Securing Windows Subsystem for Linux A Behavioral Detection Approach

1st Andrei Mermeze
dept. name of organization (of Aff.)
name of organization (of Aff.)
Cluj-Napoca, Romania
andrei.mermeze@gmail.com

2nd Phd. Radu Dragos
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address

***Abstract*—The release of Windows Subsystem for Linux (WSL) revealed a whole new attack surface, comprised of kernel drivers as well as user-mode services. The performed research is going to reveal how behavioral detection techniques can be applied for detecting potential threats (bashware) that abuse WSL. This paper will provide an insight into the security issues created by this subsystem, while also explore both Kernel-Mode and User-Mode based detection heuristics and techniques in order to identify and block this new type of malware. First sections focus on WSL internals, the next sections present what mechanisms can be used to acquire the needed information for identifying bashware, and finally some heuristic behavioural based approaches on identifying the bashware.**

***Index Terms*—Windows Subsystem for Linux Security Behavioral Analysis Malware Bashware Detection Event Tracing Minifilter Monitoring Kernel Driver**

I. Introduction

Windows Subsystem for Linux was first released in the annniversary update and it provides a way of executing Linux ELF 64 bit binaries on native Windows 10. Since WSL is not a virtualization based system, the Linux processes running in it can access any resource on the computer, making it a dangerous threat untill a AV Solutions are updated to support this new type of processes.

II. Overview of WSL

A. minimal processes and pico processes

A minimal process has a parent, protection level, name and security toekn. It has no initial thread, no process enviroment-block (PEB), no ntdll, and its address space is empty. Its threads are called "minimal threads", and, similarly to minimal processes, they have no thread enviroment block (TEB). A pico process is a minimal process, but it has an associated pico provider, which handles the system calls, exceptions, pico process or threads creation and termination.

B. Pico Providers

A pico provider is a kernel driver that implements the required functionalities to handle pico process events. Registering as a pico provider is done by calling the PsRegisterPicoProvider API, however, calling this API requires that the PspPicoRegistrationDisabled is set to FALSE. This value is set to false before any other third party driver is loaded, therefore, at least for now, it is not possible to implement custom pico providers. In order to secure pico providers, they also register with PatchGuard in order to protect its syscalls. This renders the classic process monitoring solution(syscall hooking) useless, as any attempt to hook the syscalls will result into a BSOD.

Lxcore.sys does the pico provider registration in the LxInitialize function, which is called by lxss.sys in its DriverEntry.

```
result = PsRegisterPicoProvider(&v4, &LxpRoutines);
```

Fig. 1. LXCORE pico provider registration in

C. Sycalls

Currently, as of Windows 10 1709, 242 Linux syscalls are implemented in lxcore.sys.

D. File System

- VFS
- VolFs
- DrvFs
- SysFs - /sys
- TmpFs - /tmp
- ProcFs - /proc

III. Differences Between lxcore and Linux Kernel

IV. Exploring the Attack Surface

Lxcore is the kernel-mode component of WSL that implements the Linux syscalls, multiple filesystems, etc...

V. Known Vulnerabilities

A. Execve Exploit

Discovered by Saar Amar

B. Local Denial of Service

VI. Monitoring WSL Activity

There are multiple ways of monitoring WSL activity, which, if correctly combined, can provide enough information in order to identify potentially malicious activity.

- User-Mode Hooking Framework - API usage monitoring

- File System Minifilter - file system I/O and process monitoring
- Windows Filtering Platform - network monitoring

A. User-Mode hooking driven WSL monitoring

In order to obtain more granular monitoring capabilities, a function hooking framework is required. The ability of synchronously monitoring specific API usage opens up a range of possibilities, from exploit detection to syscall graph based detection heuristics. In order to load a shared library into any Linux pico process, an entry with the path to the library must be added into the ld.so.conf.

B. Kernel-Mode driven WSL monitoring

This is the most reliable way of monitoring and blocking potentially malicious Linux applications. Unlike a user-mode hooking framework approach, using a Windows driver provides mechanisms of monitoring that cannot be bypassed or tampered with by the monitored processes, making it a very reliable and secure approach.

The driver we've implemented filters file system I/O and also keeps track of active processes. In the next sections we are going to present how we implemented the driver.

1) File System Filtering: In Windows I/O request packets(IRPs) are used to communicate between drivers. For example, in the context of the file system, IRPs are used to describe file I/O operations, like file read (IRP_MJ_READ) or file create (IRP_MJ_CREATE). A minifilter driver can register callbacks with fltmgr by calling FltRegisterFilter.

The main difference between filtering I/O requests issued by win32 processes and Linux processes is that, for Linux processes, the PreviousMode is KernelMode, because IRPs are issued by the kernel, not by the actual process. Considering this, we filtered kernel issued IRPs too, but only those for which the requestor id was not system's PID.

2) Process Filtering: In order to receive process creation and termination notifications, the driver must register a callback with PsSetCreateProcessNotifyRoutineEx2. This callback will be called for both win32 processes and pico processes.

To identify a WSL process, the driver must call ZwQueryInformationProcess with SubsystemInformationTypeWSL information type on the process' handle and check that the subsystem type is SubsystemInformationTypeWSL.

VII. Detection Heuristics

A. Ransomware

Ransomware is a type of malware that encrypts the users' files and demands money, usually in the form of crypto currency, like bitcoin, in order for the user to regain access to the files. More advanced ransomware could go as far as preventing the operating system to boot untill the payment is made. An example of such ransomware would be petya.

A ransomware that targets WSL would be leverage the fact that most AV minifilters do not filter IRPs that are coming from the kernel, allowing the ransomware to silently encrypt files. Even more, the ransomware might silently communicate with a server without being detected by network filters.

A reliable solution would be to monitor kernel issued IRPs that are done on behalf of a Linux process, and to use a either a scoring engine or some AI expert system (i.e. Support Vector Machine) to identify the ransomware. We are going to cover only the behavioral heuristics approach in this paper.

In case of a scoring engine, the process could be assigned points according to the file system activity it does. For example, a file delete or write would add more points than a file read. Moreover, even more points could be added if the action involves sensitive paths (i.e. system root, Windows directory, etc). When the process reaches a certain threshold, a detection allert should be issued and the process should be killed. In order to detect multi-process ransomware we used process groups algorithm.

We could further optimize by allowing or adding fewer points to processes that are known to access certain Windows paths, for example, a latex compiler that writes the compiled PDF in the user's "Documents" folder. In order to do this, we keep a dictionary (internally, a trie) with all paths we monitor. Each path has an associated score for each I/O operation (i.e. 5 points for a delete, 2 points for a write, etc). Whenever we filter an I/O request for a linux process, or a windows process that is in a linux process' group, we match the path against our dictionary, and add points accordingly.

B. Windows Privillege Escalation

Privillege escalation is a powerful exploit that is meant to elevate an unelevated process without users' consent or knowledge. It is very important to understand the elevation process in order to identify the exploit. In this section we are going to describe in detail how elevation is done in Windows and how we can detect the exploit.

On Windows, there are two ways of starting an elevated process:

- from an unprivilleged process, through UAC (user account control)
- from a privilleged process

When an unprivilleged process needs to start a privilleged process, it sends a request to svchost, which shows the User Account Control pop-up, notifying the user that the process needs admin rights. If admin rights are granted by UAC,

svchost starts the process. If a process already is elevated, any process spawned by it will also be elevated. We can easily see that, if wsl.exe is started with admin rights, all Linux processes running in that WSL instance will be granted admin rights inside Windows, which would be disastrous considering the security of the machine.

This kind of exploit can be detected easily, while not very reliable, from a kernel driver, or, more reliable, by using memory introspection techniques from a hypervisor. While the latter is extremly reliable, it is much more complex than a kernel-driver. We will cover only the first technique and try to make it as reliable as possible withouto adding too much performance overhead.

We need to check that the process' token was not tampered with. The algorithm for checking if a process is elevated is fairly simple. We use ZwOpenProcessTokenEx to get the process' security token and then ZwQueryInformationToken with TokenElevation information class. The returned boolean will tell if the process is elevated or not. This check can easily be done at in the process notification callback because we get the process' handle as a parameter.

Seeing how execve exploit works, it is clearly obvious that while this method works for the most basic case, it is quite unreliable. That's because we need to poll the security token at certain moments since the exploit just overwrites the process' security token with the System security token. Such key moments would be whenever we intercept actions that would naturally require admin privilleges, i.e. acessing the Windows or System32 folders. It's easy to see several issues:

- If the exploit elevates the current process while running
- If the exploit is used to elevate another process
- If the elevated process is a Windows process
- Performance overhead added by polling the security token

I'm going to explain how we can tackle these problems and make the heuristic more reliable.

1) Detecting privillege escalation on already running processes: In order to do this we need to keep track of all active processes in a structure that we'll call a process collector. When we add a new process into the collector, we will also store its initial elevation. Then, whenever the processes does sensitive actions, like dropping executables in paths like "C:\Windows", we will recheck the security token with the algorithm previously presented. If the new elevation differs from the initial elevation, the process should be detected and killed.

2) Detecting elevation of a running Windows process: The same algorithm could be applied for Windows processes that are started by Linux processes as well, with two observations. We need to keep track of Windows processes as well, but only if they are part of a process tree that has a Linux process as well, and we need to detect and kill the whole process tree.

3) Minimizing the performance overhead: Polling the security token for every Linux process would become costly performance-wise very quickly, especially if the process does I/O intensive operations. It is neccessary to narrow the set of actions for which we do security token registraiton, as well as use an efficient algorithm for matching strings against a predefined dictionary.

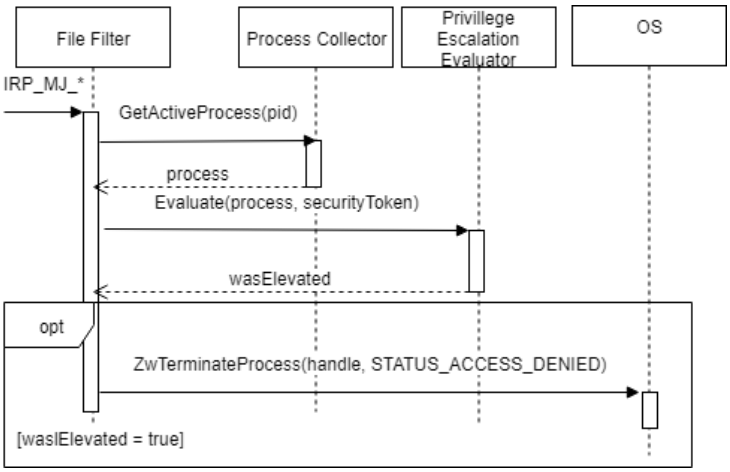

Fig. 2. privillege escalation detection diagram

VIII. Detection Framework

References

[1] Pavel Yosifovich, Alex Ionescu, Mark E. Russinovich, and David A. Solomon "Windows Internals"

[2] Alex Ionescu "THE LINUX KERNEL HIDDEN INSIDE Windows 10," BlackHat2016

[3] Alex Ionescu "GAINING VISIBILITY INTO LINUX BINARIES ON Windows: DEFEND AND UNDERSTAND WSL," BlueHat2016

[4] Saar Amar "LINUX VULNERABILITIES, Windows EXPLOITS Escalating Privileges with WSL," BlueHat2018

[5] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel "A Survey on Automated Dynamic Malware Analysis Techniques and Tools,", ACM Computing Surveys (CSUR) ,February 2012