ATLAS

Release 0.1

r00tten

Jun 09, 2023

CONTENTS

According to Merriam-Webster:

atlas noun

at·las | at-ls

1. capitalized : a Titan who for his part in the Titans' revolt against the gods is forced by Zeus to support the heavens on his shoulders

3. a : a bound collection of maps often including illustrations, informative tables, or textual matter

—

ATLAS is an analysis description of malware or kill-chain. Malware is a combination of techniques crafted for a purpose. With an ATLAS rule, these techniques and capabilities are like LEGO pieces. In this way, it tries to help malware researchers to focus single piece at a time and nothing more. ATLAS interpretation of the rule does the rest. It also removes the language boundaries. Different pieces can be written in other script languages.

If these techniques are transformed into LEGO pieces properly, it eventually creates a memory. Then, the total time to write an ATLAS rule will decrease.

```
meta:
name: "rtf_template_injection"
description: "A rule to extracts rtf template injection"
reference: "https://www.proofpoint.com/us/blog/threat-insight/injection-new-black-novel-
↪rtf-template-inject-technique-poised-widespread"
version: "1.0"

scripts:
# import re
# import base64

# def run(data: bytes) -> str:
#     result = ''

#     encoded_template_pattr = "XHtcXFwqXFx0ZW1wbGF0ZVxzKyguKylccypcfQ=="
#     result = re.search(base64.b64decode(encoded_template_pattr), data).group(1).
↪decode()

#     return result
s1:
↪"aW1wb3J0IHJlCmltcG9ydCBiYXNlNjQKIApkZWYgcnVuKGRhdGE6IGJ5dGVzKSAtPiBzdHI6CiAgICByZXN1bHQgPSAnJwoKICAg
↪"

chain:
file_read:
   input: $param.file
   func: file_read_bin

template_extract:
   input:
      - $scripts.s1
      - $file_read
   func: python_executor

download:
   input: $template_extract
```

(continues on next page)

(continued from previous page)

```
   func: download_from_remote_server

save_template:
   input:
      - $download
      - "template_"
   func: save_file_bytes
```

When the above rule is processed by ATLAS:

- It reads the file according to command-line argument,
- Then runs the python script that is defined in **scripts** section,
- Tries to downloads the template from the matched pattern,
- And saves the downloaded data to the disk.

Note: This project is under active development.

CHAPTER

ONE

CONTENTS

1.1 Installation

Install using Python's PIP:

```
pip install malware-atlas
```

Clone directly from Github:

```
git clone https://github.com/malware-atlas/atlas
```

1.1.1 Test Run

The HelloWorld rule can be used to test the installation.

```
atlas -a HelloWorld.atl
```

1.2 Rule

ATLAS rules or atl files follow YAML syntax. It has at most 4 keys: **meta**, **modules**, **scripts** and **chain**.

```
meta:
    name: "Hello World"
    description: "Traditional way to start."
    version: "1.0"

chain:
    printer_subchain:
        input: 'Hello World, P.S. ATLAS.'
        func: printer
```

Note: **chain** is the only section that a valid rule must contain.

1.3 scripts

```
scripts:
    <script_name>: <base64_encoded_script>
```

Scripts are distinct atomic functionalities of an ATLAS rule, like *functions* of the functional programming languages.

Note: Currently Python and Powershell are supported. Javascript support is in the backlog.

1.3.1 Key-Value

A key can be an arbitrary keyword or the function name of the entry point in the script. First, it tries to call the **key**. Then, if it fails, it calls the **run**.

Note: The value must be base64 encoded.

```
scripts:
    # def printer(*args) -> bool:

    #     print(" ".join(args))
    #     return True
    s1:
↪"ZGVmIHByaW50ZXIoKmFyZ3MpIC0+IGJvb2w6CgogICAgcHJpbnQoIiAiLmpvaW4oYXJncykpCiAgICByZXR1cm4gVHJ1ZQ==
↪"
```

```
import base64

with open('script.py', 'rb') as file:
    data = file.read()

encoded_script = base64.b64encode(data)
```

```
# It is important to give -Encoding as UTF8 instead of directly getting Byte.
# The byte option appends byte order marker at the beginning thus ruins everthing
$content = Get-Content .\script.ps1 -Raw -Encoding UTF8
[System.Convert]::ToBase64String([System.Text.Encoding]::UTF8.GetBytes($content))
```

1.3.2 Python's execution

python_executor function inside the core library is used. The function gets the script as base64 encoded, the script's name and arguments to pass to its entry point.

- It does in-memory import for the script.
- Calls the entry point.
- Gets the return data.

1.3.3 Null Bytes

If the script contains null bytes, they must be encoded in the script:

Listing 1: Problematic script example

```
def run(data: bytes) -> bool:
    try:
        keyIL = re.search(b'(?<=((\x72)[\x00-\xff]{4}(\x80)[\x00-\xff]{4}(\x72)))[\x00-\
↪xff]{4}', data).group()
        su = re.search(b'[\x00-\xff]{4}(?=([\x00-\xff]{4}(\x23\x55\x53\x00)))', data).
↪group()
    except:
        return False

    return True
```

When ATLAS tries to execute a rule that contains the above function as a script, the base64 module will raise an exception due to null bytes.

Listing 2: Null byte solution

```
def run(data: bytes) -> bool:
    try:
        keyIL_encoded = b'KD88PSgocilbAC3/XXs0fSiAKVsALf9dezR9KHIpKSlbAC3/XXs0fQ=='
        su_encoded = b'WwAt/117NH0oPz0oWwAt/117NH0oI1VTACkpKQ=='
        keyIL = re.search(b64decode(keyIL_encoded), data).group()
        su = re.search(b64decode(su_encoded), data).group()
    except:
        return False
```

1.3.4 Powershell's Execution

powershell_executor function inside the core library is used. The function gets the script as base64 encoded, the script's name and arguments to pass to its entry point.

- Creates a Powershell process.
- **Inside this process;**
  - Decodes the script and creates a ScriptBlock.
  - Prepares the arguments.
  - Calls the entry point.
  - Base64 encodes the return data.
  - Creates TemporaryFile, prints the path, and writes encoded data to the file.
  - Stores the encoded data to the file.
- In Python, it reads the file and decodes it.

Note: Right now **str** and **bytes** type arguments are supported for powershell execution.

1.4 chain

```
chain:
    <sub-chain_name>:
        input: <input>
        func: <function_name_from_core>
```

This section is a description of the rule's execution. **chain** consists of one or more sub-chains. It is like a linked list.

```
chain:
    file_read:
        input: <path>
        func: file_read_bin
    save_file:
        input: $file_read
        func: save_file_bytes
```

Note: For the above example, **file_read** and **save_file** are sub-chains.

1.4.1 Sub-chain

A sub-chain can be thought like a function. It takes inputs, processes them, and gives output.

The keyword that defines a sub-chain is just arbitrary, like variable names.

Note: Sub-chain key mustn't be one of the special keys like **chain**, **scripts**, etc.

Sub-chains are executed one by one unless it catches an exception or fails to expect satisfaction if there is any. So it is safe to assume that there is AND relation between same-level sub-chains.

When a step isn't singular, and there are a bunch of variations, OR block might be a solution. It is a sub-chain that has inner sub-chains instead of func details. In this way, the execution continues one by one until the expectation is satisfied. If there is none, then the execution is stopped immediately.

Note: It is a must to define expect key for OR blocks.

To create OR relation, it should be defined inner-subchains:

Listing 3: remcos.atl

```
meta:
    name: "Remcos"
    description: "A rule to process Remcos png files"
    reference: "https://r00tten.com/in-depth-analysis-attack-vector-triggered-by-risk/"
    version: "1.0"

scripts:
    png_extract:
↪"ZnJvbSB0eXBpbmcgaW1wb3J0IERpY3QsIExpc3QKCmRlZiBydW4oZGF0YTogYnl0ZXMpIC0+IExpc3RbYW55XToKICAgIGFyck0g
```

(continues on next page)

(continued from previous page)

```
↪"
    find_signature:
↪"aW1wb3J0IHJlCmRlZiBkb3RuZXRfbm9uZV9zdHJ1Y3RfZmluZFNpZ25hdHVyZUFkZHJfMShkYXRhOiBieXRlcykgLT4gYnl0ZXM6
↪"
    get_key:
↪"aW1wb3J0IHJlCgpmcm9tIFBJTCBpbXBvcnQgSW1hZ2UKZnJvbSBiYXNlNjQgaW1wb3J0IGI2NGRlY29kZQoKCmRlZiBydW4oZGF0
↪"
    1pxHeightImageProcess:
↪"aW1wb3J0IGlvCmltcG9ydCBoYXNobGliCmZyb20gUElMIGltcG9ydCBJbWFnZQoKCmRlZiBydW4oYXJyKToKICAgIGRhdGEgPSBb
↪"
    1000pxImageProcess:
↪"aW1wb3J0IGlvCmltcG9ydCBoYXNobGliCmZyb20gUElMIGltcG9ydCBJbWFnZQoKCmRlZiBydW4oYXJyKToKICAgIGRhdGEgPSBb
↪"
    gzipImageProcess:
↪"aW1wb3J0IGlvCmltcG9ydCBnemlwCmltcG9ydCBoYXNobGliCmZyb20gUElMIGltcG9ydCBJbWFnZQoKCmRlZiBydW4oYXJyLCBr
↪"
    run:
↪"ZnVuY3Rpb24gcnVuKCRpbWFnZV9kYXRhLCAka2V5KSB7DQogICAgQWRkLUNvbnRlbnQgYzpcXFVzZXJzXFxKb2huXFxEZXNrdG9w
↪"


chain:
    file_read:
        input: <path>
        func: file_read_bin

    pngExtract:
        input:
            - $scripts.png_extract
            - $file_read
        func: python_executor

    key:
        input:
            - $scripts.get_key
            - $file_read
        func: python_executor

    pngProcess:
        expect: is_pe

        1pxHeight:
            input:
                - $scripts.1pxHeightImageProcess
                - $pngExtract
            func: python_executor

        1000px:
            input:
                - $scripts.1000pxImageProcess
                - $pngExtract
            func: python_executor
```

(continues on next page)

(continued from previous page)

```
        gzip:
            input:
                - $scripts.gzipImageProcess
                - $pngExtract
                - $key
            func: python_executor

        powershell_executor:
            input:
                - $scripts.run
                - $file_read
                - <key>
            func: powershell_executor

    save_file2:
        input: $pngProcess
        func: save_file_bytes
```

For the example above, the **pngProcess** sub-chain is OR block. **expect: is_pe** is the top-level key for the sub-chain. It applies to all inner sub-chains. So repeated keys can be combined in this way.

During the execution of the **pngProcess** it starts from the first inner sub-chain, which is **1pxHeight**, and continues until the expectation is satisfied.

In the OR block, the satisfying output assigns to the top-level sub-chain key rather than the inner sub-chain to solve the ambiguity. **save_file2** sub_chain references OR block's output through **$pngProcess**.

1.4.2 input

A sub-chain could have zero or more inputs. An input's value might be static like a string. But it is possible to reference dynamic content through the $ prefix:

- Previous sub-chains outputs. Like the above example, **file_read**'s output is **pngExtract**'s input.
- A script,

```
meta:
    name: "rtf_template_injection"
    description: "A rule to extracts rtf template injection"
    version: "1.0"

scripts:
    # import re
    # import base64

    # def run(data: bytes) -> str:
    #     result = ''

    #     encoded_template_pattr = "XHtcXFwqXFx0ZW1wbGF0ZVxzKyguKylccypcfQ=="
    #     try:
    #         result = re.search(base64.b64decode(encoded_template_pattr), data).
↪group(1).decode()
```

(continues on next page)

(continued from previous page)

```
    #     except Exception as e:
    #         print(e)
    #         return result

    #     return result
    s1:
↪"aW1wb3J0IHJlCmltcG9ydCBiYXNlNjQKIApkZWYgcnVuKGRhdGE6IGJ5dGVzKSAtPiBzdHI6CiAgICByZXN1bHQgPSAnJwoKICAg
↪"

chain:
    file_read:
        input: <path>
        func: file_read_bin

    template_extract:
        input:
            - $scripts.s1
            - $file_read
        func: python_executor

    print_template:
        input: $template_extract
        func: printer
```

- Key-value from command-line arguments.

```
meta:
    name: "Param"
    description: "Test ATLAS rule for param"

chain:
    printer:
        input: $param.printer
        func: printer
```

1.4.3 func

This key holds the function name to call from the core library for the execution.

Note: **func** is the only key that a valid sub-chain must contain.

1.4.4 expect

This key hold the function name to call from the expect library after the execution for validation. The corresponding sub-chain output is passed to the function. The execution is stopped if the output doesn't satisfy the expected value.

```
meta:
    name: "is_pe"
    description: "A rule to validate whether the input file's reverse is peexe or not."
    version: "1.0"

chain:
    file_read:
        input: <path>
        func: file_read_bin

    pe_validation:
        input: $file_read
        func: reverse
        expect: is_pe

    save_file:
        input: $pe_validation
        func: save_file_bytes
```

1.5 modules

```
modules:
    - <path_of_the_module_relative_to_process>
```

At its very best, ATLAS is a solution to the lack of memory. Because of that, it follows modular design paradigms.

A rule can reference other rules and re-used its components. This way, it is possible to create library-like rules for centralized analysis memory.

modules is a list, and each value references other modules by a path. After defining them, they can be used in the chain section:

Listing 4: decryption_lib.atl

```
scripts:
s1:
↪"ZGVmIHJ1bihkYXRhOiBzdHIsIHhvcl9rZXk6IHN0cikgLT4gc3RyOgogICAgcmVzdWx0ID0gIiIKICAgIGtleSA9IGludCh4b3Jf
↪"

chain:
    xor_decrpytion:
        input:
            - $scripts.s1
            - '0x30'
        func: python_executor
```

Listing 5: xor.atl

```
meta:
    name: "Decrytion rule"
    description: "A rule to decrypt."
    version: "1.0"

modules:
    - ATLAS/decryption_lib

chain:
    file_read:
        input: $param.file
        func: file_read_bin

    decryption_routine: $decryption_lib.chain.xor_decryption

    printer:
        input: $decryption_routine
        func: printer
```

Note: Right now module's path is relative to the process, but this will change in the future to be relative to the main rule's path.

1.6 meta

```
meta:
 <key>: <value>
```

This section contains metadata of the rule. This is similar to YARA's meta section.

```
meta:
 name: "Simple"
 description: "Simple description to describe the simple rule."
 version: "1.0"

 chain:
 printer:
     input: 'Simple.'
     func: printer
```

1.7 Core Library

The library that does the actual execution.

reverse(...)

Returns the reverse of the argument.

Parameters
data (*any*) – The data to be reversed.

Returns
Reversed data.

download_from_remote_server(...)

Downloads from the server that is passed as a argument.

Parameters
addr (*str*) – Address of the server.

Returns
Bytes object of the response.

powershell_executor(...)

Executes the powershell script that is passed as a argument.

Parameters

- **script_name** (*tuple*) – Tuple object, script's name and base64 encoded content. It is enough to pass script key.
- ***args** (*any*) – Arguments to pass corresponding script.

Returns
Return data of the execution.

file_read_bin(...)

Performs binary file read operation.

Parameters
path (*str*) – File's path.

Returns
Bytes object of the file's content.

file_read_utf8(...)

Performs utf8 file read operation.

Parameters
path (*str*) – File's path.

Returns
The file's content.

save_file_bytes(...)

Performs binary file write operation.

Parameters

- **data** (*any*) – Content to save.
- **prefix** (*str*) – File's name prefix.

Returns
Bool, condition of the operation.

`save_file_arr`(...)

Performs file write one by one according to the list type argument.

Parameters

- **arr** (*list*) – List of contents to save.
- **prefix** (*str*) – File's name prefix. Default value is 'output'.

Returns
Bool, condition of the operation.

`python_executor`(...)

Executes the python script that is passed as an argument.

Parameters

- **script_name** (*tuple*) – Tuple object, script's name and base64 encoded content. It is enough to pass script key.
- ***args** (*any*) – Arguments to pass corresponding script.

Returns
Return data of the execution.

`printer`(...)

Prints the arguments by joining them.

Parameters
***args** (*any*) – Strings to print.

Returns
Bool, condition of the operation.

`hello_world`(...)

Prints "Hello World, ATLAS." string. Can be used as a test.

Returns
None.

`bytes_to_str_utf8`(...)

UTF8 decodes byte object.

Parameters
data (*bytes*) – Bytes data.

Returns
UTF8 string.

`get_sha256`(...)

Calculates sha256 checksum.

Parameters
data (*bytes*) – Bytes data.

Returns
sha256 checksum.

1.8 Expect Library

The library that is used for output validation.

is_not_none(...)

Validates whether the argument is None or not.

is_not_empty_list(...)

Validates whether the list type argument is not empty.

is_not_empty_dict(...)

Validates whether the dict type argument is not empty.

is_not_empty_str(...)

Validates whether the str type argument is not empty.

is_pe(...)

Validates whether the argument is 'application/x-dosexec' or not.

INDEX