The Metabrik Platform: Rapid Development of Reusable Security Tools

@PatriceAuffret

@Metabrik
https://www.metabrik.org/
GomoR@metabrik.org

Abstract

During the course of a penetration test, a malware analysis, a forensic analysis or a *Capture The Flag* contest, who has never been in the situation of having to develop something? Who has already thought of having a small set of reusable library of tools? Or that current tools all have a different syntax and that it was a shame and never given-up on using it because its syntax was too complex?

The Metabrik Platform [1] goal is to normalize how we use tools and how they interoperate or are bound together by providing a *Platform* in the form of a *UNIX shell*, merged with a *Perl Read-Eval-Loop-Print* (*REPL*) interpreter and a virtually infinite number of *Brik*s: small reusable components that just do one task in the good way.

This document presents the design and usage of this *Platform* along with some real-life examples on how it helps to solve problems and prototype more complex *Metatool*s. You will also learn that starting with it is as simple as knowing the five *Command*s.

1 Introduction

This paper proposes to introduce you to all the concepts and ideas behind the platform. *Metabrik* is a platform and not a framework because it comes with a complete environment and a set of methods, terminologies, a new way of working. We tried to bring the concept from a well-known Danish brand that sells toys to tools development.

1.1 Concept of the platform

Metabrik is a platform giving you *Brik*s to play with. You can bind them together -or glue them together- through a *UNIX shell*. The idea is to have a set of common bricks, each having its own purpose or goal, and connect them to create a new tool or a new *Brik*. The more you write *Brik*s, the more complex tasks you will be able to achieve. For instance, many *Brik*s were written from already existing ones and *Metabrik* itself is written using *Brik*s. We will cover *forensic::scalpel* [9] *Brik* as an example of reusing *Brik*s and improving already existing tools. We will then show how we solved a root-me challenge by using a few *Brik*s.

1.2 Problem with off-the-shelf tools

There are many off-the-shelf tools (opensource or not) that helps perform daily tasks. Sometimes they are very specialized, sometimes more generic. But in all cases, you want something that is missing. If they are closed source, you can't improve them, if they are opensource, you have to master the language they are written in. The common point between all of these tools -we consider only command line tools here- is that they need input and generate output. That's all we need to make them cooperate in the direction we want. We tried to bring the concept of scenario based development by *Brikifying* tools.

Current tools usually have a complex syntax and they all have a different one. We want to normalize their usage so you don't have to learn yet another syntax and more importantly to remember from one day to another how it works.

We will go with Virtual Machine (*VM*) instrumentation for extracting Indicators Of Compromise (*IOC*s) from malwares as an example. In some other examples, we will start a *Tor* server and redirect all traffic to through it, or we will also show how we can add some information to a traceroute command like adding geolocation of every hop's IP address.

1.3 Shell-style design

As command-line lovers, we designed *Metabrik* as a *UNIX shell* from the ground-up. Those old C or Bourne shells are good, but we needed something more powerful like a good programming language that has good performance and good memory handling. And as *Perl* lovers, the answer was easy to find.

We also wanted the tool to be as easy as possible to install as all the dependencies it may have. Of course, as it relies on many external tools, you don't want to bother with that and how to configure them. The *server::tor* along with *network::linux::iptables* *Brik*s will be the perfect example of how to hide configuration directives to let you focus on your goal, on the scenario you want to achieve and not to waste time on actually searching how to use *yet-another-tool*.

1.4 Related work

The concept is very similar to that of *REbus* [2] presented in 2015 at a French security conference [3]. Funnily, we started to work on our concept in mid-2014 and were not aware of this project at that time. *REbus* first public commit dates back from beginning of 2014. Anyhow, we will see that the approach we took is a bit different.

The major differences between *REbus* and our tool is we used a shell approach to let the human interactively use *Brik*s as *REbus* prefers to automate interactions between tools. Also, *REbus* suffers from the same drawbacks as tools we integrate within *Metabrik*: a complex syntax difficult to remember from one day to another.

2 *The Metabrik Shell*

2.1 A classic *UNIX shell*

Some old *UNIX shell* concepts are kept and some new one are introduced. We have mainly kept the concept of builtin commands and aliases. As time goes by, we will also implement pipes and redirections or job management. Regarding pipes and redirections, you will see we already have those functions implemented in a different way thanks to automatic *Variable*s.

2.2 Data sculpture concept

The major new concept is: *run once, sculpt the output*. This is a shell you can use to literally sculpt *Variable*s. Because the shell is capable of *Variable*s introspection by default, you see the content of the data you need to accomplish your goal. You then use basic *Perl* functions to cut, delete, slice, grep or whatever to shape data you have in front of your eyes.

When you use a *Brik Command*, a global *Variable* is set at the end to save its output. The *Variable* is named *$RUN*. This is how you are then able to sculpt the data. Enough is enough, you need some example right now, see the example after this paragraph. We decided to go with the concept of sculpting output data to give as input to the next *Brik* because you rarely know what you will need in advance, and this is the same for any program. You can try to automate plugging of *Brik*s with normalization -that's what we actually do-, but we prefer an approach where the user decides what he needs. We can't automate every interaction between tools. If you do so, you restrict the freedom of the user and you go back to forcing him to code.

```
Meta:~> $data
{
  ip => {
    hostname => "legion.localdomain",
    ip => "192.168.0.10",
    mac => "00:11:22:33:44:55",
  },
}
Meta:~> my $mac = $data->{ip}{mac}
"00:11:22:33:44:55"
Meta:~> $mac
"00:11:22:33:44:55"
```

Python developers are already familiar with such an introspection of data or shell behavior, but don't forget we are within a *UNIX shell* so we can also use classic builtins like changing the current working directory to read files hosted there. For example, we want to put all files in current directory within a *Variable* to play with as in this example:

```
Meta:~> mkdir test
1
Meta:~> cd test/
Meta:~/test> touch file1.txt file2.txt
1
Meta:~/test> run shell::command \
  capture ls *txt
["file1.txt", "file2.txt"]
Meta:~/test> $RUN
["file1.txt", "file2.txt"]
Meta:~/test> $RUN->[1]
"file2.txt"
Meta:~/test> my $file2 = $RUN->[1]
"file2.txt"
```

2.3 Automatic *Variable*s and completion

We just introduced the basic way of using *The Metabrik Shell*, but what is more important is how we use those *Brik*s. We have just seen the *shell::command Brik* which allows us to capture the output of any external command, something really useful.

You see that we don't need to redirect output in any way because it is automatically saved into the *$RUN Variable*. You can then use it to sculpt the data and keep only what you need or add information to this data. We will speak about *lookup::* Brik*s later on, as we show usage with the *lookup::alexa Brik*. Note that instead of using a long command, we could have used an *alias* instead like:

```
Meta:~> alias ls 'run shell::command \
  capture ls'
Meta:~> cd test/
Meta:~/test> ls *txt
["file1.txt", "file2.txt", "file3.txt"]
```

The shell has some other features like completion for *Variable*s, files and directories, external commands (by searching in the PATH environment variable) and for *Metabrik Command*s (of course). You then have specific completion for *Brik Command*s. In fact, the completion is done through *GNU's libreadline*, which also gives access to all your preferred keyboard shortcuts. You can also write *Perl* code in multi-line mode, something really great for prototyping or simply testing one-liners.

As a shell, there is also a special initialization file called *.metabrik_rc* where you can *use*, *set Attribute*s or *alias* any *Brik*. Since it is read by a *Perl* interpreter, you can execute whatever *Perl* code you want. We use it to set global *Variable*s too like the path to one or more custom *Repository* [5].

A PAGER environment variable is also used so when output is written to the shell, it is automatically piped through *less* command by default. You don't need to call an external command a second time anymore, at least if you called it with the *capture Command* from *shell::command Brik*.

2.4 A *Metasploit*-like syntax

Regarding the syntax to use *Brik*s, we wanted something already known to most of us: *Metasploit* [4]. So we used the following *Command*s: *use*, *set*, *get*, *run* and *help*. We can see these commands as builtins, as opposed to external commands. But more on that later on. You got it, the name *Metabrik* has some roots from *Metasploit* for that reason.

2.5 Glueing tools input/output with *Variable*s

The glue between *Brik*s are *Variable*s. They are used as input and output to feed and create data. A *Brik Command* outputs a *Variable* you can play with like it is something you want to sculpt. There is no automation on how you plug *Brik*s together. You have to choose which *Brik*s you need for your task. That's why input and output *Variable*s are only from simple *Perl* types: *ARRAY*s, *HASH*es or *SCALAR*s. For the non-*Perl* reader, *ARRAY*s are like lists in *Python*, *HASH*es would be dictionaries and *SCALAR*s the simple strings or numbers (this is a simplified view).

```
Meta:~> my $h = { a => 1, b => 2}
{ a => 1, b => 2 }
Meta:~> my $a = [ 'a', 'b', 'c' ]
["a", "b", "c"]
Meta:~> my $n = 100
100
Meta:~> my $s = "str"
"str"
```

3 Definition and usage of *Brik*s

3.1 Basic concept

The main concept behind a *Brik* is: *do something once, the good way*. We were tired of:

- writing again some scripts or modules to do same things
- searching where we put this script to do such thing (and eventually not finding it)
- forgetting what the syntax is for this tool I rarely use (and syntax change from one tool to another)
- wasting time doing a simple thing because the tool to use is too complex (and author of this paper has his share of responsibility for some of them)

We want to reduce a tool to the first task it has been designed for. Every other function of a tool that is not its core business has to be moved to another one. A good *Brik* hides the details to let you concentrate on the task you want to accomplish, it is a black-box. *Metabrik* itself is built using a few *Brik*s: *core::context*, *core::global*, *core::log* and *core::shell*.

That's the minimum you have to load for using *Metabrik*. But don't be afraid, *core::context* auto-loads the three others and set them with appropriate default values. And you just have to launch the *metabrik* program [6] which does that for you.

3.2 Age of tool normalization

We thought it was time to normalize tools usage. Normalization has to take into account the name of tools, the name of usable parameters, and of course the handling of input and output so we can easily plug them together. In the end, a *Brik* is a blackbox you want to use to do something. You don't want to waste time searching how to do it. You can still know how to do it by looking at the source code. *Metasploit* did a great job at normalizing how we write and use *exploits*, we think it is time to do the same for all other tools.

Once a *Brik* is written, you want to remember how to use it weeks or months after. That's also where normalization helps. Self-documentation also helps, and that's why documentation has been integrated directly into the source code within *Brik's Properties*. Thanks to that, you can really develop quickly a reusable *Metatool*s that will be maintainable in the long run.

A first (failed) attempt at that goal was *Redacted for anonymous review*. It was so complex to use that even the author stopped using it. That's why there is a *Redacted for anonymous review Brik* today.

3.3 Vocabulary

You also have the list of *Command*s the *Brik* is capable of along with their *Argument*s. A *Command* has a list of *Argument*s you have to pass and some other optional ones that are always placed at the end of this list.

For instance, if you want to call the *from_string Command* from the shell, you just have to run it like: *run lookup::alexa from_string google.com*. It will return a true or a false value depending on the fact it is listed or not. See Appendix A for the implementation of *from_string Command*.

You see that there are also default values for *Attribute*s, where needed. Furthermore, this *Brik* is a subclass from another one: *client::www*. Thus, we can use all inherited *Command*s as we would do in any object-oriented language. The following is an an example of how to use and lookup some domains with this *Brik*.

```
Meta:~/tmp> use lookup::alexa
[*] core::shell: use: Brik \
  [lookup::alexa] success
Meta:~/tmp> run lookup::alexa update
[+] mirror: file [/...exa-top1m.csv.zip] \
    not modified since last check
[]
Meta:~/tmp> run lookup::alexa \
  from_string google.com
1
```

```
Meta:~/tmp> run lookup::alexa \
  from_string metabrik.org
0
Meta:~/tmp> $RUN
0
```

A *Brik* has also a *Category* and may have a *Subcategory*. Our *lookup::alexa Brik* is from *lookup Category*. You can search for other *lookup* using the *brik::search Brik* which we encourage you to add to your *.metabrik_rc* file. Just like the *brik::tool* one. *Brik*s have *Tag*s you can search for with the *brik::search Brik* too. As an exercise, try to search for the *Tag string* with *brik::search Brik*.

*Brik*s are sorted by *Categories*. For instance, you have a bunch of string manipulation *Brik*s in the *string::* Category*. *Brik*s from a *Category* usually share a set of common *Command* names, and that's where we introduce the concept of *Command* normalization. For instance, if you want to encode or decode *Base64*, *ROT13*, *XML* or *JSON*, you will always call the *encode* or *decode Command*s. Other popular normalized *Command*s are *scan* or *update*.

Another example of normalization can be taken from the *system Category*. You have a suite of *system::*::package Brik*s. The *Subcategory* is used to differentiate between systems (ie. *Ubuntu*, *FreeBSD*, ...). What would you expect from a *system::package Brik*? To perform search, installation, removing or upgrading of system packages, for instance. But do you really need to know the exact command to run for every system in the world? That's why when you call *system::package install Command* the corresponding system dependent command is run for you. We applied the same principle for *system::service* suite of *Brik*s.

This categorization system is directly inspired from the *FreeBSD ports* [7] system. What we also took from this system is the *UPDATING* [8] file, something you have to read each time you do an update of the platform. If there is an *API* change in a *Brik*, it will be listed here and you should be able to easily update your own tools accordingly.

As we speak about *Categories*, we have to speak about *Repositories*: this is where *Brik*s will be stored so we can use them. You have a public *Repository*, but you can have private ones if you want to. But remember, the more *Brik*s we have, the easier our tasks will become, so please consider contributing.

3.4 Playing with *Brik*s

We have seen some of the available *Command*s, but you still don't know how to load a *Brik*. You use the *use Command*, like in *Metasploit*. You can then *set* or *get* its *Attribute*s and *run Command*s on them. Usually, you call the *help Command* just after having used a *Brik*, except if you already know it by heart. Here is an example of how to use the *lookup::alexa Brik*:

```
Meta:~> use lookup::alexa
```

```
[*] core::shell: use: Brik \
  [lookup::alexa] success
Meta:~> help lookup::alexa
[+] set lookup::alexa datadir <datadir>
[+] set lookup::alexa input <file>
[+] set lookup::alexa url <url>
[+] run lookup::alexa from_string <domain>
[+] run lookup::alexa load [ <input> ]
[+] run lookup::alexa update
```

Remember automatic *Variable*s? There are more of them: *$SET*, *$GET*, *$RUN* for the main ones. They are respectively set when calling *set*, *get* or *run Command*s. There are also some special *Variable*s: *$GLO*, *$LOG*, *$SHE* and *$CON*. They give you access to internal *Brik*s respectively named *core::global*, *core::log*, *core::shell* and *core::context*. Try calling *help Command* on them. *Metabrik* is built on top of these four *Brik*s.

core::context is of special importance: it is at the heart of *Metabrik*. It will manily keep track of *Variable*s and loaded *Brik*s. This is the only *Brik* you must use if you want to write a *Metatool* and it will load the three other *core Brik*s that are also mandatory.

3.5 Shell for prototyping, scripts for *Metatool*ing

Once you have prototyped your new awesome tool, you may want to create a reusable script. There is a helper for you which can create a skeleton of a new *Metatool*: the *brik::tool Brik*. This *Brik* has many other usages, like simplifying the platform update process or installation of dependencies. See below for complete help:

```
Meta:~> help brik::tool
[+] set brik::tool datadir <datadir>
[+] set brik::tool repository <Repository>
[+] run brik::tool create_brik <Brik> \
  [ <Repository> ]
[+] run brik::tool create_tool \
  <filename.pl> [ <Repository> ]
[+] run brik::tool get_need_packages \
  [ <Brik> ]
[+] run brik::tool get_require_modules \
  [ <Brik> ]
[+] run brik::tool \
  install_all_need_packages
[+] run brik::tool \
  install_all_require_modules
[+] run brik::tool install_needed_packages \
  <Brik>
[+] run brik::tool \
  install_required_modules <Brik>
[+] run brik::tool test_repository
```

```
[+] run brik::tool update_core
[+] run brik::tool update_repository
```

The *create_tool* is the *Command* we want. You can see that there is also the *create_brik Command*, which will write a skeleton of a new *Brik* for you. Try creating a *Brik* called *test.pl* by running the *Command run brik::tool create_tool test.pl*. You should obtain something like in Appendix C.

To convert a prototype to a *Metatool* is not very complicated, you just have to understand how you transform a *shell Command* to a *Perl* call. If we take back our *lookup::alexa Brik*, we could write the tool as in Appendix D. For now, it is not really a *Metatool* because it only uses one *Brik*. As an exercise for the reader, try adding *DNS* name resolution by using the *client::dns Brik*.

4 Installing and updating the platform

It is also important to speak about installation and update features we put in *Metabrik*. The *brik::tool* is at the heart of this procedure, this is like the *system::package* of this platform. A guide is online [10] to help you bootstrap with *The Metabrik Shell*. Once you have bootstraped, you can update the platform directly from the *Mercurial* repository hosted on a *Trac* instance [11]. First, update both *core* and *repository*, then, if you wish, install all system packages and *Perl* modules dependencies. The order is important. Note that we have suppressed output from the following *Command*s. Beware, it may take some time.

```
Meta:~> use brik::tool
Meta:~> run brik::tool update_core
Meta:~> run brik::tool update_repository
Meta:~> run brik::tool \
  install_all_need_packages
Meta:~> run brik::tool \
  install_all_require_modules
```

If you don't want to mess with your system, you can just install a targeted set of system packages and/or *Perl* modules: the one you just need to make a *Brik* work. See below an example. Again, the order is important, and we have suppressed output from *Command*s.

```
Meta:~> use brik::tool
Meta:~> run brik::tool \
  install_needed_packages network::device
Meta:~> run brik::tool \
  install_required_modules network::device
```

You now have the basic skillset to play with the platform, it is time to turn into real-life scenarios to put your hands on.

5 Real-life scenarios

The best way to understand and learn about the platform is to show real-life problems that can be solved by using *Brik*s. In the following examples, we will show how to find required *Brik*s, use them, and in the end solve problems. We will not show output of every *Command*s, so *[..]* will be written where output has been truncated. You don't need all the output to understand how *Brik*s are plugged together. We also suppressed the *Meta* prefix from *Command*s. You now know all *Command*s starts with either *use*, *set*, *get*, *run* or *help*.

5.1 *Scalpel* improvements

We learned about *Scalpel* during a forensic training. This tool allows to extract data from a raw image of a disk or from any support. It searches for patterns revealing the existence of known file formats like Word or PDF documents, or JPEG images to name a few. It can extract all sorts of formats that you have to define in a configuration file.

Quickly, we spotted some room for improvement on the way this tool is working: writing the configuration file to select/unselect the documents you want to extract can be cumbersome, and a major weakness is it does not check for *MAGIC* type of extracted documents. Checking for *MAGIC* type would avoid the human to manually check valid and invalid files. Both these deficiencies can be solved by *Brik*s: *forensic::scalpel* and *file::type Brik*s.

Brikification is the process to improve existing tools by wrapping them in *Brik*s. Thus, we *Brikified* the *Scalpel* tool to add file *MAGIC* type checking and adding a *Command* to easily generate a configuration file with only wanted document types we search to extract for our current forensic work.

```
my $file = "raw-disk-image.dd"
my $odt = [ qw(odt) ]
use forensic::scalpel
run forensic::scalpel generate_conf $odt
run forensic::scalpel scan $file
my $verified = $RUN->{verified}
[
   '/tmp/ch9.scalp/odt-0-0/00000000.odt',
   '/tmp/ch9.scalp/odt-0-0/00000001.odt',
];
```

We wanted to extract some *OpenOffice document*s from a raw disk image. Thus, we asked to generate a configuration file to check for *odt* files only. The configuration file is saved in the *Brik*'s *datadir*, you don't care about how it is formatted or where it is stored anyway. Then we *scan* the file to extract them. Eventually, *Scalpel* tool finds some verified and/or unverified files. Verified files are the ones that were checked with *file::type Brik*, a *Brik* called and used under cover for you. At the end of this run, we

end up with only two files instead of a dozen.

Note: A full write-up describing the solving of a root-me challenge is available online on *Metabrik* blog [12].

5.2 Running a Tor server and redirecting all traffic

Setting up a *Tor* server can also be painful as well as configuring all your client tools to go through it. And if you fail configuring your client tools, you may end-up leaking your true IP address. We found a script called *nipe.pl* [13] that comes with a configuration file for *Tor* and a bunch of commands to install required dependencies to finally redirect all traffic to the configured server.

That's exactly the way we write *Brik*s. So we created a *server::tor Brik* that knows how to install dependencies thanks to *system::package Brik* and knows how to generate a configuration file to start a *Tor* server. We then use *GNU/Linux iptables* command to redirect all DNS and TCP traffic through it. As *server::tor* is also a system service, the *Brik* inherits *Properties* from the *system::service* one.

```
use server::tor
get server::tor
[+] server::tor conf torrc
[+] server::tor datadir /tmp/server-tor
[+] server::tor dns_listen 127.0.0.1
[+] server::tor dns_port 9061
[+] server::tor pidfile tor.pid
[+] server::tor tor_listen 127.0.0.1
[+] server::tor tor_port 9051
[+] server::tor user metabrik
[+] server::tor virtual_network \
  10.20.0.0/255.255.0.0
run server::tor generate_conf
run server::tor start
# Wait few seconds for the Tor
# connection to establish
```

We first loaded and showed available configuration options for the *Tor* server. We then generated a configuration file and started the server. You don't care how the configuration format works, but if you want to modify what has been generated for you, the *torrc* file can be found in *datadir* directory: *$HOME/metabrik/server-tor*. Many *Brik*s have *datadir Attribute* that tells where locally generated data files should be read or written. All these *datadir*s can be found in *$HOME/metabrik/BRIK_NAME* directory.

```
use network::linux::iptables
run network::linux::iptables save \
  current.txt
```

```
run network::linux::iptables \
  start_redirect_target_to 53 \
  127.0.0.1:9061
run network::linux::iptables \
  start_redirect_target_tcp_to \
   0.0.0.0/0 127.0.0.1:9051
```

We then redirect all network traffic destined to port *53/UDP* and *TCP* to the *Tor* network as all *TCP* connections to any target. You can now send emails through *SMTP*, connect through *SSH* or simply surf the Web anonymously as easily as not configuring any of your client tools. For the attentive reader, you may think that all traffic redirection includes the *Tor* server traffic, but the *start_redirect_target_to Command* added an *iptables* rule to let already established connections go through unredirected. That's why we had to wait a few seconds before setting rules with the *network::linux::iptables Brik*.

```
# Check your new IP address is a
# Tor exit node
use network::device
run network::device internet_address
my $ip = $RUN
run server::tor list_exit_nodes
my $ok = grep {/^$ip$/} @$RUN
# You can use Tor now
```

Before using *Tor*, you probably want to check that you are surfing through the relay. You need *network::device Brik* which serves as checking various network interface information to gather your address as seen from the Internet and compare it with the *Tor* exit node list. The good thing is the *list_exit_nodes Command* which returns an *ARRAY* of known exit nodes. Now you should feel confident and start using it. Once you are done using *Tor*, just clean your firewall state and stop the service.

```
# You are done using it,
# time to clean stuff
run network::linux::iptables \
  stop_redirect_target_tcp_to \
  0.0.0.0/0 127.0.0.1:9051
run network::linux::iptables \
  stop_redirect_target_to \
  53 127.0.0.1:9061
run network::linux::iptables restore \
  current.txt
run server::tor stop
```

5.3 Augmenting a traceroute tool by adding IP location data

We want to add some geolocation data to a *TCP* traceroute but we don't want to modify the sourcecode of the *tcptraceroute* program for that. Thus, we use the *network::traceroute tcp Command* which calls this traceroute program for us under the

hood. It parses its results and outputs a *HASH* with different hops taken to reach a target.

```
use network::traceroute
use lookup::iplocation
run lookup::iplocation brik_self
my $li = $RUN
```

You should understand the first few lines of commands except the call to *brik_self* one. This *Command* is from the base class and allows to extract a *Perl Variable* to make it accessible to the *Perl* interpreter. This access is important when you want to use true *Perl* code like in the for loop shown in the figure. *brik_self* simply returns a pointer to the *lookup::iplocation* object so you can use it from *Perl* code.

```
run network::traceroute tcp google.com 80
{
  5   => "80.236.1.161",
  6   => "72.14.239.145",
  7   => "72.14.233.83",
  8   => "216.58.211.110",
[..]
```

Finally, you loop through all the hops found by the traceroute, lookup the geolocation of the IP address (we just want the country name but there is more in *lookup::iplocation Brik*) and finally enrich the $RUN data with this new country information. We have to save the new generated output to a *$save* variable to avoid loosing that work.

```
for (keys %$RUN) {
.. my $loc = $li->from_ip($RUN->{$_})
..   or next;
.. $RUN->{$_} = {
..   ip => $RUN->{$_},
..   country => $loc->{country_code}
.. };
.. }
my $save = $RUN
{
  5   => { country => "FR",
           ip => "80.236.1.161" },
  6   => { country => "US",
           ip => "72.14.239.145" },
  7   => { country => "US",
           ip => "72.14.233.83" },
  8   => { country => "US",
           ip => "216.58.211.110" },
[..]
```

As you can see, a few *Metabrik* command lines are enough to create a new and augmented tool instead of modifying the sourcecode of an existing one.

5.4 Extracting IOCs from an instrumented scapegoat VM

In this scenario, we may want to extract *IOC*s from a malware. The idea would be to take a snapshot of an Operating System state like running processes, registry keys, network sessions or files on-disk. After saving this state, you execute a malware on the scapegoat machine and then perform a *diff* on the state before and after execution of the malware. In our example here, the malware will simply be *calc.exe* and we will only take a snapshot of process list. We will first use *WMI* to take this snapshot information and then use *forensic::volatility Brik* which is a better way of doing.

We load all required *Brik*s: *remote::wmi*, *remote::winexe*, *forensic::volatility* and *system::virtualbox Brik*s as you can see here:

```
use remote::wmi
use remote::winexe
use forensic::volatility
use system::virtualbox
```

We use the *system::virtualbox* to search for the identifier of a *Windows* VM, start the system in headless mode (we really don't want to use that *Windows GUI*) and take a snapshot of memory state before we execute a malware remotely (file has to be already hosted locally on the target system) by using *remote::wmi Brik* like:

```
my $id = '602782ec-...-4e56a8bd5657'
use system::virtualbox
run system::virtualbox list
set system::virtualbox type headless
run system::virtualbox start $id
run system::virtualbox snapshot_live $id \
   "before calc.exe"
```

We list processes before executing the malware and after. We finally use the *grep Perl* function to verify *calc.exe* has been run:

```
my $win = '192.168.56.101'
my $user = 'Administrator'
my $password = 'YOUR_SECRET'
set remote::wmi host $win
set remote::wmi user $user
set remote::wmi password $password
run remote::wmi get_win32_process
for (@$RUN) {
.. print $_->{Name}."\n";
.. }

set remote::winexe host $win
set remote::winexe user $user
set remote::winexe password $password
run remote::winexe execute \
```

```
  "cmd.exe /c calc.exe"
run remote::wmi get_win32_process
my @processes = map { $_->{Name} } @$RUN
my $found = grep { /calc.exe/ } @processes
```

Another and better option would have been to take a live snapshot using the hypervisor memory snapshoting options to let *Volatility* analyze the memory in search of those *IOC*s. Our VM is ran by *VirtualBox*, we can use the *dumpguestcore Command* to take that memory snapshot.

Unfortunately, *Volatility* is not able to read such dump by default. We have to write a *Command* to extract that memory dump from *VirtualBox* and save it as a file known to *Volatility*. See below how we extracted it. Under the hood, it is using a few different *Brik*s to do so: *file::readelf* is one of them and is used to show *ELF* headers of this *dumpguestcore* file.

```
run system::virtualbox dumpguestcore $idÂ \
   dump.core
run system::virtualbox \
  extract_memdump_from_dumpguestcore \
  dump.core dump.volatility
```

In the end, we can use *forensic::volatility pslist Command* to list processes. We described quickly how we can use all these *Brik*s, a full write-up can be found online at [14].

```
use forensic::volatility
set forensic::volatility input dump.volatility
run forensic::volatility imageinfo
set forensic::volatility profile $RUN->[0]
set forensic::volatility pslist
```

6 Conclusion and future work

We presented *The Metabrik Platform* and demonstrated its capabilities. We just scratched the surface of what you can do with its more than 30 *Categories* of tools and over 180 *Brik*s. We hope the platform will be well perceived and adopted by the community at large. The more *Brik*s we have, the more quick tasks will become.

For the future, we want to be able to completely replace our login shell by *The Metabrik Shell* which will feature everything that we need like pipes, redirections, job management and a better *readline* than the *GNU* ones. We hope to merge the *Zoid shell* [15] as this is the currently most advanced *Perl* shell out there.

Finally, we will continue to develop many *Brik*s and write *Metatool*s around them, as this is the ultimate goal of this platform to provide easy to use tools to perform complex tasks. Last thing, more examples of using *Metabrik* can be found on our blog [16]. Stay tuned by following us on Twitter @*Metabrik*.

A Implementation of the *from_string Command*

```
sub from_string {
   my $self = shift;
   my ($domain) = @_;

   $self->brik_help_run_undef_arg(
     'from_string', $domain
   ) or return;

   my $data = $self->_loaded;
   if (! defined($data)) {
      $data = $self->load or return;
      $self->_loaded($data);
   }

   for my $this (@$data) {
      if ($this->[1] eq $domain) {
         return 1;
      }
   }

   return 0;
}
```

B Properties defining the *lookup::alexa Brik*

```
use base qw(Metabrik::Client::Www);

sub brik_properties {
   return {
      revision => '$Revision: d7018a3cea2f $',
      tags => [ qw(unstable top million 1m) ],
      author => 'XXX <XXX[at]metabrik.org>',
      license => 'http://opensource.org/'.
                 'licenses/BSD-3-Clause',
      attributes => {
         datadir => [ qw(datadir) ],
         url => [ qw(url) ],
         input => [ qw(file) ],
         _loaded => [ qw(INTERNAL) ],
      },
      attributes_default => {
         url => 'http://s3.amazonaws.com/'.
                'alexa-static/top-1m.csv.zip',
         # Stored in datadir by default
         input => 'top-1m.csv',
      },
      commands => {
         update => [ ],
         load => [ qw(input|OPTIONAL) ],
         from_string => [ qw(domain) ],
      },
      require_modules => {
         'Metabrik::File::Compress' => [ ],
         'Metabrik::File::Csv' => [ ],
      },
   };
}
```

C A new *Metatool* skeleton

```
#!/usr/bin/env perl
#
# $Id$
#
use strict;
use warnings;

# Uncomment to use a custom repository
#use lib
#  qw(/home/metabrik/repository/lib);

use Data::Dumper;
use Metabrik::Core::Context;
# Put other Briks to use here
# use Metabrik::File::Text;

my $con = Metabrik::Core::Context->new
   or die("core::context");

# Init other Briks here
# my $ft =
#   Metabrik::File::Text->
#     new_from_brik_init($con)
#       or die("file::text");

# Put Metatool code here
# $ft->write("test", "/tmp/test.txt");

exit(0);
```

D Creating the *lookup::alexa Metatool*

```
#!/usr/bin/env perl
#
# $Id$
#
use strict;
use warnings;

use Data::Dumper;
use Metabrik::Core::Context;
use Metabrik::Lookup::Alexa;

my $con = Metabrik::Core::Context->new
   or die("core::context");

my $la =
   Metabrik::Lookup::Alexa->
      new_from_brik_init($con)
        or die("lookup::alexa");
$la->update or die("update");

my $yes = $la->from_string('google.com');
print "google.com: $yes\n";

my $no = $la->from_string('metabrik.org');
print "metabrik.org: $no\n";

exit(0);
```

References

1. The Metabrik Platform
   https://www.metabrik.org/
2. REbus - Overview
   https://bitbucket.org/iwseclabs/rebus
3. SSTIC 2015 - REbus - Philippe Biondi, Sarah Zennou, Xavier Mehrenberger
   https://www.sstic.org/2015/presentation/rebus/
4. Metasploit - Penetration Testing Software
   https://metasploit.com/
5. Metabrik Repository
   http://trac.metabrik.org/browser/repository/
6. metabrik program
   http://trac.metabrik.org/browser/core/bin/metabrik
7. About FreeBSD Ports
   https://www.freebsd.org/ports/
8. UPDATING file
   http://trac.metabrik.org/browser/repository/UPDATING

9. Scalpel - ForensicsWiki
http://www.forensicswiki.org/wiki/Scalpel
10. Installation guide for Metabrik
https://www.metabrik.org/metabrik/install/
11. Metabrik Trac server
http://trac.metabrik.org/
12. Solving a root-me forensic challenge with Metabrik and Scalpel
https://www.metabrik.org/?s=root-me
13. nipe.pl by Heitor Gouvea (@HeitorG)
https://github.com/HeitorG/nipe
14. Malware analysis with VM instrumentation, WMI, winexe, Volatility and Metabrik
https://www.metabrik.org/?s=malware
15. Zoidberg
https://metacpan.org/pod/Zoidberg
16. The Metabrik Blog
https://www.metabrik.org/