BABE-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
SPECIALIZATION IN COMPUTER SCIENCE

DIPLOMA THESIS

A Behavioral Analysis Approach on Bashware Detection

Supervisor
PhD. DRAGOS Radu, Lecturer

Author
MERMEZE Andrei

2018

Contents

1. Introduction

With the release of Windows Subsystem for Linux (WSL) in the "Redstone 1" version of Windows 10, 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 up a new attack surface, one that was not covered by monitoring tools untill recent, and doesn't seem to be covered by anti-virus vendors.

WSL provides a way of executing native 64 bit Linux binaries, also known as ELF64 files, on Windows 10. Since WSL is not virtualization based, the Linux processes running in WSL can access the same resources as the Windows processes, while not being monitored by existent tools or anti-malware solutions.

1.1 Motivation

Seeing that currently no anti-virus vendor tried to tackle this issue and develop a security solution that could cover this new complex attack surface, I have decided to do more research about WSL and try to come up with a full stack security solution that covers this attack surface.

Moreover, seeing a proof of concept of a privilege escalation exploit I had another reason to think that developing a security solution for WSL was imperative.

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 interaction 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. The monitoring and detection system had to be self contained in a software development kit (SDK), in order to be easily integrable by any third party AV vendors.

1.3 Personal Contributions

My personal contributions towards monitoring and detecting malicious applications that leverage WSL are comprised of a few behavioral heuristic algorithms for detecting potentially malicious applications that target WSL, techniques for monitoring Linux applications and stopping potentially malicious processes.

Another personal contribution would be the discovery of a local denial of service attack that leverages an access violation vulnerability in lxcore.sys.

Moreover, I have integrated this SDK in an application in order to exemplify real world usage and integration of the designed system.

1.4 Thesis Outline

The thesis can be divided in four main parts. The first part consisting of chapter 2 will contain the current state of WSL and how it may be used to run malicious programs, how it can bypass monitoring tools and anti-malware solutions and explain some exploits that target WSL.

The second part, ranging from chapter 3 to chapter 7, presents the technologies and the reasons why they were used in order to develop the application as a whole, its architecture along with some implementation details and finally the testing process.

The third part consists of chapter 7 and 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 developement 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

Windows Subsystem for Linux (WSL) is a new and complex Windows component which implements a lot of new functionality (linux syscalls), parsing (ELF files parsing) and is based on existing Windows technologies, like pico providers and pico processes.

This new optional feature was released in version 1607 of Windows 10 [16], also known as "Redstone 1" or "Anniversary Update".

2.1.1 Minimal Processes and Pico Processes

A minimal process has no process enviroment-block (PEB), no initial thread, ntdll is not loaded in its memory as it is done for all windows processes, and its address space is empty. However, a minimal process has a security token, a protection level, and a parent. Threads inside a minimal process are called "minimal threads". Similarly to minimal processes, minimal threads have no thread-enviroment block (TEB).

A pico process is a minimal process, with an associated pico provider, which handles the system calls, exceptions, pico process or threads creation and termination.

2.1.2 Pico Providers

A pico provider is a kernel driver that implements the required functionalities to handle the pico process events enumerated before. Essentially, a pico provider is a custom written kernel module that implements the necessary callbacks to respond to the list of possible events (shown earlier) that a Pico process can cause to arise[16]. Drivers can register as pico providers by calling the PsRegisterPicoProvider API, however, calling this API requires that the PspPicoRegistrationDisabledis set to FALSE. This value is set to false before any other third party driver is loaded.

Securing pico providers is done with the help of PatchGuard in order to protect its syscalls handlers. Kernel Patch Protection, also known as PatchGuard, was designed to protect kernel structures from being patched. This renders the classic linux monitoring solution (syscall hooking) useless, as any attempt to hook the syscalls will result in a Blue Screen of Death (BSOD).

Linux processes are implemented as pico processes which have lxcore as their associated pico provider, thus the reason why lxcore implements the Linux syscalls.

2.1.3 WSL Components

The main components of WSL are:

- lxcore.sys
- lxss.sys
- LxssManager
- init

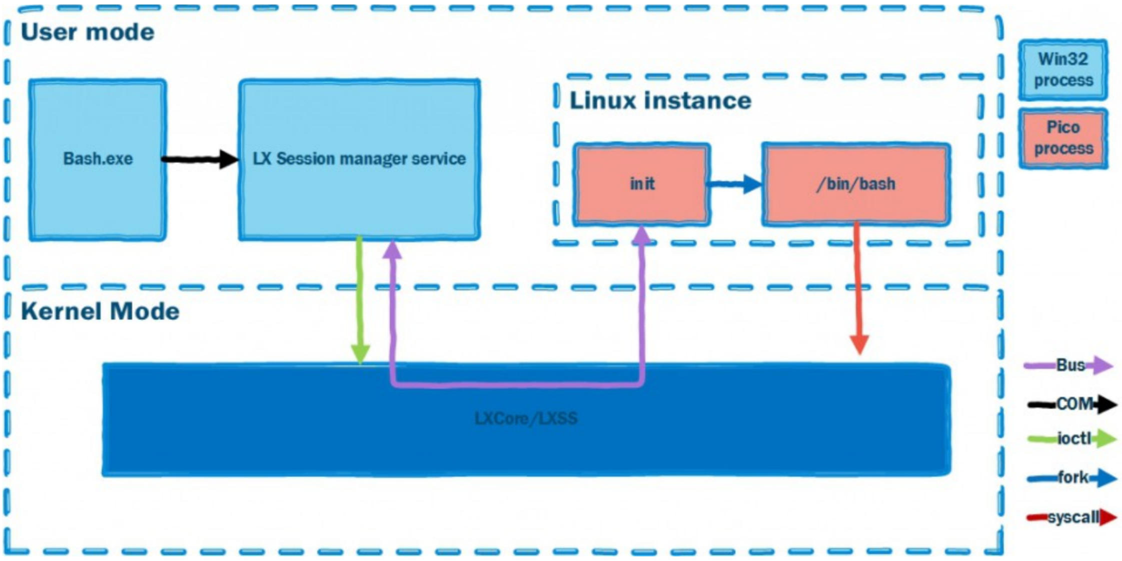
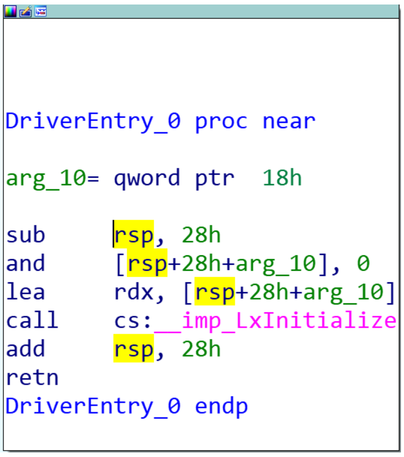


Figure 2.1: Windows Subsystem for Linux Overview [5]

Lxss.sys

Lxss.sys is reponsible for initialing the WSL enviroment by calling the LxInitialize function exported by lxcore.sys, as it can be seen below.



```
DriverEntry_0 proc near

arg_10= qword ptr  18h

sub     rsp, 28h
and     [rsp+28h+arg_10], 0
lea     rdx, [rsp+28h+arg_10]
call    cs:__imp_LxInitialize
add     rsp, 28h
retn
DriverEntry_0 endp
```

Figure 2.2: Lxcore initialization

This being its only responsability, it is currently not of great interest.

Lxcore.sys

Lxcore is the pico provider driver which implements the Linux-compatible kenel API and ABI, as well as a device object for command and control[7], making it the core component of WSL. Over 200 syscalls are implemented in lxcore as well as multiple file systems.

Exposes the ADSS communication bus in order to enable communication between Windows applications and linux applications. Windows applications need to register a server on the ADSS using the ILxssSesion COM interface. The linux application has to open the "/dev/lxss" and connect to the server by sending the correct ioctl. This is currently undocumented and undocumented method of communicating between linux and windows processes and is prone to changes.

LxssManager

LxssManager is a user-mode service which provides a COM interface and communicates with lxcore through its Device Object (\Device\Lxss). Is is responsible with starting the init process of a WSL instance and, as shown in 2.1, does so with the help of lxcore through the ADSS bus.

init

The init daemon process is the well known linux root process (first process started by the kernel), making it the ancestor for every other Linux process. A new init daemon process is created for each WSL instance.

2.1.4 WSL File Systems

Lxcore implements multiple file systems in order to cover well known linux functionality as well as WSL and Windows interoperability.

- VFS (virtual file system)
- DrvFs for Linux files system that maps Windows drives, LxFs,
- VolFs for the linux file system
- ProcFs for /proc
- SysFs for /sys
- TmpFs for /tmp

VFS provides an interface indepentent of the underlying file system. In this model, all file system operations go through the same interface, regardless of the underlying file system type[10].

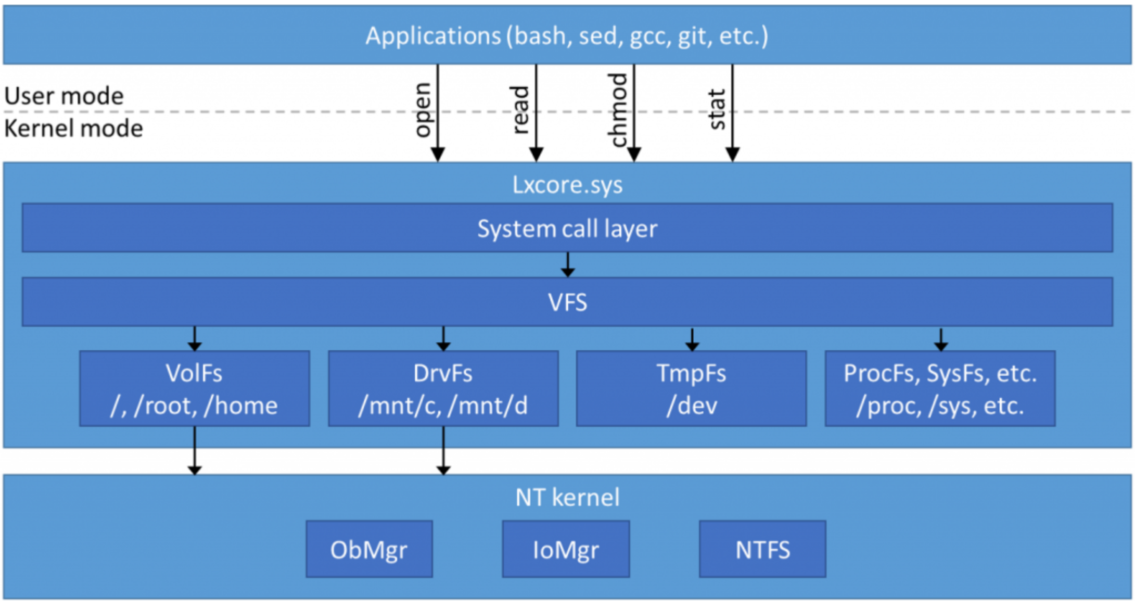
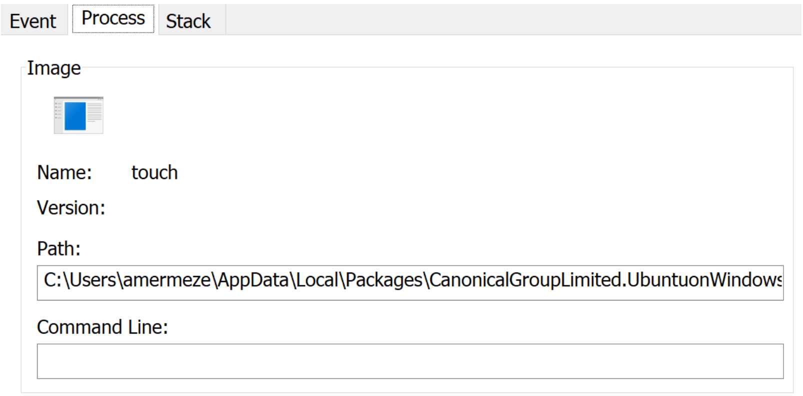


Figure 2.3: WSL File System Support [6]

As can be seen in 2.3, VolFs and and DrvFs operations actually redirected to the Windows kernel after some pre processing, while the requests for the other file systems are resolved by lxcore.

2.1.5 Security Issues

When WSL was first released, multiple monitorinig and analysis tools encountered issues in identifying and displaying information about Linux processes. For example, as it can be seen in the following picture, procmon had issues displaying the commandline a process was started with:



Name: touch

Version:

Path:

C:\Users\amermeze\AppData\Local\Packages\CanonicalGroupLimited.UbuntuonWindows

Command Line:

Figure 2.4: commandline for "touch file"

Callbacks registered for process notifications with the old APIs (PsSetCreateProcessNotifyRoutine and PsSetCreateProcessNotifyRoutineEx) were not called for pico processes, therefore activity monitors would not know of Linux processes. In order to receive notification for pico processes too, a new API called PsSetCreateProcessNotifyRoutineEx2 had to be used, meaning that all activity monitor drivers had to be updated to use the new API if available.

Moreover, since most activity monitors would not filter I/O activity done by the kernel, Linux process I/O activity would not be filtered, because lxcore issues IRPs on behalf of the process that started the I/O operation.

2.2 Malware, Bashware and Exploits

Software that deliberately fulfills the harmful intent of an attacker is commonly referred to as malicious software or malware [13].

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, execve exploit[1] 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 token over another process' security token, 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. Even though this can generally be prevented by software developers by signing their binaries, assuring that, if loaded, were not tampered with, unsigned binaries still exist and are vulnerable to being hijacked.

2.2.3 Local Denial of Service

A local denial of service exploit would greatly distrupt the user experience by causing considerable slow-down of the machine, crashes, freezes, essentially preventing the user from using the machine.

While experimenting with Event Tracing for Windows (ETW) I've found a way of consistently causeing lxcore.sys to run into a 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 disclosure 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

One traditional way of detecting malware is matching the executable file against a set of static, pre generated, signatures. These signatures are created in a way so that they only match malicious software[4]. The problem with signature-based detection is that signature extraction is time consuming, and, once extracted, the anti-virus product must be updated with the new signatures. Moreover, the signature could be rendered useless if the parts from the malware that were used in the signature are modified, if the malware code is encrypted or if the malware modifies its code at runtime. This leaves the user vulnerable to new "0-day" threats.

As stated in "Behavioral detection of malware: from a survey towards an established taxonomy", a behavioral detector needs to monitor the actions of a program and identify actions which betray a malicious intent or activity. In the past, interrupt filtering was the main method of gathering information about a program's behavior. Later it was replaced by hooking system calls (Linux or older Windows version). The modern approach of intercepting a program's actions on Windows systems is to use technologies and APIs that reside in the operating system. For example, using a minifilter driver for intercepting file system activity, or using the PsSetCreateProcessNotifyRoutine API to intercept process creation and termination.

Unlike signature based anti-malware solutions, which is used to detect already known malware, behavioral detection has the advantage of detecting 0-day threats. A behavioral detection anti-malware solution would analyze and evaluate a running process' actions in order to determine whether it is malicious or not. There are many examples of potentially malicious behavior, ranging from assuring persistence on the system to injecting and hijacking other processes.

One obvious disadvantage is the performance overhead that comes with analyzing the behavior of a running process. Another disadvantage is that the process might be able to cause some damage before being stopped by the anti-malware solution, as an example, a ransomware might encrypt a few files or fragments of files before it is stopped.

3. Used technologies

In order to develop a robust and modern application, but also an efficient and reliable monitoring and detection system, I have used various technologies, ranging from low level kernel technologies to high level .NET frameworks.

3.1 Minifilter Drivers

A Legacy Filter driver is a kernel-mode driver that can 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 device stack 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 SDK, 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 reliable choice.

3.2 C++ in Kernel Drivers

C++ was developed from the C programming language and, with few exceptions, retains C as a subset [14]. It 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. Therefore, I had to implement C++ support from scratch.

Firstly, in order to support global initializers, we need to define two sections, ".CRT$XCA" and ".CRT$XCZ". All pointers to initializers reside between these two sections.

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. However, since memroy allocations done in a kernel driver need to specify a memory pool type and a tag, I had also overloaded the new and delete operators, in order to support two additional parameters. This way, the new operator can be called as follows:

```
auto ptr = new(NonPagedPoolNx, 'GAT#') Process{ arg1, arg2 };
```

This will allocate a Process object from the non executable non paged pool with the "#TAG" allocation tag.

Lastly, we need to implement the _purecall() function, 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

I decided to use .NET technology for the integration project because it was easier to design a good and easy to use GUI application and communicate with it through a RPC like interface with the help of WCF. Although, this caused a problem with integration the C++ DLL that provided the control API and notifications for the driver.

3.4 C++/CLI

The C++ modified Common Language Infrastracutre (CLI) 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

Below we can see a high level overview of the sdk and integrator components and how they can interract.

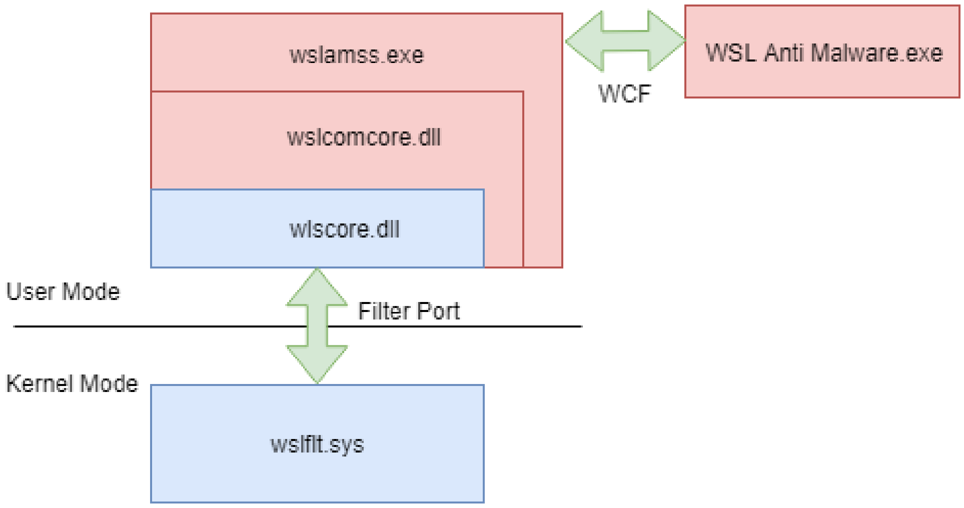


Figure 4.1: High Level Overview

Blue colored components are part of the wslmon SDK, while the red colored components are the integrator components.

4.1.1 wslmon

Wslmon is an SDK that could be integrated by any OEM to include it in an existing security solution. Its architecture is meant to provide an easy way of integration in real anti-virus products, offer enough control of the system while encapsulating the monitoring and detection logic algorithms.

The SDK consists of two components:

- 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 a 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.

wslcore.dll is meant to ease the integration process of the sdk by third party anti-virus vendors. It offers an interface for controlling the driver state (enable/disable monitoring) as well as callback registration for several events (i.e detection).

4.1.2 wslam

Integrating the wslmon SDK, it consists of:

- wslcomcore.dll - C++/CLI DLL
- wslamss.exe - .NET Service
- WSL Anti-Malware.exe - WPF GUI Application

wslcomcore.dll is built on top of C++/CLI to wrap wslcore.dll and export managed .NET classes for use in the .NET service. It's main purpose is to hide the filtering and detection logic and to provide an easy way to integrate the system in a complete security solution.

wslamss.exe contains the integration logic (ie. detection notifications, logging, etc) and serves as a communication endpoint, built using WCF framework, for the GUI application. It uses the previously mentioned dll to control the driver state, process driver's notifications as well as reqests sent from the GUI application. It is designed as a Windows Service application, therefore, making it easily convertible to an anti-malware protected process, once an Early Launch Anti-Malware (ELAM) driver is properly signed.

WSL Anti-Malware.exe is the GUI application that gives control to the user while also containing the user notification mechanism. It is designed as a system tray application in order to not distrupt the user experience, while the GUI itself is designed for usability.

4.2 SDK Architecture

The driver, which is responsilbe with monitoring of processes and detection of potentially malicious applicatoins, is comprised of two "sensors", the first being responsible of monitoring process creation and termination, called ProcessFilter, and the second being responsible of filtering file system I/O operations, called FileFilter.

Most of the main components in the driver are implemented as singleton classes, and that is because of the inherent architecture of the minifilter APIs. For example, the callback for process notification has no context parameter, making it impossible to access any other component, except through singleton classes.

This architecture is based on the single responsability principle, which states that a given method/class/component should have a single reason to change[11]. Every component is self contained and self sufficient in order to be easily testable and maintainable.

4.2.1 Process Filter and Process Collector

The process filter is a stateful component, in the snese that the filter can be enabled or disabled, and responsible for registering the callback for process creation and termination notifications. It contains the logic for deciding whether or not a process should be monitored.

It also contains a process collector, which is used as a databse of active processes, internally represented by instaces of the Process class. A Process object holds the needed information for uniquely identifying the process throughout it's lifetime as well as other information useful for behavioral analysis, like the parent process, security token or commandline.

4.2.2 File Filter

The file filter is also a stateful and singleton component. It's responsible for providing the IRP filtering callbacks and analyzing potentially malicious file system activity.

Currently, the following operations are being monitored by the file filter:

- IRP_MJ_CREATE
- IRP_MJ_WRITE

4.3 Integration Architecture

The SDK integration logic is encapsulated into a wslamss .NET Windows service application and is the juncture between the SDK and any client application which would serve as a user interface.

This architecture makes the integration project highly extensible, allowing the developement of a central application that can monitor reported incidents and other events from multiple clients (systems with the anti-malware solution installed). For example, this could be useful for the network administrators or security reposibles in business enviroments, allowing for faster incident response.

The service has an external-facing WCF interface which is accessible trhough the net.tcp://localhost:8000. The metadata is available at net.tcp://localhost:8000/mex. The metadata exchange endpoint is done via SOAP messages that contain various information that describes the service, such as: available operations, parameter types, return types, callbacks that need to be implemented, etc. In other words, the metadata exchange endpoint exposes the WCF contract implemented by the service.

The settings, detection history and any other persistent data is saved in a local databse that is managed with the help of SQLite database management system and .NET EntityFramework. This was the prefered choice as it doesn't require any special configuration on the user's machine. The only requirement is for the SQLite .NET assemblies to be present in the PATH enviromental variable so they can be loaded by the service.

Throughout the service application, we have two types of persistent data:

- detections history
- current settings

All this data is managed by the classes described below:

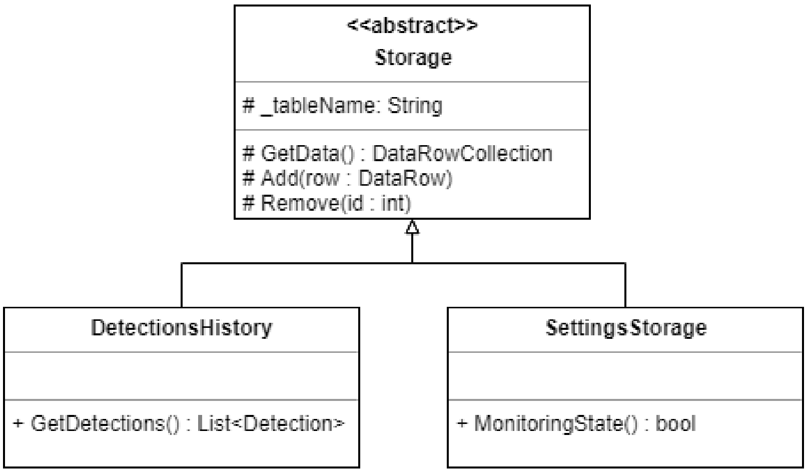


Figure 4.2: Data Persistence Diargram

5. Implementation Details

In this section I will go into detail about monitoring techniques, communication between various components and how the system was implemented.

5.1 Activity Monitoring

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

5.1.1 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.

All monitored processes are stored in a structure named ”Process Collector”. We monitor only Linux processes and windows processes that have their parent process in the collector. The second condition is needed because we have to monitor Windows processes that are started by Linux processes.

For each monitored process we hold an object called Process, which is allocated from a paged memory lookaside list, in order to optimize allocations. A lookaside list is a cache of free memory blocks. Whenever we allocate an object from a lookaside list, if an empty block is available, we pop the empty block from the list and use it, otherwise we allocate from the paged pool. Whenever we free an object allocated from a lookaside list, we push it into the cache if it is not full. Otherwise we actually free the memory block.

Since many processes are started and terminate throught the run time of the system, using lookaside lists for our ”Process” objects turned out to be an important optimization.

The thread safety of the lookaside list's push and pop operations is given by the usage of InterlockedPushEntrySList and InterlockedPopEntrySList Windows APIs, that use SLIST_ENTRY nodes, which are actually nodes in a singly linked list.

5.1.2 File System Activity Monitoring

File system activity monitoring refers to monitoring the disk I/O operations a process does and identifying relevant events such as rename, delete, etc. These events that play a vital role in identifying potentially malicious behavior (i.e dropping an executable in C:\Windows\System32).

Monitoring file system activity can be easily done with a minifilter driver by registering "pre" and "post" callbacks for various operations (i.e. IRP_MJ_CREATE).

Example callbacks registration:

```
constexpr FLT_OPERATION_REGISTRATION OperationRegistration[] =
{
    {
        IRP_MJ_CREATE,
        0,
        PreCreateCallback,
        PostCreateCallback
    },
    ...

    { IRP_MJ_OPERATION_END, }
};
```

After registering the callbacks above, for every IRP_MJ_CREATE IRP (I/O request packet), "PreCreateCallback" will be called before actually doing the operation, giving us the oportunity to skip the post callback call if needed, and "PostCreateCallback" will be called after the operation was completed, giving us information about the completion status, used file object, etc.

An IRP is a partially opaque and self contained structure that represents an I/O operation and holds all of the information needed by a driver to handle that I/O request. The I/O manager creates an IRP in memory to represent an I/O operation, passing a pointer to the IRP to the correct driver and disposing of the packet when the I/O operation is complete[17].

In order to identify the requestor of the IRP we call FltGetRequestorProcessId, which gives us the PID of the process. The RequestorMode field can be either KernelMode or UserMode, telling us whether the IRP was issued by the kernel or not. Traditionally, most AV-solutions would skip KernelMode IRPs, as they would not be important.

This was one of the issues that came along with WSL, as linux processes, not knowing of the Windows kernel, can't issue IRPs by their own. Instead, the pico provider (lxcore.sys) issues them, therefore the IRPs have RequestorMode equal to KernelMode. However, the requestor PID is that of the actual process that started the I/O operation, therefore the IRP filtering algorithm would filter KernelMode IRPs too, if and only if the requestor PID is different than system process PID.

5.2 Communication

Two communication channels were implemented for this application:

1. wslflt.sys to wslcore.dll
2. wslamss.exe to WSL Anti-Malware.exe

The first communication channel is designed to provide a user-mode component with control over the driver while also giving the driver the ability to notify a user-mode component about certain events (i.e. detection).

Communication between the driver and the dll used for integration is implemented using Windows Filter Ports. This is a minifilter specific communication technology which enables two-way communication between a minifiter driver, acting as a server, and a user-mode application, acting as a client.

The second communication channel has the purpose of connecting the .NET service which integrates the SDK to the GUI application and is a two-way, RPC like, net.tcp protocol channel.

The server endpoint is located in the .NET service which integrated the SDK, while the client service is located in the GUI application. Both endpoints have similar architecture, where requests from the other side are implemented as .NET events, and requests towards the other side are done through an service contract interface.

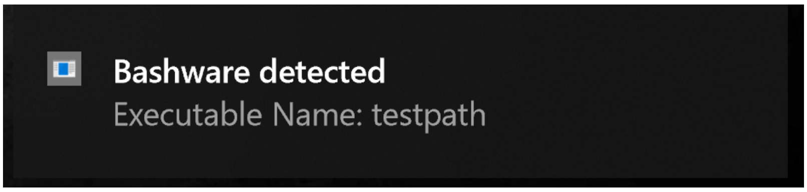
For now, the service provides only one type of notification, which is the process detected notification, which is listened to by the GUI aplication in order to send a toast notification to the system, informing the user of the detection. The requests that the service can resolve are:

- start monitoring
- stop monitoring
- get detection history
- query current settings

5.3 Detection Flow

Based on the communication model presented above, I will present the flow of a detection, starting from the wslflt.sys kernel driver to the GUI Application and toast notification.

The kernel driver is reponsible for issuing the detection and stopping the malicious process. After doing this the user mode system service is informed through the filter port. The system service relays the detection information to the GUI application through the WCF endpoint. Finally, a toast notification is displayed in order to inform the user of the detection.



Bashware detected
Executable Name: testpath

Figure 5.1: Detection Notification

Clicking the notifications brings the application to front and selects the detections tabs, so that the user can see more information about the detection.

5.4 C++ Memory Allocation Framework in the Kernel

Allocating memory in the kernel is done via the ExAllocatePool API family (i.e. ExAllocatePoolWithQuotaTag, ExAllocatePool, ExAllocatePoolWithTag). These APIs have in common the POOL_TYPE parameter, which is used to specify the type of memory that the system should allocate. For example, PagedPool type would allocate pageable memory while NonPagedPoolNx would allocate non-pageable and non-executable memory. Pageable memory means that the memory could be paged-in/paged-out between the RAM and the secondary storage (hard-disk).

Usually, drivers use the ExAllocatePoolWithTag API to allocate memory in order to ease memory leak debugging. This API takes 3 parameters: the previously mentioned POOL_TYPE, the number of bytes to allocate and an allocation tag. This tag is used in order to keep track of memory allocations and allow the developer to mark each allocation as needed in order to track memory leaks.

The problem that rises is integrating this API with C++ ”new” operator. This can be done by defining a custom ”new” operator which takes the aditional parameters and overload the default ”new” operator to use the ExAllocatePoolWithTag API.

The additional ”new” operator would use a C++ specific concept called ”traits” [12]. A traits class is usually a small class that stores information about implementation details and policies and provides it to other objects or algorithms. These are usually used to provide meta information at compile time.

Example of traits class:

```
template <typename _Ty>
struct TypeTraits
{
    static constexpr ULONG Tag = WSLFLT_TAG_DEFAULT;
    static constexpr POOL_TYPE PoolType = PagedPool;
};

template<>
struct TypeTraits<StreamHandleContext>
{
    static constexpr ULONG Tag = WSLFLT_TAG_STREAMHANDLE_CONTEXT;
    static constexpr POOL_TYPE PoolType = PagedPool;
};
```

In this example, the templated TypeTraits class defines the properties it contains and provides default values for each property. Below the TypeTraits definition, there is a specialized definition for the StreamHandleContext class, for which we define the allocation tag and pool type we want to use throughout the driver.

This design helps in decoupling allocation logic and allows for implementing the C++ allocator concept. Essentially, whenever we need to allocate memory for a type T (be it template or not), we would call the ”new” operator as follows:

```
new(TypeTraits<T>::PoolType, TypeTraits<T>::Tag) T { parameter1, parameter2 };
```

5.5 Application Tracing and Instrumentation

In order to ease debugging and identifying issues in production code, it is important for the code to be properly traced with relevant and meaningful log messages, without easing the reverse engineering process by havin the log message strings in the executable. All of this can be achieved by using Windows software trace preprocessor (WPP), a framework based on Event Tracing for Windows. This way, log messages can't be decoded without having the traced application symbols, which are stored in pdb files.

WPP was used for both wslflt.sys and wslcore.dll, while the integration service create service event log named WSL Anti-Malware System Service. This log will hold information about WSL events and incidents that can be viewed by system administrators via the Event Viewer utility.

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

In order to use the Test Driven Development paradigm, a whitebox unit testing framework was required. The unit test project contains mock definitions of kernel functions (e.g. ExAllocatePoolWithTag), and unit tests 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 uni ttests 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.

In the context of the minifilter driver, whitebox testcases were implemented in order to verify that the Windows kernel APIs are correctly. These verifications include checks for handle leaks and correct error status handling. Moreover The driver's components (i.e. file filter, process collector) are tested individually to confirm that they work correctly. Also, generic framework classes like SharedPointer or LookasideObject are covered by unit tests as they are widely used throughout the project.

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. detection) and non-functional requirements (i.e. stability, security, reliability).

In order to identify memory or handle leaks, potential deadlocks, incorrect IRP handling, code integrity violation and other potential bugs, all blackbox tests are run with driver verifier enabled for wslft.sys. Using the driver verifier in the testing process helped in building a more robust and stable driver.

Most blackbox testcases were implemented as bash scripts running in WSL, but some more complex testcases were also implemented using powershell.

6.3 Static Code Analysis

Static code analysis is a technique used to find potential bugs in a software without actually running it. Most code analysis tools use either the source code itself or some form of intermediate form of the code, (i.e. object code).

SAL is a source code annotation language developed by Microsoft for C and C++. Its purpose is to clarify the intent behind the code, locking behavior or function parameters while also enabling static code analysis.

Below is an example of SAL annotated function:

```
_Must_inspect_result_
void * memcpy(
    _Out_writes_bytes_all_(count) void *dest,
    _In_reads_bytes_(count) const void *src,
    _In_ size_t count
);
```

_Must_inspect_result_- indicates that the return value of the function must be checked

_Out_writes_bytes_all_(count) - indicates an output parameter where we write exactly "count" bytes

_Out_read_bytes_(count) - indicates an input parameter from which we read exactly "count" bytes

During code analysis, all these annotations are checked and any anomaly (i.e. writing to an _In_ parameter) is reported as a compilataion warning.

6.4 WHQL Testing

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. These complex test suite contains extensive checks for reparse point handling, I/O stress conditions, transactional I/O, code integrity checks and much more for multiple file systems.

7. Heuristics and Detection Algorithms

The gathered information about a process' behavior is used by heuristic algorithms to identify potentially malicious activity. As the name implies, these algorithms are not 100% accurate and can result in false positives as well as false negatives. However, combining multiple heuristic detection algorithms has been proven to be an efficient detection technique.

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 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 is 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 considering 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 hypervisor memory introspection techniques, 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.

In order to detect if a process was maliciously elevated after it was started, we need to keep track of monitored active processes in the called process collector, storing their initial elevation status. If the process was initially not elevated, it should never become elevated.

Whenever a monitored process issues file operations for files in sensitive locations (i.e. C:\Windows), we will recheck the security token with the described algorithm. If the new token query reveals that the process is elevated, and initially it was not, then the process should be detected and stopped.

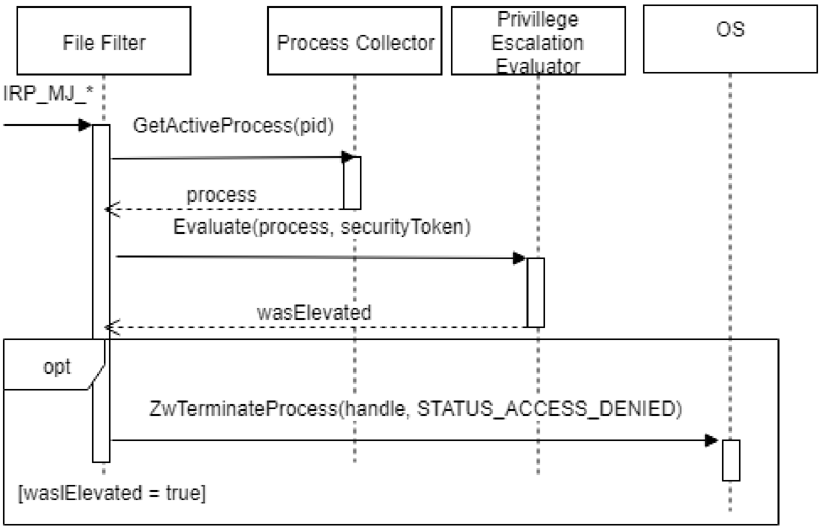

File Filter
Process Collector
Privillege Escalation Evaluator
OS
IRP_MJ_*
GetActiveProcess(pid)
process
Evaluate(process, securityToken)
wasElevated
opt
ZwTerminateProcess(handle, STATUS_ACCESS_DENIED)
[wasIElevated = true]

Figure 7.1: privillege escalation detection diagram

A more detailed explanation for the diagram can be found below:

1. for any IRP filtered by the file filter we get the process from the collector
2. we check if the process was elevated since it was started
3. we kill the process if the process was elevated

Even though the vulnerability that led to the execve exploit[1] was fixed by Microsoft, we can see that the described algorithm is a generic approach as it does not take into consideration any particularities in the previously mention exploit.

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, the safer decision would be to treat 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 a IRP_MJ_WRITE operation 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.

Moreover, detection could be improved by integrating next-gen anti-virus features, like AI based detection. Multiple solutions are possible and should be investigated, for example Markov Model and Markov Chains or Recurent Neural Networks. These could prove extremely efficient because most of the complexity is in the training of the model, while using the model in order to identify malicious actions would just follow a simple linear algorithm.

Finally, in order to more reliably and efficiently detect exploits in general, especially kernel exploits, memory introspection techniques could be implemented at hypervisor level.) Hypervisor perform virtualization without a host operating system between them and physicalsystem hardware[3]. The Xen hypervisor [2] has been used in several surveyed analysis systems.

For example, privilege escalation could by detecting by hooking memory access at process security tokens and making use of Extended Page Tables (EPT).

8.3 Reverting 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

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. Since any process with root privilleges (inside the linux system) can update the said config file, we must protect it from a kernel driver, so that we are sure that our shared library isn't removed. This can be done by filtering IRP_MJ_WRITE operations on that file and readding the entry if removed.

When loaded in a linux process, the library hooks the functions we want to monitor (i.e. ptrace, open) and synchronously notifies a service via local sockets about every hooked function call, informing the service about exploitation intent and failing the function call if needed.

We consider this method unreliable against smarter bashware that actively checks and avoids hooked functions, and possibly completely useless starting with Windows RS5 if executable memory can't be allocated anymore.

9. Conclusion

The reasearch of Windows Subsystem for Linux revealed an attack surface that, even though has not been seen yet to be exploited in the wild by actual malware, is a security issue that needs to be addressed and anti-malware solutions should be updated to take the Linux subsystem into consideration.

I have addressed these issues by developing a complete product that can offer basic protection against the described security issues by making use of behavioral detection heuristic algorithms and by improving these algorithms. The proposed solution, being based on a minifilter driver as the core component which encapsulates the monitoring and detection logic, is, as I have proven, a reliable method to cover the new attack surface and defend against basic bashware.

Finally, the possible directions in which research can be continued in order to improve efficiency and detection provide a starting point for further research and innovation in strengthening the security of WSL, making it safer for the end user.

References

[1] Amar S. *execve exploit.* https://github.com/saaramar/execve_exploit. 2018.

[2] Barham, P. Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A. "Xen and the Art of Virtualization". In: *SIGOPS Oper. Syst. Rev.* 37.5 (Oct. 2003), pp. 164–177. ISSN: 0163-5980. DOI: 10.1145/1165389.945462. URL: http://doi.acm.org/10.1145/1165389.945462.

[3] Bulazel, A., Yener,B. "A Survey On Automated Dynamic Malware Analysis Evasion and Counter-Evasion: PC, Mobile, and Web". In: *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium.* ROOTS. Vienna, Austria: ACM, 2017, 2:1–2:21. ISBN: 978-1-4503-5321-2. DOI: 10.1145/3150376.3150378. URL: http://doi.acm.org/10.1145/3150376.3150378.

[4] Egele, M., Scholte, T., Kirda, E., Kruegel, C. "A Survey on Automated Dynamic Malware Analysis Techniques and Tools". In: *ACM Computing Surveys* 44.6 (Feb. 2012). DOI: 10.1145/2089125.2089126\url{http://doi.acm.org/10.1145/2089125.2089126}.

[5] Hammons J. "Windows Subsystem for Linux Overview". https://blogs.msdn.microsoft.com/wsl/2016/04/22/windows-subsystem-for-linux-overview/. 2016.

[6] Hammons J. "WSL File System Support". https://blogs.msdn.microsoft.com/wsl/2016/06/15/wsl-file-system-support/. 2016.

[7] Ionescu A. "GAINING VISIBILITY INTO LINUX BINARIES ON WINDOWS: DEFEND AND UNDERSTAND WSL". http://www.alex-ionescu.com/publications/BlueHat/bluehat2016.pdf. 2016.

[8] Ionescu A. "THE LINUX KERNEL HIDDEN INSIDE WINDOWS 10". http://www.alex-ionescu.com/publications/blackhat/blackhat2016.pdf. 2016.

[9] Jacob, G., Debar, H., Filiol, E. "Behavioral detection of malware: from a survey towards an established taxonomy". In: *Journal in computer Virology* 4.3 (2008), pp. 251–266. DOI: https://doi.org/10.1007/s11416-008-0086-0.

[10] Ligh, M.H., Case A., Levy, J., Walters, A. *The Art of Memory Forensics.* Indianapolis: John Wiley & Sons, Inc., 2014.

[11] Martin R.C. *Clean Code: A Handbook of Agile Software Craftsmanship.* Boston: Pearson Education, Inc, 2009.

[12] Meyers S. *Effective Modern C++.* Sebastopol: O'Reilly, 2015.

[13] Moser, A., Kruegel, C., Kirda, E. “Exploring multiple execution paths for malware analysis”. In: *Proceedings of the IEEE Symposium on Security and Privacy.* Berkeley: IEEE, 2007, p. 1.

[14] Stroustrup B. *The C++ Programming Language.* Third Edition. Murray Hill, New Jersey: Addison-Wesley, 1997.

[15] Yosifovich, P., Ionescu, A., Russinovich, M.E., Russinovich, D.A. *Windows Internals, Part 1.* Sixth Edition. Redmond: Microsoft Press, 2012.

[16] Yosifovich, P., Ionescu, A., Russinovich, M.E., Russinovich, D.A. *Windows Internals, Part 1: System architecture, processes, threads, memory management, and more.* Seventh Edition. Redmond: Microsoft Press, 2017.

[17] Yosifovich, P., Ionescu, A., Russinovich, M.E., Russinovich, D.A. *Windows Internals, Part 2.* Sixth Edition. Redmond: Microsoft Press, 2012.