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

DFRWS is dedicated to the sharing of knowledge and ideas about digital forensics research. Ever since it organized the first open workshop devoted to digital forensics in 2001, DFRWS continues to bring academics and practitioners together in an informal environment.

As a non-profit, volunteer organization, DFRWS sponsors technical working groups, annual conferences and challenges to help drive the direction of research and development.

https://dfrws.org

Memory Analysis of .NET and .NET Core Applications

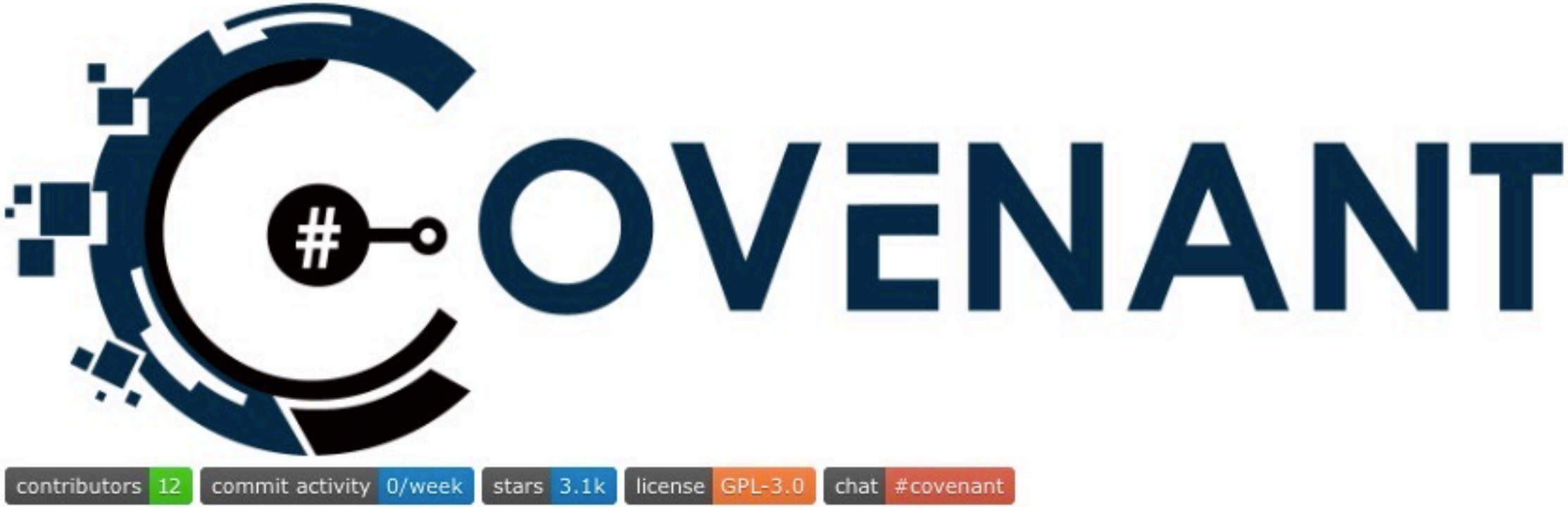
Modhuparna Manna, **Andrew Case**, Aisha Ali-Gombe, Golden G. Richard III

Motivation

- .NET-based malware is increasingly common in sophisticated, real-world attacks
- This shift occurred as Powershell became unusable in many scenarios
- .NET provides built-in support for memory-only loading, crypto routines, and full access to the Windows API
- Current memory forensic algorithms are very weak against detection of .NET-based malware

PowerShell Empire

Empire is a pure PowerShell post-exploitation agent built on cryptologically-secure communications and a flexible architecture. Empire implements the ability to run PowerShell agents without needing powershell.exe, rapidly deployable post-exploitation modules ranging from key loggers to Mimikatz, and adaptable communications to evade network detection, all wrapped up in a usability-focused framework.



Covenant is a .NET command and control framework that aims to highlight the attack surface of .NET, make the use of offensive .NET tradecraft easier, and serve as a collaborative command and control platform for red teamers.

Covenant is an ASP.NET Core, cross-platform application that includes a web-based interface that allows for multi-user collaboration.

Project Page: https://github.com/cobbr/Covenant

.NET Memory-Only Loading

Load(Byte[])

Loads the assembly with a common object file format (COFF)-based image containing an emitted assembly. The assembly is loaded into the application domain of the caller.

C# Copy

```
public static System.Reflection.Assembly Load (byte[] rawAssembly);
```

Parameters

rawAssembly Byte[]

A byte array that is a COFF-based image containing an emitted assembly.

Bypassing AMSI

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

- Recent Black Hat presentation on modern AMSI bypasses: https://www.blackhat.com/asia-22/briefings/schedule/#amsi-unchained-review-of-known-amsi-bypass-techniques-and-introducing-a-new-one-26120

Malicious Activity – Gathering Screenshots

CopyFromScreen(Int32, Int32, Int32, Int32, Size)

Performs a bit-block transfer of the color data, corresponding to a rectangle of pixels, from the screen to the drawing surface of the Graphics.

C# Copy

```
public void CopyFromScreen (int sourceX, int sourceY, int destinationX, int destinationY,
System.Drawing.Size blockRegionSize);
```

Malicious Activity - Microphone Recording

NAudio

NAudio is an open source .NET audio library written by Mark Heath

Malicious Activity – Keystroke Logging

```
[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);
```

Malicious Activity - WebCamera Recording

capCreateCaptureWindowA function (vfw.h)

Article • 10/05/2021 • 2 minutes to read

The **capCreateCaptureWindow** function creates a capture window.

Syntax

C++ Copy

```
HWND VFWAPI capCreateCaptureWindowA(
  LPCSTR lpszWindowName,
  DWORD  dwStyle,
  int    x,
  int    y,
  int    nWidth,
  int    nHeight,
  HWND   hwndParent,
  int    nID
);
```

.NET Framework vs .NET Core

- .NET Framework
  - Closed source*, Windows-only implementation
  - The main version used in production environments
  - Windows-only features, such as AMSI

- .Net Core
  - Open source, cross-platform implementation
  - Supports mobile platforms and is highly scalable
  - Microsoft claims it will replace .NET Framework (unlikely)

https://www.geeksforgeeks.org/differences-between-net-core-and-net-framework/

.NET Framework vs .NET Core Analysis Differences

- .NET Framework
  - PDB files only have a few symbols
  - *NO* data structure types
  - Requires reverse engineering per version to recover the offsets

- .NET Core
  - PDB files have full symbol set and type information
  - Symbol and type information can easily be integrated into Volatility 2 and automatically integrated into Volatility 3

Runtime Memory Analysis Goals

- Recover the set of load classes
- For each class, its fields and methods
- For each field, its name and type
- For each method, its definition and code location
- Instances of each class (objects)

Domains, Assemblies, and Modules

- Each application has at least one *application domain*
  - A container (process) inside the CLR, all the .NET code and data
  - Only one allowed in .NET Core

- Each *domain* has at least one *assembly*, which is a PE (.exe or .dll) file that contains managed code and resources

- *modules* are used to track each component of an assembly
  - .NET Core only allows one module per assembly

Covenant Memory Only Assemblies

```
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);
}
```

```
$ 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
```

Recovering Loaded Classes

- Each *module* has a a *MethodTable* that holds the information on contained classes, each of which is represented by the *EEClass* type

- ”Querying” the in-memory metadata database leads to recovery of the name and namespace of a class

```
$ python vol.py  -f data.lime --profile=Win10x64_19041 dotnet_classes
Volatility Foundation Volatility Framework 2.6
Class Address      Class Name
------------------ ----------
0x00007ffc64827950 System.AppContext
0x00007ffc64765d10 System.Array
0x00007ffc64892758 System.CLRConfig
<snip>
```

Recovering Fields

- Each class stores a list where each element is a field
- Fields have a *name, type,* and *offset*

```
$ 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
```

Recovering Methods – NDirect (native)

```
[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);
```

```
$ 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
```

Recovering Methods – .NET (IL)

- Methods written in .NET are not JIT'd until they are called
- We only analyze these functions as otherwise they have no x86 or x86-64 code present
- The code is statically analyzed for calls to suspicious functions

```
$ 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
```

Conclusions & Future Work

- Modern DFIR requires memory forensics to be effective and complete
- Our research effort has provided a strong set of support to routine integration of .NET analysis into memory forensics workflows
- Automation of type extraction needed for fully automated workflows

Questions? Comments?

- Contact
  andrew@dfir.org
  golden@cct.lsu.edu

- Social Media
  @volatility
  @attrc
  @nolaforensix

- 2022 Volatility Plugin Contest now open!
  https://volatility-labs.blogspot.com/2022/07/the-10th-annual-volatility-plugin-contest.html