`TV Hack` is a challenge in the `reverse` category of the FCSC 2024.
The challenge is splits in two parts : a medium (★★) and a hard (★★★).

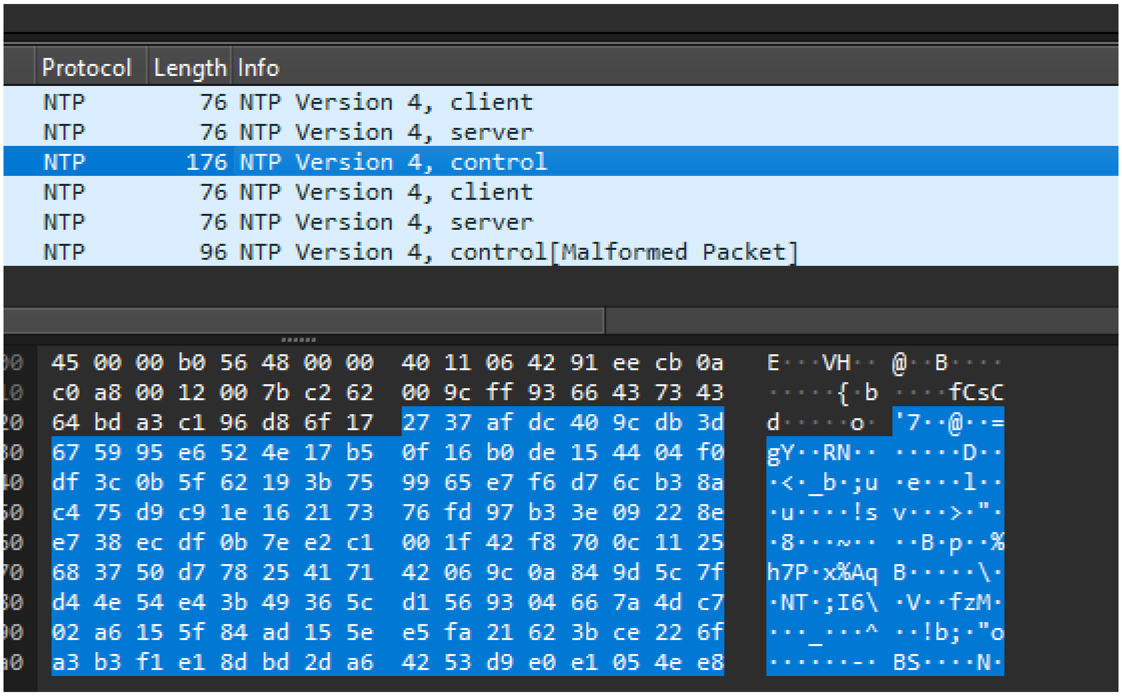
Part 1/2

Gathering information

A device who emits TV video flux received some suspicious NTP packets, a malicious kernel module was installed on it.

As input, we got a network capture (.pcap) and the module kernel binary.

The pcap contains 6 packets, 2 pairs of standard NTP Packet and 2 suspicious control packets. Both control packet, contains the string `fCsC` 🤔.



The kernel module is an ELF x86-64, I used binary ninja to reverse it.

Running `strings` on the binary yields interesting results :

```
$> strings ipopt.ko
...
vermagic=3.2.0-4-amd64 SMP mod_unload modversions
description=IP optimizer
license=GPL
...
crypto_alloc_base
skb_make_writable
nf_register_hooks
getnstimeofday
call_usermodehelper_exec
...
```

This module is compiled for the linux kernel 3.2 (released 13 years ago).

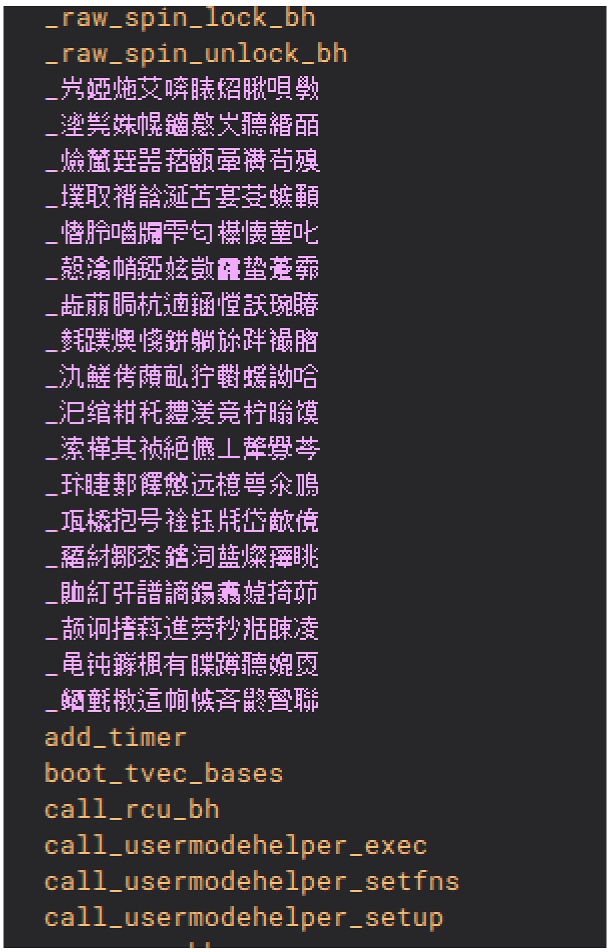
For searching and exploring the linux kernel codebase, I used this website : https://elixir.bootlin.com/linux/v3.2.102

Interesting imports :

- `crypto_alloc_base` : allocate a new cryptographic object

- `skb_make_writable` : `skb` stands for `socket buffer`, this module modifies packet content ?
- `nf_register_hooks` : add hooks in `net_filter`, seems legit for an `IP optimizer`
- `getnstimeofday` : keep this in mind, we will comeback to it later
- `call_usermodehelper_exec` : this function is used to execute process in userland from the kernel, seems a lot less legit for an `IP optimizer` 😉

The binary contains a lot of Chinese-like utf-8 character, unfortunately they don't seems to mean anything. But I learned through the challenge, that these symbols mark the added stuff from the original module.



The reversing part was hard and tedious, the kernel module uses a lot of kernel internals APIs (crypto, network buffers, ...).
I was not able to import kernels header in binary ninja, nor Ghidra.

This function was interesting, it looks up symbols from strings encrypted with a simple algorithm.

```
000022f7  uint64_t decrypt_symbols()

000022f7      int64_t r11
000022f7      int64_t var_8 = r11
00002314      create_write_pipe = kallsyms_lookup_name(name: decrypt(&enc_str_create_write_pipe))
0000232f      free_write_pipe = kallsyms_lookup_name(name: decrypt(&enc_str_free_write_pipe))
0000234a      create_read_pipe = kallsyms_lookup_name(name: decrypt(&enc_str_$create_read_pipe))
00002365      expand_files = kallsyms_lookup_name(name: decrypt(&enc_str_expand_files))
00002383      int64_t rax_9 = __alloc_workqueue_key(decrypt(&enc_str_dm-0), 2, 1, 0, 0)
0000238c      wk = rax_9
00002399      return zx.q(sbb.d(rax_9.d, rax_9.d, rax_9 u< 1) & 0xfffffff4)  {"+>"}  {">"}
```

By searching other calls to this function, i found these other encrypted strings :

- `hmac(sha256)`
- `cbc(aes)`
- `ctr(aes)`
- `HOME=/`
- `TERM=linux`
- `PATH=/sbin:/usr/sbin:/bin:/usr/bin`

After several hours, I gave up the static analysis without proper type information.

The dynamic analysis rabbit hole

We know the kernel version, so I tried compiling and running the kernel with the module inside QEMU.

Since the kernel code is 13 years old, it don't compiles with recent gcc versions, but by compiling in an Ubuntu VM old enough (12.04), success !

The kernel boots with no problems, but installing the module was failing with weird errors.

In a kernel module, there are some static offsets which must match with the kernel module loader, and these offsets are affected by a LOT of parameters (kernel compile flags, distribution, patches, ...).

So this was a dead end ...
... except I have now a `vmlinux` full of DWARF symbols !

Static analysis with types

These DWARF symbols contains all struct layouts and symbols of the entire kernel.
I managed to import all dwarf symbols in binary ninja, and start annotating all extern functions.

It was a lot more easy to get an overview of the module.

Since the challenge is related to network, I starts with `netfilter` hooks.

```
# Pseudo code : Hook -> RCE procedure

# hook entry point
def nf_receive_packet_hook():
        assert udp_type()
        assert contains_magic_data("fCsC")

        ntp_data = unpack_udp(unpack_ip())
        handle_packet(ntp_data)

# handle packet with magic data "fCsC"
def handle_packet(ntp_data):
        payload = ntp_data[:-0x10]
        sig = ntp_data[-0x10:] # last 16 bytes

        # check payload signature
        if check_signature(payload, sig):
                # decrypt the payload body
                ivs = body[0x4:0x14]
                enc_payload = body[0x14:]
                clear_payload = decrypt_payload(ivs, enc_payload)

                if clear_payload[0] == 0: # check the command flag
                        # update `prk`
                        update_prk_from_payload(
                                new_prk_source = clear_payload[0x01:0x11]
                        )
                else:
                        # spawn a userland thread and execute the command
                        cmd = clear_body
                    exec_in_userland(cmd)
```

The hook listen for NTP/UDP packet with the magic string `fCsC`, check the signature, decrypt the command and execute it in userland.

Now I was sure, that the flag was in one of two encrypted packet from the pcap.

The challenging part is the cryptography.
After each valid packet, the key `prk` is updated from the payload.

```
# every packet update this key
prk = [ ... ] # 32 bytes
```

```python
# derive a key from prk
def hkdf(info, needed_length):
        return hdkf(
                key=prk,
                info,
                needed_length
        )

def check_signature(body, sig):
        sign_key = [ ... ] # constant 8 bytes
        key = hkdf(info=sign_key, needed_length=0x20)

        computed_sig = hmac_sha256(key, body)
        return sig == computed_sig

def decrypt_payload(ivs, data):
        cbc_key = [ ... ] # constant 8 bytes
        key = hkdf(info=cbc_key, needed_length=0x10)

        return aes_cbc(key, ivs, data)
```

This seems impossible, if we don't know `prk`, we can't decrypt packet, so we can't found the next value of `prk`.

But the update of `prk` contains a weird piece of code.

```python
# new_prk_source : 16 bytes
def update_prk_from_payload(new_prk_source):
        # ==== WIERD PART START ====

        masks = [ ... ] # constant 0x30 bytes

        # mask 3 times each byte ???
        for i in range(0,0x30):
                new_prk_source[i % 0x10] &= mask[i]

        # ==== WIERD PART END ====

        salt = [ ... ] # constant 32 bytes
        prk = hmac_sha256(key=salt, input=new_prk_source)
```

The source of derivation is masked 3 times by a constant array of bytes.

When applying all masks, the key possible size shrinks from 128 bits to 21 bits.

```
0 : forced to 0
1 : 0 or 1

all masks combined :
 00100010 01000010 00000010 00000110 00000100 00001000 00000000 01001000
 10000011 00000100 00000000 00001000 00110000 00010000 00010000 00000100

only 21 bits possible -> 2,097,152 combinations
```

So to wrap up, we can brute force all possible values of the `prk` key, against the signature and decrypt the 2 packets !

First packet :

```
/bin/sh -c echo
"FCSC{5d58e776e659866d110ac50dc2bce631e634222953a234893cb4978594ec0ae1}" >
/root/flag1
```

Second packet :

```
/bin/cat /root/flag2
```

Nice, then in the part 2, we need to find the exfiltrated data !

Part 2/2

Challenge | 2 résolutions

TV Hacks 2/2

498 points

★★★

> **Important :** *Les deux étapes de la série `TV Hacks` sont à réaliser dans l'ordre. Nous vous conseillons fortement de ne pas tenter la résolution de cette épreuve tant que vous n'avez pas validé la première étape.*

Merci pour la première analyse effectuée dans `TV Hacks 1/2` !

Au vu de la criticité de l'équipement, les règles du pare-feu sont très strictes. A part le protocole NTP, aucun paquet n'est envoyé sur Internet directement. Cependant, le flux vidéo diffusé par l'équipement est transmis à notre prestataire en charge de la diffusion et, les débits étant trop importants, nous ne pouvons pas garder une trace de ces données.

Heureusement, un ami à vous, fan de TV Hacks, enregistrait l'émission qui passait au moment de l'attaque. Il a pu vous envoyer un extrait de vidéo : cela devrait vous permettre d'aller plus avant.

De plus, le fournisseur de la solution de streaming a pu nous transmettre le code source du module noyau développé par un prestataire. Cela pourrait faciliter votre analyse même si **nous avons exclu une attaque par la chaîne logistique.**

Remerciements :

- Sprite Fright : (CC) Blender Foundation | studio.blender.org
- Sprite Fright version française : (CC-BY) Touhoppai | www.youtube.com/@Touhoppai

capture.pcap | ipopt.ko | ipopt.tar.gz | capture.ts

Flag | Soumettre

Now we have the source code of the original module and a video stream of the TV Flux. We can suppose that the flag is exfiltrated within the flux.

- ori_code
  - avc.c
  - bs.h
  - common.h
  - ipopt.tar.gz
  - main.c
  - Makefile
  - ts.c

- `main.c` : contains the setup code (register the hooks, ...)
- `ts.c` : contains a MPEG-TS decoder
- `avc.c` : contains a NAL / AVC decoder

The original code is hard to fully understand, there are 0 comments, but it seems to prioritize some UDP packets over others.

MPEG-TS is a protocol to transport multiple data streams in a single `stream` (multiplexing).

The code of decoders is a bit complex because they are structured to handle fragmented packets in streaming and they 'spawn' sub-decoders (MPEG -> NAL -> AVC).

I started by matching functions in the original source code in the decompiler.

In the `ts_pmt_parse` function, this snippet is different than the original code.

```
struct pid_ops* rdx_15 = &ts_pes_private_ops
if (pid s>= 0)
    rdx_15 = &ts_pes_avc_ops
```

This code spawns a sub decoder based on the PID (stream identifier in MPEG-TS).
The original code did not contains the `pes_private` decoder, it was added !

Extract the private MPEG-TS Stream

I used this rust crate : https://crates.io/crates/mpeg2ts-reader

First I dumped, all streams in the `capture.ts`.

```
1 stream  : H264
2 streams : AtscDolbyDigitalAudio
2 streams : H2220PesPrivateData      /!\ interesting
```

And with the same crate, I dumped the raw stream data.

```
$> xxd private_data | head
00000000: 2000 0f14 0001 0005 1707 8004 380f 1000   ...........8...
00000010: 0100 021e eb0f 8000 0100 00ff 2000 0f14  ............ ...
00000020: 0001 0005 1707 8004 380f 1000 0100 081e  ........8.......
00000030: fb00 ff02 8803 ff0f 1100 0100 1000 f702  ................
00000040: 7000 1e4b 0000 0300 0000 00f0 000f 1200  p..K............
00000050: 0100 6200 ff00 5f00 8080 fd01 5f00 8082  ..b..._....._...
00000060: 7802 5f04 8181 1c03 5f00 8080 c804 5f00  x._....._....._.
```

```
00000070: 8081 9705 5f04 8482 4606 5f29 9f70 1007  ...._...F._).p..
00000080: 5fdb 5f92 0108 5f17 72a9 1b09 5ffd 8080  _._..._.r..._...
00000090: 000a 5f5f 5baf 000b 5f5f aa57 000c 5f9f  ..__[...__.W.._.
```

Keep in mind the header : `2000 0f14 0001` 😉

Private data parser

At address `0x2eb8`, we have a function that I called `parse_private_unit`.

```
            break
else
    rbp = 0
    stream->last_parser = 0
    stream->packet_len = 0xffff
    if (size u> 2)
        u8* data_1 = data
        if (*data_1 == 0x20 && data_1[1] == 0 && data_1[2] == 0xf)
            stream->state = 0
            data = data_1.b + 2
            ate_del = size - 2
            stream->packet_len = 0
            goto decode_loop
return rbp  {"+>"}  {">"}  {"@3"}
```

This part searches the header : `20000f` ! This confirms our lead.

The private stream is structured like :

```
private_stream = unit_group*
unit_group     = 0x2000 unit* 0xff
unit           = 0xf id 0x0001 len data
id             = 0x10 | 0x11 | 0x12 | 0x13 | 0x14 |  ...
```

The unit ID `0x10` to `0x14` is special cased.

```
if (state == 1)
    u8 private_unit_type = *data.b - 0x10
    if (private_unit_type u> 4)
        stream->last_parser = private_fallback
    else
        switch (jump_table_37a8[private_unit_type])
            case 0x2f9c  // 0x10
                stream->last_parser = private_10
            case 0x2fa5  // 0x11
                stream->last_parser = private_11
            case 0x2fae  // 0x12
                stream->last_parser = private_12
            case 0x2fb7  // 0x13
                stream->last_parser = private_13
            case 0x2fc0  // 0x14
                stream->last_parser = private_14
    stream->state = 2
```

For now it is a classic decoder, but when looking in each unit type parser, there is a function which pull bits from a global queue, and one who insert the bits in the packet.

```
switch (local_x.b)
    case 0, 2, 3, 4, 5, 7, 8, 0xa, 0xb, 0xd, 0x10
        local_x = zx.q(local_x.d + 1)
    case 1
        shift_amt = 0
        mw_bytes = pop_n_bit_to_exfiltrate(bit_count: 3)
        mask = 0xf8
    case 6, 9
        shift_amt = 0
        mw_bytes = pop_n_bit_to_exfiltrate(bit_count: 2)
        mask = 0xfc
    case 0xc
        local_x.b = **data
        stream->x = 0xd
        local_x.b = local_x.b u>> 6
        stream->y = local_x.b
    case 0xe
        shift_amt = 4
        mw_bytes = pop_n_bit_to_exfiltrate(bit_count: 4)
        mask = 0xf
    case 0xf
        local_x.b = stream->y
        local_x = zx.q((sbb.d(local_x.d - 1, local_x.d - 1, (local_x.d - 1).b u< 2) & 6) + 0xa)
    case 0x11
        stream->x = 0xa
switch (local_x.b)
    case 0, 2, 3, 4, 5, 7, 8, 0xa, 0xb, 0xd, 0xf, 0x10
        stream->x = local_x.b
    case 1, 6, 9, 0xe
        local_x = insert_bit_in_the_packet(data: *data, mask, mw_bits: mw_bytes, shift_amt)
        stream->x = stream->x + 1
```

Let's summarize what we got so far, the file `capture.ts` is a MPEG-TS file which contains a private data stream. The kernel module intercept each packet and insert some bits in the legit stream.

But wait, where does these bits come from ?

```
# Pseudo code after the attack (from part 1)
exfiltrate_bit_queue = []

def attack(cmd):
        stdout = exec_command_line(cmd)

        # encrypt stdout with AES CTR
        ctr_key = [ ... ] # constant 8 bytes
        key = hkdf(info=crt_key, needed_length=0x10)
        enc_buf = aes_ctr(key, data, nonce=current_timestamp())

        # push encrypted data to the queue
        exfiltrate_bit_queue.push(0xf408f1b2) # start tag
        exfiltrate_bit_queue.push(enc_buf)
        exfiltrate_bit_queue.push(0x0bf70e4d) # end tag
```

After the command, the output is encrypted with the current `prk` key with the current timestamp as the nonce.

To avoid false positives, the binary adds a start and an end tag.

So all this makes sense, in the `capture.ts` there is the flag encrypted and spread all over in tiny chunks of bits !

Find and rebuild the flag

I wrote a python script that parses and extracts all bits of the flag, this was tedious, and I double checked every decompiled code from binary ninja and Ghidra to ensure that my script behave exactly like the kernel module.

After some trial and error, I got an output which contains the start and the end tag !

```
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffb2f108f469a7e1c7c22
4ba0627519fea6f817a54a4c9bf44240532d9ea8118e095aacdbdbdb1631efcc7cbd6da8e25644
67bacc210c3f3fbe1e04c1fc5129554607ecc2e6b00f409d3e3f64d0ef70bfffffffffff
```

After extracting the data, removing these 2 tags, I bruteforce the nonce with timestamps around the date found in the pcap file.

```
FCSC{c70b3d645cfdef72c12848e1d2b96524aa85a60c0d91378b58b2529ec549f3e5}
```