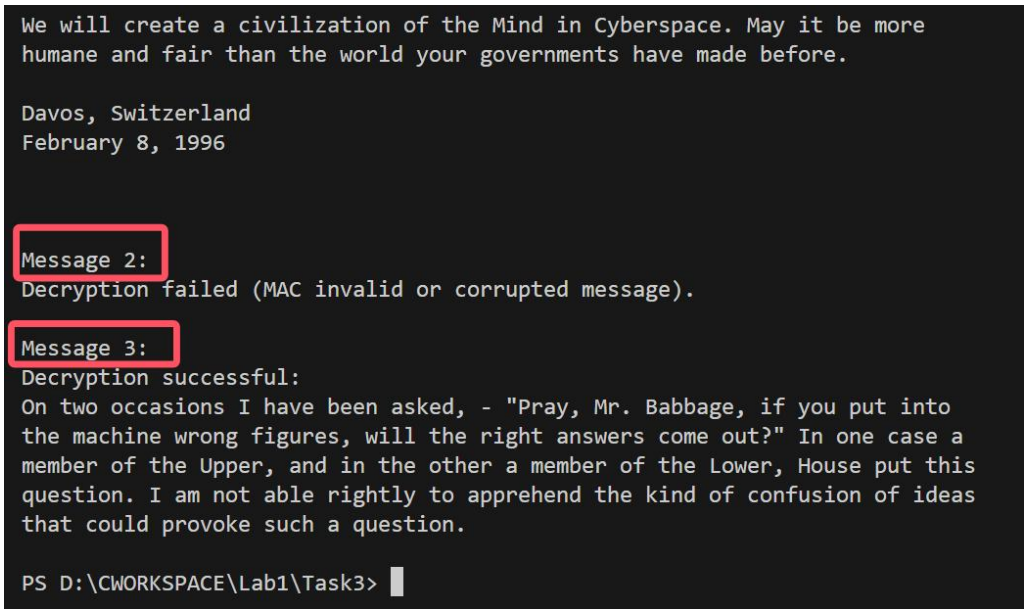
Result

Which message has been corrupted?

The 19th line of the above - mentioned code is used to decrypt the ciphertext using the shared key, nonce, and MAC. If `ok == 0`, it indicates that the decryption is successful; otherwise, it means the decryption fails.

As can be seen from the following running results, `message1` and `message3` have been successfully decrypted, while the decryption of `message2` has failed. We can see the contents of `message1` and `message3` after they are successfully decrypted.





The output of the running results is as follows:

Message 1:
Decryption successful:
A Declaration of the Independence of Cyberspace
by John Perry Barlow

Governments of the Industrial World, you weary giants of flesh and steel, I come from Cyberspace, the new home of Mind. On behalf of the future, I ask you of the past to leave us alone. You are not welcome among us. You have no sovereignty where we gather.

We have no elected government, nor are we likely to have one, so I address

you with no greater authority than that with which liberty itself always speaks. I declare the global social space we are building to be naturally independent of the tyrannies you seek to impose on us. You have no moral right to rule us nor do you possess any methods of enforcement we have true reason to fear.

Governments derive their just powers from the consent of the governed. You have neither solicited nor received ours. We did not invite you. You do not know us, nor do you know our world. Cyberspace does not lie within your borders. Do not think that you can build it, as though it were a public construction project. You cannot. It is an act of nature and it grows itself through our collective actions.

You have not engaged in our great and gathering conversation, nor did you create the wealth of our marketplaces. You do not know our culture, our ethics, or the unwritten codes that already provide our society more order than could be obtained by any of your impositions.

You claim there are problems among us that you need to solve. You use this claim as an excuse to invade our precincts. Many of these problems don't exist. Where there are real conflicts, where there are wrongs, we will identify them and address them by our means. We are forming our own Social Contract. This governance will arise according to the conditions of our world, not yours. Our world is different.

Cyberspace consists of transactions, relationships, and thought itself, arrayed like a standing wave in the web of our communications. Ours is a world that is both everywhere and nowhere, but it is not where bodies live.

We are creating a world that all may enter without privilege or prejudice accorded by race, economic power, military force, or station of birth.

We are creating a world where anyone, anywhere may express his or her beliefs, no matter how singular, without fear of being coerced into silence or conformity.

Your legal concepts of property, expression, identity, movement, and context do not apply to us. They are all based on matter, and there is no matter here.

Our identities have no bodies, so, unlike you, we cannot obtain order by physical coercion. We believe that from ethics, enlightened self-interest, and the commonweal, our governance will emerge. Our identities may be distributed across many of your jurisdictions. The only law that all our constituent cultures would generally recognize is the Golden Rule. We hope we will be able to build our particular solutions on that basis. But we cannot accept the solutions you are attempting to impose.

In the United States, you have today created a law, the Telecommunications

Reform Act, which repudiates your own Constitution and insults the dreams of Jefferson, Washington, Mill, Madison, DeToqueville, and Brandeis. These dreams must now be born anew in us.

You are terrified of your own children, since they are natives in a world where you will always be immigrants. Because you fear them, you entrust your bureaucracies with the parental responsibilities you are too cowardly to confront yourselves. In our world, all the sentiments and expressions of humanity, from the debasing to the angelic, are parts of a seamless whole, the global conversation of bits. We cannot separate the air that chokes from the air upon which wings beat.

In China, Germany, France, Russia, Singapore, Italy and the United States, you are trying to ward off the virus of liberty by erecting guard posts at the frontiers of Cyberspace. These may keep out the contagion for a small time, but they will not work in a world that will soon be blanketed in bit-bearing media.

Your increasingly obsolete information industries would perpetuate themselves by proposing laws, in America and elsewhere, that claim to own speech itself throughout the world. These laws would declare ideas to be another industrial product, no more noble than pig iron. In our world, whatever the human mind may create can be reproduced and distributed infinitely at no cost. The global conveyance of thought no longer requires your factories to accomplish.

These increasingly hostile and colonial measures place us in the same position as those previous lovers of freedom and self-determination who had to reject the authorities of distant, uninformed powers. We must declare our virtual selves immune to your sovereignty, even as we continue to consent to your rule over our bodies. We will spread ourselves across the Planet so that no one can arrest our thoughts.

We will create a civilization of the Mind in Cyberspace. May it be more humane and fair than the world your governments have made before.

Davos, Switzerland
February 8, 1996

Message 2:
Decryption failed (MAC invalid or corrupted message).

Message 3:
Decryption successful:
On two occasions I have been asked, - "Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?" In one case a member of the Upper, and in the other a member of the Lower, House put this question. I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.

Task4

Brief introduction

First, create your own public/private key pair.
Then, encrypt the message using the shared key and a nonce.
Finally, print the hexadecimal dump of the encrypted message and simulate the recipient to verify whether the message can be successfully decrypted using the previous decryption method.

Code:

Lab1 > Task4 > C task4.c > ...

```c
#include <stdio.h>
#include <string.h>
#include "monocypher.h"
#include "keys.h"  // rx_sk
#include "devurandom_windows.h"
void hex_dump(const uint8_t *data, size_t size) {
    for (size_t i = 0; i < size; i++) {
        printf("0x%02x, ", data[i]);
        if ((i + 1) % 16 == 0) printf("\n");
    }
    printf("\n");
}
int main() {
    // Step 1: Generate private + public key
    uint8_t my_sk[32];
    uint8_t my_pk[32];
    randombytes(my_sk, 32);
    crypto_key_exchange_public_key(my_pk, my_sk);
    // Step 2: Generate shared key
    uint8_t shared_key[32];
    crypto_key_exchange(shared_key, my_sk, rx_sk); // receiver's public key from keys.h
    // Step 3: Generate nonce
    uint8_t nonce[24];
    randombytes(nonce, 24);
    // Step 4: Encrypt message
    const char* msg = "Don't rush, feel the way.";
    size_t msg_len = strlen(msg);
    uint8_t ciphertext[msg_len];
    uint8_t mac[16];
    crypto_lock(mac, ciphertext, shared_key, nonce, (const uint8_t*)msg, msg_len);
    // Step 5: Print results
    printf("My original message:\n");
    printf("%s\n", msg);
    printf("\nEncrypted message:\n");
    hex_dump(ciphertext, msg_len);
    printf("\nMAC:\n");
    hex_dump(mac, 16);
    printf("Nonce:\n");
    hex_dump(nonce, 24);
    return 0;
}
```

行 6，列 32

Result:

Input message: Summer will come around again. People who meet will meet again.

```
PS D:\CWORKSPACE\Lab1\Task4> gcc -o task4 task4.c monocypher.c devurandom_windows.c
PS D:\CWORKSPACE\Lab1\Task4> ./task4
My original message:
Summer will come around again. People who meet will meet again.

Encrypted message:
0x17, 0x1d, 0xab, 0xf3, 0x2b, 0x83, 0xec, 0x1e, 0x59, 0x21, 0x62, 0x5c, 0x3e, 0x37, 0x22, 0xe9,
0xcc, 0xe8, 0x71, 0x17, 0xec, 0xdd, 0xa5, 0x2e, 0x6c, 0x82, 0x59, 0x62, 0xb6, 0xa0, 0x02, 0x37,
0xc9, 0xa7, 0x01, 0x10, 0x05, 0x6a, 0xac, 0x27, 0xab, 0x92, 0x12, 0x20, 0xdd, 0x4c, 0x36, 0x8d,
0xfc, 0xdc, 0x9d, 0x6c, 0xf7, 0x8f, 0x3d, 0x54, 0xfc, 0xb6, 0xad, 0x14, 0x52, 0x7d, 0xfd,

MAC:
0xa4, 0x38, 0x18, 0xa2, 0x21, 0x26, 0xac, 0x56, 0x19, 0x47, 0x89, 0xee, 0x6f, 0xf4, 0xf5, 0x65,

Nonce:
0x7d, 0x11, 0xc5, 0x54, 0xf9, 0xca, 0x67, 0x46, 0x22, 0xb0, 0x03, 0x1a, 0x73, 0x81, 0x41, 0x84,
0x7d, 0x44, 0x1c, 0x60, 0x10, 0xf0, 0x80, 0xee,
PS D:\CWORKSPACE\Lab1\Task4>
```

Input message: Don't rush, feel the way.

```
PS D:\CWORKSPACE\Lab1\Task4> gcc -o task4 task4.c monocypher.c devurandom_windows.c
PS D:\CWORKSPACE\Lab1\Task4> ./task4
My original message:
Don't rush, feel the way.

Encrypted message:
0x0a, 0x89, 0x3c, 0xfb, 0xd4, 0x41, 0xec, 0x19, 0x84, 0x63, 0xa6, 0xfe, 0xa6, 0x86, 0xfc, 0xfd,
0x1e, 0x6f, 0x93, 0x0f, 0xd8, 0xaa, 0x9f, 0x0f, 0x8e,

MAC:
0x3e, 0x2c, 0xef, 0x17, 0xaf, 0xc5, 0x0a, 0xbb, 0x8d, 0x42, 0x6a, 0x86, 0x5f, 0x3a, 0x6a, 0xf2,

Nonce:
0x5f, 0x3a, 0x13, 0xd5, 0x05, 0x61, 0xd7, 0x24, 0xe1, 0x71, 0xc7, 0x3b, 0x78, 0xe7, 0x54, 0x18,
0x9e, 0x88, 0xfa, 0xee, 0xbd, 0x23, 0xef, 0x87,
```

Lab2

Task1

Brief introduction

- Read and understand the source code of simple.c.
- Find out the fixed-size buffer in it (such as char buffer[8]).
- Try to overwrite the key variable (such as allow) by inputting an extremely long string, making the program mistakenly think that the verification is passed.
- Disable the stack protector when compiling with gcc.

Experimental Objectives

- Understand the basic principle of Buffer Overflow.
- Master how to change the value of a variable through overflow.
- Observe how variables are overwritten in memory.
- Cultivate security awareness: seemingly harmless functions like `gets()` and code that doesn't check the input length can lead to serious vulnerabilities.

Code

Lab2 > Task1 > C simple.c > ...

```c
#include <stdio.h>

int main() {
    char username[10];
    int allow = 0;
    char password[10];

    printf("Enter username.\n");
    gets(username);

    while (!allow) {
        printf("Enter password.\n");
        gets(password);
        // allow = (hash(password)==xxx);

        printf("allow: %d", allow);
    }

    printf("Welcome.\n");

}
```

Result

As can be seen from the following figure, when the length of the password I input exceeds the password length specified by the program, the program will mistakenly think that the verification is passed, and thus output "Welcome."

```
PS D:\CWORKSPACE\Lab2\Task1> gcc simple.c -g -fno-stack-protector -o simple
simple.c: In function 'main':
simple.c:9:5: warning: call to 'gets' declared with attribute warning: Using gets() is always unsafe - use fgets() instead [-Wattribute-warning]
    9 |     gets(username);
      |     ^~~~~~~~~~~~~~
simple.c:13:9: warning: call to 'gets' declared with attribute warning: Using gets() is always unsafe - use fgets() instead [-Wattribute-warning]
   13 |         gets(password);
      |         ^~~~~~~~~~~~~~
PS D:\CWORKSPACE\Lab2\Task1> ./simple
Enter username.
QW
Enter password.
AAAAAAAAAAAAAAAAAAAAAAAA
Welcome.
PS D:\CWORKSPACE\Lab2\Task1>
```

Use GDB to observe how the values of variables are changed.

```
PS D:\CWORKSPACE\Lab2\Task1> gdb ./simple
GNU gdb (GDB for MinGW-W64 x86_64, built by Brecht Sanders, r3) 16.2
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-w64-mingw32".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
--Type <RET> for more, q to quit, c to continue without paging--

Reading symbols from ./simple...
(gdb) tui enable
```

The stack grows from higher addresses to lower addresses. So in the stack, `password[]` is “below” `allow`. When `gets(password)` receives content exceeding 10 bytes, it will continue to write upwards and eventually overwrite the `allow` variable.

As can be seen from the output, `allow` is non - zero, indicating that the overwrite was successful.

```
(gdb) start
start
Temporary breakpoint 1 at 0x1400016dd: file simple.c, line 5.
Starting program: D:\CWORKSPACE\Lab2\Task1\simple.exe
[New Thread 209092.0x3dce8]

Thread 1 hit Temporary breakpoint 1, main () at simple.c:5
warning: Source file is more recent than executable.
5           int allow = 0;
(gdb)

(gdb) next
next
8           printf("Enter username.\n");
(gdb)

Enter username.
9           gets(username);
(gdb) next
next

11          while (!allow) {
(gdb) next
next
12              printf("Enter password.\n");
(gdb) next
next
Enter password.
13              gets(password);
(gdb) next
next
AAAAAAAAAAAAAAAAAAAAAAAAAAAA
16              // printf("allow: %d", allow);
(gdb) print allow
print allow
$1 = 1094795585
(gdb)

$2 = 1094795585
(gdb)
```

Task2

Brief introduction

- Read and understand the source code of stack.c;
- Try to input an extremely long string to make the program crash (causing a Segmentation fault);
- Comprehend that the crash is due to the corruption of the return address;
- Compare the effects by using two different compilation methods:

Experimental Objectives

- Understand the stack frame structure (return address, local variables) during function calls;
- Grasp the impact of buffer overflow on the return address;
- Learn to use the stack protection mechanism (Stack Canary) to detect stack corruption;
- Recognize that the overflow after function separation is more harmful and requires more complex exploitation methods.

Code

Lab2 > Task2 > C stack.c > ...

```c
#include <stdio.h>

int check_password() {
    char password[10];
    gets(password);
    return 0;
}

int main() {
    char username[10];
    int allow = 0;

    printf("Enter username.\n");
    fgets(username, 10, stdin);

    while (!allow) {
        printf("Enter password.\n");
        allow = check_password();
    }

    printf("Welcome.\n");

}

```

Result:

Disable the stack protector (unsafe, easy for the overflow to succeed):

- gcc stack.c -g -fno-stack-protector -o stack
- ./stack

```
PS D:\CWORKSPACE\Lab2\Task2> gcc stack.c -g -fno-stack-protector -o stack
stack.c: In function 'check_password':
stack.c:5:5: warning: call to 'gets' declared with attribute warning: Using gets() is always unsafe - use fgets() instead [-Wattribute-warning]
    5 |     gets(password);
      |     ^~~~~~~~~~~~
PS D:\CWORKSPACE\Lab2\Task2> ./stack
Enter username.
AA
Enter password.
1234567890
Enter password.
12345678910
PS D:\CWORKSPACE\Lab2\Task2>
```

Enable the stack protector (relatively safe, automatically detect overflow):

- gcc stack.c -g -fstack-protector -o stack
- ./stack

```
PS D:\CWORKSPACE\Lab2\Task2> gcc stack.c -g -fstack-protector -o stack
stack.c: In function 'check_password':
stack.c:5:5: warning: call to 'gets' declared with attribute warning: Using gets() is always unsafe - use fgets() instead [-Wattribute-warnin
g]
    5 |     gets(password);
      |     ^~~~~~~~~~~~~~
PS D:\CWORKSPACE\Lab2\Task2> ./stack
Enter username.
ASD
Enter password.
AAAAAAAAAAAAAAAAAAAAAA
*** stack smashing detected ***:  terminated
PS D:\CWORKSPACE\Lab2\Task2>
```

Task3

Brief introduction

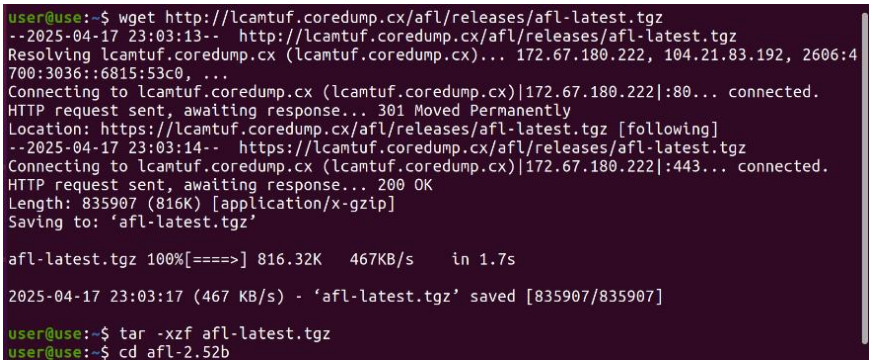
- Download and compile fuzzgoat (a defective JSON parser);
- Perform fuzzing on it using the AFL tool;
- Observe how AFL automatically generates inputs and discovers crashes;
- Check the "poisonous" inputs generated by AFL in out/crashes/;
- Answer: How many inputs are difficult for humans to come up with during manual testing?

Experimental Objectives

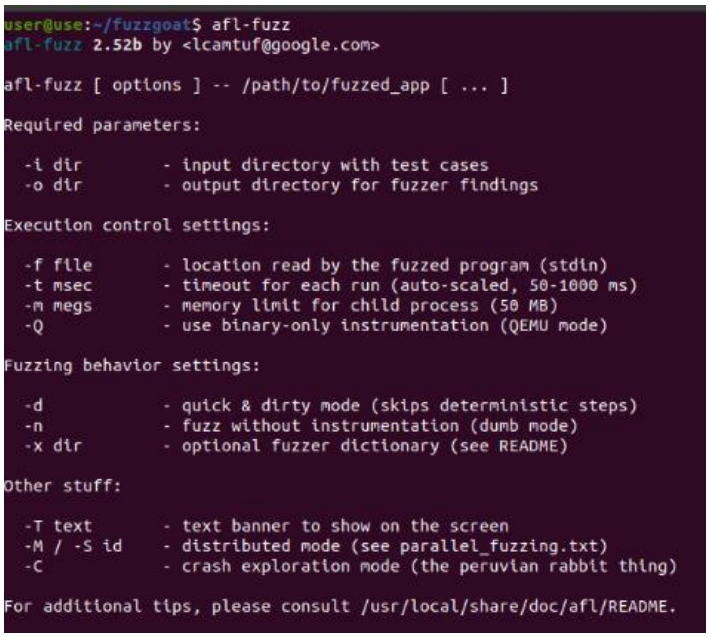
- Learn to use the modern fuzz testing tool AFL;
- Understand the automated security testing process based on mutation and path feedback;
- Compare the depth of coverage between human testing and tool-based testing;
- Enhance the ability and awareness of automatically discovering unknown vulnerabilities.

Experimental Procedure

Step 1: Download and compile AFL.



Step 2: Verify that AFL is installed successfully.



Step 3: Instrument and compile fuzzgoat using AFL.

```
user@use:~$ cd afl-2.52b
user@use:~/afl-2.52b$ make
[*] Checking for the ability to compile x86 code...
[+] Everything seems to be working, ready to compile.
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH=\"/usr/local
/lib/afl\" -DDOC_PATH=\"/usr/local/share/doc/afl\" -DBIN_PATH=\"/usr/local/bin\" afl-gcc.c -
o afl-gcc -ldl
set -e; for i in afl-g++ afl-clang afl-clang++; do ln -sf afl-gcc $i; done
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH=\"/usr/local
/lib/afl\" -DDOC_PATH=\"/usr/local/share/doc/afl\" -DBIN_PATH=\"/usr/local/bin\" afl-fuzz.c
-o afl-fuzz -ldl
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH=\"/usr/local
/lib/afl\" -DDOC_PATH=\"/usr/local/share/doc/afl\" -DBIN_PATH=\"/usr/local/bin\" afl-showmap
.c -o afl-showmap -ldl
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH=\"/usr/local
/lib/afl\" -DDOC_PATH=\"/usr/local/share/doc/afl\" -DBIN_PATH=\"/usr/local/bin\" afl-tmin.c
-o afl-tmin -ldl
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH=\"/usr/local
/lib/afl\" -DDOC_PATH=\"/usr/local/share/doc/afl\" -DBIN_PATH=\"/usr/local/bin\" afl-gotcpu.
c -o afl-gotcpu -ldl
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH=\"/usr/local
/lib/afl\" -DDOC_PATH=\"/usr/local/share/doc/afl\" -DBIN_PATH=\"/usr/local/bin\" afl-analyze
.c -o afl-analyze -ldl
cc -O3 -funroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH=\"/usr/local
/lib/afl\" -DDOC_PATH=\"/usr/local/share/doc/afl\" -DBIN_PATH=\"/usr/local/bin\" afl-as.c -o
 afl-as -ldl
ln -sf afl-as as
[*] Testing the CC wrapper and instrumentation output...
unset AFL_USE_ASAN AFL_USE_MSAN; AFL_QUIET=1 AFL_INST_RATIO=100 AFL_PATH=. ./afl-gcc -O3 -fu
nroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH=\"/usr/local/lib/afl\"
 -DDOC_PATH=\"/usr/local/share/doc/afl\" -DBIN_PATH=\"/usr/local/bin\" test-instr.c -o test-
instr -ldl
echo 0 | ./afl-showmap -m none -q -o .test-instr0 ./test-instr
echo 1 | ./afl-showmap -m none -q -o .test-instr1 ./test-instr
[+] All right, the instrumentation seems to be working!
[+] All done! Be sure to review README - it's pretty short and useful.
NOTE: If you can read this, your terminal probably uses white background.
This will make the UI hard to read. See docs/status_screen.txt for advice.
user@use:~/afl-2.52b$
```

```
user@use:~/afl-2.52b$ sudo make install
[*] Checking for the ability to compile x86 code...
[+] Everything seems to be working, ready to compile.
[*] Testing the CC wrapper and instrumentation output...
unset AFL_USE_ASAN AFL_USE_MSAN; AFL_QUIET=1 AFL_INST_RATIO=100 AFL_PATH=. ./afl-gcc -O3 -fu
nroll-loops -Wall -D_FORTIFY_SOURCE=2 -g -Wno-pointer-sign -DAFL_PATH=\"/usr/local/lib/afl\"
 -DDOC_PATH=\"/usr/local/share/doc/afl\" -DBIN_PATH=\"/usr/local/bin\" test-instr.c -o test-
instr -ldl
echo 0 | ./afl-showmap -m none -q -o .test-instr0 ./test-instr
echo 1 | ./afl-showmap -m none -q -o .test-instr1 ./test-instr
[+] All right, the instrumentation seems to be working!
[+] All done! Be sure to review README - it's pretty short and useful.
NOTE: If you can read this, your terminal probably uses white background.
This will make the UI hard to read. See docs/status_screen.txt for advice.
mkdir -p -m 755 ${DESTDIR}/usr/local/bin ${DESTDIR}/usr/local/lib/afl ${DESTDIR}/usr/local/s
hare/doc/afl ${DESTDIR}/usr/local/share/afl
rm -f ${DESTDIR}/usr/local/bin/afl-plot.sh
install -m 755 afl-gcc afl-fuzz afl-showmap afl-tmin afl-gotcpu afl-analyze afl-plot afl-cmi
n afl-whatsup ${DESTDIR}/usr/local/bin
rm -f ${DESTDIR}/usr/local/bin/afl-as
if [ -f afl-qemu-trace ]; then install -m 755 afl-qemu-trace ${DESTDIR}/usr/local/bin; fi
if [ -f afl-clang-fast -a -f afl-llvm-pass.so -a -f afl-llvm-rt.o ]; then set -e; install -m
 755 afl-clang-fast ${DESTDIR}/usr/local/bin; ln -sf afl-clang-fast ${DESTDIR}/usr/local/bin
/afl-clang-fast++; install -m 755 afl-llvm-pass.so afl-llvm-rt.o ${DESTDIR}/usr/local/lib/af
l; fi
if [ -f afl-llvm-rt-32.o ]; then set -e; install -m 755 afl-llvm-rt-32.o ${DESTDIR}/usr/loca
l/lib/afl; fi
if [ -f afl-llvm-rt-64.o ]; then set -e; install -m 755 afl-llvm-rt-64.o ${DESTDIR}/usr/loca
l/lib/afl; fi
set -e; for i in afl-g++ afl-clang afl-clang++; do ln -sf afl-gcc ${DESTDIR}/usr/local/bin/$
i; done
install -m 755 afl-as ${DESTDIR}/usr/local/lib/afl
ln -sf afl-as ${DESTDIR}/usr/local/lib/afl/as
install -m 644 docs/README docs/ChangeLog docs/*.txt ${DESTDIR}/usr/local/share/doc/afl
cp -r testcases/ ${DESTDIR}/usr/local/share/afl
cp -r dictionaries/ ${DESTDIR}/usr/local/share/afl
user@use:~/afl-2.52b$
```

Step 4: Create an input directory and test cases.

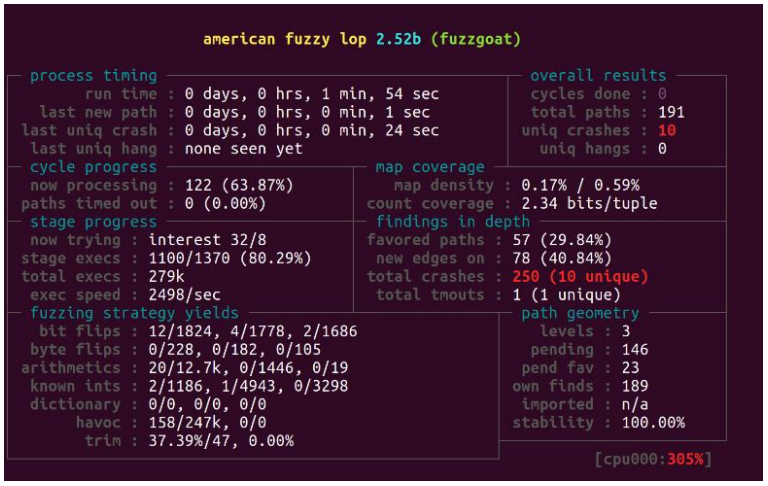
Set "CC=afl-gcc" to tell make to use the compiler provided by AFL to instrument the program.

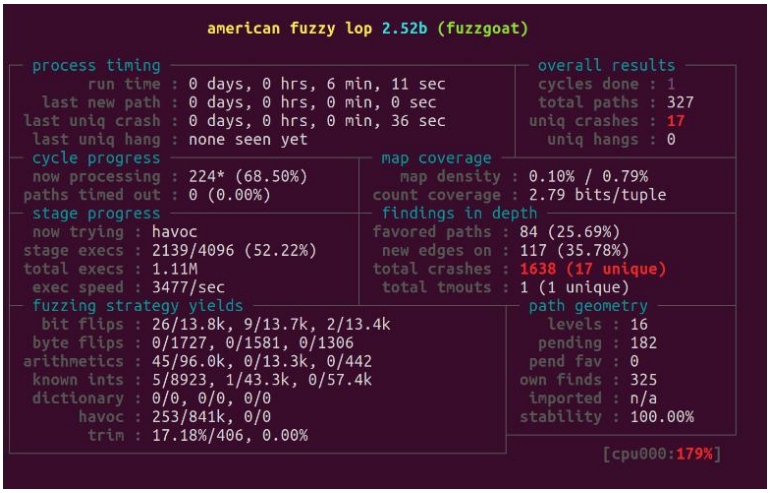
```
user@use:~/fuzzgoat$ CC=afl-gcc make
afl-gcc -o fuzzgoat -I. main.c fuzzgoat.c -lm
afl-cc 2.52b by <lcamtuf@google.com>
afl-as 2.52b by <lcamtuf@google.com>
[+] Instrumented 75 locations (64-bit, non-hardened mode, ratio 100%).
afl-as 2.52b by <lcamtuf@google.com>
[+] Instrumented 375 locations (64-bit, non-hardened mode, ratio 100%).
afl-gcc -fsanitize=address -o fuzzgoat_ASAN -I. main.c fuzzgoat.c -lm
afl-cc 2.52b by <lcamtuf@google.com>
afl-as 2.52b by <lcamtuf@google.com>
[+] Instrumented 51 locations (64-bit, ASAN/MSAN mode, ratio 33%).
afl-as 2.52b by <lcamtuf@google.com>
[+] Instrumented 335 locations (64-bit, ASAN/MSAN mode, ratio 33%).
user@use:~/fuzzgoat$
```

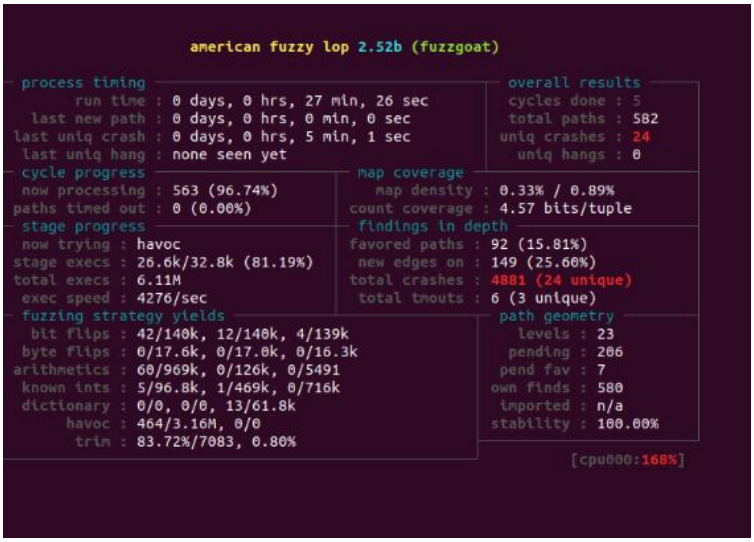
Step 5: Run AFL for fuzz testing.

```
user@use:~/fuzzgoat$ afl-fuzz -i in -o out ./fuzzgoat @@
afl-fuzz 2.52b by <lcamtuf@google.com>
[+] You have 2 CPU cores and 1 runnable tasks (utilization: 50%).
[+] Try parallel jobs - see /usr/local/share/doc/afl/parallel_fuzzing.txt.
[*] Checking CPU core loadout...
[+] Found a free CPU core, binding to #0.
[*] Checking core_pattern...
[*] Setting up output directories...
[+] Output directory exists but deemed OK to reuse.
[*] Deleting old session data...
[+] Output dir cleanup successful.
[*] Scanning 'in'...
[+] No auto-generated dictionary tokens to reuse.
[*] Creating hard links for all input files...
[*] Validating target binary...
[*] Attempting dry run with 'id:000000,orig:sample.json'...
[*] Spinning up the fork server...
[+] All right - fork server is up.
    len = 3, map size = 79, exec speed = 314 us
[*] Attempting dry run with 'id:000001,orig:seed'...
    len = 8, map size = 79, exec speed = 296 us
[+] All test cases processed.

[+] Here are some useful stats:

    Test case count : 2 favored, 0 variable, 2 total
       Bitmap range : 79 to 79 bits (average: 79.00 bits)
        Exec timing : 296 to 314 us (average: 305 us)

[*] No -t option specified, so I'll use exec timeout of 20 ms.
[+] All set and ready to roll!


                    american fuzzy lop 2.52b (fuzzgoat)

┌─ process timing ─────────────────────┬─ overall results ─────┐
        run time : 0 days, 0 hrs, 0 min, 24 sec  cycles done : 0
   last new path : 0 days, 0 hrs, 0 min, 0 sec   total paths : 128
```

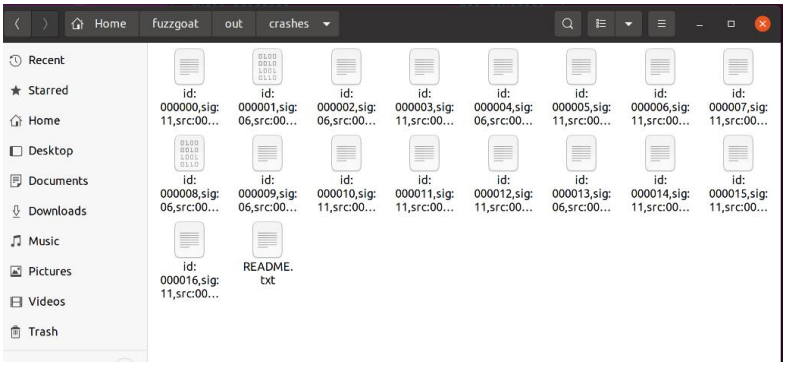
Step 6: Wait for AFL to run and monitor the status: Check if new paths or crashes are found (e.g., whether total crashes is non-zero).



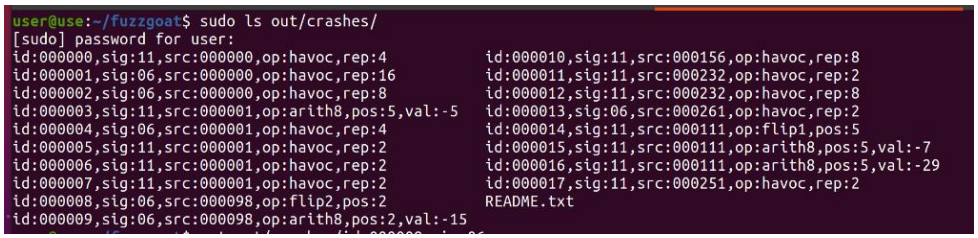




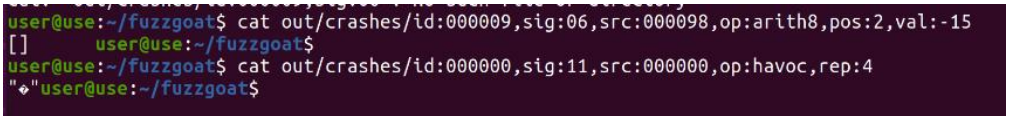
Step 7: Check the contents of the output directory. Go to out/crashes/ to view the test results.



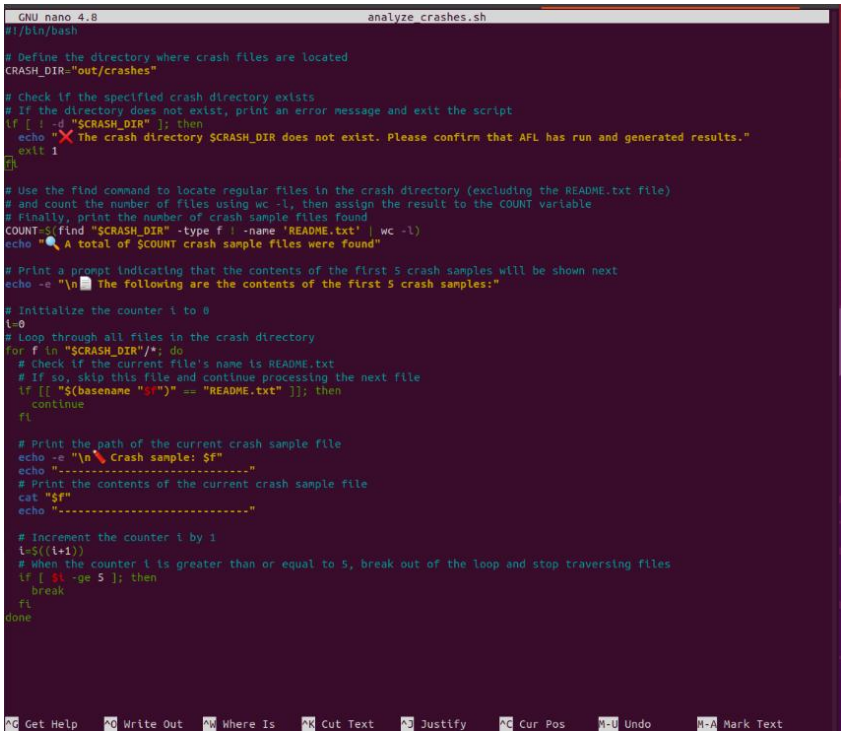
Step 8: Examine the crash test cases.



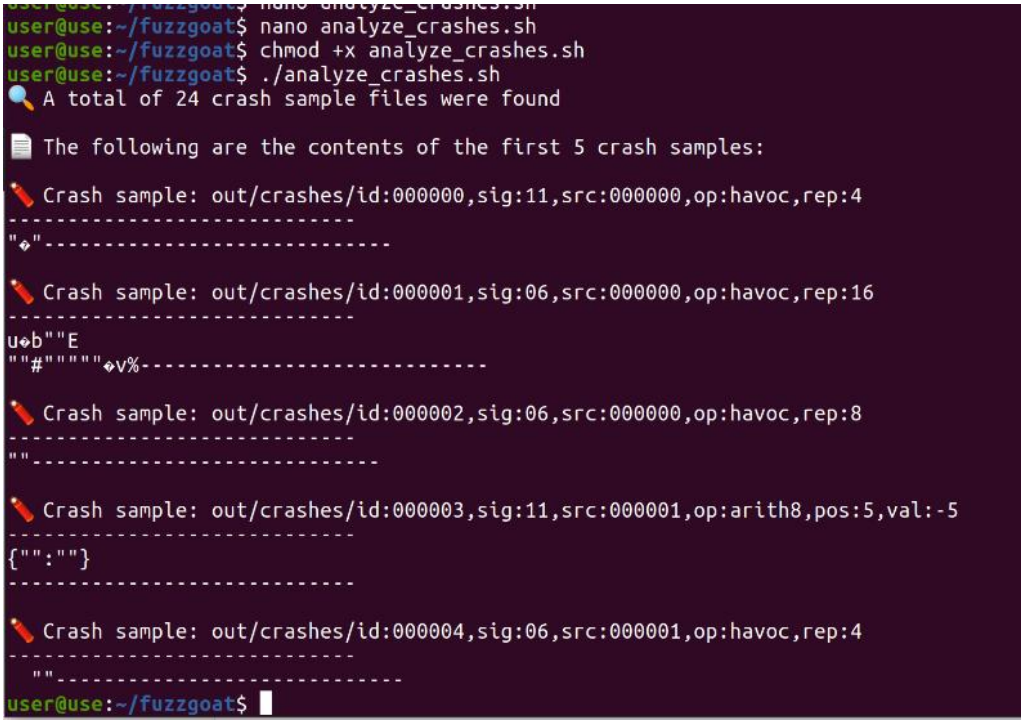
Step 9: Read the contents of the crash samples.



Step 10: Determine whether these inputs are ones that humans might write.
The script code is as follows.



The running result of the script:



Result

Question: How many of them do you think a human trying to write a test suite might plausibly have generated?

From the following table, you can observe whether certain input types are likely to be written by humans. For example, it is very difficult for humans to manually write Extreme Numerical Values, Garbled Characters, Special Characters, as well as Random Bit Flips or Unconventional Encoding.

Input Type	Can Humans Write It?
Normal JSON Structure	Possible to Write
Slightly Malformed (Missing Braces, Out of Order)	Somewhat Possible
Extreme Numerical Values, Garbled Characters, Special Characters	Difficult to Write Manually
Random Bit Flips or Unconventional Encoding	Almost Impossible

After I ran the AFL fuzz testing, I found approximately 24 crash samples in total. After checking them one by one, I discovered that most of them contained non-UTF-8 characters, extremely long strings, or disorganized JSON data. Most of these inputs were randomly generated by AFL's mutation algorithm. If I were to write test cases myself, I might only come up with 2 to 3 normal or slightly malformed inputs, and it would be almost impossible for humans to write the remaining dozen or so crash samples. Therefore, I think that test cases written by humans can cover at most about 10-20% of the AFL crash samples.

So the strength of AFL lies in its ability to automatically discover those inputs that humans cannot think of.