Behavior Analysis and Detection of Bashware

Andrei Mermeze
email andrei.mermeze@gmail.com

Advisor: Phd. Dragos Radu

Contents

1. Introduction

1.1 Motivation

With the release of Windows Subsystem for Linux (WSL), consisting of two new kernel-mode drivers, lxcore.sys and lxss.sys, which implement more than 200 linux system calls, a new service, LxssManager, and other user mode components. This opened an attack surface that was not covered by monitoring tools untill recent, and doesn't seem to be covered at all by anti-virus vendors.

1.2 Objectives

The main objective is to provide a partial solution to this new security problem. It should defend a system from bashware, suspicious interraction between WSL processes and Windows processes and to provide system administrators with enough logs to properly respond to incidents, while not disrupting the user experience.

1.3 Personal Contributions

My personal contribusions towards monitoring and detecting malicious applications that leverage Windows Subsystem for Linux (WSL) are comprised of a few behavioral heuristic algorithms for detecting potentially malicious applications that target WSL, and how these applications can be monitored and stopped if malicious. Moreover, I have integrated this monitoring and detection system in a application in order to exemplify how easily it could be integrated by any other third party anti-virus vendor.

1.4 Thesis Outline

Chapter 2 will contain the current state of WSL and how it can be used maliciously ... ceva ceva. The following four chapters will present the technologies used in order to develop the application as a whole, its architecture along with some implementation details and finally the testing process. Chapter 7 will be dedicated to describing the detection algorithms logic and what steps I followed in creating them, as well as some implementation details and limitations. Lastly, chapter 8 will show possible directions for this project, how the current monitoring solution can be improved and how it could be extended to achieve more in-depth monitoring and more accurate detection.

2. State of the art

2.1 Windows Subsystem for Linux

2.1.1 WSL Main Components

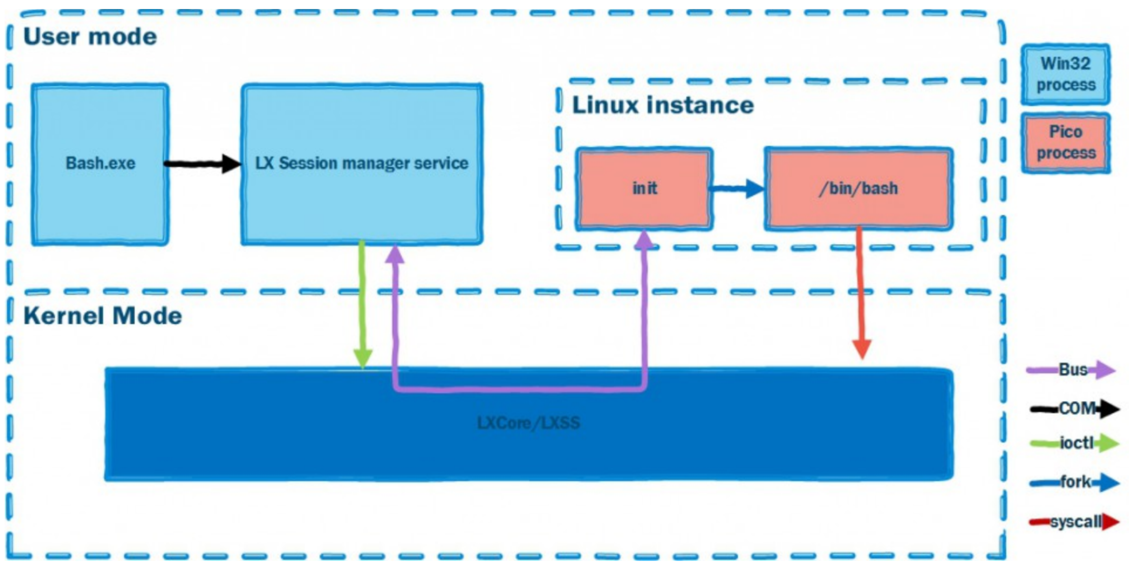

Figure 2.1: WSL Compoents [**wslcomponents**]

2.1.2 WSL File System

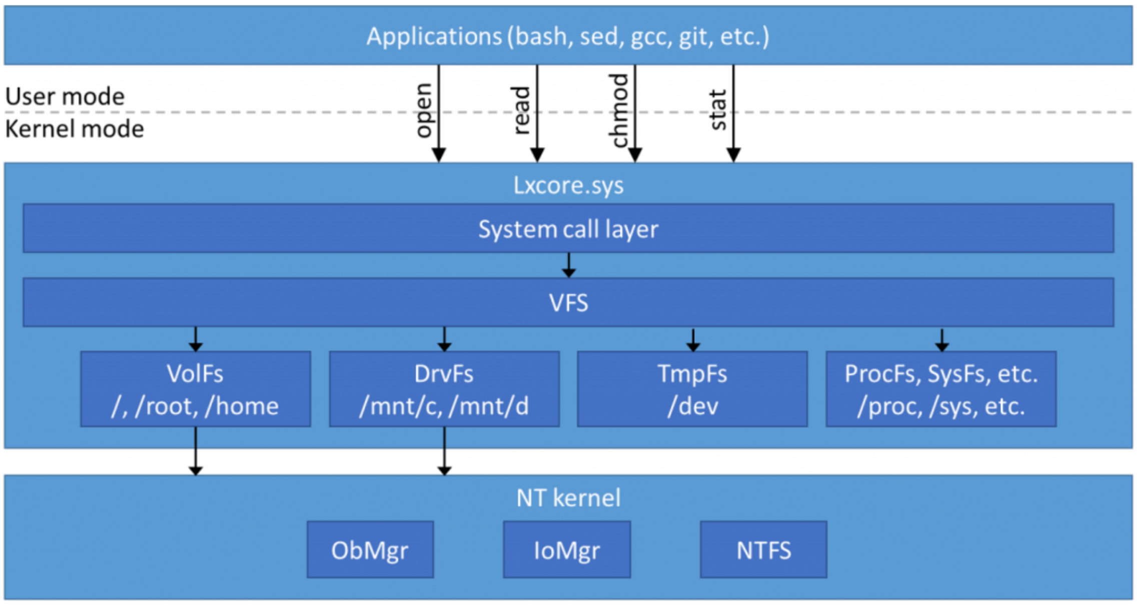

Figure 2.2: WSL File System [**wslfilesystem**]

2.1.3 Security Issues

2.2 Bashware and Exploits

Bashware is a type of malware that leverages WSL in order to bypass monitoring tools and anti-virus products in order to do damage to the operating system.

2.2.1 execve exploit

Discovered in February 2018 by Saar Amar, security researcher, is a privilege escalation exploit that leverages a vulnerability in lxcore, more exactly an integer overflow in LxpUtilReadUserStringSet. Starting from this vulnerability, the exploit proof of concept copies the system security descriptor over another process' security descriptor, elevating it at runtime.

2.2.2 File Infector

A wsl file infector would essentially inject into windows executable files, altering the code in order to deliver some malicious code.

2.2.3 Local Denial of Service

While experimenting with Event Tracing for Windows (ETW) I've found a way of consistently cause lxcore.sys to run into a Blue Scren of Death (BSOD), due to an access violation exception. This can be triggerd from a normal process with no admin rights. I will not go into more detail about this yet, as it would not respect the responsible disclore policy, since it wasn't yet fixed by Microsoft. Releasing information about how to trigger this local denial of service attack could cause exploitation in the wild.

2.3 Behavioral Detection

3. Used technologies

3.1 File System Legacy Filter and Minifilter Drivers

A Legacy Filter driver is a kernel-mode that could attach to a device's stack. In the context of file system filtering, these filter drivers could intercept file system I/O operations. Developing legacy filter drivers was quite troublesome and led to many incompatibilities between filter drivers. This is one of the reasons for which minifilter drivers were added.

Minifilters have the same abilities as file system legacy filter drivers, but they are easier to develop and are overall safer. Their load order no longer depends on the attach order, but on a predefined value named altitude. Minifilters are managed by FltMgr, which is a legacy filter driver implemented by Microsoft.

I have used this technology for the core component of the developed application, which had to filter file system operations as well as process creation and termination. All of this information can't be reliably be acquired from a user mode application, so a minifilter driver was the only choice.

3.2 C++ in Kernel Drivers

C++ has the advantage of being easy to use in developing coherent, object-oriented, robust and safe applications. However, msvc compiler does not, by default, support c++ in kernel drivers.

Firstly, in order to support global initializers, we need to define two sections, ".CRT$XCA" and ".CRT$XCZ". Between these two sections all pointers to global initializers are held.

Secondly, In order to support static object initialization, we need iterate through the list of global initializers and call each constructor in the DriverEntry function, and call the destructors during driver unload.

Thirdly, we need to define the global new and delete operators.

Lastly, we need to implement an the _purecall() which will be called whenever a pure virtual function is called. Microsoft's implementation of this function causes the program to immediately terminate if there is no exception handler for it.

3.3 .NET Framework

I have used .NET Framework for the integration project, both for the system service and the GUI application. The GUI was developed with the help of Windows Presentation Foundation (WPF), with the interface designed using Extensible Application Markup Language (XAML), and the communication between the service and GUI was done through Windows Communication foundation (WCF), more exactly, through the net.tcp protocol. WCF was particulary useful as it exposes an easy to use, RPC-like comomunication interface

3.4 C++/CLI

The C++ modified Common Language Infrastracutre is a c++ based language specification developed by Microsoft. It offers the possibility to use both the unmanaged CRT heap (allocating and freeing objects) as well as the .NET managed heap, where objects are garbage collected and don't have to be freed by the programmer. This technology was helpful because it allows an easy linkage with a C or C++ DLL and provides a simple way of wrapping the dll's functions and classes in order to export them as .NET objects to .NET application, which, in this case, was a service.

4. Architecture

4.1 High Level Overview

The system is composed of 2 main components

- wslmon - an sdk that contains the detection logic as well as communication logic between kernel-mode and user-mode
- wslam - the integrator

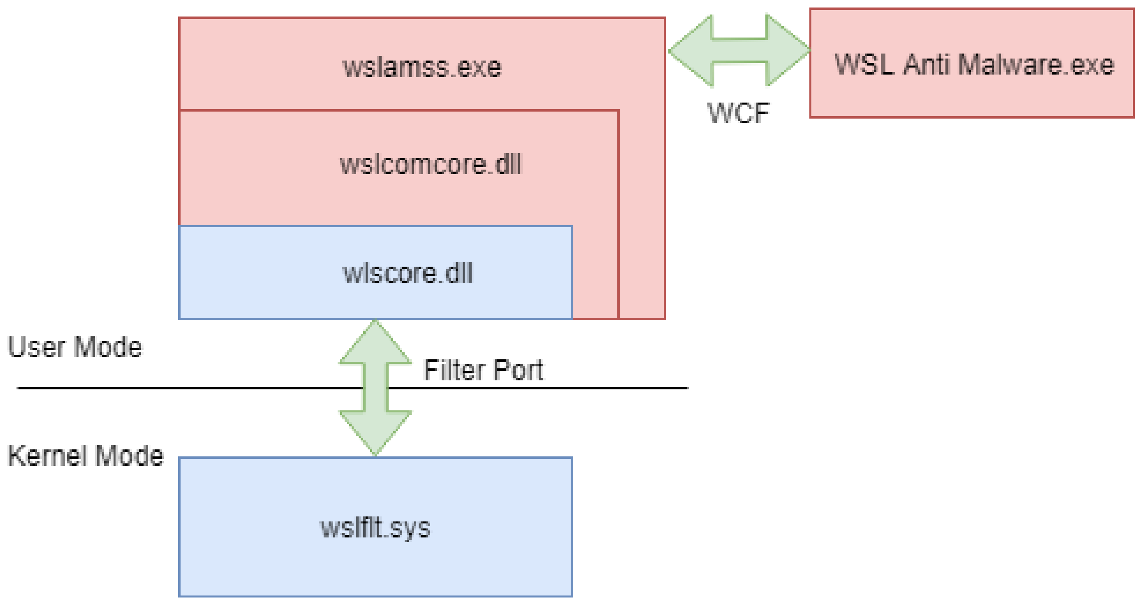

Figure 4.1: High Level Overview

4.1.1 wslmon

Wslmon is a software development kit that could be integrated by any OEM to include it in an existing security solution. It consists of:

- wslflt.sys - a minifilter driver that encapsulates the monitoring, analysis and detection logic
- wslcore.dll - a shared library that provides an interface to communicate with the minifilter driver

wslflt.sys is C++ minifilter driver that contains the requirued sensors for monitoring process' activity. It's key components are the process filter, file filter, event dispatcher, the communication framework and the detection heuristics.

4.1.2 wslam

Integrates the wslmon SDK. It consists of:

- uwpcomm.dll - WinRT
- wslamsscomcore.dll - C++/CLI
- wslamss.exe - .NET Service
- WSL Anti-Malware.exe - UWP Store application
- NotificationsListener - Background app service deployed by the WSL Ant-Malware.exe store app

uwpcomm.dll uses WinRT in order to send notifications to the background app service via an AppServiceConnection.

wslammsscore.dll is built on top of C++/CLI (Common Language Infrastracture) to wrap uwpcomm.dll and wslcore.dll and export managed .NET classes for use in the .NET service.

wslamss.exe contains the integration logic (ie. exceptions, notifications) and servers as a communication endpoint, built using WCF framework, for the store application.

WSL Anti-Malware.exe is a windows store app that

NotificationsListener is an app service, implemented as a background task, that listens to integrator notifications even if the store application is closed, and notifies the user about specific events via toast notifications

5. Implementation Details

5.1 Process Filtering

```
PsSetCreateProcessNotifyRoutineEx2(a, b);
```

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 best choice for achieving reliable file system i/o and process monitoring is a file system minifilter driver.

5.2 File System Filtering

Windows file system i/o requests are contained in IRPs (i/o request packets). The minifilter driver registers its callback with the filesystem by calling FltRegisterFilter and passing an array of callbacks. The main difference between filtering i/o requests issued by win32 processes and linux processes is that, for linux processes, the RequestorMode is KernelMode, because IRPs are issued by the kernel. This means that unlike the usual filtering algorithm, which skips KernelRequests, now we must process and check that the Requestor PID is not system.

When filtering file system operations, we are mostly interested in 2 fields of the FLT_CALLBACK_DATA structure, while the FILE_OBJECT is taken from the FLT_RELATED_OBJECTS structure.

```
0: kd> dt fltmgr!_FLT_CALLBACK_DATA
...
0x10 Iopb         : Ptr64 _FLT_IO_PARAMETER_BLOCK
...
0x50 RequestorMode: Char

0: kd> dt fltmgr!_FLT_RELATED_OBJECTS
...
0x20 FileObject   : Ptr64 _FILE_OBJECT
...
```

5.3 Process Monitoring

Process monitorig refers to getting synchronous notifications on both process creation and process termination in order to keep track of the currently active processes and process hierarchy.

In order to receive these 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.

5.4 Communication

5.5 Detection Flow

5.6 Scoring Engine

6. Testing

Rigorous testing, especially in the context of kernel-mode modules and security solutions, is essential in order to provide a stable, usable and efficient product. Combining whitebox and blackbox techniques together with test driven developement paradigms throughout the development of a product is crucial in order to offer a high quality product.

6.1 Whitebox testing framework

In order to use the Test Driven Development paradigm, a whitebox unittesting framework was required. The unittest project contains mock definitions of kernel functions (e.g. ExAllocatePoolWithTag), and unittests for each class. Mocking kernel functions helped in achieving high coverage of the driver code in a user-mode enviroment, thus saving development time. Moreover, these unittests can be easily configured to run at each solution build, making it a very reliable and easy to use continuous integration tool.

The other alternative was having another kernel-mode driver that would test the API exported by wslflt.sys. This is an unreliable and slow testing method because most bugs would cause a blue screen and can even corrupt the testing enviroment. Moreover, analysing kernel dumps in order to find bugs is a much slower process compared to analysing user-mode crashes or exceptions. Also, building a continuous integration system around this testing framework is a lot more complicated because it involves virtual machines, automatiically applying OS images and re-applying them in case a BSOD occurs.

6.2 Blackbox testing

Blackbox testing, both manual and automated, is the main method of attesting a product's value, usefuleness and correctness regarding the users' needs. I have used blackbox techniques in order to verify both functional (i.e. detection0 and non-functional requirements (i.e. stability, security, reliability).

6.3 WHQL Testing

Windows Hardware Quality Lab[**whql**] tests involve running various testcases regarding Disk I/O, code integrity checks, pipes I/O, transactional I/O and more.

In order to release a driver to the market, it needs to be digitally signed by microsoft, and the only way of receiving the signature, it must pass the WHQL test suite. In the case of wslflt.sys, that would be the Filter Driver Test Suite[**fdts**].

7. Heuristics and Detection Algorithms

7.1 Privilege Escalation Detection

Privilege escalation is a type of exploit that elevates a running process or starts an elevated process while bypassing operating system's security or user notification mechanisms (eg. User Account Control on Windows). In order to correctly identify the exploit it is essential to understand how elevation works on Windows. In this section I am going to go into detial about how windows elevation works and the algorithm I used in order to detect it.

There are multiple legitimate ways of starting a process as admin:

- from an unprivileged process, via ShellExecuteEx
- request administrator rights at process start via manifest
- from a privileged process by simply calling CreateProcess

However, there is no legitimate way to legitimately elevate an already running process.

In order for a process with no admin rights to start an elevated process, it has to send 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. This can be done with the ShellExecuteEx API, or by setting the requestedExecutionLevel field to ”requireAdministrator” in the manifest file.

Manifest snippet for requesting administrator rights on process start:

```
<trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
<security>
<requestedPrivileges xmlns="urn:schemas-microsoft-com:asm.v3">
    <requestedExecutionLevel level="requireAdministrator" uiAccess="false" />
</requestedPrivileges>
</security>
</trustInfo>
```

If a process already is elevated, any process spawned by it will also be elevated by default. 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 consoidering the security of the computer.

An easy, yet not completly reliable method for detecting this type of exploit can be implemented in a kernel driver. A more reliable method would be using memory introspection techniquest from a hypervisor, but I will only cover the first.

I'm going to describe the most basic way of detecting privilege escalation, which is, detecting if a process was maliciously started with admin rights. As I've previously mentioned, a process can start with admin rights if and only if its parent is elevated or it was started by a svchost. In order to do this, we need check whether or not the process is elevated in the process create callback. Below is a summary implementaion for a function that takes a process handle and returns its current elevation status.

```
ZwOpenProcessTokenEx(processHandle, GENERIC_READ, OBJ_KERNEL_HANDLE, &token);
ZwQueryInformationToken(token, TokenElevation, &tokenElevation, sizeof(tokenElevation),
                        &returnLength);
return tokenElevation.TokenIsElevated;
```

Seeing how execve exploit works, it is clearly obvious that while this method works for the most basic case, it will not be enough for an exploit that elevates an already running process. Therefore, we need to check that the security token was not tampered with whenever we suspect that the process might be doing something potentially malicious. For example, any file operation done in the %SystemRoot% directory.

At this point, it's easy to see several issues:

- if the exploit elevates a running process
- if the elevated process is a Windows process
- performance overhead added by polling the security token

Further, I will explain how I have tackled these issues and enhanced the algorithm.

7.2 File Infectors and Droppers Detection

File infectors tamper with existing executables, injecting malicious code into them. Even though these can be prevented by the provider of the executable by digitally signing it, unsigned executables are vulnerable to being infected, posing a threat to the users that use them. Droppers however would write malicious executables on disk and are commonly disguised as installers or application bundles.

These are relatively simple to detect, as linux processes would have no reason to create or modify windows executables. The more complicated case is when a linux process starts a windows process to carry out the potentially malicious actions on behalf of the linux process. Untill now, there haven't been observed any WSL applications interact with windows processes it is still not clear how to classify this behavior. For now, untill the market evolves (if it does), the safer decision would be to consider windows processes that are in a linux process tree (there is at least a linux process ancestor) same as linux processes. This means that whether the linux process drops an executable or starts a windows process to drop it, both cases would cause a detection that kills the whole tree (up to the last linux process)

The relevant action a file infector does writing in windows executable files, which actually is, IRP_MJ_WRITE in terms of file system filtering.

8. Possible improvements

This being a partial and basic solution to the problem, there are many improvements that can be made, for functional aspects like detection, and monitoring capabilities as well as for non-functinoal aspects, like performance, usability and overall robustness of the application.

8.1 Performance

There are multiple components that could be further optimized, for example, as of current implementation, the process collector is a doubly linked list, which is very inefficient for lookups (O(n) complexity). A more efficient implementation would be a resizable hash table using quadratic or lilnear probing. Another example of a component that needs optimization is the dictionary used to match paths for which we recheck the security token. A better implementation would use a compressed trie and aho corrasick algorithm for matching the path.

Another performance improvement would be to not rematch the path for the same opened file each time we encounter it, but to cache the results in a minifilter define structure called "context", which can be attached to any filter manager object (i.e. file object, volume, instance, etc.).

8.2 Detection

Currently, only privilege escalation is covered and, partially, file infectors. A very important type of malware that should be covered is ransomware. Ransomware is a type of malware that encrypts files and asks for a ransom, usually in the form of cryptocurency, in order to decrypt the user's files.

8.3 Revert malicious actions

In case of behavioral detection, a process could do some malicious actions before being detected. For example, a ransomware may actually encrypt some files or parts of some files before being stopped by the anti-virus solution. In this case, it would be needed to store all permanent actions (i.e. file delete) and revert them when and if the process is detected.

8.4 Network Filtering

A network filter kernel-mode driver is required in order to monitor and report suspicious network activity inside WSL. For example, such a component could possibly detect reverse shell behavior.

8.5 User-Mode Hooking Framework

Injecting a shared library into monitored process could be, at least for now, the only way to detect WSL process hollowing or process injection.