Preventing Reflective DLL Injection on UWP Apps

Mojtaba Zaheri[1], Salman Niksefat[2], and Babak Sadeghiyan[3]
[1,2,3]APA Research Center, Amirkabir University of Technology

Abstract **- Universal Windows Platform (UWP) is the Microsoft's recent platform-homogeneous application architecture. It al-lows a code to run on variety of devices including PC, mobile devices, etc., without needing to be rewritten or recompiled. UWP apps are becoming more and more popular and consequently this new application platform is becoming the next attack target for hackers and malware developers. In this paper, we first study the issue of host-based code injection attacks (HBCIA) in UWP apps. We show that de-spite the embedded mechanisms in UWP to maintain code integrity and to only allow legitimate DLLs to be loaded in memory, it is still possible to circumvent the defensive mechanisms and launch a variant of HBCIA called Reflective DLL Injection on UWP apps. We then propose a novel defence mechanism against reflective DLL injection attacks on UWP apps. Our proposed method can detect malicious/benign injection attempts on UWP apps and prevents malicious injections while allowing the benign injections to proceed as normal. Our experiments show that the proposed defence has less than 1% impact on system's overall performance and can be used inside anti-virus (AV) products to strengthen their protection capabilities.**

KEYWORDS – DLL Injection, Universal Windows Platform, UWP

I. INTRODUCTION

Universal Windows Platform (UWP), first introduced in Windows 10, is the Microsoft's platform homogeneous application architecture. Its purpose is to allow development of universal applications that run on a variety of platforms including PC, mobile devices, and IoT devices. This relieves the code from the need to be rewritten or recompiled for each platform. Similar to Android and IOS, this platform has its own proprietary software store through which Microsoft can have more control over the distributed UWP applications. Since its release, Microsoft has encouraged the software developers to write code in UWP and the company itself included some UWP applications in Windows 10, including Microsoft Edge browser and Microsoft Groove Music.

With rapid popularity of UWP applications among soft-ware developers and considering the strong support of Microsoft, UWP apps are becoming more and more popular among end users and consequently this new platform has become the next attack target for hackers and malware developers. One important category of intra-host attacks that can potentially target UWP applications is Host Based Code Injection Attacks (HBCIA). HBCIA is defined as locally copying a code from a malicious source process into the address space of a target process and executing the code [1]. A recent research in [1] shows that near 64% of the total of 162850 sample malware use HBCIA as part of their malicious behaviour.

One strong motivation for using HBCIA by malware is to evade detection and bypass host-based firewalls: Mal-ware usually connect to their C&C [1] servers for sending in-formation and receiving new commands. Thus host-based firewalls are generally sensitive to outgoing connections of locally running applications and they have rules to prevent unknown applications from accessing the network. To prevent being caught by the firewalls, new malware generally uses smart techniques for connecting to the Internet. One such technique is taking advantage of HBCIAs, i.e., injecting a software module to another legitimate running process such as Mozilla Firefox, Internet Explorer or Google Chrome and communicating using the injected module. Among these browsers, Microsoft Internet Explorer has been more promising for hackers as it is generally available in Windows family of operating systems by default. Moreover, Microsoft-Edge, the Microsoft's new UWP-based browser introduced in Windows 10, can be the next injection target for malware that are willing to launch a code-injection attack.

In this paper, we demonstrate that it is still possible to launch successful DLL injection attacks by a technique called reflective DLL injection [2][3] despite the new security

mechanisms embedded in UWP framework to maintain code integrity and prevent un-signed/malicious DLL injections. Then, we propose a defence mechanism against such attacks on UWP apps.

Some currently published methods such as [4][5] try to parse the victim process memory and find if a malicious DLL is loaded into the process memory. Then, they try to remove it and clean the memory. How-ever, it's not a sound and complete countermeasure, as the malware is already loaded in the memory and can do its malicious activities before being removed from the memory. In contrast, in our proposed mitigation, we try to prevent the malware to load the malicious DLL from the very beginning. Another challenge in countering such attacks is that not all code injections are malicious. The operating system may inject some legitimate DLLs into processes. Moreover, processes may inject code into their own address space for purposes like loading plug-ins, etc. Therefore, we need a method to distinguish between malicious and benign injections. Our proposed defence mechanism does this with high precision. In case of a malicious injection, it successfully prevents the DLL to be written into the target process and raises an alarm. On the other hand, in case of a benign injection, the injection proceeds as normal. Finally, by taking advantage of PCMark benchmarking tool, we show that our proposed technique imposes a little overhead on operating system.

To summarize, our contribution in this paper is a mitigation technique against reflective DLL injection on UWP apps that provides the following original advantages:

i. It entirely prevents a malware from loading its malicious module into the target process memory.
ii. The proposed mechanism is very efficient as it only monitors and modifies the behaviour of one API (NtWriteVirtualMemory), which leads to a very low overhead on the system performance.
iii. It doesn't have any effects on normal DLL injections, as it's possible to load legitimate/signed DLLs into target UWP apps through calling the LoadLibrary API.

This paper is organized as follows: In section II, we re-view related work including HBCIA methods and the existing countermeasures. In section III, we review the security mechanisms embedded in UWP apps that are related to HBCIA attacks. In section IV, we demonstrate the methods that can circumvent the integrity mechanisms of UWP and perform the reflective DLL injection attack in UWP frame-work. In section V, we present our defensive mechanism to reflective DLL injection attacks. In section VI, we present the results of the evaluation of the proposed system. Finally, section VII concludes the paper.

II. RELATED WORK

The works in host-based code injection attacks can be classified into methods for performing such attacks, and mechanisms for detection and prevention. In this section, we review these works and considering the detection and prevention mechanisms, we claim that none of them is suitable for defending against reflective DLL injection on UWP apps.

A. Performing HBCIA

Since these methods have rather a technical nature, the concept has received much more attention in the technical forums rather than the research papers.

In [1], the authors have presented a semi-formal definition for host-based code injection attacks that we cited in the introduction. The paper has presented the basic idea of the technique in three main steps including I) Victim process selection, II) Code copying, and III) Code execution. This paper also mentions several motivations behind using HBCIAs including interception of critical information, privilege escalation, and detection avoidance.

In [6], a classification on various DLL Injection techniques is presented. This paper classifies these techniques as follows: CreateRemoteThread [7], Creating a Proxy DLL [8], Modification of Windows Registry [9], Windows Hooks [10][11], Using a Debugger [12], Patching the IAT [13] and Reflective Injection [2][14].

Most of the above techniques can't be used to inject into UWP apps because the LoadLibrary API has been limited by UWP

framework code integrity mechanism. However, a specific type of DLL Injection called Reflective Injection which was introduced in [2] can be used to circumvent this mechanism. This method can load a DLL on UWP apps through the concept of reflective programming without directly using LoadLibrary API. In section 4, further details about this technique is presented.

B. Detecting and Preventing HBCIA

Since the HBCIAs need to have local access to the tar-get system, these types of attacks had not been considered very hazardous in the past. However, the advances in HB-CIA techniques and ever-increasing number of malwares in recent years have motivated the security researchers to work on mitigation mechanisms for these attacks. In the following, we review some of these methods.

In [1], a mechanism named BeeMaster is proposed to prevent host-based code injections through using honeypot paradigm. In this mechanism, a master bee and multiple worker bees are used. The master bee creates and instruments the workers to find if a code injection is occurred. If so, the master bee creates a memory dump and terminates the worker bee. The downside of this mechanism is that the detection only works on the processes that are created by the master bee, and therefore it cannot detect the targeted injections that occur on other processes of the system.

[15] aims to detect malicious DLL injections by evaluating the injected DLLs through the information provided by the process snapshots. For this purpose, it checks some common malicious DLL characteristics in the loaded DLLs to find a match. Nevertheless, one of the drawbacks of this technique is that it cannot detect the attack before the injection, so it cannot prevent the malicious DLL from being loaded. In [16] a similar technique is opted for to detect malicious DLLs through their characteristics by using machine learning methods and has the similar defects of [17].

Some of the code injection methods mentioned in section 2.1 are useful for detection and prevention purposes. For instance, in [18] a mechanism called DLL Preemptive Injection is used that whenever the system is loading the UrlMon DLL to a process, it interrupts the process and loads a monitoring module that later checks the API call patterns in the target process to see if its behavior is malicious. However, the proposed method is only effective against Trojan downloaders.

Also, Detecting the Code Injection Engine (DCIE) [19] tries to reject all the suspicious thread creating calls by hooking APIs and tracing three main steps of code injection attacks: allocating memory, writing to the memory, and creating the thread. Although this method prevents the injection attacks, it has two major weaknesses;

i. It rules out injection of legitimate and signed DLLs, and
ii. It hooks three APIs, which decreases system's performance.

In case of reflective injection, the articles [20][21] propose ideas to check the memory of running processes periodically and search to find if there is any malicious content, and then they try to delete the infected memory pages, change their permissions, or even kill the infected process. However, during the time span between the two checks the malware can harm the system.

In comparison with previous methods, in this paper we propose a countermeasure against reflective DLL Injection on UWP apps, which is very effective, hooks only one API so it does not depend on API succession and has a very low impact on the system's performance. Moreover, through its combination with UWP Binary mitigation mechanism, it still lets legitimate DLLs to be loaded without any limitation. In section V5, our proposed countermeasure is presented.

III. UWP SECURITY MECHANISMS

Before addressing the issue of code injection attacks on UWP apps, we should first review several security mechanisms embedded in UWP framework to prevent the classic injection attacks to happen. Microsoft's attitude toward UWP is not only a better user experience but a more secure environment for application development that makes it harder for malware to penetrate UWP-based devices. Two important security mechanisms in UWP are "App Container" and "Code Integrity Enforcement" which are

directly related to HBCIA attacks. We review these mechanisms in this section.

A. App Container

UWP framework is equipped with a new security sandbox called App Container which provides more fine-grained per-mission assignments and limits unauthorized read and write operations throughout the system. App Container helps to make sure that an UWP app is only restricted to its defined security permissions. In the following, we review a number of App Container capabilities.

Limit access to files and peripherals. UWP apps are restricted to access directly to only two directories: the app's WindowsApps directory in Program Files, and the app's package directory located in AppData. The full path to the WindowsApps is [Win_Drive]:\Program Files\WindowsApps.

All files stored by apps in WindowsApps have to be static files that don't change through the app's lifetime. To enforce this rule, files stored by applications in this directory go through integrity checks before the app is launched. If a file in this directory is modified, the app will fail those integrity checks and refuse to launch. Also, the app's local AppData directory is located in [Win_Drive]:\Users\[UserName]\AppData\Local\Packages.

This directory is meant to be a place for apps to store dynamic files that can change over the time. As such, files in this directory don't go through integrity checks because it is meant to be a place for apps to store cache files, settings files, save files, and more.

Integrity Levels. App Container is implemented using the concept of Integrity Levels. Considering the definition in Microsoft's MSDN (Microsoft, n.d.-c), the Level has one of labels as System, High, Medium, Low, Untrusted.

This notion has been introduced in Windows Vista and is attributed to processes and objects. This mechanism prevents low level processes from reading or modifying high level processes and objects.

In Windows 8, Integrity Levels have been combined with the App Container, and limit processes to only read and write in their restricted area. This concept helps to ensure that the program does not have any access to the areas that are out of its range, unless the access is explicitly granted. To address this issue, every app container is assigned with a SID[2], and like users, the programs that are running in app containers.

Security Identifier can be part of Built-In groups, and consequently, have access to specific resources on the system. The associated name for these App Container Built-In groups is "Capabilities".

Specifically, in case of DLL loading, it's worth mentioning that all DLLs must have the read/execute per-missions of SID "S-1-15-2-1" which is equivalent ID for ALL_APPLICATION_PACKAGES, in DLL's Access Control List (ACL) (VoxelBlock, 2016).

B. Code Integrity Enforcement

Another important security mechanism in UWP apps is the Code Integrity Enforcement [22]. This mechanism is applicable in both process and kernel levels. The process-level enforcement is useful until the time the process is not compromised because the code integrity check can be disabled in a hacked process by the malware. Therefore, Microsoft has implemented the enforcement in the kernel-level to strengthen it against hacked processes and to prevent mal-ware from disabling this mechanism.

This mechanism activates during the LoadLibrary() API call. When a binary is going to be loaded, the kernel calls NtCreateSection() and then MiCreateSection() APIs. This last API finally invokes MiValidateSectionCreate() API which uses ci.dll (Code Integrity) to check the file signatures. If the verification does not match the defined policy, the kernel won't create the section and will return an error. The mitigation is performed by the kernel, so to turn off the mitigation, the intruder must have the kernel-level (ring 0) privilege [23].

The integrity check policies are defined in a structure called Process Signature Policy in "WinNT.h" (Microsoft, n.d.-a). Using the latest Windows SDK, one can see this structure as shown below:

```
typedef struct
_PROCESS_MITIGATION_BINARY_SIGNATURE_
POLICY
{
    union {
        DWORD Flags;
        struct {
        DWORD   MicrosoftSignedOnly : 1;
        DWORD   StoreSignedOnly : 1;
        DWORD   MitigationOptIn     : 1;
        DWORD   ReservedFlags :     29;
```

```
        }
    DUMMYSTRU
    CTNAME;     }
    DUMMYUNIO
    NNAME;
}
PROCESS_MITIGATION_BINARY_SIGNAT
URE_POLICY,
*PPROCESS_MITIGATION_BINARY_SIGN
ATURE_POLICY;
```

The flags specified in the structure enforce integrity restrictions. MicrosoftSignedOnly can be set to prevent the process from loading images that are not signed by Microsoft. StoreSignedOnly can be set to prevent the process from loading images that are not signed by the Windows Store and finally MitigationOptIn can be set to prevent the process from loading images that are not signed by Microsoft, the Windows Store and the Windows Hardware Quality Labs (WHQL).

All in all, the above integrity mechanism makes loading an unsigned DLL using LoadLibrary API impossible. Nevertheless, in the next section we review a number of recent techniques that allow intruders to circumvent this mitigation and load arbitrary DLLs into the memory of UWP apps even in the presence of an anti-virus.

IV. HBCIAS ON UWP APPS

One way to perform a host-based code injection attack is to put the code inside a DLL file and inject the DLL to the target process. This is called DLL injection. A classic DLL injection attack in Windows operating system is usually carried out by the following steps [7]:

i. Obtaining a handle to the victim process through calling OpenProcess API by setting the process's ID as the input parameter of this API.
ii. Allocating space inside the target process, by invoking VirtualAllocEx API.
iii. Writing malicious DLL's path into the allocated memory space, by using WriteProcessMemory API.
iv. Obtaining a handle of Kernel32.dll module by calling GetModuleHandle API.
v. Obtaining the address of LoadLibrary API through using GetProcAddress API, with Kernel32.dll's handle and LoadLibrary's name as the input parameters.
vi. Calling LoadLibrary API by one of thread creating APIs like CreateRemoteThread, RtlCreateUserThread, and NtCreateThreadEx, by using handle of the target process, address of LoadLibrary API, and written memory address of the DLL path as input parameters to accomplish the attack.

Due to the new code integrity security mechanism avail-able for UWP apps, it is possible to only allow signed DLLs to be loaded this way [22]. Thus, attackers must not be able to inject arbitrary DLLs on a target process that is being protected by the code integrity mechanism.

However, Microsoft's code integrity mechanism only triggers on the LoadLibrary API call, it is still possible to inject binary shellcodes into the target process as stated in [23]. However, working with shellcodes is very di cult and the attacker has to handle many complexities. Hence, attackers are still looking for methods that de-spite the existence of Microsoft's binary mitigation mechanism, inject their arbitrary DLLs to the memory of processes. A little surfing of the security and hacking technical forums reveals that it is possible to use a tiny bootstrap shell-code to perform a so-called Reflective DLL Injection [2][3] and load an arbitrary DLL into a target process without directly using the LoadLibrary API call. However, the reflective DLL injection technique has been proposed for classic Windows applications and their use against UWP apps is not yet documented in academic papers or technical forums. We confirmed that this technique works successfully against UWP apps too by injecting an arbitrary DLL into the Microsoft Edge browser's memory. The details for the reflective DLL injection attack elaborate in the next section.

A. Reflective DLL Injection

Assuming the attacker has code execution capability in the target process and the whole content of the library (s)he wishes to inject has been written into an arbitrary location of

memory in the target process, Reflective DLL Injection [2][3] works as follows:

i. Execution is passed via a tiny bootstrap shellcode to the library's ReflectiveLoader function which is an exported function found in the library's export table.
ii. Since the library's image currently exists in an arbitrary location in memory, the ReflectiveLoader first calculates its own image's current location in memory so as to be able to parse its own headers for use later on.
iii. The ReflectiveLoader will next parse the processes kernel32.dll export table in order to calculate the addresses of three functions required by the loader, namely Load-LibraryA, GetProcAddress and VirtualAlloc.
iv. The ReflectiveLoader will then allocate a continuous region of memory into which it will proceed to load its own image. The location is not important as the loader will correctly relocate the image later on.
v. The library's headers and sections are loaded into their new locations in memory.
vi. The ReflectiveLoader will then process the newly loaded copy of its image's import table, loading any additional library's and resolving their respective imported function addresses.
vii. The ReflectiveLoader will then process the newly loaded copy of its image's relocation table.
viii. The ReflectiveLoader will then call its newly loaded image's entry point function, DllMain with DLL_PROCESS_ATTACH. The library has now been successfully loaded into memory.
ix. Finally, the ReflectiveLoader will return execution to the initial bootstrap shellcode which called it.

Since the technique doesn't need a direct call to LoadLibrary, the security mechanism embedded in UWP apps is not able to prevent loading of the DLL. In the next section, we propose our mitigation mechanism to prevent this type of attack.

V. THE PROPOSED DEFENSE

In section 4, we discussed that despite the embedded mechanism in UWP framework against code injection at-tacks [22], it is still possible to bypass protection and inject arbitrary DLLs in UWP apps. We explained that the reflective DLL injection can be used to inject a DLL into UWP apps (e.g. Microsoft Edge browser) without direct call to the Loadlibrary API. In this section we propose a technique for defending against code injection attacks in UWP apps. The general idea for the defence is to precisely monitor an API call that is commonly used in reflective DLL injection attacks. More specifically, our idea is to monitor the input parameters to NtWriteVirtualMemory() API, which is used to write into the memory of a target process, and only allow valid parameters to get into.

To implement this, we use a hooking library to build a hooking DLL that hooks into all user-mode processes by means of a system-wide Kernel-mode injection driver. Since Microsoft strictly forbids patching or hooking in the driver land, we implemented the hooking in user-level, and made it system-wide by a driver that does the DLL injection in the kernel-level.

A. Preliminaries

Before presenting the proposed defence mechanism, we should first discuss some preliminaries about the underlying Windows internals that are used to build our mitigation engine.

User-Mode API Hooking is a technique by which developers can instrument and modify the behavior of API calls, for different purposes like monitoring programs' behavior, forcing them to function in a different way, etc. Hooks are widely used by anti-viruses, security applications, system utilities, programming tools, and so on. There are multiple hooking libraries such as Microsoft Detours [24], Mhook [25], Deviare [26], EasyHook [27], and others that can provide the user mode hooking capabilities. Their typical function is as follows:

i. Storing beginning bytes of the original code of the tar-get function somewhere else. It is needed for the correct behavior of the hooked function.

ii. Overwriting the beginning bytes of the target function with a custom code (called trampoline). So, when the function executes, it jumps to the hook handler.
iii. If needed, calling the stored original target function, at the end of the hook handler.

In this paper, we use Mhook [25] which is an open-source library and supports API hooking in both 32- and 64-bit programs. Microsoft also has introduced kernel-mode callbacks with Windows Vista. These callbacks are registered in kernel mode and provide notifications to the registrar upon a certain event (e.g. if you register a callback for a specific activity then you can have your callback function invoked before/after the action has occurred on the system). Three important callbacks for AV products are triggered for Create Process, Create Thread, and Load Image events. These callbacks are registered by invoking:

i. PsSetCreateProcessNotifyRoutine
ii. PsSetCreateThreadNotifyRoutine, and
iii. PsSetLoadImageNotifyRoutine.

Our proposed mitigation mechanism which is written in a hooking DLL is deployed system-wide using the kernel-level injection in a LoadImage callback routine.

B. The Mitigation Engine

In this section we explain the proposed technique that mitigates the code injection attacks by monitoring the calls to NtWriteVirtualMemory API. Figures 1 and 2 depict reflective versus normal DLL injections while our proposed defence mechanism in action. Our proposed mechanism consists three main steps:

Determining if the Binary Mitigation is enforced in the target process: The proposed countermeasure aims to prevent the malware from circumventing the binary mitigation mechanism. In fact, we want to tighten up the mitigation currently enforced in UWP apps, and consequently the proposed mechanism should only be activated for UWP binaries that are already protected by Windows mitigation policy. In other words, if the binary mitigation is not active, the attacker can use the LoadLibrary API directly to load its malicious DLL in to the target process.

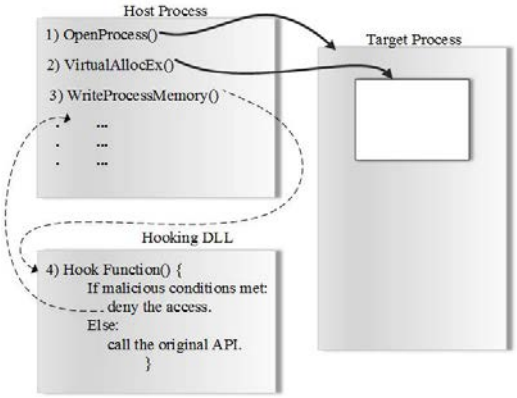

Figure 1: Proposed defence in action while a malicious reflective DLL Injection is being launched

For this purpose, we check the Signed Only flags in PROCESS_MITIGATION_ BINARY_SIGNATURE_POLICY structure to find if the target app is forced to load only signed DLLs. This information is provided by "WinNT.h" in the Windows SDK version 10.0.14393.0, and can be accessed by calling GetProcessMitigationPolicy() API with ProcessSignaturePolicy type, and ProcessHandle structure passed to NtWriteVirtualMemory API, as inputs. This way, the mechanism neglects the injections to other windows applications, just like the way the Windows 10 itself does.

Detecting inter-process writes. It's possible for applications to write into their own address space using NtWriteVirtualMemmory API call, which is apparently a non-malicious act. Therefore, we consider these intra-process writes as safe injections and continue to check whether we detect a NtWriteVirtualMemory call in which the process IDs of the caller and the target process are different. Since the NtWriteVirtualMemory API is invoked in the source process, the hook function is also executed in the con-text of this process and we can get the process ID of this process by calling GetCurrentProcessId API. The process ID of the target process can also be obtained from the ProcessHandle structure passed into the NtWriteVirtualMemory API. The GetProcessId(ProcessHandle) can obtain this data for us. If these two process IDs are equal, we consider it as a legitimate intra-process injection, and call the original NtWrite-VirtualMemory API without any modification. Otherwise we go to the next step for further checking.

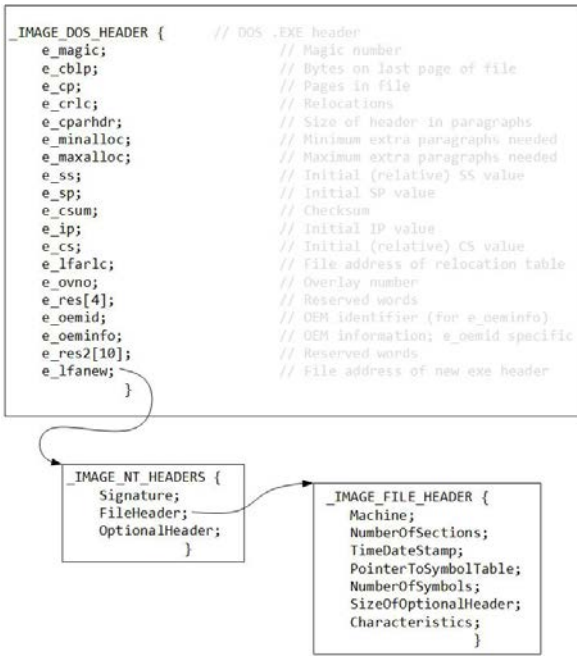

Figure 2: Proposed defence in action while a benign normal DLL Injection is being launched

Preventing the call if the input includes a DLL. The main difference between the reflective injection and normal DLL injection is that instead of writing the path of the desired DLL, it directly writes the DLL content into the target process memory, and consequently makes it possible to circumvent the Microsoft Mitigation Policy. So, we can utilize this fact, and prevent the write operation if the writing content contains a DLL. To check this please note that all Windows executables begin with a MS-DOS executable stub. So, we first check if a MS-DOS program header exists at the beginning of the injected data. We then check for markers for a Windows executable. If we learned that the writing content is a Windows executable, we look for information that deter-mines whether the file is an application or is a DLL. So, we check the following conditions respectively:

i. We check the first bytes of data for a valid DOS header. To do this we check the DOS header size field which should be 64 bytes at minimum.
ii. All DOS program files (and therefore Windows executables) begin with a magic number; the word value $5A4D ("MZ" in ASCII). So, we check if e_magic field of DOS header is equal to $5A4D.
iii. The Windows NT header begins with a magic number word whose value indicates whether this is a NE[3] format or PE[4] format executable or a virtual device driver with LE[5] format. The word is $454E ("NE" in ASCII), $4550 ("PE") or $454C ("LE"). So, we check if the Signature field of NT header is equal to $4550.
iv. Windows executables have a file header immediately following the $4550 magic number. This header structure has a Characteristics field which is a bit mask. If the bit mask contains the flag IMAGE_FILE_DLL then the file is a DLL, otherwise it is a program file.

Figure 3 illustrates the important structures in "WinNT.h" header file of Windows Kits version 10, considered in the pro-posed mitigation. If all the conditions are met, the mitigation engine considers the API call as malicious, aborts the call, and raises an alarm.

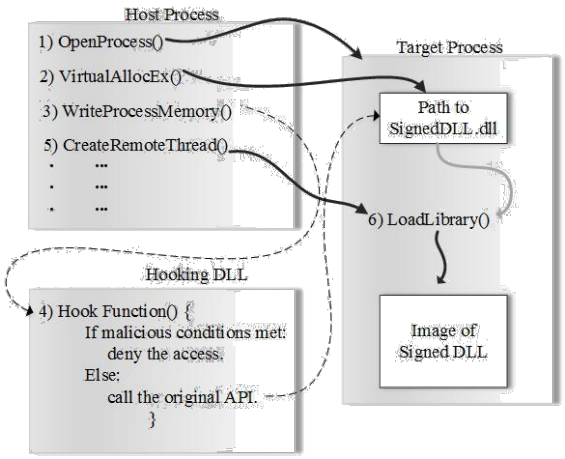

Figure 3: Structures in a Windows executable file

Since benign injections can be done in the normal way by writing only the path of the DLL into target process, there is no need to write the executable content directly, so the mitigation has no side effects on these benign injections.

C. System-Wide DLL Injection

We need a mechanism to load our mitigation engine DLL into all running processes upon their execution. To do this we have taken advantage of a system-wide DLL loading technique. A common method for system-wide DLL loading is the AppInit_DLLs infrastructure [9]. This mechanism loads an arbitrary list of DLLs in user-mode processes immediately after loading User32 DLL. However, it is not

enough as it does not load the DLLs in processes that don't load User32.dll. Like modern anti-virus products, we have written a kernel driver to implement an AppInit_DLLs-like infrastructure that loads our mitigation DLL immediately after loading Ntdll module instead of User32. This way, we will be ensured that the DLL is loaded in all windows processes, and the Mitigation is enforced system-wide. As mentioned earlier, the PsSetLoadImageNotifyRoutine is used to register a callback for Image Load events. This routine has the following signature:

```
NTSTATUS PsSetLoadImageNotifyRoutine(
        _In_PLOAD_IMAGE_NOTIFY_ROUTINE
        NotifyRoutine
);
```

After setting this routine, whenever an Image Load event occurs our defined NotifyRoutine will be run with PUNICODE_STRING FullImageName, HANDLE Proces-sId, PIMAGE_INFO ImageInfo, and BOOLEAN Create as input parameters. Our NotifyRoutine does the system-wide DLL injection in five steps:

i. Check if the loading image is Ntdll. Ntdll is the first DLL that will be automatically loaded for every process on the system, and also contains the target API for hooking in our Mitigation DLL, the NtWriteVirtualMemory API.
ii. Find the address of LdrLoadDll. Another reason to wait for Ntdll to be loaded is because we can parse the PE headers and find out the user mode address of LdrLoadDll. As explained in section 4, in user mode DLL injection, the LoadLibrary API is used for DLL loading, which is part of Kernel32 DLL. This API finally calls LdrLoadDll after some initializations. Thus, as we want to load our Mitigation DLL before loading Kernel32 DLL, we need to do the initialization in the callback routine and call the LdrLoadDll directly from Ntdll.
iii. Prepare an assembly code to load the Mitigation DLL through LdrLoadDll call into target process. Since we are working on x64 Windows, we need to write two different x64 and x86 assembly codes, to call the LdrLoadDll with the name of the proper version of Mitigation DLL as input, into target 64- and 32-bit processes. Also, two distinct versions of Mitigation DLL are placed in following directories:

    64 bit : [Win_Drive]:\Windows\System32

    32 bit : [Win_Drive]:\Windows\SysWOW64

iv. Allocate memory into the target process and write the assembly code there. Since the callback is called in the context of the target process, we can simply use NtCurrentPro-cess() to specify what process the memory will be allocated and written into.
v. Prepare an APC[6] to call the assembly code. APCs al-low user programs and system components to execute code in the context of a particular thread and, therefore, within the address space of a particular process. One advantage of APC is that it runs the code in the context of an existing thread and does not need to create a new thread for its operations, so makes it suitable for the case of system wide injection, as we need to load our DLL into all processes with no impact on performance. Following steps are required to add the code in the Thread APC Queue:

    Find a thread in the target process

    KeInitializeApc

    KeInsertQueueApc

Then, the Mitigation DLL will be loaded into the process when the APC runs the assembly code. Finally, we have a mechanism like the AppInit_DLLs infrastructure that can load our Mitigation DLL in all processes immediately after loading the Ntdll. Our implementation codes for Mitigation Engine and System Wide Injection Driver are available in Github [28].

VI. EVALUATION

To evaluate the proposed mitigation and assess its efficiency, we first used PCMark benchmarking tool [29] to measure the impact of the new technique on the overall performance of the system. PCMark is one of a series of Windows performance testing tools that are provided by the Futuremark. It includes a variety of bench-mark tests reflecting the different ways people use their computers. Each benchmark produces detailed results for gaining a deep understanding of performance during each

individual workload. The technical guide in [29] explains specific tests the tool conducts on systems, and the formulas it uses to produce the scores.

We used a virtual machine with the following specifications in our experiments which is Windows 10 x64 Enterprise Build 14393 as Operating System, Intel Xeon X5670 @ 2.93 GHz @ 2933 MHz as CPU, 1 Core(s), 1 Logical Processor(s) and 8.00 GB Memory.

We selected 5 common benchmarks and measured the system performance while our mitigation engine is on or off. Based on the results provided in Table 1, the overall performance degradation is at most 0.59 percent which is very small and negligible. In fact, the mitigation DLL does not have significant influence on typical user activities like web browsing, text writing, video chat and others. Since the technique only checks the NtWriteVirtualMemory API calls for inter-process writes into UWP apps and this event is not very common in ordinary usages of the system, it doesn't have tangible impact on the system's usual functionalities and performance.

Next, we assess the proposed countermeasure's impact on NtWriteVirtualMemory which the specific API is involved in the mitigation. To do so, we called the API to write a 100 KB memory block into a target process for 10000 times and calculated the average time. The detailed results are provided in Table 2. Whenever a DLL is being written into an UWP process memory, the write operation will be aborted, and the user will be informed about the malicious activity, so the first row of the table is not a usual write operation and its overhead doesn't have any impact on the system's performance. If the target of the write operation is a non-UWP process, the mitigation will be stopped in the first step, and based on the results of the second and fourth rows of the table, its overhead impact is around 4.6 percent. However, if the writing content is not a DLL, and the target process is UWP, the mitigation mechanism will be stopped in the third step and will have an overhead around 6.5 %. However, since it doesn't occur commonly in the system, it doesn't have a tangible impact on the system overall performance, as shown in Table 1.

Finally, to assess the number of NtWriteVirtualMemory API calls during execution of common Windows programs, we took advantage of API Monitor program [30] to illustrate the fact that the NtWriteVirtualMemory API call is not frequently used in prevalent Windows programs. API Monitor is a free monitoring tool that lets us monitor and control API calls made by applications and services. We selected a set of Windows programs, and ran each program for five minutes, to check call frequency of NtWriteVirtualMemory API. As illustrated in Table 3, call frequency of the API is at most 0.0002% in Google Chrome application.

Table 1: Overall Performance Impact on System (Time-Based).

Benchmark	Normal	Hooked	Overhead %
Web Browsing - JunglePin	0.373 s	0.375 s	0.54
Web Browsing - Amazonia	0.141 s	0.141 s	0.0
Writing	6.31 s	6.31 s	0.0
Phone Editing v2	1.867 s	1.878 s	0.59
Video Chat v2/Video Chat Encoding v2	704.7 ms	706.5 ms	0.26

Table 2: Performance Impact on NtWriteVirtualMemory API.

Content is DLL	Target is UWP	Normal ms	Hooked ms	Overhead %
✓	✓	0.0481	1.0270	2035.14
✓	×	0.0482	0.0504	4.56
×	✓	0.0480	0.0511	6.46
×	×	0.0481	0.0503	4.57

Table 3: NtWriteVirtualMemory Call Frequency in Windows Programs

Program	NtWriteVirtualMemory Call	Total Number of Call
Vmware Workstation	1	2330721
Telegram	0	2495273
Twitter	0	1658813
Spark Instant Messenger	0	5255403
Notpad++	0	1317342
Windows Media Player	1	12404209
VLC Media Player	0	11506828
TeamViewer	0	6176240
Mozilla Firefox	0	15501030
Google Chrome	23	10272584
Microsoft Edge	0	2464095
Wireshark Network Analyzer	5	2843753
Internet Download Manager	0	4944558

VII. CONCLUSION

In this paper, we studied the issue of reflective DLL injection attacks on UWP apps and proposed a defence mechanism to counter such attacks. We discovered that despite the embedded security mechanism in UWP framework, it is still possible to inject malicious/unsigned DLLs into UWP apps even in the presence of an antivirus software. To defend against these attacks, we proposed a mechanism that monitors the input parameters to NtWriteVirtualMemory() API and aborts malicious DLL injection attacks. We implemented the proposed idea by leveraging the hooking libraries and Windows kernel callbacks. This allows us to monitor the processes and prevent malicious injections into UWP apps while allowing the benign injections to proceed as normal.

VIII. REFERENCES

[1] Barabosch, T., Eschweiler, S., & Gerhards-Padilla, E. (2014). Bee master: Detecting host-based code injection attacks [Conference Proceedings]. In International conference on detection of intrusions and malware, and vulnerability assessment (p. 235-254). Springer.

[2] Fewer, S. (2008). Reflective DLL injection [Journal Article]. Harmony Security, Version, 1.

[3] Staples, D. (2015). Improved reflective DLLinjection [Web Page]. https://github.com/dismantl/ImprovedReflectiveDLLInjection.

[4] Mertsarica. (2010). Antimeter tool [Web Page].https://www.mertsarica.com/antimeter-tool/.

[5] King, A. (2012). Detecting reflective injection [Web Page]. https://www.defcon.org/html/defcon-20/dc-20-speakers.html#King. DEF CON R 20 Hacking Conference.

[6] Berdajs, J., & Bosnic,´ Z. (2010). Extending applications using an advanced approach to DLL injection and API hooking [Journal Article]. Software: Practice and Experience, 40(7), 567-584.

[7] Richter, J. (1994). Load your 32 bit DLL into another process's address space using injlib [Journal Article]. Microsoft Systems Journal-US Edition, 13-40.

[8] Lam, L.-c., Yu, Y., & Chiueh, T.-c. (2006). Secure mobile code execution service. In Proceedings of the 20th conference on large installation system administration (pp. 5–5).

[9] Help, M., & Support. (2010). Working with the appinit DLLs registry value [Web Page]. https://support.microsoft.com/en/us/help/197571/working-with-the-appinit-dlls -registry-value.

[10] Kuster, R. (2003). Three ways to inject your code into another process [Web Page]. https://www.codeproject.com/ Articles/4610/Three-Ways-to-Inject-Your-Code -into-Another-Process

[11] Newcomer, J. M. (2001). Hooks and DLLs [Web Page]. https://www.codeproject.com/Articles/1037/ Hooks-and-DLLs.

[12] Shewmaker, J. (2010). Analyzing DLL injection [Web Page]. http://www.bluenotch.com/.

[13] NTCore. (2012). Explorer suite [Web Page]. www.ntcore.com/ exsuite.php.

[14] Barabosch, T., & Gerhards-Padilla, E. (2014). Host-based code injection attacks: A popular technique used by malware [Conference Proceedings]. In Malicious and unwanted software: The americas (malware), 2014 9th international conference on (p. 8-17). IEEE.

[15] Jang, M., Kim, H., & Yun, Y. (2007). Detection of DLL inserted by windows malicious code [Conference Proceedings]. In Convergence information technology, 2007. international conference on (p. 1059-1064). IEEE.

[16] Glendowne, D., Miller, C., McGrew, W., & Dampier, D. (2015). Characteristics of malicious DLLs in windows memory [Conference Proceedings]. In Ifip international conference on digital forensics (p. 149-161). Springer.

[17] VoxelBlock. (2016). Basic and intermediate techniques of uwp app modding [Web Page]. https://www.unknowncheats.me/ forum/general-programming-and-reversing/177183-basic intermediate-techniques-uwp-app-modding.html.

[18] Yucheng, G., Peng, W., Juwei, L., & Qingping, G. (2011). A way to detect computer trojan based on DLL preemptive injection [Conference Proceedings]. In Distributed computing and applications to business, engineering and science (dcabes), 2011

[19] Sun, H.-M., Tseng, Y.-T., Lin, Y.-H., & Chiang, T. (2006). Detecting the code injection by hooking system calls in windows kernel mode [Conference Proceedings]. In 2006 international computer symposium, ics.

[20] DLL [Web Page]. http://www.codeguru.com/ cpp/g-m/directx/directx8/article.php/c11453/

Intercept-Calls-to-DirectX-with-a-Proxy-DLL.htm.

[21] Microsoft. (n.d.-a). Process mitigation binary signature policy structure [Web Page]. https://msdn.microsoft.com/en-us/library/windows/desktop/mt706242(v=vs.85).aspx

[22] Cowan, C. (2015). Protecting microsoft edge against binary injection [Web Page]. https://blogs.windows.com/msedgedev/2015/11/17/microsoft-edge-module-code -integrity/.

[23] Rascagneres, P. (2016). Microsoft edge binary injection mitigation overview [Web Page]. http://www.sekoia.fr/blog/microsoft-edge-binary-injection-mitigation -overview/.

[24] Microsoft. (2002). Detours [Web Page]. https://www.microsoft.com/enus/research/project/detours/.

[25] Mhook, an API hooking library [Web Page]. (2014). https:// github.com/martona/mhook.

[26] Deviare API hook [Web Page].(2017). http://www.nektra.com/products/deviare-api-hook-windows/.

[27] Easyhook the reinvention of windows API hooking [Web Page]. (2017). https://easyhook.github.io/.

[28] Zaheri, M., & Niksefat, S. (2017). Github project for preventing reflective DLL injection on UWP apps [Web Page]. https:// github.com/m0jt4b4/UWPHardening.

[29] FutureMark. (2016a). Pcmark 8: The complete benchmark for windows [Web Page]. http://www.futuremark.com/benchmarks/pcmark.

[30] rohitab.com. (2017). APIMonitor: Spy on API calls and COM inter-faces [Web Page]. http://www.rohitab.com/apimonitor.