Reflection

Reflection - Introduction

Enables the following:

1. Type introspection
2. Overriding member visibility - an extension to #1
3. Dynamic code invocation/generation - a.k.a. metaprogramming

Reflection - Type Introspection

Use cases:

1. You want to determine all .NET assemblies that reference System.Management.Automation.dll
2. You want to determine what classes and methods exist in an assembly
3. You are performing .NET malware analysis

Reflection - Overriding Member Visibility

Use cases:

1. Borrowing .NET code that isn’t publicly accessible
   - e.g. P/Invoke definitions
2. Editing internal properties/fields

Reflection - Overriding Member Visibility

Some clarifying terminology:

- **Type** - Essentially, a class. A type can have sub-types aka “nested types”.
- **Field** - A named value within a class
- **Property** - A special type of method that gets/sets a field.
- **Constructor** - a special type of method that help instantiate/initialize a class.
- **Member** - A catchall for all .NET “types” - e.g. types, events, interfaces, properties, fields, methods, etc.

Reflection - Overriding Member Visibility

- With access to the reflection API, absolutely any method, property or field is accessible within a given type (i.e. class) in PowerShell.
- Look at the Get* methods within a System.Type instance.
  - `[Object] | Get-Member -MemberType Method -Name Get*`
- Many Get* methods will require specifying the visibility/member type via System.Reflection.BindingFlags
  - `[Reflection.BindingFlags] | Get-Member -Static -MemberType Property`
- For example, specifying an internal, static member:
  - `[Reflection.BindingFlags] 'NonPublic, Static'`

Reflection - Overriding Member Visibility - Exercise

- Consider how you Base64 encode content - [Convert]::ToBase64String(byte[] inArray)
- How does .NET know to use the standard Base64 alphabet?
- What if, as an attacker, we wanted to alter the Base64 alphabet to subvert analysis?
- Maybe reflection can help us out…
- See how it could be done: Base64Hijack.ps1

Reflection - Dynamic Code Generation/Invocation

Use cases:

1. .NET assembly in-memory loading/execution
2. Dynamic .NET malware analysis
3. .NET malware repurposing
4. Wanting to avoid dropping unnecessary compilation disk artifacts

Exercise: Add-Type Artifacts

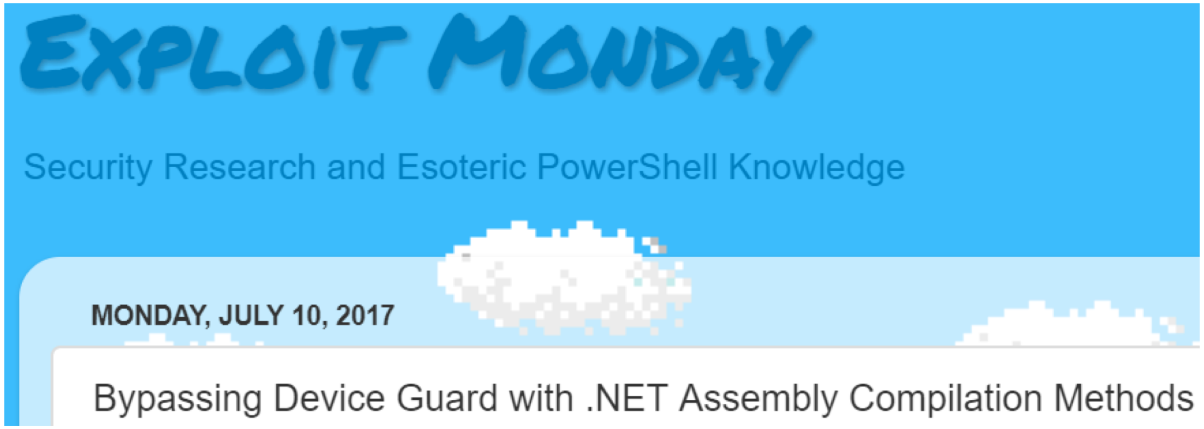
- Run the Add-Type invocation in **AddTypeArtifactLab.ps1** with procmon running.
- Identify command lines of any child processes.
- Identify any interesting files that are created.
- Bonus: Attempt to capture the files prior to being deleted.

Draw some conclusions:

- As a defender, what detections could be written. What mitigations?
- Is there any chance for false positives.

Add-Type Artifacts - Vulnerability

- You noticed that all the disk artifacts were written to a user-writeable directory, right?
- Race condition anyone?
- It’s not just Add-Type that’s vulnerable...



EXPLOIT MONDAY

Security Research and Esoteric PowerShell Knowledge

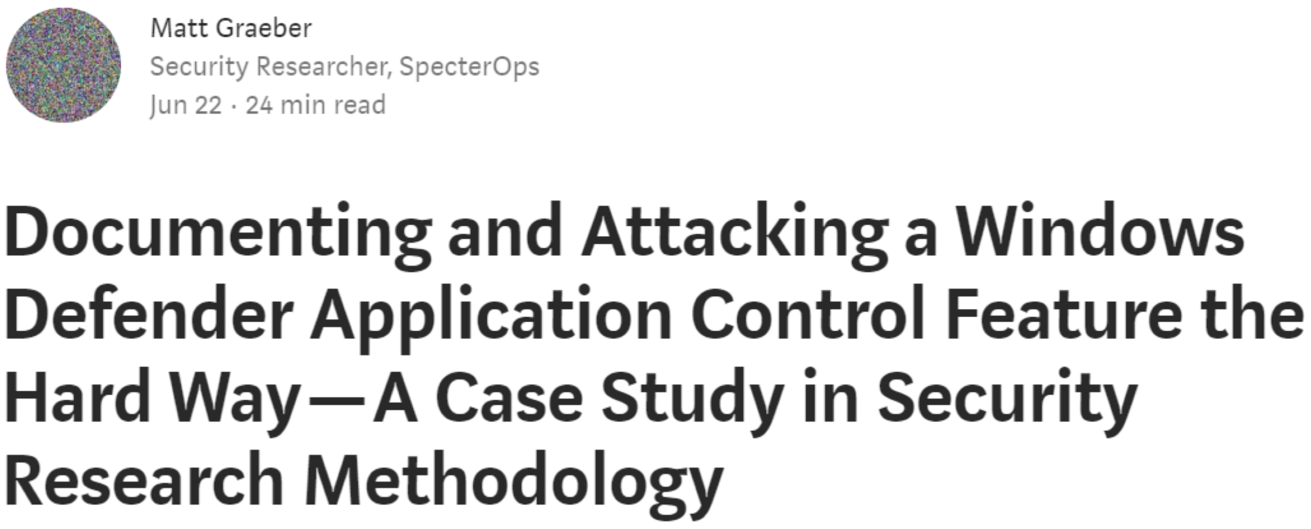
MONDAY, JULY 10, 2017

Bypassing Device Guard with .NET Assembly Compilation Methods

http://www.exploit-monday.com/2017/07/bypassing-device-guard-with-dotnet-methods.html

Add-Type Artifacts - Vulnerability Mitigation

- Fixed in the latest version of Windows Defender Application Control
  - “Dynamic Code Security” now a CI policy rule option



Matt Graeber
Security Researcher, SpecterOps
Jun 22 · 24 min read

Documenting and Attacking a Windows Defender Application Control Feature the Hard Way—A Case Study in Security Research Methodology

https://posts.specterops.io/documenting-and-attacking-a-windows-defender-application-control-feature-the-hard-way-a-case-73dd1e11be3a

Reflection - Type Retrieval

```
# Type retrieval standard method
[System.Diagnostics.ProcessStartInfo]

# Type retrieval reflection method
# Referencing a known public class from the same assembly.
# Note: the full class name must be specified
[System.Diagnostics.Process].Assembly.GetType('System.Diagnostics.ProcessStartInfo')
```

Reflection - Object Instantiation

```
# Standard
$ProcStartInfo = New-Object -TypeName System.Diagnostics.ProcessStartInfo -ArgumentList 'cmd.exe'

# Reflection method #1
$ProcStartInfo = [Activator]::CreateInstance([System.Diagnostics.ProcessStartInfo], [Object[]] @('cmd.exe'))

# Reflection method #2
$ProcessStartInfoStringConstructor = [System.Diagnostics.ProcessStartInfo].GetConstructor([Type[]] @([String]))
$ProcStartInfo = $ProcessStartInfoStringConstructor.Invoke([Object[]] @('cmd.exe'))
```

Reflection - Method Invocation

```
# Converting an Int32 to a hex string. Standard method.
(1094795585).ToString('X8')

# Reflection method
$IntToConvert = 1094795585
$ToStringMethod = [Int32].GetMethod('ToString',
[Reflection.BindingFlags] 'Public, Instance', $null, [Type[]] @([String]),
$null)
$ToStringMethod.Invoke($IntToConvert, [Object[]] @('X8'))
```

Reflection - Offensive Use Case

Goal: We would like to load and execute a .NET assembly in memory.

Let’s load a hello world program in memory and execute it.

```
Add-Type -TypeDefinition @'
using System;

public class MyClass {
              public static void Main(string[] args) {
        Console.WriteLine("Hello, world!");
              }
}
'@ -OutputAssembly HelloWorld.exe
```

Follow along with HelloWorldLoaders.ps1

Reflection - Offensive Use Case

Using System.Reflection.Assembly.Load to load the assembly in memory:

```
$AssemblyBytes = [IO.File]::ReadAllBytes("$PWD\HelloWorld.exe")
$HelloWorldAssembly = [System.Reflection.Assembly]::Load($AssemblyBytes)
# Invoking the public method using standard .NET syntax:
[MyClass]::Main(@())
# Using reflection to invoke the Main method:
$HelloWorldAssembly.EntryPoint.Invoke($null, [Object[]] @(@(,([String[]] @()))))
```

Exercise: Write a function that converts a file to a Base64 -encoded string and emits code to decode the string and call Assembly.Load.

One solution: AssemblyLoaderGenerator.ps1

Reflection - Offensive Use Case

Imagine a point in the future where calls to Assembly.Load are monitored/blocked. A realistic future, by the way. Pure reflection to the rescue!

Warning: This is an advanced concept with no generic solution for automatic reflection code generation!

Knowledge required: .NET internals and MSIL assembly

Additional requirements: Patience and curiosity

Reflection - Offensive Use Case

```
$Domain = [AppDomain]::CurrentDomain
$DynAssembly = New-Object System.Reflection.AssemblyName('HelloWorld')
$AssemblyBuilder = $Domain.DefineDynamicAssembly($DynAssembly, [Reflection.Emit.AssemblyBuilderAccess]::Run)
$ModuleBuilder = $AssemblyBuilder.DefineDynamicModule('HelloWorld.exe')
$TypeBuilder = $ModuleBuilder.DefineType('MyClass', [Reflection.TypeAttributes]::Public)
$MethodBuilder = $TypeBuilder.DefineMethod('Main', [Reflection.MethodAttributes] 'Public, Static', [Void], @([String[]]))
$Generator = $MethodBuilder.GetILGenerator()
$WriteLineMethod = [Console].GetMethod('WriteLine', [Type[]] @([String]))
# Recreate the MSIL from the disassembly listing.
$Generator.Emit([Reflection.Emit.OpCodes]::Ldstr, 'Hello, world!')
$Generator.Emit([Reflection.Emit.OpCodes]::Call, $WriteLineMethod)
$Generator.Emit([Reflection.Emit.OpCodes]::Ret)
$AssemblyBuilder.SetEntryPoint($MethodBuilder)
$TypeBuilder.CreateType()
[MyClass]::Main(@())
```

Reflection - Offensive Use Case

Conclusion:

At this point, it’s worth mentioning that PowerShell doesn’t have to be an end-all-be-all. As you’ve seen, PowerShell can just be a fantastic in-memory loader for a full-featured .NET implant! A minimal PowerShell loader is small enough to that it could be built to evade most/all PowerShell detections.

PowerShell could just be another implant loader option in the same way that something like msbuild.exe would be.

This would be a good time to attempt Lab: Malware Repurposing Lab