Quick look at Nazar's backdoor - Network Communication

blog.malwarelab.pl/posts/nazar_eyservice_comm/

April 27, 2020

Intro

In previous episode we described capabilities of Nazar's EYService, an passive backdoor that utilize PSSDK to sniff on network traffic. In this post we'll take a look at how this malware communicates with outside world.

Binary Diffing

Malware is statically linked with PSSDK which makes analysis not very pleasant, and the fact that this software is long dead and has no documentation doesn't help either! However it was quite popular back in the day and its not that hard to find examples of usage, the most notable one being metasploit. Looking at their source code can give us some ideas how PSSDK API should be used.
But that is not enough, while we can guess some APIs a lot of them remains mystery - here with help comes binary diffing and Diaphora[1]. We only need to find a good binary to diff against.
Fortunately authors of PSSDK used a lot of uniqe names for their classes such as `CHNSyncList` or `CHNMemoryStream` or even `CBpfAsmLexer` - armed with those names finding a PSSDK dll is a matter of throwing them in your favorite search engine[2]. Once we had a binary our next problem turns out to be Diaphora and IDA Pro, Diaphora was ported to use newest version of IDA's API but it was also ported to Python3 and we prefer to stick to Python2 as long as it is possible! If you are like us you can find our changes here. Ok problems solved lets do some diffing!

C2 Protocol

After recovering API names from PSSDK we are presented with rather simple main function

```
DWORD __stdcall main_thread_func(LPVOID lpThreadParameter)
{
  struc_1 *v1; // edi
  int v2; // esi
  const void **v4; // edi

  v1 = MgrCreate();
  MgrInitialize((int)v1);
  v2 = MgrGetFirstAdapterCfg(v1);
  do
  {
    if ( !AdpCfgGetAdapterType(v2) )
      break;
    v2 = MgrGetNextAdapterCfg(v1, v2);
  }
  while ( v2 );
  g_Adp = AdpCreate();
  AdpSetConfig((int)g_Adp, v2);
  if ( !AdpOpenAdapter(g_Adp) )
  {
    AdpGetConnectStatus((int)g_Adp);
    Size = AdpCfgGetMaxPacketSize(v2);
    g_My_IP = (char *)AdpCfgGetIpA(v2, 0);
    AdpCfgGetMACAddress(v2, &g_My_MAC, 6);
    v4 = (const void **)BpfCreate();
    BpfAddCmd((int)v4, 0x30, 0x17);             // BPF_LD+BPF_B+BPF_ABS, [offset of
packet.ip.proto]
    BpfAddJmp((int)v4, 0x15, 0x11, 0, 1);       // BPF_JMP+BPF_JEQ+BPF_K,
IP_PROTO_UDP
    BpfAddCmd((int)v4, 6, -1);                  // BPF_RET+BPF_K
    BpfAddCmd((int)v4, 6, 0);                   // BPF_RET+BPF_K
    AdpSetUserFilter((int)g_Adp, v4);
    AdpSetUserFilterActive((int)g_Adp, (void *)1);
    AdpSetOnPacketRecv((int)g_Adp, (int)recive_packet, 0);
    AdpSetMacFilter((int)g_Adp, 2);             // mfOwnerRecv
    while ( 1 )
    {
      if ( g_SendPong == 1 )
      {
        g_My_IP = (char *)AdpCfgGetIpA(v2, 0);
        C2::response(2);
        g_SendPong = 0;
      }
      Sleep(0x3E8u);
    }
  }
  return 0;
}
```

Whats happening here, is basically boilerplate to set up BPF filter and a callback function that will handle incomming packets. BPF filtere here checks if 23th byte of a packet equals 17, 23th byte should be (in normal internet traffic) `Protocol` field of IPv4 header, 17 is a value denoting UDP in that field [3]. BPF filter checks if incoming packet is using UDP protocol [4].

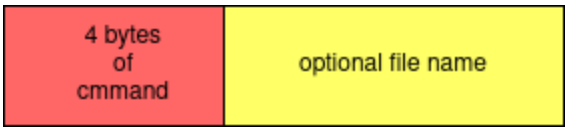
Requests from C2

Function `recive_packet` is responsible for stripping down headers of next protocols and finally call a function that we described in previous post. Two important things are happening during this parsing `Identification` field is extracted from IPv4 header and `Destination Port` from UDP header. First one is save for later use, it will be important during crafting response to c2, second is checked against **1234**. If value of this field is different nothing will happen. That tells us that this backdoor is passively listing on port **1234**.

```
int __cdecl handle_udp(udp_hdr *a1, int a2, int src_ip, int ip_id)
{
  int size; // edi

  size = HIBYTE(a1->len) - 8;
  ntohs(a1->src_port);
  if ( ntohs(a1->dst_port) != 1234 )
    return 0;
  handle_commands((c2_packet *)&a1[1], src_ip, ip_id, size);
  return 1;
}
```

After all headers are striped the content of UPD packet is straightforward



Response to C2

Lets move to responses, backdoor supports 3 types of responses

- send `pong`
- send victim info
- send file

Since UDP packets need to be crafted from scratch the code is quite messy but after applying proper types everything looks nice and clear

```
pPacket.ip_hdr._bf_0 = (pPacket.ip_hdr._bf_0 & 0xF | 0x40) & 0xF5 | 5;
pPacket.ip_hdr.ip_tos = 0;
v5 = htons(payload_size + 28);
pPacket.ip_hdr.ip_id = 1;
pPacket.ip_hdr.ip_len = v5;
pPacket.ip_hdr.ip_off = 0;
pPacket.ip_hdr.ip_ttl = -1;
pPacket.ip_hdr.ip_proto = 17;
pPacket.ip_hdr.ip_chksum = 0;
pPacket.ip_hdr.ip_src.S_un.S_addr = inet_addr(g_My_IP);
pPacket.ip_hdr.ip_dst = v2;
pPacket.udp_hdr.src_port = htons(1234u);
pPacket.udp_hdr.dst_port = htons(4000u);
pPacket.udp_hdr.len = htons(payload_size + 8);
pPacket.udp_hdr.checksum = 0;
```

We can even see some oddities that would make an good base for snort/suricata rule

- Ping

C2 can request a live check issuing command **999**, when malware sees a packet with this command it will replay to port **4000** with UDP packet containing simple string `101;0000;`

- OS info

C2 can request informations on infected host issuing command **555** when malware sees a packet with this command it will replay to port **4000** with UDP packet that contains:

- Computer name
- Version of operating system

packet will have a fallowing format: `100;%COMPNAME%;%WINNAME%;` [5]

- Send file

Various commands can produce a files with logs and bot has to exfiltrate them, this is done in a same way as previous commands however destination port is different. Instead of using hardcoded one, previously saved value from `Identification` field of incoming packet is used. In order to exfiltrate a file bot will create packets containing content of the file and one more packet with content '—%FILE_SIZE%' where %FILE_SIZE% denotes a size of file being send to c2.

Closing words

In this post we showed how EYService communicates with c2. Digging into network protocol allows us to better understand its capabilities and fix previous wrong assumptions, gaining full view of this malware. Understanding how malware is communicating is crucial for

detection as patterns used in network communication tend to stay longer unchanged contrary to code of malware itself. Finally passive backdoors are pretty rare and their analysis require knowledge of how internet protocols are build, so we are thankful for this opportunity to brush it up ;]

Snort/Suricata Rules

```
alert udp $HOME_NET 1234 -> $EXTERNAL_NET 4000 (msg:"Nazar EYService Pong
response");id:1; ttl:-1;content:"101;0000;";reference: url,
https://blog.malwarelab.pl/posts/nazar_eyservice_comm;classtype:trojan-activity;sid;1
alert udp $HOME_NET 1234 -> $EXTERNAL_NET 4000 (msg:"Nazar EYService OSInfo
response");id:1; ttl:-1;content:"100;";reference: url,
https://blog.malwarelab.pl/posts/nazar_eyservice_comm;classtype:trojan-activity;sid;1
alert udp $HOME_NET 1234 -> $EXTERNAL_NET any (msg:"Nazar EYService File exfiltrate
response");id:1; ttl:-1;content:"---";reference: url,
https://blog.malwarelab.pl/posts/nazar_eyservice_comm;classtype:trojan-activity;sid;1
```

Please note that those rules are provided as is, and are created based on code rather than actual traffic, and where not battle tested!

---

1. BinDiff is also an option but it has some problems with our binary ↩
2. example of PSSDK dll, c5ef3bd6a93edaca685e2ea796f0684b208b4700b8bdcf8dfbf78c47aa9562c9 ↩
3. https://en.wikipedia.org/wiki/List_of_IP_protocol_numbers ↩
4. there is an assumption here that all traffic is using Ethernet and IPv4 protocols on lower levels of OSI model ↩
5. Looking at the possible strings for os version shows us how old this malware can be as a versions goes from win95 to win xp ↩