Zircolite documentation

Zircolite is a standalone tool written in Python 3 that allows you to use SIGMA rules on:

- MS Windows EVTX (EVTX, XML, and JSONL formats)
- Auditd logs
- Sysmon for Linux
- EVTXtract
- CSV and XML logs

Key Features

- **Fast Processing**: Zircolite is relatively fast and can parse large datasets in just seconds.
- **SIGMA Backend**: It is based on a SIGMA backend (SQLite) and does not use internal SIGMA-to-something conversion.
- **Advanced Log Manipulation**: It can manipulate input logs by splitting fields and applying transformations, allowing for more flexible and powerful log analysis.
- **Flexible Export**: Zircolite can export results to multiple formats using Jinja templates, including JSON, CSV, JSONL, Splunk, Elastic, Zinc, Timesketch, and more.

You can use Zircolite directly in Python or use the binaries provided in the releases.

Usage

> [!NOTE]
> If you use the packaged version of Zircolite don't forget to replace `python3 zircolite.py` in the examples by the packaged binary name.

Requirements and Installation

- Release versions are standalone, they are easier to use and deploy. Be careful, **the packager (nuitka) does not like Zircolite being run in from another directory**.
- If you have an **ARM CPU, it is strongly recommended to use the release versions**
- The repository version of Zircolite works with **Python 3.8** and above
- The repository version can run on Linux, Mac OS and Windows
- The use of evtx_dump is **optional but required by default (because it is for now much faster)**, I you do not want to use it you have to use the '–noexternal' option. The tool is provided if you clone the Zircolite repository (the official repository is here).

Installation from releases

- Get the appropriate version here

```
# DECOMPRESS
7z x zircolite_lin_amd64_glibc_2.20.0.zip
cd zircolite_lin_amd64_glibc/

# EXAMPLE RUN
git clone https://github.com/sbousseaden/EVTX-ATTACK-SAMPLES.git
./zircolite_lin_amd64_glibc.bin -e EVTX-ATTACK-SAMPLES/Execution/ \
                                -r rules/rules_windows_sysmon_pysigma.json
```

Installation from repository

Using venv on Linux/MacOS

```
# INSTALL
git clone https://github.com/wagga40/Zircolite.git
cd Zircolite
python3 -m venv .venv
source .venv/bin/activate
pip3 install -r requirements.txt

# EXAMPLE RUN
git clone https://github.com/sbousseaden/EVTX-ATTA^C-SAMPLES.git
python3 zircolite.py -e EVTX-ATTACK-SAMPLES/ -r rules/rules_windows_sysmon_pysigma.json
deactivate # Quit Python3 venv
```

Using Pdm or Poetry

```
# INSTALL
git clone https://github.com/wagga40/Zircolite.git
cd Zircolite
pdm init -n
cat requirements.txt | xargs pdm add

# EXAMPLE RUN
git clone https://github.com/sbousseaden/EVTX-ATTACK-SAMPLES.git
pdm run python3 zircolite.py -e EVTX-ATTACK-SAMPLES/ \
    -r rules/rules_windows_sysmon_pysigma.json
```

If you want to use *poetry*, just replace the "pdm" command in the above example by "poetry".

Known issues

- Sometimes `evtx_dump` hangs under MS Windows, this is not related to Zircolite. If it happens to you, usually the use of `--noexternal` solves the problem. If you can share the EVTX files on whose the blocking happened, feel free to post an issue in the evtx_dump repository.
- If you use the packaged/release version, please note that the packager (nuitka) does not like Zircolite being run in from another directory (i.e : `c:\SOMEDIR\Zircolite\Zircolite.exe -e sample.evtx -r rules.json`).

Basic usage

Help is available with `zircolite.py -h`.

Basically, the simplest way to use Zircolite is something like this:

```
python3 zircolite.py --events <LOGS> --ruleset <RULESET>
```

Where :

- `--events` is a filename or a directory containing the logs you want to analyse (`--evtx` and `-e` can be used instade of `--events`) . Zircolite support the following format : EVTX, XML, JSON (one event per line), JSON Array (one big array), EVTXTRACT, CSV, Auditd, Sysmon for Linux
- `--ruleset` is a file or directory containing the Sigma rules to use for detection. Zircolite as its own format called "Zircolite ruleset" where all the rules are in one JSON file. However, as of version *2.20.0*, Zircolite can directly use Sigma rules in YAML format (YAML file or Directory containing the YAML files)

Multiple rulesets can be specified, results can be per-ruleset or combined (with `--combine-rulesets` or `-cr`) :

```
# Example with a Zircolite ruleset and a Sigma rule. Results will be displayed per-ruleset
python3 zircolite.py --events sample.evtx --ruleset rules/rules_windows_sysmon_pysigma.json \
    --ruleset schtasks.yml
# Example with a Zircolite ruleset and a Sigma rule. Results will be displayed combined
python3 zircolite.py --events sample.evtx --ruleset rules/rules_windows_sysmon_pysigma.json \
    --ruleset schtasks.yml --combine-rulesets
```

By default :

- `--ruleset` is not mandatory but the default ruleset is `rules/rules_windows_generic_pysigma.json`
- Results are written in the `detected_events.json` in the same directory as Zircolite, you can choose a CSV formatted output with `--csv`
- There is a `zircolite.log` file that will be created in the current working directory, it can be disabled with `--nolog`
- When providing a directory for then event logs, `Zircolite` will automatically use a file extension, you can change it with `--fileext`. This option can be used with wildcards or Python Glob syntax but `*.` will automatically be added before the given parameter value : `*.<FILEEXT PARAMETER VALUE>`. For example `--fileext log` will search for `*.log` files in the given path and `--fileext log.*` will search for `*.log.*` which can be useful when handling linux log files (auditd.log.1…)

EVTX files

If your evtx files have the extension ".evtx" :

```
python3 zircolite.py --evtx <EVTX_FOLDER/EVTX_FILE> \
    --ruleset <Converted Sigma ruleset (JSON)/Directory with Sigma rules (YAML)/>
python3 zircolite.py --evtx ../Logs --ruleset rules/rules_windows_sysmon_pysigma.json
```

XML logs

`evtx_dump` or services like **VirusTotal** sometimes output logs in text format with XML logs inside.

To do that with `evtx_dump` you have to use the following command line :

```
./evtx_dump -o xml <EVTX_FILE> -f <OUTPUT_XML_FILE> --no-indent --dont-show-record-number
```

And it produces something like this (1 event per line):

```
<?xml version="1.0" encoding="utf-8"?><Event xmlns="http://schemas.microsoft.com/win/2004/08/events/eve
```

VirusTotal : if you have an enterprise account will allow you to get logs in a pretty similar format :

```
<?xml version="1.0" encoding="utf-8"?>
<Events>
<Event xmlns="http://schemas.microsoft.com/win/2004/08/events/event"><System><Provider Guid="XXXXXXX" Na
</Events>
```

Zircolite will handle both format with the following command line :

```
python3 zircolite.py --events <LOGS_FOLDER_OR_LOG_FILE>  --ruleset <RULESET> --xml
python3 zircolite.py --events  Microsoft-Windows-SysmonOperational.xml \
    --ruleset rules/rules_windows_sysmon_pysigma.json --xml
```

EVTXtract logs

Willi Ballenthin has built called EVTXtract a tool to recovers and reconstructs fragments of EVTX log files from raw binary data, including unallocated space and memory images.

Zircolite can work with the output of EVTXtract with the following command line :

```
python3 zircolite.py --events <EVTXTRACT_EXTRACTED_LOGS>  --ruleset <RULESET> --evtxtract
```

Auditd logs

```
python3 zircolite.py --events auditd.log --ruleset rules/rules_linux.json --auditd
```

> [!NOTE]
> `--events` and `--evtx` are strictly equivalent but `--events` make more sense with non-EVTX logs.

Sysmon for Linux logs

Sysmon for linux has been released in October 2021. It outputs XML in text format with one event per-line. As of version 2.6.0, **Zircolite** support of Sysmon for Linux log files. You just have to add `-S`, `--sysmon4linux`, `--sysmon-linux`, `--sysmon-linux-input` to your command line :

```
python3 zircolite.py --events sysmon.log --ruleset rules/rules_linux.json --sysmon-linux
```

> [!NOTE]
> Since the logs come from Linux, the default file extension when using `-S` case is `.log`

JSONL/NDJSON logs

JSONL/NDJSON logs have one event log per line, they look like this :

```
{"EventID": "4688", "EventRecordID": "1", ...}
{"EventID": "4688", "EventRecordID": "2", ...}
...
```

It is possible to use Zircolite directly on JSONL/NDJSON files (NXLog files) with the `-j`, `--jsonl`, `--jsononly` or `--json-input` options :

```
python3 zircolite.py --events <LOGS_FOLDER> --ruleset <RULESET> --jsonl
```

A simple use case is when you have already run Zircolite and use the `--keeptmp` option. Since it keeps all the converted EVTX in a temp directory, if you need to re-execute Zircolite, you can do it directly using this directory as the EVTX source (with `--evtx <EVTX_IN_JSON_DIRECTORY>` and `--jsononly`) and avoid to convert the EVTX again.

JSON Array / Full JSON object

Some logs will be provided in JSON format as an array :

```
[
    {"EventID": "4688", "EventRecordID": "1", ...},
    {"EventID": "4688", "EventRecordID": "2", ...},
... ]
```

To handle these logs you will need to use the `--jsonarray`, `--json-array` or `--json-array-input` options :

```
python3 zircolite.py --events <LOGS_FOLDER> --ruleset <RULESET> --json-array-input
```

CSV

It is possible to use Zircolite directly on CSV logs **if the CSV are correctly formatted**. The field names must appear on the first line :

```
EventID,EventRecordID,Computer,SubjectUserSid,...
4624,32421,xxxx.DOMAIN.local,S-1-5-18,xxxx,DOMAIN,...
...
```

To handle these logs you will need to use the `--csv-input` options (**Do not use `--csv` !**):

```
python3 zircolite.py --events <LOGS_FOLDER> --ruleset <RULESET> --csv-input
```

SQLite database files

Since everything in Zircolite is stored in a in-memory SQlite database, you can choose to save the database on disk for later use. It is possible with the option `--dbfile <db_filename>`.

```
python3 zircolite.py --evtx <EVTX_FOLDER> --ruleset <CONVERTED_SIGMA_RULES> \
    --dbfile output.db
```

If you need to re-execute Zircolite, you can do it directly using the SQLite database as the EVTX source (with `--evtx <SAVED_SQLITE_DB_PATH>` and `--dbonly`) and avoid to convert the EVTX, post-process the EVTX and insert data to database. **Using this technique can save a lot of time... But you will be unable to use the `--forwardal`option**

Rulesets / Rules

Zircolite has his own rulesets format (JSON). Default rulesets are available in the rules directory or in the Zircolite-Rules repository.

Since version 2.20.0, Zircolite can directly use native Sigma rules by converting them with pySigma. Zircolite will detect whether the provided rules are in JSON or YAML format and will automatically convert the rules in the latter case :

```
# Simple rule
python3 zircolite.py -e sample.evtx -r schtasks.yml

# Directory
python3 zircolite.py -e sample.evtx -r ./sigma/rules/windows/process_creation
```

Using multiple rules/rulesets

It is possible to use multiple rulesets by chaining or repeating with the `-r`or `--ruleset` arguments :

```
# Simple rule
python3 zircolite.py -e sample.evtx -r schtasks.yml -r ./sigma/rules/windows/process_creation
```

By default, the detection results are displayed by ruleset, it is possible to group the results with `-cr` or `--combine-rulesets`. In this case only one list will be displayed.

Pipelines

By default, Zircolite does not use any pySigma pipelines, which can be somewhat limiting. However, it is possible to use the default pySigma pipelines.

Install and list pipelines

However, they must be installed before check pySigma docs for that, but it is generaly as simple as :

- `pip3 install pysigma-pipeline-nameofpipeline`
- `poetry add pysigma-pipeline-nameofpipeline`

Installed pipelines can be listed with :

- `python3 zircolite_dev.py -pl`
- `python3 zircolite_dev.py --pipeline-list`

Use pipelines

To use pipelines, employ the -p or –pipelines arguments; multiple pipelines are supported. The usage closely mirrors that of **Sigma-cli**.

Example :

```
python3 zircolite.py -e sample.evtx -r schtasks.yml -p sysmon -p windows-logsources
```

The converted rules/rulesets can be saved by using the `-sr` or the `--save-ruleset` arguments.

> [!NOTE]
> When using multiple native Sigma rule/rulesets, you cannot differenciate pipelines. All the pipelines will be used in the conversion process.

Field mappings, field exclusions, value exclusions, field aliases and field splitting

If your logs require transformations to align with your rules, Zircolite offers several mechanisms for this purpose. You can configure these mechanisms using a file located in the config directory of the repository. Additionally, you have the option to use your own configuration by utilizing the `--config` or `-c` options.

The configuration file has the following structure :

```
{
    "exclusions" : [],
    "useless" : [],
    "mappings" :
    {
        "field_name_1": "new_field_name_1",
        "field_name_2": "new_field_name_2"
    },
    "alias":
    {
        "field_alias_1": "alias_1"
    },
    "split":
    {
        "field_name_split": {"separator":",", "equal":"="}
    }
}
```

Field mappings

Field mappings enable you to rename a field from your logs. Zircolite leverages this mechanism extensively to rename nested JSON fields. You can view all the built-in field mappings here.

For instance, to rename the “CommandLine” field in **your raw logs** to “cmdline”, you can add the following entry to the fieldMappings.json file:

```
{
    "exclusions" : [],
    "useless" : [],
    "mappings" :
    {
        "CommandLine": "cmdline"
    },
    "alias":{},
    "split": {}
}
```

Please keep in mind that as opposed to field alias, the original field name is not kept.

Field exclusions

field exclusions allow you to exclude a field. Zircolite already uses this mechanism to exclude the `xlmns` field. You can check all the builtin field exclusions here.

Value exclusions

value exclusions allow you to remove field which value is to be excluded. Zircolite already uses this mechanism to remove *null* and empty values. You can check all the builtin value exclusions here.

Field aliases

field aliases allow you to have multiple fields with different name but the same value. It is pretty similar to field mapping but you keep the original value. Field aliases can be used on original field names but also on mapped field names and splitted fields.

Let's say you have this event log in JSON format (the event has been deliberately truncated):

```
{
    "EventID": 1,
    "Provider_Name": "Microsoft-Windows-Sysmon",
    "Channel": "Microsoft-Windows-Sysmon/Operational",
    "CommandLine": "\"C:\\Windows\\System32\\WindowsPowerShell\\v1.0\\powershell.exe\"",
    "Image": "C:\\Windows\\System32\\WindowsPowerShell\\v1.0\\powershell.exe",
    "IntegrityLevel": "Medium",
}
```

Let's say you are not sure all your rules use the "CommandLine" field but you remember that some of them use the "cmdline" field. To avoid any problems you could use an alias for the "CommandLine" field like this :

```
{
    "exclusions" : [],
    "useless" : [],
    "mappings" : {},
    "alias":{
        "CommandLine": "cmdline"
    },
    "split": {}
}
```

With this configuration, the event log used to apply Sigma rules will look like this :

```
{
    "EventID": 1,
    "Provider_Name": "Microsoft-Windows-Sysmon",
    "Channel": "Microsoft-Windows-Sysmon/Operational",
    "CommandLine": "\"C:\\Windows\\System32\\WindowsPowerShell\\v1.0\\powershell.exe\"",
    "cmdline": "\"C:\\Windows\\System32\\WindowsPowerShell\\v1.0\\powershell.exe\"",
    "Image": "C:\\Windows\\System32\\WindowsPowerShell\\v1.0\\powershell.exe",
    "IntegrityLevel": "Medium",
}
```

Be careful when using aliases because the data is stored multiple times.

Field splitting

field aliases allow you to split fields that contain key,value pairs. Zircolite already uses this mechanism to handle hash/hashes fields in Sysmon logs. You can check all the builtin field splittings here. Moreover, Field aliases can be applied to splitted fields.

For example, let's say we have this Sysmon event log :

```
{
    "Hashes": "SHA1=XX,MD5=X,SHA256=XXX,IMPHASH=XXXX",
    "EventID": 1
}
```

With the following configuration, Zircolite will split the `hashes` field like this :

```
{
    "exclusions" : [],
    "useless" : [],
    "mappings" : {},
    "alias":{},
    "split": {
        "Hashes": {"separator":",", "equal":"="}
    }
}
```

The final event log used to apply Sigma rules will look like this :

```
{
    "SHA1": "x",
    "MD5": "x",
    "SHA256": "x",
    "IMPHASH": "x",
    "Hashes": "SHA1=x,MD5=x,SHA256=x,IMPHASH=x",
    "EventID": 1
}
```

Field Transforms

What Are Transforms?

Transforms in Zircolite are custom functions that manipulate the value of a specific field during the event flattening process. They allow you to:

- Format or normalize data
- Enrich events with additional computed fields
- Decode encoded data (e.g., Base64, hexadecimal)
- Extract information using regular expressions

By using transforms, you can preprocess event data to make it more suitable for detection rules and analysis.

Enabling Transforms

Transforms are configured in the config file (the default one is in `config/fieldMappings.json`) under the `"transforms"` section. To enable transforms, set the `"transforms_enabled"` flag to `true` in your configuration file:

```
{
  "transforms_enabled": true,
  "transforms": {
    // Transform definitions
```

```
  }
}
```

Configuring Transforms

Transforms are defined in the `"transforms"` section of the configuration file. Each transform is associated with a specific field and consists of several properties.

Transform Structure

A transform definition has the following structure:

- **Field Name**: The name of the field to which the transform applies.
- **Transform List**: A list of transform objects for the field.

Each transform object contains:

- **info**: A description of what the transform does.
- **type**: The type of the transform (currently only `"python"` is supported).
- **code**: The Python code that performs the transformation.
- **alias**: A boolean indicating whether the result should be stored in a new field.
- **alias_name**: The name of the new field if `alias` is `true`.
- **source_condition**: A list specifying when the transform should be applied based on the input type (e.g., `["evtx_input", "json_input"]`).
- **enabled**: A boolean indicating whether the transform is active.

Source conditions possible values

Sets `source_condition` Value
`"json_input"`
`"json_array_input"`
`"db_input"`
`"sysmon_linux_input"`
`"auditd_input"`
`"xml_input"`
`"evtxtract_input"`
`"csv_input"`
`"evtx_input"`

Example Transform Object

```
{
  "info": "Base64 decoded CommandLine",
  "type": "python",
  "code": "def transform(param):\n    # Transformation logic\n    return transformed_value",
  "alias": true,
  "alias_name": "CommandLine_b64decoded",
  "source_condition": ["evtx_input", "json_input"],
  "enabled": true
}
```

Available Fields

You can define transforms for any field present in your event data. In the configuration, transforms are keyed by the field name:

```
"transforms": {
  "CommandLine": [
    {
      // Transform object
    }
  ],
  "Payload": [
    {
      // Transform object
    }
  ]
}
```

Writing Transform Functions

Zircolite uses `RestrictedPython` to safely execute transform functions. This means that certain built-in functions and modules are available, while others are restricted. The function must be named `transform` and accept a single parameter `param`, which is the original value of the field.

Available Modules and Functions:

- **Built-in Functions**: A limited set of Python built-in functions, such as `len`, `int`, `str`, etc.
- **Modules**: You can import `re` for regular expressions, `base64` for encoding/decoding, and `chardet` for character encoding detection.

Unavailable Features:

- Access to file I/O, network, or system calls is prohibited.
- Use of certain built-in functions that can affect the system is restricted.

Example Transform Functions

Base64 Decoding

```
def transform(param):
    import base64
    decoded = base64.b64decode(param)
    return decoded.decode('utf-8')
```

Hexadecimal to ASCII Conversion

```
def transform(param):
    decoded = bytes.fromhex(param).decode('ascii')
    return decoded.replace('\x00', ' ')
```

Applying Transforms

Transforms are automatically applied during the event flattening process if:

- They are **enabled** (`"enabled": true`).
- The current input type matches the **source condition** (`"source_condition": [...]`).

For each event, Zircolite checks if any transforms are defined for the fields present in the event. If so, it executes the transform function and replaces the field's value with the transformed value or stores it in a new field if `alias` is `true`.

Example

Use Case: Convert hexadecimal-encoded command lines in Auditd logs to readable ASCII strings.

Configuration:

```
"proctitle": [
  {
    "info": "Proctitle HEX to ASCII",
    "type": "python",
    "code": "def transform(param):\n    return bytes.fromhex(param).decode('ascii').replace('\\x00', '
    "alias": false,
    "alias_name": "",
    "source_condition": ["auditd_input"],
    "enabled": true
  }
]
```

Explanation:

- **Field**: `proctitle`
- **Function**: Converts hexadecimal strings to ASCII and replaces null bytes with spaces.
- **Alias**: `false` (the original `proctitle` field is replaced).

Best Practices

- **Test Your Transforms**: Before enabling a transform, ensure that the code works correctly with sample data.
- **Use Aliases Wisely**: If you don't want to overwrite the original field, set `"alias": true` and provide an `"alias_name"`.
- **Manage Performance**: Complex transforms can impact performance. Optimize your code and only enable necessary transforms.
- **Keep Transforms Specific**: Tailor transforms to specific fields and input types using `"source_condition"` to avoid unexpected behavior.

Generate your own rulesets

Default rulesets are already provided in the `rules` directory. These rulesets only are the conversion of the rules located in rules/windows directory of the Sigma repository. These rulesets are provided to use Zircolite out-of-the-box but you should generate your own rulesets.

As of v2.9.5, Zircolite can auto-update its default rulesets using the `-U` or `--update-rules`. There is an auto-updated rulesets repository available here.

Generate rulesets using PySigma

Using Pdm or Poetry

```
# INSTALL
git clone https://github.com/SigmaHQ/sigma.git
cd sigma
pdm init -n
pdm add pysigma pip sigma-cli pysigma-pipeline-sysmon pysigma-pipeline-windows pysigma-backend-sqlite

# GENERATE RULESET (SYSMON)
pdm run sigma convert -t sqlite -f zircolite -p sysmon -p windows-logsources sigma/rules/windows/ -s -o
# GENERATE RULESET (GENERIC / NO SYSMON)
pdm run sigma convert -t sqlite -f zircolite -p windows-audit -p windows-logsources sigma/rules/windows/
```

In the last line :

- `-t` is the backend type (SQlite)
- `-f` is the format, here "zircolite" means the ruleset will be generated in the format used by Zircolite
- `-p` option is the pipeline used, in the given example we use two pipelines
- `-s` to continue on error (e.g when there are not supported rules)
- `-o` allow to specify the output file

If you want to use *poetry*, just replace the "pdm" command in the above example by "poetry".

Generate rulesets using sigmatools [DEPRECATED]

[**DEPRECATED**] Zircolite use the SIGMA rules in JSON format. Since the SQLite backend is not yet available in pySigma, you need to generate your ruleset with the official legacy-sigmatools (**version 0.21 minimum**) :

```
pip3 install sigmatools
```

[**DEPRECATED**] since you need to access the configuration files directly it is easier to also clone the repository :

```
git clone https://github.com/SigmaHQ/legacy-sigmatools.git
cd legacy-sigmatools
```

[DEPRECATED] Sysmon rulesets (when investigated endpoints have Sysmon logs)

```
sigmac \
    -t sqlite \
    -c tools/config/generic/sysmon.yml \
    -c tools/config/generic/powershell.yml \
    -c tools/config/zircolite.yml \
    -d rules/windows/ \
   --output-fields title,id,description,author,tags,level,falsepositives,filename,status \
   --output-format json \
   -r \
   -o rules_sysmon.json \
   --backend-option table=logs
```

Where :

- `-t` is the backend type (SQlite)
- `-c` options are the backend configurations from the official repository
- `-r` option is used to convert an entire directory (don't forget to remove if it is a single rule conversion)
- `-o` option is used to provide the output filename
- `--backend-option` is used to specify the SQLite table name (leave as is)

[DEPRECATED] Generic rulesets (when investigated endpoints *don't* have Sysmon logs) [DEPRECATED]

```
sigmac \
    -t sqlite \
    -c tools/config/generic/windows-audit.yml \
    -c tools/config/generic/powershell.yml \
    -c tools/config/zircolite.yml \
    -d rules/windows/ \
   --output-fields title,id,description,author,tags,level,falsepositives,filename,status \
   --output-format json \
   -r \
```

```
-o rules_generic.json \
--backend-option table=logs
```

Why you should build your own rulesets

The default rulesets provided are the conversion of the rules located in `rules/windows` directory of the Sigma repository. You should take into account that :

- **Some rules are very noisy or produce a lot of false positives** depending on your environment or the config file you used with genRules
- **Some rules can be very slow** depending on your logs

For example :

- "Suspicious Eventlog Clear or Configuration Using Wevtutil" : **very noisy** on fresh environment (labs etc.), commonly generate a lot of useless detections
- Notepad Making Network Connection : **can slow very significantly** the execution of Zircolite

Generate embedded versions

Removed

- You can use DFIR Orc to package Zircolite, check here
- Kape also has a module for Zircolite : here

Docker

Zircolite is also packaged as a Docker image (cf. wagga40/zircolite on Docker Hub), which embeds all dependencies (e.g. `evtx_dump`) and provides a platform-independant way of using the tool. Please note this image is not updated with the last rulesets !

You can pull the last image with : `docker pull wagga40/zircolite:latest`

Build and run your own image

```
docker build . -t <Image name>
docker container run --tty \
    --volume <Logs folder>:/case
    wagga40/zircolite:latest \
    --ruleset rules/rules_windows_sysmon_pysigma.json \
    --events /case \
    --outfile /case/detected_events.json
```

This will recursively find log files in the `/case` directory of the container (which is bound to the `/path/to/evtx` of the host filesystem) and write the detection events to the `/case/detected_events.json` (which finally corresponds to `/path/to/evtx/detected_events.json`). The given example uses the internal rulesets, if you want to use your own, place them in the same directory as the logs :

```
docker container run --tty \
    --volume <Logs folder>:/case
    wagga40/zircolite:latest \
    --ruleset /case/my_ruleset.json \
    --events /case/my_logs.evtx \
    --outfile /case/detected_events.json
```

Event if Zircolite does not alter the original log files, sometimes you want to make sure that nothing will write to the original files. For these cases, you can use a read-only bind mount with the following command:

```
docker run --rm --tty \
    -v <EVTX folder>:/case/input:ro \
    -v <Results folder>:/case/output \
    wagga40/zircolite:latest \
    --ruleset rules/rules_windows_sysmon_pysigma.json \
    --events /case/input \
    -o /case/output/detected_events.json
```

Docker Hub

You can use the Docker image available on Docker Hub. Please note that in this case, the configuration files and rules are the default ones.

```
docker container run --tty \
    --volume <EVTX folder>:/case docker.io/wagga40/zircolite:lastest \
    --ruleset rules/rules_windows_sysmon_pysigma.json \
    --evtx /case --outfile /case/detected_events.json
```

Advanced use

Working with large datasets

Zircolite tries to be as fast as possible so a lot of data is stored in memory. So :

- **Zircolite memory use oscillate between 2 or 3 times the size of the logs**
- It is not a good idea to use it on very big EVTX files or a large number of EVTX **as is**

There are a lot of ways to speed up Zircolite :

- Using as much CPU core as possible : see below "Using GNU Parallel"
- Using Filtering

  [!NOTE]
  There is an option to heavily limit the memory usage of Zircolite by using the `--ondiskdb <DB_NAME>` argument. This is only usefull to avoid errors when dealing with very large datasets and if you have have a lot of time… **This should be used with caution and the below alternatives are far better choices**.

Using GNU Parallel

Except when `evtx_dump` is used, Zircolite only use one core. So if you have a lot of EVTX files and their total size is big, it is recommended that you use a script to launch multiple Zircolite instances. On Linux or MacOS The easiest way is to use **GNU Parallel**.

[!NOTE]
On MacOS, please use GNU find (`brew install find` will install `gfind`)

- **"DFIR Case mode" : One directory per computer/endpoint**

  This mode is very useful when you have a case where all your evidences is stored per computer (one directory per computer containing all EVTX for this computer). It will create one result file per computer in the current directory.

  ```
  find <CASE_DIRECTORY> -maxdepth 1 -mindepth 1 -type d | \
      parallel --bar python3 zircolite.py -e {} \
      -r rules/rules_windows_sysmon_pysigma.json --outfile {/.}.json
  ```

  One downside of this mode is that if you have less computer evidences than CPU Cores, they all will not be used.

- **“WEF/WEC mode” : One zircolite instance per EVTX**

  You can use this mode when you have a lot of aggregated EVTX coming from multiple computers. It is generally the case when you use WEF/WEC and you recover the EVTX files from the collector. This mode will create one result file per EVTX.

  ```
  find <CASE_DIRECTORY> -type f -name "*.evtx" \
      parallel -j -1 --progress python3 zircolite.py -e {} \
      -r rules/rules_windows_sysmon_pysigma.json --outfile {/.}.json
  ```

  In this example the `-j -1` is for using all cores but one. You can adjust the number of used cores with this arguments.

Keep data used by Zircolite

Zircolite has a lot of arguments that can be used to keep data used to perform Sigma detections :

- `--dbfile <FILE>` allows you to export all the logs in a SQLite 3 database file. You can query the logs with SQL statements to find more things than what the Sigma rules could have found
- `--keeptmp` allows you to keep the source logs (EVTX/Auditd/Evtxtract/XML…) converted in JSON format
- `--keepflat` allow you to keep the source logs (EVTX/Auditd/Evtxtract/XML…) converted in a flattened JSON format

Filtering

Zircolite has a lot of filtering options to speed up the detection process. Don’t overlook these options because they can save you a lot of time.

File filters

Some EVTX files are not used by SIGMA rules but can become quite large (a good example is `Microsoft-Windows-SystemDataArchiver%4Diagnostic.evtx`), if you use Zircolite with a directory as input argument, all EVTX files will be converted, saved and matched against the SIGMA Rules.

To speed up the detection process, you may want to use Zircolite on files matching or not matching a specific pattern. For that you can use **filters** provided by the two command line arguments :

- `-s` or `--select` : select files partly matching the provided a string (case insensitive)
- `-a` or `--avoid` : exclude files partly matching the provided a string (case insensitive)

  [!NOTE]
  When using the two arguments, the “select” argument is always applied first and then the “avoid” argument is applied. So, it is possible to exclude files from included files but not the opposite.

- Only use EVTX files that contains “sysmon” in their names

  ```
  python3 zircolite.py --evtx logs/ --ruleset rules/rules_windows_sysmon_pysigma.json \
      --select sysmon
  ```

- Exclude “Microsoft-Windows-SystemDataArchiver%4Diagnostic.evtx”

  ```
  python3 zircolite.py --evtx logs/ --ruleset rules/rules_windows_sysmon_pysigma.json \
      --avoid systemdataarchiver
  ```

- Only use EVTX files with “operational” in their names but exclude “defender” related logs

  ```
  python3 zircolite.py --evtx logs/ --ruleset rules/rules_windows_sysmon_pysigma.json \
  --select operational --avoid defender
  ```

For example, the **Sysmon** ruleset available in the `rules` directory only use the following channels (names have been shortened) : *Sysmon, Security, System, Powershell, Defender, AppLocker, DriverFrameworks, Application, NTLM, DNS, MSexchange, WMI-activity, TaskScheduler.*

So if you use the sysmon ruleset with the following rules, it should speed up `Zircolite`execution :

```
python3 zircolite.py --evtx logs/ --ruleset rules/rules_windows_sysmon_pysigma.json \
    --select sysmon --select security.evtx --select system.evtx \
    --select application.evtx --select Windows-NTLM --select DNS \
    --select powershell --select defender --select applocker \
    --select driverframeworks --select "msexchange management" \
    --select TaskScheduler --select WMI-activity
```

Time filters

Sometimes you only want to work on a specific timerange to speed up analysis. With Zircolite, it is possible to filter on a specific timerange just by using the `--after` and `--before` and their respective shorter versions `-A` and `-B`. Please note that :

- The filter will apply on the `SystemTime` field of each event
- The `--after` and `--before` arguments can be used independently
- The timestamps provided must have the following format : YYYY-MM-DD**T**HH:MM:SS (hours are in 24h format)

Examples :

- Select all events between the 2021-06-02 22:40:00 and 2021-06-02 23:00:00 :

  ```
  python3 zircolite.py --evtx logs/ --ruleset rules/rules_windows_sysmon_pysigma.json \
      -A 2021-06-02T22:40:00 -B 2021-06-02T23:00:00
  ```

- Select all events after the 2021-06-01 12:00:00 :

  ```
  python3 zircolite.py --evtx logs/ --ruleset rules/rules_windows_sysmon_pysigma.json \
      -A 2021-06-01T12:00:00
  ```

Rule filters

Some rules can be noisy or slow on specific datasets (check here) so it is possible to skip them by using the `-R` or `--rulefilter` argument. This argument can be used multiple times.

The filter will apply on the rule title. To avoid unexpected side-effect **comparison is case-sensitive**. For example, if you do not want to use all MSHTA related rules :

```
python3 zircolite.py --evtx logs/ \
    --ruleset rules/rules_windows_sysmon_pysigma.json \
    -R MSHTA
```

Limit the number of detected events

Sometimes, SIGMA rules can be very noisy (and generate a lot of false positives) but you still want to keep them in your rulesets. It is possible to filter rules that returns too mich detected events with the option `--limit <MAX_NUMBER>`. **Please note that when using this option, the rules are not skipped the results are just ignored** but this is useful when forwarding events to Splunk.

Forwarding detected events

> [!WARNING]
> Forwarding is DEPRECATED and will likely be disabled in a future release

Zircolite provide multiple ways to forward events to a collector :

- the HTTP forwarder : this is a very simple forwarder and pretty much a **"toy"** example and should be used when you have nothing else. An **example** server called is available in the tools directory
- the Splunk HEC Forwarder : it allows to forward all detected events to a Splunk instance using **HTTP Event Collector**
- the ELK ES client : it allows to forward all detected events to an ELK instance

There are two modes to forward the events :

- By default all events are forwarded after the detection process
- The argument `--stream` allow to forward events during the detection process

If you forward your events to a central collector you can disable local logging with the Zircolite `--nolog` argument.

Forward events to a HTTP server

If you have multiple endpoints to scan, it is useful to send the detected events to a central collector. As of v1.2, Zircolite can forward detected events to an HTTP server :

```
python3 zircolite.py --evtx sample.evtx  --ruleset rules/rules_windows_sysmon_pysigma.json \
    --remote "http://address:port/uri"
```

An **example** server called is available in the tools directory.

Forward events to a Splunk instance via HEC

As of v1.3.5, Zircolite can forward detections to a Splunk instance with Splunk **HTTP Event Collector**.

1. Configure HEC on you Splunk instance : check here
2. Get your token and you are ready to go :

```
python3 zircolite.py --evtx /sample.evtx  --ruleset rules/rules_windows_sysmon_pysigma.json \
    --remote "https://x.x.x.x:8088" --token "xxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx" \
    [--index myindex]
```

Since Splunk HEC default to the first associated index, `--index` is optional but can be used to specify the choosen index among the available ones.

> [!WARNING]
> On Windows do not forget to put quotes

Forward to ELK

As of version 2.8.0, Zircolite can forward events to an ELK stack using the ES client.

```
python3 zircolite.py --evtx /sample.evtx  --ruleset rules/rules_windows_sysmon_pysigma.json \
    --remote "https://x.x.x.x:8088" --index "zircolite-whatever" \
    --eslogin "yourlogin" --espass "yourpass"
```

> [!NOTE]
> the `--eslogin` and `--espass` arguments are optional.

> [!WARNING]
> **Elastic is not handling logs the way Splunk does. Since Zircolite is flattening the field names in the JSON output some fields, especially when working with EVTX files, can have different types between Channels, logsources etc. So when Elastic uses automatic field mapping, mapping errors may prevent events insertion into Elastic.**

No local logs When you forward detected events to an server, sometimes you don't want any log file left on the system you have run Zircolite on. It is possible with the `--nolog` option.

Forwarding all events

Zircolite is able to forward all events and not just the detected events to Splunk, ELK or a custom HTTP Server. you just to use the `--forwardall` argument. Please note that this ability forward events as JSON and not specific `Windows` sourcetype.

> [!WARNING]
> **Elastic is not handling logs the way Splunk does. Since Zircolite is flattening the field names in the JSON output some fields, especially when working with EVTX files, can have different types between Channels, logsources etc. So when Elastic uses automatic field mapping, mapping errors may prevent events insertion into Elastic.**

Templating and Formatting

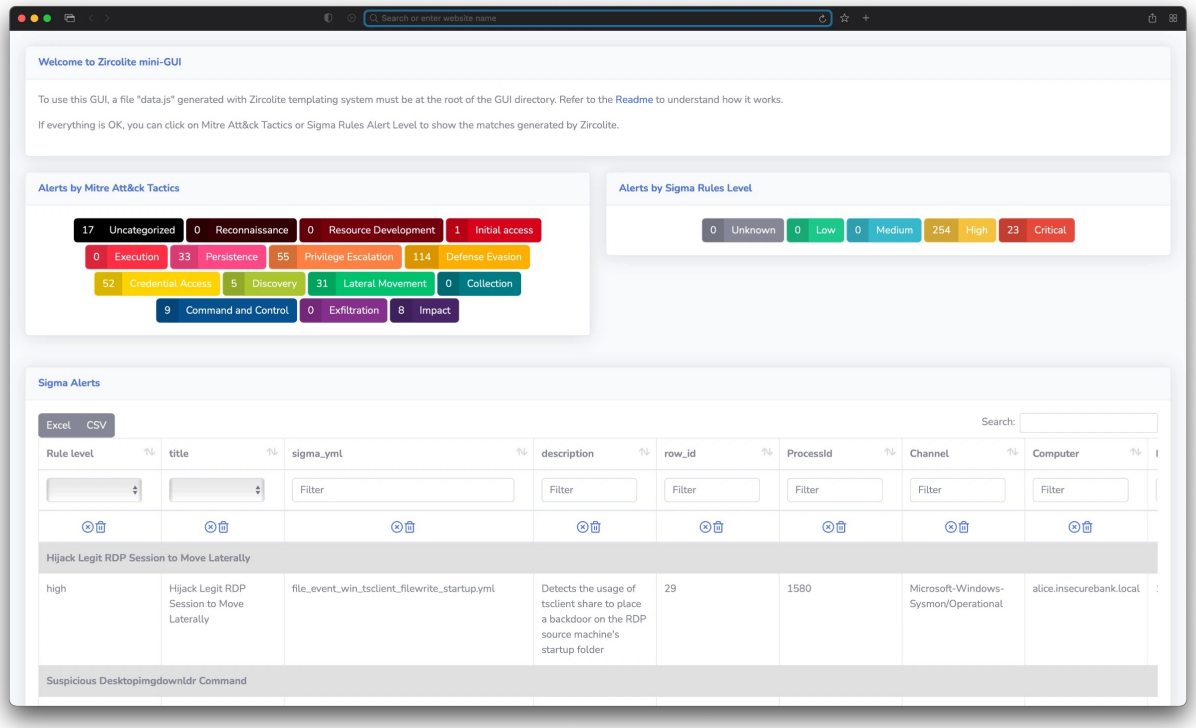
Zircolite provides a templating system based on Jinja 2. It allows you to change the output format to suits your needs (Splunk or ELK integration, Grep-able output…). There are some templates available in the Templates directory of the repository : Splunk, Timesketch, … To use the template system, use these arguments :

- `--template <template_filename>`
- `--templateOutput <output_filename>`

```
python3 zircolite.py --evtx sample.evtx  --ruleset rules/rules_windows_sysmon_pysigma.json \
--template templates/exportForSplunk.tmpl --templateOutput exportForSplunk.json
```

It is possible to use multiple templates if you provide for each `--template` argument there is a `--templateOutput` argument associated.

Mini-GUI



The Mini-GUI can be used totally offline, it allows the user to display and search results. It uses datatables and the SB Admin 2 theme.

Automatic generation

As of Zircolite 2.1.0, the easier way to use the Mini-GUI is to generate a package with the `--package` option. A zip file containing all the necessary data will be generated at the root of the repository.

Manual generation

You need to generate a `data.js` file with the `exportForZircoGui.tmpl` template, decompress the zircogui.zip file in the gui directory and replace the `data.js` file in it with yours :

```
python3 zircolite.py --evtx sample.evtx
    --ruleset rules/rules_windows_sysmon_pysigma.json \
    --template templates/exportForZircoGui.tmpl --templateOutput data.js
7z x gui/zircogui.zip
mv data.js zircogui/
```

Then you just have to open `index.html` in your favorite browser and click on a Mitre Att&ck category or an alert level.

> [!WARNING]
> **The mini-GUI was not built to handle big datasets**.

Packaging Zircolite

PyInstaller

- Install Python 3.8 on the same OS as the one you want to use Zircolite on
- Install all dependencies : `pip3 install -r requirements.txt`
- After Python 3.8 install, you will need PyInstaller : `pip3 install pyinstaller`
- In the root folder of Zircolite type : `pyinstaller -c --onefile zircolite.py`
- The `dist` folder will contain the packaged app

Nuitka

- Install Python 3.8 on the same OS as the one you want to use Zircolite on
- Install all dependencies : `pip3 install -r requirements.txt`
- After Python 3.8 install, you will need Nuitka : `pip3 install nuitka`
- In the root folder of Zircolite type : `python3 -m nuitka --onefile zircolite.py`

> [!WARNING]
> When packaging with PyInstaller or Nuitka some AV may not like your package.

Using With DFIR Orc

DFIR Orc is a Forensics artefact collection tool for systems running Microsoft Windows (pretty much like Kape or artifactcollector). For more detailed explanation, please check here : https://dfir-orc.github.io.

ZikyHD has done a pretty good job at integrating **Zircolite** with **DFIR Orc** in this repository : https://github.com/Zircocorp/dfir-orc-config.

Basically, if you want to integrate Zircolite with **DFIR Orc** :

- Clone the DFIR Orc Config repository : `git clone https://github.com/Zircocorp/dfir-orc-config.git`
- Create a `DFIR-ORC_config.xml` (or add to an existing one) in the `config` directory containing :

```
<?xml version="1.0" encoding="utf-8"?>
<wolf childdebug="no" command_timeout="1200">
    <log disposition="truncate">DFIR-ORC_{SystemType}_{FullComputerName}_{TimeStamp}.log</log>
    <outline disposition="truncate">DFIR-ORC_{SystemType}_{FullComputerName}_{TimeStamp}.json
    </outline>
```

```
    <!-- BEGIN ZIRCOLITE SPECIFIC CONFIGURATION-->
    <!-- This part creates a specific archive for Zircolite -->
    <archive name="DFIR-ORC_{SystemType}_{FullComputerName}_Zircolite.7z"
             keyword="Zircolite" concurrency="1"
             repeat="Once"
             compression="fast"
             archive_timeout="120" >
        <restrictions ElapsedTimeLimit="480" />
        <command keyword="GetZircoliteSysmon" winver="6.2+">
            <execute name="zircolite_win10_nuitka.exe"
                     run="7z:#Tools|zircolite_win10_nuitka.exe"/>
            <input  name='rules_windows_generic.json'
                    source='res:#rules_windows_generic.json'
                    argument='-r {FileName}' />
            <input  name='fieldMappings.json'
                    source='res:#fieldMappings.json'
                    argument='-c {FileName}' />
            <argument> --cores 1 --noexternal -e C:\windows\System32\winevt\Logs</argument>
            <output  name="detected_events.json" source="File" argument="-o {FileName}" />
            <output  name="zircolite.log" source="File" argument="-l {FileName}" />
        </command>
    </archive>
    <!-- /END ZIRCOLITE SPECIFIC CONFIGURATION-->

</wolf>
```

[!NOTE]
Please note that if you add this configuration to an existing one, you only need to keep the part between `<!-- BEGIN ... -->` and `<!-- /END ... -->` blocks.

- Put your custom or default mapping file `zircolite_win10_nuitka.exe` (the default one is in the Zircolite repository `config` directory) `rules_windows_generic.json` (the default one is in the Zircolite repository `rules` directory) in the the `config` directory.
- Put **Zircolite** binary (in this example `zircolite_win10_nuitka.exe`) and **DFIR Orc** binaries (x86 and x64) in the the `tools` directory.
- Create a `DFIR-ORC_Embed.xml` (or add to an existing one) in the `config` directory containing :

```
<?xml version="1.0" encoding="utf-8"?>
<toolembed>
    <input>.\tools\DFIR-Orc_x86.exe</input>
    <output>.\output\%ORC_OUTPUT%</output>

    <run64 args="WolfLauncher" >7z:#Tools|DFIR-Orc_x64.exe</run64>
    <run32 args="WolfLauncher" >self:#</run32>

    <file name="WOLFLAUNCHER_CONFIG"
          path=".\%ORC_CONFIG_FOLDER%\DFIR-ORC_config.xml"/>

    <!-- BEGIN ZIRCOLITE SPECIFIC CONFIGURATION-->
    <file name="rules_windows_generic.json"
          path=".\%ORC_CONFIG_FOLDER%\rules_windows_generic.json" />
    <file name="fieldMappings.json"
          path=".\%ORC_CONFIG_FOLDER%\fieldMappings.json" />
```

```
    <!-- /END ZIRCOLITE SPECIFIC CONFIGURATION-->

    <archive name="Tools" format="7z" compression="Ultra">
        <file name="DFIR-Orc_x64.exe"
              path=".\tools\DFIR-Orc_x64.exe"/>

        <!-- BEGIN ZIRCOLITE SPECIFIC CONFIGURATION-->
        <file name="zircolite_win10_nuitka.exe"
              path=".\tools\zircolite_win10_nuitka.exe"/>
        <!-- /END ZIRCOLITE SPECIFIC CONFIGURATION-->

    </archive>
</toolembed>
```

> [!NOTE]
> Please note that if you add this configuration to an existing one, you only need to keep the part between `<!-- BEGIN ... -->` and `<!-- /END ... -->` blocks.

- Now you need to generate the **DFIR Orc** binary by executing `.\configure.ps1` at the root of the repository
- The final output will be in the `output` directory

Other tools

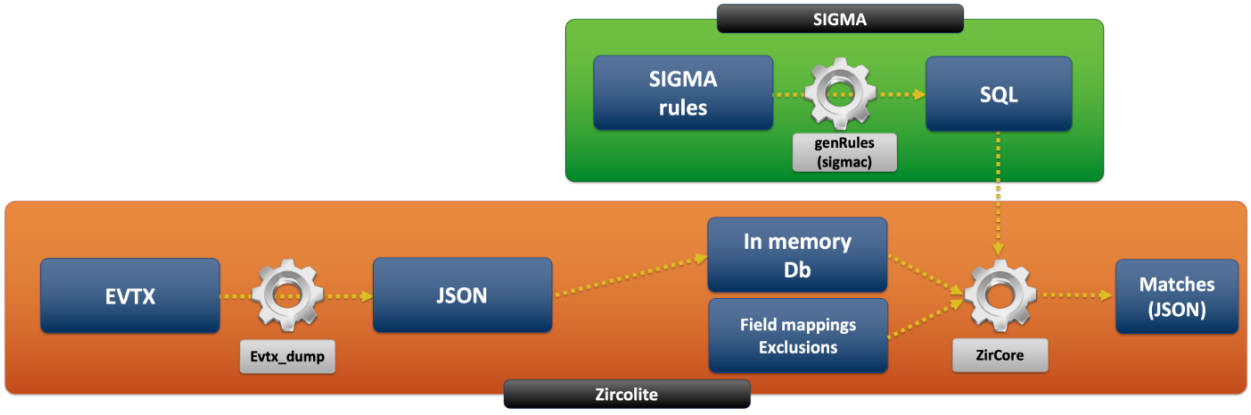
Some other tools (mostly untested) have included a way to run Zircolite :

- Kape has a module for Zircolite : here
- Velociraptor has an artifact for Zircolite : here

Internals

Zircolite architecture

Zircolite is more a workflow than a real detection engine. To put it simply, it leverages the ability of the sigma converter to output rules in SQLite format. Zircolite simply applies SQLite-converted rules to EVTX stored in an in-memory SQLite DB.



Project structure

```
Makefile                # Very basic Makefile
Readme.md               # Do I need to explain ?
bin                     # Directory containing all external binaries (evtx_dump)
```

```
config                  # Directory containing the config files
docs                    # Directory containing the documentation
pics                    # Pictures directory - not really relevant
rules                   # Sample rules you can use
templates               # Jinja2 templates
tools                   # Directory containing all tools (genRules, zircolite_server)
zircolite.py            # Zircolite !
```