DEBUGGING AND MONITORING MALWARE NETWORK ACTIVITIES WITH HAKA

Benoît Ancel & Mehdi Talbi
Stormshield, France

Email {benoit.ancel, mehdi.talbi}@stormshield.eu

ABSTRACT

Malware analysts have an arsenal of tools with which to reverse engineer malware but lack the means to monitor, debug and control malicious network traffic. In this paper we propose the use of Haka, an open source security-oriented language, to address this problem. The rationale for this is fourfold: first, Haka features a grammar that allows one to naturally express malware protocol dissectors. Second, Haka provides the most advanced API for packet and stream manipulation. One can drop, create and inject packets. Haka also supports on-the-fly packet modification. For example, this enables us to hijack some botnet commands to automatically disinfect a set of compromised computers if the malware supports such C&C commands (e.g. uninstall). Third, Haka has an interactive mode that enables it to break into particular packets/streams and inspect their content. Finally, Haka provides a dedicated and customizable tool (*Hakabana*) to provide a real-time visualization of malware network activities through *Kibana* dashboards.

In this paper we will a provide a set of protocol dissectors and security rules in order to monitor the C&C activities of well-known malware families and show some statistics and interesting results from our tracking of real-word C&C traffic.

1. INTRODUCTION

In this paper, we propose to use Haka in order to debug and monitor the C&C activities of well-known malware families. The motivations underlying this choice are multiple: first, Haka features a simple grammar that allows one to specify protocols naturally. Thus, malware analysts can focus on reverse engineering malware rather than wasting their time with the tedious task of writing protocol dissectors.

Second, Haka provides the most advanced API to filter unwanted packets and streams. One can drop packets or create new ones and inject them. Haka also supports on-the-fly packet modification. This is one of the main features of Haka since all complex tasks, such as resizing packets and setting correct sequence numbers, are done transparently to the user. For example, this enables us to hijack some botnet commands to disinfect a set of compromised computers automatically if the malware supports such C&C commands (e.g. uninstall).

Third, Haka ships with several security modules, enabling end-users to write fine-grained security rules. For example, Haka embeds a disassembler module (based on the *Capstone* engine) that allows one to disassemble streams into ASM instructions and detect obfuscated shellcodes at the network level. Haka also ships with a crypto module that is helpful for decrypting C&C communications, and it features a pattern matching engine (PCRE) that, for example, is useful for detecting an obfuscated payload scattered over multiple packets.

Fourth, Haka provides a way to filter packets interactively. This mode enables users to break into particular packets and inspect their content. This is useful for debugging a packet capture file or fuzzing a protocol by modifying some packets fields on the fly and observing how the C&C server behaves.

Finally, Haka provides a dedicated and customizable tool (*Hakabana*) to provide a real-time visualization of malware network activities through *Kibana* dashboards. *Hakabana* consists of a set of security rules that push network data (e.g. protocol info, geolocalization, HTTP traffic details) into an *Elasticsearch* index and make them available through *Kibana* dashboards.

In this paper, we propose a set of protocol dissectors and security rules of known malware families, namely China-Z and Athena, and present some interesting results from our tracking of real-world C&C traffic.

The remainder of this paper is organized as follows:

Section 2 presents the Haka architecture with a focus on protocol dissection and security rule writing. In Section 3 we provide an analysis of the network monitoring of two malware families using Haka. Section 4 discusses related work and, finally, we sketch some concluding remarks on this work in Section 5.

2. HAKA – AN OPEN SOURCE SECURITY-ORIENTED LANGUAGE

Haka is an open source security-oriented language that allows one to specify both the protocol and the security rules. Haka is based on the Lua language [1–3]. It is a simple, lightweight (~200KB) and fast (a JiT compiler is available) scripting language.

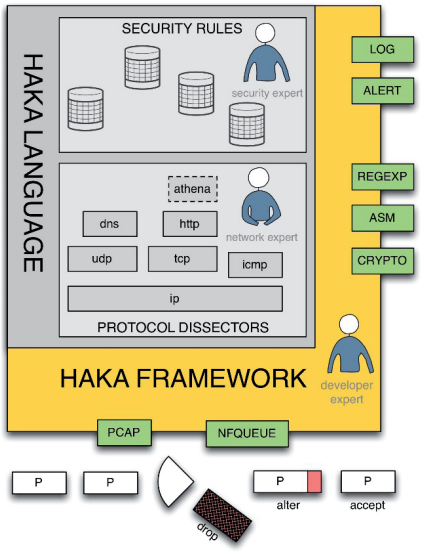

Figure 1: Haka architecture.

The Haka language is embedded into a modular framework. In the bottom of Figure 1 we have packet capture modules (pcap and nfqueue) enabling users to apply their security policy on live captured traffic or to replay it on a packet trace file. The framework also provides logging and alerting modules (syslog and Elasticsearch) to log and report suspicious activities, respectively. Finally, the framework features auxiliary modules (crypto, disassembler and pattern matching engine). These modules enable one to write fine-grained security rules, for example to detect obfuscated shellcodes.

2.1 Protocol dissection

Haka features a grammar allowing one to specify both text-based protocols (e.g. HTTP) and binary-based protocols (e.g. DNS). It also supports stream-based protocols (e.g. HTTP) as well as packet-based protocols (e.g. IP). The grammar defines a set of constructs to represent elementary types such as numbers and bytes, but also complex data structures like arrays, records and branches. From the resulting specifications, we can build a graph that is used to parse protocol messages. All protocol fields depicted at this stage are provided in read/write access to security rules.

Additionally, end-users can specify the protocol state machine. A state machine is defined as a set of states and transitions between these states.

The overall specification (protocol message syntax + protocol state machine) allows one to check if protocol messages conform to their syntax and to discard unexpected messages (e.g. protocol state machine violation).

Thanks to that grammar we were able to specify several TCP/IP-based protocols (IP, ICMP, TCP, UDP, HTTP, SMTP, DNS) and SCADA protocols (e.g. Modbus) with little effort.

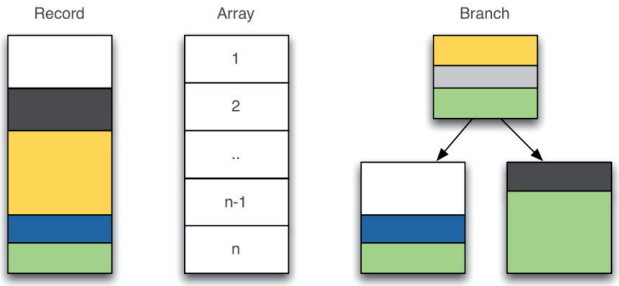

Figure 2: Compound elements.

Haka provides two types of grammar entities to specify protocol message syntax: final entities and compound entities. The first one allows one to parse elementary data types such as bytes, numbers, flags or regular expressions. The latter is used to parse complex data structures such as those illustrated in Figure 2. The *record* represents a structure of final and/or compound entities. The *array* entity could, for instance, represent a table of records. For example, it is useful for dealing with a list of HTTP headers. Finally, the *branch* entity allows one to select a parsing path according to previously parsed data.

The example 1 shown in Figure 3 is a specification in Haka grammar of a native protocol *P*. The protocol consists of a single message *M* made up of value field *N* followed by *N* couple of integers *A* and *B*.

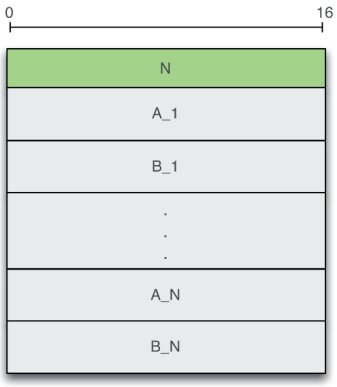

Figure 3: Protocol P.

The specification starts by creating the grammar for protocol *P*. Then we define a record made up of two named fields, *A* and *B*. These fields will be parsed as numbers and their value will be made accessible. The endianness can be specified as an extra argument to the number entity (e.g. *number(32, 'little')*). Then, we define the format of message *M* as a record made up of named field *N* and an array of the previously defined record *E*. The size of the array is determined by applying a *count* option on the array entity. More precisely, the size of the array is set dynamically from a previously parsed field *N*. Finally, the desired elements of the grammar (*M*) are compiled, which make them ready for parsing.

```
P.grammar = haka.grammar.new("P", function ()
    E = record{
        field("A", number(32)),
        field("B", number(32))
    }

    M = record{
        field("N", number(16)),
        field("L", array(E)
            :count(function (self)
                return self.N
            end)
        )
    }

    export(M)
end)
```

Figure 4: Specification of protocol P in Haka.

Note that the grammar also supports advanced features such as recursion and inheritance. For example, the specification shown in Figure 5 extends the previous one in order to overload *E* element definition. Now, *A* and *B* fields fit on 16 bits each.

```
P2.grammar = haka.grammar.new("P2", function()
    extend(P)

    E = record{
        field("A", number(16)),
        field("B", number(16))
    }
end)
```

Figure 5: Grammar inheritance.

These features are useful, for instance, for coping with malware families that change slightly the way they communicate.

In Haka, one can also specify the state machine of a protocol. A protocol state machine is defined as a set of states and transitions between states. In Haka, a transition consists of:

- An event to attach to. Possible events are: *up* and *down* (events triggered when up-coming and down-coming data is received, respectively), *parse error* (event triggered on data parsing error), etc. The full list of available events is available in [4].
- An optional checking function *when* to decide whether this transition should be taken or not.
- An optional function *execute* to perform a specific action.
- A target state *jump* to jump to.

Assuming that the previously defined protocol *P* (see Figure 3) is a request/response protocol, one could define its state machine as shown in Figure 6.

```
P.state_machine =
    haka.state_machine.new("P", function ()

    state_type(BidirectionalState)

    request  = state(P.grammar.M, nil)
    response = state(nil, P.grammar.M)

    request:on{
        event = events.up,
        execute = function (self, res)
            self.request = res
            self.response = nil
        end,
        jump = response,
    }

    response:on{
        event = events.down,
        execute = function (self, res)
            self.response = nil
        end
        jump = request,
    }

    any:on{
        event = events.parse_error,
        execute = function (self, err)
            haka.alert{
                description = "invalid message format",
                severity = "low"
            }
        end,
        jump = fail,
    }

    any:on{
        event = events.missing_grammar,
        execute = function (self, dir, payload)
            haka.alert{
                description = "unexpected data",
                severity = "low"
            }
        end,
        jump = fail,
    }

    any:on{
        event = events.fail,
        execute = function (self)
            self:drop()
        end,
    }

    initial(request)
end)
```

Figure 6: Specification of the state machine of protocol P.

The code shown in Figure 6 instantiates a new state machine for protocol *P*. The type of the state machine is set to *BidirectionalState* as we are expecting data in both directions: up (from client to server) and down (from server to client). Then we create two states, *request* and *response*. For each state and for each direction we indicate the grammar that should be used to parse messages.

P requests and responses should comply with their specification written during the protocol message specification step.

We define a single transition on state '*request*' that jumps to *response* after successfully parsing a *request*.

In the same way, we define a single transition on state '*response*' that jumps to state *response* if *responses* conform to the provided syntax.

Finally, we define default transitions through the reserved state '*any*'. The first two common transitions jump to a failure state in case of parsing errors (malformed messages) and unavailable grammar (unexpected messages), respectively. The last transition performs a specific action to drop the connection when the *fail* event is triggered.

The last line of the state machine definition allows one to select an initial state.

Thanks to this feature, we were able to write a complex state machine of protocols such as TCP, which consists of several states and which requires the capacity to handle timeouts.

2.2 Security rule writing

Haka provides a simple way to write security rules in order to filter, modify, create and inject packets and streams. When a flow is detected as malicious, users can report an alert or drop the flow.

Hereafter, we present the syntax of security rules through some examples.

2.2.1 Packet filtering

Figure 7 shows a basic packet filtering rule that blocks all connections from a given network address. The first lines load the required protocol dissectors, namely *IPv4* and *tcp_connection* dissectors. The first one handles IPv4 packets. The latter is a stateful TCP dissector that maintains a connection table and manages TCP streams. The next line defines the network address that we want to block.

```
local ipv4 = require("protocol/ipv4")
local tcp = require("protocol/tcp_connection")

local net = ipv4.network("172.16.3.0/24")

haka.rule{
    hook = tcp.events.new_connection,
    eval = function (flow, pkt)
        haka.log("tcp connection %s:%i -> %s:%i",
            flow.srcip, flow.srcport,
            flow.dstip, flow.dstport
        )

        if net:contains(flow.dstip) then
            haka.alert{
                severity = "low",
                description = "connection refused",
                start_time = pkt.ip.raw.timestamp
            }
            flow:drop()
        end
    end
}
```

Figure 7: Packet filtering.

The security rule is defined through the *haka.rule* keyword. A security rule is made up of a *hook* and an evaluation function, *eval*. The *hook* is an event that will trigger the evaluation of the security rule. In this example, the security rule will be evaluated at each attempt to establish a TCP connection. The parameters passed to the evaluation function depend on the event. In the case of a *new_connection* event, *eval* has two parameters: *flow* and *pkt*. The first one holds details about the connection and the latter is a table containing all TCP (and lower layer) packet fields.

In the core of the security rule, we first log (*haka.log*) some information about the current connection. Then, we check if the source address belongs to the non-authorized IP address range defined previously. If this test succeeds, we raise an alert (*haka.alert*) and drop the connection. Note that we reported only a few details in the alert. One can add more information, such as the source and the targeted service.

2.2.2 Stream filtering

Malware analysts can write advanced security rules to detect malware infection attempts at an early stage and block them. For example, the rule shown in Figure 8 allows for the detection of a shellcode by leveraging the pattern-matching and disassembler modules. These two modules are stream-based which, for instance, enables us to detect a malicious payload scattered over multiple packets.

This security rule uses a regular expression to detect a nop sled. We enable the *streamed* option, which means that the matching function will block and wait for data to be available to proceed with matching. If a nop sled is detected, we raise an alert and dump the shellcode instructions. Note that the pattern-matching function updates the iterator position, which afterwards points to the shellcode.

```
local tcp = require("protocol/tcp_connection")

local rem = require("regexp/pcre")
local re = rem.re:compile("%x90{100,}")

local asm = require("misc/asm")
local dasm = asm.new_disassembler("x86", "32")

haka.rule {
    hook = tcp.events.receive_data,
    options = {
        streamed = true,
    },
    eval = function (flow, iter, direction)
        if re:match(iter, false) then
            haka.alert{
                description = "nop sled detected",
            }
            -- dump instructions following nop sled
            dasm:dump_instructions(iter)
        end
    end
}
```

Figure 8: Stream filtering.

2.2.3 Interactive filtering mode

Haka provides a way to filter packets interactively. One can break on specific packets and inspect their contents through the Haka API. For example, the security rule shown in Figure 9 enables the interactive mode on HTTP traffic. More precisely, Haka will halt whenever an HTTP request with a specific User-Agent is received (Sality's malware User-Agent).

```
local http = require("protocol/http")

http.install_tcp_rule(80)

haka.rule{
    hook = http.events.request,
    eval = function (http, req)
        if request.headers["User-Agent"] == "Opera/8.81 (Windows NT 6.0; U; en)" then
            haka.interactive_rule("debug")(http, req)
        end
    end
}
```

Figure 9: Interactive filtering mode.

Note that this mode is best suited for an offline packet trace analysis as packet addition will add a lot of delay.

3. USE CASES – C&C TRAFFIC ANALYSIS

3.1 China-Z

China-Z malware is a piece of ELF malware that belongs to a huge family of Chinese DDoS bots (AES, BillGates, Elknot, MrBlack, etc.) [5]. For almost three years, these pieces of malware have been known for infecting servers by exploiting various vulnerabilities (Telnet, SSH, Elasticsearch, etc.).

There are five known variants of this malware: China- Z/A, China-Z/B, China-Z/C, China-Z/O and China-Z/S. This use case is based on the last variant. Throughout the paper, we refer to the China-Z/S variant as China-Z.

China-Z is mainly used for DDoS campaigns. It also includes features such as configuration and binary update.

Figure 10 shows the format of requests sent by the bot. The size of the initial request is 228 bytes. The rest of the requests are padded with eight null bytes. The first field is equal to 0 if the infected host is currently running a DoS attack and is equal to 1 otherwise. The rest of the payload encodes in text format the result of commands run on the infected host to extract various info such as the OS (64 bytes), the available RAM (32 bytes), the CPU (32 bytes), etc.

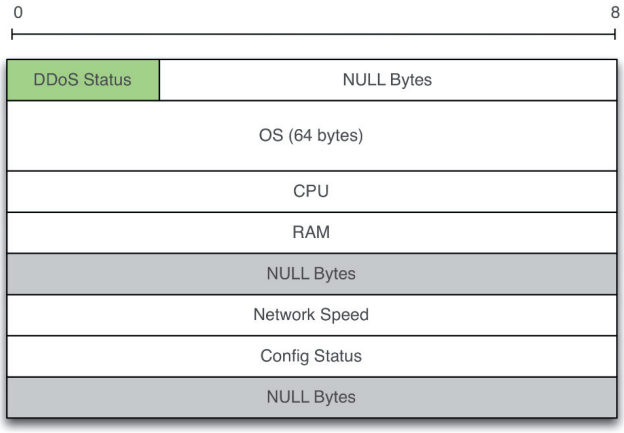

Figure 10: China-Z – request message format.

Figure 11 shows the format of commands sent by the botnet. The first field is the ID of the command. We distinguish five types of commands: 0x00 (DDoS Start), 0x01 (Config Update), 0x02

(DDoS Stop), 0x03 (Payload Update), and 0x31 (Heartbeat). All responses are 516 bytes long except for the heartbeat message (1 byte). The payload depends on the command. For example, the DDoS command has four parameters: IP (128 bytes), port (4 bytes), protocol (4 bytes) and duration bytes (bytes). Update's commands are followed by a URL.

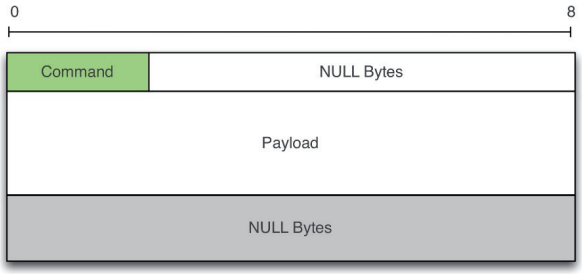

Figure 11: China-Z – response message format.

3.1.2 Dissector and security rule writing

Hereafter, we present the main steps towards the specification of China-Z dissector. The full code is available in [6]. In Figure 12, we create the dissector and events.

```
local chinaz_dissector = haka.dissector.new{
    type = tcp_connection.helper.TcpFlowDissector,
    name = 'chinaz'
}

chinaz_dissector:register_event('request')
chinaz_dissector:register_event('response')
```

Figure 12: China-Z – dissector and events.

First, we create the dissector by specifying its name and type. The type is set to *TcpFlowDissector* since multiple packets are exchanged during a China-Z session. The next step creates two events: *request* and *response*. Events are the glue between dissectors and security rules. Dissectors create events and then trigger them. As a result, all security rules hooking to an event will be evaluated.

In Figure 13, we give the protocol grammar which depicts the format of exchanged messages. We first specify terminal tokens to match strings of different lengths. The initial request is defined

```
chinaz_dissector.grammar = haka.grammar.new("chinaz", function ()

    string_40 = token('.{64}')
    string_20 = token('.{32}')
    string_null = token('[^%0]+[%0]+')

    init_request = record{
        field('command', number(32, 'little')),
        field('os',      string_40),
        field('payload', array(field('data', string_20)):count(5)),
    }

    request = record{
        init_request,
        bytes():count(8)
    }

    url_data = record{
        bytes():count(3),
        field('url', string_null),
        bytes():count(368)
    }

    ip_data = record{
        bytes():count(3),
        field('ip',      string_null:convert(ipv4_addr_convert, true)),
        field('port',    number(32, 'little')),
        field('type',    number(32, 'little')),
        field('duration', number(32, 'little')),
        bytes():count(372)
    }

    response = record{
        field('command', number(8)),
        branch(
            {
                [0x00] = ip_data,
                [0x01] = url_data,
                [0x02] = bytes():count(515),
                [0x03] = url_data,
                [0x31] = empty(),
                default = bytes():count(515)
            },
            function(self, ctx)
                return self.command
            end
        )
    }

    export(init_request, request, response)

end)
```

Figure 13: China-Z – grammar.

as a record made up of a command (four bytes in little-endian format), an OS (64 chars) and a payload. The latter is an array of strings of 20 bytes each. The rest of the requests are encoded by extending the initial request with eight-byte padding. The response is defined as a record made up of a command and a payload. The latter is encoded using a *branch* entity, which allows us to select a parsing path according to the command value.

Finally, in Figure 14, we create the protocol state machine and set its type to *BidirectionalState*. In a bidirectional setting, we

```
chinaz_dissector.state_machine = haka.state_machine.new("chinaz", function ()
    state_type(BidirectionalState)

    init = state(chinaz_dissector.grammar.init_request, nil)
    session = state(chinaz_dissector.grammar.request, chinaz_dissector.grammar.response)

    init:on{
        event = events.up,
        execute = function (self, res) self:trigger("request", res) end
        jump = session
    }

    session:on{
        event = events.up,
        execute = function (self, res) self:trigger("request", res) end
    }

    session:on{
        event = events.down,
        execute = function (self, res) self:trigger("response", res) end
    }

    initial(init)
end)
```

Figure 14: China-Z – state machine.

create new states by selecting the expected grammar for each direction (e.g. *up* and *down*). In our case, we create two states: *init* and *session*. On the first state, we set a single transition in order to handle the initial request sent by the bot. Once this packet is received we jump to the *session* transition. The latter triggers the *request* event on up-coming data and triggers the *response* event on down-coming data.

We found it useful to add security rules to block DDoS attacks while monitoring a piece of malware that is used primarily to carry out such attacks. In Figure 15, we first load the previously defined dissector. Then, we register our dissector on port 25005 (i.e. the TCP port used by China-Z).

```
local chinaz = require("protocol/chinaz")
local udp = require("protocol/udp_connection")

chinaz.install_tcp_rule(25005)

local blacklist = {}

haka.rule{
    hook = chinaz.events.response,
    eval = function (chinaz, response)
        if response.command == 0 then -- DDoS Command
            if not table.contains(blacklist, response.ip.packed) then
                blacklist[response.ip.packed] = true
            end
        end
    end
}

haka.rule{
    hook = udp.events.new_connection,
    eval = function(flow, pkt)
        if table.contains(blacklist, flow.dstip.packed) then
            pkt:drop()
        end
    end
}
```

Figure 15: China-Z – block DDoS attacks.

The first security rule is evaluated whenever we receive a command from the botnet (i.e. event response). If the command is a DDoS command, then we extract the targeted IP address and store it in a table of blacklisted IP addresses. This table is used by the second security rule that drops each UDP connection establishment if the IP address belongs to the list of blacklisted IP addresses. Note that one could use a similar security rule to block TCP-based DDoS attacks.

3.1.3 C&C traffic analysis

We have designed a *Kibana* dashboard to monitor China-Z C&C activities (see Figure 16). We have monitored the network traffic of an infected host over three days and observed that the botnet behind this malware mainly targets sites located in Asia.

Unfortunately, we haven't observed any updates of the malware binary or its configuration.

3.2 Athena

Athena is a malware family that has been seen in the wild since 2012 [7]. First appearing in an IRC version, Athena moved to HTTP in 2013. Very popular due to many builders leaking it, this malware is still in use today though it is not sophisticated and is not up to date for recent OSs.

Athena has several features such as DDoS, click fraud, backdooring and botkiller options. The latter is used to kill other malware already present in the targeted host.

Unlike China-Z, Athena communicates over HTTP protocol and therefore represents a good candidate to show how Haka can handle protocols over layer 7. Moreover, Athena exhibits a large sets of C&C commands that are useful to monitor.

Figure 17 shows the decoding steps of Athena requests and responses (commands).

The bot contacts its gate through an encoded POST request. The request is made up of three fields, namely *a*, *b* and *c*. The first field is encoded twice (base64 + hex encoding). Once decoded, it is made up of two fields of the same length that serve as a mapping to decode the payload in *b*.

The field *b* encapsulates the payload. It is encoded in base64. The result is then transformed by applying the mapping derived

Figure 16: China-Z – Kibana dashboard.

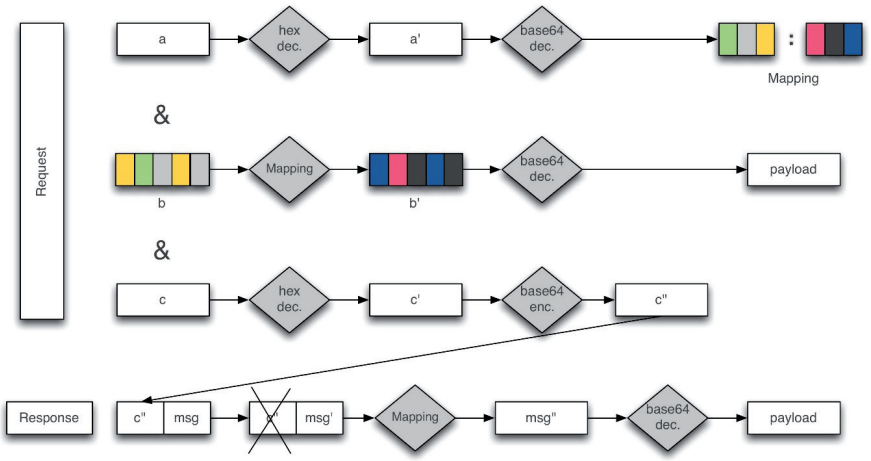

Figure 17: Athena – decoding requests and responses.

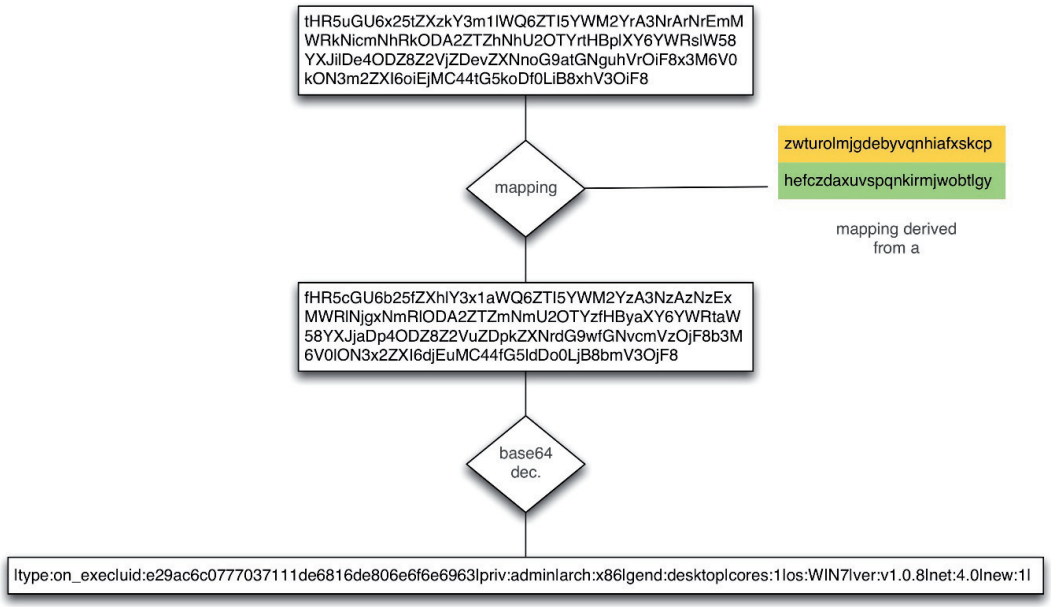

Figure 18: Athena – decoding request payload.

from field *a*. Figure 18 illustrates an example of a decoded payload.

Finally, the field *c* is hex-encoded. It is used as a data marker to match requests and responses. More precisely, its base64 form is appended to the C&C response.

The botnet replies with a data blob that starts with field *c* encoded in base64. The plain text is derived by applying first the mapping of field *a* and then decoding the result in base64. Figure 19 shows the format of a decoded response. A response consists of an update interval followed optionally by a set of commands. A command is made up of a *taskid*, a command to execute and a set of arguments.

l interval = 90 l
l taskid = 1 l !shell calc.exe l
l taskid = 2 l !download http://www.example.com/example.exe 1 l
..
l taskid = N l !command arg_1 arg_2 ... arg_n l

Figure 19: Athena – decoding commands.

3.2.2 Dissector and security rule writing

The specifications of the Athena protocol parser are pretty

```
athena_dissector.grammar = haka.grammar.new('athena', function ()
    SEP = token("&")
    EQUAL = token("=")

    post = record{
        field("name", token("[abc]")),
        EQUAL,
        field("value", token("[%%[:alnum:]]+"))
    }

    request = record{
        field("_a", post)
            :apply(function (self, res, ctx)
                local a = athena_utils.unquote(self.value)
                a = crypto.base64.decode(a)
                for k, v in string.gmatch(a, "([^:]+):(.+)") do
                    a1 = k
                    a2 = v
                end
                res._user.translator = athena_utils.maketrans(a2, a1)
            end),

        SEP,

        record{
            field("_b", post)
        }:extra({
            request = {
                get = function (res)
                    local b = athena_utils.unquote(res._b.value)
                    b  = athena_utils.translate(b, res._user.translator)
                    b = crypto.base64.decode(b)
                    return b
                end,

                set = function (res, value)
                    local b = crypto.base64.encode(value)
                    b = athena_utils.translate(b, res._user.translator, true)
                    b = string.gsub(b, "=", "%%3D")
                    b = string.gsub(b, "+", "%%252B")
                    res._b.value = b
                end
            }
        }),

        SEP,

        field("_c", post)
            :apply(function (self, res, ctx)
                res._user.garbage = athena_utils.unquote(self.value)
                res._user.garbage = crypto.base64.encode(res._user.garbage)
            end)
    }

    response = --[[
        ...
    ]]

    export(request, response)
end)
```

Figure 20: Athena – protocol specification.

```
local athena = require('protocol/athena')
local athena_utils = require('protocol/athena_utils')

athena.install_tcp_rule(80)

haka.rule{
    hook = athena.events.request,
    eval = function(athena, req)
        print("CnC Request")
        local r = athena_utils.cnc.split(req.request)
        debug.pprint(r)
    end
}

haka.rule{
    hook = athena.events.response,
    eval = function(athena, res)
        print("CnC Response")
        for _, v in ipairs(res.response) do
            r = athena_utils.cnc.split(v, '=')
            debug.pprint(r)
        end
    end
}
```

Figure 21: Athena – dumping C&C commands.

similar to the previously defined one. One has to define the grammar, the protocol state machine, and a set of events that will trigger the evaluation of the security rules.

In the following we will focus on the grammar of the Athena protocol. More precisely, we will discuss the specification of the request part. The full code is available in [6].

First, we define terminal tokens that will be used later to specify the request message. Then, we define a POST field as a *record* made up of a *name* and a *value* separated by an equal sign.

Finally, we define the request as a *record* made up of three POST fields separated by an '*&*' sign. We apply options on each of these fields. We enable extra computation on fields *a*, *b* and *c* in order to decode their content according to Figure 20.

In the sequel, we provide a set of security rules that could be enabled in conjunction with the Athena protocol parser.

The rules in Figure 21 are given for debugging purposes. They simply dump C&C commands on stdout.

As described earlier, Athena has uninstall capabilities. If we manage to substitute a C&C command with an *uninstall* command, then we can disinfect on-the-fly all compromised hosts monitored by Haka. The first rule alters a C&C command with an uninstall command and fakes the task identifier. The latter will be used in the second security rule to check if the infected host has succeeded in uninstalling the malware.

```
haka.rule{
    hook = athena.events.response,
    eval = function(self, res)
        res.response = {'|interval=2|', '|taskid=100|command=!uninstall|'}
    end
}

haka.rule{
    hook = athena.events.request,
    eval = function(athena, req)
        local r = athena_utils.cnc.split(req.request)
        local taskid = r['taskid'] or 0
        if taskid == '100' then
            haka.log("malware uninstalled successfully")
        end
    end
}
```

Figure 22: Athena – uninstall.

```
local INF, MAX = 0, 10
local time_elapsed = 0
local found = false
local sql = ""

haka.rule{
    hook = athena.events.request,
    eval = function(athena, req)
        if (MAX ~= INF) then
            local HALF = math.floor((INF+MAX)/2)
            sql = "' union select if(count(*) between " .. INF
                .. " and " .. HALF
                .. ",sleep(2),0) from botlist #"

            req.request = "|type:on_exec|uid:" .. sql .. "|"
            time_elapsed = haka.network_time().secs
            haka.log("trying %s injection", sql)
        else
            if not found then
                haka.log("Botnet made of %d bots", MAX)
                found = true
            end
        end
    end
}

haka.rule{
    hook = athena.events.response,
    eval = function(athena, response)
        time_elapsed = haka.network_time().secs - time_elapsed
        local HALF = math.floor((INF+MAX)/2)
        if (time_elapsed < 2) then
            INF = HALF + 1
        else
            MAX = HALF
        end
    end
}
```

Figure 23: Athena – SQL injection.

As a bonus, we provide an extra security rule that exploits an SQL injection vulnerability present in the Athena panel. The following rules exploit a time-based SQL injection by alerting C&C requests in order to determine the botnet size (Figure 23).

3.2.3 C&C traffic analysis

In the same way as for China-Z, we provide a *Kibana* dashboard to monitor Athena C&C activities (Figure 24). We can visualize all the commands available (update interval, downloads, DDoS, shell commands, etc.) and information about the infected host (OS, RAM usage, user, etc.).

In this example, the tracked botnet is used to force the user to view a gaming *YouTube* video in order to influence the number of views and hence enhance its credibility. The botnet master has placed a scam link in the comments section of the video.

4. RELATED WORKS

There are several security tools that provide a language to write security rules. Most of them are intrusion detection systems (e.g. Snort [8], Suricata [9]) that enable one to write basic vulnerability signatures. Intrusion detection and prevention systems are useful to monitor and prevent attacks, but usually (with the exception of Bro [6]) do not provide any means for writing protocol dissectors.

Numerous domain-specific languages have been proposed in the literature [4, 10–12] to specify protocols for parsing purposes.

Packettypes [10] is a language used to express binary packet format. It is quite useful for specifying lower protocol layers but fails to correctly handle application layer protocols, which are mostly ASCII-based.

Gapa [4] and Haka share multiple features. Both allow one to specify ASCII-based protocols and binary-based protocols. The specification covers the protocol and the security rules at the same time. The main difference lies in the separation of concerns between the protocol logic and its analysis. In Haka, events represent the glue between dissectors and security rules. Dissectors create events and then trigger them. As a result, all security rules hooking to these events will be evaluated. In contrast, users embed security controls while specifying the protocol in Gapa, which is not convenient for code readability and reusability, especially for a large set of security checks. Finally, Gapa leverages an interpreted C-like language that is less familiar to non-developer experts.

Binpac [12] and Haka target the same features with some slight differences: first of all, Binpac focuses only on protocol parsing. It was designed as a separate language to be used in conjunction with a third network traffic analysis tool such as Bro [6] while Haka was designed as a self-contained system that applies a protocol parsing description on live captured traffic. Second, Binpac grammar does not provide any constructs to handle protocol states and transitions. Finally, the Binpac specification makes use of C/C++ code to deal with some parsing issues.

Meiners *et al.* stated in [11] that their protocol field extraction framework, Flowsifter, generates faster protocol parsers. Unlike previous solutions, Flowsifter leverages a formal grammar that requires a substantial effort to fully specify a protocol.

5. CONCLUSION

Analysing and monitoring malware networking is an important step during threat intelligence analysis. In this paper, we have proposed the use of Haka to supplement the malware analyst's toolset.

In future work, we plan to provide a repository of malware dissectors in order to help the community to track the C&C traffic of different malware families.

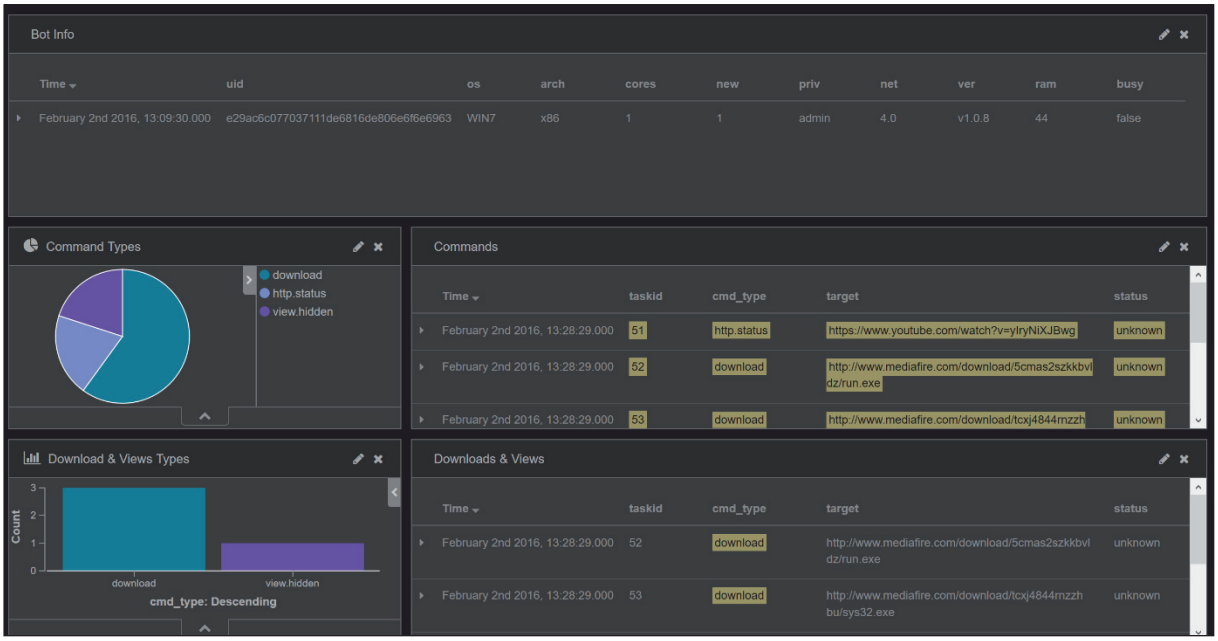

Figure 24: Athena – Kibana dashboard.

6. ACKNOWLEDGEMENTS

We thank the developers of the Haka project, namely Pierre-Sylvain Desse and Paul Fariello, for their active support throughout this project. We also thank Fayçal Daira for reviewing this paper.

REFERENCES

[1] HAKA. Haka documentation. http://doc.haka-security.org/.

[2] HAKA. Haka source code. https://github.com/haka-security.

[3] LUA. The programming language lua. http://www.lua.org/.

[4] Borisov, N.; Brumley, D. J.; Wang, H. J.; Guo, C. Generic application-level protocol analyzer and its language. In NDSS07 (2007).

[5] New ELF malware on Shellshock: The ChinaZ. http://blog.malwaremustdie.org/2015/01/mmd-0030-2015-new-elf-malware-on.html.

[6] BRO. The bro network security monitor. http://www.bro.org/.

[7] Athena Botnet Shows Windows XP Still Widely Used. https://blogs.mcafee.com/mcafee-labs/athena-botnet-shows-windows-xp-still-widely-used/.

[8] SNORT. The open source network intrusion detection system. http://www.snort.org/.

[9] SURICATA. High performance network IDS, IPS and network security monitoring engine. http://www.suricata-ids.org/.

[10] McCann, P. J.; Chandra, S. Packet types: Abstract specifications of network protocol messages. In SIGCOMM (2000), pp.321–333.

[11] Meiners, C. R.; Norige, E.; Liu, A. X.; Torng, E. Flowsifter: A counting automata approach to layer 7 field extraction for deep flow inspection. In INFOCOM (2012), A. G. Greenberg and K. Sohraby, Eds., IEEE, pp.1746–1754.

[12] Pang, R.; Paxson, V.; Sommer, R.; Peterson, L. binpac: A yacc for writing application protocol parsers. Proceedings of the 2006 Internet Measurement Conference (2006).