SSTIC 2021 Challenge: Write-up

May 20, 2021

SSTIC 2021 Challenge

SSTIC is an annual French IT security conference. Every year they put out a challenge. For quite some time I've heard good things about the SSTIC challenge but I had never tried it myself, until now that is. This year I decided to give it a go. It consisted of five steps with the last three revolving around a DRM system and the first two being something of a prologue. The goal was to find an email address at the end of the last step and send an email to it. Optionally, you could pick up flags on the way and mark your progress on the SSTIC website.

This is my write-up of the challenges. However, note that I have written this with a lot of hindsight. I spent six weeks trying to solve the challenges and during the steps I went back and forth between different parts of the challenge, different ideas, etc. Instead of trying to show exactly what I did in which order I have reordered my findings in a more linear fashion to make it easier to follow and not have this write-up be unbearably long. If you have any comments or question about my explanation, feel free to reach out to me.

Part 1 - USB Forensics

The challenge starts with a pcap file containing USB traffic. Inspecting it in Wireshark reveals that it is a file transfer to some kind of USB drive using the SCSI over USB protocol. When writing a file to a drive you typically need to do at least two things: transfer the actual file data and update the file system information. Thus, we can expect to see some data relating to each of those two things. We don't really care about the file system data but want extract the data corresponding to the file being written. Unfortunately the SCSI dissector in Wireshark doesn't seem very good. A lot of the values seem to come out as the wrong datatypes and some fields are not really tagged at all. To handle this I wrote a small Python script that uses pyshark and does some processing of the packets to extract the blocks being written.

```
#!/usr/bin/env python3

import pyshark

PCAP_FILE = 'usb_capture_CO.pcapng'
cap = pyshark.FileCapture(PCAP_FILE, include_raw=True, use_json=True)

writes = {}
next_block_write = None
for i, pkt in enumerate(cap):
    if 'scsi' not in pkt:
        continue

    if pkt.scsi.get('scsi_sbc.opcode') == '42':
        if next_block_write:
            writes[next_block_write] = writes.get(next_block_write, -1) +
            print(f'Store write block {next_block_write} ({writes[next_blo
            with open(f'writes/{next_block_write}_{writes[next_block_write
                fout_write.write(pkt.get_raw_packet()[0x40:])
            next_block_write = None

        elif pkt.scsi.get('scsi_sbc.rdwr10.lba') != None:
            next_block_write = int(pkt.scsi.get('scsi_sbc.rdwr10.lba'))
            print(f'Found write block {next_block_write}')
```

The script will do the following:

1. If the current packet is not a SCSI packet, skip to the next
2. If the packet is a SCSI Block Command (SBC) with opcode 42 we will process it
3. If the packet is a read/write command and has its Logical Block Address (LBA) set, we will store this address in `next_block_write` and know that the next packet we will see will contain the actual data being written to that address.
4. If `next_block_write` is set, we will extract all the packet data starting at offset 0x40 and save it in a file named after the address and how many times we have written to the address (in case something is overwritten)

Running this script in the PCAP gives us a bunch of separate blocks. Combining the following block: 33055_0.dat, 33311_0.dat, 33695_0.dat and 33951_0.dat gives us a valid 7-zip archive containing four files. One of the files is a slightly corrupted JPG file with the first flag.

Step 1 flag

Part 2 - Windows Exploitation

In the archive file we also find three additional files:

- Readme.md - Some story background and hints on what to do next
- A..Mazing.exe - A Windows PE executable which we want to exploit
- env.txt - Information about the environment the remote server is running

The readme file contains the following:

> *Hey Trou,*
>
> *Do you remember the discussion we had last year at the secret SSTIC party? We planned to create the next SSTIC challenge to prove that we are still skilled enough to be trusted by the community.*
> *I attached the alpha version of my amazing challenge based on maze solving. You can play with it in order to hunt some remaining bugs. It's hosted on my workstation at home, you can reach it at challenge2021.sstic.org:4577. I've written in the env.txt file all the information about the remote configuration if needed.*
>
> *Have Fun,*

Running the program gives a prompt of the various actions we can perform:

> *Menu*
>
> 1. *Register*
> 2. *Create maze*
> 3. *Load maze*
> 4. *Play maze*
> 5. *Remove maze*
> 6. *View scoreboard*
> 7. *Upgrade*
> 8. *Exit*

The game allows you to setting your player name through the "register" option and then create or load a maze which you can play. Playing and solving a maze gives you a score (lower is better) based on the number of moves plus the value of any traps you have walked on. This scoreboard can then be viewed. There are three different variants of mazes and you can upgrade from a lower tier to a higher through the update option. When you create or upgrade the maze or when the highscore is updated, the data will be written to disk in two files: .maze and .rank. When loading a maze you can either specify the full filename .maze or leave out the suffix and only load in which case the program will check if there is an exact match and otherwise append .maze and try that. This ambiguity leads to the first vulnerability but before describing it or the other bugs we need to understand the main data structures involved.

The main game state and maze structures look like this:

```
struct game_state {
    uint64_t score;
    uint8_t pos_x;
    uint8_t pos_y;
    uint8_t player_name[128];
    maze *current_maze;
};

struct maze {
    uint8_t width;
    uint8_t height;
    uint8_t level;
    uint8_t maze_name[128];
    uint8_t player_name[128];

    uint8_t num_traps;
    struct trap {
        uint64_t penalty;
        uint16_t offset;
        uint8_t icon;
        uint32_t active;
    } traps[256];

    uint8_t *cells;
    uint8_t wall_remove;

    uint8_t num_highscore;
    struct highscore {
        uint64_t score;
        uint8_t player_name[128];
    } highscores[128];
};
```

Most of the fields should be fairly self-explanatory I will calrify two of them. The `level` field indicates the type of maze, 1 is “Classic maze”, 2 is “Multipass maze” and 3 is “Multipass maze with traps”. The `wall_remove` field indicates how many percent of the walls should be removed in a “multipass maze”, i.e. a maze where there is more than one path to the goal. When a maze is saved, it is stored as two files: a maze file and a rank file. The maze file always begin with the some basic information represented by the `maze_file` struct and if it is a level 3 maze it also contains additional data represented by the `maze_file_traps`. The rank file contains the hishscore data represented by the `rank_file` struct. These structures are listed below:

```
struct maze_file {
    uint8_t name_len;
    uint8_t player_name[name_len];
    uint8_t level;
    uint8_t width;
    uint8_t height;
    uint8_t cells[width*height];
};

struct maze_file_traps {
    uint8_t num_traps;
    struct file_trap {
        uint64_t penalty;
        uint16_t offset;
        uint8_t icon;
    } traps[num_traps];
};

struct rank_file {
    uint8_t num_scores;
    struct score {
        uint8_t name_len;
        uint8_t player_name[name_len];
        uint64_t score;
    } highcores[num_scores];
};
```

As mentioned above, the first bug occurs when loading a maze. Since the loader allows specifying the file name as or .maze we can create a maze called `foo` which will create two files `foo.maze` and `foo.rank`. We can then load `foo.rank` _as a maze_ so that the highscore data will be interpreted as a maze. It will then also to load a file called `foo.rank.rank` as the highscore data. This will fail but does not crash the program. Looking at the contents of the rank file we see that we can control most of the data in the file with a few constraints. The name is read with `scanf_s("%s", ...)` so it can contain any non-whitespace character. The score is affected by the number of moves we make plus penalties from traps. Unfortunately when creating a maze, the score value of a trap is read with `scanf_s("%d", ...)` so realistically, the upper ~28 bits of score will be either all 0 or all 1 (if negative penalties are used). The number of score entries and length of the names are of course also controllable but the file needs to be long enough so `num_scores` can't be too small. The take-away is that we can create a maze object where we control many of the attributes.

We can then use this to control the cells pointer. There's also an off-by-one vulnerability in the highscore feature which allows us to leak a heap address. Using the upgrade maze this then gives us a arbitrary read/write primitive. Using this we can first read heap memory, traverse the heap and leak a pointer to ntdll. From there we can leak a pointer to PEB, then we can use a fixed offset on that to leak TEB which gives us the stack. We can then write a ROP chain and shellcode to the stack, have the ROP chain call mprotect to make the stack executable which will run the shellcode and launch a shell.

Unfortuantely we are very memory constrained but we can launch a small Powershell shim to read the zip file from the server chunk by chunk like this:

```
$bufSize = 64
$file = "C:\\users\\challenge\\Desktop\\DRM.zip"
$fileStream = [System.IO.File]::OpenRead($file)
$chunk = New-Object byte[] $bufSize
while ( $bytesRead = $fileStream.Read($chunk, 0, $bufSize) ){
    [Convert]::ToBase64String($chunk)
}
$fileStream.Close()
```

Intermission - The DRM System

Once we have the zip file fully downloaded we can extract it and find the following files:

- Readme - A readme explaining the DRM system
- libchall_plugin.so - A plugin for the VLC media player which communicates with the media and key servers.
- DRM_server.tar.gz - A local, qemu-based instance of the key server setup without the HSM device attached.

The readme looks like this:

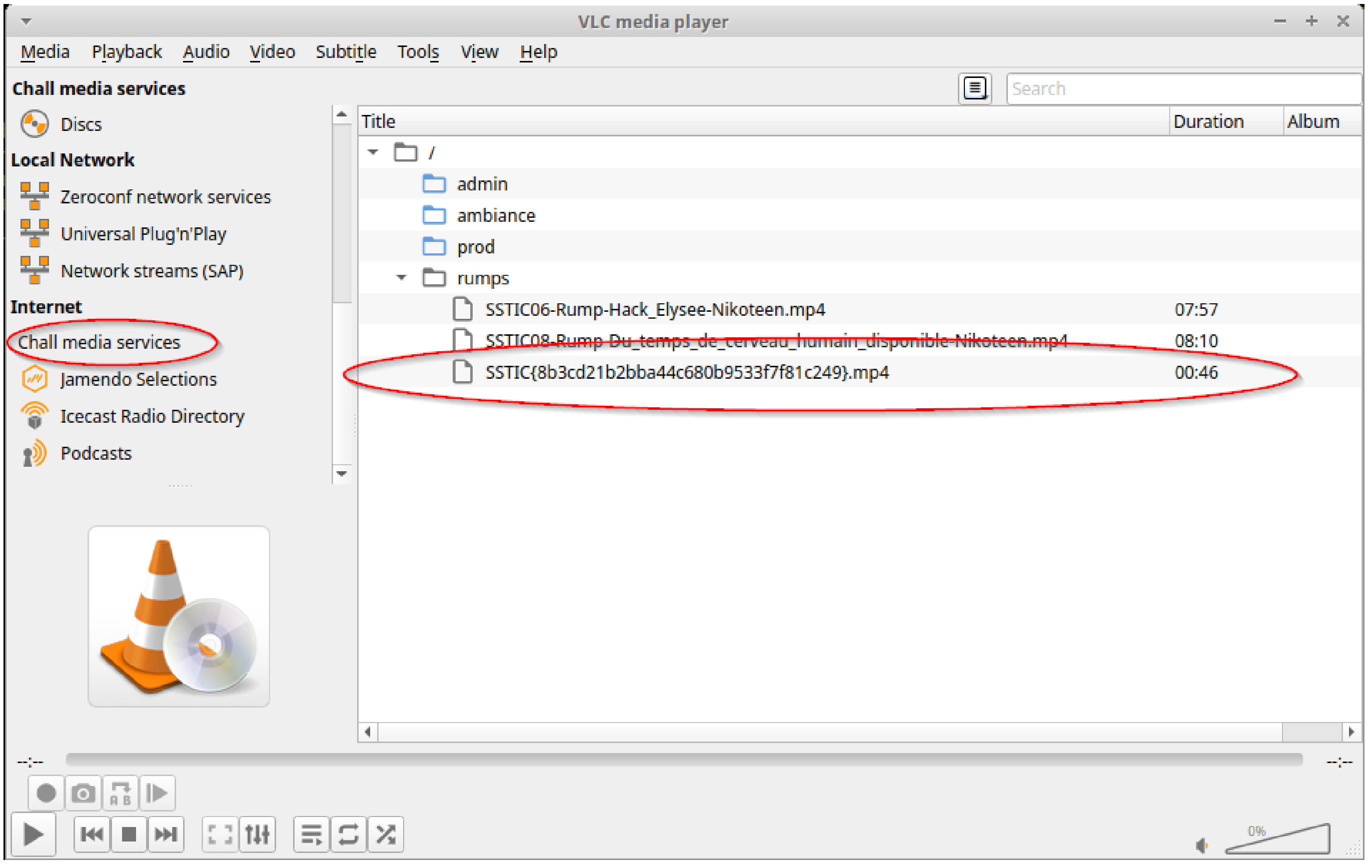
> *Here is a prototype of the DRM solution we plan to use for SSTIC 2021. It's 100% secure, because keys are stored on a device specifically designed for this. It uses a custom*

> *architecture which garantee even more security! In any case, the device is configured in debug mode so production keys can't be accessed.*
>
> *The file DRM_server.tar.gz is the remote part of the solution, but for now we can't emulate the device, so some feature are only available remotely. The file libchall_plugin.so is a VLC plugin that will allow you to test the solution, if you ever decide to install Linux :)*
>
> *Trou*

The plugin can be installed by copying it to a subdirectory of the VLC plugin directory (`/usr/lib/x86_64-linux-gnu/vlc/plugins/sstic` on my machine). After doing this, the plugin options can be viewed:

```
$ vlc -p chall
VLC media player 3.0.9.2 Vetinari (revision 3.0.9.2-0-gd4c1aefe4d)

 Chall media services (chall)
      --media-server <string>                  media server URL
      --key-server-addr <string>               key server address
      --key-server-port <integer [1 .. 65535]> key server port
      --media-server-login <string>            Login
      --media-server-pass <string>             Password
```

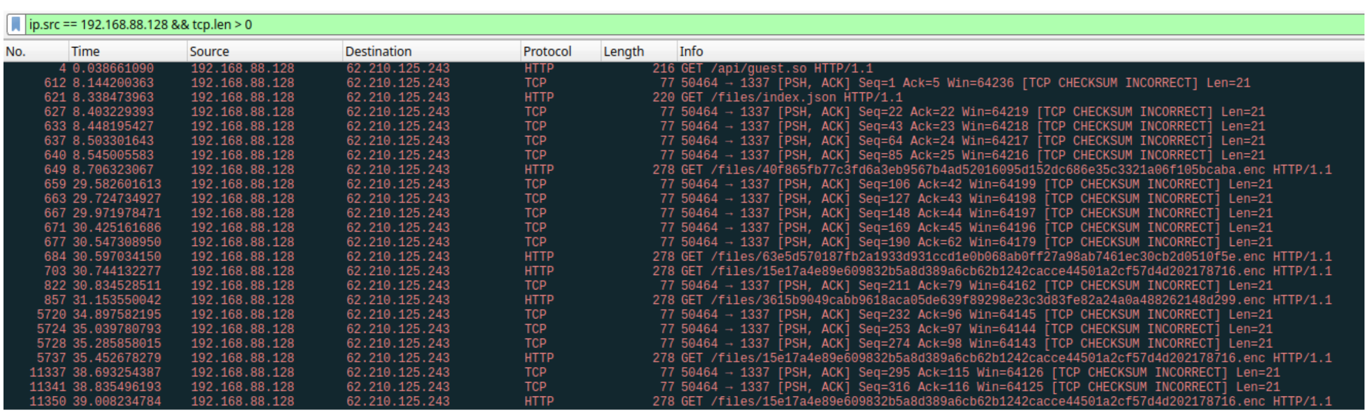
We don't know what these values indicate at the moment but by simply starting VLC and looking at the media browser menu we find a new entry named "Chall media services". Clicking it takes us to a media browser view where we only have access to one of the four directories called "rumps" in which we find the second flag.



By reverse engineering the plugin file we can find the options and their default values:

```
a1(a2, v4, 4096LL, (__int64 *)"media-server");
a1(a2, v4, 4097LL, (__int64 *)"http://challenge2021.sstic.org:8080");
...
a1(a2, v4, 4096LL, (__int64 *)"key-server-addr");
a1(a2, v4, 4097LL, (__int64 *)"62.210.125.243");
```

If we look in Wireshark while redoing the above mentioned browsing and limiting ourself to these hosts (actually host, the domain points to the same ip), we can see what's going on.



First, VLC downloads a guest.so file from the media server (port 8080) and sends a request to the key server (port 1337), then follows a sequence of first requesting a file from the media server and then sending requests to the key server. Out of these, only index.json is readable and looks like this:

```
[
  {
    "name": "930e553d6a3920d05c99bc3111aaf288a94e7961b03e1914ca5bcda32ba94
    "real_name": "admin",
    "type": "dir_index",
    "perms": "0000000000000000",
    "ident": "75edff360609c9f7"
  },
  {
    "name": "4e40398697616f77509274494b08a687dd5cc1a7c7a5720c75782ab9b3cf9
    "real_name": "ambiance",
    "type": "dir_index",
    "perms": "00000000cc90ebfe",
    "ident": "6811af029018505f"
  },
  {
    "name": "e1428828ed32e37beba57986db574aae48fde02a85c092ac0d358b39094b2
    "real_name": "prod",
    "type": "dir_index",
    "perms": "0000000000001000",
    "ident": "d603c7e177f13c40"
  },
  {
    "name": "40f865fb77c3fd6a3eb9567b4ad52016095d152dc686e35c3321a06f105bc
    "real_name": "rumps",
    "type": "dir_index",
    "perms": "ffffffffffffffff",
    "ident": "68963b6c026c3642"
  }
]
```

These four entries correspond to the directories we saw when browsing the media. Further looking inside the VLC plugin we can see that it decrypts the data it gets using AES-CTR with a nonce set to 0 and the initial counter to 1. We can also find references to both a `guest.so` and an auth.so depending on whether your username/password is set. Trying to access `auth.so` directly in the browser for example gives an HTTP basic auth prompt to which we don't have credentials.

```
if ( asprintf(&ptr, "%s/files/%s", *(const char **)(a1 + 24), a2) == -1 )
    ...
}
...
// GCRY_CIPHER_AES = 7, GCRY_CIPHER_MODE_CTR = 6
if ( (unsigned int)gcry_cipher_open(&gcry_hdl, 7LL, 6LL, 0LL) )  {
    ...
}
if ( (unsigned int)gcry_cipher_setkey(gcry_hdl, a3, 16LL) ) {
    ...
}
// Set nonce=0, ctr=1
if ( (unsigned int)gcry_cipher_setctr(gcry_hdl, &default_counter, 16LL) )
    ...
}


...
    v6 = asprintf(&ptr, "%s://%s:%s@%s/%s/api/auth.so", s[0], user_state->
...
else if ( asprintf(&ptr, "%s/api/guest.so", (const char *)user_state->fiel
...
```

We now turn our attention to the `guest.so` file. The library exports the following functions:

- useVm(char *in, char *out)
- getPerms(char *out)
- getIdent(char *out)

When the library is downloaded by VLC, it is loaded and handles to the three functions are stored:

```
user_state->ident_hdl = (__int64)dlopen(templatea, 1);
unlink(templatea);
handle = (void *)user_state->ident_hdl;
if ( !handle )
return 255LL;
useVM = (unsigned int (__fastcall *)(char *, char *))dlsym(handle, "useVM"
handle2 = (void *)user_state->ident_hdl;
user_state->useVM = useVM;
getPerms = (void (__fastcall *)(__int64 *))dlsym(handle2, "getPerms");
handle3 = (void *)user_state->ident_hdl;
user_state->getPerms = getPerms;
```

```
getIdent = (void (__fastcall *)(char *))dlsym(handle3, "getIdent");
v16 = user_state->useVM == 0LL;
```

From the usage later in the plugin, we see that `getPerms` and `getIdent` write 8 and 4 bytes respectively to the pointer you provide. The `useVm` function takes an 16 byte out of which the last 8 are the output from `getPerms` value and writes 16 bytes to the out pointer.

```
*(_QWORD *)vm_in = 0LL;
state->getPerms((__int64 *)&vm_in[8]);
if ( !state->useVM(vm_in, vm_out) )
```

We can write a small program to load and call these functions. We don't know what the first half of the argument to `useVm` is supposed to be so let's just set some easily identifiable value.

```
int main(int argc, char** argv, char** envp) {
    int res;
    char buf1[16], buf2[16];
    void *guest_lib = dlopen(argv[1], RTLD_NOW);
    ...
    int (*useVM)(char *in, char *out) = dlsym(guest_lib, "useVM");
    ...
    int (*getPerms)(char *out) = dlsym(guest_lib, "getPerms");
    ...
    int (*getIdent)(char *out) = dlsym(guest_lib, "getIdent");
    ...

    res = getIdent(buf1);
    printf("getIdent: %d\n", res);
    hexdump(buf1, 4);

    res = getPerms(&buf1[8]);
    printf("getPerms: %d\n", res);
    hexdump(&buf1[8], 8);

    *(long *)buf1 = 0x0011223344556677;
    res = useVM(buf1, buf2);
    printf("useVM: %d\n", res);
    hexdump(buf2, 16);

    return 0;
}
```

Running this gives us the following output:

> *getIdent: 0 0x000000: c9 5d a4 60 .].`*
>
> *getPerms: 0 0x000000: ff ff ff ff ff ff ff ff ……..*
>
> *useVM: 0 0x000000: be c7 c4 45 70 e7 a1 c0 04 59 36 da 6b 20 76 e2 …Ep….Y6.k v.*

Furthermore, we can see that the outputs from `getIdent` and `useVm` are sent together with a constant 0 to the key server:

```
v4 = _mm_loadu_si128((const __m128i *)sig);
v5 = *((_DWORD *)sig + 4);
v7 = 17LL;
packet[0] = 0;
*(__m128i *)&packet[1] = v4;
*(_DWORD *)&packet[17] = v5;
result = send_recv_constprop_1(state, (__int64)packet, (__int64)&v8, &v7);
```

Let's look at what the key server does with this. The tarball we got contains three files:

- bzImage - A Linux 5.10.27 kernel
- rootfs.img - The filesystem to be mounted in the VM
- run_qemu.sh - A shell to run qemu with the correct configuration

We can further unpack the `rootfs.img` to get a small set of files available inside the VM:

- /bin/busybox - A busybox binary
- /etc/{group,hosts,passwd} - Very simple user files
- /home/sstic/service - An ELF executable binary
- /lib/sstic.ko - An ELF kernel module

Starting the qemu VM will run the `service` binary which acts as a server. The service will read 17 bytes and treat the first byte as a command type. The command with id 0 will then read 4 bytes and treat it as a timestamp. It will check that the timestamp is no more than 3600 seconds (1 hour) old. It will then pass the 16 bytes and the timestamp to a function which will use `ioctl()` on a handle to `/dev/sstic` to decrypt the 16 byte value. Thus, we can describe the packet that VLC sends like this:

```
struct packet_header {
    uint8_t command;
    uint8_t payload[16];
    uint32_t timestamp;
}
```

Studying the service some more reveals that there are four different packet commands and that the 16 byte payload consists of an 8 byte key id and 8 byte permissions:

- 0 - Validate: Check that `timestamp` is not too old, decrypt `payload` and send back the results.
- 1 - Get key: Check `timestamp`, decrypt `payload` and fetch the key stated in `payload`
- 2 - Execute: Check `timestamp`, decrypt `payload` and execute code - We'll get back to this in part 4
- 3 - Execute debug: Same as above but also provide debug output, again, will be relevant in part 4.

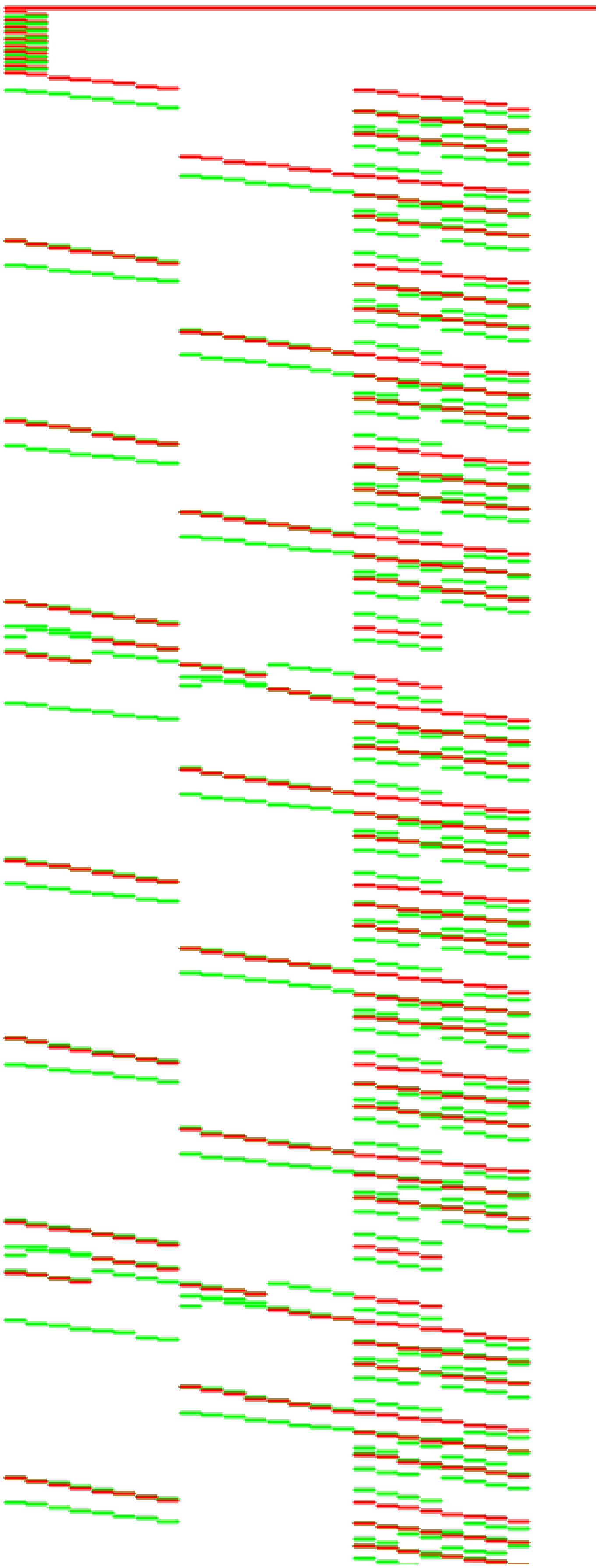
Calling command 0 and 1 requires the permission to be less than or equal to `0xFFFFFFFFFFFFFFFF` (always true), but command 2 and 3 requires it to be less than or equal to `0x100` and `0x10` respectively. The get key command will take the 64 bit key id, check that your permission is high enough (lower value means higher permission) for that key and call a function which uses `ioctl` on the `/dev/sstic` handle to fetch the key and return it to the VLC plugin.

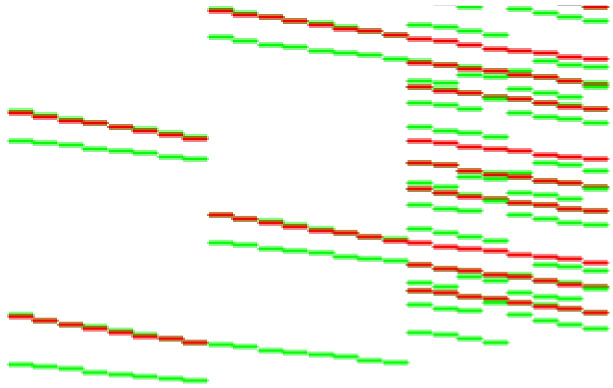
At this point we know that DRM system works by downloading the `guest.so` file and the `index.json`, using the library functions to generate packets and make a request to the key server to get a key to decrypt the content. This is then repeated for every entry. Thus, without the correct permission level you will not get the key for the media.

Part 3 - Whitebox Cryptography

We now ask, how is the encrypted key id and permission block created by the `guest.so` file? By reverse engineering the `guest.so` library we can see roughly how it works. The library consists of a few functions and a massive block of data. In the `.init_array` array we find a small function which will decrypt the huge chunk of data, again using AES-CTR with a nonce of 0 and an initial counter of 1 but with a key hard-coded in the library. This is simply an obfuscation layer. It should be noted that the AES implementation is very odd and seems to expand up every bit into a byte of `0x00` or `0xFF` and then use vector instructions to implement the AES algorithm. As a result of this, the round key table is `11*128` *bytes* large instead of the usual `11*128` *bits*. Next, all of the three exported functions in turn call a single function which is a VM with the usual switch statement in a loop structure. The VM seems to implement about ten or so instructions. The decrypted huge chunk of data contains the code for the VM. Instead of trying to disassemble and understand the VM and the VM code I instead went for a dynamic analysis approach.

Using Tracer and Intel PIN I traced an execution of the small program from before calling the library functions. At first this completely choked and after running for a long time and producing gigabytes of output I had to stop it. To make it more managable, I NOP:ed out the decryption function and performed the decryption outside the library and patched it with the huge chunk of data already decrypted. Running again now quickly produced a trace. Looking at it quickly reveals a very interesting pattern:





The execution can interpreted as some pattern repeating `3*6` times with another pattern in between, all while reading and writing chunks of 64 bits of data. This matches the structure of the Camellia cipher which is a block cipher with an 18 round Feistel Network with a 128 bit block size.

The implementation of Camellia is not a vanilla implementation but instead a whitebox implementation implemented with lookup tables. Using a combination of pen & paper, linear algebra, Sagemath, Z3 and Qiling this can be broken.

Finally, we manage to extract the key used in the encryption which means we now can encrypt our own blocks with whatever value we want. We can use this to craft a packet with the highest permission level and fetch more keys from the key server. Unfortunately, keys with permission level 0 can't be fetched despite us having the highest level. The same applies for the keys with their highest id bit set if we are in debug mode (which we are). Regardless, we still manage to get the keys corresponding to the `ambience` directory. We can then manually download the corresponging `*.enc` files from the media server and decrypt them to finally get the third flag.

Part 4 - Blackbox Reverse Engineering

With the whitebox crypto broken, we can keep forging admin keys and therefore access command 2 and 3 on the key server. Command 3 will send a piece of hard-coded data to the device together with input from us using a series of `ioctl` commands. The sequence of `ioctl` commands essentially sets up a call to the VM inside the device. It maps four regions of memory: stdin, stdout, code and debug data, associates those regions with their respective function inside the device, executes and reads the results. It will then check that the output contains a sequence of 48 `0xFF` bytes followed by the string `EXECUTE FILE OK!`. If the check passes, we are allowed to upload any file which will be executed, giving us code execution on the server. If we instead call command 2 on the server, it performs the same sequence of calls to the device but we get to choose the code and we get the debug output. The debug output contains the state of all registers and the stack.

Using this function, we can start sending pieces of the hard-coded code and observe the state of the registers to reverse engineer the architecture so that we eventually can reverse engineer the password checking program. The architecture has 16 bit addresses and program counter, 8 general purpose 128 bit registers named `R0-7` and a special `RC` register. The instructions are all 4 bytes long. What followed was a process of sending sequences of instructions, observing the states of the registers, forming an hypothesis of what the instructions were doing and so on. Throughout, I created an architecture plugin for Binary Ninja and using this I could disassemble the first part of the password checker code.

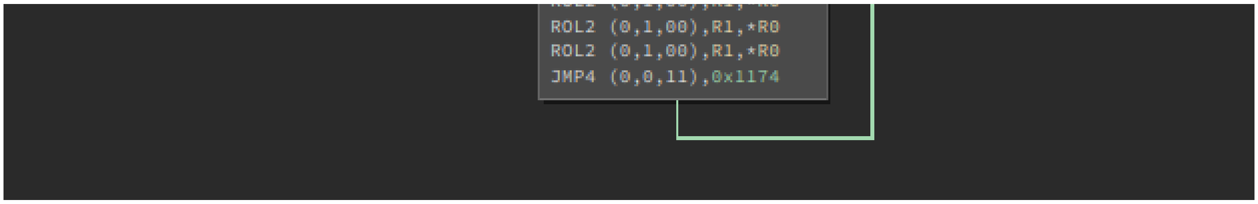
This part of the code looked at 16 bytes of the input and checked that it satisfied a series of contraints. I transformed this into a Z3 script to extract it.

```
#!/usr/bin/env python3
from z3 import *

c1 = BitVecVal(int.from_bytes(bytes.fromhex('0e03070a9e040c0b2c0dd30774026
c2 = BitVecVal(int.from_bytes(bytes.fromhex('0e03040a88b3060b000b0d070f029
c3 = BitVecVal(int.from_bytes(bytes.fromhex('0e870b8a1c04090b001c0d070f020
c4 = BitVecVal(int.from_bytes(bytes.fromhex('000c0d07000c0d07000c0d07000c0
c5 = BitVecVal(int.from_bytes(bytes.fromhex('0f0206010f0206010f0206010f020

password = [BitVec(f'p_{i}', 8) for i in range(16)]
password2 = [Concat(*ps[::-1]) for ps in zip(password[::2], password[1::2]
password3 = [Concat(*ps[::-1]) for ps in zip(password[::4], password[1::4]
password4 = [Concat(*ps[::-1]) for ps in zip(password[::8], password[1::8]

s = Solver()
for p in password:
    s.add(p < 0x10)
    s.add(p >= 0)

# Part 1
s.add(Distinct(password))

# Part 2
for i in range(0, 8):
    c1_part = simplify(Extract(16*i+15, 16*i, c1))
    print(hex(c1_part.as_long()))
    s.add(c1_part >= password2[i])

print('---')

# Part 3
```

```
for i in range(0, 4):
    c2_part = simplify(Extract(32*i+31, 32*i, c2))
    print(hex(c2_part.as_long()))
    s.add(c2_part <= password3[i])

# Part 4
for i in range(0, 2):
    c3_part = simplify(Extract(64*i+63, 64*i, c3))
    print(hex(c3_part.as_long()))
    s.add(c3_part > password4[i])

# Part 5
parts1 = []
for i in range(0, 4):
    c4_part = simplify(Extract(32*i+31, 32*i, c4))
    print(hex(c4_part.as_long()))
    parts1.append(c4_part == password3[i])
s.add(Or(*parts1))

# Part 5
parts2 = []
for i in range(0, 4):
    c5_part = simplify(Extract(32*i+31, 32*i, c5))
    print(hex(c5_part.as_long()))
    parts2.append(c5_part == password3[i])
s.add(Or(*parts2))

# Part 6
parts3 = []
for p in password2:
    parts3.append(p == 0x0408)
s.add(Or(*parts3))

if s.check() == sat:
    m = s.model()
    val = [m[p].as_long() for p in password]
    print(val)
else:
    print('unsat')
```

Running this gave me the correct 16 bytes `[14, 3, 5, 10, 8, 4, 9, 11, 0, 12, 13, 7, 15, 2, 6, 1]`. This value was then used to decrypt the next stage of the program which looked like this in my Binary Ninja plugin:


sub_1100:
XOR4 (0,1,10),R1,R1
CMP4 (0,1,11),R1,0x40
JMP4 (7,0,11),0x112c
MOV4 (0,7,11),R7,0x2000
MOV4 (0,6,11),R6,0x3000
ADD4 (0,7,10),R7,R1
ADD4 (0,6,10),R6,R1
LOD4 (0,0,00),R0,*R7
STR4 (0,0,00),R0,*R6
ADD4 (0,1,11),R1,0x10
JMP4 (0,0,11),0x1104
XOR4 (0,7,10),R7,R7
CMP4 (0,7,11),R7,0x14
JMP4 (7,0,11),0x1238
MOV4 (0,0,11),R0,0x2000
MOV4 (0,2,11),R2,0x100
LOD4 (0,1,01),R1,*0x3000
LOD4 (0,3,00),R3,*R2
ADD2 (0,1,10),R1,R0
XOR4 (0,1,10),R1,R3
STR4 (0,1,01),R1,*0x3000
ADD4 (0,0,11),R0,0x10
ADD4 (0,2,11),R2,0x10
LOD4 (0,1,01),R1,*0x3010
LOD4 (0,3,00),R3,*R2
ADD2 (0,1,10),R1,R0
XOR4 (0,1,10),R1,R3
STR4 (0,1,01),R1,*0x3010
ADD4 (0,0,11),R0,0x10
ADD4 (0,2,11),R2,0x10
LOD4 (0,1,01),R1,*0x3020
LOD4 (0,3,00),R3,*R2
ADD2 (0,1,10),R1,R0
XOR4 (0,1,10),R1,R3
STR4 (0,1,01),R1,*0x3020
ADD4 (0,0,11),R0,0x10
ADD4 (0,2,11),R2,0x10
LOD4 (0,1,01),R1,*0x3030
LOD4 (0,3,00),R3,*R2
ADD2 (0,1,10),R1,R0
XOR4 (0,1,10),R1,R3
STR4 (0,1,01),R1,*0x3030
RET0 (0,0,00),R0,*R0
MOV4 (0,6,11),R6,0x3000
LOD4 (0,0,00),R0,*R6
ADD4 (0,6,11),R6,0x10
LOD4 (0,1,00),R1,*R6
ADD4 (0,6,11),R6,0x10
LOD4 (0,2,00),R2,*R6
ADD4 (0,6,11),R6,0x10
LOD4 (0,3,00),R3,*R6
XOR4 (0,6,10),R6,R6
ADD4 (0,6,11),R6,0x01
AND4 (0,6,10),R6,R7
XOR4 (0,5,10),R5,R5
CMP4 (0,6,10),R6,R5
JMP4 (5,0,11),0x1198
CAL7 (0,0,11),R0,0x11d0
MOV4 (0,6,11),R6,0x3000
STR4 (0,0,00),R0,*R6
ADD4 (0,6,11),R6,0x10
STR4 (0,1,00),R1,*R6
ADD4 (0,6,11),R6,0x10
STR4 (0,2,00),R2,*R6
ADD4 (0,6,11),R6,0x10
STR4 (0,3,00),R3,*R6
JMP4 (0,0,11),0x1130
ROL2 (0,1,00),R1,*R0
ROL2 (0,2,00),R2,*R0
ROL2 (0,2,00),R2,*R0
ROL2 (0,3,00),R3,*R0
ROL2 (0,3,00),R3,*R0
ROL2 (0,3,00),R3,*R0
CAL7 (0,0,11),R0,0x11d0
ROL2 (0,3,00),R3,*R0
ROL2 (0,2,00),R2,*R0
ROL2 (0,2,00),R2,*R0
ROL2 (0,1,00),R1,*R0



This code takes the remaining 48 bytes of the input and runs it through 20 iterations of a transformation function and performing some extra rotations every other iteration. Finally they apply a few invertible operations such as XOR:ing with some constants and outputting the result. By implementing the algorithm in Python, validating it by comparing the values from the real VM and finally inverting the operations, I was able to extract the correct 64 bytes of input.

Using this input, I could then call command 3, pass the check and upload a file to be executed. Remember what I said about not allowing keys with permission level 0 to be extraced from the server? This check is done in user-space so writing a small binary to interact with the device, compiling it staicallly and uploading it to the server, we can bypass that check and extract the keys.

```
int main() {
    int fd_dev = open("/dev/sstic", 2);

    long key_ids[] = {
        0x6FC51949A75BFA98, 0x583C5E51D0E1AB05, 0x675160EFED2D139B, 0x08AB
        0x3A8AD6D7F95E3487, 0x325149E3FC923A77, 0x46DCC15BCD2DB798, 0x4CE2
        0x675B9C51B9352849, 0x3B2C4583A5C9E4EB, 0x58B7CBFEC9E4BCE3, 0x272F
        0x6811AF029018505F, 0x59BDD204AA7112ED, 0x75EDFF360609C9F7
    };

    long get_key_cmd[] = { 0, 0, 0 };

    for(size_t i = 0; i < sizeof(key_ids)/sizeof(long); i++) {
        get_key_cmd[0] = key_ids[i];
        get_key_cmd[1] = 0;
        get_key_cmd[2] = 0;
        ioctl(fd_dev, 0xC0185304uLL, get_key_cmd);
        printf("id: %lx k1: %lx, k2 %lx\n", key_ids[i], get_key_cmd[1], ge
    }

    return 0;
}
```

This way, I could extract all the keys I already had plus the keys with permission level 0 but not the three keys marked as “production keys”, ie. the one with their highest bit set on the key id because this check is also performed in the kernel module (and, it later turned out, on the device itself).

Part 5 - Linux Kernel Exploitation

With user-space code execution capabilities, it was now time to find an exploit in the kernel module itself. As mentioned above when performing the execution functions, it is possible to allocate memory inside the device. This memory can then be accessed by calling `mmap` to map the memory to user space. However, there's a flag in the handling of the allocated memory which leads to a use-after-free vulnerability. The vulnerability can be triggered as follows:

Image: region/phy/pages objects diagram Code: exploit

1. allocate 32 pages inside the device
2. `mmap` those pages to get a user space mapping
3. delete the allocation (in the device)
4. remap page 16-24 (call this C)
5. remap page 8-16 (call this B)
6. remap C again (same size)
7. remap C again

At this point, both ranges B and C point to the same 8 physical memory pages. We then continue by:

1. Call `munmap` on range B
2. Allocate a bunch of new pages (call these E) and write a canary value to all of them
3. One of the pages in the C range now contains a page table, find which one of them, this is now our "control pointer"
4. Use the control pointer to modify the page table to point at physical page 0
5. Check which of the tables in the E range no longer contains the canary, this is our "read/write pointer".

Now we can set a physical page with the control pointer and then read or write to/from it using the read/write pointer.

1. Keep incrementing the page table entry in increments of 0x1000 using the control pointer.
2. Do this until the read/write pointer contains the driver code.
3. Modify the driver code in `ioctl_get_key` to instead of checking the debug flag perform `iowrite32(0, dev_hdl->debug)` to disable debug mode on the device.
4. Request to read the prod keys using the normal `ioctl` call.

This gives the production keys which now can be used to decrypt the `prod` directory and get the final flag. The email address we need is then found in the second video track of the video file in that directory.

Conclusion

Thanks a lot to the SSTIC challenge organisers for creating some really great challenges. I learnt a lot throughout these weeks. As I said in the beginning, if you have and questions or comments about this writeup, feel free to contact me on Twitter (@ZetaTwo) or email (calle.svensson@zeta-two.com).