The Security Analyst's Guide to Suricata

By Éric Leblond & Peter Manev

STAMUS NETWORKS

The Security Analyst's Guide to Suricata

Published by Stamus Networks
450 E. 96th Street, Suite 500
Indianapolis, IN 46240

ISBN: 979-8-9871510-1-3

CONTENTS:

PREFACE

We are pleased to present the industry's first open-source book on the world's most popular open-source network security engine, Suricata. The idea for this book emerged after it became obvious to us that many security practitioners using Suricata either struggle to effectively use the most powerful capabilities of the tool or simply don't realize they exist.

Each year, we speak at many industry conferences and train hundreds of users in workshops on behalf of the Open Information Security Foundation (OISF). In our engagements with the audience at these events, we have noticed that users share the common perception that Suricata is a classic signature-based intrusion detection system (IDS), albeit a powerful and high-performance one. Most fail to realize that the Suricata engine can also simultaneously produce protocol transaction logs and flow records that are correlated with the IDS alerts. These can be incredibly powerful for security analysts during an incident investigation or a threat hunt. And they can be even more powerful for the development of anomaly detection using Suricata.

So we decided to write a simple book to introduce the most powerful features and concepts developed in Suricata over its 12-year history.

As you may be aware, there is a dedicated team of Suricata developers continuously working to improve Suricata and releasing new capabilities regularly. So, we decided to take a more open-source software development approach to the content and release cadence of the book in order to keep it relevant and up-to-date.

The book is structured as a loose collection of chapters, each focused on a single subject area, such as Suricata rule writing or TLS detection and threat hunting. All its content is developed and managed on a GitHub repository[1] and is open to all who wish to comment or contribute ideas. Readers who are looking for a simple text edition may access all content there. Of course, we also package the book in PDF and eReader format for those who prefer source 'code' of the book. And we hope to offer a printed version soon.

The open-source format makes it a living book that will grow and evolve over time with ongoing input from the authors as well as contributions and feedback from the Suricata community.

We would like to thank everyone at Stamus Networks for their support during the making of this book. And this book would not have been possible without the help of the amazing team at OISF.

Note: This book is not meant to act as a replacement for the Suricata manual, which is an excellent reference tool for those installing and deploying the platform. Instead, The Security Analyst's Guide to Suriata was written for the SOC analysts and threat hunters who have been tasked with effectively defending their network using Suricata. We aim to provide vital information on entry points and in-depth coverage for the most important Suricata features.

We welcome your feedback. Enjoy.

[1] https://github.com/StamusNetworks/suricata-4-analysts

CHAPTER

ONE

INTRODUCTION TO SURICATA

1.1 An Open Source Network Threat Detection engine

Suricata is an open source intrusion detection (IDS), intrusion prevention (IPS), and network security monitoring (NSM) system. It is developed and maintained by a vast community under the guidance of the Open Information Security Foundation (OISF). The project started in 2009, and had its first official release in 2010.

The original goal of the Suricata IDS project was to develop an intrusion detection engine based on signatures, similar to its ancestor Snort but with different technological choices. The aim was to build a network IDS that would share the same detection language as Snort and have a strong focus on community. The early technology choices were to implement multi-threading, advanced HTTP support, and a port-independent protocol recognition.

Over the following dozen years or so, the project—made possible thanks to public funding—has continued to evolve. Overseeing the evolution of the platform is the OISF[2], a non-profit organization created to receive the funds and take care of promoting and organizing Suricata's growth.

Most of the funding for the project now comes through private organizations via consortium membership. The foundation and the consortium members play a significant role in advancing the technology, but Suricata development has always been and remains a community undertaking.

People outside the foundation have made revolutionary proposals, which have profoundly changed the face of the project. They have helped Suricata to evolve over the years in order to stay current, attractive, and focused on threat detection. The timeline below illustrates some of the major Suricata milestones.

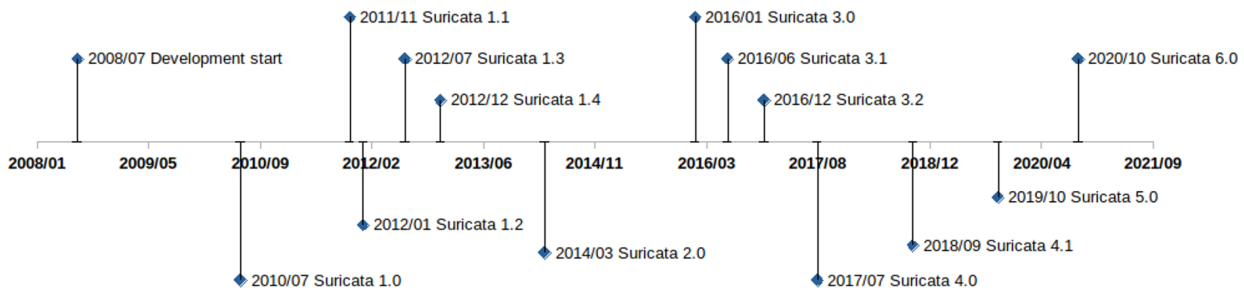


Fig. 1.1: Timeline of major Suricata versions.

[2] https://https://oisf.net/

1.2 12 years of innovation

1.2.1 Suricata 1.0 (July 2010) - Welcome to the HTTP World

Understanding the HTTP protocol was by far the most important breakthrough in the first release. Suricata 1.0, published in July 2010 after two years of development, was able to read a Snort ruleset but could use a series of new keywords to look for content in the protocol fields of HTTP using a port-agnostic approach.

For the first time, a signature could ask for a specific protocol field without having to do the protocol parsing by itself. Because of this, the complexity of the writing of signatures was decreased while, at the same time, performance was increased. Techniques such as multi-pattern matching enabled accelerated searches in these specific fields.

Another important feature of Suricata 1.0 was protocol recognition. The engine analyzes the beginning of the exchange on a stream to find out what protocol it is - completely independent of the Layer 4 port. This port-agnostic feature had a big impact in terms of detection rate, as a lot of malware at that time was using a high numbered port to connect to command and control servers and HTTP to exchange information. By being able to find HTTP independently of the port - a big accomplishment - it allowed Suricata to accurately detect the malware.

Suricata also offered multi-step detection thanks to the inclusion of keywords that were the first step toward overcoming the low expressivity of the signature language inherited from Snort.

For example, the "flowbits" keyword family provided a way to pass information between signatures, and thus allow users to construct a state engine. While it was limited to the description of the state inside a single flow, it was real progress.

One other feature of this first release broke the previously strict definition of what an IDS was: logging HTTP requests to a file. This was not in the initial specifications, but it turned out to not be too complex to build and did not have a major impact on performance. This opportunistic approach continues to uniquely define all Suricata development.

1.2.2 Suricata 1.2 (January 2012) - File Extraction

File transaction info was added with version 1.2, which was released in January 2012, and then was extended in version 1.3 which released six months later. Understanding the HTTP protocol gave Suricata the visibility to see what was transmitted in requests, so it was natural to perform an extraction of the transmitted files.

This was added in version 1.2, along with file checksum computation and file transaction logging. In Suricata 1.3, the keyword "filemd5" was added to verify if the md5 checksum of the transmitted file was present in a list stored in a file. The feature would be extended later to sha1 and sha256, with the "filesha1" and "filesha256" keywords.

Note: extraction of files using SMTP protocol was contributed by BAE Systems in Suricata 2.1.

1.2.3 Suricata 1.3 (July 2012) - Transport Layer Security (TLS)

In July 2012, Suricata 1.3 was released and with it came support for TLS – contributed by Pierre Chifflier working for ANSSI, the French agency responsible for cyberdefense. This TLS implementation does not include decryption but is instead an analysis of the TLS handshake with extraction of unique transaction characteristics such as certificate subject, issuer, and its fingerprint.

At this stage, it is clear that Suricata is shifting away from the classic role of IDS as the presenter of simple data. The system is embracing complex decoding and extracting data that is simply not visible to the naked eye. This began with HTTP message decompression and continued from this point forward.

This TLS support is now used to address the evolution of malware that began to use encrypted communication. For example, currently available signatures now readily detect connections to servers using default OpenSSL configurations.

Dedicated TLS keywords were also added with this release, and all TLS events are logged into a dedicated file.

This mixed approach - using both IDS and network security monitoring (NSM) at the same time - builds upon what was done with HTTP and will prove to become the standard going forward for each new protocol supported: adding the dynamic protocol identification, logging events, offering dedicated keywords, and extracting files.

The evolution of TLS support has continued over multiple versions of Suricata.

1.2.4 Suricata 1.4 (December 2014) - Support for Lua

With the release of Suricata 1.4 in December 2014, Suricata added a second major signature language in Lua, a lightweight, multi-paradigm programming language designed primarily for embedded use in applications.

Signatures could now include a Lua script as a feature. This script uses Suricata-exposed buffers such as the packet content or the TLS information, and its return value is 1 for a match and 0 for no match. The Lua script may also create or modify flowbits variables.

With this added capability, Suricata now had a real programming language that could be used by the system to save states. This opened up a range of possibilities. The Lua support, for instance, could be used to write a very accurate signature to detect Heartbleed attack attempts. In fact, that signature was available a few hours after an attack was announced, and it would be the only IDS signature-based approach to provide accurate detection of Heartbleed.

Unfortunately, Lua support did not have the success that the development team had expected - and for a trivial reason. In order to be evaluated with the signature, the Lua script for a signature must be inserted as a file next to the signatures file. But adding this type of file was not supported by the existing signature/rule management tools, and no major threat research organizations distributed signatures with Lua for this simple reason. Interest in Lua still exists today, and the increased activity around signature management tools means there is still some hope for the Lua signatures.

1.2.5 Suricata 2.0 (March 2014) - Welcome JSON

Suricata 2.0 was published in March of 2014, marking a major milestone in the evolution of Suricata. This came with the addition of JSON as the preferred format for Suricata-generated events. Thanks to this contribution from Tom Decanio, the project was finally leaving the dated format of the 1990s as JSON replaced the non-structured text format or binary format such as that seen in unified2. JSON provided an easy-to-extend and easy-to-use format for all Suricata events.

Thanks to JSON formatting, sending Suricata-generated data to tools such as the Elastic stack or Splunk was easy to do. Suricata 2.0 came with a native "correlation" capability that can be made using the name of the fields used. A source IP is always the "src_ip" field. On top of that, all events can now be found in one file (by default), containing, for example, different types of logs and alerts and/or separate DNS, SSH, TLS, HTTP transactions, and even performance data.

On the intrusion detection side, having an alternative to the unified2 format was a big improvement. This binary format dedicated to alerts only supported IP fields, the payload, and basic information about the signature. Unfortunately, it was almost impossible to extend it to add more contextual information to the alerts.

Because Suricata now supported more protocols, it was possible to add contextual information to alerts. Being able to look at the extracted fields and run statistics on them has the potential to make the job of the analyst simpler and more efficient.

For Suricata 2.1, this philosophy was embraced more completely, by adding application layer metadata in the alerts, starting with HTTP. The work on this feature continued throughout the release, and metadata was added for many other protocols. Later, in version 4.0, this logic was pushed further by adding the logging of the HTTP body. These fields are often compressed, so logging the content was not directly useful. Providing the decompressed data did, however, allow for direct analysis.

```
"tls" : {
  "ja3s" : {
    "hash" : "f47b284bf7f61821a407e4f140a02686"
    "string" : "771,49172,65281-11"
  }
  "serial" : "00:9B:53:84:91:9E:52:B0:70"
  "subject" : "C=XX, ST=1, L=1, O=1, OU=1, CN=*"
  "fingerprint" : "5b:14:93:e9:8f:c8:e7:7a:2e:a9:69:34:b3:da:83:b3:21:83:b1:9c"
  "version" : "TLS 1.2"
  "issuerdn" : "C=XX, ST=1, L=1, O=1, OU=1, CN=*"
  "notbefore" : "2019-07-04T14:07:58"
  "ja3" : {
    "agent" : [
      0 : "Tofsee (from abuse.ch)"
    ]
    "hash" : "4d7a28d6f2263ed61de88ca66eb011e3"
    "string" : "771,60-47-61-53-5-10-49191-49171-49172-49195-49187-49196-49188-49161-49162-64-50-106-56-19-4,65281-0-10-11-13,23-24,0"
  }
  "sni" : "c54rng3686.com"
  "notafter" : "2029-07-01T14:07:58"
}
```


Fig. 1.2: TLS event in JSON form.

1.2.6 Suricata 3.0 (January 2016) - Debut of Xbits Keyword

Suricata 3.0 was published in January 2016, with the primary new feature being the "xbits" keyword. The concept of xbits is to go beyond the limitations of flowbits, which could not be used in multi-flow attacks. Xbits is an evolution of flowbits, in which the variable is attached to an IP address or to an IP pair. Signatures can then collaborate inside a state machine that is not limited to a single flow.

1.2.7 Suricata 4.0 (July 2017) - In Rust we Trust

In addition to support for a number of new protocols, Suricata 4.0 introduced a more secure and efficient common parsing technique into the core. Using a combination of the Rust language and Nom parser (see https://github.com/Geal/nom), it set the stage for the rapid increase in the protocols supported by Suricata without sacrificing security and stability of the engine. This will prove critical for paving the way for the complete NSM functionality.

On the functional side, Network File System (NFS) and Network Time Protocol (NTP) were the two big protocol additions in version 4.0.

Support for several other new protocols - specifically Server Message Block (SMB) and Dynamic Host Configuration Protocol (DHCP) - was added in release 4.1. These are mainly used in internal networks and with their support Suricata can more effectively analyze internal traffic, providing two primary benefits: primarily, increased visibility in encrypted environments; secondly, providing more complete detection of threats as they move laterally within a network.

1.2.8 Suricata 4.1 (December 2018) - Samba Time

The major highlight of Suricata 4.1 was the support for the SMB protocol family. Complete protocol support was added, including dedicated keywords, metadata logging, and file extraction. The impact on the deployment of Suricata on internal traffic has been quite huge. The metadata records are complete and enable the creation of a fine-grained analysis strategy. The following event is an example of a transaction on a share:


```
"smb": {
  "id": 3,
  "dialect": "2.10",
  "command": "SMB2_COMMAND_TREE_CONNECT",
  "status": "STATUS_SUCCESS",
  "status_code": "0x0",
  "session_id": 4398046511121,
  "tree_id": 1,
  "share": "\\\\admin-pc\\c$",
  "share_type": "FILE"
}
```


Fig. 1.3: SMB sub object in an smb event.

1.2.9 Suricata 5 (October 2019) - Introduction of Datasets

The introduction of datasets was the primary enhancement included with Suricata version 5, released in October 2019. This added the ability to match on a list of more than 50 different buffers and check a list of hostnames against a "known bad" database in the HTTP hostname, TLS Server Name Indication, or an HTTP user agent list.

It is important to note that these lists may include anywhere from a few items to millions of them without degrading the system performance. This is a key feature, considering the trend toward threat intelligence sharing and the use of tools such as MISP.

Another interesting aspect of datasets is Suricata's capability to add and delete elements from a set by triggering changes with signatures. This feature has, for example, been used to create a learned list, tracking what is seen on the network and when and to build a new class of machine-learning based detection.

1.2.10 Suricata 6 (October 2020) - Additional Protocol Support

The primary contribution of Suricata 6 increased the body of supported protocols. From a user perspective, the introduction of HTTP/2 support was critical. Given that almost half of the top 10 millions websites are supporting this protocol, it was essential for Suricata to be able to log HTTP/2 protocol transactions and run threat detection on it.

This version also added support for other important protocols, including Message Queuing Telemetry Transport (MQTT, contributed by DCSO) for Internet of Things (IoT) environments and Remote Frame Buffer (used for remote desktop sessions).

Although it was an "under the hood" feature, the switch to an internally developed JSON generator in Suricata 6 is worth mentioning. With users deploying Suricata in 100 Gbps environments and with application layer logging being an important feature, the number of events per second generated can be quite high. For example, it is not uncommon for a 100 Gbps deployment to generate hundreds of thousands alert events per second on a single probe. As a consequence, the generation of JSON events using the original libjansson library ended up being a bottleneck. With Suricata 6, this was replaced by a custom JSON generator written in Rust which significantly lowers the performance burden of logging.

CHAPTER

TWO

SURICATA ECOSYSTEM

Some tools will be used throughout the document. They are part of the central tooling around Suricata.

2.1 JQ

JQ[3] is a command line tool that allows users to format, search, and modify JSON objects.

2.2 Elastic stack

The Elastic stack[4] is a software suite that implements a distributed NoSQL database (Elasticsearch) with a visualization interface (Kibana) and a log ingestion tool (Logstash). There are other components in the stack that will not be covered here.

Some useful Kibana dashboards have been published by Stamus Networks on Github[5].

2.3 Splunk

The Splunk[6] platform is a search, analysis, and visualization engine that features a really powerful query language.

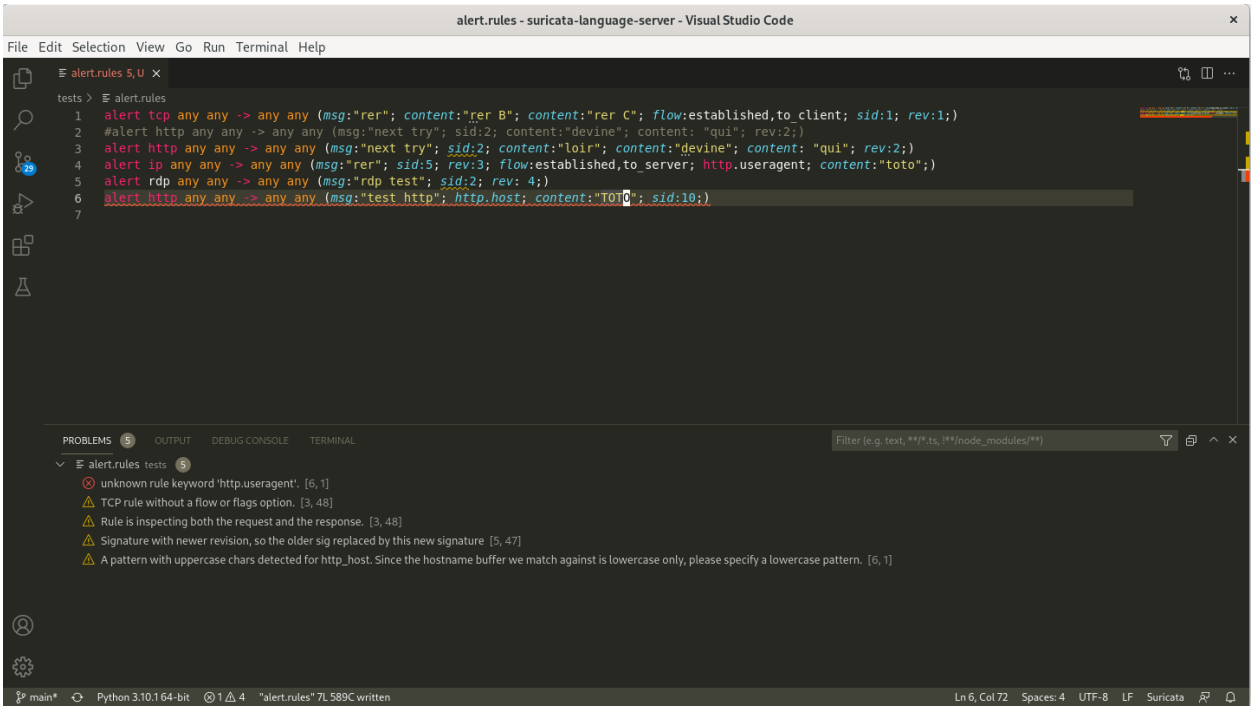
If you are a Splunk user you may want to get a look at the Stamus Networks app for Splunk[7] that provides ready to use dashboards and reports for Suricata and Stamus Networks users.

2.4 Suricata Language Server

The Suricata Language Server is an implementation of the Language Server Protocol for Suricata signatures. It adds syntax checks and hints as well as auto-completion to your preferred editor once it is configured. Information displayed in the editor is highly valuable when writing Suricata signatures as it ensures the rules syntax is correct while providing hints about writing performant rules.

Editors that are known to support the Suricata Language Server are Neovim, Visual Studio Code, Sublime Text 3, and Kate, but any editor supporting the Language Server Protocol should also support the Suricata Language Server.

[3] https://stedolan.github.io/jq/
[4] https://www.elastic.co/
[5] https://github.com/StamusNetworks/KTS7
[6] https://splunk.com
[7] https://splunkbase.splunk.com/app/5262/



The Suricata Language Server currently supports auto-completion and advanced syntax checking. Both features use the capabilities of the Suricata deployment available on the system. This means that the list of keywords (with documentation information) and the syntax checking both come from Suricata itself. While this comes at the cost of Suricata needing to be installed on the system, it also guarantees a strict check of signatures with respect to the version of Suricata you are running. Pushing signatures to production will not return a bad surprise as the syntax has already been checked by the same engine.

Syntax checking is completed when files are saved. A configuration test is started using Suricata, in turn providing errors to the diagnostic. Warnings and hints are also provided by using Suricata's detection engine analysis. This analysis can return warnings and hints about potential issues seen within the signatures.

You can get the Suricata Language Server[8] from GitHub.

[8] https://github.com/StamusNetworks/suricata-language-server

CHAPTER

THREE

PROTOCOL INDEPENDENT THREAT HUNTING

3.1 Threat Hunting with IDS and NSM data

Suricata is both an IDS and an NSM tool. It will extract and generate protocol transaction logs independently of alerts. As a result, the threat hunter has the responsibility of finding the types of events where searching for results on the data makes the most sense.

Let's take two examples of an Indicator Of Compromise (IOC):

- an SMB user name created by a threat actor when he has taken control of an Active Directory. Let's say this username is 'pandabear'.
- a domain that is an uncommonly used cloud provider. Let's say the domain is 'sovereigncloud.eu'

What these IOCs have in common is that first thing the threat hunter must do is query the NSM data to see if the IOC has been seen in the network.

For 'pandabear', we can do two queries (using Splunk syntax), one to match in the SMB logs and the other one in the Kerberos logs:

- *event_type=smb AND smb.ntlmssp.user=pandabear*
- *event_type=krb5 AND krb5.cname=pandabear*

For the domain, we can do queries on DNS (looking for the query), TLS (one Server Name Indication), and HTTP (looking for the hostname):

- *event_type=dns AND dns.query.rrname=sovereigncloud.eu*
- *event_type=tls AND tls.sni=sovereigncloud.eu*
- *event_type=http AND http.host=sovereigncloud.eu*

In the first example, we are really in trouble if the IOC is seen in the organization because the stage of the compromise is advanced, and because 'pandabear' is not likely a regular user. In the second, seeing the IOC is just an indicator because we can have users of this cloud provider, causing a need to discriminate among them further by doing more investigation.

For the domain, a regular check of the NSM data may be enough. For the username, on the other hand, we may want to make the switch to incident response much faster. Adding IDS signatures to detect this username if ever it appears may be a good solution:

```
alert smb any any -> $HOME_NET any (msg:"pandabear"; smb.ntlmssp_user; content:"pandabear
↪"; ...
alert krb5 any any -> $HOME_NET any (msg:"pandabear"; krb5.cname; content:"pandabear"; ..
↪.
```

Please note that the first signature will require Suricata 7.0 and that dataset is a far better way to match IOCs with Suricata signatures.

To summarize this example, because Suricata is both an IDS and an NSM, there are multiple complementary approach options when threat hunting with Suricata.

3.2 Correlation using flow_id

Suricata performs flow tracking over most TCP/IP protocols. In the case of TCP, this is a direct mapping of flows to TCP sessions. For UDP, this is completed by looking at the IP information (source IP, port and destincation IP, and port) and applying a timeout logic.

So a flow tracks what is happening during a communication between a client and a server:

All IP events contain a 'flow_id' key that is the same for all events in a single flow. This allows a user to to group all events.

An example seen in jq on a simple HTTP request. You can also see here that jq is used to reformat the events:

```
jq 'select(.flow_id==1541199918082444)|{"time": .timestamp, "type": .event_type, "src_ip
↪":.src_ip, "src_port": .src_port, "dest_ip": .dest_ip, "dest_port": .dest_port}' -c␣
↪eve.json
{"time":"2017-07-24T15:54:12.716673+0200","type":"http","src_ip":"10.7.24.101","src_port
↪":49163,"dest_ip":"216.239.38.21","dest_port":80}
{"time":"2017-07-24T15:56:28.177134+0200","type":"fileinfo","src_ip":"216.239.38.21",
↪"src_port":80,"dest_ip":"10.7.24.101","dest_port":49163}
{"time":"2017-07-24T16:15:05.777324+0200","type":"flow","src_ip":"10.7.24.101","src_port
↪":49163,"dest_ip":"216.239.38.21","dest_port":80
```

We have three events here:

- an HTPP request
- a file information event (analysis of the data of the transferred file)
- a flow entry containing the packets and bytes accounting as well as the duration of the flow

The flow event is generated once the flow is timed out by Suricata.

Some flows can have a lot more events if the protocol (like SMB) is doing a lot of transactions on a single flow.

3.3 Learning datasets

At first look, the dataset[9] feature belongs to the IDS world (see *Matching on IOCs* for example) as it provides matching on a list of elements. But 'dataset' can be enriched from the packet path, meaning it can be used to store the never-before-seen metadata.

For example, to collect all internal HTTP user agents:

```
alert $HOME_NET any -> any any (msg:"new agent"; http.user_agent; \\
  dataset:isset,http-ua,type string, state /var/lib/http-ua.lst; \\
```

[9] https://suricata.readthedocs.io/en/latest/rules/datasets.html

Every time Suricata will detect an HTTP user agent that has never been seen on the network by Suricata, it will trigger an alert. These events can be used to build a list of previously unseen items for all the fields that can be matched with a sticky buffer.

In our signature, the file '/var/list/http-ua.lst' is used to store the state. Suricata will dump the contents of the list it built into memory to the file (in our case, as a base64 string). This ensures that no new events will be generated if Suricata is forced to restart.

CHAPTER

FOUR

IDS FEATURES

4.1 Suricata rule language

Suricata rule language is derived from Snort rule language from 2010 and it has since evolved to become a separate language sharing a common root.

Suricata documentation[10] is very comprehensive with regards to signature language and keywords and should be considered the ultimate reference guide.

4.2 Anatomy of a signature

A signature has 3 parts:

- A keyword for the action: alert, drop, pass, reject
- IP options to indicate the characteristics of the IP flow
- Match and information for the signature

Let's see an example:

```
alert http any any -> any any (msg:"http"; \
  http.host; content:"suricata.io"; \
  sid:1; rev;1)
```

Here, Suricata will generate an alert when there is a flow where the HTTP application layer has been identified and when the HTTP host in the request contains `suricata.io`.

`msg` is the text that will be used as message in the alert event.

The `sid` keyword is the identifier of the signature (must be unique in the ruleset) and `rev` is the version of the signature.

Let's take a more complete example where we want the flow to be from the internal network (identified by the variable '$HOME_NET') to the outside world (identified by the variable '$EXTERNAL_NET') and with destination port `8080`:

```
alert http $HOME_NET any -> $EXTERNAL_NET 8080 (msg:"http"; \
  http.host; content:"suricata.io"; \
  sid:1; rev;1)
```

[10] https://redmine.openinfosecfoundation.org/projects/suricata/wiki/Suricata_Rules

4.3 Suricata rule keywords

4.3.1 Types of keywords

There are 3 types of matching keywords:

- Sticky buffer keywords: the preferred type for performance and readability
- Content modifier: they set the context to the previous content match
- The keyword value: simple content match on a field

It is recommended to only use sticky buffer keywords in newly written rules.

4.3.2 Sticky buffer keywords

The sticky buffer keyword sets the context for the next content matches. For example:

```
http.host; content:"www"; content:"toto"; pcre:"/toto.[com|org]$/"; \
http.method; content:"GET";
```

In this case, the host field in the HTTP header will match `www` and `toto` (via the content keywords) and do a regular expression match to detect the domains. Then there is a switch of context to the HTTP method and a match on GET when the method is done.

4.3.3 Content modifiers keywords

The content modifier keywords alter the context of the previous content keyword. As a result, the keywords need to be repeated. So if we want to implement the previous example we will need to have:

```
content:"www"; http_host; content:"toto"; http_host; pcre:"/toto.[com|org]$/W"; \
content:"GET"; http_method;
```

Please note that in addition to the repetition of the keyword a modifier (`W` in this example) has been added to the regular expression match to indicate that the match has to be done on the HTTP host.

4.3.4 Getting keywords from Suricata

You can use the following commands:

```
suricata --list-keywords
=====Supported keywords=====
- sid
- priority
- rev
- classtype
- app-layer-protocol
```

Information about a specific keyword can be obtained via:

```
suricata --list-keywords=http.host
= http.host =
Description: sticky buffer to match on the HTTP Host buffer
Features: No option,sticky buffer
Documentation: https://suricata.readthedocs.io/en/latest/rules/http-keywords.html#http-
↪host-and-http-raw-host
```

And a full export of the keywords in CSV format can be generated with:

```
suricata --list-keywords=csv
name;description;app layer;features;documentation
sid;set rule ID;Unset;none;https://suricata.readthedocs.io/en/latest/rules/meta.html#sid-
↪signature-id;
priority;rules with a higher priority will be examined first;Unset;none;https://suricata.
↪readthedocs.io/en/latest/rules/meta.html#priority;
rev;set version of the rule;Unset;none;https://suricata.readthedocs.io/en/latest/rules/
↪meta.html#rev-revision;
classtype;information about the classification of rules and alerts;Unset;none;https://
↪suricata.readthedocs.io/en/latest/rules/meta.html#classtype;
```

CHAPTER

FIVE

PRACTICAL RULE WRITING

5.1 Methodology

There are a few techniques that greatly improve the rule writing experience.

5.1.1 Use a PCAP file

Writing a rule is an iterative process, so it is easier to write the rule using a PCAP file that can be replayed multiple times instead of doing it on live traffic.

So, try to capture a PCAP trace of the behavior you want to inspect, then you can replay it when your signature needs to be tested.

To replay the pcap, you can use something like (create data directory first)

```
rm data/eve.json
suricata -r ./trace.pcap -l data/
cat eve.json | jq 'select(.alert.signature_id==1000000)'
```

if your signature ID is 1000000.

The 1000000-1999999 range is reserved for internal usage, so it is a good choice. Contact the Sid Allocation project[11] if you want to publish your rules publicly.

5.1.2 Replay with only your rules file

To speed up the writing of a rule, you need tests to be fast. The -S flag is here to help. Suricata will only load the rules in the file provided after the option. As a result, the run will take only a few seconds instead of 30 seconds or more if Suricata needs to build a complete detection engine.

With this option, the testing process becomes

```
rm data/eve.json
suricata -r ./trace.pcap -l data/ -S ./my.rules
cat eve.json | jq 'select(.event_type=="alert")'
```

[11] https://sidallocation.org/

5.1.3 Add IP filtering in later stage

It is better to write a signature starting with 'any any -> any any' then add a filter like '$HOME_NET any -> $EXTERNAL_NET any'. The source and destination IP depends on the signature and the HOME_NET may not be correctly defined with regards to the data in the PCAP file. The result is that the signature might just not match because of that and not because of a complex regular expression you added in the signature.

5.2 Writing a rule - step by step

The following is a suggestion for a process to use when writing signatures:

5.2.1 Get a pcap file

First step is to get a PCAP file with the content you want triggering the rule. Don't hesitate to filter out things in the pcap. For example, if you want to match on a single flow you can do something like

```
tcpdump -r input.pcap -w work.pcap port 53535 and port 443
```

where 53535 and 443 are the source and destination ports of the flow you want to match on. You can also add a few 'host' filters in the BPF if the previous command returned more than one flow.

Now we can use the file 'work.pcap' for our tests.

5.2.2 Run the file inside Suricata

By running Suricata without any rules on the file, we can extract all the metadata seen by Suricata:

```
rm data/eve.json
suricata -r ./trace.pcap -l data/ -S /dev/null
# explose data/eve.json
```

In most cases, it will be good enough to get an idea of what fields we should have matched on. As the data is coming from Suricata itself, the string will be exactly what we should use in the signature.

If you need more inspection, you can use Wireshark[12] to do so. You can also see Suricata data in Wireshark by using Suriwire[13].

5.2.3 Write your signature

We highly recommend using a text editor supported by the *Suricata Language Server* for editing. Using the editor with the Suricata Language Server extension allows you to easily identify errors and take advantage of auto-completion. During the writing phase, this is easier to have a file containing a single signature.

We can then test if the rule is alerting by running:

```
rm data/eve.json
suricata -r ./trace.pcap -l data/ -S my.rules -v
cat eve.json | jq 'select(.event_type=="alert")'
```

[12] https://www.wireshark.org/
[13] https://github.com/regit/suriwire

The last command may not even be necessary. This is because by adding '-v' we will have the number of alerts at the end of the output.

```
[9093] 9/8/2022 -- 23:50:47 - (counters.c:871) <Info> (StatsLogSummary) -- Alerts: 1
```

If you are not using the *Suricata Language Server*, you need to do an engine analysis with Suricata to get warnings and performance hints on the signature:

```
suricata --engine-analysis -l data/ -S my.rules -v
cat data/rules_analysis.txt
```

As mentioned before, the easiest approach is to get an iterative approach here:

- Start with a simple content match on one of the sticky buffer keywords
- Add some more contents match if needed
- Complete with a regular expression if needed
- Set up the variable for the IPs (HOME_NET, EXTERNAL_NET for example)
- Add the metadata keyword for more usable data

Between each step, run Suricata to verify that your output is correct.

See the chapter *Write performant Suricata rules* for more details and explanation on the steps described above and especially the *Performance improvement process* section.

CHAPTER

SIX

WRITING PERFORMANT SURICATA RULES

6.1 Suricata detection engine optimizations

6.1.1 The detection engine optimization challenge

In demanding enterprise environments, Suricata must operate at very high network speeds – often between 40Gbps and 100Gbps – with the full ETPro ruleset loaded. That ruleset is approximately 60,000 signatures, and in order to keep up with line rate, Suricata must inspect all those packets at a rate of 3,333,333 packets per second (when operating at 40Gbps).

So, at 40Gbps there is a budget of .000000000005 seconds per rule. And in this .005 ns per rule, Suricata must do protocol analysis, content matching, and execute regular expressions.

In a typical 3GHz CPU, we have a CPU cycle of 3 ns. As a result, using a brute force approach in the detection engine is 3 orders of magnitude too little, even if a test takes only a single cycle.

Thus, some serious optimizations are needed. Scaling via multithreading to use all cores on the system is a key point here, and Suricata does this very well. But even on a one hundred core system, it will only lead to a 100 factor improvement, and this still leaves us an order of magnitude below the bare minimum needed for the task.

Running load balancing on the CPU is incredibly important, but we still cannot address the 60,000 rules. In this case, we would need to reduce the number of rules processed. Unfortunately, running fewer rules will reduce the threat coverage, so we need a better solution.

6.1.2 Grouping signatures

This initial approach is quite simple: why should we evaluate a rule on a UDP flow if we are currently inspecting a TCP packet? By doing a protocol split, we can, in a perfect case, divide the number of signatures to evaluate by two.

While we are at it, we can group signatures by protocol port, group network parameters into a tree, and place groups of signatures in the leafs.

This is an interesting first step, but I'm sure some readers are already concerned about the fact that everything in their network is HTTP or TLS. Thus, they have only 2 used groups.

Something else is needed.

6.1.3 Multi pattern matching

Since we can not differentiate on the IP parameters, we need to go higher in the protocol stack to complete the task; however, an alert can match on an HTTP user agent or on file data transferred over SMB. Given the complexity of the fields we are matching on, we cannot do an implementation of the tree.

So let's take one step back. In this case, we are pattern matching on one buffer (HTTP user agent, file data, etc…) and would have a wonderful increase in performance if we could have an automatic tree built up for the patterns we are looking for on this buffer.

This type of algorithm is named multi pattern matching (MPM) and the most famous implementation is called Aho–Corasick algorithm[14].

This method allows for a really effective split of signatures.

First, Suricata separates the signatures by IP parameters. Then, it looks at the fast pattern buffer (which has been selected for use with the multi pattern algorithm). There can be only one buffer in order to guarantee a perfect partition of the ruleset. Once the MPM algorithm has returned, there will be only a small subset of signatures to evaluate. Ideally, if the pattern is well chosen, Suricata could have just a single signature to evaluate.

Let's use this signature as example

```
alert http any any -> any any (msg:"Bad Agent"; http.user_agent; content: "Winhttp";␣
↪fast_pattern; startswith; pcre:"/^Winhttp [0-9]+\/[0-9]+/"; sid:1;)
```

The evaluation of this signature by Suricata will be as follows:

It will be attached to the set of signatures that have the HTTP user agent as the fast pattern buffer. As a result, the *Winhttp* content match will be evaluated during the MPM phase with all the other matches. One pass algorithm to rule them all. If there is ever a match, the signature will be fully evaluated, content will be checked (which starts with modification), and the regular expression *pcre:"/^Winhttp [0-9]+/[0-9]+/"* will be verified. So, if *Winhttp* is an efficient differentiator among the HTTP user agent's value, Suricata might have just one signature to fully evaluate instead of the original 60000.

This approach allows Suricata to analyze the full ruleset in a way that is not dependent on the number of signatures. This is dependent on whether or not the signature are correctly written. For example, we cannot have half of them using *Mozilla* as fast pattern buffer on the HTTP user agent because it will result in evaluating a huge number of signatures for each HTTP request since the ` Mozilla` string is present in the HTTP user agent of most common browsers.

6.2 Testing performance and correctness of written rules

Suricata provides a set of tools to help users write correct rules.

6.2.1 Engine analysis

Simply run the following command:

```
suricata -S mynew.rules -l /tmp/analysis --engine-analysis
```

If inputted correctly, you will receive information about the syntax of the rules

[14] https://en.wikipedia.org/wiki/Aho%E2%80%93Corasick_algorithm

```
ls -l /tmp/analysis/
total 16
-rw-r--r-- 1 eric eric    0 Feb 17 18:58 eve.json
-rw-r--r-- 1 eric eric    0 Feb 17 18:58 fast.log
-rw-r--r-- 1 eric eric  733 Feb 17 18:58 rules_analysis.txt
-rw-r--r-- 1 eric eric  643 Feb 17 18:58 rules_fast_pattern.txt
-rw-r--r-- 1 eric eric  665 Feb 17 18:58 rules.json
-rw-r--r-- 1 eric eric    0 Feb 17 18:58 stats.log
-rw-r--r-- 1 eric eric 2314 Feb 17 18:58 suricata.log
```

Information is provided in the files `rules_analysis.txt` and `rules_fast_pattern.txt`. In the first one, we can see a previous signature and a variant:

```
-------------------------------------------------------------------
Date: 17/2/2021 -- 19:30:28
-------------------------------------------------------------------
== Sid: 1 ==
alert http any any -> any any (msg:"Bad Agent"; http.user_agent; content: "Winhttp";␣
↪fast_pattern; startswith; pcre:"/^Winhttp [0-9]+\/[0-9]+/"; sid:1;)
    Rule matches on http user agent buffer.
    App layer protocol is http.
    Rule contains 0 content options, 1 http content options, 0 pcre options, and 1 pcre␣
↪options with http modifiers.
    Fast Pattern "Winhttp" on "http user agent (http_user_agent)" buffer.
    Warning: TCP rule without a flow or flags option.
             -Consider adding flow or flags to improve performance of this rule.

== Sid: 2 ==
alert http any any -> any any (msg:"Bad Agent, bad perf"; http.user_agent; pcre:"/^
↪Winhttp [0-9]+\/[0-9]+/"; sid:2;)
    Rule matches on http user agent buffer.
    App layer protocol is http.
    Rule contains 0 content options, 0 http content options, 0 pcre options, and 1 pcre␣
↪options with http modifiers.
    Warning: TCP rule without a flow or flags option.
             -Consider adding flow or flags to improve performance of this rule.
```

What we see here is that the first signature has a fast pattern and missed some options on TCP flow. For the second signature, where there is just a regular expression, we can see that there is no fast pattern and that the TCP flow options are also missing.

For the fast pattern analysis there is

```
-------------------------------------------------------------------
Date: 17/2/2021 -- 19:30:28
-------------------------------------------------------------------
== Sid: 1 ==
alert http any any -> any any (msg:"Bad Agent"; http.user_agent; content: "Winhttp";␣
↪fast_pattern; startswith; pcre:"/^Winhttp [0-9]+\/[0-9]+/"; sid:1;)
    Fast Pattern analysis:
        Fast pattern matcher: http user agent (http_user_agent)
        Flags: Depth
        Fast pattern set: yes
        Fast pattern only set: no
```

(continues on next page)

(continued from previous page)

```
        Fast pattern chop set: no
        Original content: Winhttp
        Final content: Winhttp

== Sid: 2 ==
alert http any any -> any any (msg:"Bad Agent, bad perf"; http.user_agent; pcre:"/^
↪Winhttp [0-9]+\/[0-9]+/"; sid:2;)
    Fast Pattern analysis:
        No content present
```

This confirms the fact that the second rule will trigger an evaluation of the regular expression for all the http requests (where there is an http user agent).

Information about the structure of the signature is also available in `rules.json`. It is less human friendly, but follows the evolution of Suricata's detection engine more closely. For example, this output is used by the *Suricata Language Server* to build advanced analysis of the signatures file.

6.2.2 Rules profiling

The information provided by Suricata in the engine analysis is really valuable, but it is often better to see the impact on a real run. To do so, there is a profiling system inside Suricata that needs to be activated during the build and can be setup in the configuration.

To build it you need to add `--enable-profiling` to the `./configure` command line. Suricata performance will be impacted and this should not be used in production, but you will have a `rule_perf.log` file in your log directory with performance information.

```
{
  "timestamp": "2021-02-17T19:41:56.012543+0100",
  "sort": "max ticks",
  "rules": [
    {
      "signature_id": 2,
      "gid": 1,
      "rev": 0,
      "checks": 1628,
      "matches": 4,
      "ticks_total": 2173774,
      "ticks_max": 49498,
      "ticks_avg": 1335,
      "ticks_avg_match": 23204,
      "ticks_avg_nomatch": 1281,
      "percent": 93
    },
    {
      "signature_id": 1,
      "gid": 1,
      "rev": 0,
      "checks": 4,
      "matches": 4,
      "ticks_total": 149520,
      "ticks_max": 41118,
```

(continues on next page)

(continued from previous page)

```
      "ticks_avg": 37380,
      "ticks_avg_match": 37380,
      "ticks_avg_nomatch": 0,
      "percent": 6
    }
  ]
}
```

Here, we see that signature 2 did take 93% of CPU cycles compared to the second one at 6%. This was expected as we evaluated the regular expression for all HTTP requests. An interesting observation is that `ticks_avg_nomatch` is 0 for the signature with fast pattern. The reason is that when there is no `Winhttp` string in the HTTP user agent the MPM algorithm simply skips the evaluation of the rules and hence its cost is null. With the incorrect signature we can see that the cost is 1281 ticks for every match attempt, and we have 4 `checks` for signature 1 and 1628 for signature 2. Hence, the performance ratio is calculated.

A perfect signature should have zero in `ticks_avg_nomatch` and should have a really low `ticks_avg_match`. The first point being the most important as it means the multi pattern matching on the signature is not triggering when the signature is not matching. This will be the case when the pattern used in MPM is discriminative enough that no other signatures are using it.

6.3 Guideline for performant rules

6.3.1 Trigger multi pattern matching

This is the main recommendation:

When writing a rule you need to find a way to trigger MPM in an efficient way. This means the signature must have a content match on a pattern that is on a differentiator. It should be almost unique in the ruleset so it reduces the signature group to the lowest number possible.

In our previous example, we used `http.user_agent; content: "Winhttp";` because the string `Winhttp` is not common among HTTP user agents. This guaranteed us an efficient prefiltering by the MPM engine. As we have seen previously in the profiling output, all the checks done on the signature have been successful. The rest of the filters were just confirmation filters to avoid potential false positives.

6.3.2 Prefilter everything

This is just a reformulation of the previous exigency. Even if the real match is a nasty regular expression, you still need to find the longest string possible with an efficient differentiator capability.

6.3.3 Matching on IOCs

In a lot of cases, indicators of compromises comes as a list of domains, IPs, and user agents to match against the produce data. An already seen approach consists of generating a rule for each indicator of compromise (IOC).

This will match, but the performance impact will be huge.

If you have to match on an IP list, it is better to use the IP reputation system via the iprep[15] keyword that allows a fast match and one single rule for any number of IP addresses.

[15] https://suricata.readthedocs.io/en/latest/rules/ip-reputation-rules.html

The same can be done for file hash via the keywords filemd5[16], filesha1[17], and filesha256[18] that match on the list of file hashes.

For example, with a list of sha256 file hashes named `known-bad-sha256.lst`, one can use the following signatures:

```
alert smb any any -> any any (msg:"known bad file on SMB"; filesha256:"known-bad-sha256.
↪lst"; sid:1; rev:1;)
alert nfs any any -> any any (msg:"known bad file on NFS"; filesha256:"known-bad-sha256.
↪lst"; sid:2; rev:1;)
alert http any any -> any any (msg:"known bad file on HTTP"; filesha256:"known-bad-
↪sha256.lst"; sid:3; rev:1;)
alert ftp-data any any -> any any (msg:"known bad file on FTP"; filesha256:"known-bad-
↪sha256.lst"; sid:4; rev:1;)
alert smtp any any -> any any (msg:"known bad file on SMTP"; filesha256:"known-bad-
↪sha256.lst"; sid:5; rev:1;)
```

Introduced in Suricata 5.0, dataset[19] is filling the gap for over existing IOCs. It can be used with any sticky buffers. For example, if you have a list of HTTP user agents in `bad-http-agent.lst`, you can use a signature similar to the following

```
alert http any any -> any any (msg:"bad user agent"; \
    http.user_agent; dataset:isset,bad-http-agent,type string,load:http-user-agent.lst,
↪memcap:1G,hashsize:1000000; \
    sid 6; rev:1;)
```

Please note: in the case of a dataset with string type, the set needs to first be encoded to base64 (without the trailing character).

6.4 Real life example

When Sunburst[20] was made public, a set of signatures was soon created to detect some of the offensive tools used by Fireeye. Among them we had this Snort-like signature:

```
alert tcp any $HTTP_PORTS -> any any (msg:"Backdoor.HTTP.BEACON.[CSBundle MSOffice␣
↪Server]"; content:"HTTP/1."; depth:7; \
      content:"{\"meta\":{},\"status\":\"OK\",\"saved\":\"1\",\"starttime\":17656184060,\
↪"id\":\"\",\"vims\":{\"dtc\":\""; \
      sid:25887; rev:1;)
```

This signature has some serious problems when run inside Suricata. The engine analysis gives the following result:

```
Rule matches on packets.
Rule matches on reassembled stream.
Rule contains 2 content options, 0 http content options, 0 pcre options, and 0 pcre␣
↪options with http modifiers.
Fast Pattern "{\x22meta\x22:{},\x22status\x22:\x22OK\x22,\x22saved\x22:\x221\x22,\
↪x22starttime\x22:17656184060,\x22id\x22:\x22\x22,\x22vims\x22:{\x22dtc\x22:\x22" on
```

(continues on next page)

[16] https://suricata.readthedocs.io/en/latest/rules/file-keywords.html?highlight=filemd5#filemd5
[17] https://suricata.readthedocs.io/en/latest/rules/file-keywords.html?highlight=filemd5#filesha1
[18] https://suricata.readthedocs.io/en/latest/rules/file-keywords.html?highlight=filemd5#filesha256
[19] https://suricata.readthedocs.io/en/latest/rules/datasets.html
[20] https://www.fireeye.com/blog/threat-research/2020/12/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor.html

(continued from previous page)

```
↪"payload and reassembled stream" buffer.
Warning: TCP rule without a flow or flags option.
         -Consider adding flow or flags to improve performance of this rule.
Warning: Rule has depth/offset with raw content keywords.  Please note the offset/depth␣
↪will be checked against both packet payloads and stream.  If you meant to have the␣
↪offset/depth checked against just the payload, you can update the signature as "alert␣
↪tcp-pkt..."
Warning: Rule is inspecting both the request and the response.
```

The first warning is about the lack of options because the signature is not checking the direction (to the client in our case) or ensuring that the flow is established. The second warning is more interesting because it warns us that Suricata will inspect the content twice: one time for every TCP packet and one time for each TCP stream. And finally, the third warning mentions that the signature could inspect request and response (in the event that the HTTP_PORTS variable is broad).

But the presence itself of HTTP_PORTS is a problem. If the attacker ever changes the port of the web server to something not covered by the variable we will miss the detection. A typical Suricata signature will fix that by making use of the port independent protocol detection.

This can simply be done by doing:

```
alert http any any -> any any
```

As we are looking at the stream to the client, we can add *flow:established,to_client* to the rule

If we run the modified rules through the detection engine, we see:

```
Warning: Rule app layer protocol is http, but content options do not have http_*␣
↪modifiers.
         -Consider adding http content modifiers.
```

Yes, we are still doing TCP stream matching on a signature on the HTTP protocols instead of matching inside the fields of the HTTP protocol.

Let's look at the first content match:

```
content:"HTTP/1."; depth:7;
```

We are matching on the beginning of the server answer because HTTP_PORTS was on the left in the initial signature. So what we have now is a confirmation that the answer starts by the *HTTP/1.* string. A potential solution is to use the keyword *http.response_line*:

```
http.response_line; content:"HTTP/1."; depth:7;
```

The second match is the following:

```
content:"{\"meta\":{},\"status\":\"OK\",\"saved\":\"1\",\"starttime\":17656184060,\"id\
↪":\"\",\"vims\":{\"dtc\":\"";
```

We don't have access to the packet, but it looks like a good guess to assume that the data was in the response body from the server.

So now we can do:

```
http.response_body; content:"{\"meta\":{},\"status\":\"OK\",\"saved\":\"1\",\"starttime\
↪":17656184060,\"id\":\"\",\"vims\":{\"dtc\":\"";
```

We end up with the following rules that have no warning:

```
alert http any any -> any any (msg:"Backdoor.HTTP.BEACON.[CSBundle MSOffice Server]"; \
      http.response_line; content:"HTTP/1."; depth:7; \
      http.response_body; content:"{\"meta\":{},\"status\":\"OK\",\"saved\":\"1\",\
↪"starttime\":17656184060,\"id\":\"\",\"vims\":{\"dtc\":\""; \
      flow:established,to_client; sid:25887; rev:1; )
```

The initial signature was published by Proofpoint in the emerging threats ruleset, but it was fully rewritten the next day by the Proofpoint team to instead read:

```
alert http $EXTERNAL_NET any -> $HOME_NET any (msg:"ET CURRENT_EVENTS [Fireeye] Backdoor.
↪HTTP.BEACON.[CSBundle MSOffice Server]"; \
      flow:from_server,established; \
      http.response_line; content:"HTTP/1."; depth:7; \
      file.data; content:"|7b 22|meta|22 3a 7b 7d 2c 22|status|22 3a 22|OK|22 2c␣
↪22|saved|22 3a 22|1|22 2c 22|starttime|22 3a|17656184060|2c 22|id|22 3a 22 22 2c␣
↪22|vims|22 3a 7b 22|dtc|22 3a 22|"; fast_pattern; \
      reference:url,github.com/fireeye/red_team_tool_countermeasures; \
      classtype:trojan-activity; sid:2031279; rev:3; \
      metadata:affected_product Windows_XP_Vista_7_8_10_Server_32_64_Bit, attack_target␣
↪Client_Endpoint, created_at 2020_12_08, deployment Perimeter, signature_severity Major,
↪ updated_at 2020_12_12;)
```

As expected, we have no warnings when doing the engine analysis:

```
Rule matches on http server body buffer.
Rule matches on http response line buffer.
App layer protocol is http.
Rule contains 0 content options, 2 http content options, 0 pcre options, and 0 pcre␣
↪options with http modifiers.
Fast Pattern "{\x22meta\x22:{},\x22status\x22:\x22OK\x22,\x22saved\x22:\x221\x22,\
↪x22starttime\x22:17656184060,\x22id\x22:\x22\x22,\x22vims\x22:{\x22dtc\x22:\x22" on
↪"http response body, smb files or smtp attachments data (file_data)" buffer.
No warnings for this rule.
```

This signature has some differences to our attempt. It uses *file.data* to match in the *http.response_body* but it is quite the same thing. It also forces the *fast_pattern* on this part of the content which should not be necessary but is always safe to do.

The rest is metadata and information. We first have the reference:

```
reference:url,github.com/fireeye/red_team_tool_countermeasures;
```

Then the classification:

```
classtype:trojan-activity;
```

And finally the metadata:

```
metadata:affected_product Windows_XP_Vista_7_8_10_Server_32_64_Bit, attack_target Client_
↪Endpoint,\
      created_at 2020_12_08, deployment Perimeter, signature_severity Major, updated_at␣
↪2020_12_12;
```

These pieces of metadata are important because we will find them in the alert event as shown on Fig. 6.1

Fig. 6.1: Metadata in the alert event

This allows efficient and flexible classifications of the alert events that can be used in queries and the interface. For example, it can be used to present the variety of alerts seen in a system like the one shown on Fig. 6.2

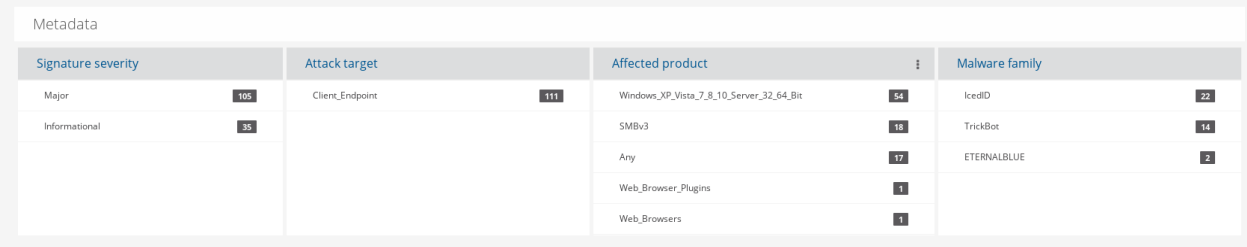


Fig. 6.2: Panels using signature metadata in Scirius

The result is shown in the Scirius[21] interface but any data lake that understands JSON will be able to build the same type of visualization.

Or for the created and updated date, a nice way to see which recent signatures did fire on the probes like shown on Fig. 6.3

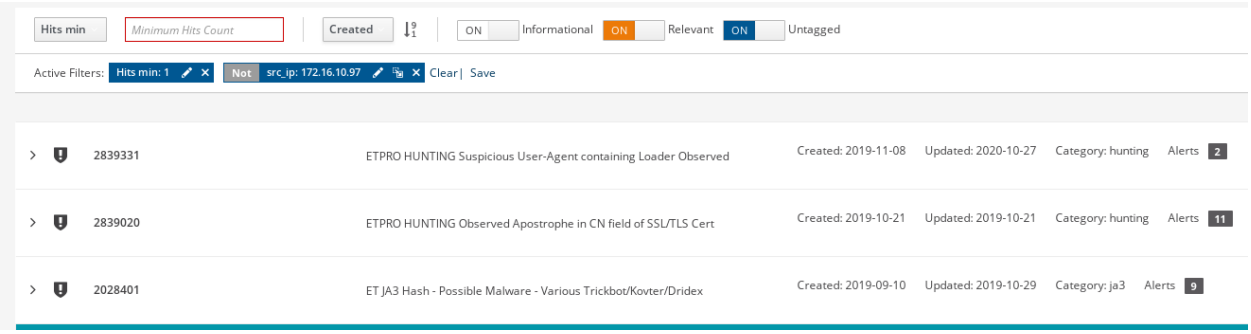


Fig. 6.3: Signatures ordered by creation date in Scirius

6.5 Fixing warnings from Suricata Language Server

The *Suricata Language Server* uses Suricata features to display warning and hints in IDE and text editors that support LSP. Some of the warnings may appear confusing at first, so let's take a tour to understand them and discover how to fix them.

6.5.1 Directionality warning

The signature

```
alert tcp any any -> any any (msg:"toto out"; content:"toto"; sid:1; rev:1;)
```

triggers the following warning: 'Rule inspect server and client side, consider adding a flow keyword`

In this signature, the *content* match has no sticky buffer or content modifier attached. As a result, the match is done on the TCP stream data. TCP stream goes two ways, so the inspection will be done for all data going to the server and all data going to the client. In most cases, this is not what we want to match as we usually know that the pattern should be in a client or server message.

[21] https://github.com/StamusNetworks/scirius

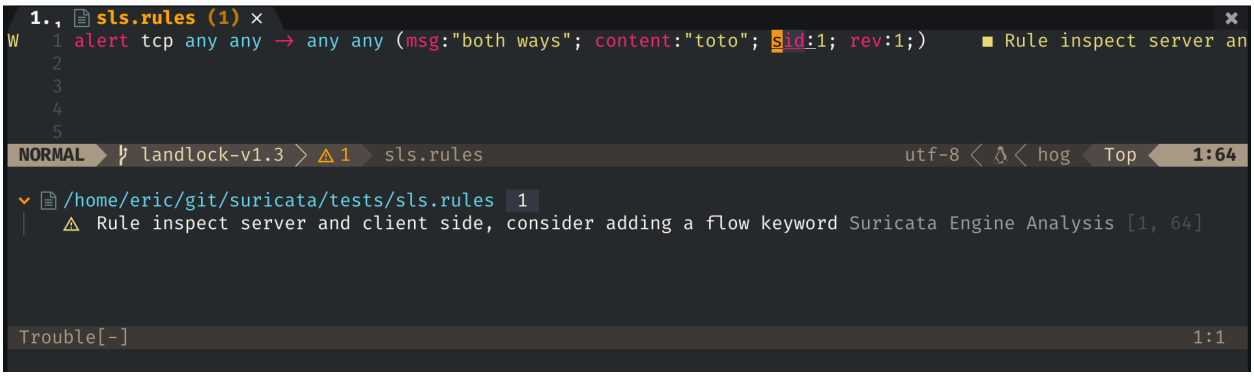


Fig. 6.4: Directionality warning seen in Neovim

So the correct signature would look something like this:

```
alert tcp any any -> any any (msg:"toto out"; content:"toto"; \\
          flow:established,to_server; \\
          sid:1; rev:1;)
```

By doing this, the inspection will only be done on the packet going to the server. As a result, the inspection work is cut in half as we are just inspecting one way.

6.5.2 Mixed content

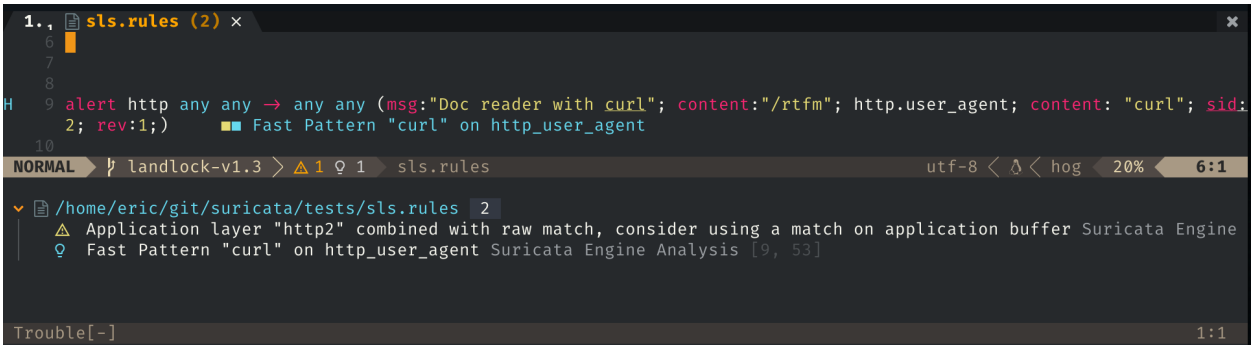


Fig. 6.5: Mixed content warning seen in Neovim

The signature

```
alert http any any -> any any (msg:"Doc reader with curl"; \\
          content:"/rtfm"; \\
          http.user_agent; content:"curl"; \\
          sid:2; rev:1;)
```

triggers the following warning: 'Application layer "http2" combined with raw match, consider using a match on application buffer'

In the signature the first match *content:"/rtfm"* is done on TCP stream data as there is no sticky buffer or content modifier associated with it. But the second match, *http.user_agent; content:"curl";*, is done on the HTTP user agent buffer. This setup is not natural as it is better to work on one of the HTTP fields for all the matches. If we look at the first match, it looks like an URL.

So the correct signature would look something like

```
alert http any any -> any any (msg:"Doc reader with curl"; \\
           http.uri; content:"/rtfm"; \\
           http.user_agent; content:"curl"; \\
           sid:2; rev:1;)
```

6.5.3 Missing HTTP keywords

Fig. 6.6: Missing HTTP keywords warning seen in Neovim

The signature

```
alert http any any -> any any (msg:"Doc reader"; content:"GET /rtfm"; sid:3; rev:1;)
```

triggers the following warning: 'pattern looks like it inspects HTTP, use http.request_line or http.method and http.uri instead for improved performance'

In this signature, we have a single content match that searched for 2 words and looks like a part of an HTTP request. Suricata did detect that and is warning that it would be better to use proper HTTP keywords. This will be better for multiple reasons. First, the HTTP keywords match on normalized strings and it will improve the resilience of the signature to evasion compared to a simple content match. Second, it is far more accurate to use matches on HTTP fields. In this particular case, the signature will alert on any HTTP stream that contains *GET /rtfm*. As a consequence, it will, for example, alert if the signature file is downloaded over HTTP.

So the correct signature would look more like this:

```
alert http any any -> any any (msg:"Doc reader with curl"; \\
           http.method; content: "GET"; \\
           http.uri; content:"/rtfm"; \\
           sid:2; rev:1;)
```

We have a match on the HTTP method followed by a match on the URI.

6.6 Performance Improvement process

There are always multiple ways to write a rule. The variants depend on what you are going to match on and what methods are being used for that match. For example, the two following rules may match the same way on a sample, but could have varying levels of performance:

```
alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"Test - Rule variant - 1"; \\
           flow:established,to_server; \\
           http.method; content:"GET"; http.uri; \\
           content:"lookforthis"; \\
           classtype:command-and-control; sid:1000002; rev:1; \\
           metadata:created_at 2022_08_10, updated_at 2022_08_10;)

alert http $HOME_NET any -> $EXTERNAL_NET any (msg:"Test - Rule variant - 2"; \\
           flow:established,to_server; urilen:25; \\
           http.method; content:"GET"; http.uri; \\
           content:"lookforthis"; http.cookie; content:"lookforthat"; \\
           classtype:command-and-control; sid:1000003; rev:1; \\
           metadata:created_at 2022_08_10, updated_at 2022_08_10;)
```

To validate the performance of a rule and select the best one, it must be be ran and evaluated over both relevant and non relevant pcaps so the impact of the rule can be seen on all types of traffic. To do so, you must run the rule through both types of pcaps while having *rule-profiling* enabled.

The signature needs to be complete (See steps in *Signature writing process*) before you can test its performance.

1. Verify the rule syntax with Suricata Language Server or run Suricata with *–engine-analysis*
2. Use a pcap with relevant traffic
    - Run the pcap and the rules with suricata that has rules profiling enabled. A relevant section in the suricata *suricata.yaml* config can be used to adjust sorting or to enable text and JSON outputs
    - Review the results in *rule_perf.log* and make further adjustments as needed. See *Profile information* for details
3. Use a pcap with non relevant traffic.
    - Run with rules profiling
    - Review the results

The winning rule is the one with the lowest impact to performance on the relevant traffic and ideally done not appear (aka is not being evaluated at all) in the non-relevant traffic pcap run.

CHAPTER

SEVEN

FILE ANALYSIS

7.1 Introduction

Because Suricata understands most major application layers, it is able to track the file transfered over the wire. The list of application layers supporting file extraction includes:

- HTTP
- FTP
- SMB
- NFS
- SMTP
- HTTP2

Interesting features result from this. First, it allows Suricata to generate events containing information about the files. The *fileinfo events* are generated once any tracked file transfer is over (independantly of any detection). These events contain details about the file such as its name, various hashes of its content (sha1, sha256, ...), and identification of the file type based on its content.

The second interesting feature is the extraction of the file which is triggered by the filestore[22] keyword in signature. Extraction can also be switched on globally, but it is really intensive in term of performance. One thing to mention about extraction is that it is deduplicated as the storage of the file on the disk is done once per sha256.

The third feature associated with the file is the analysis of file content that can be done via the *file_data* keyword. Signatures can be written to match on the content of a file which, for example, can be compressed in the case of HTTP or under a base64 encoded form in the case of SMTP.

Please see the Suricata manual for how to set up file extraction[23].

7.2 Fileinfo events

The structure of a *fileinfo* event is as follows:

```
{
  "timestamp": "2019-07-05T22:01:04.745891+0200",
  "flow_id": 2209746386047329,
  "pcap_cnt": 33861,
```

(continues on next page)

[22] https://suricata.readthedocs.io/en/latest/rules/file-keywords.html?#filestore
[23] https://suricata.readthedocs.io/en/latest/file-extraction/file-extraction.html

(continued from previous page)


```
  "event_type": "fileinfo",
  "src_ip": "5.188.168.49",
  "src_port": 80,
  "dest_ip": "10.7.5.101",
  "dest_port": 49686,
  "proto": "TCP",
  "community_id": "1:shQmhcocLIrJ1WtOAbgShXgB5FY=",
  "http": {
    "hostname": "5.188.168.49",
    "url": "/sin.png",
    "http_user_agent": "WinHTTP loader/1.0",
    "http_content_type": "image/png",
    "http_method": "GET",
    "protocol": "HTTP/1.1",
    "status": 200,
    "length": 110718
  },
  "app_proto": "http",
  "fileinfo": {
    "filename": "/sin.png",
    "magic": "PE32 executable (GUI) Intel 80386, for MS Windows",
    "gaps": false,
    "state": "CLOSED",
    "sha1": "2408c5380ddca2bbd53b87c27132b72f0927c70f",
    "sha256": "110743634989ed7a3293b2e39ad85c255fc131c752e029f78d37d4fb8c1dc7f6",
    "stored": false,
    "size": 369664,
    "tx_id": 1
  }
}
```


The event contains a *fileinfo* object that contains the following fields:

- *filename* announced by the servers
- *magic* computed by analyzing the beginning of the file
- *size* to receive the file size

It also contains a regular *http* subobject as this file was captured on an HTTP flow. On a different application's layers, a different subobject would have been present. The field *app_proto* is a good way to know which suboject will be present.

This event is a good example of the value of the various mechanisms in place in Suricata. The HTTP parser told us that the file content type (*http.http_content_type*) announced by the server is an 'imagepng'. This would be fine if the analysis of content of the file did not find out (in the key *fileinfo.magic*) that the file is, in reality, an executable. For reference, this file was used in an infection by the Trickbot malware.

This can be confirmed by checking the sha1 or sha256 hash of the file in Virustotal[24]. This file is flagged as malicous by more than 50 security vendors and associated to Trickbot by some of them as well.

[24] https://www.virustotal.com/gui/file/110743634989ed7a3293b2e39ad85c255fc131c752e029f78d37d4fb8c1dc7f6

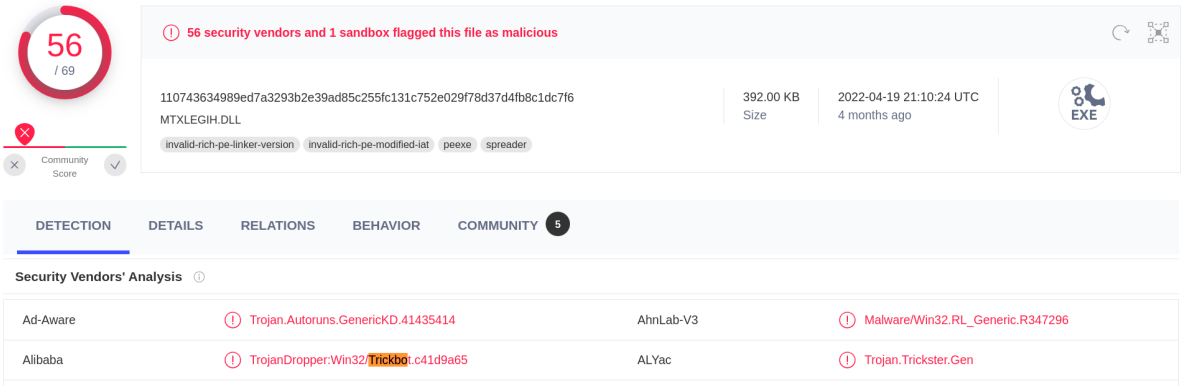


Fig. 7.1: Information from Virustotal on the file.

7.3 Detection on tracked files

7.3.1 file.data keywords

The *file.data* keyword matches on the content of the file, so it can be used to do an analysis of the content of the transferred file with the inspection capability of Suricata. This keyword is aliased to *file_data* (which is used in a lot of available signatures as it is the original name). The keyword alias *file.data* is a sticky buffer, so it will trigger matching on the file content for all subsequent match keywords.

Let's take an example with the following signature from the Emerging Threats ruleset:

```
alert http $EXTERNAL_NET any -> $HOME_NET any ( \\
  msg:"ET SCADA PcVue Activex Control Insecure method (AddPage)"; \\
  flow:to_client,established; \\
  file.data; content:"<OBJECT "; nocase; content:"classid"; nocase; distance:0; \\
     content:"CLSID"; nocase; distance:0; \\
     content:"083B40D3-CCBA-11D2-AFE0-00C04F7993D6"; nocase; distance:0; \\
     content:".AddPage"; nocase; \\
     content:"<OBJECT"; nocase; \\
     pcre:"/^[^>]*?classid\s*=\s*[\x22\x27]?\s*clsid\s*\x3a\s*\x7B?\s*?083B40D3-CCBA-
↪11D2-AFE0-00C04F7993D6/Rsi"; \\
     reference:url,exploit-db.com/exploits/17896; classtype:attempted-user; \\
     sid:2013730; rev:4; \\
  )
```

This is triggering on https://www.exploit-db.com/exploits/17896 that is a DOS on Activex. This signature is over the HTTP protocol and it is using the *file.data* keyword. This happens because the HTTP protocol is usually compressing the data sent from the server to lower the bandwith. As a result, a simple match on the content would have failed. By using a content match on *file.data*, we ensure a correct match on the content that is seen by the browser even if there is server-side compression as Suricata will uncompress the content to pass the clear text content to the *file.data* keyword.

The matching done in the signature is an interesting use of sticky buffer. It first does multiple content matches to check that all fixed string parts of the attack are there. This lowers the risk of evaluating the costly regular expression that is used as a final check for the presence of the attack in the server message.

7.3.2 Magic analysis

Among the keywords dealing with the file, we find *file.magic*. This is a sticky buffer matching on the result of Magic inspection. This can, for example, be used to detect the executables masqueraded as an image seen in the previous section:

```
alert http any any -> any any (msg:"masquerade file"; \\
      http.content_type; content:"image"; \\
      file.magic; content:"executable";)
```

Another simple possibility offered by *file.magic* is file extraction selection. For example, to extract all PDF to disk, one can use:

```
alert tcp any any -> any any (msg:"PDF extraction"; \\
      file.magic; content:"pdf"; nocase; \\
      filestore;)
```

7.3.3 Known bad and known good list

If checksum of file is really interesting information found in the *fileinfo* events, they can also be matched on via the filemd5[25], filesha1[26], and filesha256[27] keywords. All of these work the same way: they are given a file as an argument that has to contain one checksum per line and they will match if the checksum of the file is on the list (or not if the match is negated).

For example, to alert on all excutables that are not on the list of known good executables (built from another tool), one can use:

```
alert smb any any -> any any (msg:"Unknown executable file on SMB"; \\
      filesha256:!sha256-goodexe; \\
      file.name; content:".exe"; endswith; \\
      sid:1; rev:1;)
```

7.4 Threat hunting with file

7.4.1 Masqueraded files

The masqueraded files described in *Fileinfo events* can be detected by looking at the *fileinfo* events.

In Elasticsearch, you can simply detect executable masqueraded as PDF with the following request:

```
fileinfo.filename.keywords:*.pdf AND fileinfo.magic:"executable"
```

You can also be more generic with querying all executables that do not end up with a regular extension:

```
fileinfo.magic:"executable" -fileinfo.filename.keyword:*.exe -fileinfo.filename.
↪keyword:*.dll -fileinfo.filename.keyword:*.com
```

And if you want to zoom on internal protocol, you can do:

[25] https://suricata.readthedocs.io/en/latest/rules/file-keywords.html#filemd5
[26] https://suricata.readthedocs.io/en/latest/rules/file-keywords.html#filesha1
[27] https://suricata.readthedocs.io/en/latest/rules/file-keywords.html#filesha256

```
(app_proto:"smb" OR app_proto:"nfs") AND  \\
   (fileinfo.magic:"executable" -fileinfo.filename.keyword:*.exe -fileinfo.filename.
↪keyword:*.dll -fileinfo.filename.keyword:*.com)
```

Splunk users can write this last one with:

```
app_proto IN ("smb", "nfs") |
  regex fileinfo.magic = "(?i)executable" |
  NOT (fileinfo.filename="*.exe" OR fileinfo.filename="*.dll" OR fileinfo.filename="*.com
↪")
```

7.4.2 Long file name

The file names are usually kept short when they are linked to legitimate behavior because nobody likes to type or read lengthy strings. Because of this, it is interesting to look at any executable file transfer where the filename is at least 15 characters long and does not finish on ".exe" (installers could have a longer name).

This can be done with:

```
fileinfo.type:"executable" AND fileinfo.filename.keyword:/.{15}.*/  \\
  -fileinfo.filename.keyword:*.exe
```

7.4.3 Entropy on SMB file transfer

Entropy[28] is the next logical step after looking into a long filename because it measures the randomness of the data. In a lot of cases, malware uses randomly generated file names to avoid collision with existing files.

Entropy can be computed in Splunk by using the URL Toolbox App[29]. For example, let's compute the entropy of the executable filename and get the list of filename sorted by entropy:

```
event_type=fileinfo app_proto=smb |
regex fileinfo.magic = "(?i)executable" |
`ut_shannon(fileinfo.filename)` |
eval entropy = round(ut_shannon, 2) |
stats min(timestamp), max(timestamp) by fileinfo.filename, entropy, fileinfo.sha256 |
sort -entropy
```

An entropy value of 4 is already high with regards to a filename, so filtering on value can allow you to focus on suspect elements.

[28] https://en.wikipedia.org/wiki/Entropy_(information_theory)
[29] https://splunkbase.splunk.com/app/2734/

CHAPTER

EIGHT

FLOW ANALYSIS

8.1 Introduction

Flow data, also known as Netflow[30] data, is a well known network analysis tool that has long been used for security purposes. The concept, which was introduced by Cisco in 1996, centered around doing accounting on sessions. Basic information found in flow data includes the number of bytes and packets, the start and end time of a flow, and the IP information needed to identify the flow.

The concept of a session is straight forward for TCP, but for sessionless protocols an approximation is used based on an internally defined timeout. For example, a flow on UDP will be opened when a client sends data to a server from a given port and to a given port. It will end when no information is sent one way or another for a predefined duration. The timeout really depends of the systems collecting the information, and in the case of Suricata it can be setup per protocol.

8.2 Flow events in Suricata

Suricata generates its own flow log independent of other events like alerts or protocol logs. It internally tracks UDP, TCP, SCTP, and ICMP for analysis purpose and uses the collected information to generate entries for every flow.

There are two types of flow events: - `flow`: one entry per flow so volumetry for client and server data is available in the same event - `netflow`: two entries per flow, client and server accounting are in 2 different events

The `flow_id` key is obviously present in flow and can be used to correlate flow data with other events. In the case of netflow, it is used to correlate both sides of the flow.

A typical `flow` entry looks like the following:


```
{
  "timestamp": "2021-11-17T23:43:24.129401+0100",
  "flow_id": 1115914617724757,
  "in_iface": "enp5s0",
  "event_type": "flow",
  "src_ip": "2a02:1511:5172:1d50:3615:b3a2:a98a:c71f",
  "src_port": 34096,
  "dest_ip": "2a00:87c0:2021:2021:0050:0000:4000:0539",
  "dest_port": 443,
  "proto": "TCP",
  "app_proto": "tls",
  "flow": {
```


(continues on next page)

[30] https://en.wikipedia.org/wiki/NetFlow

(continued from previous page)


```
    "pkts_toserver": 3300,
    "pkts_toclient": 6684,
    "bytes_toserver": 334979,
    "bytes_toclient": 9887597,
    "start": "2021-11-17T23:32:59.915179+0100",
    "end": "2021-11-17T23:38:59.483429+0100",
    "age": 360,
    "state": "closed",
    "reason": "shutdown",
    "alerted": false
  },
  "tcp": {
    "tcp_flags": "1f",
    "tcp_flags_ts": "1e",
    "tcp_flags_tc": "1b",
    "syn": true,
    "fin": true,
    "rst": true,
    "psh": true,
    "ack": true,
    "state": "closed"
  }
}
```


We find the traditional IP information (`src_ip`, `src_port`, `dest_ip`, `dest_port`) and some information for the application layer with the key `app_proto` that is here set to `tls`. As this is a TCP flow, we have a `tcp` subobject that contains a set of keys. If `state` is coding the state of the session in the TCP engine at the end of the flow, the rest of the keys are coding information about the flags seen on the TCP session. The boolean values are set to true if the corresponding flag has been seen. The three `tcp_flags*` key contain the hexadecimal value of the integer obtained by setting to 1 all flags seen during the life of the session. `tcp_flags` is global, and `tcp_flags_ts` stores the information for packets sent to the server while `tcp_flags_tc` stores the ones sent to the client.

Note: Check the eve Flow format[31] page in Suricata manual for more information on the flow events.

8.3 Flow Analysis

There is a great amount of techniques which use flow to find anomalies in the traffic by applying statistical analysis or machine learning to the events. We are not going to cover this in this document, but we will showcase some simple examples.

[31] https://suricata.readthedocs.io/en/latest/output/eve/eve-json-format.html?highlight=http#event-type-flow

8.3.1 ICMP flow with abnormal size

ICMP flows are usually short as they are mostly used to check connectivity from one point in the network to another. Being short, the amount of bytes exchanged are usually small. Some data exfiltration techniques abuse ICMP to send the data out without being noticed. But in a lot of cases, these techniques send their data on the same flow. So looking at flows that shows important data transfer (like 150kb for example) is interesting.

Using Lucene syntax, it can be written in Kibana as follows:

```
event_type:flow AND \\
  (proto:"ICMP" OR proto:"IPv6-ICMP") AND \\
  (flow.bytes_toclient:>150000 OR flow.bytes_toserver:>150000 ) AND \\
  flow.bytes_toclient:>0 AND flow.bytes_toserver:>0
```

In Splunk, one can use:

```
event_type=flow AND
 (proto="ICMP" OR proto="IPv6-ICMP") AND
 (flow.bytes_toclient>150000 OR flow.bytes_toserver>150000) AND
 flow.bytes_toclient>0 AND flow.bytes_toserver>0
```

8.3.2 High volume DNS flow

Similar to the previous example, DNS can also be used for data exfiltration and a potential consequence of the technique used is the existence of DNS flow where a big amount of data has been transfered.

Using Lucene syntax, it can be written in Kibana as follows:

```
event_type:flow AND app_proto:dns AND \\
  flow.bytes_toclient:>5000 OR flow.bytes_toserver:>5000
```

8.3.3 Potential ICMP evasion

In a standard environment, the reply to an ICMP query is an ICMP response that contains the same data. As a result the size of the data in the direction of the client and in the direction of the server are equal.

Using Splunk, matching events can be obtained by doing:

```
event_type=flow AND proto=icmp AND flow.bytes_toclient!=flow.bytes_toserver
```

Using Kibana, it is possible to do the same in 2 steps. First define a Query DSL as follows:

```
{
  "query": {
    "bool": {
      "filter": {
        "script": {
          "script": {
            "lang": "painless",
            "source": "doc['flow.bytes_toclient'].value!=doc['flow.bytes_toserver'].value
↪"
          }
        }
```

(continues on next page)

(continued from previous page)

```
        }
      }
    }
}
```

See Fig. 8.1 for help on adding this Query DSL filter in discover window:

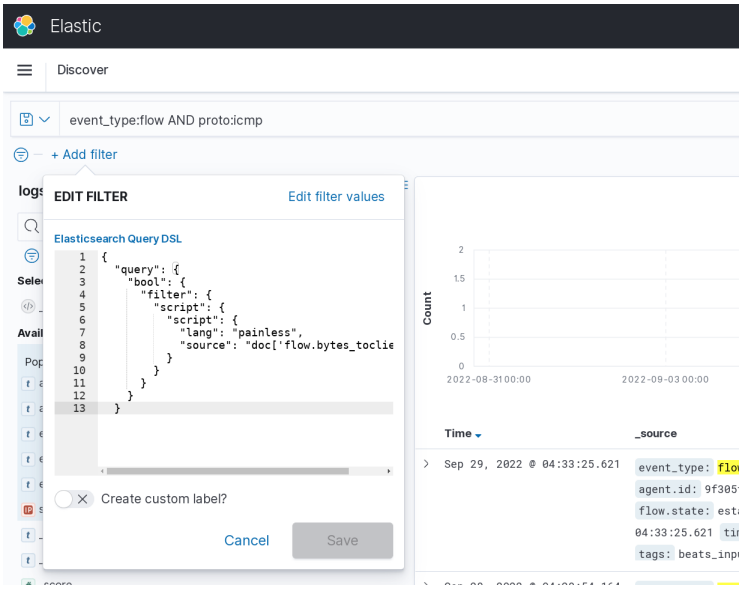


Fig. 8.1: Query DSL edition in Kibana.

Then, the following filter can be added to select the ICMP messages:

```
event_type:flow AND proto:icmp
```

CHAPTER

NINE

TLS DETECTION AND THREAT HUNTING

9.1 Introduction

The TLS protocol is everywhere. The Secure Socket Layer implementation, initially developed for the Mozilla browser, has evolved into one of the most prominent standards. It is widely used in HTTPS and other communications protocols to encrypt communication. Yes, encrypt, which for network security is equivalent of saying, "hide all the juicy details".

But there is still information that can be extracted or built from encrypted communication. This can be used for threat hunting as well as an IDS approach.

9.2 Protocol overview

In all versions of TLS, the client is opening a connection to the server and then sending an initial message. It contains the client capabilities in terms of encryption. Using that, the server then replies with potential agreement on encryption technique to use, as well as its certificates. The client analyzes this message and checks that the server certificate is valid. If everything is fine, the client sends its certificates and a seed that is needed to start the encrypted exchange. The server then initiates the encryption and the session switches to encryption.

Before TLS 1.3, the X509 certificate was in clear text. But since TLS 1.3, it is encrypted. As a result, visibility had been really limited with TLS 1.3.

In most implementations, there is a TCP connection and then a TLS handshake, but in some cases the server offers a clear text and an encrypted service on the same port. In this case, a mechanism is needed on the clear text protocol to trigger the switch. In most implementations this is the `STARTTLS` message. Most common protocols using this are SMTP, IMAP, and FTP.

9.3 TLS analysis in Suricata

9.3.1 TLS handshake analysis

Suricata does not decrypt the traffic but rather realizes an analysis of the TLS handshake. By doing this, it manages to extract information on the TLS characteristics as well as on the X509 certificates. This data is written in the `tls` event type and are also added to the `alert` when they are available.

Suricata can also extract the certificate chain sent by the server and store it inside the event or as a separate file.

9.3.2 Extracted fields

Suricata extracts information about the TLS handshake and outputs this information in `tls` events.

A typical event looks like the following:


```
{
  "timestamp": "2020-05-08T23:32:34.218590+0200",
  "flow_id": 1737090126716212,
  "pcap_cnt": 41441,
  "event_type": "tls",
  "src_ip": "10.0.0.128",
  "src_port": 52046,
  "dest_ip": "64.233.179.94",
  "dest_port": 443,
  "proto": "TCP",
  "tls": {
    "subject": "C=US, ST=California, L=Mountain View, O=Google LLC, CN=*.gstatic.com",
    "issuerdn": "C=US, O=Google Trust Services, CN=GTS CA 1O1",
    "serial": "74:E6:32:EA:F9:C6:35:C2:02:00:00:00:00:63:98:DD",
    "fingerprint": "f5:af:1c:45:74:1b:2e:f2:5a:85:d1:49:be:dc:97:0d:2e:0c:97:a2",
    "sni": "www.gstatic.com",
    "version": "TLS 1.2",
    "notbefore": "2020-04-15T20:24:10",
    "notafter": "2020-07-08T20:24:10"
  }
}
```


Among the interesting fields, we have the `tls.sni` which stands for TLS Server Name Indication and is in fact the host name requested by the client. This is sent by the client in the first message to allow the server to choose which certificate to send in his answer. This way the server can honor multiple services on the same port.

In this case, we have the `tls.subject` equals to `"C=US, ST=California, L=Mountain View, O=Google LLC, CN=*.gstatic.com"` which means because of the `CN` field that the certificate can serve any site that matches `*.gstatic.com`. So we have some supplementary information thanks to the TLS SNI.

Note: Check the eve TLS format[32] page in Suricata manual for more information on the TLS events.

9.3.3 TLS JA3

In a standard TLS handshake, little is known about the client side. This is because the client certificate is not usually familiar nor is it sent over the wire. If we compare this with HTTP, we don't have the user agent field that (even if it is a declarative field) is a valuable source of information, allowing us to identify and classify protocol clients.

JA3[33] was created by John B. Althouse, Jeff Atkinson, and Josh Atkins (hence the name of the method) to address this issue. It is based on the fact that similar implementations will send similar negotiation parameters in the initial message. By carefully selecting some of these parameters, we can build an identifier that discriminates the implementations with a fine granularity. As with most clever techniques, this looks very simple on the surface, but it has proven to be an incredibly efficient way to fingerprint a TLS client.

Identifying malware traffic with JA3 has proven to be successful even if there is a non-zero false positive.

[32] https://suricata.readthedocs.io/en/latest/output/eve/eve-json-format.html?highlight=http#event-type-tls
[33] https://github.com/salesforce/ja3

The following example is a Suricata TLS event with JA3 activated:


```
{
  "timestamp": "2020-05-08T23:35:24.922820+0200",
  "flow_id": 995065818031171,
  "pcap_cnt": 51204,
  "event_type": "tls",
  "src_ip": "10.0.0.128",
  "src_port": 52047,
  "dest_ip": "144.91.76.208",
  "dest_port": 443,
  "proto": "TCP",
  "tls": {
    "subject": "C=GB, ST=London, L=London, O=Global Security, OU=IT Department,␣
↪CN=example.com",
    "issuerdn": "C=GB, ST=London, L=London, O=Global Security, OU=IT Department,␣
↪CN=example.com",
    "serial": "00:9C:FC:DA:1D:A4:70:87:5D",
    "fingerprint": "b8:18:2d:cb:c9:f8:1a:66:75:13:18:31:24:e0:92:35:42:ab:96:89",
    "version": "TLSv1",
    "notbefore": "2020-05-03T11:07:28",
    "notafter": "2021-05-03T11:07:28",
    "ja3": {
      "hash": "6734f37431670b3ab4292b8f60f29984",
      "string": "769,47-53-5-10-49171-49172-49161-49162-50-56-19-4,65281-10-11,23-24,0"
    },
    "ja3s": {
      "hash": "623de93db17d313345d7ea481e7443cf",
      "string": "769,49172,65281-11"
    }
  }
}
```


The ja3 part is the following:


```
{
  "ja3" {
    "hash": "6734f37431670b3ab4292b8f60f29984",
    "string": "769,47-53-5-10-49171-49172-49161-49162-50-56-19-4,65281-10-11,23-24,0"
  }
}
```


It is composed of 2 fields: a string that is built by concatenating a predefined list of negotiation parameters and a hash value that is simply the md5 hash of the string.

This hash has been linked to Trickbot[34] by John B. Althouse. So just using this information is enough to identify a potential malware. Even if the server infrastructure is composed of multiple services and evolves, the JA3 of the client will stay the same as the data is based on the client's first message that can not be influenced by the server.

[34] https://twitter.com/4a4133/status/1043246635239854081?lang=en

9.3.4 TLS JA3s

JA3s is almost enough to define what JA3s is. It is a technique similar to JA3 that is used to fingerprint the TLS implementation of server. By analyzing the first message from the server, a predefined list of parameters is concatenated and a md5 hash is built. This leads to the following result in our previous entry:

```
{
  "ja3s": {
    "hash": "623de93db17d313345d7ea481e7443cf",
    "string": "769,49172,65281-11"
  }
}
```

But there is a big difference between JA3 and JA3s. Because the first message from the server is an answer to the client to continue the negotiation, the server message is dependant of the client. As a result, the JA3s is in fact an identifier of a client and server connection more than a server identification. To be fully explicit, two different clients connecting to a server will result in two different JA3s value.

9.4 TLS and Detection

9.4.1 TLS keywords

As usual, it is recommended to use all sticky buffers variants as they offer greater flexibility and better performance.

There are two classes of keywords: the one matching the TLS certificate information and the one matching on ja3 and ja3s data.

Name	Description
tls.sni	sticky buffer to match specifically and only on the TLS SNI
tls.certs	sticky buffer to match the TLS certificate
tls.cert_issuer	sticky buffer to match specifically and only on the TLS cert issuer
tls.cert_subject	sticky buffer to match specifically and only on the TLS cert subject
tls.cert_serial	sticky buffer to match the TLS cert serial
tls.cert_fingerprint	sticky buffer to match on the TLS cert fingerprint
tls.version	match on TLS/SSL version
tls_cert_notbefore	match TLS certificate notBefore field
tls_cert_notafter	match TLS certificate notAfter field
tls_cert_expired	match expired TLS certificates
tls_cert_valid	match not expired TLS certificates
ja3.hash	sticky buffer to match the JA3 hash
ja3.string	sticky buffer to match the JA3 string
ja3s.hash	sticky buffer to match the JA3S hash
ja3s.string	sticky buffer to match the JA3S string

Extensive documentation and syntax explanation is available in Suricata documentation in the TLS keywords page[35].

[35] https://suricata.readthedocs.io/en/latest/rules/tls-keywords.html

9.4.2 Cookbook

Detecting expired certificates

Let's get an alert when one of the servers we monitor has an expired certificate:

```
alert tls $SERVERS any -> any any (msg:"Expired certs on server"; \\
     tls_cert_expired; \\
     sid:1; rev:1;)
```

Here, we simply use the *tls_cert_expired* keyword and the *$SERVERS* variable that needs to be placed on the left because the certificate data we want to check is coming from the servers.

Checking that internal PKI is used

The company we work for is running an expensive Public Key Infrastructure (PKI) and we want to be sure it is used for all the services running on our servers. If the TLS issuer of our PKI is *C=US, O=My Company*, we can simply use the following signature that leverages the *tls.cert_issuer* sticky buffer keyword.

```
alert tls $SERVERS any -> any any (msg:"Non Company PKI on server"; \\
     tls.cert_issuer; content:!"C=US, O=My Company"; \\
     sid:2; rev:1;)
```

We use an *!* on the content keyword to negate the match.

If we need to deal with historal data, we can just do a trigger alert for certificates where the beginning of validity is after the date when the PKI is supposed to be implemented everywhere:

```
alert tls $SERVERS any -> any any (msg:"Non Company PKI on server"; \\
     tls.cert_issuer; content:!"C=US, O=My Company"; \\
     tls_cert_notbefore:>2021-04-01; \\
     sid:2; rev:1;)
```

Checking Tactiques, Techniques and Procedure on certificate building

Correctly creating TLS certificates is not necessarily a trivial task for either a threat hunter or attacker. For example, some Ursnif campaigns have been using certificates where the subject DN was of the form *C=XX, ST=1, L=1, O=1, OU=1, CN=**. This *XX* and *1* are not something expected in regular certificates and it is a mark of the Tactics, Techniques, and Procedures (TTP) of the attacker.

This is something we can detect with a signature:

```
alert tls $EXTERNAL_NET any -> $HOME_NET any (msg:"Ursnif like certificate"; \\
     tls.cert_subject; content:"C=XX"; content:"=1,"; \\
     sid:3; rev:1;)
```

Here, we alert when a certificate on an external server is using a certificate that follows the pattern we have found in the Ursnif campaign.

Verifying a list of known bad JA3

```
alert tls $HOME_NET any -> any any (msg:"New internal certificate authority"; \\
      tls.ja3; dataset:set,bad-ja3, type string, load bad-ja3.lst; \\
      sid:4; rev:1;)
```

Here, we alert as soon as a TLS JA3 from the set of known bad JA3 is seen.

Build the list of internally used certificate authorities

In a production environment it is useful to know what TLS certificates authorities are using internally. This can be done with Suricata by using the dataset keyword:

```
alert tls $HOME_NET any -> any any (msg:"New internal certificate authority"; \\
      tls.issuerdn; dataset:set,internal-issuers, type string, state internal-issuers.
↪lst, memcap 10Mb, hashsize 100; \\
      sid:5; rev:1;)
```

Here we alert as soon as a TLS issuer is seen coming from the internal network that has never been seen before.

9.5 Hunting on TLS events

9.5.1 Self signed certificates

Self signed certificates can be detected via signatures. See this blog post[36] by Stamus Networks explaining the process using a lua based signature.

This can also be done using the TLS events. If *tls.issuerdn* is equal to *tls.subject*, then we have a self signed certificate.

If you have only the EVE JSON file and access to the command line, you can use *jq* to find them:

```
cat eve.json | jq 'select(.event_type=="tls" and .tls.issuerdn==.tls.subject)'
```

In Splunk, one can simply do the following:

```
event_type="tls" tls.subjectdn=tls.issuerdn
```

If your data is in Elasticsearch you can do a search in Kibana with DSL filter:

```
{
  "query": {
    "bool": {
      "must": {
        "script": {
          "script": {
            "inline": "if (doc.containsKey('tls.subject.keyword') && (!doc['tls.subject.
↪keyword'].empty)) { return (doc['tls.subject.keyword'] == doc['tls.issuerdn.keyword'])␣
↪} else { return false }"
          }
        }
```

(continues on next page)

[36] https://www.stamus-networks.com/blog/2015/07/24/finding-self-signed-tls-certificates-suricata-and-luajit-scripting

(continued from previous page)

```
      }
    }
  }
}
```

In some cases, you may have to replace *keyword* by *raw* in your search. You can access Query DSL filter by clicking + *Add filter* then *Edit as Query DSL*.

9.5.2 Unsecure protocol

Some TLS and SSL versions are considered to be unsecure due to design flaws and known successful attacks. Therefore, it is interesting to find any connection using this weak policy so any eye dropping can be prevented. Known unsecure versions are all SSL versions and TLS up to 1.1.

It is possible to search this Elasticsearch by using the following filter:

```
tls.version:SSL% OR tls.version:TLSv1 OR tls.version:"TLS 1.1"
```

In Splunk, this can be written as:

```
event_type=tls AND tls.version IN ("SSLv2", "SSLv3", "TLSv1", "TLS 1.1")
```

9.5.3 Expired certificates

The simplest way to achieve that is to use the *tls_cert_expired* keyword as seen in this signature:

```
alert tls any any -> any any (msg:"expired certs"; tls_cert_expired; sid:1; rev:1;)
```

But it is also possible to do this in Splunk:

```
event_type=tls |
eval tls_after_date = strptime('tls.notafter',"%Y-%m-%dT%H:%M:%S") |
eval event_time = strptime(timestamp,"%Y-%m-%dT%H:%M:%S.%6N%z") |
eval validity = tls_after_date - event_time |
search validity < 0 |
top tls.subject, tls.issuerdn, tls.notafter, timestamp, validity
```

The complex part consists of parsing the two time stamps we are interested in with *strptime* and then computing the validity. The result of the query is shown on Fig. 9.1.

9.5.4 TLS Cipher Suite analysis

The negotiated TLS Cipher Suites used in a network are interesting to monitor. They contain the set of algorithms used on TLS to protect the communication. The level of security and confidentiality provided by the various algorithms varies greatly. For instance, TLS_NULL_WITH_NULL_NULL is a valid TLS cipher suite and, yes, it means that nothing is done and the data is in clear text. While this is an extreme case, some other TLS cipher suites should be avoided like the one using the RC4 algorithm.

If this information is not directly available in Suricata TLS events, it is available as one of the TLS JA3S parameters. The second parameter of the JA3S string is the Cipher ID. This is an integer, as TLS is not sending a string over the wire. Nevertheless, this is interesting information anyway.

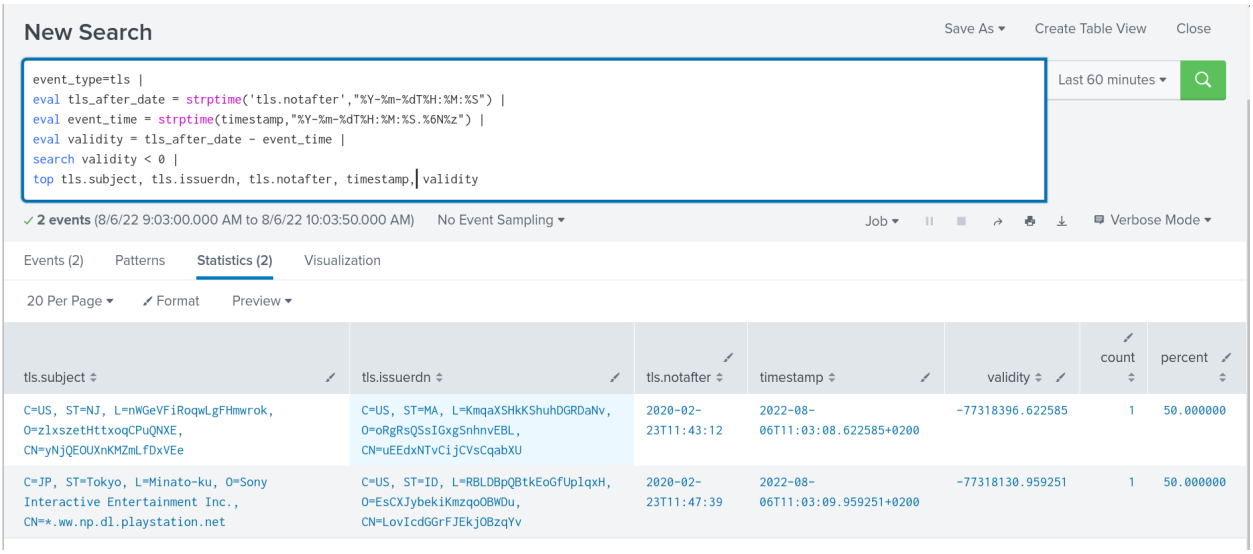


Fig. 9.1: Splunk search on expired certificates

We can use Splunk's extraction capabilities to get the value of the Cipher ID in a distinct field. All we need to do is to split the JA3S string and get the second element. This can be done as follows:

```
event_type=tls |
  spath tls.ja3s.string output=ja3s_string |
  eval ja3s_elt=split(ja3s_string,",") |
  eval cipher_id=mvindex(ja3s_elt, 1)
```

Getting from the ID to the string version of the TLS Cipher suite can then be done via a lookup table. It can be extracted from the IANA website. This mapping is available in the Stamus Splunk App[37] which also contains other interesting information.

The French National Cybersecurity Agency (ANSSI[38]) has published Security Recommendations for TLS[39] where a list of recommended TLS cipher suites is defined. Their classification also contains *degraded* TLS cipher suites that are ok to use if there are no alternatives. All other TLS cipher suites should be considered as insecure. The mapping included in the Stamus Splunk App contains this information in the lookup table, so it is possible to search and do statistics on the security of the TLS cipher suite seen on the network. For example, to list all insecure TLS connections seen on the network, one can do the following in Splunk:

```
event_type=tls |
  spath tls.ja3s.string output=ja3s_string |
  eval ja3s_elt=split(ja3s_string,",") |
  eval cipher_id=mvindex(ja3s_elt, 1) |
  lookup tls_cipher_mapping.csv id as cipher_id |
  search cipher_security=insecure
```

Here we add to the previous a call to the lookup followed by a search on the field *cipher_security* that is added by the lookup.

Using this technique, it is possible to build searches that classify the TLS cipher suites and display the insecure ones. This is available in one of the Stamus Splunk App dashboards as shown on Fig. 9.2.

[37] https://splunkbase.splunk.com/app/5262
[38] https://www.ssi.gouv.fr/
[39] https://www.ssi.gouv.fr/guide/recommandations-de-securite-relatives-a-tls/

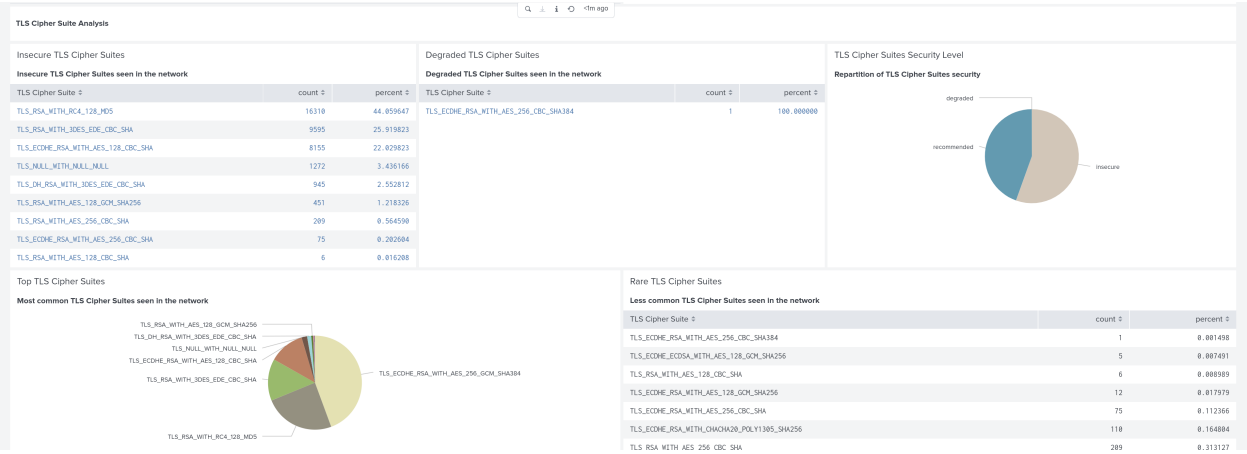


Fig. 9.2: TLS Cipher Suites analysis in Stamus Splunk App

CHAPTER

TEN

SMB DETECTION AND THREAT HUNTING

10.1 Introduction

SMB (Server Message Block) is a client-server communication protocol that has many implementations and is primarily used for sharing access to files, printers, and resources on the network. The Microsoft windows networks variant is known as Microsoft SMB Protocol. Other systems and OS types like Linux and Mac also include support for SMB.

There are many versions and history revisions

- SMB 1.0
- CIFS
- SMB 2.0
- SMB 2.1
- SMB 3.0
- SMB 3.0.2
- SMB 3.1.1

as well as third party implementations

- Samba
- Netsmb
- NQ
- MoSMB
- Fusion File Share by Tuxera
- Likewise

The implementation and the central internal usage of the protocol by many types of operating systems makes it an ideal medium to be used by threat actors for internal/lateral movement. Once a foothold is established, the actor can utilize built-in and default available functionalities.

10.2 Protocol overview

SMB Protocol functionality can also include the following

- Dialect negotiation
- Determining other Microsoft SMB Protocol servers on the network, or network browsing
- Printing over a network
- File, directory, and share access authentication
- File and record locking
- File and directory change notification
- Extended file attribute handling
- Unicode support

which makes it even more interesting and potent in terms of network visibility and monitoring.

10.3 SMB analysis in Suricata

Suricata supports protocol analysis and logging of all SMB versions like SMB 1.x, SMB 2.x and SMB 3.x. Since Suricata 6, SMB has been further improved thanks to community feedback and code donation.


```
{
  "timestamp": "2022-05-04T18:51:26.052278+0300",
  "flow_id": 1941808952834204,
  "pcap_cnt": 1189,
  "event_type": "smb",
  "src_ip": "10.136.0.69",
  "src_port": 49622,
  "dest_ip": "10.136.0.64",
  "dest_port": 445,
  "proto": "TCP",
  "pkt_src": "wire/pcap",
  "metadata": {
    "flowbits": [
      "ET.smbdcerpc.endians"
    ]
  },
  "smb": {
    "id": 85,
    "dialect": "3.11",
    "command": "SMB2_COMMAND_CREATE",
    "status": "STATUS_SUCCESS",
    "status_code": "0x0",
    "session_id": 52777564766265,
    "tree_id": 9,
    "filename": "PSEXESVC.exe",
    "disposition": "FILE_OPEN",
    "access": "normal",
    "created": 1651679428,
```


(continues on next page)

(continued from previous page)


```
    "accessed": 1651679428,
    "modified": 1651679428,
    "changed": 1651679428,
    "size": 383872,
    "fuid": "000002a0-000c-0000-0021-00000000000c"
  }
}
```


The `smb` object contains all the information about the specific SMB transaction. The `smb` object can be found in both `"event_type":"alert"` as supplemental metadata and as a stand alone SMB protocol log (`"event_type":"smb"`). It has detailed `key:value` field pairs giving information about the transaction. In the example above, `filename` is the name of the file accessed or transferred, `disposition` is instructing the action the server must take if the file already exists, `command` is containing the actual SMB command, and `status` has the return status of the command.


```
"smb": {
  "id": 3,
  "dialect": "3.11",
  "command": "SMB2_COMMAND_SESSION_SETUP",
  "status": "STATUS_SUCCESS",
  "status_code": "0x0",
  "session_id": 52777564766265,
  "tree_id": 0,
  "ntlmssp": {
    "domain": "STCONSULT",
    "user": "Administrator",
    "host": "PC1"
  }
```


Other useful information is also available depending on the different SMB transaction or request. In the example above we have information about a session setup with details about `domain` - the domain , `user` - the user establishing the session, and the `host` it is established from.


```
"smb": {
  "id": 73,
  "dialect": "3.11",
  "command": "SMB2_COMMAND_WRITE",
  "status": "STATUS_SUCCESS",
  "status_code": "0x0",
  "session_id": 52777564766265,
  "tree_id": 1,
  "dcerpc": {
    "request": "BIND",
    "response": "BINDACK",
    "interfaces": [
      {
        "uuid": "367abb81-9844-35f1-ad32-98f038001003",
        "version": "2.0",
        "ack_result": 0,
        "ack_reason": 0
      },
      {
        "uuid": "367abb81-9844-35f1-ad32-98f038001003",
```


(continues on next page)

(continued from previous page)

```
        "version": "2.0",
        "ack_result": 3,
        "ack_reason": 0
      }
    ],
    "call_id": 2
  }
```

We can also count on Suricata to give us any specific data on top of SMB , like DCERPC and specific Microsoft protocol UUID (`uuid` key).

Note: Check the eve SMB format[40] page in Suricata manual for more information on the SMB events.

10.4 SMB and detection

10.4.1 SMB keywords

Out of the box, Suricata supports the following keywords in alerts for matching inside the SMB transactions, all of which are sticky buffers:

- dcerpc.iface: Match on the UUID of the protocol
- dcerpc.opnum: Match on the opnum of the protocol
- dcerpc.stub_data: Match on the stub data (data/arguments of the remote call)
- smb.named_pipe: Match on SMB named pipe in tree connect
- smb.share: Match on SMB share name in tree connect

These keywords can be used in rules matching. It is important to note that those keywords are separate from the protocol fields matching that can further be used in SIEM queries of the SMB protocol logs produced by Suricata.

10.5 Hunting on SMB events

10.5.1 SMB Scheduled task created remotely

Hunting on SMB events is a big task, and to be more potent and successful it also needs infrastructure and organizational local knowledge. As an example, it might be interesting to know, highlight, and investigate when a `Scheduled Task` is created remotely. This is indeed a task that is definitely only done by some advanced system administrators and by some attackers.

For that we can use the following rule:

```
alert smb any any -> any any ( \\
   msg: "SN MS Scheduled task created remotely"; \\
   flow: to_server, established; \\
   dcerpc.iface:378E52B0-C0A9-11CF-822D-00AA0051E40F; dcerpc.opnum:0; \\
   reference:url,https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-tsch/
```

(continues on next page)

[40] https://suricata.readthedocs.io/en/latest/output/eve/eve-json-format.html?highlight=http#event-type-tls

(continued from previous page)

```
↪4d44c426-fad2-4cc7-9677-bfcd235dca33; \\
  metadata:created_at 2022_09_20, updated_at 2022_09_20; \\
  target:dest_ip; \\
  sid:1000001; rev:1;)
```

The resulting alert event log could look as follows, please note the `flow` and `smb` subsections of the alert event:


```
{
  "stream": 1,
  "ether": {
    "dest_mac": "ff:ff:ff:28:fe:2d",
    "src_mac": "ff:ff:ff:7a:71:40"
  },
  "timestamp": "2022-09-27T20:04:27.911458+0200",
  "dest_ip": "10.10.11.15",
  "tx_id": 9,
  "packet_info": {
    "linktype": 1
  },
  "flow_id": 1056255386940814,
  "flow": {
    "dest_ip": "10.10.11.15",
    "src_ip": "10.10.22.55",
    "pkts_toserver": 17,
    "pkts_toclient": 15,
    "bytes_toserver": 3983,
    "bytes_toclient": 3240,
    "start": "2022-09-27T20:04:27.311464+0200",
    "src_port": 55067,
    "dest_port": 445
  },
  "type": "json-log",
  "in_iface": "eth0",
  "app_proto": "smb",
  "metadata": {
    "flowbits": [
      "ET.smbdcerpc.endians"
    ]
  },
  "src_ip": "10.10.22.55",
  "alert": {
    "metadata": {
      "created_at": [
        "2022_09_20"
      ],
      "updated_at": [
        "2022_09_20"
      ]
    },
    "rev": 1,
    "source": {
      "port": 55067,
```


(continues on next page)

(continued from previous page)


```
      "ip": "10.10.22.55"
    },
    "action": "allowed",
    "gid": 1,
    "category": "",
    "severity": 3,
    "target": {
      "port": 445,
      "ip": "10.10.11.15"
    },
    "signature_id": 1000001,
    "lateral": "intranet",
    "signature": "SN MS Scheduled task created remotely"
  },
  "event_type": "alert",
  "@version": "1",
  "input": {
    "type": "log"
  },
  "dest_port": 445,
  "@timestamp": "2022-09-27T18:04:27.911Z",
  "proto": "TCP",
  "src_port": 55067,
  "smb": {
    "id": 10,
    "tree_id": 1,
    "session_id": 17607151321153,
    "dialect": "3.11",
    "dcerpc": {
      "response": "UNREPLIED",
      "request": "REQUEST",
      "req": {
        "stub_data_size": 264,
        "frag_cnt": 1
      },
      "call_id": 2,
      "opnum": 0
    },
    "command": "SMB2_COMMAND_IOCTL",
    "status": "STATUS_PENDING",
    "status_code": "0x103"
  }
}
```

10.5.2 SMB Status Access Denied

Access denied in SMB could be common occurrences in cases when creating or connecting to a shared directory via the tree connect operation:


```
{
  "timestamp": "2022-05-20T20:31:58.553243+0200",
  "flow_id": 1047258484058895,
  "event_type": "smb",
  "src_ip": "10.150.1.93",
  "src_port": 52092,
  "dest_ip": "10.150.1.46",
  "dest_port": 445,
  "proto": "TCP",
  "pkt_src": "wire/pcap",
  "metadata": {
    "flowbits": [
      "ET.smbdcerpc.endians",
      "ET.dcerpc.mssrvs",
      "ET.smb.binary"
    ]
  },
  "smb": {
    "id": 54,
    "dialect": "3.11",
    "command": "SMB2_COMMAND_TREE_CONNECT",
    "status": "STATUS_ACCESS_DENIED",
    "status_code": "0xc0000022",
    "session_id": 30786459795473,
    "tree_id": 0,
    "share": "\\\\WZVCDYTZUR6.GONE.LOCAL\\C$",
    "share_type": "UNKNOWN"
  }
}
```


However, what could be interesting is using the SMB protocol and flow transaction data in Suricata to detect brute forcing. The idea is to highlight all SMB flows that have many `STATUS_ACCESS_DENIED` command results in the same flow indicating possible brute forcing.

This could be achieved by combining 2 Suricata log fields - mainly `flow_id` and `smb.status`. We can use that combination as `flow_id` contains the Suricata native unique flow identifier which can be used to correlate events such as alerts, flows, file transactions, and protocol logs from the same flow.

JQ command line query

```
jq 'select(.event_type=="smb" and .smb.status == "STATUS_ACCESS_DENIED")|.flow_id' /var/
↪log/suricata/eve.json | sort | uniq -c
10 1047258484058895
```

The JQ query above returns a result of 10 `STATUS_ACCESS_DENIED` statuses in the flow whose `flow_id` is `1047258484058895`. So we have 10 instances of Denied Access in the same flow which is definitely suspicious.

Kibana query

Create a table visualisation that uses an aggregation in Kibana on the field `flow_id` with the following query search:

```
event_type:"smb" AND smb.status:"STATUS_ACCESS_DENIED"
```

Splunk query

Similar for Splunk the query can be:

```
event_type=smb sourcetype="suricata:smb" smb.status=STATUS_ACCESS_DENIED |
    table src_ip, dest_ip, flow_id |
    stats count by src_ip,dest_ip,flow_id |
    sort - count
```

CHAPTER

ELEVEN

HTTP DETECTION AND THREAT HUNTING

11.1 Introduction

HTTP is running the world. It is used by human's actions or, in the case of HTTPS protocol, directly or below TLS. It is also widely used by systems via REST API and other inter-server communications.

One of the great benefits of HTTP is the weak message structure which makes it easy to develop a client. It is a loose text-based protocol and as such looks very similar to free text. This makes it highly adaptable, but from a security point of view this complicates things. Hunting something that has multiple forms can be rather complex.

11.2 Protocol overview

In HTTP, the client is the first to send data via an HTTP request. This message contains headers with a few mandatory fields and a lot of optional headers which give more context to the server about the request so it can adapt its answer. The request contains an optional body. The server responds with an answer that has the same structure with headers and a body. This is because HTTP is focused on getting information from the server.

To see an example of the minimum requirements of a request is, let's look at this minimal request to google done via netcat[41] where we ask for the home page `/` with protocol version `1.1`:

```
# nc -v google.fr 80
GET / HTTP/1.1
```

This is the answer from Google:

```
HTTP/1.1 200 OK
Date: Sun, 25 Sep 2022 21:17:08 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
Server: gws
X-XSS-Protection: 0
X-Frame-Options: SAMEORIGIN
Set-Cookie: AEC=AakniGO859M8HPupnneVpexM15eeWdGOBL_LX5TGiy5GsqI_Fnm0F8UEIg; expires=Fri,␣
↪24-Mar-2023 21:17:08 GMT; path=/; domain=.google.com; Secure; HttpOnly; SameSite=lax
Accept-Ranges: none
Vary: Accept-Encoding
Transfer-Encoding: chunked
```

(continues on next page)

[41] https://en.wikipedia.org/wiki/Netcat

(continued from previous page)

```
5acf
<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" ...
```

This answer has the typical structure of an HTTP message with status line (here `HTTP/1.1 200 OK`), followed by the headers (key and value), then an empty line that is followed by the body.

This dissymmetry between the request and the response in this example emphasizes one of the main concepts of HTTP design: it should work even if client implementation is really poor.

If we look at the same HTTP request to google.fr done via Firefox, we have the following request:

```
GET / HTTP/1.1
Host: google.fr
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101 Firefox/102.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;
↪q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
```

The headers list is far longer and gives more information about what the client is able to support or what it wants from the server's answer. For example, here, because we have the header `Upgrade-Insecure-Requests` set to 1, we don't get the web page content as we got in the previous request but we have a redirection to the Secure HTTPS version of google.fr:

```
HTTP/1.1 301 Moved Permanently
Location: http://www.google.fr/
Content-Type: text/html; charset=UTF-8
Date: Sun, 25 Sep 2022 21:33:01 GMT
Expires: Tue, 25 Oct 2022 21:33:01 GMT
Cache-Control: public, max-age=2592000
Server: gws
Content-Length: 218
X-XSS-Protection: 0
X-Frame-Options: SAMEORIGIN
```

As we will see later, the fact that a lot of freedom is given in the protocol is a key point in profiling non-regular behavior that does not follow the implicit norm.

11.3 HTTP analysis in Suricata

Suricata has very robust support for HTTP. The development of the parser was initiated at the beginning of the project and has continued to evolve with continuing update releases.

HTTP request and response are logged in a single event:

```
{
  "timestamp": "2019-07-05T22:06:30.877497+0200",
  "flow_id": 1831154258612572,
  "pcap_cnt": 47339,
```

(continues on next page)

(continued from previous page)


```
  "event_type": "http",
  "src_ip": "10.7.5.5",
  "src_port": 62152,
  "dest_ip": "198.12.71.157",
  "dest_port": 443,
  "proto": "TCP",
  "pkt_src": "wire/pcap",
  "tx_id": 0,
  "http": {
    "hostname": "198.12.71.157",
    "http_port": 443,
    "url": "/login/process.php",
    "http_user_agent": "Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; rv:11.0) like␣
↪Gecko",
    "http_content_type": "text/html",
    "http_method": "GET",
    "protocol": "HTTP/1.1",
    "status": 200,
    "length": 173
  }
}
```


The `http` object contains all the information about the request and the response. Fields like `hostname` or `http_user_agent` are coming from the client and fields such as `status`, `length`, or `http_content_type` are coming from the server. The log also include the `tx_id` which stands for transaction identifier. It is giving the number of HTTP transaction (request + response) seen on the flow at the moment of the request. In this example it is 0, which means this is the first one.

As you can see, the event shown here does not contain all the headers. The dump of all headers can be activated in the configuration via the `dump-all-headers` configuration in the HTTP logging. This will provide far more information, but it is also going to be far more verbose:


```
"request_headers": [
  {
    "name": "Cookie",
    "value": "session=okmKYUc4i80CZ2Rflxy91qtVJoI="
  },
  {
    "name": "User-Agent",
    "value": "Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; rv:11.0) like Gecko"
  },
  {
    "name": "Host",
    "value": "198.12.71.157:443"
  },
  {
    "name": "Connection",
    "value": "Keep-Alive"
  }
],
"response_headers": [
  {
    "name": "Content-Type",
```


(continues on next page)

(continued from previous page)


```
    "value": "text/html; charset=utf-8"
  },
  {
    "name": "Content-Length",
    "value": "173"
  },
  {
    "name": "Cache-Control",
    "value": "no-cache, no-store, must-revalidate"
  },
  {
    "name": "Pragma",
    "value": "no-cache"
  },
  {
    "name": "Expires",
    "value": "0"
  },
  {
    "name": "Server",
    "value": "Microsoft-IIS/7.5"
  },
  {
    "name": "Date",
    "value": "Fri, 05 Jul 2019 20:06:30 GMT"
  }
]
```


Another interesting feature of HTTP support in Suricata is the transparent decompression of the HTTP response body. If the client supports the feature, the server can return the object asked for by the client in a compressed form to downsize the transfer. The result is that the content of the HTTP body in the TCP stream is just compression noise. Suricata decompresses the data in real-time and provides the decompressed content to the keyword and layers that are using the HTTP response body.

The HTTP response body can be logged in alerts and this greatly improves the context provided as the stream TCP cannot be read by a human.

Note: Check the eve HTTP format[42] page in Suricata manual for more information on the HTTP events.

Suricata supports file extraction over HTTP, so any of the techniques and information of *File Analysis* chapter apply here.

[42] https://suricata.readthedocs.io/en/latest/output/eve/eve-json-format.html?highlight=http#event-type-http

11.4 HTTP and detection

11.4.1 HTTP keywords

Suricata has more than 25 sticky buffer keywords to match on HTTP fields, covering most of the headers and the content. These last ones are interesting, specifically `http.response_body` that matches on the body of the response sent by the server. As described in the previous chapter, the content sent by the server can be on a compressed form and Suricata will provide the decompressed version to the detection engine.

Most keywords match on a normalized field. This is really convenient as the rules writer does not have to take the possible variant into account. For example, the `http.host` keyword is normalized and will always be lowercase. This prevents trivial evasion of detection by connecting to *BaDdoMAin.OrG* instead of the regular *baddomain.org*.

In some cases, the characteristic seen in the traffic is dependant of the content seen on the wire. For this reason, Suricata is providing some alternate keywords to match on the raw, unnormalized content. For example, `http.host.raw` will match on the HTTP host in its raw form.

11.4.2 Cookbook

Match on a domain and its subdomains

A domain is known to be malicious and we want to alert on all requests to this domain or any of its subdomains:

```
alert http any any -> any any (msg:"Bad domain"; \\
     http.host; dotprefix; content:".pandabear.gov"; endswith;
     sid:1; rev:1;)
```

The match is obtained by using the sticky buffer `http.host` to match on the HTTP host sent by the client. By using `dotprefix`, a `.` will be prepended to the buffer so it will not match on `lovelypandabear.gov`. Then the signature uses the `endswith` keyword to ensure the string ends with the specified content. It will prevent a match on a domain like `pandabear.governed.org`.

Checking malicious HTTP user agent

Some variants of Trickbot are using an HTTP user agent that is set to `test`. A signature to detect this behavior could be:

```
alert http any any -> any any (msg:"Bad domain"; \\
     http.user_agent; content:"test"; startswith; endswith;
     sid:1; rev:1;)
```

We use the same technique as the domain with the `endswith` keyword that we complement with `startswith` to ensure full equality of the strings.

Clear text authentication and password extraction

Clear text authentication over HTTP is still relevant in some environments. Detecting this behavior and collecting the user and password to check them against other systems to detect credential reuse is really interesting.

This can be done with a single signature:

```
alert http any any -> any any (msg:"HTTP unencrypted with password"; \\
     http.header; content:"Authorization|3a 20|Basic"; nocase; \\
     base64_decode:bytes 0, offset 1,relative; \\
     base64_data; pcre:"/([^:]+):(.+)/,flow:user,flow:password"; \\
     sid:1; rev:1;)
```

This signature first checks for the *Authorization* header and then uses `base64_decode` to convert the content from base64 to regular encoding. The `base64_data` is a sticky buffer to access the content transformed by `base64_decode`. In this buffer, we have the user name followed by the password so we can extract it via a regular expression using the `pcre` keyword.

The regular expression is really interesting as it uses the data extraction feature of Suricata:

```
pcre:"/([^:]+):(.+)/,flow:user,flow:password"
```

The regular expression has 2 groups *([^:]+)* and *(.+)*. The first one gets everything before the *:* and the second one take the rest. So the first group retrieves the user and second extracts the password. The magic appends in the modifiers: `,flow:user,flow:password`. This is a Suricata extension. It is stating here that the first group should be stored in a flow variable named `user` and that second group should be stored in a flow variable named `password`.

Doing this, the alert is augmented with a `metadata` object that contains a `flowvars` with the extracted values as shown below:


```
{
  "timestamp": "2022-01-07T15:13:40.947137+0100",
  "flow_id": 206063044707455,
  "pcap_cnt": 69,
  "event_type": "alert",
  "src_ip": "192.10.0.1",
  "src_port": 58944,
  "dest_ip": "192.10.0.2",
  "dest_port": 80,
  "proto": "TCP",
  "metadata": {
    "flowvars": [
      {
        "user": "regit"
      },
      {
        "password": "ILoveSuri"
      }
    ]
  },
```

11.5 Hunting on HTTP events

11.5.1 HTTP hunting signatures in ETOpen and ETPro

This is not a technique to hunt directly using application layer events, but the ETOpen and ETPro ruleset[43] contains a few hundred particularly interesting hunting signatures for the HTTP protocol. Enabling these signatures and considering them as pre-executed queries is highly recommended.

For example, the following signature matches on POST request using an IPv4 address as hostname and missing headers that are usually sent by regular browsers.

```
alert http $HOME_NET any -> $EXTERNAL_NET any ( \\
      msg:"ET HUNTING GENERIC SUSPICIOUS POST to Dotted Quad with Fake Browser 2"; \\
      flow:established,to_server; \\
      http.method; content:"POST";
      http.user_agent; content:"|20|Firefox/"; nocase; fast_pattern; \\
      http.host; pcre:"/^(?:\d{1,3}\.){3}\d{1,3}/"; \\
      http.header_names; content:"|0d 0a|Host|0d 0a|"; depth:8; \\
         content:!"Accept-Encoding"; \\
         content:!"Referer"; \\
         content:!"X-Requested-With"; nocase; \\
      classtype:bad-unknown; sid:2018359; rev:4; \\
      metadata:created_at 2014_04_04, former_category INFO, updated_at 2020_08_20;)
```

This signature is interesting because it matches the Tactics, Techniques, and Procedures of some actors without having to know the threat.

11.5.2 Rare HTTP user agents

As HTTP is frequently seen on network, using the rare approach is often a good way to see outliers that can be interesting to investigate.

This can be done in Splunk via the following query:

```
search event_type="http" | rare http.http_user_agent | sort count | head 10
```

11.5.3 Rare HTTP hosts queried without referrer

The list of hosts used as an entry point when browsing is fairly small in most environments. Getting the rarest one is interesting because it will exhibit potential unwanted behavior such as payload download.

This can be done in Splunk via the following query:

```
event_type="http" AND NOT http.http_refer=* | rare http.hostname | sort count
```

[43] https://www.proofpoint.com/us/resources/data-sheets/etpro-versus-et-open-ruleset-comparison

11.5.4 HTTP errors with Abnormal Content Length

Some attackers try to hide their exchange by pretending the requests are failing. As unfound pages are usually fairly small, looking at error pages with a decent size is a good start for a hunt.

This can be done in Splunk via the following query:

```
event_type="http" http.status=4* http.length>=10000 |
    sort -http.length |
    table src_ip, dest_ip, http.hostname, http.status, http.url, http.length
```

Kibana users can use the following search using Lucene syntax:

```
event_type:http AND http.status:>400 AND http.status:<500 AND http.length:>10000
```

CHAPTER

TWELVE

ABOUT

12.1 Authors and contributors

This document has been written by Éric Leblond and Peter Manev with the help of Mark Durrett, Dallon Robinette, and Phil Owens from Stamus Networks.

12.1.1 Éric Leblond

Éric Leblond is the co-founder and chief technology officer (CTO) of Stamus Networks and a member of the board of directors at Open Network Security Foundation (OISF). Éric has more than 15 years of experience as co-founder and technologist of cybersecurity software companies and is an active member of the security and open-source communities. He has worked on the development of Suricata – the open-source network threat detection engine – since 2009 and is part of the Netfilter Core team, responsible for the Linux kernel's firewall layer. Éric is also the lead developer of the Suricata Language Server, a real-time syntax checking and autocomplete app for Suricata rule writers. Éric is a well-respected expert and speaker on network security.

12.1.2 Peter Manev

Peter Manev is the co-founder and chief strategy officer (CSO) of Stamus Networks and a member of the executive team at Open Network Security Foundation (OISF). Peter has over 15 years of experience in the IT industry, including enterprise-level IT security practice. He is a passionate user, developer, and explorer of innovative open-source security software. He is responsible for training as well as quality assurance and testing on the development team of Suricata – the open-source threat detection engine. Peter is also the lead developer of SELKS, the popular turnkey open-source implementation of Suricata. Peter is a regular speaker and educator on open-source security, threat hunting, and network security.

12.2 License

12.2.1 Creative Commons Attribution-ShareAlike 4.0 International

Creative Commons Corporation ("Creative Commons") is not a law firm and does not provide legal services or legal advice. Distribution of Creative Commons public licenses does not create a lawyer-client or other relationship. Creative Commons makes its licenses and related information available on an "as-is" basis. Creative Commons gives no warranties regarding its licenses, any material licensed under their terms and conditions, or any related information. Creative Commons disclaims all liability for damages resulting from their use to the fullest extent possible.

Using Creative Commons Public Licenses

Creative Commons public licenses provide a standard set of terms and conditions that creators and other rights holders may use to share original works of authorship and other material subject to copyright and certain other rights specified in the public license below. The following considerations are for informational purposes only, are not exhaustive, and do not form part of our licenses.

- **Considerations for licensors:** Our public licenses are intended for use by those authorized to give the public permission to use material in ways otherwise restricted by copyright and certain other rights. Our licenses are irrevocable. Licensors should read and understand the terms and conditions of the license they choose before applying it. Licensors should also secure all rights necessary before applying our licenses so that the public can reuse the material as expected. Licensors should clearly mark any material not subject to the license. This includes other CC-licensed material, or material used under an exception or limitation to copyright. More considerations for licensors[44].
- **Considerations for the public:** By using one of our public licenses, a licensor grants the public permission to use the licensed material under specified terms and conditions. If the licensor's permission is not necessary for any reason–for example, because of any applicable exception or limitation to copyright–then that use is not regulated by the license. Our licenses grant only permissions under copyright and certain other rights that a licensor has authority to grant. Use of the licensed material may still be restricted for other reasons, including because others have copyright or other rights in the material. A licensor may make special requests, such as asking that all changes be marked or described. Although not required by our licenses, you are encouraged to respect those requests where reasonable. More considerations for the public[45].

12.2.2 Creative Commons Attribution-ShareAlike 4.0 International Public License

By exercising the Licensed Rights (defined below), You accept and agree to be bound by the terms and conditions of this Creative Commons Attribution-ShareAlike 4.0 International Public License ("Public License"). To the extent this Public License may be interpreted as a contract, You are granted the Licensed Rights in consideration of Your acceptance of these terms and conditions, and the Licensor grants You such rights in consideration of benefits the Licensor receives from making the Licensed Material available under these terms and conditions.

Section 1 – Definitions.

a. **Adapted Material** means material subject to Copyright and Similar Rights that is derived from or based upon the Licensed Material and in which the Licensed Material is translated, altered, arranged, transformed, or otherwise modified in a manner requiring permission under the Copyright and Similar Rights held by the Licensor. For purposes of this Public License, where the Licensed Material is a musical work, performance, or sound recording, Adapted Material is always produced where the Licensed Material is synched in timed relation with a moving image.

b. **Adapter's License** means the license You apply to Your Copyright and Similar Rights in Your contributions to Adapted Material in accordance with the terms and conditions of this Public License.

[44] http://wiki.creativecommons.org/Considerations_for_licensors_and_licensees#Considerations_for_licensors
[45] http://wiki.creativecommons.org/Considerations_for_licensors_and_licensees#Considerations_for_licensees

c. **BY-SA Compatible License** means a license listed at creativecommons.org/compatiblelicenses, approved by Creative Commons as essentially the equivalent of this Public License.

d. **Copyright and Similar Rights** means copyright and/or similar rights closely related to copyright including, without limitation, performance, broadcast, sound recording, and Sui Generis Database Rights, without regard to how the rights are labeled or categorized. For purposes of this Public License, the rights specified in Section 2(b)(1)-(2) are not Copyright and Similar Rights.

e. **Effective Technological Measures** means those measures that, in the absence of proper authority, may not be circumvented under laws fulfilling obligations under Article 11 of the WIPO Copyright Treaty adopted on December 20, 1996, and/or similar international agreements.

f. **Exceptions and Limitations** means fair use, fair dealing, and/or any other exception or limitation to Copyright and Similar Rights that applies to Your use of the Licensed Material.

g. **License Elements** means the license attributes listed in the name of a Creative Commons Public License. The License Elements of this Public License are Attribution and ShareAlike.

h. **Licensed Material** means the artistic or literary work, database, or other material to which the Licensor applied this Public License.

i. **Licensed Rights** means the rights granted to You subject to the terms and conditions of this Public License, which are limited to all Copyright and Similar Rights that apply to Your use of the Licensed Material and that the Licensor has authority to license.

j. **Licensor** means the individual(s) or entity(ies) granting rights under this Public License.

k. **Share** means to provide material to the public by any means or process that requires permission under the Licensed Rights, such as reproduction, public display, public performance, distribution, dissemination, communication, or importation, and to make material available to the public including in ways that members of the public may access the material from a place and at a time individually chosen by them.

l. **Sui Generis Database Rights** means rights other than copyright resulting from Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, as amended and/or succeeded, as well as other essentially equivalent rights anywhere in the world.

m. **You** means the individual or entity exercising the Licensed Rights under this Public License. **Your** has a corresponding meaning.

Section 2 – Scope.

a. **License grant.**

1. Subject to the terms and conditions of this Public License, the Licensor hereby grants You a worldwide, royalty-free, non-sublicensable, non-exclusive, irrevocable license to exercise the Licensed Rights in the Licensed Material to:

A. reproduce and Share the Licensed Material, in whole or in part; and

B. produce, reproduce, and Share Adapted Material.

2. **Exceptions and Limitations.** For the avoidance of doubt, where Exceptions and Limitations apply to Your use, this Public License does not apply, and You do not need to comply with its terms and conditions.

3. **Term.** The term of this Public License is specified in Section 6(a).

4. **Media and formats; technical modifications allowed.** The Licensor authorizes You to exercise the Licensed Rights in all media and formats whether now known or hereafter created, and to make technical modifications necessary to do so. The Licensor waives and/or agrees not to assert any right

or authority to forbid You from making technical modifications necessary to exercise the Licensed Rights, including technical modifications necessary to circumvent Effective Technological Measures. For purposes of this Public License, simply making modifications authorized by this Section 2(a)(4) never produces Adapted Material.

5. **Downstream recipients.**

A. **Offer from the Licensor – Licensed Material.** Every recipient of the Licensed Material automatically receives an offer from the Licensor to exercise the Licensed Rights under the terms and conditions of this Public License.

B. **Additional offer from the Licensor – Adapted Material.** Every recipient of Adapted Material from You automatically receives an offer from the Licensor to exercise the Licensed Rights in the Adapted Material under the conditions of the Adapter's License You apply.

C. **No downstream restrictions.** You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, the Licensed Material if doing so restricts exercise of the Licensed Rights by any recipient of the Licensed Material.

6. **No endorsement.** Nothing in this Public License constitutes or may be construed as permission to assert or imply that You are, or that Your use of the Licensed Material is, connected with, or sponsored, endorsed, or granted official status by, the Licensor or others designated to receive attribution as provided in Section 3(a)(1)(A)(i).

b. **Other rights.**

1. Moral rights, such as the right of integrity, are not licensed under this Public License, nor are publicity, privacy, and/or other similar personality rights; however, to the extent possible, the Licensor waives and/or agrees not to assert any such rights held by the Licensor to the limited extent necessary to allow You to exercise the Licensed Rights, but not otherwise.
2. Patent and trademark rights are not licensed under this Public License.
3. To the extent possible, the Licensor waives any right to collect royalties from You for the exercise of the Licensed Rights, whether directly or through a collecting society under any voluntary or waivable statutory or compulsory licensing scheme. In all other cases the Licensor expressly reserves any right to collect such royalties.

Section 3 – License Conditions.

Your exercise of the Licensed Rights is expressly made subject to the following conditions.

a. **Attribution.**

1. If You Share the Licensed Material (including in modified form), You must:

A. retain the following if it is supplied by the Licensor with the Licensed Material:

i. identification of the creator(s) of the Licensed Material and any others designated to receive attribution, in any reasonable manner requested by the Licensor (including by pseudonym if designated);

ii. a copyright notice;

iii. a notice that refers to this Public License;

iv. a notice that refers to the disclaimer of warranties;

v. a URI or hyperlink to the Licensed Material to the extent reasonably practicable;

B. indicate if You modified the Licensed Material and retain an indication of any previous modifications; and

C. indicate the Licensed Material is licensed under this Public License, and include the text of, or the URI or hyperlink to, this Public License.

2. You may satisfy the conditions in Section 3(a)(1) in any reasonable manner based on the medium, means, and context in which You Share the Licensed Material. For example, it may be reasonable to satisfy the conditions by providing a URI or hyperlink to a resource that includes the required information.

3. If requested by the Licensor, You must remove any of the information required by Section 3(a)(1)(A) to the extent reasonably practicable.

b. **ShareAlike.**

In addition to the conditions in Section 3(a), if You Share Adapted Material You produce, the following conditions also apply.

1. The Adapter's License You apply must be a Creative Commons license with the same License Elements, this version or later, or a BY-SA Compatible License.

2. You must include the text of, or the URI or hyperlink to, the Adapter's License You apply. You may satisfy this condition in any reasonable manner based on the medium, means, and context in which You Share Adapted Material.

3. You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, Adapted Material that restrict exercise of the rights granted under the Adapter's License You apply.

Section 4 – Sui Generis Database Rights.

Where the Licensed Rights include Sui Generis Database Rights that apply to Your use of the Licensed Material:

a. for the avoidance of doubt, Section 2(a)(1) grants You the right to extract, reuse, reproduce, and Share all or a substantial portion of the contents of the database;

b. if You include all or a substantial portion of the database contents in a database in which You have Sui Generis Database Rights, then the database in which You have Sui Generis Database Rights (but not its individual contents) is Adapted Material, including for purposes of Section 3(b); and

c. You must comply with the conditions in Section 3(a) if You Share all or a substantial portion of the contents of the database.

For the avoidance of doubt, this Section 4 supplements and does not replace Your obligations under this Public License where the Licensed Rights include other Copyright and Similar Rights.

Section 5 – Disclaimer of Warranties and Limitation of Liability.

a. **Unless otherwise separately undertaken by the Licensor, to the extent possible, the Licensor offers the Licensed Material as-is and as-available, and makes no representations or warranties of any kind concerning the Licensed Material, whether express, implied, statutory, or other. This includes, without limitation, warranties of title, merchantability, fitness for a particular purpose, non-infringement, absence of latent or other defects, accuracy, or the presence or absence of errors, whether or not known or discoverable. Where disclaimers of warranties are not allowed in full or in part, this disclaimer may not apply to You.**

b. **To the extent possible, in no event will the Licensor be liable to You on any legal theory (including, without limitation, negligence) or otherwise for any direct, special, indirect, incidental, consequential, punitive, exemplary, or other losses, costs, expenses, or damages arising out of this Public License or use of the Licensed Material, even if the Licensor has been advised of the possibility of such losses, costs, expenses, or damages. Where a limitation of liability is not allowed in full or in part, this limitation may not apply to You.**

c. The disclaimer of warranties and limitation of liability provided above shall be interpreted in a manner that, to the extent possible, most closely approximates an absolute disclaimer and waiver of all liability.

Section 6 – Term and Termination.

a. This Public License applies for the term of the Copyright and Similar Rights licensed here. However, if You fail to comply with this Public License, then Your rights under this Public License terminate automatically.

b. Where Your right to use the Licensed Material has terminated under Section 6(a), it reinstates:

   1. automatically as of the date the violation is cured, provided it is cured within 30 days of Your discovery of the violation; or

   2. upon express reinstatement by the Licensor.

   For the avoidance of doubt, this Section 6(b) does not affect any right the Licensor may have to seek remedies for Your violations of this Public License.

c. For the avoidance of doubt, the Licensor may also offer the Licensed Material under separate terms or conditions or stop distributing the Licensed Material at any time; however, doing so will not terminate this Public License.

d. Sections 1, 5, 6, 7, and 8 survive termination of this Public License.

Section 7 – Other Terms and Conditions.

a. The Licensor shall not be bound by any additional or different terms or conditions communicated by You unless expressly agreed.

b. Any arrangements, understandings, or agreements regarding the Licensed Material not stated herein are separate from and independent of the terms and conditions of this Public License.t stated herein are separate from and independent of the terms and conditions of this Public License.

Section 8 – Interpretation.

a. For the avoidance of doubt, this Public License does not, and shall not be interpreted to, reduce, limit, restrict, or impose conditions on any use of the Licensed Material that could lawfully be made without permission under this Public License.

b. To the extent possible, if any provision of this Public License is deemed unenforceable, it shall be automatically reformed to the minimum extent necessary to make it enforceable. If the provision cannot be reformed, it shall be severed from this Public License without affecting the enforceability of the remaining terms and conditions.

c. No term or condition of this Public License will be waived and no failure to comply consented to unless expressly agreed to by the Licensor.

d. Nothing in this Public License constitutes or may be interpreted as a limitation upon, or waiver of, any privileges and immunities that apply to the Licensor or You, including from the legal processes of any jurisdiction or authority.

```
Creative Commons is not a party to its public licenses. Notwithstanding, Creative Commons
may elect to apply one of its public licenses to material it publishes and in those
instances will be considered the “Licensor.” Except for the limited purpose of indicating
that material is shared under a Creative Commons public license or as otherwise permitted
by the Creative Commons policies published at creativecommons.org/policies, Creative
Commons does not authorize the use of the trademark “Creative Commons” or any other
trademark or logo of Creative Commons without its prior written consent including,
without limitation, in connection with any unauthorized modifications to any of its
```

public licenses or any other arrangements, understandings, or agreements concerning use of licensed material. For the avoidance of doubt, this paragraph does not form part of the public licenses.

Creative Commons may be contacted at creativecommons.org.

INDEX

ABOUT STAMUS NETWORKS

Stamus Networks believes in a world where defenders are heroes, and a future where those they protect remain safe. As defenders face an onslaught of threats from well-funded adversaries, we relentlessly pursue solutions that make the defender's job easier and more impactful. A global provider of high-performance network-based threat detection and response systems, Stamus Networks helps enterprise security teams accelerate their response to critical threats with solutions that uncover serious and imminent risk from network activity. Our advanced network detection and response (NDR) solutions expose threats to critical assets and empower rapid response.

5 Avenue Ingres
75016 Paris
France

450 E 96th St. Suite 500
Indianapolis, IN 46240
United States

Mail: contact@stamus-networks.com
Web: www.stamus-networks.com

ISBN: 979-8-9871510-1-3