Memory Analysis of .NET and .Net Core Applications

By:
Modhuparna Manna (Louisiana State University), Andrew Case (Volatility Foundation), Aisha Ali-Gombe (Towson University), and Golden Richard (Louisiana State University)

From the proceedings of
The Digital Forensic Research Conference
DFRWS USA 2022
July 11-14, 2022

Forensic Science International: Digital Investigation 42 (2022) 301404

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

DFRWS 2022 USA - Proceedings of the Twenty-Second Annual DFRWS USA

Memory analysis of .NET and .Net Core applications

Modhuparna Manna [b, c, *], Andrew Case [a], Aisha Ali-Gombe [d], Golden G. Richard III [b, c]

[a] Volatility Foundation, USA
[b] Center for Computation and Technology, Louisiana State University, USA
[c] School of Electrical Engineering & Computer Science, Louisiana State University, USA
[d] Department of Computer and Information Science, Towson University, USA

ARTICLE INFO

Article history:

Keywords:
Memory forensics
Language runtimes
Memory-only malware
Digital forensics

ABSTRACT

Memory analysis is a digital forensics technique whose goal is to model a computer system's state based solely on the analysis of a snapshot of physical memory (RAM). Memory forensics is frequently employed in incident response to detect and analyze modern malware and attack frameworks. Memory forensics is a particularly powerful tool for analyzing modern malware, which may exist only in memory and not touch non-volatile storage. Memory-only attacks leave no trace of the malware and its associated modules on the file system and all data that traverses the network is commonly encrypted. While initially focused on kernel level rootkits, memory analysis research efforts have recently shifted to detection of userland malware. This shift occurred as operating system vendors have strongly locked down the ability for kernel rootkits to load, and, in turn, malware authors have developed significant userland malware capabilities. In this paper, we present our effort to develop memory analysis capabilities that target a very powerful and widely abused set of userland runtimes: the .NET Framework and its replacement, .NET Core. To support automated and repeatable results, even for non-expert investigators, we developed a number of Volatility plugins that automatically target key areas of these runtimes and report any suspicious artifacts. Our suite of new plugins provides investigators with deep insight into the use of .NET on a target system as well as identification of suspicious and malicious components. These capabilities considerably advance a defenders' ability to combat, contain, and understand modern malware.

1. Introduction

The power of memory forensics is well known throughout the digital forensics and incident response (DFIR) communities. Importantly, many modern malware samples and attacker toolkits are truly "memory-only", meaning that *no* trace of an attack framework's executables and other components are written to disk. In addition, all related network traffic is commonly encrypted. These factors seriously hamper "traditional" storage-based forensic and necessitates the use of memory forensics to detect, analyze, and contain such threats. Two of the largest and most devastating attacks in 2021, the supply chain attack against SolarWinds and the widespread abuse of Exchange vulnerabilities, both relied heavily on memory-only payloads (Williams, 2021; Crowe, 2021). This reality led to the Cybersecurity and Infrastructure Security Agency (CISA) of the United States publishing several directives that explicitly ordered the collection and analysis of system memory when agencies responded to these threats (Cybersecurity Directives, 2021).

Memory analysis techniques were first developed to combat kernel level malware (Kirda, 2015). This approach was taken as rootkits could enter operating system kernels relatively easily and then take complete control of the system. Traditional forensics techniques, such as file system and live system forensics, are unable to detect kernel malware that purposely hides itself from such inspection. The widespread abuse by kernel rootkits led to all three of the major operating system vendors significantly locking down the ability of arbitrary kernel modules to load into the system (Kernel Code-Signing, 2016; Kernel Patch Protection, 2016; Pot, 2016).

* Corresponding author. Center for Computation and Technology, Louisiana State University, USA.
E-mail addresses: mmanna3@lsu.edu (M. Manna), andrew@dfir.org (A. Case), aaligombe@towson.edu (A. Ali-Gombe), golden@cct.lsu.edu (G.G. Richard).

https://doi.org/10.1016/j.fsidi.2022.301404
2666-2817/

This lock down of kernel access led to the development of significant userland (process level) malware. By abusing system APIs, such malware can still access a variety of hardware devices (web cameras, microphones), sensitive user data (keystrokes, generation of screenshots) and system activity (files accessed, network packets sent and received). Userland malware can also effectively hide itself from live forensics just as sophisticated kernel level malware does.

The power of userland malware led to a number of related memory forensics research efforts. These began with the detection of injected shellcode and PE files as those were the initial malware abuse vectors (Mandal, 2013). Attention also turned to userland runtimes, such as Swift and Objective-C (macOS) and ART (Android) (Manna et al., 2021; Android. Art and dalvik, 2016; Case, 2011). Our research effort advances userland memory forensics by providing significant analysis capabilities of the .NET runtime of Windows as well as the cross-platform .NET Core. We will simply refer to both of these runtimes as .NET where possible in the rest of the paper. The new tools that we present provide the ability to deeply inspect the loaded classes, class instances, fields, methods, and code of .NET assemblies (executables). .NET was chosen as our research target given the widespread abuse of it by real world malware, its native ability to load assemblies in a memory-only manner, and its direct control of PowerShell, which is another often abused feature of Windows. All of our techniques were implemented in Volatility 2 plugins and will be contributed to the open source project upon publication of this paper (The Volatility Framework, 2017).

2. Related work

Since the DFRWS 2005 challenge (DFRWS. DFRWS Online, 2005), which simply provided a Windows memory sample that needed to be analyzed, there has been an explosion of memory forensics research (Case and Richard, 2017). The efforts most closely related to ours include the generic detection of userland malware as well as deep inspection of userland runtimes.

2.1. Generic userland malware detection

Current and historical approaches to generic userland malware detection rely on detecting anomalies within the system state maintained by the kernel and userland loaders for tracking executables and libraries. For Windows memory samples, the use of the *malfind* plugin is a popular choice for this type of detection. It operates by inspecting the VADs (Dolan-Gavitt, 2007) (memory descriptors) of a process to discover memory regions with suspicious characteristics. This supports detection of a number of shellcode and DLL injection techniques. The *ldrmodules* plugin attempts to detect injected DLLs by cross-referencing the list of loaded DLLs tracked by the kernel data structures against those maintained by the userland loader. This can also detect a number of DLL injection and hiding techniques. We note that Volatility contains companion plugins that provide the same capabilities for macOS and Linux.

Unfortunately, these plugins and this type of analysis is not useful for detection of malicious .NET assemblies or of abuse of userland runtimes in general. This is because while Volatility (or any other memory analysis framework) can locate compiled applications, such as .NET assemblies or Objective-C executables, this is not enough information to determine if the application is malicious or benign. Instead, structured analysis must be performed against the runtime to provide enumeration of the capabilities and activities of a specific application.

2.2. Userland runtime analysis

The inability of existing approaches to detect malicious applications that abused userland runtimes led to a number of research projects that were focused solely on structured analysis of a particular runtime. Many of these projects focused on the Android runtime as this platform is heavily abused by malware and targeted attacks against Android users have led to dire consequences (Android. Art and dalvik, 2016; Macht, 2012; Case, 2011). This includes recording of web cameras, microphones, text messages, precise location information, and call history. Similarly, macOS malware that abuses Objective-C and Swift to compromise systems and spy on users has been developed (Stokes, 2020; Long, 2018; Amnesty International, 2019; Creus Tyler and Falcone, 2016; Goodin, 2019; Guarnieri and Anderson, 2017; Erwin, 2014; Falcone, 2017). Two previous research efforts deeply examined these runtimes to detect such malware (Case and Richard, 2016; Manna et al., 2021).

The end result of these projects was the automated ability to enumerate all classes, class instances, and code contained within a runtime-backed executable. This provides a comprehensive view of the activity performed by an application and alerts investigators to suspicious or malicious characteristics of applications. As mentioned previously, without the runtime-specific analysis, such capabilities are not possible.

The goal of our current research effort was to develop such capabilities for .NET and .NET Core applications. To date, there has been no such memory analysis research effort, and the sole application that delivers a partial set of these capabilities is the Son of Strike (SOS) debugging extension for WinDbg (SonofStrike. Sos, 2022). SOS is part of each .NET framework and allows for high-level extraction of runtime and per-class metadata. Several SOS extensions partially overlap our developed plugins, but SOS does not perform any analysis of method implementations and does not have any alerting capabilities for automated detection of malicious components. The shortcomings of SOS for memory analysis largely mirror those of using WinDBG or gdb for kernel level memory analysis. In general, debuggers are only capable of accessing active data structures, meaning no freed or historical data will be recovered. Furthermore, the debuggers simply list data structures found in samples, forcing an investigator to perform significant manual analysis and to rely on experience to determine what may be suspicious. This approach is time consuming, error prone, not scalable, and only available to expert investigators. Our plugins go well beyond what is recovered by SOS and provide automated alerts of malicious components and capabilities.

3. .NET Malware and attacker toolkits

To showcase the power of .NET malware, several high profile and powerful samples and attacker toolkits will now be discussed. Following this section, we will present our test environment where malicious applications were executed as well as our developed Volatility plugins that can automatically and accurately alert to the malicious behaviour.

3.1. Memory-Only payloads

The use of memory-only payloads by malware is now common place and implemented by essentially all widely used attack frameworks and malware families. The term memory-only refers to malware that does not create any data on the file system and, as such, only can be analyzed using memory forensics. This is in stark contrast to traditional malware that creates new executable files or libraries on the file system, or the recent trend of “file-less”

malware that uses existing files (shortcut files, registry hives, etc.) to persist on the disk. Both traditional and file-less malware can be detected and analyzed through filesystem forensics whereas memory-only malware cannot.

.NET has become a popular choice for malware developers who wish to develop stealthy, memory-only payloads. This shift occurred for two reasons. First, the widespread use of PowerShell as an attack framework fell out of favor after the introduction of powerful logging and detection capabilities by Microsoft. These include Script Tracing and Logging, which will record the contents of PowerShell scripts to log files before the script is executed (Microsoft. Script Tracing and Logging, 2021), as well as the introduction of the Anti-Malware Scan Interface (AMSI) introduced in Windows 10 (Microsoft. Antimalware Scan Interface, 2019). This interface allows Microsoft Defender as well as 3rd-party Anti-Virus products to scan PowerShell scripts before they are executed and to decide if the script is malicious or not.

The public shift to .NET in favor of PowerShell and other previously popular techniques occurred when version 3.11 of Cobalt Strike was released and introduced its *execute-assembly* command (Donut-Injecting, 2021; Kirk, 2018). *execute-assembly* executes a .NET assembly created in memory in the same manner as loading an assembly from disk and then executing it. Cobalt Strike is a advanced commercial tool meant for red team simulations, but is widely abused in attack campaigns by a variety of nation state-backed threat groups (MITRE. Cobalt Strike, 2021). It has significant memory-only capabilities and it only leaves traces on the local file system if the malware operator purposely chooses to do so. Cobalt Strike abuses an internal .NET mechanism to memory-only load files, but the .NET runtime also has a directly accessible API for memory-only loading of assemblies. This API, *Assembly.Load* (Microsoft. Assembly, 2021), accepts a byte array containing the .NET assembly (PE executable) and then loads it in a memory-only manner.

Covenant is a very popular and mature open source command and control (C2) framework written in .NET (Covenant. cobbr/Covenant, 2021a). The server allows direct control of one or more victim (client) systems, and its payloads execute in a memory-only manner. Capabilities against victim systems include keystroke logging, screenshot generation, lateral movement, privilege escalation, and other malicious actions. Fig. 1 shows Covenant's *AssemblyExecute* function, which first uses *Assembly.Load* to load an assembly passed as a byte array (*AssemblyBytes* parameter). It then retrieves the desired type (*TypeName*) followed by finding the reference to a method to be called inside of the type (class). It then calls the method (*Method.Invoke*) and returns the result.

Another approach to abuse *Assembly.Load* is the creation of a new instance of a desired type followed by *InvokeMethod* on the new instance. This approach leverages the *Activator.CreateInstance* API to dynamically create the desired instance. As documented by Microsoft, the malware used in the SolarWinds supply chain attack used this approach for its memory-only .NET components (Nafisi, 2021). Further reading on these loading techniques can be found in Tom Leemreize's Master's Thesis (Leemreize, 2021).

We discuss how our new Volatility plugins can automatically determine which assemblies were loaded from memory and how they can be extracted to disk for further analysis in Section 5.3.

3.2. AMSI bypasses

As mentioned previously, Microsoft developed the AMSI interface to provide much stronger protection against abuses of PowerShell and other built-in utilities. Attackers would still like the power that these languages and tools give though, so a common technique is for .NET malware to disable AMSI inside of the victim process before launching a malicious payload. This technique works as the AMSI libraries operate on a per-process basis and must be mapped into a process before scanning can be begin. Several websites catalogue the many AMSI bypasses that have been developed over time (Amsi-Bypass-Powershell, 2021; odzhan, 2019). These bypasses operate in one of two ways. The first is through the use of traditional API hooks that short circuit the scanning logic. These traditional hooks can be found with Volatility's *apihooks* plugin. The second form of bypasses perform manipulation of the .NET classes and functions that implement AMSI. A popular example of this type is shown in Fig. 2.

This bypass operates through the use of reflection to find the *System.Management.Automation.AmsiUtils* type followed by its *amsiInitFailed* member. It then sets the value of the member to true, which tricks the AMSI library into thinking loading of needed components failed. This then prevents future scans from being performed on any PowerShell scripts executed inside the victim process.

3.3. String and code obfuscation

Malware samples written in a variety of languages employ string and code obfuscation to both frustrate static analysis as well as make automated scanners have little or no useful information from which to create signatures. There are also legitimate uses of obfuscation, particularly for the .NET languages, as they are easy to decompile with a variety of tools, such as dnSpy (dnSpy. dnSpy, 2020).

.NET Reactor is a popular commercial product for highly advanced obfuscation (Reactor_download., 2021), and even the popular .NET deobfuscator, de4dot (de4dot. de4dot, 2020), is unable to fully handle Reactor protected executables. Its power has led to many malware samples using Reactor (Molerats Malware, 2021; s4tan. Analyzing the nasty, 2018; IBM X-Force Exchange, 2021). Confuser is an open source obfuscator used in APT campaigns

```
public static GenericObjectResult AssemblyExecute(byte[] AssemblyBytes,
                                                   String TypeName = "",
                                                   String MethodName = "Execute",
                                                   Object[] Parameters = default(Object[]))
{
      Reflect.Assembly assembly = Load(AssemblyBytes);
      Type type = TypeName == "" ? assembly.GetTypes()[0] : assembly.GetType(TypeName);
      Reflect.MethodInfo method = MethodName == "" ? type.GetMethods()[0] :
                                                     type.GetMethod(MethodName);
      var results = method.Invoke(null, Parameters);
      return new GenericObjectResult(results);
}
```

Fig. 1. One of Covenant's uses of Assembly.Load.

```
[Ref].Assembly.GetType('System.Management.Automation.AmsiUtils').\
    GetField('amsiInitFailed','NonPublic,Static').SetValue($null,$true)
```

Fig. 2. Powershell one-liner to disable AMSI

(Levene et al., 2017), and there are also numerous malware samples that employ custom obfuscation techniques (Malwarebytes Lab, 2016).

A significant benefit of memory forensics is the ability to extract program code and data after its transformation, such as after a binary is unpacked or strings have been decrypted. This allows for direct analysis of deobfuscated code and data and leads directly to actionable information without requiring any manual reverse engineering. In Section 5.7.2, we walk through an example of this power with our new *dotnet_field_values* plugin.

3.4. Malicious activities

Once active on a system, .NET malware employs a variety of techniques to capture sensitive user data, elevate privileges on the local system, and laterally move through an Active Directory domain.

3.4.1. Gathering screenshots

Screen capture is a common malware technique to learn the behaviour of victim users as well as scrape sensitive information (MITRE. Screen Capture, 2017). Using the *Graphics.FromImage* API followed by its *CopyFromScreen* method, .NET malware can generate screenshot images. This approach is used by many infamous .NET malware families, such as AgentTesla (Zhang, 2017; Fraunhofer. Agent Tesla, 2021), Dark Crystal (Thompson, 2020), and njRAT (CyberMaster V, 2021). It is also the approach used by Covenant (DefaultGruntTasks. Covenant, 2020).

3.4.2. Clipboard monitoring

Clipboard monitoring allows malware to steal the contents of a user's clipboard, which can contain highly sensitive data, such as passwords, user account names, and financial information. .NET provides the *Clipboard.GetText* API for obtaining access to clipboard data (Microsoft. Clipboard, 2021), and this API is widely abused by a variety of malware samples. The Evrial .NET malware abused clipboard access to hijack cryptocurrency transactions by using *Clipboard.SetText* to change the email address of the wallet that would be the recipient of transactions (Paganini, 2018).

3.4.3. Keystroke logging

The previously described malware capabilities all relied on functionality built into the .NET runtime. .NET malware is not limited to just the API provided by the runtime directly, however, as native components can be imported and used. Keystroke logging is an example of when malware leverages this facility as it will often abuse the *SetWindowsHookEx* function of *user32.dll* for this purpose. Fig. 3 shows decompiler output from dnSpy against a sample of the Orcus RAT (Morphisec Labs, 2019; Giuseppe Scalzi, 2020).

As can be seen, *GetKeyboardState*, *SetWindowsHookEx*, and *CallNextHookEx* are all being imported. Note that these imports do not populate in the normal import address table (IAT) of the assembly. Instead, .NET-specific analysis must be performed to uncover enough information for analysis, as discussed in Section 5.9.

3.4.4. Microphone recording

There are a variety of methods .NET malware uses to sample and save recordings from the system microphone. Orcus leverages the *MMDeviceEnumerator* C# interface whereas DCRat (Thompson, 2020), CrimsonRAT (Huss, 2021) and numerous other tools used by APT groups leverage the open-source NAudio project. There are also samples that abuse the ability to import native functions to directly call *MciSendStringA* and related APIs.

3.4.5. Web camera recording

Recording of web camera data poses a serious security and privacy risk to victims. Many .NET malware samples used for spyware and espionage contain the ability to record footage from attached web cameras and then exfiltrate them to the malware operators. The CrimsonRAT malware achieves this through use of the *capCreateCaptureWindowA* native function. This creates a handle that can then be used to interact with the web camera. Web camera footage is then scraped by sending a *WM_CAP_GRAB_FRAME* message to the camera, which produces an image of the current video frame. This is followed by *WM_CAP_EDIT_COPY*, which places the captured frame into the clipboard. The malware then retrieves the clipboard content, which is stored as a bitmap image, and converts it to a byte array for exfiltration.

3.4.6. Lateral movement

The mature attack frameworks used by threat actors and red teams contain the ability to laterally move through an environment once initial access is gained. Given .NET's flexibility, there are a number of stealthy mechanisms through which this can be achieved. The SharpSploit integration of Covenant shows how this goal can be achieved through the use of WMI, PowerShell Remoting, and DCOM (Covenant. cobbr/Covenant, 2021b). Numerous frameworks developed by APT groups contain similar techniques to move through compromised environments in a manner that will likely be undetected.

```
[DllImport("user32")]
internal static extern int GetKeyboardState(byte[] pbKeyState);

[DllImport("user32.dll", SetLastError = true)]
internal static extern IntPtr SetWindowsHookEx(HookType hookType,
                                               NativeMethods.HookProc lpfn,
                                               IntPtr hMod, uint dwThreadId);

[DllImport("user32.dll")]
internal static extern IntPtr CallNextHookEx(IntPtr hhk, int nCode, IntPtr wParam, IntPtr lParam);
```

Fig. 3. Importing APIs for keystroke logging.

4. Runtime versions and test environment

4.1. .NET vs .NET core

There are currently two distinct versions of the .NET framework supported by Microsoft. The original, which is now officially called the *.NET Framework*, is still included as the default version in Windows installs, is closed source, and version 4.8 will still be supported by Microsoft for the foreseeable future (Microsoft. Lifecycle FAQ, 2021). The second version was originally called *.Net Core*, but now is referred to as *.NET 5* (Microsoft., 2021a). .Net Core (.NET 5) is cross platform, open source, and the intended sole future runtime by Microsoft. Given the long term stable support of both runtime versions by Microsoft and a few significant differences in how the runtimes handles key data structures, we felt our work would not be complete if our developed memory forensic algorithms did not cover both versions. During our discussion of runtime internals and plugin development, we will explicitly refer to the areas that differ between the two, and the rest can be assumed to be the same.

4.2. Test Environment

To cover both forms of the .NET runtimes that are now used in the real world, our test environment included two systems. Both were VMware virtual machines running Windows 10 $\times$ 64 version 20H2 Build 19042. The first system ran Version 4.8.04084 of the .NET Framework, and the second system ran version 3.1 of .NET Core. Both of these were released within the last year and are still supported by Microsoft. Our study of a wide variety of changes to both frameworks showed us that including many different versions of each framework in our testing would only duplicate effort and that the changes needed to support differing build versions, which is discussed next, would not require changes to our algorithms but only the type and symbol information.

4.3. Retrieving types and symbols

Before we could develop Volatility plugins to enumerate .NET information, we first needed a method to determine the type and symbol information for a runtime version. Volatility 2 encodes type information into *vtype* files, which are Python hash tables of included data structures. Per-application symbol information has no standard method to be included into Volatility 2, so we simply incorporated it into our set of plugins directly.

4.3.1. .NET core

To support .NET Core, we needed the type information from *coreclr.dll* inside of the target process. This is Common Language Runtime (CLR) DLL that powers the runtime, and Microsoft hosts complete PDB files for each version of the DLL on its symbol server. This means the full definition for each type as well as the offset of each symbol is present within the PDB file. This should have made automated incorporation of the information into our plugins very simple, but unfortunately the tool used by Volatility to parse PDB files, *pdbparse*, broke on all *coreclr.dll* versions tested. This occurred for two reasons. First, the CLR makes heavy use of C++ and pdbparse seemingly does not support the streams produced by C++ applications. Second, pdbparse has not been frequently updated and there appears to be newer streams within the PDB files that are unknown to pdbparse. Given the significant effort it seemed that pdbparse would need to support these PDB files, we instead chose to parse them with WinDbg. We then manually wrote the vtype files for just the data structures and members that we needed for our plugins to operate.

4.3.2. .NET framework

The CLR of the .NET Framework is contained within *clr.dll*. The PDB files for this DLL are also on the Microsoft symbol server, but unfortunately they do not contain complete type information and only partial symbol information. This includes none of the type definitions we need for our analysis, which required us to the manually reverse engineer each DLL version to recover the correct offsets. Luckily, only a few of the member offsets changed between versions, which made reversing of subsequent versions only take a few minutes as the functions to analyze were already known. We have left the name of and information about the functions that reference each member as comments in our Volatility plugins to aid future investigations.

5. .NET internals and memory analysis

In this section, we discuss the .NET internals necessary to detect and analyze malware as well as our new Volatility plugins that automate the process. We would like to note that the .NET runtimes are both *extremely* complicated and a full understanding requires reading thousands of lines of code as well as runtime debugging and static analysis. As such, we only discuss the most crucial aspects to reserve room for presentation of research results.

5.1. Analysis goals

To shape the following sections, we will first present our analysis goals. When analyzing a .NET application in memory our aim is to recover the following information:

- The set of loaded classes
- For each class, its fields and methods
- For each field, its name and type
- For each method, its definition and location in memory
- Class instances (objects)

With these base capabilities, we can develop automated mechanisms to alert to suspicious and/or malicious code and data. We will now explain how we achieved these goal for both major runtime versions of .NET.

5.2. Application domains

Every .NET application runs within an *application domain*, which is conceptually a system process inside the CLR and the container for all the code and data of the application and its dependencies. Application domains are handled differently between each runtime version as the .NET Framework supports multiple application domains within the same process whereas .NET Core only supports one. To find the loaded classes in a process, we must begin by locating its application domain data structure (*class AppDomain*).

To find the application domain(s) of a .NET framework process, the global *SystemDomain::m_appDomainIdList* can be used. This is an *ArrayList* where each element is an *AppDomain* instance, and enumerating it will recover all application domains. .NET Core is much simpler since it only support one application domain, which is stored in the *AppDomain::m_pTheAppDomain* global variable. Once an application domain is found, its assemblies can then be enumerated.

5.3. Assemblies

.NET assemblies are regular PE (.exe or.dll) files compiled from C# or other .NET language that depend on the CLR (Microsoft. Assemblies in, 2021). Each loaded assembly is tracked in an

ArrayList stored in the *m_Assemblies* member of *AppDomain*. Each assembly is represented by an *DomainAssembly* instance, which points to the containing *Assembly* instance. The *Assembly* class contains several useful pieces of information starting with a reference to information about the backing PE file, which is stored as a *PEAssembly* class instance. From the *PEAssembly*, we can follow its *m_identity* member, which is of type *PEImage*.

PEImage leads us to the on-disk path of an assembly as well as its base (load) address. The full path of the assembly on disk is referenced from *PEImage.m_path*, assuming the executable was not loaded from memory. The path is held as an *SString*, which stores the string length (*m_size*) as well a pointer to the characters (*m_buffer*). When a *SString* is initialized, it is set to *empty*, which means its *m_buffer* is set to the address of the *SString::s_EmptyBuffer* global variable. Once one or more characters are added to the string, then the *m_buffer* pointer is updated to the location of the characters.

If an executable is loaded from memory, then its *m_buffer* will always point to the global empty buffer variable as it has no real path. This is extremely useful to us as it not only gives us a direct marker to determine if an executable was loaded memory-only, but it also avoids problems that memory smearing could cause if the value was set to NULL. This is a common occurrence in smeared data and would certainly lead to many false positives when detecting memory-only executables.

PEImage.m_pLayouts points to an array of *PEImageLayout* instances that describe how the assembly is loaded into memory. By using the first element of this array, we can obtain the base address of the executable from its *m_base* member. This tells us exactly where the executable loaded into memory and is the only definitive source for this information since .NET executables do not populate the data structures used by Volatility's existing *dlllist* plugin. It is also the information we need to extract the memory-only executable to disk for further analysis.

5.3.1. dotnet_memory_only plugin

To automate the detection and extraction of memory-only assemblies, we created the *dotnet_memory_only* plugin. This plugin enumerates all loaded assemblies and then checks if the *m_buffer* member of their *m_path* points *SString::s_EmptyBuffer*. This directly tells us if the assembly is memory-only. For each memory-only assembly, we then determine its base address from *PEImageLayout* as described previously. The plugin then passes the base address to the existing *dump_pe* API of Volatility to write a properly formatted executable file to disk. This allows analysts to perform static analysis on an otherwise memory-only executable. Fig. 4 show the output of *dotnet_memory_only* against a sample where Covenant was used to infect the victim system.

As part of its operations, Covenant loads several memory-only .NET executables, which are used to communicate with the C&C server as well as service tasks sent from the malware operator. As shown in the figure, our plugin automatically determines the name and location of these executables and extracts them to the given output folder. Running the *file* command on these extracted files shows that are all indeed .NET assemblies. Through the use of this plugin, investigators with no previous knowledge of .NET malware can immediately extract all memory loaded assemblies from a memory sample. We note that since the path of the executable does not exist for memory-only assemblies, we use the name of the first module stored in the assembly instead.

5.4. Modules

Modules are components stored within an assembly, such as other executable files, resources, graphics, and other items. In the .NET Framework, assemblies can consist of multiple modules, which are tracked in the *m_Modules* member. .NET Core only allows one module per assembly though, which is stored in the *m_pModule* member (Microsoft., 2021b).

Each module is represented by a *Module* structure, which contains a significant amount of relevant metadata. First, it contains a basic string name of the module. This is what *dotnet_memory_only* uses to construct a name for memory-only assemblies. Next, it contains information on the types (classes) defined by the module (*TypeDefToMethodTableMap*) as well as the types referenced by the module (*TypeRefToMethodTableMap*). Enumerating and analyzing these are discussed in sections 5.5 and 5.11. A module also contains a reference to the *PEAssembly* structure that holds the metadata database references for the types, fields, and methods of the module. This in-memory database will be discussed in several upcoming sections.

5.5. Defined classes

By enumerating each element of the *TypeDefToMethodTableMap* member of *Module*, we can locate the *MethodTable* of each type (class). It is important to note that a *method table* is used to define and track every loaded class and is not just used to track methods. Once we have the method table for a class, we can then determine its name and namespace and obtain a reference to its class (*EEClass*) instance.

Obtaining the name and namespace of a class requires analyzing the previously mentioned in-memory metadata database. To query into the database you must obtain the *metadata token* of the object you wish to query information about, which in this case is a method table. The structure of this database is highly complex, but, briefly, it is held as two parallel arrays of forty-five elements. One array holds the metadata entries needed to index into the other array to get the actual data you need. Each array element is for a specific type of information, and for *TypeDef* (class) elements, the column is *TBL_TypeDef*, which has an integer value of two (2). After determining the column, you then must know the *row* of the specific information that you wish to find, which for class names and namespaces is *COL_Name* and *COL_Namespace*. You then index your

```
$ python vol.py  -f data.lime --profile=Win10x64_19041 dotnet_memory_only -D output
Volatility Foundation Volatility Framework 2.6
Module                          Base               Result
------------------------------- ------------------ ------
aulqzbzo.3pq                    0x000001b149eb0000 OK: aulqzbzo.3pq.0x1b149eb0000.dmp
qxle4kw3.jad                    0x000001b149ed0000 OK: qxle4kw3.jad.0x1b149ed0000.dmp
td554lhy.s2d                    0x000001b149ee0000 OK: td554lhy.s2d.0x1b149ee0000.dmp
CSPHZsvIYrVNQAEdUqkPnzbwhnhK    0x000001b14a040000 OK: CSPHZsvIYrVNQAEdUqkPnzbwhnhK.0x1b14a040000.dmp

$ file *
CSPHZsvIYrVNQAEdUqkPnzbwhnhK.0x1b14a040000.dmp: PE32 executable (DLL) (console) Intel 80386 Mono/.Net assembly
aulqzbzo.3pq.0x1b149eb0000.dmp:                 PE32 executable (GUI) Intel 80386 Mono/.Net assembly
qxle4kw3.jad.0x1b149ed0000.dmp:                 PE32 executable (DLL) (console) Intel 80386 Mono/.Net assembly
td554lhy.s2d.0x1b149ee0000.dmp:                 PE32 executable (DLL) (console) Intel 80386 Mono/.Net assembly
```

Fig. 4. Locating and extracting memory-only assemblies.

desired table at the correct row and column to obtain the information needed to index the other array that holds the actual data. The "data" in the case of class names and name spaces is an integer index into the string pool of the *Module*. This then contains a pointer to a NULL-terminated C string of the class name. This extremely complicated "querying" process is performed by several of our plugins, but going forward, for space and sanity purposes, we will simply refer to it as querying the metadata database.

After obtaining the name and namespace of the class through its method table, analysis of fields and methods can begin by following the *m_pEEClass* member to the *EEClass* instance.

5.6. Fields

Each field (static and instance variable) of a class is represented by a *FieldDesc* instance. The *m_pFieldDescList* member of *EEClass* points to an array of these descriptors and there is an element for each field of the class. There are three pieces of information per field that our plugins require. The first is the name of the field, which comes from querying the metadata database. The second is the offset of the field inside of a class instance, which is needed to recover an instance's value. Third, we need the type of the field to parse it correctly. Gathering the type also requires querying the metadata database, but instead of getting a simple pointer to a string name from the string pool, our plugin must instead inspect the *blob* heap. The value returned from this query is then an element type, which can be a predefined constant for native types or a metadata token for types defined by assemblies.

Given our ability to enumerate fields and their types, we wrote two plugins that showcase these values. The first, *dotnet_amsi*, locates the *System.Automation.Management.AmsiUtils* class and then verifies its *amsiContext* and *amsiInitFailed* members. Both of these have been targeted by a variety of AMSI bypass techniques as discussed in Section 3.2.

The second plugin, *dotnet_fields*, searches for fields of types known to be abused by malware. Although malware will often use APIs for accessing the network, registry, and the file system, so will a substantial number of benign applications. To avoid these issues, we instead focused our study of malware on types that are abused by a wide variety of malware and that have few legitimate uses. This study led us to discover two main categories, with the first being system classes for dynamically loading code. To detect this activity through fields analysis, our plugin will alert to any field with a parent type of *System.Reflection.Assembly* or *System.Runtime.CompilerServices*. The second category related to malware that aimed to perform lateral movement and credential harvesting. To detect these, our plugin will alert to any field with a parent type of *System.DirectoryServices* or *System.Security.Principal.SecurityIdentifier*. Fig. 5 shows *dotnet_fields* executed against the same sample infected with Covenant as shown in Fig. 4.

As can be seen, the plugin reports that the *td554lhy.s2d* module has a class named < *Module* > with a member whose field type *System.Reflection.Assembly*. We note that this same module was reported as memory-only by *dotnet_memory_only*. Second, the field name is listed as < *INVALID* > by our plugin as Covenant and other .NET malware will destroy the actual names of fields and other properties in an attempt to hinder analysis. Our plugins deal with this by allowing direct extraction of the assemblies in memory. An investigator then knows exactly where to look and can proceed with deeper analysis using tools such as dnSpy.

5.7. Instances

5.7.1. Enumerating instances

As Fig. 5 showed, simply knowing the types of fields can directly alert to malware, but as investigators we also want to know the *value* of fields for any instances of a class of interest. Class instances are stored as objects on the .NET heap, and we find all instances of classes by scanning this heap. Our scanner leverages the fact that each object starts with a pointer to the method table for its defining class. This allows us to first enumerate every loaded class as described in Section 5.5 and then scan for pointers to the method tables of those classes. This effectively finds every instance of every class with the exception of those stored on pages not present in the memory sample.

As we are enumerating classes, we also save the per-field information of a class, including the name, type, and offset. The offset tells us the distance from an instance to the value of a particular field. Fields of primitive types are stored directly at the offset whereas fields that are references to other class objects are stored as pointers.

5.7.2. dotnet_field_values plugin

To automate the discovery and extraction of instance fields, we developed the *dotnet_field_values* plugin. By default, this plugin will list the value of every field of every instance for types that it knows how to parse. This is highly useful for situations where investigators want to use string and/or regular expression search for data that is often useful in malware investigations, such as file paths, URLs, and registry keys.

To demonstrate the value of this plugin, we wanted to test if we could generically find the URL and/or IP address of the Covenant command and control (C2) server that we setup for testing. Finding the C2 server in real investigations is often a crucial and early step as, once found, network logs can be searched to find other infected systems, IDS sensors can be configured to alert on any future connection attempts to the C2 server, and firewall rules can be added to block access to the C2 server. IOC data related the C2 server can also be published to help outside organizations defend against the threat.

Running the plugin against the Covenant-infected sample and then searching the output for IP address and URL patterns led us to discover the *Hostname* and *CovenantURI* fields of the *GruntExecutor.HttpMessenger* class. These fields are of type *System.String*. The parsed value of *Hostname* was the IP address of our Linux virtual machine used as the C2 server (*192.168.20.108*), and the parsed value of *CovenantURI* (http://192.168.20.108:12345) was our configured C2 URI. We then verified these fields as correct by first recognizing that we configured our Grunt, which is the Covenant term for a victim system, to communicate back to the C2 server over HTTP to the exact URI as stored in *CovenantURI*. We then studied the Covenant source code for *HttpMessenger* to ensure that the fields

```
$ python vol.py  -f data.lime --profile=Win10x64_19041 dotnet_fields
Volatility Foundation Volatility Framework 2.6
Module       Class        Field        Type
------------ ------------ ------------ ----
td554lhy.s2d <Module>     <INVALID>    System.Reflection.Assembly
```

Fig. 5. Automated detection of reflection field types.

were storing the configured C2 server address and URI for Grunts. Our source code study confirmed it, and our example here shows just how powerful the generic enumeration and parsing of instance object fields can be in investigations.

5.8. Methods

The ability to enumerate class methods is essential for complete analysis of the .NET runtime. Through method analysis we can detect malicious code that is imported by an application, malicious code hosted by an application, and often alert investigators to the most impactful locations to focus analysis. From a high level view, .NET classes can use and implement two sets of methods. The first set are those implemented in a .NET language, such as C#, and the other set are methods (functions) compiled from C/C++ and stored in DLL files.

The set of methods for a class are referenced from the *m_pChunks* member, which is of type *MethodDescChunk*. The chunks belonging to a class are stored in a linked list, and each chunk holds an array of *MethodDesc* instances, where each element is a method of that class. Recovering the name of a method involves querying the metadata database based on the descriptor's token. To recover the parameters and parameter types of a method, the same blob heap as fields must be examined through the metadata database. This query returns a type signature that sequentially embeds the type for each parameter in the same manner as the single signature does for a field.

Besides simply recovering the name and parameter types of a method, our plugins also recover the address of the method implementation. As described by Microsoft, each method descriptor holds little information about its implementation until the form (type) of that descriptor is understood. Of interest to our plugins are the methods that are of type *IL* and *NDirect*.

5.9. Native (C/C++) methods

NDirect methods are those implemented in a native DLL, and malware commonly abuses this feature to obtain capabilities not possible in regular .NET code. As mentioned previously, keylogging is a popular example of this abuse through the import of *SetWindowsHookEx* and related functions.

Our plugins determine the type of method by examining its *classification*, which is the lower three bits of the descriptor's *m_wFlags* member. NDirect methods have a classification of *mcNDirect*. Once determined to be a NDirect method, the method descriptor can then be treated as a *NDirectMethodDesc*. This structure tells us the name of the DLL the function is being imported from, the string name of the function being imported, and the resolved address. With this information, our plugins are able to alert to suspicious and malicious functions being imported by applications, just as a variety of malware analysis tools do for applications written in C/C++. This information replicates the process of import address table (IAT) analysis and is one of the most common static analysis tasks as it directly reveals the initial set of an executable's capabilities.

Fig. 6 shows the output of *dotnet_ndirect_methods* plugin against a sample infected with Covenant and where the keylogger task was launched from the C2 server. This action sends the keylogging assembly to the victim machine for memory-only execution.

As can be seen, *SetWindowsHookEx* along with its related functions from *kernel32.dll* and *user32.dll* are correctly reported and tied to their owning .NET module (*ohydkf5n.ceo*).

5.10. IL .NET methods

Methods implemented in .NET are initially stored as IL (.NET intermediate language) code and then converted to assembly (JITed) based on the host architecture. The CLR employs a lazy approach to method jitting, and only performs the operation once a method is called for the first time. This has a significant impact on our analysis as, for functions that were never called before the memory sample was taken, there is no native assembly (x86 or x86_64) instructions for the function. Instead, a pointer to the IL code is all that is available and is stored directly after the method descriptor.

Given this constraint, we limited our analysis of IL methods to only those that are JITed. We chose this route for a few reasons. First, JITed functions are those that were actually executed on the system being investigated, making them the most relevant. Second, there are ways to modify .NET code post-JIT, as described in (xpnweired, 2021), and only the in-memory code that we analyze would reflect these changes. This strongly motivated us to perform automated static analysis on the JITed code. Third, given our plugin's ability to extract assemblies to disk for further analysis, analysts who wish to investigate any function of an assembly can easily load the extracted assembly into powerful tools, such as dnSpy, for complete decompilation of the IL code.

To investigate JITed IL methods, we developed the *dotnet_il_methods* plugin. This plugin has a few modes, the first of which will simply print a disassembly of desired methods or of all methods of a module. The more targeted use involves the automated identification of methods that call suspicious methods. The suspicious methods called can be NDirect or IL methods. The addresses of called functions are matched to symbols (function names) by gathering the exported functions of mapped DLLs plus the symbol addresses gathered when walking the method table of each class. This allows us to identify calls to NDirect and IL methods. Fig. 7 shows *dotnet_il_methods* against a memory sample infected with SharpStage, which is a malware family used by the MoleRats APT group (molerats. Molerats apt, 2020).

In this instance, the plugin is reporting the use of the *System.Drawing.Graphics.FromImage* API by the *jgi6MuZBI* method of the *Form1* class. This immediately tells the investigator that the malware is capable of screenshot generation and the exact function to begin static analysis. This figure also shows the power of defeating obfuscation as, even though the function name is a random string, its purpose is still automatically inferred.

5.11. Referenced classes

The final capability that we developed analyzed the set of classes referenced (used) by the class under examination. This is an important capability for analyzing IL code as any classes used by a class will be populated with *TypeRef* entries within the class' metadata. These *TypeRef* entries allow us to automatically identify references to classes known to be abused by malware, such as those discussed in Section 3. The *dotnet_class_references* plugin implements this alerting capability based on a configurable set of classes. The names and types of each referenced class is recovered by querying the metadata database. This plugin provides a very powerful capability as it is the equivalent to examining the import table of native applications, but is instead on a per-class basis inside of .NET executables. Given that many malware samples contain dozens of classes, this plugin points investigator directly to ones of high interest.

```
$ python vol.py  -f data.lime --profile=Win10x64_19041 dotnet_ndirect_methods
Volatility Foundation Volatility Framework 2.6
Module       Class        DLL          Method
------------ ------------ ------------ ------------------------
ohydkf5n.ceo Keylogger    kernel32.dll GetModuleHandle
ohydkf5n.ceo Keylogger    user32.dll   CallNextHookEx
ohydkf5n.ceo Keylogger    user32.dll   GetForegroundWindow
ohydkf5n.ceo Keylogger    user32.dll   GetWindowText
ohydkf5n.ceo Keylogger    user32.dll   SetWindowsHookEx
ohydkf5n.ceo Keylogger    user32.dll   UnhookWindowsHookEx
```

Fig. 6. Automated Reporting on the use of SetWindowsHookEx

```
$ python vol.py  -f data.lime --profile=Win10x64_19041 dotnet_il_methods
Volatility Foundation Volatility Framework 2.6
Module Function  Called Function
------ --------  ---------------
Form1  jgi6MuZBI System.Drawing.Graphics.FromImage
```

Fig. 7. Automated Reporting on the use of Screenshot APIs.

6. Conclusions and future work

Userland malware continues to be a significant threat given the powerful and stealthy capabilities provided by modern runtimes and system APIs. This includes loss of privacy through monitoring of web cameras, microphones and keyboards, as well as loss of security due to malicious payloads that are well hidden from live system analysis. These payloads provide for data exfiltration, lateral movement, and privilege escalation.

In this paper, we have documented our effort against two widely abused userland runtimes, .NET Framework and .NET Core. Our new analysis techniques, implemented in new Volatility plugins, automate the analysis, detection, and extraction of malicious .NET assemblies from memory. The incorporation of our new plugins into investigative workflows will provide digital forensics analysts a significant edge over modern malware.

Going forward, our team aims to automate the incorporation of type and symbol information from all CLR DLL versions, as this is the sole remaining step for a completely automated workflow. For the closed-source .NET framework, this will involve automating the static analysis of functions that reference needed class members. We intend to explore the Seance framework presented at DFRWS 2021 for this purpose Maggio et al. (2021). For the open source .NET core, we aim to add parsing of the new PDB stream types to pdbparse so that it can correctly create the type information relied upon by both Volatility 2 and Volatility 3.

Acknowledgements

This research is supported by grant # 1703683 from the National Science Foundation.

References

Amnesty International, 2019. Phishing attacks using third-party applications against Egyptian civil society organizations. https://www.amnesty.org/en/latest/research/2019/03/phishing-attacks-using-third-party-applications-against-egyptian-civil-society-organizations.

Amsi-Bypass-Powershell, 2021. S3cur3Th1sSh1t/Amsi-Bypass-Powershell. https://github.com/S3cur3Th1sSh1t/Amsi-Bypass-Powershell.

Android. Art and dalvik. https://source.android.com/devices/tech/dalvik/, 2016.

Case, Andrew, 2011. Forensic Memory Analysis of Android's dalvik vm. Source Seattle.

Case, Andrew, Richard III, Golden G., 2016. Detecting objective-C malware through memory forensics. In: Proceedings of the 16th Annual Digital Forensics Research Workshop (DFRWS).

Case, Andrew, Richard III, Golden G., 2017. Memory forensics: the path forward. Digit. Invest. https://doi.org/10.1016/j.diin.2016.12.004.

Covenant cobbr/Covenant, 2021a. https://github.com/cobbr/Covenant.

Covenant cobbr/Covenant, 2021b. https://github.com/cobbr/Covenant/blob/master/Covenant/Data/Tasks/SharpSploit.LateralMovement.yaml.

Creus, Dani, Tyler, Halfpop, Falcone, Robert, 2016. Sofacy's 'Komplex' OS X Trojan. https://unit42.paloaltonetworks.com/unit42-sofacys-komplex-os-x-trojan/.

Crowe, Jonathan, 2021. Microsoft Exchange 0-Day vulnerabilities mitigation guide: what to know & do now. https://www.ninjaone.com/blog/microsoft-exchange-0-day-vulnerabilities-mitigation/.

CyberMaster V, 2021. Just another analysis of the njRAT malware - a step-by-step approach. https://cybergeeks.tech/just-another-analysis-of-the-njrat-malware-a-step-by-step-approach/.

Cybersecurity Directives, 2021. Emergency directive 21-01. https://cyber.dhs.gov/ed/21-01/.

de4dot. de4dot. https://github.com/de4dot/de4dot, 2020.

DefaultGruntTasks Covenant, 2020. https://github.com/cobbr/Covenant/blob/c53155615563cf68979820356b8430e4eb01207d/Covenant/Data/Tasks/DefaultGruntTasks.yaml#L332.

DFRWS. DFRWS Online. http://old.dfrws.org/2005/challenge, 2005.

dnSpy. dnSpy. https://github.com/dnSpy/dnSpy, 2020.

Dolan-Gavitt, B., 2007. The VAD tree: a process-eye view of physical memory. In: Proceedings of the 2007 Digital Forensic Research Workshop.

Donut-Injecting, Wover, 2021. NET assemblies as shellcode. https://thewover.github.io/Introducing-Donut/.

Erwin, Derek, 2014. Ventir Trojan intercepts keystrokes from Mac OS X computers. https://www.intego.com/mac-security-blog/ventir-trojan-intercepts-keystrokes-from-mac-os-x-computers.

Falcone, Robert, 2017. XAgentOSX: sofacy's XAgent macOS Tool. https://unit42.paloaltonetworks.com/unit42-xagentosx-sofacys-xagent-macos-tool/.

Fraunhofer. Agent Tesla. https://malpedia.caad.fkie.fraunhofer.de/details/win.agent_tesla, 2021.

Giuseppe Scalzi, 2020. Reversing a .NET Orcus dropper. https://blog.compass-security.com/2020/04/reversing-a-net-orcus-dropper/.

Goodin, Dan, 2019. Newly discovered Mac malware uses "fileless" technique to remain stealthy. https://arstechnica.com/information-technology/2019/12/north-koreas-lazarus-hackers-up-their-game-with-fileless-mac-malware/.

Guarnieri, Claudio, Anderson, Collin, 2017. iKittens: Iranian actor resurfaces with malware for Mac (MacDownloader). https://iranthreats.github.io/resources/macdownloader-macos-malware/.

Huss, Darien, 2021. Operation transparent tribe. https://www.proofpoint.com/sites/default/files/proofpoint-operation-transparent-tribe-threat-insight-en.pdf.

IBM X-Force Exchange, 2021. Analyzing .NET malware with windbg and sos, part II. https://exchange.xforce.ibmcloud.com/collection/Analyzing-NET-malware-with-windbg-and-sos-part-II-5cd17e4cff037144a041eb53d9145d4e.

Kernel Code-Signing, 2016. Kernel-mode code signing requirements. https://msdn.microsoft.com/en-s/library/windows/hardware/ff548239(v=vs.85 aspx.

Kernel Patch Protection, 2016. Kernel Patch protection. https://en.wikipedia.org/wiki/Kernel_Patch_Protection.

Kirda, Engin, 2015. Detecting and analyzing kernel-based malware. https://securityintelligence.com/detecting-and-analyzing-kernel-based-malware/.

Kirk, Nathan, 2018. Bring your own land (BYOL) – A novel red teaming technique. https://www.mandiant.com/resources/bring-your-own-land-novel-red-teaming-technique.
Leemreize, T., 2021. Analyzing Fileless Malware for the .NET Framework through CLR Profiling. Master's thesis, University of Twente.
Levene, B., Falcone, R., Kazuar, T. Halfpop, 2017. Multiplatform espionage backdoor with API access. https://unit42.paloaltonetworks.com/unit42-kazuar-multi-platform-espionage-backdoor-api-access/.
Long, Joshua, 2018. Privacy Exodus: spam delivers Mac spyware. https://www.intego.com/mac-security-blog/privacy-exodus-spam-delivers-mac-spyware.
Macht, Holger, 2012. Dalvikvm support for volatility. http://lists.volatilesystems.com/pipermail/vol-dev/2012-October/000187.html.
Maggio, R., Case, A., Ali-Gombe, Aisha, Richard III., Golden G., 2021. Seance: divination of tool-breaking changes in forensically important binaries. Forensic Sci. Int.: Digit. Invest. 37.
Malwarebytes Lab, 2016. Unpacking yet another .NET crypter. https://blog.malwarebytes.com/threat-analysis/2016/07/unpacking-yet-another-net-crypter/.
Mandal, Debasish, 2013. Injecting shellcode into a portable executable(PE) using Python. http://www.debasish.in/2013/06/injecting-shellcode-into-portable.html.
Manna, M., Case, A., Ali-Gombe, Aisha, Richard III., Golden G., 2021. Modern macOS userland runtime analysis. Forensic Sci. Int.: Digit. Invest. 38.
Microsoft. .NET core and .NET 5+. https://docs.microsoft.com/en-us/dotnet/core/introduction#terminology, 2021.
Microsoft. .NET framework technologies unavailable on .NET core and .NET 5+. https://docs.microsoft.com/en-us/dotnet/core/porting/net-framework-tech-unavailable, 2021.
Microsoft. Antimalware scan interface. https://docs.microsoft.com/en-us/windows/win32/amsi/antimalware-scan-interface-portal, 2019.
Microsoft. Assemblies in .NET. https://docs.microsoft.com/en-us/dotnet/standard/assembly/, 2021.
Microsoft. Assembly.Load method. https://docs.microsoft.com/en-us/dotnet/api/system.reflection.assembly.load?view=net-6.0&viewFallbackFrom=net-6.0, 2021.
Microsoft. Clipboard.GetText method. https://docs.microsoft.com/en-us/dotnet/api/system.windows.forms.clipboard.gettext?view=windowsdesktop-6.0, 2021.
Microsoft. Lifecycle FAQ - .NET framework. https://docs.microsoft.com/en-us/lifecycle/faq/dotnet-framework, 2021.
Microsoft. Script tracing and logging. https://docs.microsoft.com/en-us/powershell/scripting/windows-powershell/wmf/whats-new/script-logging?view=powershell-7, 2021, 2.
MITRE. Cobalt Strike. https://attack.mitre.org/software/S0154/, 2021.
MITRE. Screen capture. https://attack.mitre.org/techniques/T1113/, 2017.
Molerats Malware, 2021. New TA402/MOLERATS malware – Decrypting .NET reactor strings. https://www.0ffset.net/reverse-engineering/malware-analysis/molerats-string-decryption/.
molerats. Molerats apt: new malware and techniques in middle east espionage campaign. https://www.cybereason.com/blog/molerats-apt-new-malware-and-techniques-in-middle-east-espionage-campaign, 2020.
Morphisec Labs, 2019. New campaign delivers Orcus RAT. https://blog.morphisec.com/new-campaign-delivering-orcus-rat.
Nafisi, Ramin, 2021. FoggyWeb: targeted NOBELIUM malware leads to persistent backdoor. https://www.microsoft.com/security/blog/2021/09/27/foggyweb-targeted-nobelium-malware-leads-to-persistent-backdoor/.
odzhan, 2019. How red teams bypass AMSI and WLDP for .NET dynamic code. https://modexp.wordpress.com/2019/06/03/disable-amsi-wldp-dotnet/.
Paganini, Pierluigi, 2018. Evrial: the latest malware that steals bitcoins using the clipboard. https://securityaffairs.co/wordpress/69587/breaking-news/evrial-malware-steals-bitcoin.html.
Pot, J., 2016. What Mac users need to know about EL Capitan security. http://www.makeuseof.com/tag/mac-security-el-captan-rootless/.
Reactor_download. .NET reactor. https://www.eziriz.com/reactor_download.htm, 2021.
s4tan. Analyzing the nasty .NET protection of the Ploutus.D malware. https://antonioparata.blogspot.com/2018/02/analyzing-nasty-net-protection-of.html, 2018.
SonofStrike. Sos.dll (sos debugging extension). https://docs.microsoft.com/en-us/dotnet/framework/tools/sos-dll-sos-debugging-extension, 2022.
Stokes, Phil, 2020. EvilQuest rolls ransomware, spyware & data theft into one. https://www.sentinelone.com/blog/evilquest-a-new-macos-malware-rolls-ransomware-spyware-and-data-theft-into-one.
The Volatility Framework, 2017. Volatile memory artifact extraction utility framework. https://github.com/volatilityfoundation/volatility.
Thompson, Jacob, 2020. Analyzing Dark crystal RAT, a C# backdoor. https://www.mandiant.com/resources/analyzing-dark-crystal-rat-backdoor.
Williams, Jake, 2021. What you need to know about the SolarWinds supply-Chain attack. https://www.sans.org/blog/what-you-need-to-know-about-the-solarwinds-supply-chain-attack/.
xpnweired, 2021. Weird ways to run unmanaged code in .net. https://blog.xpnsec.com/weird-ways-to-execute-dotnet.
Zhang, Xiaopeng, 2017. In-Depth Analysis of A New Variant of. NET Malware AgentTesla. https://www.fortinet.com/blog/threat-research/in-depth-analysis-of-net-malware-javaupdtr.