Challenge #5 Solution

by Peter Kacherginsky

The challenge is designed to teach you about PCAP file parsing and traffic decryption by reverse engineering an executable used to generate it. This is a typical scenario in our malware analysis practice where we need to figure out precisely what the malware was doing on the network.

As part of the challenge, you were provided two files: an executable binary and a PCAP network capture file. Let's look at the PCAP file using Wireshark to see if we can recognize the traffic. You should be able to notice a series of POST request like the one below:

```
POST / HTTP/1.1
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) KEY
Host: localhost
Content-Length: 4
Cache-Control: no-cache

UDYs
```

Figure 1: Initial POST

The user-agent string is a hint that the payload of these requests likely contains the key that you need to extract where each request appears to be a part of the larger message.

Let's aggregate all of the POST requests. You could do this manually by going through each request using Wireshark; however, this may be too laborious. Instead we are going to write a script that uses the excellent Scapy (http://www.secdev.org/projects/scapy/) utility to quickly parse the PCAP file and aggregate contents of all of the POST requests:

```
import sys

from scapy.all import *

if __name__ == '__main__':
    pkts = rdpcap(sys.argv[1])

    key = ""
    for pkt in pkts:
      if TCP in pkt and Raw in pkt and 'KEY' in pkt[Raw].load:
        headers, body = pkt[Raw].load.split("\r\n\r\n",1)
        key += body
```

```
    print "[+] KEY: %s" % key
```

Figure 2: Python script to combine all the POST data

Below is the result of executing this script with the provided `challenge.pcap` file:

```
$ Python httpaggregate.py challenge.pcap
 [+] KEY: UDYs1D7bNmdE1o3g5ms1V6RrYCVvODJF1DpxKTxAJ9xuZW==
```

Figure 3: Python Script Output

A combination of mixed alphanumeric character-set and the two padding '=' characters at the end may indicate that this is a base64 encoded string. Let's test this theory by trying to decode the above line. There are a variety of tools to do this; however, I am just going to use Python console:

```
>>> import base64
>>> key="UDYs1D7bNmdE1o3g5ms1V6RrYCVvODJF1DpxKTxAJ9xuZW=="
>>> base64.b64decode(key)
"P6,\xd4>\xdb6gD\xd6\x8d\xe0\xe6k5W\xa4k`%o82E\xd4:q)<@'\xdcne"
```

Figure 4: Python Shell decoding Base64 data

The result appears to be junk. Luckily we have the executable that produced the traffic captured in the PCAP file, so we can figure out whether or not there are additional steps involved in encrypting the transmitted key. First, let's do a basic static analysis by looking at what interesting strings we can find in the binary:

```
Mozilla/5.0 (Windows NT 6.1; WOW64) KEY
localhost
[!] Could not connect to server: %s
POST
[!] Could not open internet request.
[!] Error sending key data.
key.txt
[!] Could not open key file: %s
flarebearstare
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789+/
```

Figure 5: Strings from the challenge binary

The User-Agent string is the same one that we saw in the PCAP file: `Mozilla/5.0 (Windows NT 6.1; WOW64) KEY`. If this was a real malware this unique string could really be useful for writing a detection signature. Other interesting strings include "`[!] Could not open key`

`file: %s" and "key.txt"` which may indicate that the binary opens a text file that contains the actual key.

The rest of the strings appear to be debugging log entries except the last two. The purpose of the string `flarebearstare` is not immediately clear. The last string appears almost like the base64 alphabet `abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789+/` with the upper-case and lower-case blocks flipped. It is a common practice for malware authors to use custom Base64 alphabets, so take a mental note of this string for later analysis.

For a basic dynamic analysis, let's run the challenge in a safe environment and observe its behavior:

```
C:\sender.exe
[!] Could not open key file: key.txt
```

Great! The error message confirms our previous theory that the binary attempts to open a key file. Creating a sample file `key.txt` in the same directory and rerunning the sender.exe we get the following network error:

```
[!] Error sending key data.
```

At this point you should have some idea about what `sender.exe` does: it appears to read key data from the file key.txt in the same directory and send the contents over the network. Armed with this information we are now ready to dive into the binary with a disassembler of your choice.

After you locate the main function (**0x401100**), a particular CreateFileA call should jump out at you because it opens the `key.txt` file that we saw from basic static and dynamic analysis:

```
.text:00401117 push    esi
.text:00401118 push    0               ; hTemplateFile
.text:0040111A push    80h             ; dwFlagsAndAttributes
.text:0040111F push    3               ; dwCreationDisposition
.text:00401121 push    0               ; lpSecurityAttributes
.text:00401123 push    0               ; dwShareMode
.text:00401125 push    GENERIC_READ    ; dwDesiredAccess
.text:0040112A push    offset FileName ; "key.txt"
.text:0040112F mov     [ebp+NumberOfBytesRead], 0
.text:00401139 call    ds:CreateFileA
```

The contents of the file is read into the buffer "Buffer":

```
.text:0040116B push    0               ; lpOverlapped
.text:0040116D lea     eax, [ebp+NumberOfBytesRead]
.text:00401173 push    eax             ; lpNumberOfBytesRead
.text:00401174 push    524288          ; nNumberOfBytesToRead
.text:00401179 lea     eax, [ebp+Buffer]
.text:0040117F push    eax             ; lpBuffer
.text:00401180 push    esi             ; hFile
.text:00401181 call    ds:ReadFile
```

Next, the buffer holding the contents of the key file and buffer size are supplied to the function **sub_401250**:

```
.text:00401198 mov     edx, esi          ; key buffer size
.text:0040119A lea     ecx, [ebp+Buffer] ; key buffer
.text:004011A0 call    sub_401250
```

Notice that we are using Microsoft stdcall convention for this call. You can find more information about it here: https://en.wikipedia.org/wiki/X86_calling_conventions#stdcall.

Inside the **sub_401250** you will notice a loop which appears to modify the provided buffer:

```
.text:00401260
.text:00401260 loc_401260:
.text:00401260 mov     eax, 24924925h
.text:00401265 mul     esi
.text:00401267 mov     eax, esi
.text:00401269 sub     eax, edx
.text:0040126B shr     eax, 1
.text:0040126D add     eax, edx
.text:0040126F shr     eax, 3
.text:00401272 lea     ecx, ds:0[eax*8]
.text:00401279 sub     ecx, eax
.text:0040127B mov     eax, esi
.text:0040127D add     ecx, ecx
.text:0040127F sub     eax, ecx
.text:00401281 mov     al, byte ptr ds:KEY[eax] ; "flarebearstare"
.text:00401287 add     [esi+ebx], al
.text:0040128A inc     esi
.text:0040128B cmp     esi, edi
.text:0040128D jb      short loc_401260
```

Based on the above disassembly we could write the following pseudocode to illustrate its functionality:

```
for (i = 0; i < buff_len; i++) {
  buff[i] += KEY[i % KEY_LEN]
}
```

This appears to be a simple key based encryption. The string **flarebearstare** is used as the key.

Continuing further down the disassembly we will encounter the function **sub_4012A0** which uses the string `abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789+/`. Examining this function closer would prove that it is indeed a standard base64 encoder with an added twist of using a custom alphabet, just as we have theorized in the basic static analysis.

At last, we will see the reason the key message was broken up into 4 byte chunks due to the loop below. The loop loads 4 byte chunks of the encrypted and encoded message and passes them to **sub_401000** function which performs the POST request with the key snippet in the body:

```
.text:00401200 loc_401200:
.text:00401200 lea     ecx, [esi+ebx]
.text:00401203 call    sub_401000
.text:00401208 test    eax, eax
.text:0040120A jz      short loc_401229
.text:0040120C add     esi, 4
.text:0040120F cmp     esi, edi
.text:00401211 jb      short loc_401200
```

Based on the above information we can now update our PCAP parser script to not only aggregate the POST requests but also decode and decrypt the secret key. For that we will have to implement a base64 decoder which uses a custom alphabet and reverse the modifications performed by the encryption routine by subtracting the key bytes instead of adding them:

```
import sys
import base64

from scapy.all import *

encrypt_key = 'flarebearstare'
```

```
my_b64  = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789+/"
std_b64 = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"

def decode(str):
  str = str.translate(string.maketrans(my_b64, std_b64))
  return base64.b64decode(str)

def decrypt(str):
  key = ""
  for i, b in enumerate(str):
    key = key + chr(ord(b) - ord(encrypt_key[i % len(encrypt_key)]))

  return key

if __name__ == '__main__':
    pkts = rdpcap(sys.argv[1])
    print "[*] Parsing pcap: %s" % sys.argv[1]

    key = ""
    for pkt in pkts:
      if TCP in pkt and Raw in pkt and 'KEY' in pkt[Raw].load:
        headers, body = pkt[Raw].load.split("\r\n\r\n",1)
        key += body

    key_decoded   = decode(key)
    key_decrypted = decrypt(key_decoded)

    print "[+] KEY: %s" % key_decrypted
```

Running the above script against the provided `challenge.pcap` will produce the following output:

```
$ Python httpdecrypt.py challenge.pcap
[*] Parsing pcap: challenge.pcap
[+] KEY: Sp1cy_7_layer_OSI_dip@flare-on.com
```