Hunting Koadic Pt. 2 - JARM Fingerprinting

blog.tofile.dev/2020/11/28/koadic_jarm.html

pat_h/to/file November 28, 2020

Nov 28, 2020

A year ago, I looked into Koadic C2, a post-exploitation tool used by APTs and Pentesters alike.

At the time, I was able to fingerprint all non-fronted Koadic C2 servers across the internet, due to mismatches in it’s 404 page, where it fails to make the server look like a benign Apache web server.

Koadic hasn’t had many updates in the last 12 months, but in the meantime Salesforce have released JARM, a method of fingerprinting TLS Servers. According to Salesforce, it works by querying the server in various ways to gather information about things such:

- Operating system
- Operating system version
- Libraries used
- Versions of those libraries used
- The order in which the libraries were called
- Custom configuration

I wanted to revisit Koadic, and see if JARM would be able to pick up the C2 server as good as or better than my 404 analysis.

Step 1. Get Koadic’s JARM

Koadic C2 is written in Python, so already what OS we run it on will have an impact on it’s JARM signature. But I decided to run it on an Ubuntu server, as that was probably a common place it was deployed.

First I started a Koadic Server, making sure to run it over HTTPS:

```
# First Create TLS keypair for the websever
# enter a password and enter defaults for everything else
sudo apt-get install openssl git python3 python3-pip
openssl req -x509 -newkey rsa:4096 -days 365 -keyout koadic-key.pem -out koadic-
cert.pem

# Clone Koadic and install dependecies
git clone https://github.com/zerosum0x0/koadic.git
cd koadic
pip3 install -r requirements.txt

# Start Koadic and run the server in in HTTPS mode:
./koadic
set SRVHOST 127.0.0.1
set SRVPORT 8000
set KEYPATH ../koadic-key.pem
set CERTPATH ../koadic-cert.pem
run
```

Then I grabbed the JARM tools from SalesForce and calculated my server's fingerprint:

```
git clone https://github.com/salesforce/jarm.git
cd jarm
python3 jarm.py 127.0.0.1 -p 8000
```

My Result:

```
JARM: 2ad2ad0002ad2ad00042d42d000000ad9bf51cc3f5a1e29eecb81d0c7b06eb
```

Step 2. Search for JARM Fingerprint

Shodan is currently looking to implement JARM fingerprinting, but it doesn't do so currently.

Thankfully, Silascutler on Twitter recently did an internet scan and generated JARM fingerprints from all hosts listening on port 443.

This isn't going to see all servers, but it's a place to start, so I grabbed their archive and looked for our JARM signature:

```
# First got a sample of the data to find the format
zcat 202011-443_fingerprint.json.gz | head -n 10 > sample.json

# Format was very simple and 1 event per line, so can just use zgrep
# Keep data gzipped in case there are lots of hits
zgrep
'"fingerprint":"2ad2ad0002ad2ad00042d42d000000ad9bf51cc3f5a1e29eecb81d0c7b06eb"'
202011-443_fingerprint.json.gz | gzip > koadic_filtered.json.gz

# Get a count of the number of entries
zcat koadic_filtered.json.gz | wc -l
```

When I ran the last line, it listed 16,268 hits of our fingerprint 😐

So Why didn’t this work?

16,000 hits is way too high a number to be all Koadic servers, so I did some investigation into why I got so many false positives. I grapped the IP addresses from the first 10 entries in the list:

```
# Get the first 10 entries in plain text
zcat koadic_filtered.json.gz | head -n 10 > possible_koadic.json

# jq is a great tool to pretty-print JSON from the commandline
# but if you don't have it you could also just use cat/vin/notepad
# and strain your eyeballs for a bit
cat possible_koadic.json | jq -r '.ip'
```

And then just looked up these IP addresses in Shodan. Our data might be slightly stale, but hopefully it can give us some indication as to what went wrong.

The data from Shodan starts to paint a picture of what has happened: A number of the IP addresses returned headers like the following:

```
Server: Python/3.8 aiohttp/3.6.2
Server: Python/3.8 aiohttp/3.7.1
Server: Python/3.8 aiohttp/3.6.1
```

We know Koadic C2 is written in Python, so it is possibly just running the default Python web server, which would mean the fingerprint is not Koadic’s, but Python’s.

Sure enough, we can create a simple Python TLS Server using the inbuilt `http.server`:

```
from http.server import HTTPServer, BaseHTTPRequestHandler
import ssl

httpd = HTTPServer(('localhost', 8000), BaseHTTPRequestHandler)
httpd.socket = ssl.wrap_socket (httpd.socket,
        keyfile="koadic-key.pem",
        certfile="koadic-cert.pem", server_side=True)

print("Stating Server https://localhost:8000")
httpd.serve_forever()
```

By running that code, then running JARM, we get the exact same fingerprint:

```
JARM: 2ad2ad0002ad2ad00042d42d000000ad9bf51cc3f5a1e29eecb81d0c7b06eb
```

Conclusion

JARM definitely looks to be an interesting addition to the TLS Fingerprinting suite, alongside the client fingerprinting tool JA3. But it won’t pick up all malicious servers.

On top of this example of a known-bad server using language defaults to look benign, the more common issue will be that a lot of C2 servers sit behind legitimate web server reverse-proxies, such as Apache HTTPd or NGinx.

This means the JARM fingerprints will match only the legitimate outer layer, and not the C2 server’s fingerprint that sits behind the legitimate servers.