利用JARM指纹进行TLS服务端标记

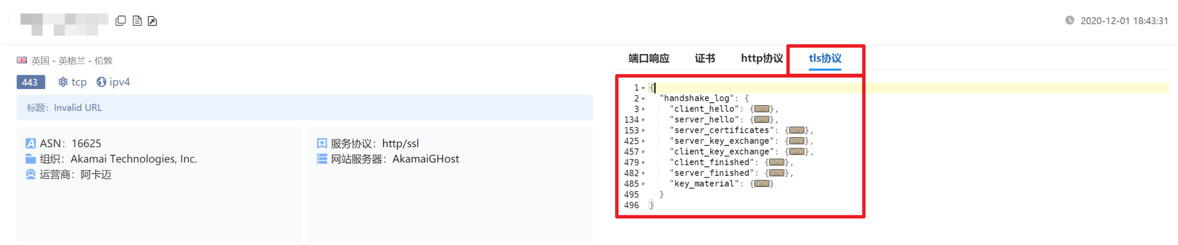
0x01 背景

对网络空间测绘数据的分析和发掘，是Quake团队一直以来的核心目标。

十几年来Web应用的飞速发展使其毋庸置疑的成为了互联网的主流。为了弥补Web应用和HTTP协议的各类安全问题，HTTP over SSL/TLS在互联网中的比例也逐年提升。因此，对全网SSL/TLS相关测绘与数据分析一直是Quake系统关注的重点之一。

当前Quake系统**已经支持任意端口、任意协议使用的SSL/TLS证书提取、分析、握手包的解析与留存。**

注册用户在`证书`窗口中就可以看到TLS证书按照x509格式进行解析后的内容，同时 **付费会员（高级会员、终身会员、企业会员）**可以在`tls协议`窗口中看到完整的TLS握手过程，并提供格式化解析后的数据，在 `server_certificates` 中就包含了对服务端证书的指纹采集计算。如下图所示：


[redacted]

2020-12-01 18:43:31

英国 - 英格兰 - 伦敦

443 tcp ipv4

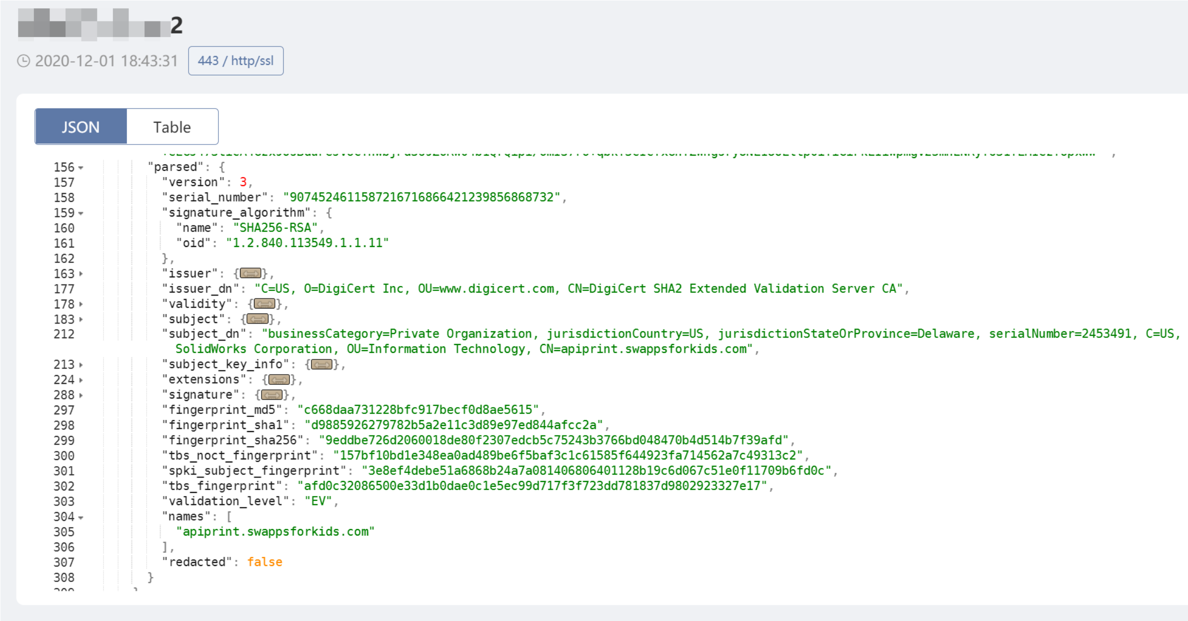
标题：Invalid URL

ASN：16625
组织：Akamai Technologies, Inc.
运营商：阿卡迈

服务协议：http/ssl
网站服务器：AkamaiGHost

端口响应 | 证书 | http协议 | tls协议

```
{
  "handshake_log": {
    "client_hello": {⇔},
    "server_hello": {⇔},
    "server_certificates": {⇔},
    "server_key_exchange": {⇔},
    "client_key_exchange": {⇔},
    "client_finished": {⇔},
    "server_finished": {⇔},
    "key_material": {⇔}
  }
}
```


[redacted]2

2020-12-01 18:43:31 | 443 / http/ssl

JSON | Table

```
        "parsed": {
          "version": 3,
          "serial_number": "9074524611587216716866421239856868732",
          "signature_algorithm": {
            "name": "SHA256-RSA",
            "oid": "1.2.840.113549.1.1.11"
          },
          "issuer": {⇔},
          "issuer_dn": "C=US, O=DigiCert Inc, OU=www.digicert.com, CN=DigiCert SHA2 Extended Validation Server CA",
          "validity": {⇔},
          "subject": {⇔},
          "subject_dn": "businessCategory=Private Organization, jurisdictionCountry=US, jurisdictionStateOrProvince=Delaware, serialNumber=2453491, C=US,
            SolidWorks Corporation, OU=Information Technology, CN=apiprint.swappsforkids.com",
          "subject_key_info": {⇔},
          "extensions": {⇔},
          "signature": {⇔},
          "fingerprint_md5": "c668daa731228bfc917becf0d8ae5615",
          "fingerprint_sha1": "d9885926279782b5a2e11c3d89e97ed844afcc2a",
          "fingerprint_sha256": "9eddbe726d2060018de80f2307edcb5c75243b3766bd048470b4d514b7f39afd",
          "tbs_noct_fingerprint": "157bf10bd1e348ea0ad489be6f5baf3c1c61585f644923fa714562a7c49313c2",
          "spki_subject_fingerprint": "3e8ef4debe51a6868b24a7a081406806401128b19c6d067c51e0f11709b6fd0c",
          "tbs_fingerprint": "afd0c32086500e33d1b0dae0c1e5ec99d717f3f723dd781837d9802923327e17",
          "validation_level": "EV",
          "names": [
            "apiprint.swappsforkids.com"
          ],
          "redacted": false
        }
```

与此同时，我们也在持续关注TLS主动测绘方向的前沿研究。近期，我们留意到有关研究人员在发布了一篇名为Easily Identify Malicious Servers on the Internet with JARM的文章，并在github上发布了一个JARM扫描工具，相关内容引起了国外部分研究人员的讨论。在Quake团队小伙伴一致努力下，现已将此功能集成入Quake系统。

经过一段时间的分析研究，我们也总结出一些关于JARM的认识与大家交流和分享。抛砖引玉，希望大家多多指正。

0x02 JARM介绍

JARM 是一个**主动式**TLS服务端指纹工具，主要用途如下：

1. 快速验证一组TLS服务器是否使用相同的TLS配置；
2. 通过TLS配置划分TLS服务器，并识别可能归属的公司；
3. 识别网站默认的应用或基础架构；
4. 识别恶意软件C&C控制节点，以及其他恶意服务器。

2.1 JARM工作原理

想要理解JARM工作原理，必须要了解TLS工作的流程，这里就不再详细讲解，我们用一句话简单概括下TLS握手的大致目的：客户端和服务端双方基于彼此的配置进行沟通、协商和校验，在达成一致后生成密钥。而JARM的核心在于：**TLS Server根据TLS Client Hello中参数的不同，返回不同的Server Hello数据包。而Client Hello的参数可以人为指定修改，因此通过发送多个精心构造的Client Hello获取其对应的特殊Server Hello，最终形成TLS Server的指纹（有点类似于Fuzz的感觉）。** 具体能够产生影响的参数包括但不限于：

- 操作系统及其版本
- OpenSSL等第三方库及其版本
- 第三方库的调用顺序
- 用户自定义配置
- ……

2.2 JARM工作流程

JARM通过主动向TLS服务器发送10个TLS Hello数据包并对Server Hello中的特定字段进行分析，以特定方式对10个TLS服务器响应进行哈希处理，最终生成JARM指纹。

JARM中的10个TLS客户端Hello数据包经过特殊设计，目的就是提取TLS服务器中的唯一响应。例如：

- JARM以不同的顺序发送不同的TLS版本，密码和扩展；
- TLS Clint将密码从最弱到最强排序，TLS Server将选择哪种密码？
- ……

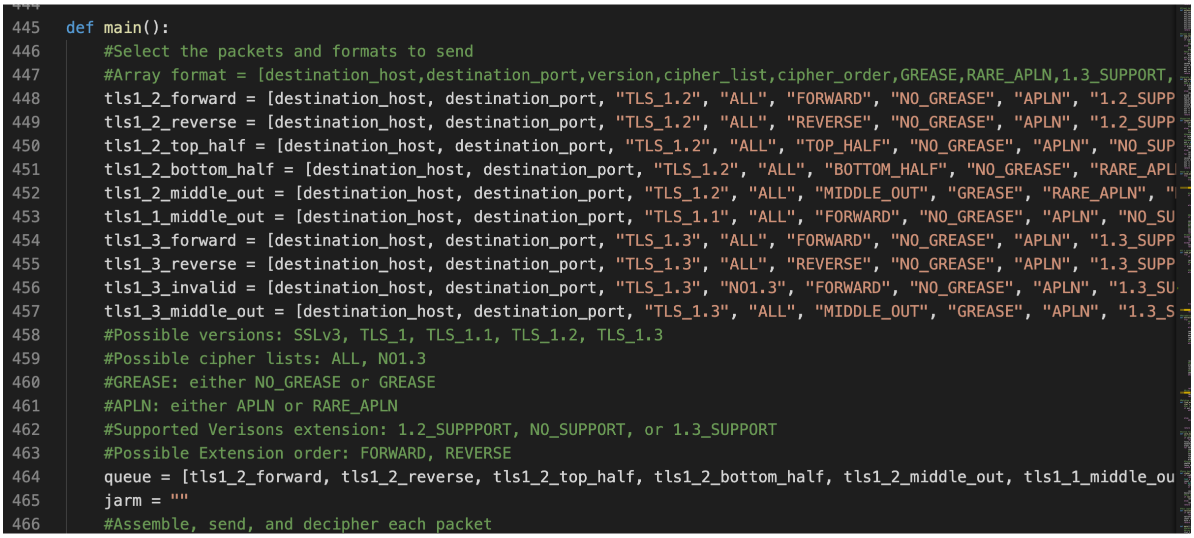
总之JARM与我们在进行流量分析威胁时常用的JA3、JA3/S不同：

- JA3、JA3/S主要基于流量
- JARM则是完全主动的扫描并生成指纹

因此有了上述的理论基础，我们尝试分析JARM工具的具体代码。

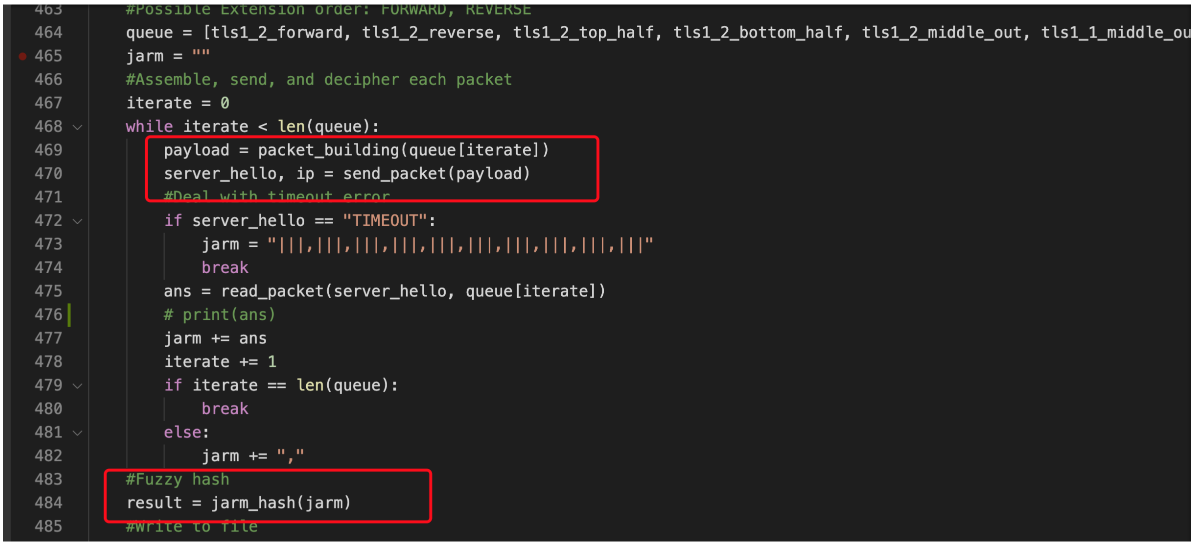
2.3 JARM工具代码分析

首先在main函数，jarm定义了10种TLS Client Hello数据包生成的结构，分别包含了待扫描的目标、端口、tls客户端加密套件、TLS扩展列表：



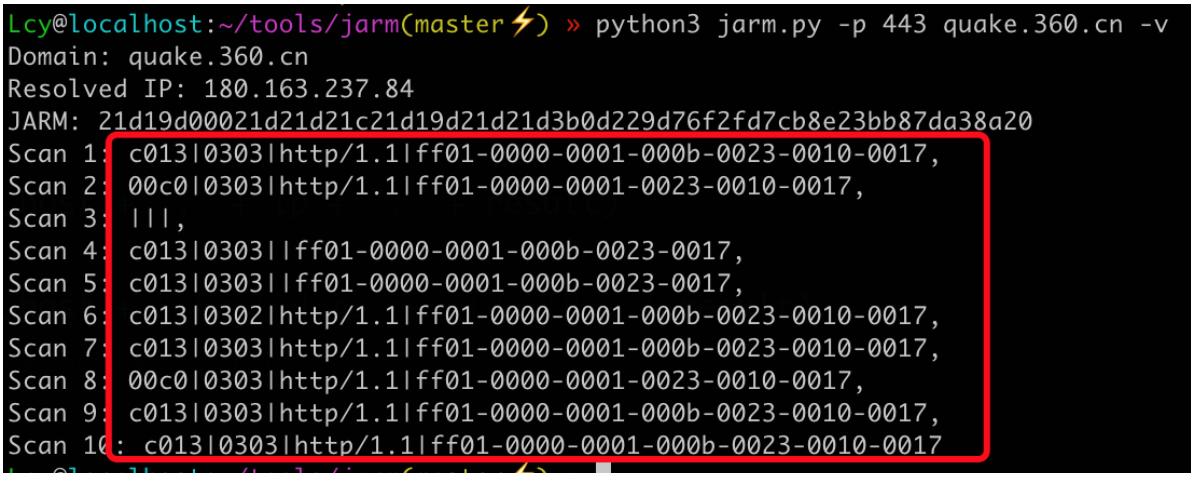
```
def main():
    #Select the packets and formats to send
    #Array format = [destination_host,destination_port,version,cipher_list,cipher_order,GREASE,RARE_APLN,1.3_SUPPORT,
    tls1_2_forward = [destination_host, destination_port, "TLS_1.2", "ALL", "FORWARD", "NO_GREASE", "APLN", "1.2_SUPP
    tls1_2_reverse = [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
```

然后依次遍历这10种TLS Client Hello结构生成数据包，并使用packet_building函数生成对应的TLS Client Hello数据包，然后依次发送数据包：



```
    #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
```

通过send_packet发送数据包以后，使用read_packet解析返回TLS Server Hello，并拼接为如下格式：



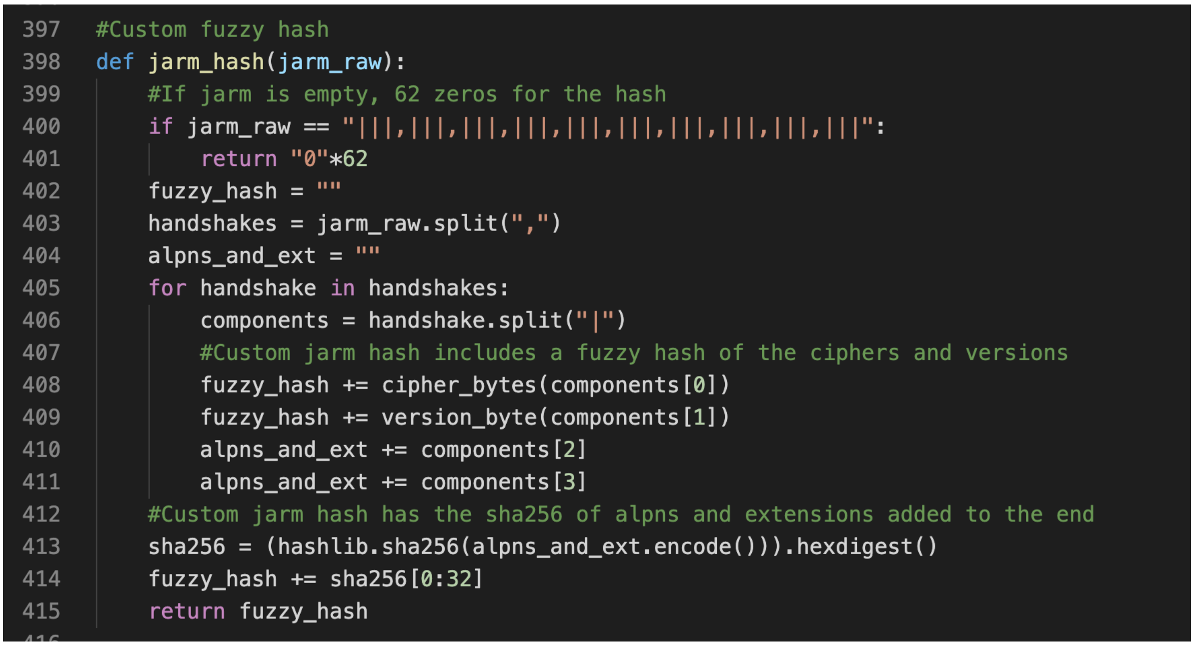
```
Lcy@localhost:~/tools/jarm(master⚡) » python3 jarm.py -p 443 quake.360.cn -v
Domain: quake.360.cn
Resolved IP: 180.163.237.84
JARM: 21d19d00021d21d21c21d19d21d21d3b0d229d76f2fd7cb8e23bb87da38a20
Scan 1: c013|0303|http/1.1|ff01-0000-0001-000b-0023-0010-0017,
Scan 2: 00c0|0303|http/1.1|ff01-0000-0001-0023-0010-0017,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

字段含义：

```
服务器返回的加密套件 | 服务器返回选择使用的TLS协议版本 | TLS扩展ALPN协议信息 | TLS扩展列表
```

通过发送10次TLS Client Hello并解析为以上格式，将10次解析的结果拼接以后最终调用jarm_hash算出最终的结果。

jarm_hash前30个字符由加密套件和TLS协议版本分别使用cipher_bytes、version_byte函数计算拼接而来，其余的32个字符是由TLS扩展ALPN协议信息和TLS扩展列表通过sha256哈希并截取而来：



```python
#Custom fuzzy hash
def jarm_hash(jarm_raw):
    #If jarm is empty, 62 zeros for the hash
    if jarm_raw == "|||,|||,|||,|||,|||,|||,|||,|||,|||,|||":
        return "0"*62
    fuzzy_hash = ""
    handshakes = jarm_raw.split(",")
    alpns_and_ext = ""
    for handshake in handshakes:
        components = handshake.split("|")
        #Custom jarm hash includes a fuzzy hash of the ciphers and versions
        fuzzy_hash += cipher_bytes(components[0])
        fuzzy_hash += version_byte(components[1])
        alpns_and_ext += components[2]
        alpns_and_ext += components[3]
    #Custom jarm hash has the sha256 of alpns and extensions added to the end
    sha256 = (hashlib.sha256(alpns_and_ext.encode())).hexdigest()
    fuzzy_hash += sha256[0:32]
    return fuzzy_hash
```

0x03 JARM的应用与问题

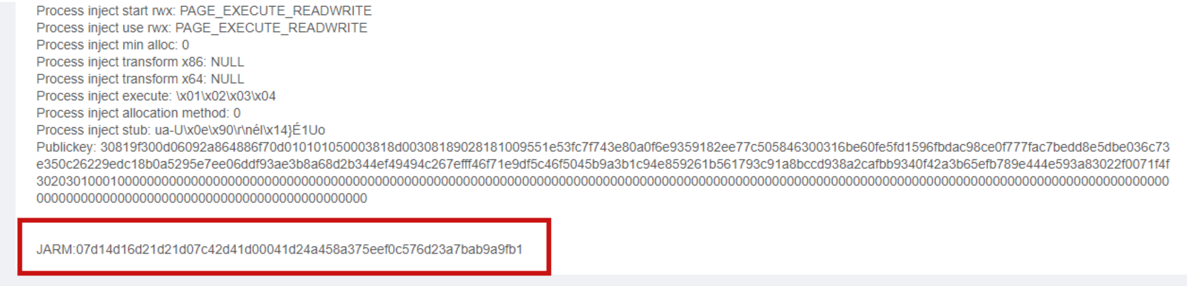
3.1 利用JARM搜寻服务端

通过上述对JARM的研究我们理解了JARM的原理。因此将JARM集成进入了Quake底层识别引擎Vscan的协议深度识别流程之中。

其实在我们之前的文章浅析 CobaltStrike Beacon 扫描中，有心的小伙伴已经留意到了在某些支持SSL/TLS的端口 `端口响应` 标签文本末尾有一串形如“JARM:xxxxxxxxxxxxxxx”的字符串，这便是该端口的JARM指纹。Quake搜索语法如下，注意替换 `JARM:` 之后的内容：

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

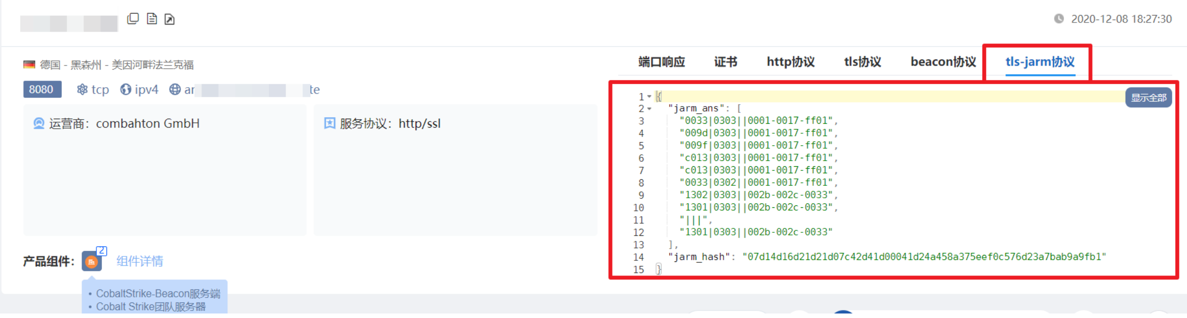
现目前Quake**注册用户**就能够在 `端口响应` 文本的末尾看到其JARM指纹。**该内容为系统自动追加后的数据，并不是该端口原始返回数据，请注意区分。**



```
Process inject start rwx: PAGE_EXECUTE_READWRITE
Process inject use rwx: PAGE_EXECUTE_READWRITE
Process inject min alloc: 0
Process inject transform x86: NULL
Process inject transform x64: NULL
Process inject execute: \x01\x02\x03\x04
Process inject allocation method: 0
Process inject stub: ua-U\x0e\x90\r\nél\x14}É1Uo
Publickey: 30819f300d06092a864886f70d010101050003818d0030818902818100 9551e53fc7f743e80a0f6e9359182ee77c505846300316be60fe5fd1596fbdac98ce0f777fac7bedd8e5dbe036c73
e350c26229edc18b0a5295e7ee06ddf93ae3b8a68d2b344ef49494c267efff46f71e9df5c46f5045b9a3b1c94e859261b561793c91a8bccd938a2cafbb9340f42a3b65efb789e444e593a83022f0071f4f
30203010001000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000

JARM:07d14d16d21d21d07c42d41d00041d24a458a375eef0c576d23a7bab9a9fb1
```

同时，所有 **终身会员、企业会员** 能够查看 `TLS-JARM` 协议深度识别的内容：



[redacted]

2020-12-08 18:27:30

德国 - 黑森州 - 美因河畔法兰克福

8080 tcp ipv4 ar[redacted]te

运营商：combahton GmbH

服务协议：http/ssl

产品组件： 组件详情

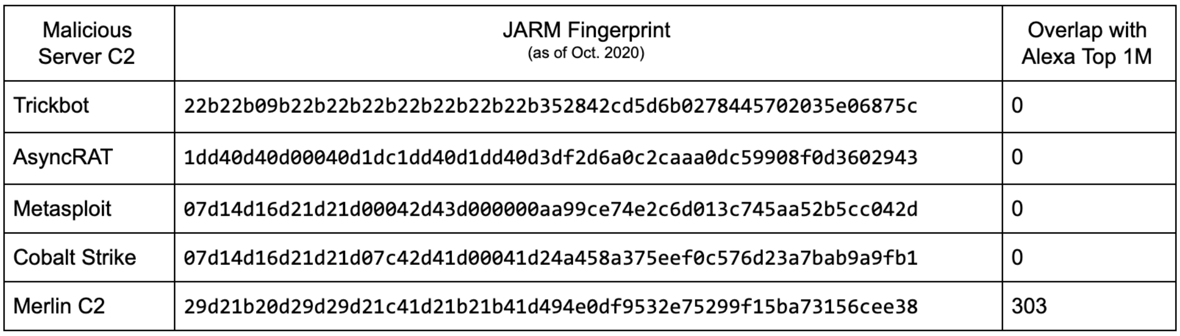
- CobaltStrike-Beacon服务端
- Cobalt Strike团队服务器

```
{
  "jarm_ans": [
    "0033|0303||0001-0017-ff01",
    "009d|0303||0001-0017-ff01",
    "009f|0303||0001-0017-ff01",
    "c013|0303||0001-0017-ff01",
    "c013|0303||0001-0017-ff01",
    "0033|0302||0001-0017-ff01",
    "1302|0303||002b-002c-0033",
    "1301|0303||002b-002c-0033",
    "|||",
    "1301|0303||002b-002c-0033"
  ],
  "jarm_hash": "07d14d16d21d21d07c42d41d00041d24a458a375eef0c576d23a7bab9a9fb1"
}
```

经过一段持续测绘后，我们发现了一些有趣的现象，下面让我们一起看看。

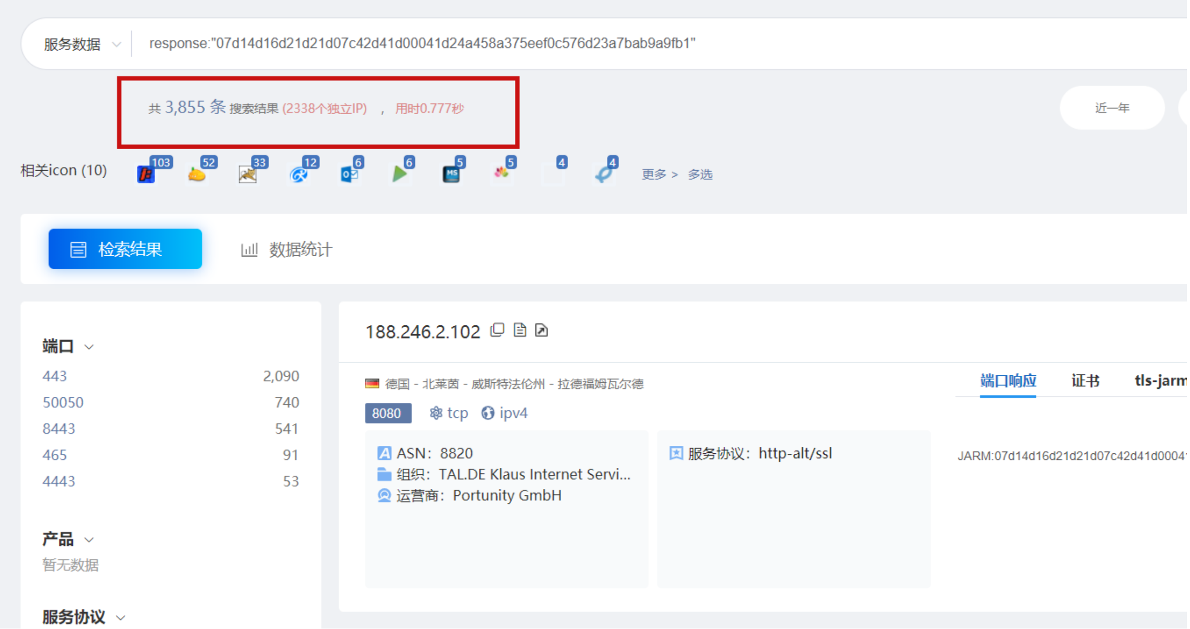
3.2 利用JARM识别C2与问题

在Easily Identify Malicious Servers on the Internet with JARM原文中，作者给出了一份C2和JARM对应的清单，这里我们就不赘述了。



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

当我们得到这些C2和JARM的时候是十分高兴的，因为在理想情况下如果JARM与C2唯一对应，那么我们就多了一种主动发现C2节点的特征。可是事与愿违，搜索上面的那个CS对应的JARM: `response:"07d14d16d21d21d07c42d41d00041d24a458a375eef0c576d23a7bab9a9fb1"`



服务数据 response:"07d14d16d21d21d07c42d41d00041d24a458a375eef0c576d23a7bab9a9fb1"

共 3,855 条 搜索结果 (2338个独立IP) ， 用时0.777秒

端口	
443	2,090
50050	740
8443	541
465	91
4443	53

产品

暂无数据

服务协议

188.246.2.102

德国 - 北莱茵 - 威斯特法伦州 - 拉德福姆瓦尔德

8080 tcp ipv4

ASN：8820
组织：TAL.DE Klaus Internet Servi...
运营商：Portunity GmbH

服务协议：http-alt/ssl

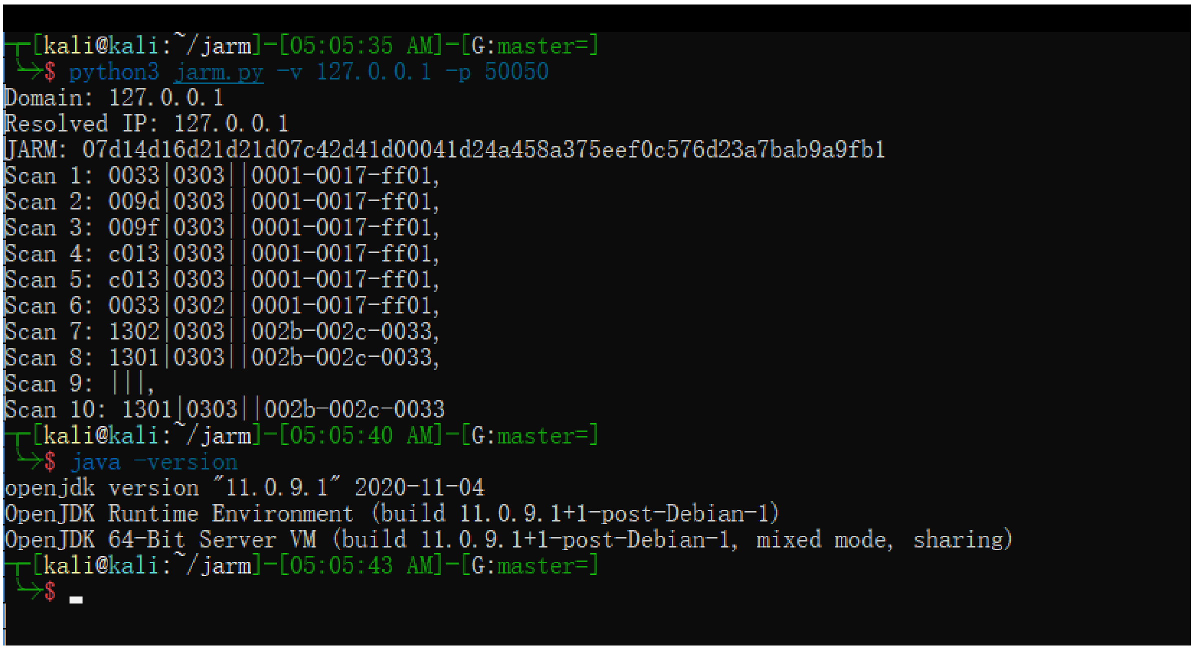
JARM:07d14d16d21d21d07c42d41d0004

我们发现数量不少，独立IP有2338个。但是 TOP5的应用为：

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

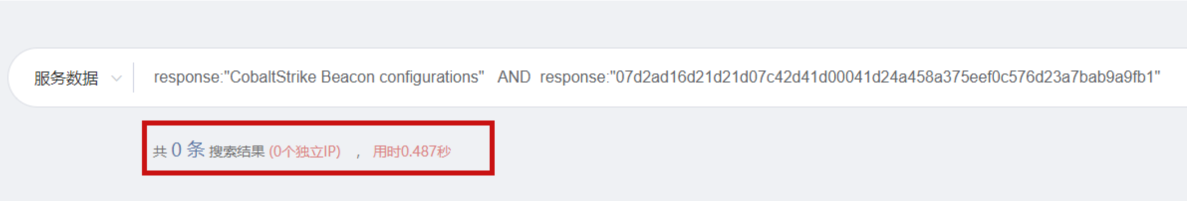
可以看到和上面CobaltStrike相同JARM的还有 Tomcat、Weblogic和WordPress等开启TLS的Web应用，也就是说CobaltStrike这个应用只是该JARM对应TLS服务器其中的一个子集。

继续在本地搭建环境进行测试，Cobalt Strike 4.0 在JDK 11.0.9.1下 JARM为 `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=]
└─>$
```

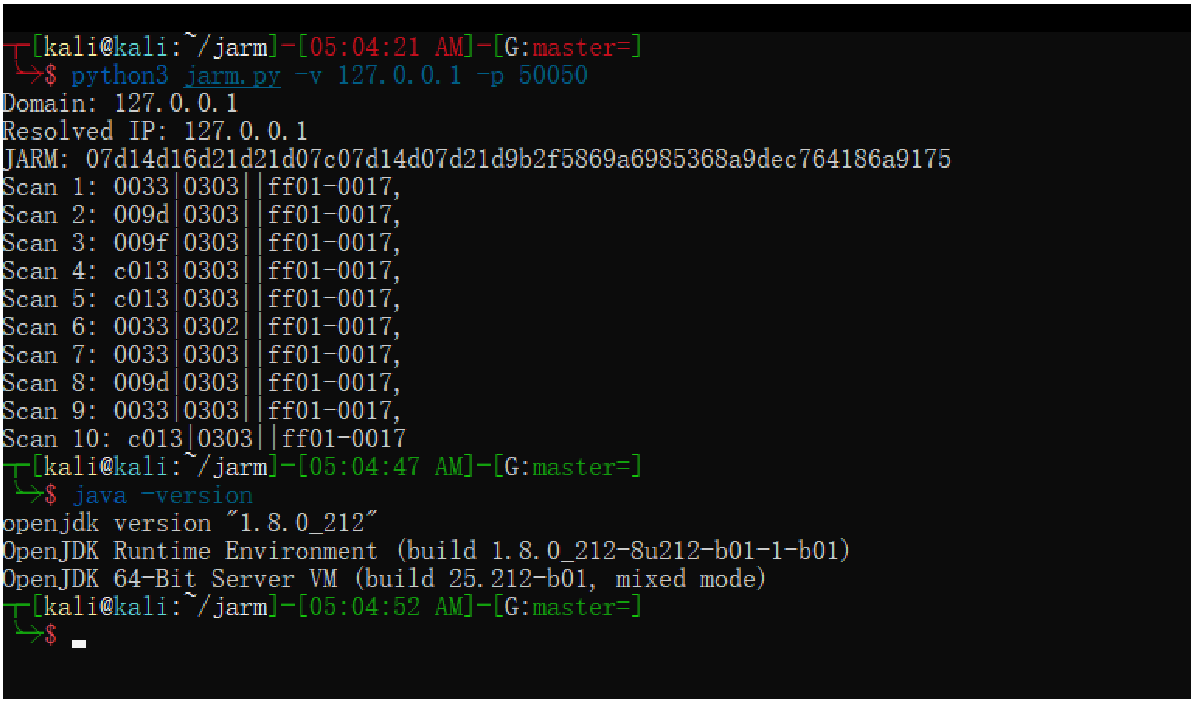
在Quake中搜索：`response:"CobaltStrike Beacon configurations" AND response:"07d2ad16d21d21d07c42d41d00041d24a458a375eef0c576d23a7bab9a9fb1"`，发现没有CobaltStrike Beacon为这个JARM。



服务数据 ⌄ response:"CobaltStrike Beacon configurations" AND response:"07d2ad16d21d21d07c42d41d00041d24a458a375eef0c576d23a7bab9a9fb1"

共 0 条 搜索结果 (0个独立IP) ，用时0.487秒

回到本地环境切换JDK版本，同一个Cobalt Strike 4.0，在JDK 1.8.0_212情况下JARM为：`07d2ad16d21d21d07c07d2ad07d21d9b2f5869a6985368a9dec764186a9175`。



```
┌─[kali@kali:~/jarm]-[05:04:21 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: 07d14d16d21d21d07c07d14d07d21d9b2f5869a6985368a9dec764186a9175
Scan 1: 0033|0303||ff01-0017,
Scan 2: 009d|0303||ff01-0017,
Scan 3: 009f|0303||ff01-0017,
Scan 4: c013|0303||ff01-0017,
Scan 5: c013|0303||ff01-0017,
Scan 6: 0033|0302||ff01-0017,
Scan 7: 0033|0303||ff01-0017,
Scan 8: 009d|0303||ff01-0017,
Scan 9: 0033|0303||ff01-0017,
Scan 10: c013|0303||ff01-0017
┌─[kali@kali:~/jarm]-[05:04:47 AM]-[G:master=]
└─>$ java -version
openjdk version "1.8.0_212"
OpenJDK Runtime Environment (build 1.8.0_212-8u212-b01-1-b01)
OpenJDK 64-Bit Server VM (build 25.212-b01, mixed mode)
┌─[kali@kali:~/jarm]-[05:04:52 AM]-[G:master=]
└─>$
```

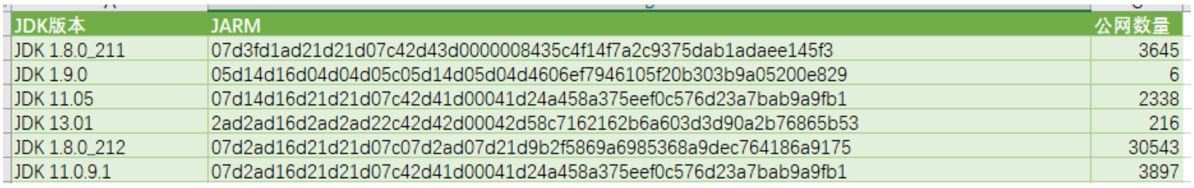
看来JARM似乎和CobaltStrike无关，为了证明这一点，在相同JDK环境下搭建Tomcat服务配置TLS。结果如下：

JDK 11.0.9.1 Tomcat 9.0.41 JARM `07d2ad16d21d21d07c42d41d00041d24a458a375eef0c576d23a7bab9a9fb1`

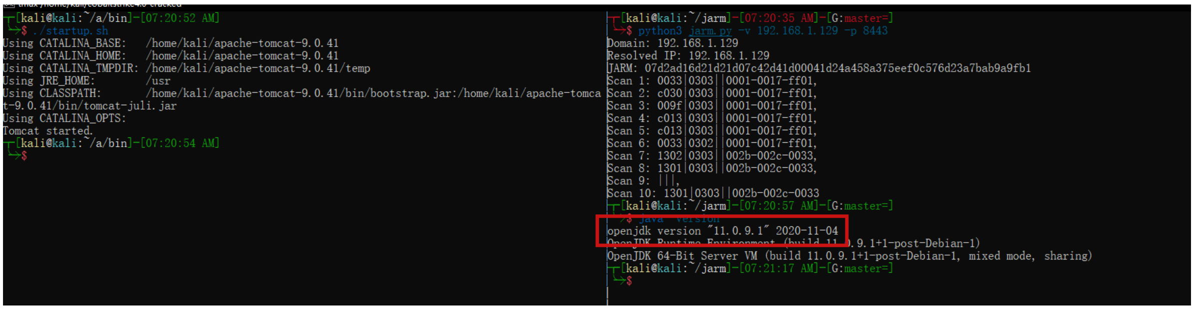
JDK 1.8.0_212 Tomcat 9.0.41 JARM `07d2ad16d21d21d07c07d2ad07d21d9b2f5869a6985368a9dec764186a9175`

发现JARM分别和CobaltStrike在两个JDK环境下的一样，看来这个和CobaltStrike不是强关联性的，也解释了为什么会有那么多的Weblogic和Tomcat应用被识别出来了。

进一步对多个JDK版本进行测试得到如下结果：

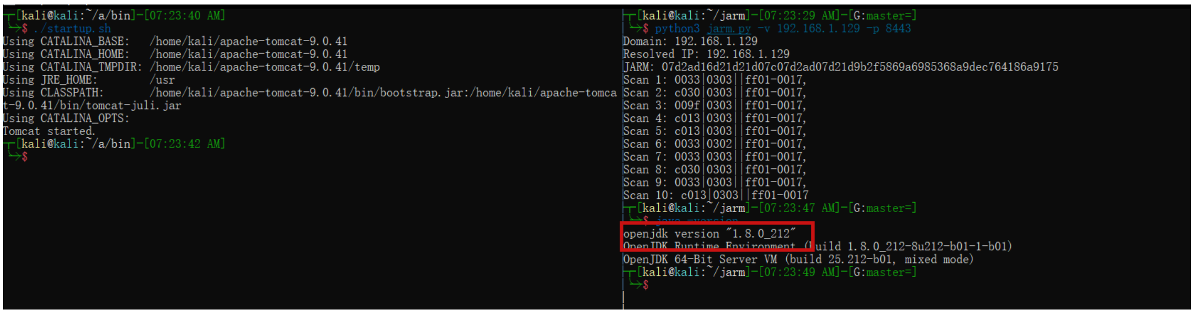


JDK版本	JARM	公网数量
JDK 1.8.0_211	07d3fd1ad21d21d07c42d43d0000008435c4f14f7a2c9375dab1adaee145f3	3645
JDK 1.9.0	05d14d16d04d04d05c05d14d05d04d4606ef7946105f20b303b9a05200e829	6
JDK 11.05	07d14d16d21d21d07c42d41d00041d24a458a375eef0c576d23a7bab9a9fb1	2338
JDK 13.01	2ad2ad16d2ad2ad22c42d42d00042d58c7162162b6a603d3d90a2b76865b53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



```
┌─[kali@kali:~/a/bin]─[07:20:52 AM]
└─>$ ./startup.sh
Using CATALINA_BASE:   /home/kali/apache-tomcat-9.0.41
Using CATALINA_HOME:   /home/kali/apache-tomcat-9.0.41
Using CATALINA_TMPDIR: /home/kali/apache-tomcat-9.0.41/temp
Using JRE_HOME:        /usr
Using CLASSPATH:       /home/kali/apache-tomcat-9.0.41/bin/bootstrap.jar:/home/kali/apache-tomca
t-9.0.41/bin/tomcat-juli.jar
Using CATALINA_OPTS:
Tomcat started.
┌─[kali@kali:~/a/bin]─[07:20:54 AM]
└─>$
```

```
┌─[kali@kali:~/jarm]─[07:20:35 AM]─[G:master=]
└─>$ python3 jarm.py -v 192.168.1.129 -p 8443
Domain: 192.168.1.129
Resolved IP: 192.168.1.129
JARM: 07d2ad16d21d21d07c42d41d00041d24a458a375eef0c576d23a7bab9a9fb1
Scan 1: 0033|0303||0001-0017-ff01,
Scan 2: c030|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]─[07:20:57 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]─[07:21:17 AM]─[G:master=]
└─>$
```

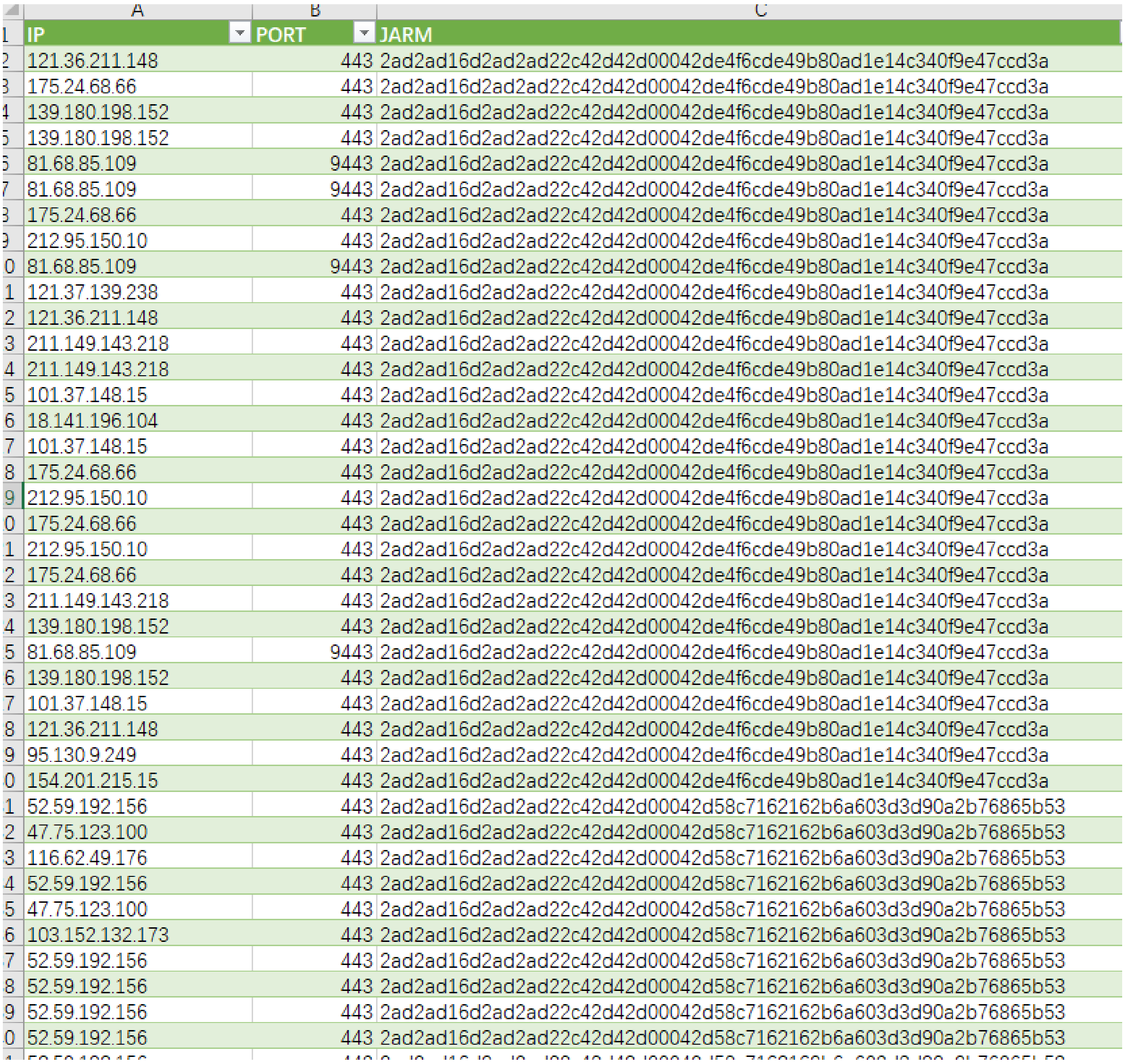


```
┌─[kali@kali:~/a/bin]─[07:23:40 AM]
└─>$ ./startup.sh
Using CATALINA_BASE:   /home/kali/apache-tomcat-9.0.41
Using CATALINA_HOME:   /home/kali/apache-tomcat-9.0.41
Using CATALINA_TMPDIR: /home/kali/apache-tomcat-9.0.41/temp
Using JRE_HOME:        /usr
Using CLASSPATH:       /home/kali/apache-tomcat-9.0.41/bin/bootstrap.jar:/home/kali/apache-tomca
t-9.0.41/bin/tomcat-juli.jar
Using CATALINA_OPTS:
Tomcat started.
┌─[kali@kali:~/a/bin]─[07:23:42 AM]
└─>$
```

```
┌─[kali@kali:~/jarm]─[07:23:29 AM]─[G:master=]
└─>$ python3 jarm.py -v 192.168.1.129 -p 8443
Domain: 192.168.1.129
Resolved IP: 192.168.1.129
JARM: 07d2ad16d21d21d07c07d2ad07d21d9b2f5869a6985368a9dec764186a9175
Scan 1: 0033|0303||ff01-0017,
Scan 2: c030|0303||ff01-0017,
Scan 3: 009f|0303||ff01-0017,
Scan 4: c013|0303||ff01-0017,
Scan 5: c013|0303||ff01-0017,
Scan 6: 0033|0302||ff01-0017,
Scan 7: 0033|0303||ff01-0017,
Scan 8: c030|0303||ff01-0017,
Scan 9: 0033|0303||ff01-0017,
Scan 10: c013|0303||ff01-0017
┌─[kali@kali:~/jarm]─[07:23:47 AM]─[G:master=]
└─>$ java -version
openjdk version "1.8.0_212"
OpenJDK Runtime Environment (build 1.8.0_212-8u212-b01-1-b01)
OpenJDK 64-Bit Server VM (build 25.212-b01, mixed mode)
┌─[kali@kali:~/jarm]─[07:23:49 AM]─[G:master=]
└─>$
```

看来，我们并不能直接通过JARM去判定CobaltStrike；同样，对于CobaltStrike而言JARM也并不唯一，其JARM与不同JDK环境下TLS服务有关。

JARM只能作为一个辅助手段，结合之前CobaltStrike的特征，我们提取了部分CobaltStrike服务器的JARM数据放置在Quake的开源仓库中，仅供业界研究使用（不作为精准威胁情报）：CobaltStrike-JARM

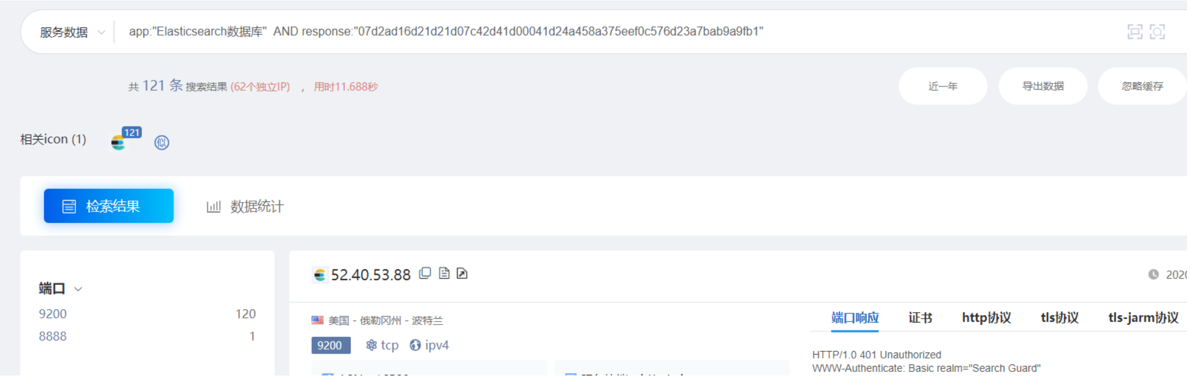


	A	B	C
1	IP	PORT	JARM
2	121.36.211.148	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
3	175.24.68.66	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
4	139.180.198.152	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
5	139.180.198.152	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
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
0	81.68.85.109	9443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
1	121.37.139.238	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
2	121.36.211.148	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
3	211.149.143.218	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
4	211.149.143.218	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
5	101.37.148.15	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
6	18.141.196.104	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
7	101.37.148.15	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
8	175.24.68.66	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
9	212.95.150.10	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
0	175.24.68.66	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
1	212.95.150.10	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
2	175.24.68.66	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
3	211.149.143.218	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
4	139.180.198.152	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
5	81.68.85.109	9443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
6	139.180.198.152	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
7	101.37.148.15	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
8	121.36.211.148	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
9	95.130.9.249	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
0	154.201.215.15	443	2ad2ad16d2ad2ad22c42d42d00042de4f6cde49b80ad1e14c340f9e47ccd3a
1	52.59.192.156	443	2ad2ad16d2ad2ad22c42d42d00042d58c7162162b6a603d3d90a2b76865b53
2	47.75.123.100	443	2ad2ad16d2ad2ad22c42d42d00042d58c7162162b6a603d3d90a2b76865b53
3	116.62.49.176	443	2ad2ad16d2ad2ad22c42d42d00042d58c7162162b6a603d3d90a2b76865b53
4	52.59.192.156	443	2ad2ad16d2ad2ad22c42d42d00042d58c7162162b6a603d3d90a2b76865b53
5	47.75.123.100	443	2ad2ad16d2ad2ad22c42d42d00042d58c7162162b6a603d3d90a2b76865b53
6	103.152.132.173	443	2ad2ad16d2ad2ad22c42d42d00042d58c7162162b6a603d3d90a2b76865b53
7	52.59.192.156	443	2ad2ad16d2ad2ad22c42d42d00042d58c7162162b6a603d3d90a2b76865b53
8	52.59.192.156	443	2ad2ad16d2ad2ad22c42d42d00042d58c7162162b6a603d3d90a2b76865b53
9	52.59.192.156	443	2ad2ad16d2ad2ad22c42d42d00042d58c7162162b6a603d3d90a2b76865b53
0	52.59.192.156	443	2ad2ad16d2ad2ad22c42d42d00042d58c7162162b6a603d3d90a2b76865b53
1	52.59.192.156	443	2ad2ad16d2ad2ad22c42d42d00042d58c7162162b6a603d3d90a2b76865b53

0x04 结论与思考

JARM其实对识别CobaltStrike等上层应用软件并不十分可靠，仅仅能够起到一个辅助的作用，实际工作中还是要结合多方面的信息来进行判断。

但是，也不是说JARM完全没有作用，JARM的本质是对TLS服务进行标记，例如：我们可以结合已知的JDK版本对应的JARM可以看到公网上运行在特定版本JDK环境下的服务，如图为运行在JDK 11.0.9.1的ELasticSearch，运行在JDK 11.05的Weblogic。



服务数据 app:"Elasticsearch数据库" AND response:"07d2ad16d21d21d07c42d41d00041d24a458a375eef0c576d23a7bab9a9fb1"

共 121 条 搜索结果 (62个独立IP) ， 用时11.688秒

相关icon (1) 121

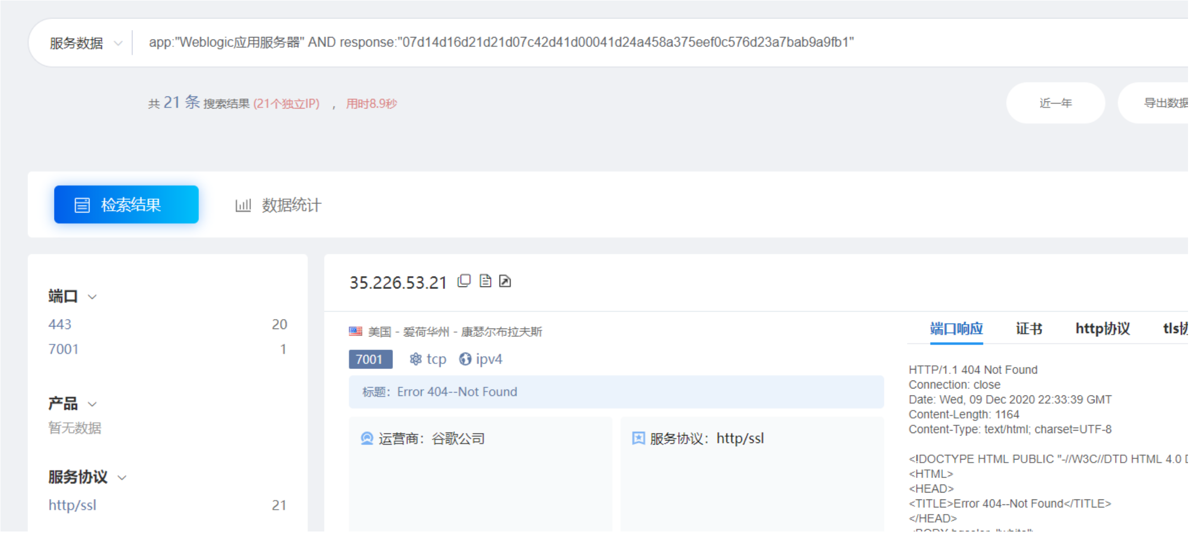
端口	
9200	120
8888	1

52.40.53.88

美国 - 俄勒冈州 - 波特兰

9200 tcp ipv4

```
HTTP/1.0 401 Unauthorized
WWW-Authenticate: Basic realm="Search Guard"
```



app:"Weblogic应用服务器" AND response:"07d14d16d21d21d07c42d41d00041d24a458a375eef0c576d23a7bab9a9fb1"

共 21 条 搜索结果 (21个独立IP) ， 用时8.9秒

端口	
443	20
7001	1

产品：暂无数据

服务协议	
http/ssl	21

35.226.53.21

美国 - 爱荷华州 - 康瑟尔布拉夫斯

7001 tcp ipv4

标题：Error 404--Not Found

运营商：谷歌公司

服务协议：http/ssl

```
HTTP/1.1 404 Not Found
Connection: close
Date: Wed, 09 Dec 2020 22:33:39 GMT
Content-Length: 1164
Content-Type: text/html; charset=UTF-8

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 D
<HTML>
<HEAD>
<TITLE>Error 404--Not Found</TITLE>
</HEAD>
```

JARM仅仅是一种TLS服务端特征的标识方式，`不能完全被用作web上层应用的唯一指纹`。

总归来说，JARM提供的思路大于其本身价值：利用主动测绘的方式，向目标发送各类数据包，根据不同的返回进而发掘、分析、提取目标特征。

正如在A Red Teamer Plays with JARM中提到的：

> 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.

> 基于主动测绘的威胁情报正在各个方向落地生根。通过对主动测绘数据各个维度的统计、分析信息，能够提供新的防护思路。

Happy hunting by using 360-Quake.

0x05 参考文章

- 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/