# MWDB training

CERT Polska

Dec 23, 2023

# EXERCISES

CHAPTER

ONE

# WORKSHOP SLIDES

Slides from the hack.lu workshop can be found here

## 1.1 Part 1 - MWDB

### 1.1.1 Exercise #1.0: Getting familiar with the interface

MWDB welcomes us with a list of recently uploaded samples.

**Recent views** allow to see basic information about latest objects in the repository and interactively explore the dataset using Lucene-based queries and clickable fields. If sample was uploaded within last 72 hours, it is additionally marked with yellowish background. Yellow color is a bit more intense if file was uploaded at least 24 hours ago.

If you click on sample hash, you will navigate to the detailed sample view. Here you can see all the details about file. Left side contains a few tabs. The first one called Details presents basic file information like original file name, size, file type and hash values.

On the right side of view you can see tags, relations with other objects and the comments section. This information are added to MWDB mainly by our analysis backend, where sample is sent on the first upload. If analysis was successful, all interesting analysis artifacts are uploaded back to the MWDB.

During our tour we will go through all of these elements, starting from tags.

### 1.1.2 Exercise #1.1: Filtering samples by tags

**Goal**: get familiar with the interface, play around with the search query

1. Go to the main view and click on any tag starting with `runnable:` to include only runnable samples
2. Click on the  character of tag starting with `feed:` to exclude that feed from the results
3. Then click quick query `Only ripped:*` to include only the original, ripped samples.

   Let's take a look at the resulting query:

   ```
   tag:"runnable:win32:exe" AND NOT tag:"feed:malwarebazaar" AND tag:"ripped:*"
   ```

   Query language is based on Lucene syntax subset and consists of two basic elements:

   - field conditions `tag:"runnable:win32:exe"` that support wildcards `*`, `?`
   - operators: `OR`, `AND`, `NOT`

As you can see, we have various tags identifying the malware based on various criteria. If we want to find everything that is recognized as specific family regardless of the source of classification, we can just use the wildcards (like in `ripped:*` case):

```
tag:*formbook*
```

You probably also noticed that tags are colored and the color is not completely random. Explanation is on the slides.

4. Now, add an additional condition to the query:

```
tag:*formbook* AND size:[10000 TO 15000]
```

5. Lucene query language also supports ranges so we can search for samples of given size or uploaded on given date

```
tag:*formbook* AND size:[10000 TO 15000] AND upload_time:<=2021-01-01
```

### 1.1.3 Exercise #1.2: Exploring sample view and hierarchy

**Goal**: explore the sample view, understand the object hierarchy

1. Copy hash to the query field in Samples view: `5762523a60685aafa8a681672403fd19`

   Click on the hash in row to navigate to the sample details.

   Here you can see all the details about file. Left side contains three tabs: Details, Relations and Preview. The first one called Details presents basic file information like original file name, size, file type and hash values.

   Blue-colored fields are clickable, you can use them to quickly search for other samples matching the given criteria.

   Just beneath the file details tab, attributes are displayed. This section contains all sorts of miscellaneous information about the sample and/or analysis.

   On the right side of view you can see tags, relations with other objects and the comments section.

   As you deduce from tags, sample is a rar archive (`archive:rar`) and comes from Malwarebazaar (`feed:malwarebazaar`). Link to the sample on MalwareBazaar platform can be found in the Attributes section.

   In this case, related samples are the file that were unpacked from that archive.

2. See the `Related samples` box on the right. Go to the child sample tagged `ripped:formbook`

   This is the actual executable contained in the malicious archive. Based on tags we can say that:

   - `runnable:win32:exe` it is Windows 32-bit executable
   - `yara:win_formbook` One of our Yara rules matched this sample as Formbook
   - `et:formbook` ET Pro traffic rules matched this sample as Formbook (more info in comments)
   - `ripped:formbook` We have successfully ripped Formbook configuration from this sample

3. Navigate to the next child tagged `dump:win32:exe`

   This is the memory dump that contains the unpacked formbook payload. We got it by running the sample on our sandbox and then performing memory dumps when specific *interesting* prerequisites are matched. It's worth nothing that while the analysis produces a bunch of memory dumps, we upload the best one that allowed us to get the complete malware configuration.

4. Check `Static config` tab.

   See the extracted static configuration.

   Configuration is the second data type in MWDB. Malware configurations are meant to parametrize the malware behavior and they usually contain useful IoCs.

   The format of configuration depends on malware family, usually deriving from the structure "proposed" by the malware author.

5. Go back to the `Details` tab and make sure you're on MD5 `8e56eee9cf853d2ec4c695282c01fe0a`

   Go to the `Relations` tab. It presents the parents and children of current object. Notice how two distinct samples have been unpacked into the same malicious core.

6. Click on the Config box in the Relations graph to expand it. Zoom out the graph to see the whole graph.

### 1.1.4 Exercise #1.3: Looking for similar configurations

**Goal**: Find configurations that are similar to Formbook config

1. Click on the config hash (`f2e216695d4ce7233f5feb846bc81b8fffe9507988c7f5caaca680c0861e5e02`) in `Related configs` tab.

2. Go to the Preview tab

   Configurations are just a simple JSON objects. The only special thing is hashing algorithm e.g. lists are hashed non-orderwise, so if domains were ripped in different order, configuration hash will be still the same.

3. Go back to the Details tab. Expand `urls` and click on `www.discorddeno.land/suod/`

   ```
   cfg.urls*.url:"www.discorddeno.land/suod/"
   ```

   The resulting query looks for all `url` keys in `urls` lists that have `www.discorddeno.land/suod/`.

4. Let's check if `/suod/` path was used in other configs as well.

   Modify query to look for other configs with `/suod/` path replacing the domain with wildcard `*`.

   ```
   cfg.urls*.url:"*/suod/"
   ```

   There are two configurations. What URL was used in the second configuration?

5. Now let's check if `/suod/` occurs in other configurations regardless of the configuration structure. For that query we can use full-text search in JSON.

   ```
   cfg:"*/suod/*"
   ```

   Are there more configurations like that?

6. If not, let's search for configurations with .land TLD

   ```
   cfg:"*.land*"
   ```

7. Then click on `agenttesla` config (`e031b192d40f6d234756f8508f7d384db315983b57d8fc3216d20567056bd88b`) - you might have to scroll down a bit.

   **Note: Hint:** Instead of scrolling, you can help yourself by adding *AND family:agenttesla* to the query

Ok, there is no .land TLD. but .landa e-mail address. To ilustrate how full-text search works, go to Preview, press CTRL-F and type ".land" to see what parts of JSON were matched

How we can improve our query? Let's add " character at the end to match the end string.

```
cfg:"*.land\"*"
```

Go to the Gandcrab configuration and check in Preview what was matched.

### 1.1.5 Exercise #1.4: Blobs and dynamic configurations

**Goal**: Familiarize yourself with the blob object type

The third object type in MWDB is blob. While config represents structured (JSON) data, blob is an unstructured one. Blobs are just simple text files, usually containing some raw, but human-readable content.

Let's take a look at some examples.

1. Navigate to https://mwdb.cert.pl/blob/60c9ad80cde64e7cae9eec0c11dd98175860243aa40a3d8439bbf142d2a0e068

   What we see is bunch of decrypted strings from AgentTesla that were ripped from the malware sample.

   They're not structured because we don't semantically analyze every string, but it's still nice to have them in repository.

2. Jump to https://mwdb.cert.pl/blob/48914f0a6b9f4499da31d2217a7ee2e8c8f35f93ab5c992333f5c1aa947d9009

   We're now looking at decrypted strings from the Remcos family. Even if data is unstructured, it can be considered a part of static configuration and used in searching for malware similarities.

   Let's take a look at the parent of this blob: the static configuration object.

   https://mwdb.cert.pl/config/29c1f3c14a446b2a77ce58cbc59619fbfe7459c56fe1c8408597538384aa56ac

   Not much, just C2 host/port and password.

3. Let's take a look for another configuration with this host by expanding `c2` key and clicking at the `host` address.

   Oh, there is another one.

   The resulting query is `cfg.c2*.host:"ongod4life.ddns.net:4344"`

   You should be looking at: https://mwdb.cert.pl/config/9afac348443a7aa9ca5d33cffcc984751cebf15f065cb90b48911943fb10e1f6

   They're pretty much the same and only the `raw_cfg` differs. How to easily compare them?

4. Go to the blob (`da2055f0e90355bfaf3cc932f7fdb2f82bfd79c26f95b61b23b9cd77f9b0e32d`). In blob view, find the `Diff with` button on the right side of tabs. Click that button.

   Now we can choose another blob to compare.

   The simplest way is to copy to clipboard the previous blob id and paste it into query bar.

   `48914f0a6b9f4499da31d2217a7ee2e8c8f35f93ab5c992333f5c1aa947d9009`

   Then press ENTER and choose the searched blob. What's the difference between these blobs?

   **Warning:** This feature has known bug in v2.10.1, so go directly into this link: https://mwdb.cert.pl/diff/48914f0a6b9f4499da31d2217a7ee2e8c8f35f93ab5c992333f5c1aa947d9009/da2055f0e90355bfaf3cc932f7fdb2f82bfd79c26f95b61b23b9cd77f9b0e32d

   But blobs are not only the strings and unstructured static things.

5. Go to the `Blobs` list and click on `dyn_cfg` in `Blob type` column or type manually `type:dyn_cfg`. Then filter out `dynamic:mirai` tag (there are lots of them but they're not that interesting).
6. Check out the `hancitor` dynamic configuration.

   https://mwdb.cert.pl/blob/3b032876cc2d77d28625b9dfee0686663e60385cda7f9031afac6cf2b0c6d6e4

   Dynamic configurations also parametrize the malware behavior, but they're fetched from external source. In that case we have set of commands to run a second stage malware.

   Fetched second stage is linked as a blob child.

   Other possible types of dynamic configuration are injects, mail templates for spam botnets or malware updates.

   Example of more verbose Kronos dynamic configuration:

   https://mwdb.cert.pl/blob/2e4d109edb8b2fa7c1f1d7592a284bbf15e3e51d24d1d9cdda91c9ae582cf05c/config

   Blob children are files dropped from C&C and structured (parsed) fragments of dynamic configuration

### 1.1.6 Exercise #1.5: Let's upload something!

**Goal**: Learn how object sharing and access inheritance work.

All objects you've seen so far are shared with all MWDB accounts: the 'public' group.

If you use a query:

```
NOT shared:public
```

You should not have any results, because all samples you see are public. So how to gather some 'private' samples? You need to upload them!

1. Fetch an example sample from GitHub `ex5malware.zip`. **Don't unpack it**, just download to some temporary location.

   ```
   $ wget https://github.com/CERT-Polska/training-mwdb/raw/main/ex5malware.zip
   ```

2. Click on `Upload` in the navbar (https://mwdb.cert.pl/upload)

   Select the sample you have just downloaded.

3. Take a look at `Share with` options. There are four options:
   - `All my groups` sample will be shared with all your **private** groups, so it will be visible to the general public. E.g. it will be shared only within your organization
   - `Single group` in case you belong to multiple user groups, you can select a specific one
   - `Everybody`, everyone will see the sample
   - `Only me`, not event your colleagues from the same organization will be able to see the sample
4. Upload a sample with a default `All my groups` option.
5. Take a look at the Shares box. Who has access to your sample?
6. Use the Relations tab to traverse to the Config. Who has access to the configuration?

In MWDB sharing model, if you upload a private sample you get immediate access to all its descendants. So you always get all the data related with ripped configuration, but not necessarily all of its parents.

If you want, you can always change your mind and share the sample with somebody else. But you can't reverse the action, so if something was shared by mistake, contact the administrators.

7. Go back to the original sample and share it with `public` using `Share with group` input field.

## 1.2 Part 2 - mwdblib

### 1.2.1 Setup

Create a virtualenv and activate it

```
$ python3 -m venv venv
$ . venv/bin/activate
```

Install mwdblib with CLI extras + ipython shell for convenience

```
(venv) $ pip install mwdblib[cli] ipython
```

If you have problems during installation with `cryptography` package, try to upgrade pip and retry the previous step.

```
(venv) $ pip install -U pip
...
(venv) $ pip install mwdblib[cli] ipython
```

### 1.2.2 Exercise #2.1: Get information about 10 recent files using mwdblib

**Goal**: learn how the `recent_files` method works

The main interface is MWDB object that provides various methods to interact with MWDB. Let's start with log in to mwdb.cert.pl service.

```
In [1]: from mwdblib import MWDB

In [2]: mwdb = MWDB()

In [3]: mwdb.login()
Username: demologin
Password:
```

After successful login, let's begin with `recent_files` to get recently uploaded file from API.

```
In [4]: mwdb.recent_files()
Out[4]: <generator object MWDB._recent at ...>
```

`recent_files` function returns generator which does the same job as scrolling down the Samples view to view the older entries. Let's use the `next` function to get the most recent file:

```
In [5]: files = mwdb.recent_files()

In [6]: file = next(files)

In [7]: file
Out[7]: <mwdblib.file.MWDBFile at ...>
```

. . . and we got the file!

To get the next 10 files, we can use itertools.islice method:

```
In [8]: import itertools

In [9]: recent_10 = list(itertools.islice(files, 10))

In [10]: recent_10
Out[10]:
[<mwdblib.file.MWDBFile at ...>,
 <mwdblib.file.MWDBFile at ...>,
 <mwdblib.file.MWDBFile at ...>,
 <mwdblib.file.MWDBFile at ...>,
 <mwdblib.file.MWDBFile at ...>,
 <mwdblib.file.MWDBFile at ...>,
 <mwdblib.file.MWDBFile at ...>,
 <mwdblib.file.MWDBFile at ...>,
 <mwdblib.file.MWDBFile at ...>,
 <mwdblib.file.MWDBFile at ...>]
```

But what we can do with these file objects?

### 1.2.3 Exercise #2.2: Check properties of `780e8fb254e0b8c299f834f61dc80809`

**Objectives**:

- Check file's name, tags and children
- Get the first 16 bytes of the file
- Get the linked configuration of this file.
- Check names of the other files that are parents of that configuration

Let's start with getting a file by hash. Use `mwdb.query_file` method to get an object.

```
In [11]: file = mwdb.query_file("780e8fb254e0b8c299f834f61dc80809")

In [12]: file
Out[12]: <mwdblib.file.MWDBFile at ...>
```

Using the retrieved `MWDBFile` object we can get some details about the file e.g. name, tags, child objects.

```
In [13]: file.name
Out[13]: '400000_1973838fc27536e6'

In [14]: file.tags
Out[14]: ['dump:win32:exe', 'avemaria']

In [15]: file.children
Out[15]: [<mwdblib.file.MWDBConfig at ...>]
```

We can also download its contents

```
In [16]: file.download()[:16]
Out[16]: b'MZ\x90\x00\x03\x00\x00\x00\x04\x00\x00\x00\xff\xff\x00\x00'
```

As you can see in [15], there is a configuration attached to the file. We can get it by index operator or use config attribute to get the latest configuration object. Let's see what has been ripped:

```
In [17]: file.children[0].config
Out[17]: {'c2': [{'host': '172.111.210.207'}], 'type': 'avemaria'}

In [18]: file.config
Out[18]: <mwdblib.file.MWDBConfig at ...>

In [19]: file.config.config
Out[19]: {'c2': [{'host': '172.111.210.207'}], 'type': 'avemaria'}
```

Many malware samples can share the same configuration. Let's explore them:

```
In [20]: avemaria = file.config

In [21]: avemaria.parents
Out[21]:
[<mwdblib.file.MWDBFile at 0x7faed2383f10>,
 <mwdblib.file.MWDBFile at 0x7faed2383070>,
 <mwdblib.file.MWDBFile at 0x7faed2335b20>,
 <mwdblib.file.MWDBFile at 0x7faed2335a00>]

In [22]: [parent.name for parent in avemaria.parents]
In [22]:
['400000_2236f1a1cacde1dc',
 '400000_1973838fc27536e6',
 '400000_2bf452f7796153ef',
 '400000_3539b9d228df73c6']
```

**Task:** use `mwdb.search_configs("family:valak")` to iterate over the list of all configs related to the valak malware family. Use mwdblib to get a list of all URLs referenced by this family (config field `urls`). How many URLs are there in total? (Note: some URLs are repeating. For this exericse you don't have to worry about deduplicating them).

### 1.2.4 Exercise #2.3: Using mwdblib CLI

**Objectives**:

- Download 10 files that were tagged as `ripped:lokibot` using mwdblib CLI

1. First exit ipython using `exit()` or CTRL+D
2. Then type `mwdb` command in terminal and press ENTER

   mwdblib library installs CLI tool along with the Python binding which can be used in fancy oneliners and Bash scripts

   Login permanently using `mwdb login`. Try `mwdb login --no-keyring` if you have problems.
3. Type `mwdb list` to see the list of recent files (note: you can also try to add `-o short` or `-o json`)
4. Copy one of the hashes and paste as a `mwdb get <hash>` argument
5. Download file contents using `mwdb fetch <hash>`
6. Search for other files (e.g. ripped:lokibot) using `mwdb search 'tag:"ripped:lokibot"'`
7. You can also get only file hashes for further processing by adding `-o short -n 10`

(-o short means hash-only output and -n 10 fetches only 10 first files)

8. Make an oneliner that will download first 10 samples tagged as `ripped:lokibot`

Answer:

```
mwdb search 'tag:"ripped:lokibot"' -o short -n 10 | xargs -n 1 mwdb fetch
```

or

```
for f in $(mwdb search 'tag:"ripped:lokibot"' -o short -n 10); do mwdb fetch $f; done
```

**Task** Use `mwdb get` to get information about hash `c6f50cb47d61092240bc9e7fd6631451ddb617011ab038b42a674585668dc54a`. What is the malware family of this sample (you can use the tags to get this information)?

### 1.2.5 Exercise #2.4: Joining CLI with other tools

**Objectives**:

- Get 10 most recent Mutexes from `nanocore` configs. External tool called `jq` may be useful.

Click to see the intended solution

```
for f in $(mwdb search configs 'family:nanocore' -n 10 -o short )
        mwdb fetch $f /tmp/$f
        cat /tmp/$f | jq '.Mutex'
end
```

or

```
for f in $(mwdb search configs 'family:nanocore' -n 10 -o short )
        mwdb fetch $f - | jq '.Mutex'
end
```

## 1.3 Part 3 - Karton

### 1.3.1 Setup

This part requires Karton Playground to be set up.

```
git clone https://github.com/CERT-Polska/karton-playground.git
cd karton-playground
sudo docker-compose up  # this may take a while
```

Also take a look at the Karton documentation

**Available services**

- `127.0.0.1:8030` karton-dashboard
- `127.0.0.1:8080` mwdb-core (user: admin, password: admin)
- `127.0.0.1:8090` minio (user: mwdb, password: mwdbmwdb)

### 1.3.2 Exercise #3.1: Adding new service to the Karton pipeline

**Goal**: Learn how to connect new karton systems to your network

1. Integrate an existing karton service into your pipeline: karton-autoit-ripper

```
$ python3 -m venv venv
$ source ./venv/bin/activate
$ pip install karton-autoit-ripper

$ # playground-specific: copy local config to cwd
$ cp config/karton.ini karton.ini
$ karton-autoit-ripper
[2021-04-11 17:19:57,867][INFO] Service karton.autoit-ripper started
```

2. Download a sample, and verify its hash

```
$ wget https://github.com/CERT-Polska/training-mwdb/raw/main/autoit-malware.bin
$ sha256sum autoit-malware.bin
a4816d4fecd6d2806d5b105c3aab55f4a1eb5deb3b126f317093a4dc4aab88a1 autoit-malware.bin
```

3. Finally, upload it to your local mwdb (http://127.0.0.1:8080, admin:admin)

### 1.3.3 Exercise #3.2: Write your own service

**Goal**: Learn how to create a new karton service from ground up

1. Download a template:

https://github.com/CERT-Polska/training-mwdb/blob/main/karton-template.py

2. Edit the template, and:

- Run the `strings` utility on every incoming sample
- Save the result in a variable (use subprocess.check_output)
- Upload the result to mwdb (already handled in the template)

## 1.4 Part 4 - Malduck

### 1.4.1 Setup

```
$ python3 -m venv venv
$ source ./venv/bin/activate
$ pip install malduck
```

### 1.4.2 Exercise #4.1: Getting familiar with Malduck

**Goal**: Learn how Malduck can be used to make your malware analysis life easier

#### 1. Explore the CLI

While malduck is primarily used as a python module, it also exposes a few CLI commands. Let's check them out by running `malduck`:

```
Commands:
  extract    Extract static configuration from dumps
  fixpe      Fix dumped PE file into the correct form
  resources  Extract PE resources from an EXE into a directory
```

We'll come back to `extract` in a few minutes, let's try out `resources` and extract PE resources from `unknown_sample_07c69147626042067ef9adfa89584a4f93f8ccd24dec87dd8f291d946d465b24.bin`.

Just by looking at the unpacked resource, can you guess what type of malware this is and which family does it belong to?

#### 2. Crypto functions

Malduck implements some of the most commonly used crypto/compression functions used by malware.

See if you can use the available malduck functions to decrypt the following cryptograms:

- xor `u:p+Z{3}` with a password `3v1l!`
- RC4 hexstring `c08d5e066fb0afcc90bbc53eb592` using password `black`
- Decrypt hexstring `538cce7ccaa57d03860863207dfe345f` using AES-ECB and the key derived from md5 of `secret`

Click to see the intended solution

```python
from malduck import xor
# key comes first in args
print(xor(b"3v1l!", b"u:p+Z{3}"))

from malduck import rc4
print(rc4(key=b"black", data=bytes.fromhex("c08d5e066fb0afcc90bbc53eb592")))

from malduck import aes
from hashlib import md5

print(aes.ecb.decrypt(key=md5(b"secret").digest(), data=bytes.fromhex(
↪"538cce7ccaa57d03860863207dfe345f")))
```

### 3. Disassembly engine

We've compiled the following function to x86 (32bit). Let's see if you can get the secret integer using malducks `disasm` engine!

```
bool check_secret(unsigned int num) {
    unsigned int result = num ^ SECRET;
    return result == 0;
}
```

**Hint**: look for instructions that use the `xor` opcode.

Compiled assembly shellcode: `5589e583ec108b4508357777adde8945fc837dfc000f94c0c9c3`

Click to see the intended solution

```
from malduck import disasm

assembly = "5589e583ec108b4508357777adde8945fc837dfc000f94c0c9c3"

for c in disasm(data=bytes.fromhex(assembly), addr=0):
    if c.mnem == "xor" and c.op2.is_imm:
        value = c.op2.value
        print(f"Found the magic value: {hex(value)}")
```

## 1.4.3 Exercise #4.2: Extracting Warzone RAT C2 server info

**Goal**: Learn how to create malware extraction modules, work on some real-life samples

In this exercise we'll try to create a module that will automatically extract information about the C2 server used in WARZONE RAT/AVE MARIA

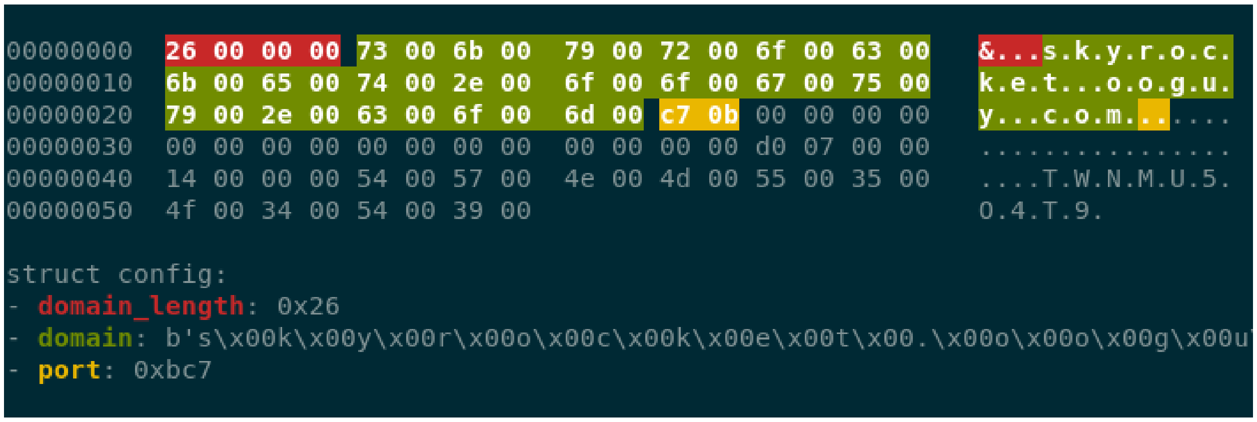
### 1. Start by downloading archive `warzone_exercise.zip` containing the files for this exercise

- `warzone_samples/*` - these are the malware samples we'll be trying to extract information from
- `modules` - a module extractor stub, you'll need to implement the missing code to make it work

### 2. Preliminary info

Because reverse-engineering malware is out of scope for this course, we'll provide you with enough information that should allow you to create the extractor module without previous knowledge.

- The configuration is stored encrypted in the `.bss` section
- RC4 is used to encrypt the config, the key starts at offset 4 and is always exactly 50 bytes
- The encrypted data is stored right after the key
- The C2 address is encoded using UTF-16 and its length is stored behind it as a little-endian 4-byte integer (see image below for an overview)



```
00000000  26 00 00 00 73 00 6b 00  79 00 72 00 6f 00 63 00   &...s.k.y.r.o.c.
00000010  6b 00 65 00 74 00 2e 00  6f 00 6f 00 67 00 75 00   k.e.t...o.o.g.u.
00000020  79 00 2e 00 63 00 6f 00  6d 00 c7 0b 00 00 00 00   y...c.o.m.......
00000030  00 00 00 00 00 00 00 00  00 00 00 00 d0 07 00 00   ................
00000040  14 00 00 00 54 00 57 00  4e 00 4d 00 55 00 35 00   ....T.W.N.M.U.5.
00000050  4f 00 34 00 54 00 39 00                            O.4.T.9.

struct config:
- domain_length: 0x26
- domain: b's\x00k\x00y\x00r\x00o\x00c\x00k\x00e\x00t\x00.\x00o\x00o\x00g\x00u
- port: 0xbc7
```

## 3. The template

Because boilerplate code can be intimidating at first, we've provided you with a stub module.

- `modules` - directory for all of your extractor modules
  - `warzone` yara rules/python files belonging to this specific module
    - `warzone.yar` - YARA rule used to detect whether the extraction code should be executed
    - `__init__.py` - The extractor code, the core object is a class that implements the `malduck.extractor.Extractor` interface

## 4. Implementing the module

This one is up to you - use the provided information to decrypt the configuration blob and extract the important information. Don't be afraid to ask us if you have any questions!

## 5. Running the module

When you're ready (or not) and you would to test the module, you can execute it using the `malduck extract` command like so:

```
malduck extract --modules modules <sample_file_or_directory>
```

## 6. Expected results

Assuming you've implemented the module correctly, here are the expected configs:

- `warzone_samples/017c61631075514428dd2183757cd18727db1bb1a4a4aa779ed4e59989f61b94`

```
{
    "c2": [
        {
            "host": "76.8.53.133",
            "port": 1198
        }
    ],
```

(continues on next page)

(continued from previous page)

```
    "family": "warzone"
}
```

- warzone_samples/29ac61c528572e6d7bfb8e5ef43ef3650da5de960fa97409fb8e4c279618eb6e

```
{
    "c2": [
        {
            "host": "skyrocket.ooguy.com",
            "port": 3015
        }
    ],
    "family": "warzone"
}
```

- warzone_samples/4562a5e32db3b136439669eb27eb13b6c27232928b92d183c5c94562281f10f7

```
{
    "c2": [
        {
            "host": "chezam.giize.com",
            "port": 3698
        }
    ],
    "family": "warzone"
}
```

### 7. Solution

Click to see the intended solution

```
class WarzoneRAT(Extractor):
    family = "warzone"
    yara_rules = ("warzone",)

    def handle_match(self, p: procmempe, match: YaraRuleMatch) -> None:
        if not type(p) is procmempe:
            self.log.warning("File is not a PE")
            return

        bss_section = p.pe.section('.bss')

        if not bss_section:
            self.log.warning("Couldn't get bss section")
            return

        data = bss_section.get_data()

        if not data:
            self.log.warning("Couldn't get data from bss")
            return
```

(continues on next page)

(continued from previous page)

```
        data = data[4:]
        decrypted = rc4(data[:50], data[50:])
        if not decrypted:
            self.log.warning("Decrypted data is kinda empty")
            return

        domain_length = uint32(decrypted[:4])
        if domain_length > 100:
            return None

        decrypted = decrypted[4:]

        c2 = decrypted[:domain_length].decode('utf-16')
        decrypted = decrypted[domain_length:]

        c2_port = uint16(decrypted[:2])

        self.push_config({
            "family": "warzone",
            "c2": [{"host": c2, "port": c2_port}],
        })
```

### 1.4.4 Exercise #4.3: Creating extraction modules from the ground up

**Goal**: Create fully fledged extraction modules

#### 1. Task

Now that you've gotten familiar with most of Malduck's functions, it's time to put them to use.

The `crackmes.zip` archive contains a bunch of ELF 64-bit binaries. Each one accepts user input from stdin and then prints whether the provided password is correct.

Your task is to create 3 extractor modules (including the YARA rules) that automatically extract the correct user input.

#### 2. Hints

- All password are in `flag{<hexdigits>}` format
- You can create the YARA rules just using the output from `strings`
- Remember to pass `x64=True` in disasm calls

### 3. Solution

View the first module

```
rule elf_simple {
    strings:
        $entry_input = "Enter the password"
        $flag_prefix = "flag{"
    condition:
        all of them
}
```

```python
from malduck import Extractor, procmem
from malduck.yara import YaraRuleMatch


class SimpleExtractor(Extractor):
    family = "simple"
    yara_rules = ("elf_simple",)

    @Extractor.string
    def flag_prefix(self, p: procmem, addr: int, matches: YaraRuleMatch):
        return {"flag": p.asciiz(addr)}
```

View the second module

```
rule elf_xor {
    strings:
        $entry_input = "What's the secret"
        $function_prologue = { 55 48 89 E5 }
    condition:
        all of them
}
```

```python
from malduck import Extractor, procmem, xor
from malduck.yara import YaraRuleMatch


class XorExtractor(Extractor):
    family = "xor"
    yara_rules = ("elf_xor",)

    @Extractor.string
    def function_prologue(self, p: procmem, addr: int,  matches: YaraRuleMatch):

        data_length = None
        movs = []

        for c in p.disasmv(addr=addr, count=64, x64=True):
            if c.mnem == "movzx" and c.op2.is_mem:
                movs.append(c.op2.value)

            if c.mnem == "cmp" and c.op2.is_imm:
```

(continues on next page)

(continued from previous page)

```
                data_length = c.op2.value
                self.log.info("Found the data length: %d", data_length)
                break

        if not data_length:
            self.log.error("Failed to find the data length")
            return None

        if len(movs) != 2:
            self.log.error("Fetched wrong amount of movs")
            return None

        data_addr, key_addr = movs
        key = p.asciiz(key_addr)

        self.log.info("Found the key: %s", key)

        data = p.readv(data_addr, data_length+1)

        return {
            "flag": xor(key, data)
        }
```

View the third module

```
rule elf_rc4 {
    strings:
        $entry_input = "Enter the magic word:"
        $function_prologue = { 55 48 89 E5 }
    condition:
        all of them
}
```

```
from malduck import Extractor, procmem, rc4
from malduck.yara import YaraRuleMatch
from typing import Optional


def try_rc4_decrypt(p: procmem, data_addr, data_length, key_addr) -> Optional[bytes]:
    key = p.asciiz(key_addr)
    data = p.readv(data_addr, data_length)

    if not key or not data:
        return None

    return rc4(key, data)


class RC4Extractor(Extractor):
    family = "rc4"
    yara_rules = ("elf_rc4",)

```

(continues on next page)

(continued from previous page)

```
    @Extractor.string
    def function_prologue(self, p: procmem, addr: int, matches: YaraRuleMatch):
        movs = []

        for c in p.disasmv(addr=addr, count=64, x64=True):
            if c.mnem == "mov" and (c.op2.is_mem or c.op2.is_imm):
                movs.append(c.op2.value)

            if c.mnem == "call":
                if len(movs) >= 3:
                    data_length, data_addr, key_addr = movs[-3:]

                    decrypted = try_rc4_decrypt(p, data_addr, data_length, key_addr)
                    if decrypted and decrypted.startswith(b"flag{"):
                        return {"flag": decrypted}

                movs = []

        return None
```

### 1.4.5 Exercise #4.4: Bonus: Integrating implemented modules into karton-config-extractor

**Goal**: See how it all can be used to automatically extract malware configuration in your karton pipeline

Now that you've created several extractor modules, you'd probably want to see them in action.

Adding your modules to the karton pipeline is pretty straight forward.

Start by installing the `karton-config-extractor`

```
pip install karton-config-extractor
```

And then run the service by pointing it to the modules folder:

```
karton-config-extractor --modules modules/
```

Upload the samples from earlier exercises and watch the new configs appear!

# CHAPTER TWO

# PREREQUISITES

Open a terminal and check if these tools are installed:

- **Python 3** (recommended 3.8 or newer)

  ```
  $ python3 -m pip
  ```

- **Git**

  ```
  $ git
  ```

- **Docker engine with Docker Compose**

  ```
  $ docker-compose
  ```

  Docker Engine installation instructions for Ubuntu (docs.docker.com)

  Docker Compose installation instructions (docs.docker.com)

Recommended environment is Ubuntu 20.04/22.04. Software used in training was not tested on other platforms (e.g. Mac OS, Windows), so prepare a fresh Ubuntu VM for the best workshop experience, especially if your host environment is unusual.

CHAPTER

THREE

# GUIDES FOR EXERCISES

- mwdb-core - Advanced search based on Lucene queries
- mwdblib - Guide for REST API and mwdblib usage
- mwdblib - API reference
- karton - documentation
- malduck - API reference