TLS server-side tagging

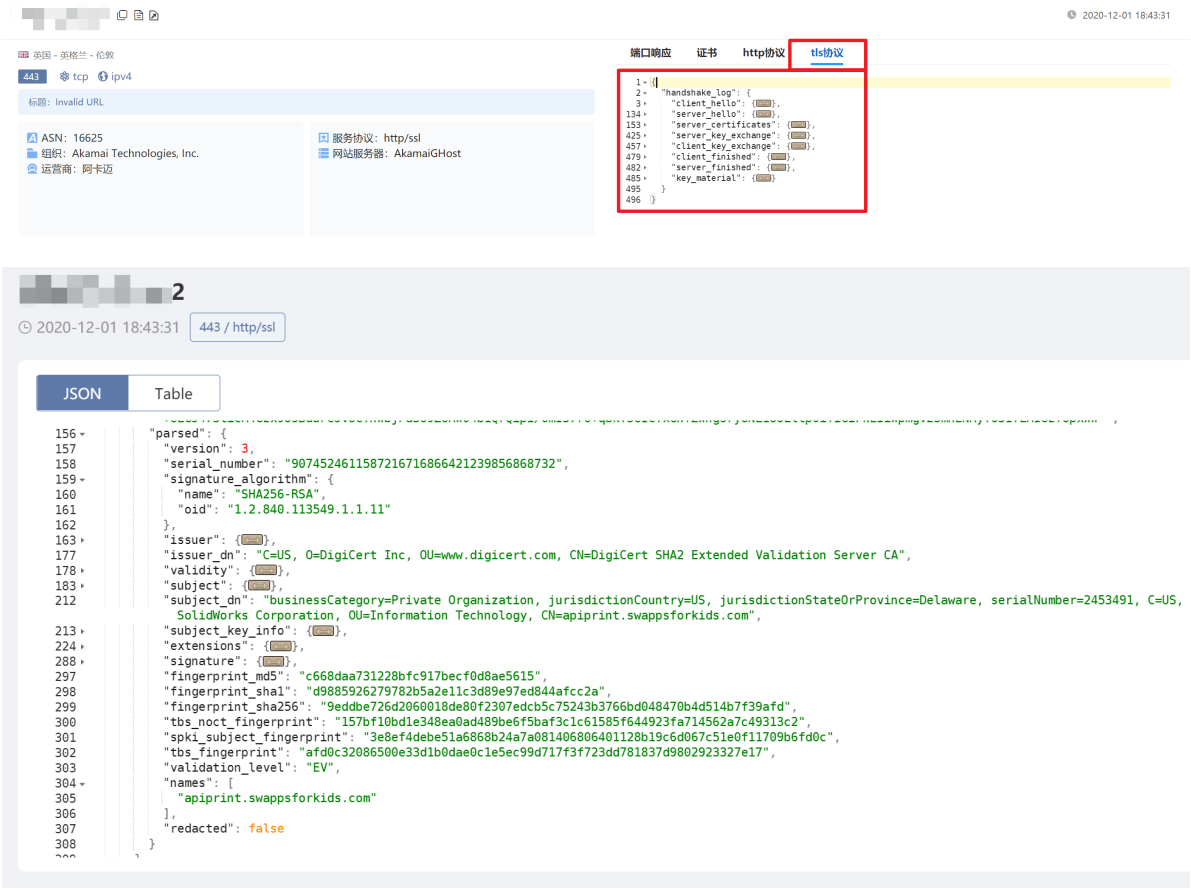
0x01 Background

Analysis and discovery of cyberspace mapping data has always been a core objective of the Quake team.

The rapid development of web applications over the past decade has made them undoubtedly the mainstream of the Internet. To compensate for the various security issues of Web applications and the HTTP protocol, the proportion of HTTP over SSL/TLS in the Internet is also increasing year by year. Therefore, the mapping and data analysis of SSL/TLS related to the whole network has been one of the key concerns of Quake System.

The current Quake system** already supports SSL/TLS certificate extraction, analysis, handshake packet parsing and retention** for any port and any protocol used.

Registered users can see the TLS certificate parsed in x509 format in the `certificates` window, while **paying members (premium, lifetime and corporate members)** can see the complete TLS handshake process in the `tls protocol` window and provide formatted parsed data in `server_certificates` contains the fingerprinting calculation for the server-side certificate. This is shown in the following diagram.



At the same time, we are also continuing to monitor cutting-edge research in the direction of TLS active mapping. Recently, we have noticed that the researchers concerned have published a paper entitled [Easily Identify Malicious Servers on the Internet with JARM] and a [JARM scanning tool] on github , the relevant content has generated discussion among some researchers abroad. Thanks to the concerted efforts of the Quake team, this feature has now been integrated into the Quake system.

After a period of analysis and research, we have also concluded some knowledge about JARM to exchange and share with you. We hope that you will give us a few pointers.

0x02 What is JARM

JARM is an **active** TLS server-side fingerprinting tool with the following main uses.

1. to quickly verify that a group of TLS servers are using the same TLS configuration.
2. segmenting TLS servers by their TLS configuration and identifying the companies to which they may belong.
3. identifying the default application or infrastructure of a website.
4. identifying malware C&C control nodes, and other malicious servers.

2.1 How it works

To understand how JARM works, it is necessary to understand the flow of TLS work, so we will not explain it in detail here. We will summarize the general purpose of the TLS handshake in one simple sentence: both the client and the server communicate, negotiate and verify based on each other's configuration, and generate the key after reaching an agreement. The core of JARM is that the **TLS Server returns different Server Hello packets depending on the parameters in the TLS Client Hello. The parameters of the Client Hello can be artificially specified and modified, so by sending multiple carefully constructed Client Hello's to obtain their corresponding special Server Hello's, the fingerprint of the TLS Server is eventually formed (somewhat similar to the feeling of Fuzz).** Specific parameters that can have an impact include, but are not limited to.

- Operating systems and their versions
- Third party libraries such as OpenSSL and their versions
- The order in which third party libraries are called
- User defined configuration
- ……

2.2 The JARM workflow

JARM generates the JARM fingerprint by actively sending 10 TLS Hello packets to the TLS server and analysing specific fields in the Server Hello to hash the 10 TLS server responses in a specific way.

The 10 TLS client Hello packets in JARM are specially designed to extract unique responses from the TLS server. Example.

- JARM sends different TLS versions, passwords and extensions in different orders.
- TLS Clint sorts the passwords from weakest to strongest, which one will the TLS Server choose?
- ……

In summary JARM differs from JA3, JA3/S, which we commonly use for traffic analysis of threats.

- JA3, JA3/S are primarily traffic based
- JARM, on the other hand, is completely proactive in scanning and generating fingerprints

So with the above theoretical basis in mind, let's try to analyse the specific code of the JARM tool.

2.3 Code Analysis

First in the main function, jarm defines 10 structures for TLS Client Hello packet generation, containing the target to be scanned, the port, the tls client encryption suite, the list of TLS extensions.

```
def main():
    #Select the packets and formats to send
    #Array format = [destination_host,destination_port,version,cipher_list,cipher_order,GREASE,RARE_APLN,1.3_SUPPORT,
    tls1_2_forward = [destination_host, destination_port, "TLS_1.2", "ALL", "FORWARD", "NO_GREASE", "APLN", "1.2_SUPP
    tls1_2_reverse = [destination_host, destination_port, "TLS_1.2", "ALL", "REVERSE", "NO_GREASE", "APLN", "1.2_SUPP
    tls1_2_top_half = [destination_host, destination_port, "TLS_1.2", "ALL", "TOP_HALF", "NO_GREASE", "APLN", "NO_SUP
    tls1_2_bottom_half = [destination_host, destination_port, "TLS_1.2", "ALL", "BOTTOM_HALF", "NO_GREASE", "RARE_APL
    tls1_2_middle_out = [destination_host, destination_port, "TLS_1.2", "ALL", "MIDDLE_OUT", "GREASE", "RARE_APLN", "
    tls1_1_middle_out = [destination_host, destination_port, "TLS_1.1", "ALL", "FORWARD", "NO_GREASE", "APLN", "NO_SU
    tls1_3_forward = [destination_host, destination_port, "TLS_1.3", "ALL", "FORWARD", "NO_GREASE", "APLN", "1.3_SUPP
    tls1_3_reverse = [destination_host, destination_port, "TLS_1.3", "ALL", "REVERSE", "NO_GREASE", "APLN", "1.3_SUPP
    tls1_3_invalid = [destination_host, destination_port, "TLS_1.3", "NO1.3", "FORWARD", "NO_GREASE", "APLN", "1.3_SU
    tls1_3_middle_out = [destination_host, destination_port, "TLS_1.3", "ALL", "MIDDLE_OUT", "GREASE", "APLN", "1.3_S
    #Possible versions: SSLv3, TLS_1, TLS_1.1, TLS_1.2, TLS_1.3
    #Possible cipher lists: ALL, NO1.3
    #GREASE: either NO_GREASE or GREASE
    #APLN: either APLN or RARE_APLN
    #Supported Verisons extension: 1.2_SUPPPORT, NO_SUPPORT, or 1.3_SUPPORT
    #Possible Extension order: FORWARD, REVERSE
    queue = [tls1_2_forward, tls1_2_reverse, tls1_2_top_half, tls1_2_bottom_half, tls1_2_middle_out, tls1_1_middle_ou
    jarm = ""
    #Assemble, send, and decipher each packet
```

The 10 TLS Client Hello structures are then traversed in turn to generate packets, and the corresponding TLS Client Hello packets are generated using the packet_building function, and the packets are then sent in turn.

```
    #Possible Extension order: FORWARD, REVERSE
    queue = [tls1_2_forward, tls1_2_reverse, tls1_2_top_half, tls1_2_bottom_half, tls1_2_middle_out, tls1_1_middle_ou
    jarm = ""
    #Assemble, send, and decipher each packet
    iterate = 0
    while iterate < len(queue):
        payload = packet_building(queue[iterate])
        server_hello, ip = send_packet(payload)
        #Deal with timeout error
        if server_hello == "TIMEOUT":
            jarm = "|||,|||,|||,|||,|||,|||,|||,|||,|||,|||"
            break
        ans = read_packet(server_hello, queue[iterate])
        # print(ans)
        jarm += ans
        iterate += 1
        if iterate == len(queue):
            break
        else:
            jarm += ","
    #Fuzzy hash
    result = jarm_hash(jarm)
    #Write to file
```

After sending the packet via send_packet, the TLS Server Hello is returned using read_packet parsing and spliced into the following format.

```
Lcy@localhost:~/tools/jarm(master⚡) » python3 jarm.py -p 443 quake.360.cn -v
Domain: quake.360.cn
Resolved IP: 180.163.237.84
JARM: 21d19d00021d21d21c21d19d21d21d3b0d229d76f2fd7cb8e23bb87da38a20
Scan 1: c013|0303|http/1.1|ff01-0000-0001-000b-0023-0010-0017,
Scan 2: 00c0|0303|http/1.1|ff01-0000-0001-0023-0010-0017,
Scan 3: |||,
Scan 4: c013|0303||ff01-0000-0001-000b-0023-0017,
Scan 5: c013|0303||ff01-0000-0001-000b-0023-0017,
Scan 6: c013|0302|http/1.1|ff01-0000-0001-000b-0023-0010-0017,
Scan 7: c013|0303|http/1.1|ff01-0000-0001-000b-0023-0010-0017,
Scan 8: 00c0|0303|http/1.1|ff01-0000-0001-0023-0010-0017,
Scan 9: c013|0303|http/1.1|ff01-0000-0001-000b-0023-0010-0017,
Scan 10: c013|0303|http/1.1|ff01-0000-0001-000b-0023-0010-0017
```

Field Meaning:

```
The encryption suite returned by the server | The server returns the version of
the TLS protocol selected for use | TLS extension ALPN protocol information |
List of TLS extensions
```

The final result is calculated by sending the TLS Client Hello 10 times and parsing it into the above format, and then calling jarm_hash after the 10 parses.

The first 30 characters of the jarm_hash are calculated from the cipher_bytes and version_byte functions of the encryption suite and TLS protocol respectively, while the remaining 32 characters are obtained by hashing the TLS extension ALPN protocol information and the TLS extension list with sha256 and intercepting.

```
397  #Custom fuzzy hash
398  def jarm_hash(jarm_raw):
399      #If jarm is empty, 62 zeros for the hash
400      if jarm_raw == "|||,|||,|||,|||,|||,|||,|||,|||,|||,|||":
401          return "0"*62
402      fuzzy_hash = ""
403      handshakes = jarm_raw.split(",")
404      alpns_and_ext = ""
405      for handshake in handshakes:
406          components = handshake.split("|")
407          #Custom jarm hash includes a fuzzy hash of the ciphers and versions
408          fuzzy_hash += cipher_bytes(components[0])
409          fuzzy_hash += version_byte(components[1])
410          alpns_and_ext += components[2]
411          alpns_and_ext += components[3]
412      #Custom jarm hash has the sha256 of alpns and extensions added to the end
413      sha256 = (hashlib.sha256(alpns_and_ext.encode())).hexdigest()
414      fuzzy_hash += sha256[0:32]
415      return fuzzy_hash
```

0x03 Applications and shortcomings of JARM

3.1 Search for a server using JARM

By studying JARM as described above we have understood the principles of JARM. As a result, JARM was integrated into the protocol depth recognition process of Vscan, Quake's underlying recognition engine.

In fact, in our previous article A Brief Analysis of CobaltStrike Beacon Scanning, those of you who are interested have noticed that in some of the ports that support The Quake search syntax is as follows, replacing the text at the end of the `Port Response` tag with a string like `JARM:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx`, which is the JARM fingerprint of the port.

```
response:"JARM:07d14d16d21d21d07c42d41d00041d24a458a375eef0c576d23a7bab9a9fb1"
```

Currently Quake **registered users** can see their JARM fingerprint at the end of the `port response` text. **This is automatically appended data and is not the original return data for the port, so please be aware of the distinction**.

At the same time, all **Lifetime Members, Corporate Members** are able to view the `TLS-JARM` protocol in depth identifying the content of.

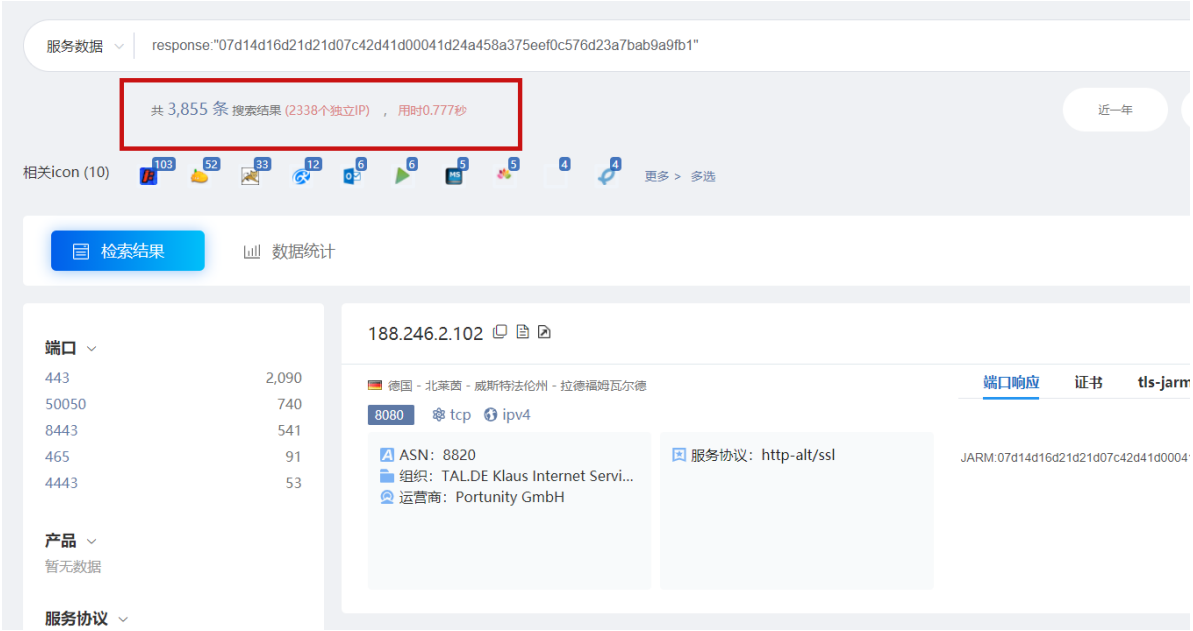
After a period of continuous mapping, we found some interesting phenomena, let's take a look below.

3.2 Identifying C&C and shortcomings with JARM

In [Easily Identify Malicious Servers on the Internet with JARM](), the author gives a list of C2 and JARM equivalents, which we won't go into here.

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

We were very happy when we got these C2s and JARMs, because ideally if the JARM uniquely corresponds to a C2, then we have one more feature to actively discover C2 nodes. But it was not to be, and searching for the JARM corresponding to that CS above: `response: "07d14d16d21d21d07c42d41d00041d24a458a375eef0c576d23a7bab9a9fb1"`



We found a lot of them, with 2,338 unique IPs. But the TOP 5 applications are：

应用	数量
Cobalt Strike团队服务器	1,137
CobaltStrike-Beacon服务端	373
Tomcat-Web服务器	40
Weblogic应用服务器	21
WordPressCMS博客系统	14

You can see that the same JARM as CobaltStrike above also includes TLS-enabled web applications such as Tomcat, Weblogic and WordPress, which means that CobaltStrike is only a subset of the JARM corresponding to TLS servers.

Continuing to build the environment locally for testing, Cobalt Strike 4.0 under JDK 11.0.9.1 JARM is `07d2ad16d21d21d07c42d41d00041d24a458a375eef0c576d23a7bab9a9fb1`.

```
┌[kali@kali:~/jarm]-[05:05:35 AM]-[G:master=]
└─>$ python3 jarm.py -v 127.0.0.1 -p 50050
Domain: 127.0.0.1
Resolved IP: 127.0.0.1
JARM: 07d14d16d21d21d07c42d41d00041d24a458a375eef0c576d23a7bab9a9fb1
Scan 1: 0033|0303||0001-0017-ff01,
Scan 2: 009d|0303||0001-0017-ff01,
Scan 3: 009f|0303||0001-0017-ff01,
Scan 4: c013|0303||0001-0017-ff01,
Scan 5: c013|0303||0001-0017-ff01,
Scan 6: 0033|0302||0001-0017-ff01,
Scan 7: 1302|0303||002b-002c-0033,
Scan 8: 1301|0303||002b-002c-0033,
Scan 9: |||,
Scan 10: 1301|0303||002b-002c-0033
┌[kali@kali:~/jarm]-[05:05:40 AM]-[G:master=]
└─>$ java -version
openjdk version "11.0.9.1" 2020-11-04
OpenJDK Runtime Environment (build 11.0.9.1+1-post-Debian-1)
OpenJDK 64-Bit Server VM (build 11.0.9.1+1-post-Debian-1, mixed mode, sharing)
┌[kali@kali:~/jarm]-[05:05:43 AM]-[G:master=]
└─>$
```

Using Quake search: `response:"CobaltStrike Beacon configurations" AND response:"07d2ad16d21d21d07c42d41d00041d24a458a375eef0c576d23a7bab9a9fb1"`, found no CobaltStrike Beacon for this JARM.

Back in the local environment switching JDK versions, the same Cobalt Strike 4.0, JARM in the case of JDK 1.8.0_212 is `07d2ad16d21d21d07c07d2ad07d21d9b2f5869a6985368a9dec764186a9175`。

It seems that JARM does not seem to have anything to do with CobaltStrike. to prove this, the Tomcat service was set up to configure TLS in the same JDK environment. the result is as follows.

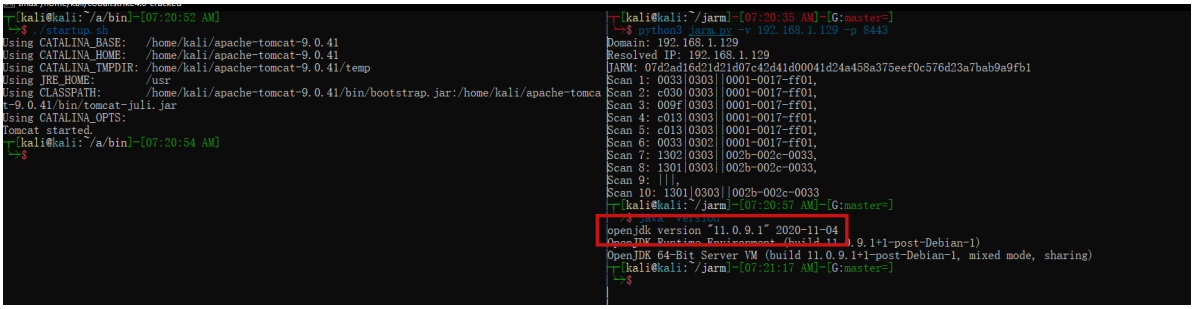
JDK 11.0.9.1 Tomcat 9.0.41 JARM
`07d2ad16d21d21d07c42d41d00041d24a458a375eef0c576d23a7bab9a9fb1`

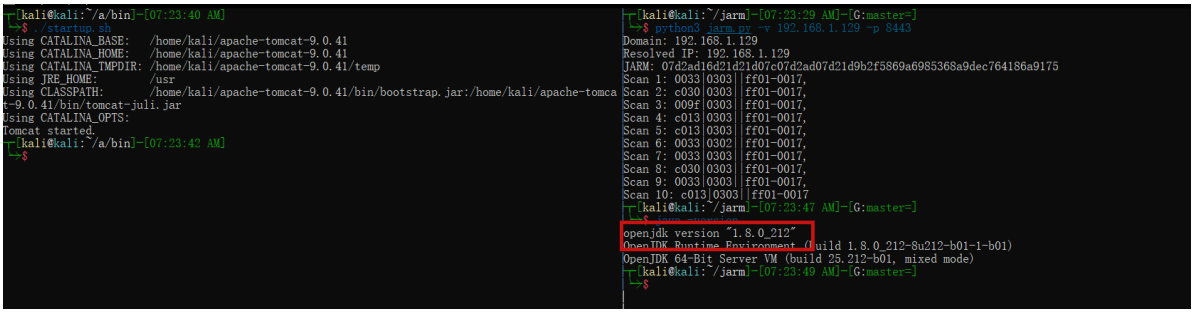
JDK 1.8.0_212 Tomcat 9.0.41 JARM
`07d2ad16d21d21d07c07d2ad07d21d9b2f5869a6985368a9dec764186a9175`

It was found that JARM is the same as CobaltStrike in both JDK environments respectively. It seems that this and CobaltStrike are not strongly correlated and explains why so many Weblogic and Tomcat applications are identified.

Further testing on multiple JDK versions gave the following results：

JDK版本	JARM	公网数量
JDK 1.8.0_211	07d3fd1ad21d21d07c42d43d0000008435c4f14f7a2c9375dab1adaee145f3	3645
JDK 1.9.0	05d14d16d04d04d05c05d14d05d04d4606ef7946105f20b303b9a05200e829	6
JDK 11.05	07d14d16d21d21d07c42d41d00041d24a458a375eef0c576d23a7bab9a9fb1	2338
JDK 13.01	2ad2ad16d2ad2ad22c42d42d00042d58c7162162b6a603d3d90a2b76865b53	216
JDK 1.8.0_212	07d2ad16d21d21d07c07d2ad07d21d9b2f5869a6985368a9dec764186a9175	30543
JDK 11.0.9.1	07d2ad16d21d21d07c42d41d00041d24a458a375eef0c576d23a7bab9a9fb1	3897





It seems that we cannot directly determine CobaltStrike by its JARM; similarly, JARM is not unique to CobaltStrike and is related to TLS services in different JDK environments.

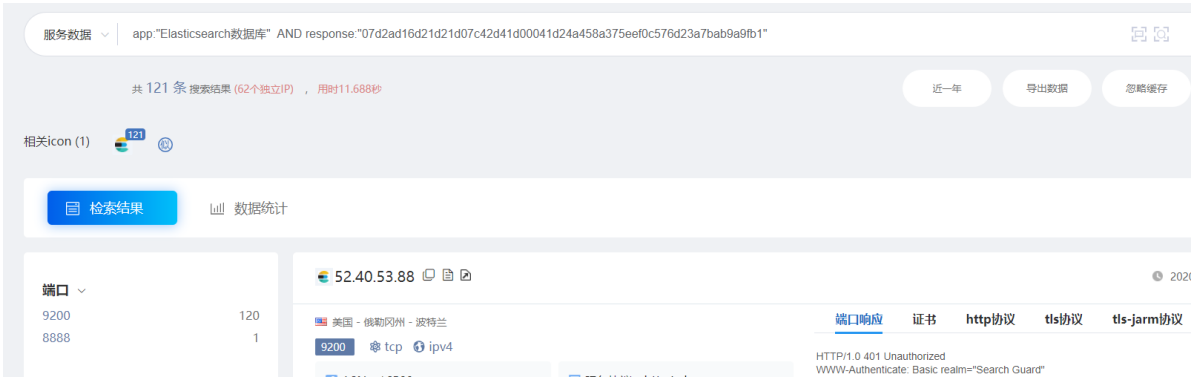
JARM can only be used as an aid. Combined with previous CobaltStrike features, we have extracted some of the CobaltStrike server JARM data and placed it in Quake's open source repository for industry research purposes only (not as accurate threat intelligence): [CobaltStrike-JARM]

	A	B	C
1	IP	PORT	JARM
2	121.36.211.148	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
3	175.24.68.66	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
4	139.180.198.152	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
5	139.180.198.152	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
6	81.68.85.109	9443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
7	81.68.85.109	9443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
8	175.24.68.66	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
9	212.95.150.10	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
10	81.68.85.109	9443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
11	121.37.139.238	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
12	121.36.211.148	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
13	211.149.143.218	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
14	211.149.143.218	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
15	101.37.148.15	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
16	18.141.196.104	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
17	101.37.148.15	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
18	175.24.68.66	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
19	212.95.150.10	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
20	175.24.68.66	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
21	212.95.150.10	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
22	175.24.68.66	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
23	211.149.143.218	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
24	139.180.198.152	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
25	81.68.85.109	9443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
26	139.180.198.152	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
27	101.37.148.15	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
28	121.36.211.148	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
29	95.130.9.249	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
30	154.201.215.15	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
31	52.59.192.156	443	2ad2ad16d2ad2ad22c42d42d00042d58c7162162b6a603d3d90a2b76865b53
32	47.75.123.100	443	2ad2ad16d2ad2ad22c42d42d00042d58c7162162b6a603d3d90a2b76865b53
33	116.62.49.176	443	2ad2ad16d2ad2ad22c42d42d00042d58c7162162b6a603d3d90a2b76865b53
34	52.59.192.156	443	2ad2ad16d2ad2ad22c42d42d00042d58c7162162b6a603d3d90a2b76865b53
35	47.75.123.100	443	2ad2ad16d2ad2ad22c42d42d00042d58c7162162b6a603d3d90a2b76865b53
36	103.152.132.173	443	2ad2ad16d2ad2ad22c42d42d00042d58c7162162b6a603d3d90a2b76865b53
37	52.59.192.156	443	2ad2ad16d2ad2ad22c42d42d00042d58c7162162b6a603d3d90a2b76865b53
38	52.59.192.156	443	2ad2ad16d2ad2ad22c42d42d00042d58c7162162b6a603d3d90a2b76865b53
39	52.59.192.156	443	2ad2ad16d2ad2ad22c42d42d00042d58c7162162b6a603d3d90a2b76865b53
40	52.59.192.156	443	2ad2ad16d2ad2ad22c42d42d00042d58c7162162b6a603d3d90a2b76865b53

0x04 Conclusion

JARM is not very reliable in identifying upper layer applications such as CobaltStrike, it is only an aid, and in practice it has to be combined with a wide range of information to make a judgement.

However, it is not true that JARM is completely useless, the essence of JARM is to mark TLS services, for example, we can combine the known JDK versions corresponding to JARM to see the services running in a specific version of the JDK environment on the public network, for example, the picture is running in JDK 11.0.9.1 ELasticSearch, running in JDK 11.05 Weblogic running at JDK 11.05.



JARM is only a way of identifying TLS server-side features, and cannot be fully used as a unique fingerprint for upper-level web applications.

In summary, JARM offers more of an idea than it is worth: using active mapping to send various types of packets to a target, and depending on the different returns then uncovering, analysing and extracting target characteristics.

As mentioned in [A Red Teamer Plays with JARM]:

> This is a commoditized threat intelligence practice. If your blue team uses this type of information, there are a lot of options to protect your infrastructure.

> Threat intelligence based on active mapping is taking root in every direction. Through statistical and analytical information on various dimensions of active mapping data, new protection ideas can be provided.

Happy hunting by using 360-Quake.

0x05 References

- https://engineering.salesforce.com/easily-identify-malicious-servers-on-the-internet-with-jarm-e095edac525a
- https://github.com/salesforce/jarm
- https://blog.cobaltstrike.com/2020/12/08/a-red-teamer-plays-with-jarm/