Easily Identify Malicious Servers on the Internet with JARM - Salesforce Engineering Blog

By John Althouse, Laura Lindeman

Published: 2020-11-17 · Archived: 2026-04-05 17:17:40 UTC

TL;DR

JARM is an active Transport Layer Security (TLS) server fingerprinting tool.

Scanning with JARM provides the ability to identify and group malicious servers on the Internet.

> JARM is available here: https://github.com/salesforce/jarm

JARM fingerprints can be used to:

- Quickly verify that all servers in a group have the same TLS configuration.
- Group disparate servers on the internet by configuration, identifying that a server may belong to Google vs. Salesforce vs. Apple, for example.
- Identify default applications or infrastructure.
- Identify malware command and control infrastructure and other malicious servers on the Internet.

In this blog post you will learn:

- How JARM works.
- How JARM can be used to identify malicious servers.
- Moving from reactive to proactive cybersecurity blocklists.
- How to deploy JARM into your detection and response pipeline.
- How JARM can be used for configuration validation and application identification.

How JARM Works

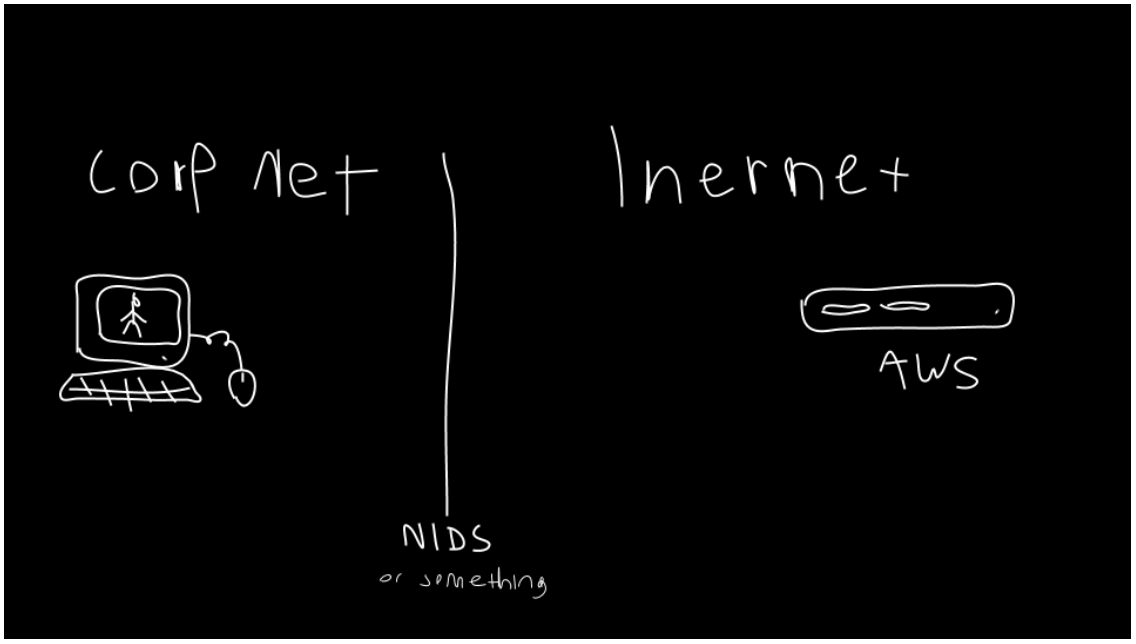
```
test@kobayashi-maru:~/jarm$ python3 jarm.py -V
JARM version 1.0
test@kobayashi-maru:~/jarm$ python3 jarm.py -h
usage: jarm.py [-h] [-i INPUT] [-p PORT] [-v] [-V] [-o OUTPUT] [scan]

Enter an IP address and port to scan.

positional arguments:
  scan                  Enter an IP or domain to scan.

optional arguments:
  -h, --help            show this help message and exit
  -i INPUT, --input INPUT
                        Provide a list of IP addresses or domains to scan, one domain or IP address per line.
                        Optional: Specify port to scan with comma separation (e.g. 8.8.4.4,853).
  -p PORT, --port PORT  Enter a port to scan (default 443).
  -v, --verbose         Verbose mode: displays the JARM results before being hashed.
  -V, --version         Print out version and exit.
  -o OUTPUT, --output OUTPUT
                        Provide a filename to output/append results to a CSV file.
```

Before learning how JARM works, it's important to understand how TLS works. TLS and its predecessor, SSL, are used to encrypt communication for both common applications like Internet browsers, to keep your data secure, and malware, so it can hide in the noise. To initiate a TLS session, a client will send a TLS Client Hello message following the TCP 3-way handshake. This packet and the way in which it is generated is dependent on packages and methods used when building the client application. The server, if accepting TLS connections, will respond with a TLS Server Hello packet.



This exquisite mouse-drawn network diagram shows how TLS works, for the most part, basically.

TLS servers formulate their Server Hello packet based on the details received in the TLS Client Hello packet. The manner in which the Server Hello is formulated for any given Client Hello can vary based on how the application or server was built, including:

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

All of these factors lead to each TLS Server responding in a unique way. The combinations of factors make it unlikely that servers deployed by different organizations will have the same response.

Below is an example of a TLS Client Hello and the Server Hello response as viewed in Wireshark.

Example TLS Client Hello packet (left) and Server Hello response (right)

JARM works by actively sending 10 TLS Client Hello packets to a target TLS server and capturing specific attributes of the TLS Server Hello responses. The aggregated TLS server responses are then hashed in a specific way to produce the JARM fingerprint.

This is not the first time we've worked with TLS fingerprinting. In 2017 we developed JA3/S, a passive TLS client/server fingerprinting method now found on most network security tools. But where JA3/S is **passive**, fingerprinting clients and servers by listening to network traffic, JARM is an **active** server fingerprinting scanner. You can find out more about TLS negotiation and JA3/S passive fingerprinting here.

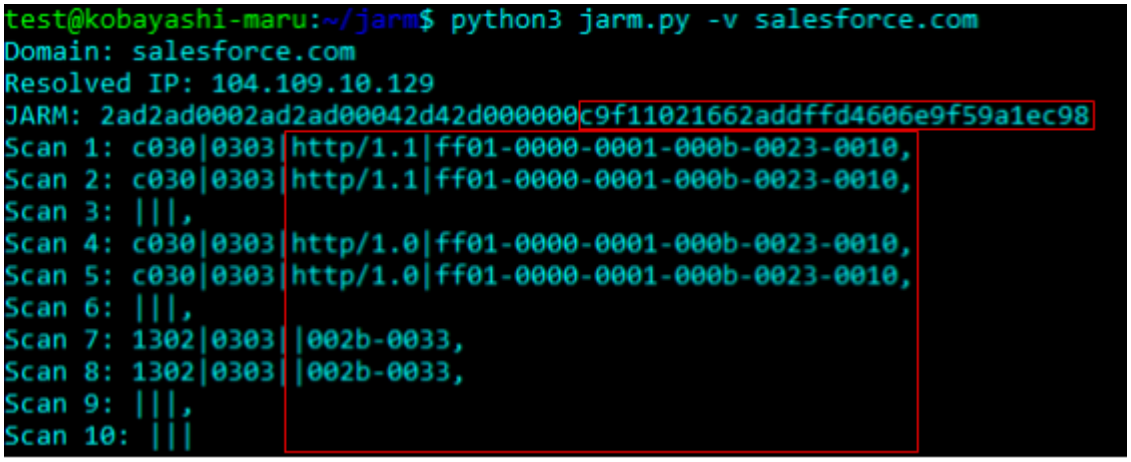
The 10 TLS Client Hello packets in JARM have been specially crafted to pull out unique responses in TLS servers. JARM sends different TLS versions, ciphers, and extensions in varying orders to gather unique responses. Does the server support TLS 1.3? Will it negotiate TLS 1.3 with 1.2 ciphers? If we order ciphers from weakest to strongest, which cipher will it pick? These are the types of unusual questions JARM is essentially asking the server to draw out the most unique responses. The 10 responses are then hashed to produce the JARM fingerprint.

```
test@kobayashi-maru:~/jarm$ python3 jarm.py salesforce.com
Domain: salesforce.com
Resolved IP: 23.1.99.130
JARM: 2ad2ad0002ad2ad00042d42d000000c9f11021662addffd4606e9f59a1ec98
```

The JARM fingerprint hash is a hybrid fuzzy hash, it uses the combination of a reversible and non-reversible hash algorithm to produce a 62 character fingerprint. The first 30 characters are made up of the cipher and TLS version chosen by the server for each of the 10 client hello's sent. A "000" denotes that the server refused to negotiate with that client hello. The remaining 32 characters are a truncated SHA256 hash of the cumulative extensions sent by the server, ignoring x509 certificate data. When comparing JARM fingerprints, if the first 30 characters are the same but the last 32 are different, this would mean that the servers have very similar configurations, accepting the same versions and ciphers, though not exactly the same given the extensions are different.

```
test@kobayashi-maru:~/jarm$ python3 jarm.py -v salesforce.com
Domain: salesforce.com
Resolved IP: 104.109.10.129
JARM: 2ad2ad0002ad2ad00042d42d000000c9f11021662addffd4606e9f59a1ec98
Scan 1: c030|0303|http/1.1|ff01-0000-0001-000b-0023-0010,
Scan 2: c030|0303|http/1.1|ff01-0000-0001-000b-0023-0010,
Scan 3: |||,
Scan 4: c030|0303|http/1.0|ff01-0000-0001-000b-0023-0010,
Scan 5: c030|0303|http/1.0|ff01-0000-0001-000b-0023-0010,
Scan 6: |||,
Scan 7: 1302|0303||002b-0033,
Scan 8: 1302|0303||002b-0033,
Scan 9: |||,
Scan 10: |||
```

The different parts that make up the reversible 1st part of the fingerprint



The extensions that are hashed to make the 2nd part of the fingerprint

After receiving each TLS server hello message, JARM closes the connection gracefully with a FIN as to not leave the sockets open.

JARM examples:

Domain	JARM Fingerprint
google.com	27d40d40d29d40d1dc42d43d00041d4689ee210389f4f6b4b5b1b93f92252d
youtube.com	27d40d40d29d40d1dc42d43d00041d4689ee210389f4f6b4b5b1b93f92252d
blogger.com	27d40d40d29d40d1dc42d43d00041d4689ee210389f4f6b4b5b1b93f92252d
facebook.com	27d27d27d29d27d1dc41d43d00041d741011a7be03d7498e0df05581db08a9
instagram.com	27d27d27d29d27d1dc41d43d00041d741011a7be03d7498e0df05581db08a9
oculus.com	29d29d20d29d29d21c41d43d00041d741011a7be03d7498e0df05581db08a9

Cipher picked and TLS version, SHA256 of TLS extensions

It is important to note that JARM is a high-performance fingerprint function and should not be considered, or confused with, a secure crypto function. We designed the JARM fingerprint to be human consumable as much as machine consumable. This means it is small enough to eyeball, share, and tweet with enough room for contextual details.

How JARM Can Be Used to Identify Malicious Servers

This reference to the 1980 arcade game Missile Command really has nothing to do with anything, honestly.

Malware command and control (C2) and malicious servers are configured by their creators like any other server and then deployed across their fleet. These therefore tend to produce unique JARM fingerprints. For example, when scanning Trickbot Malware C2s from a list compiled by abuse.ch, 80% of the live IPs on the list produced the same JARM fingerprint. When comparing this JARM fingerprint against the Alexa Top 1 Million websites, there was no overlap.

Continuing to test JARM against common malware and offensive tools found the following:

Malicious Server C2	JARM Fingerprint (as of Oct. 2020)	Overlap with Alexa Top 1M
Trickbot	22b22b09b22b22b22b22b22b22b352842cd5d6b0278445702035e06875c	0
AsyncRAT	1dd40d40d00040d1dc1dd40d1dd40d3df2d6a0c2caaa0dc59908f0d3602943	0
Metasploit	07d14d16d21d21d00042d43d000000aa99ce74e2c6d013c745aa52b5cc042d	0
Cobalt Strike	07d14d16d21d21d07c42d41d00041d24a458a375eef0c576d23a7bab9a9fb1	0
Merlin C2	29d21b20d29d29d21c41d21b21b41d494e0df9532e75299f15ba73156cee38	303

With little to no overlap of the Alexa Top 1M Websites, it should be extremely unlikely for a host within an organization to connect to a server with these JARM fingerprints. We wanted to dive further, so our friends at SecurityTrails scanned the entire Internet, billions of IPs, over port 443 with JARM and found the following:

Malicious Server C2	JARM Fingerprint (as of Oct. 2020)	Matches on Internet (IPv4 Port 443)
Trickbot	22b22b09b22b22b22b22b22b22b352842cd5d6b0278445702035e06875c	252
Metasploit	07d14d16d21d21d00042d43d000000aa99ce74e2c6d013c745aa52b5cc042d	1,538
Cobalt Strike	07d14d16d21d21d07c42d41d00041d24a458a375eef0c576d23a7bab9a9fb1	1,328

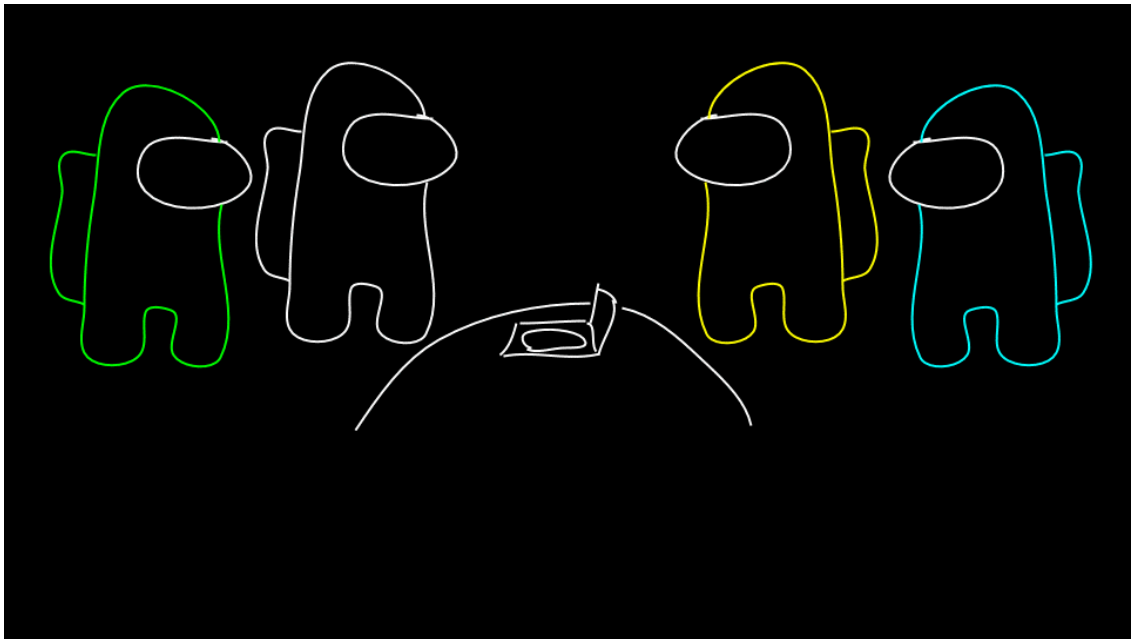
When taking a closer look at Cobalt Strike, a common offensive security tool used by red teams and threat actors alike, we found obvious indicators that most of the results were indeed Cobalt Strike, with server names including things like “redteam.server” “cobaltstrike” “totslegit,” as well as some of them having the default Cobalt Strike management port of 50050 open with the same JARM fingerprint. **We believe that this scan found most, if not all, Cobalt Strike C2’s listening on the Internet on port 443 at the time of scan.**

We did, however, find false positives in the list. It’s inevitable that in the sea of billions of IPs that some legitimate servers somewhere just happen to be configured in exactly the same way as Cobalt Strike. In the list we identified that the JARM also matches Burp Collaborator, another security tool used by red teams and threat actors alike, as well as miscellaneous legitimate servers, and a point of sale system. *(Point of sale systems listening on the Internet is a subject for another blog post…)* So while we believe JARM identified most, if not all, Cobalt Strike C2s listening on the Internet, we also had some legitimate servers caught in the net. This is like using a large magnet to pull all needles out of a haystack but getting some pieces of hay with them.

In order to use this as a potential blocklist, we need to filter out the false positives. One easy way to differentiate likely legitimate results with malicious ones is just by looking at the server’s history. Malicious C2s are generally ephemeral; they’re coming and going quite frequently, while legitimate servers tend to stay the same for long periods of time. This is where vendors with Internet historical data really come in handy. If the server matching the Cobalt Strike JARM has had its attributes unchanged for over a year, it’s more likely a legitimate false positive, while a server matching the Cobalt Strike JARM that didn’t exist 2 months ago is much more likely to be a malicious true positive. Combine that with other server attributes like name, hosting provider, certificate authority, etc. and we have ourselves a high quality Proactive Blocklist.

Cobalt Strike C2 List Identified from JARM Internet Scan Oct. 13 2020 (this is not the full list, just a small excerpt for example purposes)	
IP Address	Domain
34.238.192.43	offics365.com
190.210.180.57	secure-bankofameria-com-h829129hswu-server.she.org.np
52.19.14.90	cdn.paylesscreditcard.com
95.179.247.174	www.microsoftupdates.ml, testginwebsite.tk
69.10.48.173	as201.intellicyber.com
161.35.51.98	throwaway-site.xyz gjdfjgoa.com
202.182.122.79	entry-kali.kro.kr
173.82.153.84	adobefax.ml
64.227.121.168	collaboration.mtcrosoft.com
167.172.209.134	office-update.com
99.79.101.225	cs.ifred.team
45.76.53.114	www2.amazon.co.jp.pppbhbhfnb.monster
207.219.199.120	www.oktaa.info, www.insurancetoronto.org
45.63.122.195	hengqiangserver.tk
45.77.23.209	goooodserver.xyz
64.227.6.38	nahbruh.nextgenerationmeetings.com
35.176.207.20	Ns2.windows-updates-cdn.com, ns1.windows-updates-cdn.com
144.202.63.63	o365outlook.com
167.99.197.196	myteamserver.live, caffeinatedtutorials.com
149.6.167.60	www.googleapp.fr
78.129.165.207	update004.outlook360.org
185.150.117.63	www.matjari.site
46.166.161.159	bmwparklanne.co.uk
69.81.117.24	www.moocraft.org
106.53.232.176	ns2.360unsafe.com, ns1.360unsafe.com
204.16.247.65	www.oceanicbank.net, ns1.thearbawards.com
106.15.197.67	[no domain]
68.183.64.4	[no domain]
137.59.16.168	3mzi.com
172.241.29.156	amamai-tecnologies.digital
172.241.29.155	amamai-tecnologies.space
172.241.27.72	amatai-technologies.digital
172.241.27.66	amatai-technologies.space
3.16.1.87	ashinetech.com
104.248.88.66	cdndeliverytests.com, auth.service.asheori.com
35.241.143.134	control.commanderinthe.cloud
18.222.171.22	ec2-18-222-171-22.us-east-2.compute.amazonaws.com
106.13.212.242	honglanjun.net
192.236.193.184	hwsrv-635847.hostwindsdns.com, atakai-technologies.work
192.236.232.228	hwsrv-758602.hostwindsdns.com, atakai-technologies.website
18.130.155.157	hypedsensor.space

Proactive Blocklists



Emergency Meeting

Historically, the cybersecurity industry has been focused on reactive blocklists of atomic indicators of compromise (IOCs). That is, the industry waits until a malware campaign is observed in the wild, analyzes it, then takes the observed IOCs and publishes them for blocklists. The problem is that, by the time the IOCs are published, the malware has already been distributed and security engineers are automatically on the defensive, playing damage control.

JARM Internet scanning, coupled with other metadata and historical analysis, allows for the possibility of proactive IOC identification for new campaigns using existing malware. For example, a cybersecurity researcher or company could scan the Internet with JARM, correlate known JARM results with the domain and IP history and reputation along with certificate details to build a high fidelity blocklist. This allows the cybersecurity industry to move towards the possibility of programmatically building out high fidelity blocklists before the first piece of malware is even distributed, placing threat actors on the defensive for the first time in a long time.

Deployment into Detection and Response

One could utilize JARM for detection and response by automatically scanning all destination hosts observed on their network for event enrichment, utilizing a summary table so as to not scan the same hosts multiple times in a given timeframe. They could then run queries of known-bad against the JARM list or utilize the list for correlation in response scenarios.

To simplify the process, one could utilize a security vendor, like SecurityTrails or Shodan, and query their API for destination JARM enrichment. Security researchers and vendors are likely to be better suited to maintain historical analysis of TLS servers and can therefore provide greater levels of metadata to utilize in measuring a host’s risk score.

There is also a lot of potential for security researchers and vendors to utilize JARM with correlation and historical analysis to identify and track malicious servers. This data could then be used to build high fidelity proactive

blocklists for easy consumption. But please note that extra care should be put into these blocklists to ensure minimal false positives.

How JARM Can Be Used For Configuration Validation and Application Identification

A fleet of application servers that are all running the same TLS configuration should have the same JARM fingerprint. One could regularly scan the fleet with JARM to confirm that they are the same. If a server in the fleet produces a different JARM fingerprint than the rest, then it is not running the same configuration. One major financial institution is already planning to use this capability to identify servers that are not running their latest TLS standard.

For example, if we use JARM to scan all Tor nodes maintained by [username] we find the following:

```
test@kobayashi-maru:~/jarm$ wc -l tor-user1.list
100 tor-user1.list
test@kobayashi-maru:~/jarm$ ./jarm.sh tor-user1.list tor-user1.csv
test@kobayashi-maru:~/jarm$ cat tor-user1.csv | cut -d "," -f3 | sort | uniq -c
    100 2ad2ad16d2ad2ad00042d42d00000000a021006ff37c875489e9a9cb08b3f2
```

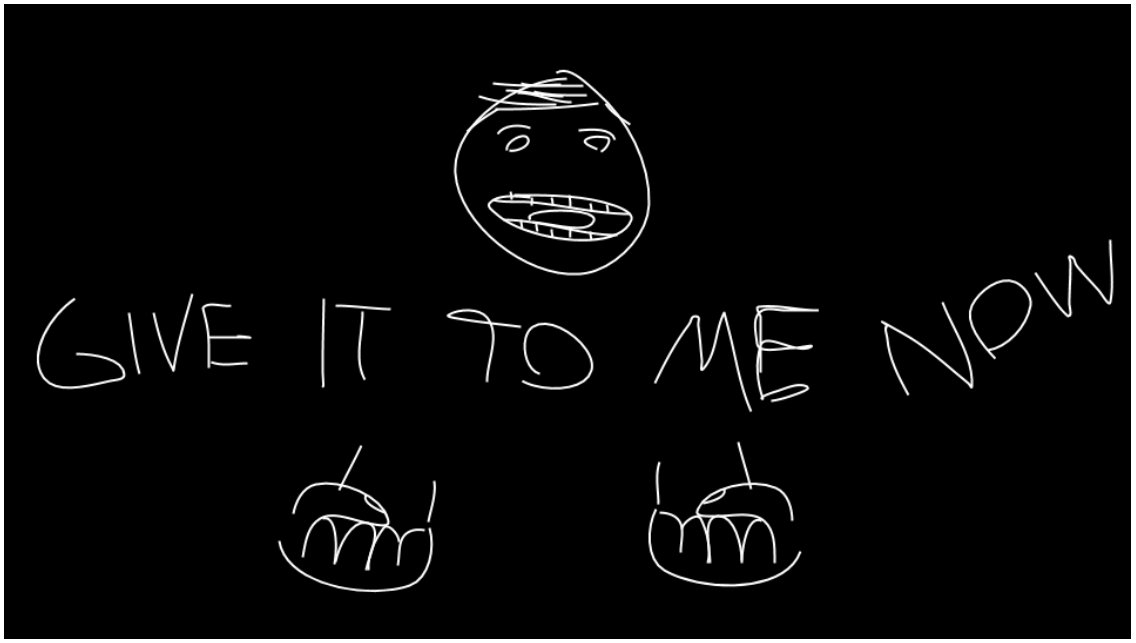
This shows that of the 100 Tor nodes the user maintains, 100 of them have the same JARM fingerprint. We essentially just ran a configuration drift check on this user's Tor node deployment and found that they indeed have a well-maintained fleet. However, if one host had a different JARM than the others, then it would mean it's not running the same configuration and may warrant investigation. To simulate this, I'll throw a random IP into the list and run it again:

```
test@kobayashi-maru:~/jarm$ wc -l tor-user1.list
101 tor-user1.list
test@kobayashi-maru:~/jarm$ ./jarm.sh tor-user1.list tor-user1.csv
test@kobayashi-maru:~/jarm$ cat tor-user1.csv | cut -d "," -f3 | sort | uniq -c
      1 27d3ed3ed0003ed1dc42d43d00041dac6f89ec17d1cb5afe80531c72353af5
    100 2ad2ad16d2ad2ad00042d42d00000000a021006ff37c875489e9a9cb08b3f2
test@kobayashi-maru:~/jarm$
```

JARM fingerprints appear to also be unique to default configurations and patch levels for certain servers and appliances. Because of this, it may be possible to associate a JARM fingerprint with a specific version of Apache, for example. There has yet to be exhaustive research put into this, but here are some preliminary findings:

JARM Fingerprint	Server / Application	Possible Version
27d3ed3ed0003ed1dc42d43d00041dac6f89ec17d1cb5afe80531c72353af5	Cloudflare	
2ad2ad0002ad2ad22c2ad2ad2ad2adccf03243f40d306256e19e3e47c11072	Nginx	1.13.10.1
29d29d00029d29d00029d29d29d29d3838d058a0d21ac352877290630da4ed	Fastly Varnish	
2ad2ad16d2ad2ad0002ad2ad2ad2adaa3acdc5b419ea9e93160032149d6831	Apache	2.2.15

There are certainly a lot of research possibilities with JARM that have yet to be explored so I encourage you to dive in and share what you find!



JARM has been open sourced and is available here: https://github.com/salesforce/jarm

JARM was created by:

John Althouse
Andrew Smart
RJ Nunnally
Mike Brady

With special thanks to Caleb Yu for re-writing JARM in Python for operational use.

If you have any questions or feedback on JARM, please reach out to me (John Althouse) via LinkedIn or on Twitter @4A4133.
