Pcraft Handbook
a companion to Pcraft

Sebastien Tricaud <sebastien.tricaud@devo.com>
Lead Developer of Pcraft
http://www.github.com/devoinc/pcraft

October 27, 2021

Abstract

Those are my principles, and if you don't like them... well, I have others.

—Groucho Marx

Pcraft originally meant "Packet Crafter", but now does much more! It is available from `http://www.github.com/devoinc/pcraft`. Pcraft will help you archive three things:

1. Define Attack Scenarios (using the AMI language)
2. Create a Packet Capture (pcap) from an Attack Scenario
3. Write Logs for Simulated Devices

Figure 1: **Document Status**

Date	Author	Description
July 19, 2021	Sebastien Tricaud	First version

Contents

1 Introduction

This Handbook will walk the reader through all the capability Pcraft gives.

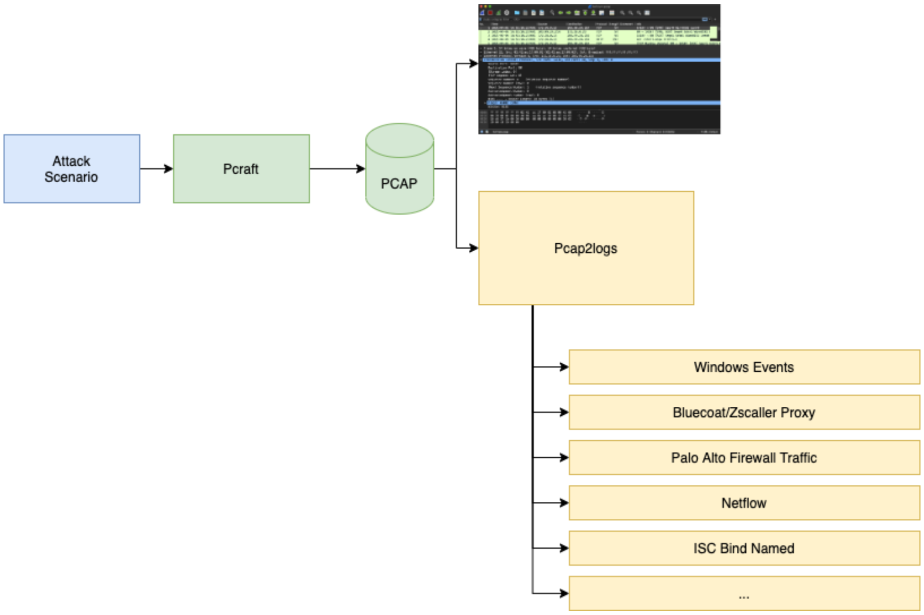
1.1 Pcraft overview

Pcraft generates consistent logs across a variety of devices/applications. To achieve that, it uses a language called AMI, which sole purpose is to **describe** what is expected. This is called a scenario as it contains the script that will drive Pcraft through the various steps it requires to complete the job.

The advantage of this scenario is its reading simplicity.

Once the scenario is written, Pcraft needs to create a database that can be either in Pcap or in Avro. We will get into details of those two ways later in the handbook. From the database output, it is read to create logs for the supported devices/applications.

An overview of the various steps using the Pcap for the database can be summarized in the following graph:



1.2 Creating DNS bind logs

Writing a DNS connection requires using the plugin **DNSConnection** inside an action block. Each action done is wrapped into an action block. There can be as many action block as required. Also the **av 1** in the header is mandatory, it helps to version the actual language specification so the AMI language can evolve in the future while preserving backward compatibility. **av** is short for **ami_version** which can also be used instead.

```
av 1

action dns {
  $domain = "example.com"
  exec DNSConnection
}
```

Listing 1: AMI file to create DNS bind logs

This action generates three files:

File Name	Description
bind_query.log	DNS Bind query log
netflow_v9.log	Netflow v9 log
paloalto_firewall.log	Palo Alto firewall log

The file *bind_query.log* contains the following:

```
0,14-Jul-2021 17:42:02.000000 queries: info: client
    ↪ 192.168.191.70#31928: query: example.com A IN +
    ↪ (1.1.1.1)
```

Listing 2: bind_query.log output

Which is exactly what the **Bind Named daemon** would have written. In this case however, the source IP making the request, as well as the resolver have not been defined in the AMI scenario, which made Pcraft choosing for default values.

2 Pcraft Tools

Pcraft is actually the project name for several tools. This section covers what they are so you can decide what is best.

2.1 The Packet Capture world

The original project was simply a pcap writer to then extract logs for it. If we take the DNS query AMI example seen in the introduction section, it is contained in a file called *dns.ami*; To generate a pcap from this file, the **pcrafter** program can be run:

```
$ /pcrafter dns.ami dns.pcap
Opening Script File dns.ami
Scenariofile: [dns.ami]
All plugins loaded!
Final Sleep Cursor: 0 seconds; 0 hours; 0 days
0 writing errors
```

Listing 3: Writing a pcap

We now have *dns.pcap* which contains the request and reply to it, we can use any tool that can read a pcap, as it is a standardized format. For example with tshark:

```
$ tshark -r dns.pcap
    1   0.000000 192.168.133.32 ? 1.1.1.1      DNS 71 Standard query 0x0000 A example.com
    2   0.000000      1.1.1.1 ? 192.168.133.32 DNS 98 Standard query response 0x0000 A example.com A 10.140.81.215
```

Listing 4: Reading dns.pcap using tshark

The next tool we want to use is **pcap2logs** which will write logs from simulated devices and application using this pcap as input. Note pcap2logs works with **any pcap**, including those which have not been generated using **pcrafter**. Here is how **pcap2logs** will generate logs:

```
$ ./pcap2log.py dns.pcap dns
Loaded 21 plugins
{'http': ['bluecoat-proxysg-main', 'zscaler-access'], 'ntp': ['corelight-ntp'], 'no-active-layer': ['fidelis', 'hbss-agent', 'hbss-audit', 'hbss-epo', 'hbss-intrusion', 'mcafee-edr', 'mcafee-emailgateway', 'mswin-o365', 'mswin-powershell', 'mswin-security', 'mswin-sysmon', 'paloalto-threat', 'tanium'], 'dns': ['named'], 'ip': ['netflow', 'paloalto-firewall'], 'snmp': ['snmptrapd'], 'syslog': ['syslog']}
Layers that did not match a plugin: 4
Writer process done, now rewriting from config

$ ls dns/
bind_query.log  netflow_v9.log  paloalto_firewall.log
```

Listing 5: Generating logs from a pcap

To use **pcap2logs** the first parameter is the **pcap file** and the second the **output directory** where those logs must be written.

2.2 The Avro database

The pcap file is convenient, however limited by its writing speed simply because a pcap must be crafted with some extra data and consistency across each frame. While this speed does not matter to write a large amount of devices interracting with each other, Pcraft was designed to simulate real infrastructure with hundred of thousand devices. Hence Pcap will not be suitable for such a scale at this time.

The Avro database is simply the output of the data run from the AMI scenario file, to be used later to write logs. In order to write an Avro database and not a pcap, another tool must be used: **ccraft**.

```
  $ ./ccraft/ccraft dns.ami dns.db
  Start Time: None
```

Listing 6: Creating an Avro database

Notice the speed at which this is written. It is much faster than **pcrafter**.

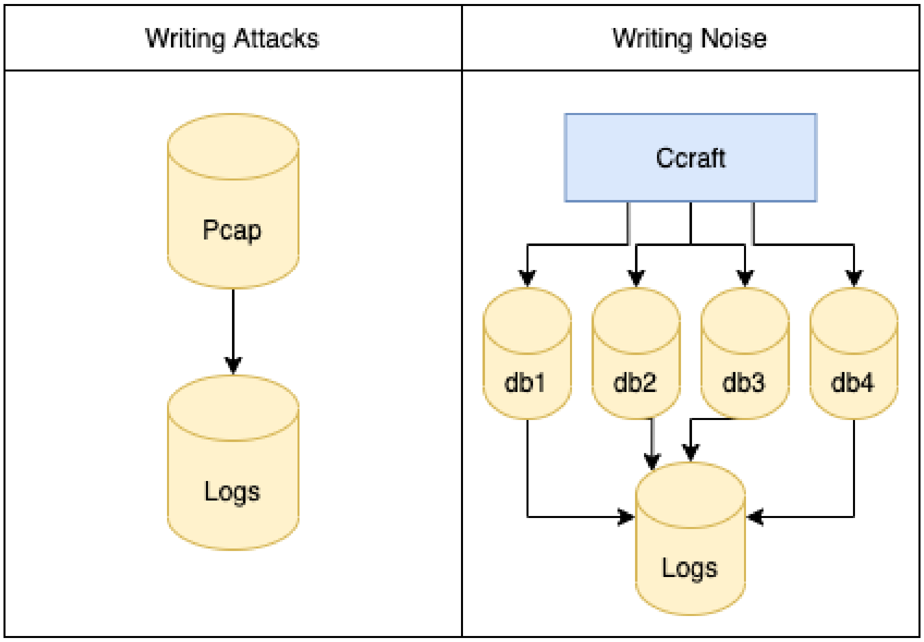
Now the database is generated, it must be used to create logs from it. The tool used is called **db2logs**, as it translates this database into the logs using the same plugins as before.

```
./db2logs.py dns.db dns -f
Loaded 21 plugins
{'http': ['bluecoat-proxysg-main', 'zscaler-access'], 'ntp':
    ↪  ['corelight-ntp'], 'no-active-layer': ['fidelis', '
    ↪ hbss-agent', 'hbss-audit', 'hbss-epo', 'hbss-intrusion
    ↪ ', 'mcafee-edr', 'mcafee-emailgateway', 'mswin-o365',
    ↪ 'mswin-powershell', 'mswin-security', 'mswin-sysmon',
    ↪ 'paloalto-threat', 'tanium'], 'dns': ['named'], 'ip':
    ↪ ['netflow', 'paloalto-firewall'], 'snmp': ['snmptrapd
    ↪ '], 'syslog': ['syslog']}
```

Listing 7: Genering logs from an Avro database

2.3 List of Pcraft tools

Tool Name	**Description**	**Example use**
pcrafter	Generate a pcap from an AMI scenario	./pcrafter dns.ami dns.pcap
ccraft	Generate an Avro database from an AMI scenario	./ccraft/ccraft dns.ami dns.db
pcap2logs	Generate logs from a pcap	./pcap2logs.py dns.pcap dns -f
db2logs	Generate logs from an Avro database	./db2logs.py dns.db dns -f



3 Ami

Ami is the language which powers Pcraft to generate data correctly. It offers a way to describe actions, repeat loops, and how long does one sleep until the next action. The reason for Ami to have been designed was looking at the requirements of what generating those data required. From running actions both in sequence and parallel, tied to a time sequence with variables that can be defined in either a given action, or from a called action execution.

To understand how Ami works, it is best to start with the famous ”Hello, World!” example:

```
1  av 1
2
3  action hello {
4    exec Void
5  }
```

Listing 8: Hello world in Ami

Ami runs action, there must be at least one action block to execute something. In this case, **line 4** of the hello world example, we execute nothing (**Void**), which actually just dumps to the standard output the variables defined at this stage.

Line 3 is the declaration of the action block, which takes a name that you want.

On **line 1**, this is a required header which defines the Ami Version, which must be 1 for now, but will evolve in the future to include more features and keep backward compatibility.

We can run this example using the **pcrafter** tool like this:

```
./pcrafter hello.ami hello.pcap
Opening Script File hello.ami
Scenariofile: [hello.ami]
All plugins loaded!
{'__amifile__': 'hello.ami'}
Final Sleep Cursor: 0 seconds; 0 hours; 0 days
0 writing errors
```

Listing 9: Running Hello world

As you can see there is a defined variable, called **__amifile__** which is an internal variable displaying the actual file name being parsed.

3.1 Header keywords

It is mendatory to start the Ami scenario using the **ami_version** or **av** keyword, the others are optional but can strengthen the quality of the scenario, such as:

- **reference**: To set a reference string towards information used to build that scenario. We can add as many as necessary
- **revision**: An integer to manually increment when work has been done on the file
- **author**: To reference the file Author
- **description**: A description about the scenario
- **shortdesc**: A short description about the scenario
- **tag**: A tag you may want to apply (similar to revision, there can be as many tags as needed
- **start_time**: To force a time start (in epoch)

3.2 Variables

In Ami, a variable is by default globally defined and accessible by the following actions. Which allows any execution plugin to create new ones to be used later. This is because any action can consume a previously defined variable to work. A Variable is declared with a **word** prepended with the **$** dollar sign, such as:

$$\boxed{\$hello = ”world”} \tag{1}$$

would set the string ”world” to the **hello** variable.

If we take the previous example but adding this variable, we can see it printed now:

```
  av 1

  action hello {
    $hello = "world"
    exec Void
  }
```

Listing 10: Hello world with a variable

```
$ ./pcrafter hello.ami hello.pcap
Opening Script File hello.ami
Scenariofile: [hello.ami]
All plugins loaded!
{'__amifile__': 'hello.ami', 'hello': 'world'}
Final Sleep Cursor: 0 seconds; 0 hours; 0 days
0 writing errors
```

Listing 11: Running our example

We can see here the **world** string set to the **hello** variable.

If we run two consecutives actions, the **hello** variable would still be visible by the other block:

```
  av 1

  action hello {
    $hello = "world"
    exec Void
  }

  action second {
    exec Void
  }
```

Listing 12: Hello world with a variable and two action blocks

```
1 All plugins loaded!
2 {'__amifile__': 'hello.ami', 'hello': 'world'}
3 {'__amifile__': 'hello.ami', 'hello': 'world'}
4 Final Sleep Cursor: 0 seconds; 0 hours; 0 days
```

Listing 13: Printing variables for each action

Line 3 prints the variables for the action block **hello** and **line 4** prints them for the block **second**. They both see the **hello** variable.

To stop a variable from being carried to the next action block, one can use the keyword **delete**, such as:

```
av 1

action hello {
  $hello = "world"
  exec Void
}

delete $hello

action second {
  exec Void
}
```

Listing 14: Deleting our variable

A variable declared as **local** is also not carried outside the action:

```
av 1

action hello {
  local $hello = "world"
  exec Void
}

action second {
  exec Void
}
```

Listing 15: Local variable

I is also possible to declare a variable with spaces, as long as it is surrounder by double-quotes, such as:

```
$"my variable" = "yo"
$"another variable here" = "world"
```

Listing 16: Variables with space

3.3 Functions

A Function can define the value to be set for a variable. The actual functions have been designed to help generating data faster, we can add as many as we see fit.

3.3.1 Usage

Let us start with an example, where we assign to a variable the base64 encoding of ”Hello, world!”:

```
  av 1

  action hello {
    $mystr = base64.encode("Hello, world!")
    exec Void
  }
```

Listing 17: Encoding a string

```
$ ./pcrafter hello.ami hello.pcap
Opening Script File hello.ami
Scenariofile: [hello.ami]
All plugins loaded!
{'__amifile__': 'hello.ami', 'mystr': 'SGVsbG8sIHdvcmxkIQ=='}
Final Sleep Cursor: 0 seconds; 0 hours; 0 days
0 writing errors
```

Listing 18: Seeing our encoded string

Functions can be recusively called, for example, if we want to get a random string of 10 characters, we can use the function **random.string(10)**, and to encode in base64 this string, we can write:

$$\boxed{base64.encode(random.string(10))} \tag{2}$$

3.3.2 Available Functions

Function Name	Description	Example
base64.encode	Encode a string in base64	*base64.encode(”test”)*
base64url.encode	Encode a string in base64 URL	*base64url.encode(”test”)*
printvars	Print the variables stack	*printvars()*
add	Add two numbers	*add(1, 2)*
random.macaddr	Create a random MAC address	*random.macaddr()*
sin	Returns the sin from radians	*sin(10)*
ip.gethostbyname	Returns the first IP resolved from the given host	*ip.gethostbyname(”example.com”)*
ip.cidr	Returns the wanted IP from a CIDR	*ip.cidr(”10.0.0.0/8”, 14, false)*
crypto.md5	Returns the MD5 of the given string	*crypto.md5(”test”)*
crypto.sha1	Returns the SHA1 of the given string	*crypto.sha1(”test”)*
crypto.sha256	Returns the SHA256 of the given string	*crypto.sha256(”test”)*
crypto.rc4	Returns the RC4 of the given string	*crypto.rc4(”test”)*
string.upper	Make the string uppercase	*string.upper(”TesT”)*
string.lower	Make the string lowercase	*string.lower(”TesT”)*
hostname_generator	Generate a consistent hostname from an IP	*hostname_generator(”10.0.2.3”)*
file.amidir	Get the current Ami file directory	*file.amidir(”foo.ami”)*
file.readall	Get a string from a file content	*file.readall(”photo.jpg”)*
file.linescount	Counts the number of lines for a give	*file.linescount(”readme.1st”)*
uuid.v5	Create a UUIDv5	*uuid.v5(”My Agent”)*
uuid.v4	Create a UUIDv4 (no arguments)	*uuid.v4(””)*
random.int	Get a random integer within bounds	*random.int(1, 10)*
random.float	Get a random float within bounds	*random.float(0.01, 0.95)*
random.string	Get a random string	*random.string(10)*
random.hexstring	Get a random hexadecimal string	*random.hexstring(10)*
csv	Read a CSV file	*csv(file.amidir(”file.csv”), line=2, field=”domain”, has_header=true)*
csv.linescount	Count the lines of a CSV (no header count)	*csv.linescount(”file.csv”)*

3.4 Strings

A string is usually given using the double-quote char (”), however there are circumstances where this character needs to be in a string too, to make a string still readable, we use the triple double-quote char ((”””)) and also allows line returns, such as:

```
$mystr = """This string has double quotes such as " or "
but also has two lines
returns"""
```

Listing 19: Verbatim String

This triple double-quote enclosed string is called a **verbatim string**, where anything in between is ignored by the parser. Allowing to include special characters without any problem.

One can also include the string from a variable into a string. To do this, the variable must be called enclosed with the braces characters **{** and **}**. Such as:

$mystr = ”${hello} world”

which would aquire the content from the variable **hello**.

However in the **verbatim string** since anything between the three double-quote characters is ignored, we must tell to look for replacement by prefixing the **s** character to that string, such as:

$mystr = s”””a verbatim ${hello} world”””

3.5 Loops

Ami handles loop to repeat a set of defined actions the wanted number of times. To repeat one action **n** number of times, we can use the keyword **repeat** with the variable that must be defined to hold the repeat counter.

```
repeat 10 as $index {
  action hello {
    exec Void
  }
  action world {
    exec Void
  }
}
```

Listing 20: Repeat loop

This would assign the **$index** variable to the actual repeat count, so it could be use to print the repeat loop count like this:

```
repeat 10 as $index {
  message "Repeat counter: ${index}"
  ...
}
```

or would be used to iterate through the values from a CSV, such as:

```
repeat 10 as $index {
  $field = csv("file.csv", $index , "username", true)
}
```

3.6 Time

3.6.1 Sleep

Time is a concept, because we are simulating the data creation. To understand how time works in Pcraft, we need to understand that when an action occurs and we need a delay before the second one, say 10 seconds, we can write something like this:

```
action first {
  ...
}

sleep 10

action second {
  ...
}
```

Listing 21: Time Introduction

When we write the pcap, we could either:

- Write the first action, sleep for 10 seconds and write the second action
- Look at how long we need to sleep, write the first action at the time, and write the last action at the actual running time

We have decided to take the second approach, so data can be written without having to wait for the wanted time, and make sure the last action is always the current execution time.

It is possible to sleep a random amount of time by using the **random.int** or **random.float** function, such as

sleep random.int(1, 20)

3.6.2 start_time

Timing is generated at the time the program is being run. So if it is Feb 23rd 2021 10:13pm UTC (1614118380 in epoch), the packets and logs will be written at this exact date and time... unless defined otherwise.

One way to define this is to add "start_time" in the AMI file header.

```
av 1
start_time 1614118380
```

Listing 22: Setting start time

3.6.3 Group sleep

Group sleeping is how Pcraft would define a way to sleep without intefering the global sleep cursor. When we execute the **sleep** function, what happens under the hood is an automatic assignment to the **_global** cursor. A regular **sleep** is a **group sleep**, it is a shortcut for now writing this:

sleep group ”_global” 10

Each action will then inherit from that global sleep cursor. In order to have a way to write multiple consecutive actions in the same time frame, while still sleeping between different actions, we must use the **group sleep** feature.

To use the **group sleep**, we must define it, and then use it for each action, such as:

```
av 1

sleep group "custom" 10

action first {
  exec Void
}

action second {
  sleep fromgroup "custom"
  exec Void
}
```

Listing 23: Group sleep

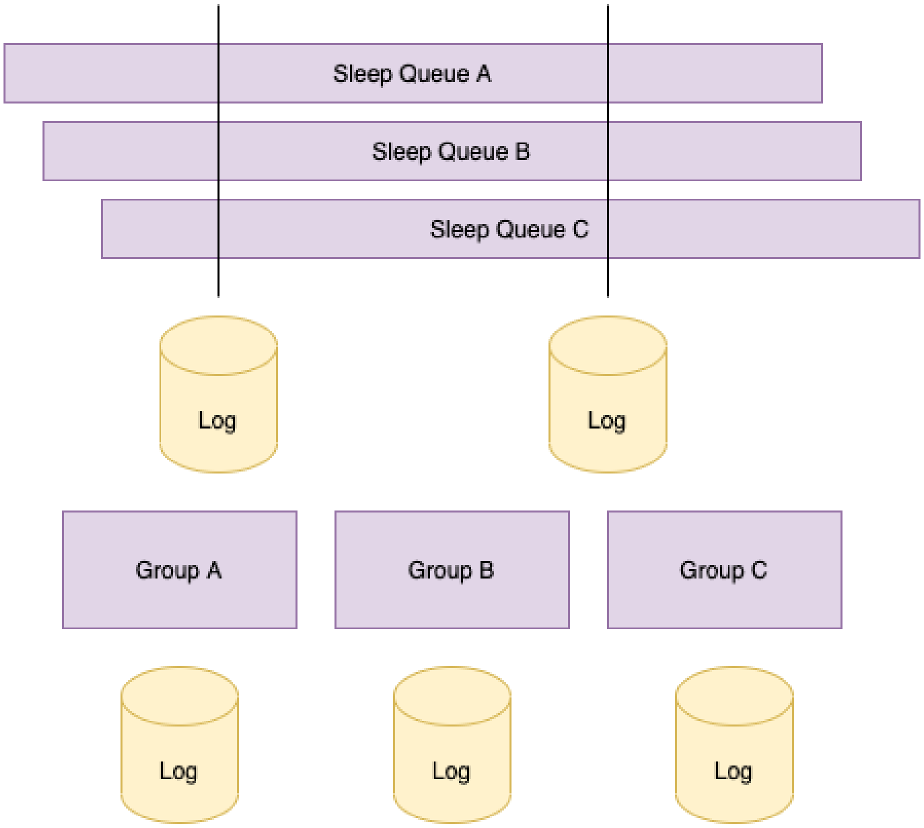
Because we explicitely told the second action to sleep from the group **custom**, it will add the the global sleep count the time set with this group sleep. Because the first action did not inherit from this sleep group, it will have no effect.

Now, an easy way to paralellize sleep groups is to use it with a **repeat** loop.

```
av 1

repeat 10 as $index {
  sleep group "custom${index}" random.int(1,3)

  action first {
    sleep fromgroup "custom${index}"
    exec Void
  }

  action second {
    sleep fromgroup "custom${index}"
    exec Void
  }
}
```

Listing 24: Group sleep in a loop

Using the string ”custom${index}” would use the $index number to create a group, thus creating 10 groups that would not overlap each-other and keep a random sleep pattern.



3.7 Sets and substitutions

Ami can define a **set**, which could influence the value a field must have, by defining it explicitely, using the **field** keyword.

field[”fieldname”] = ”value”

It is also possible to define a field to look for and assign a value to replace it with, such as:

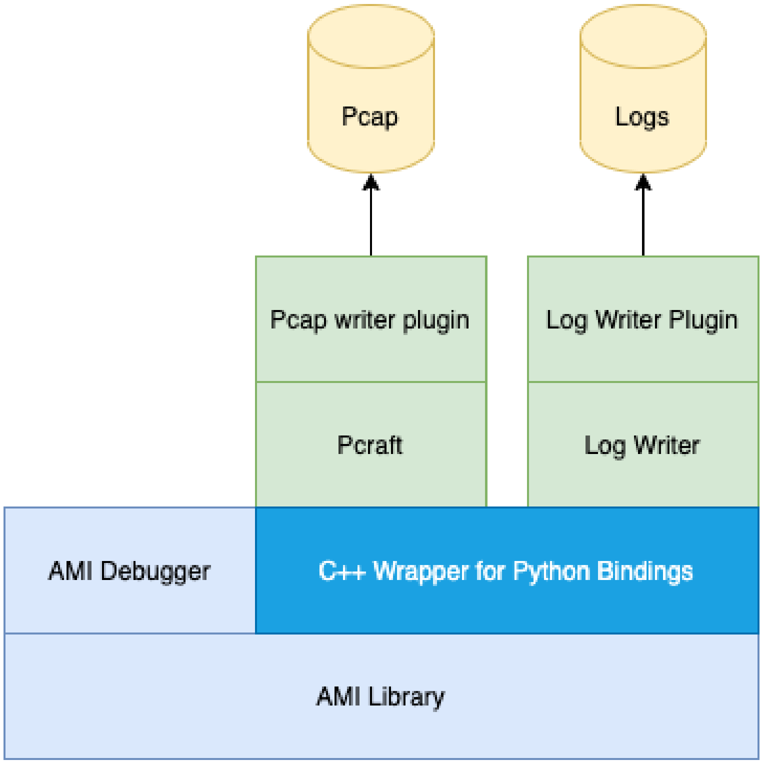
field[”ip”].replace(”10.0.0.1” => ”172.16.0.23”, ”10.2.3.4” => ”172.16.0.254”)

This is the syntax Pcraft will interpret as:

- Search for the value ”10.0.0.1” in the field ”ip” and replace it with ”172.16.0.23”
- Search for the value ”10.2.3.4” in the field ”ip” and replace it with ”172.16.0.254”

4 Pcraft Project

Pcraft is built around the Ami language which defines the scenario and offers tools, such as ”amidebug” to debug a given scenario and break step by step to read its content.



Pcraft is actually a C library which powers the Python one. Pcraft plugins are written in Python. Plugins offer the Packet Crafting capability, as well as the log template logic. This section will go into details on what those plugins can deliver.

4.1 Pcraft plugins for Packet Writing

Until now, by following this documentation you only wrote a DNS request and reply (**DNSConnection**) and used the **Void** plugin. Each plugin is called from using the **exec** keyword in an action.

```
1  action useplugin {
2    exec PluginName
3  }
```

Listing 25: Packet Writing plugins

In the previous listing, **line 2**, the **PluginName** can be one of those: Controller, DNSConnection, FakeNames, HTTPConnection, PCAPImport, Ping, Suricata, TcpRst, TcpSynAck or Void.

4.1.1 Controller

The **Controller** plugin is used to create specific data that is not Network Data, such as Endpoint. It will not create a typical TCP/IP request and reply. To operate, this plugin must have the **$log_plugin** variable set. This variable will then call the proper log writer plugin.

Also, data created by this plugin must be encoded into fields sets.

Example

```
action user_logon {
  $log_plugin = "Controller"

  field["event_id"] = "4624"

  field["winlog_event_data_IpPort"] = random.int(10000, 60000)
  field["winlog_event_data_SubjectUserName"] = $username
  field["winlog_event_data_LogonGuid"] = uuid.v5($username )

  exec Controller
}
```

Listing 26: Controller plugin writing Windows User Login

4.1.2 DNSConnection

The **DNSConnection** plugin write a DNS request and reply. Those variables can be used: **$ip-src**, **$ip-dst**, **$protocol**, **$resolver**, **$port-src**, **$port-dst**, **$domain**.

4.1.3 FakeNames

The **FakeNames** plugin creates a fake user and defines four variables: **$firstname**, **$lastname**, **$name** and **$domain**.

If the **$orgdomain** variable is not defined, it will defaults to **yoda.com**.

4.1.4 HTTPConnection

The **HTTPConnection** plugin create an HTTP request and reply. The variables it can use as input are: **$ip-src**, **$ip-dst**, **$protocol**, **$domain**, **$port-src**, **$port-dst**, **$ssl**, **$method**, **$user**, **$user-agent**, **$uri**, **$resp-httpver**, **$resp-code**, **$resp-server**, **$resp-content-type**, **$resp-content**, **$client-headers**, **$client-content**.

4.1.5 PcapImport

This plugin will import a Pcap into the flow, and requires the **$filename** variable to be set. Then with the field replacement, it can replace the IP addresses defined.

Also, it is possible to set the **$onlyreplace** variable to **"true"** to force the import of only the IP addresses that are being replaced, discarding the other.

Example

```
  action pcapin {
    $filename = "myfile.pcap"
    $onlyreplace = "true"

    field["ip"].replace("10.0.0.1" => $client-ip)

    exec PcapImport
  }
```

Listing 27: Importing a Pcap file

4.1.6 Ping

Given an **$ip-src** and **$ip-dst**, it will generate an ICMP echo request and reply.

4.1.7 Suricata

Use a Suricata rule as input to create the wanted pcap that would trigger an alert. This feature works only on a subset of alerts and is highly experimental.

Example

```
action SuricataRule {
       exec Suricata
       $EXTERNAL_NET = "192.168.0.55"
       $HTTP_SERVERS = "141.193.213.20"
       $ip-dst = "141.193.213.20"
       $domain = "grayhat.co"
       $rule = """alert http $EXTERNAL_NET any ->
           ↪ $HTTP_SERVERS any (msg:"ET WEB_SERVER Possible
           ↪ Custom Content Type Manager WP Backdoor Access
           ↪ "; flow:established,to_server;http.uri; content
           ↪ :"/plugins/custom-content-type-manager/auto-
```

```
            ↪ update.php"; fast_pattern; nocase; reference:
            ↪ url,blog.sucuri.net/2016/03/when-wordpress-
            ↪ plugin-goes-bad.html; classtype:trojan-activity
            ↪ ; sid:2022596; rev:4; metadata:created_at 2016
            ↪ _03_06, updated_at 2020_06_24;)"""
}
```

Listing 28: Suricata crafting

4.1.8 TcpRst

Create an exchange where **$ip-src** sends a TCP packet with the SYN flag towards **$ip-dst** and receives a TCP with the RST—ACK flags.

4.1.9 TcpSynAck

Create an exchange where **$ip-src** sends a TCP packet with the SYN flag towards **$ip-dst** and receives a TCP with the SYN—ACK flags. Then **$ip-src** returns an ACK. This is a typical TCP/IP three way handshake.

4.1.10 Void

Debugging plugin, it will dump of the available variables at the time of its execution.

5 Building Scenarios tips

A typical scenario stores required information in a CSV, such as users, etc. and iterates through it to write the wanted event. It would use the group sleep feature to have one sleep pattern per user or machine one want data simulated from and the heavy use of loops help to fetch a given field to set the appropriate consistent value.

Do not forget to use the **delete** keyword to remove unwanted variables as soon as one does not need it.

The **message** function is a great way to debug thing you may want to understand on the fly, such as the $index number or a variable content.

We can use the keyword **exit** at anytime, which would exit the parsing of the file when encoutered, allowing to debug the scenario.

The **skip_repeat** keyword is also useful to run everyone once, thus avoiding getting into repeat loops for the sake of debugging.