國立臺灣大學電機資訊學院電機工程學系資訊安全碩士班

碩士論文

Master Program of Cybersecurity

Department of Electrical Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

使用符號執行和污點分析增強搜索 WDM 驅動程式漏洞

IOCTLance: Enhanced Vulnerability Hunting in WDM Drivers Using Symbolic Execution and Taint Analysis

林哲宇

Che-Yu Lin

指導教授: 雷欽隆 博士

Advisor: Chin-Laung Lei, Ph.D.

中華民國 112 年 8 月

August, 2023

doi:10.6342/NTU202302404

國立臺灣大學碩士學位論文
口試委員會審定書
MASTER'S THESIS ACCEPTANCE CERTIFICATE
NATIONAL TAIWAN UNIVERSITY

使用符號執行和污點分析增強搜索 WDM 驅動程式漏洞

IOCTLance: Enhanced Vulnerability Hunting in WDM Drivers Using Symbolic Execution and Taint Analysis

本論文係___林哲宇___(姓名)__R10921A04__（學號）在國立臺灣大學電機工程學系完成之碩士學位論文，於民國_112_年_07_月_17_日承下列考試委員審查通過及口試及格，特此證明。

The undersigned, appointed by the Department of Electrical Engineering on 17 (date) 07 (month) 2023 (year) have examined a Master's thesis entitled above presented by__LIN, CHE-YU__ (name)____R10921A04___ (student ID) candidate and hereby certify that it is worthy of acceptance.

口試委員 Oral examination committee:

雷欽隆 （指導教授 Advisor）	郭斯彥	[illegible]

系主任 Director: 李建模

doi:10.6342/NTU202302404

doi:10.6342/NTU202302404

Acknowledgements

我要由衷地感謝以下人士對我碩士學業的支持與陪伴，使我能順利完成這段學術旅程：

首先，我要感謝我的爸爸和媽媽。你們無私的支持和鼓勵讓我充滿信心地追求我的學術目標。你們的無私奉獻和家庭的陪伴是我學業成功的重要基石。

接著，我要感謝我的導師和教授們。你們對我的教導和論文的指導使我獲益良多，我在學術研究和專業知識上有了長足的成長。你們的激勵和指導對我影響深遠，我將永遠珍惜這段寶貴的學習經驗。

我也要特別感謝 TeamT5 的 Kenny。在我實習和學習 Windows 系統安全的過程中，你給予我寶貴的幫助和指導，使我能夠克服困難並獲得實際經驗。你的專業知識和熱心幫助對我來說意義重大。

此外，我要感謝教育部提供的 AIS3 和台灣好厲駭計畫。這些計畫不僅提供了學習資源和平台，也讓我有機會參與各種安全相關活動和競賽。這些經驗讓我能夠不斷學習和深化對安全領域的理解。

特別感謝 HITCON 活動組的組員們，你們一起陪伴我成長，並提供了一個充滿挑戰和激勵的環境。你們的專業精神和團隊合作精神讓我受益匪淺。

同樣地，我要感謝 BambooFox 和 XxTSJxX CTF 戰隊的隊員們。在與你們一

doi:10.6342/NTU202302404

起切磋學習的過程中，我獲得了寶貴的技術知識和解決問題的能力。你們的激情和努力激發了我對資訊安全領域的熱情。

最後，我要感謝我的朋友們。在我碩士班的這兩年裡，你們的陪伴和支持給予我無限的力量和鼓勵。我感激你們一直在我身邊，給予我情感上的支持和忠告。

再次向以上所有人致以最衷心的感謝，沒有你們的幫助和支持，我無法取得這樣的成就。感謝你們一路上的陪伴和鼓勵。

doi:10.6342/NTU202302404

摘要

發現 WDM 驅動程式的資安漏洞很困難，因為它們大多不是開源的，有些驅動程式甚至需要指定的環境才能將它們加載到系統核心中。符號執行和污點分析是軟體安全中常用的技術，用於識別程式中的漏洞。然而，符號執行可能會出現「路徑爆炸「問題，當程式複雜度增加時，可能的程式路徑數量呈指數級增長。污點分析也可能會出現「污點爆炸」問題，當程式複雜度增加時，可能被污染的輸入數量呈指數級增長。

本研究提出了一種名為 IOCTLance 的解決方案，它利用符號執行和污點分析來檢測 WDM 驅動程式中的漏洞。通過將目標輸入緩衝區從用戶模式進程標記為「污點」，IOCTLance 能夠檢測各種漏洞類型，例如「映射物理內存」、「可控進程句柄」、「緩衝區溢出」、「空指針引用」、「可讀/可寫可控地址」、「任意 shellcode 執行」、「任意 wrmsr」、“任意 out」以及「危險的文件操作」。此外還開發了幾個可調整的選項，以解決符號執行中的「路徑爆炸「問題。將 IOCTLance 應用於 104 個已知有漏洞的驅動程式上，在其中 22 個驅動程式中發現了 117 個未知的漏洞，目前已回報並取得 41 個 CVE，其中包括 25 個拒絕服務，5 個訪問權限控制不足以及 11 個提權漏洞。

關鍵字： Windows 核心、符號執行、汙點分析、漏洞

doi:10.6342/NTU202302404

doi:10.6342/NTU202302404

Abstract

Discovering the security vulnerabilities of WDM drivers is challenging because most of them are not open-source and some drivers even need the specified environment to load them into the kernel. Symbolic execution and taint analysis are common techniques used in software security to identify vulnerabilities in software. However, symbolic execution can suffer from the ”path explosion” problem, where the number of possible paths through a program grows exponentially as the program complexity increases. Taint analysis can also suffer from the ”taint explosion” problem, where the number of potentially tainted inputs grows exponentially as the program complexity increases.

This research paper presents a solution called IOCTLance that aims to detect vulnerabilities in WDM drivers using symbolic execution and taint analysis. By marking the target input buffer from the user mode process, IOCTLance is able to detect various vulnerability types, such as ”map physical memory”, ”controllable process handle”, ” buffer overflow”, ”null pointer dereference”, ”read/write controllable address”, ”arbitrary

doi:10.6342/NTU202302404

shellcode execution”, ”arbitrary wrmsr”, ”arbitrary out”, and ”dangerous file operation”. Several customizable options have also been developed to improve the performance while symbolic execution. IOCTLance is evaluated on 104 known vulnerable WDM drivers and 318 unknown WDM drivers and discovered 117 previously unknown vulnerabilities in 26 unique drivers, resulting in 41 CVEs, including 25 denial of service, 5 insufficient access control, and 11 elevation of privilege vulnerabilities.

Keywords: Windows Kernel, Symbolic Execution, Taint Analysis, Vulnerability

doi:10.6342/NTU202302404

Contents

doi:10.6342/NTU202302404

doi:10.6342/NTU202302404

doi:10.6342/NTU202302404

doi:10.6342/NTU202302404

List of Figures

doi:10.6342/NTU202302404

doi:10.6342/NTU202302404

List of Tables

doi:10.6342/NTU202302404

doi:10.6342/NTU202302404

Denotation

BYOVD	Bring Your Own Vulnerable Driver
BSOD	Blue Screen of Death
WDM	Windows Driver Model
IOCTL	Input/Output Control
IRP	I/O Request Packet

doi:10.6342/NTU202302404

doi:10.6342/NTU202302404

Chapter 1 Introduction

Since Windows 2000, Microsoft has recommended that hardware device manufacturers use WDM (Windows Driver Model) [20] drivers to provide basic support for devices. It plays a crucial role in the operation of Windows-based systems, as they act as intermediaries between hardware devices and the operating system. As a result, WDM drivers account for most of the Windows kernel drivers in the market. However, as with any software, WDM drivers are not immune to security vulnerabilities. Over the years, many security vulnerabilities have been discovered in WDM drivers that could be exploited by attackers to gain unauthorized access, execute arbitrary code, or elevate privileges. These vulnerabilities pose a significant threat to the security of Windows-based systems and can lead to system crashes, data breaches, and even full system compromise.

BYOVD (Bring Your Own Vulnerable Driver) is a type of attack on the Windows operating system that can be used by an attacker. This attack involves loading a vulnerable driver with elevated privileges into the kernel, enabling the attacker to gain access to sensitive system resources or execute arbitrary kernel code. The BlackByte [1] cyber attack abused the ”read/write controllable address” vulnerability in RTCore.sys driver to bypass antivirus software, granting full kernel-level privileges, then executing ransomware. Similarly, Candiru [13] exploited the ”map physical memory” vulnerability in the HW.sys driver, used for custom hardware communication, to install a rootkit and spyware on in-

doi:10.6342/NTU202302404

fected systems.

To tackle these challenges, we introduce a system named IOCTLance, which applies symbolic execution and taint analysis to detect vulnerabilities in WDM drivers. One of the primary obstacles of symbolic execution is the path explosion problem, where the number of feasible paths through a program grows exponentially with the program's complexity. To address this issue, IOCTLance intercepts opcodes and functions, sets breakpoints, and employs several methods to minimize the program's complexity for simulation. In conclusion, this paper contributes by proposing the following enhancements.

1. By extending the symbolic execution engine, IOCTLance enhances its ability to detect additional vulnerability types in WDM drivers.

2. We conducted an evaluation of IOCTLance by testing various options on a test dataset of 104 known vulnerable WDM drivers and 328 unknown WDM drivers, enabling us to statically and efficiently identify vulnerabilities at scale.

3. During our evaluation, IOCTLance successfully detected 117 previously unknown vulnerabilities across 26 distinct drivers. As a result, we were able to report and receive 41 CVEs, including 25 instances of denial of service, 5 cases of insufficient access control, and 11 examples of elevation of privilege.

The rest of this paper is organized as follows. Chapter 2 provides background information on the WDM driver, symbolic execution, and taint analysis. Chapter 3 describes the design of IOCTLance, including the preprocessing steps, setting breakpoints, and reporting vulnerabilities. Chapter 4 discusses several target vulnerability types that the tool can detect. Chapter 5 of the paper illustrates the implementation of the symbolic execution engine, while Chapter 6 evaluates the performance of IOCTLance. In Chapter 7

doi:10.6342/NTU202302404

explores various options for customizing the analysis process and examine real-world scenarios encountered during the search for vulnerabilities. Finally, Chapter 8 presents the conclusions of this paper.

doi:10.6342/NTU202302404

doi:10.6342/NTU202302404

Chapter 2 Background

2.1 WDM Driver

WDM (Windows Driver Model) is a driver framework introduced by Microsoft for Windows 2000 and later versions of Windows operating systems. WDM provides a layered message-passing architecture for Windows drivers, allowing them to communicate with the operating system kernel and peripheral devices in a standardized way. WDM drivers are binary-compatible and can run on multiple versions of the Windows operating system, making it easier for hardware manufacturers to develop and deploy drivers for their devices. WDM drivers can be developed using C or C++ programming languages, and they interact with the operating system through the DDI (Device Driver Interface) provided by Windows.

2.1.1 Driver Object

The driver object [40] is a data structure in the WDM driver that represents a loaded driver in the system and contains various information about the driver, including its entry points, driver extension, and device object list.

When a driver is loaded into the kernel, the I/O manager [23] creates a driver object

doi:10.6342/NTU202302404

for it. The driver object serves as an entry point for the driver and contains pointers to the driver's DriverEntry, AddDevice, Unload, and dispatch routines handling I/O requests that are directed to the driver's devices.

The driver object also contains a pointer to the driver's device object list. Each device object represents a physical or logical device that the driver controls. The device object list is used by the system to manage devices and to route I/O requests to the appropriate driver.

Additionally, the driver object may contain a driver extension, which is a structure that can be used by the driver to store additional driver-specific data. The driver extension is allocated by the driver itself and is typically used to store global driver state or other driver-specific data that is not associated with a particular device object.

2.1.2 Device Object

In the WDM driver framework, a device object represents a physical or logical device that is managed by the driver. Device objects are created by the driver during initialization using the IoCreateDevice [45] function. The device object is associated with the driver object and contains a set of device-specific data structures and information.

The device object provides a way for the driver to communicate with the system and other drivers, and it serves as the interface between the driver and the operating system. When the system sends a request to the driver, it sends the request to the device object associated with the request. The device object then forwards the request to the driver's dispatch routines [25], which are responsible for processing the request and returning a response.

doi:10.6342/NTU202302404

2.1.3 IRP

IRP (I/O Request Packet) [29] is a fundamental data structure used in Windows operating systems to communicate between various drivers and the operating system. In a WDM driver, the IRP is a data structure that represents an I/O request made by a user mode application or by another driver in the system.

Figure 2.1 illustrates how to send IRP from a user-mode process to a kernel-mode driver. When a user-mode application sends an I/O request to a device driver, the I/O Manager creates an IRP and populates it with information about the request, such as the type of request, the data buffer, and the target device object. The IRP is then passed down through the driver stack to the appropriate driver for processing.

Each driver in the driver stack can examine and modify the IRP as it passes through, and perform the necessary processing for the particular I/O request. After processing is complete, the driver may either pass the IRP down the stack to the next driver or complete the IRP and return it to the I/O Manager.

There are several members in IRP that IOCTLance concerns, including SystemBuffer, InputBufferLength, OutputBufferLength, UserBuffer, Type3InputBuffer, and MajorFunction [49].

SystemBuffer is a pointer to the kernel buffer that contains the data associated with the I/O request. For example, if the IRP is associated with a read request, the data that is being read from the device will be stored in the SystemBuffer.

InputBufferLength indicates the length of the input buffer associated with the I/O request. The input buffer is typically used to pass data from user mode to kernel mode.

doi:10.6342/NTU202302404

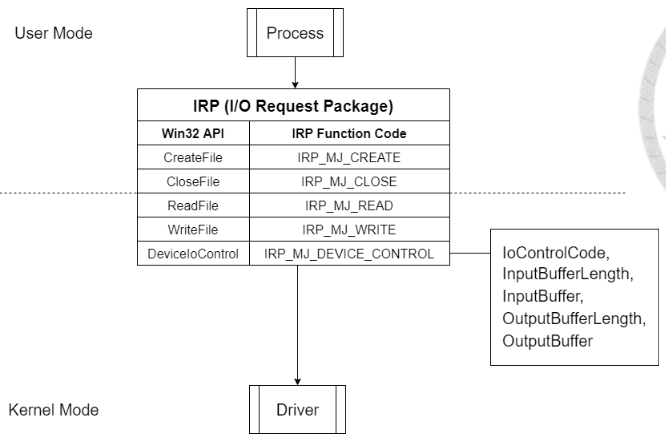

Figure 2.1: Send IRP from a user-mode process to a kernel-mode driver.

OutputBufferLength indicates the length of the output buffer associated with the I/O request. The output buffer is typically used to pass data from kernel mode to user mode.

UserBuffer is a pointer to the user buffer that contains the data associated with the I/O request. This field is only relevant if the I/O request originated in user mode.

Type3InputBuffer is only used for method buffered I/O requests. It contains a pointer to the input buffer associated with the I/O request.

MajorFunction in the IRP specifies the type of I/O operation that is being requested by the I/O Manager. This field can have values such as IRP_MJ_CREATE [51], IRP_MJ_CLOSE [50], IRP_MJ_READ [53], IRP_MJ_WRITE [54], IRP_MJ_DEVICE_CONTROL [52], etc. The driver's dispatch routine processes the IRP based on its MajorFunction value.

2.1.4 IOCTL Handler

In a WDM driver, an IOCTL (I/O control) [18] handler is a function that handles a specific input/output control request from an application. When an application sends an IOCTL request to a device object, the request is passed down to the driver's dispatch

doi:10.6342/NTU202302404

routine, which then routes the request to the appropriate IOCTL handler.

The IOCTL handler is responsible for processing the IOCTL request, which typically involves reading or writing data to or from a device, setting or retrieving device configuration parameters, or performing some other specific operation on the device. The handler uses the input and output buffers provided by the application to exchange data with the device.

2.2 Symbolic Execution

Symbolic Execution is a technique used in software engineering and computer security to analyze software code without having to run it on a real system. It involves creating a mathematical model of a program and executing the program symbolically by assigning symbolic values to its input variables. The symbolic execution engine then tracks how these values propagate through the program and how they affect program behavior, including control flow, data flow, and memory access.

Symbolic execution is useful for identifying bugs and security vulnerabilities in software, as it can systematically explore all possible paths through the program and generate test cases that trigger edge cases that might be missed in traditional testing.

2.3 Taint Analysis

Taint analysis is a technique used in software security to identify and track potentially unsafe inputs that may be used to exploit a program. It involves tracing the flow of data through a program and marking any data that has originated from an untrusted source as ”

doi:10.6342/NTU202302404

tainted". This can include data from user input, network communication, or other external sources.

Once the data has been marked as tainted, the taint analysis tool can track its propagation through the program and identify any points where the tainted data may be used in an unsafe or unintended way.

2.4 Related Work

Several tools exist for fuzzing Windows kernel drivers, such as ioctlfuzzer [8], ioctlbf [15], iofuzz [7], and Microsoft's IoAttack [26]. However, these tools are limited in their ability to provide comprehensive coverage and insights beyond kernel interface return values. Additionally, the implementation of these tools is relatively straightforward, and they cannot analyze code in-depth.

CAB-FUZZ [14] gives priority to the boundary states of symbolic memories and loops because they have the potential to cause stack or heap overflows and underflows. Several studies have attempted to specialize in symbolic execution to detect overflows and underflows. For example, IntScope [67] and SmartFuzz [64] use symbolic execution to detect integer overflows, with SmartFuzz also covering integer underflows, narrowing conversions, and signed/unsigned conversions. Dowser [11] focuses on identifying overflows and underflows in a buffer within a loop. Conversely, DIODE [66] concentrates on identifying integer overflow errors at specific memory locations using a detailed dynamic taint analysis that identifies all memory allocation sites, extracts target and branch constraints from instrumented execution, solve the constraints, and executes goal-directed conditional branch enforcement. On the contrary, IOCTLance can directly perform symbolic analysis

doi:10.6342/NTU202302404

on kernel driver binaries without requiring the environment to load the driver.

Other tools and frameworks, such as Screwed-Drivers [9] and POPKORN [10], have been developed to identify and exploit vulnerabilities in Windows kernel drivers. These tools employ symbolic execution and taint analysis to detect opcodes and kernel APIs that are likely to be exploited. However, these tools do not address the path explosion issue in symbolic execution and are limited to targeting API-type and opcode-type vulnerabilities only. In contrast, IOCTLance targets a wider range of API-type, opcode-type, and memory-type vulnerabilities. Additionally, IOCTLance reduces both false positives and false negatives and improves the performance of the symbolic execution engine.

doi:10.6342/NTU202302404

doi:10.6342/NTU202302404

Chapter 3 Design

IOCTLance leverages symbolic execution and taint analysis to detect vulnerabilities in WDM drivers and consists of the following steps: information gathering, preprocess, find IOCTL handler, hunt vulnerabilities, and report. Figure 3.1 shows the design of IOCTLance.

3.1 Information Gathering

To facilitate communication with other kernel modules, a WDM driver utilizes IoCreateDevice to register a device. IoCreateDevice is a function provided by the Windows Driver Model that is used to create a new device object within a driver. Device objects represent physical or virtual devices that the driver manages. As a result, it is essential to verify the presence of IoCreateDevice in the import table.

IOCTLance uses a string signature to determine the target device name and symbolic link name created by IoCreateSymbolicLink [28] which are used to connect to WDM drivers from the user mode process. With this information, we can use it to write the PoC to exploit the target vulnerable drivers.

IOCTLance parses WDM drivers to get information such as opcodes, function ad-

doi:10.6342/NTU202302404

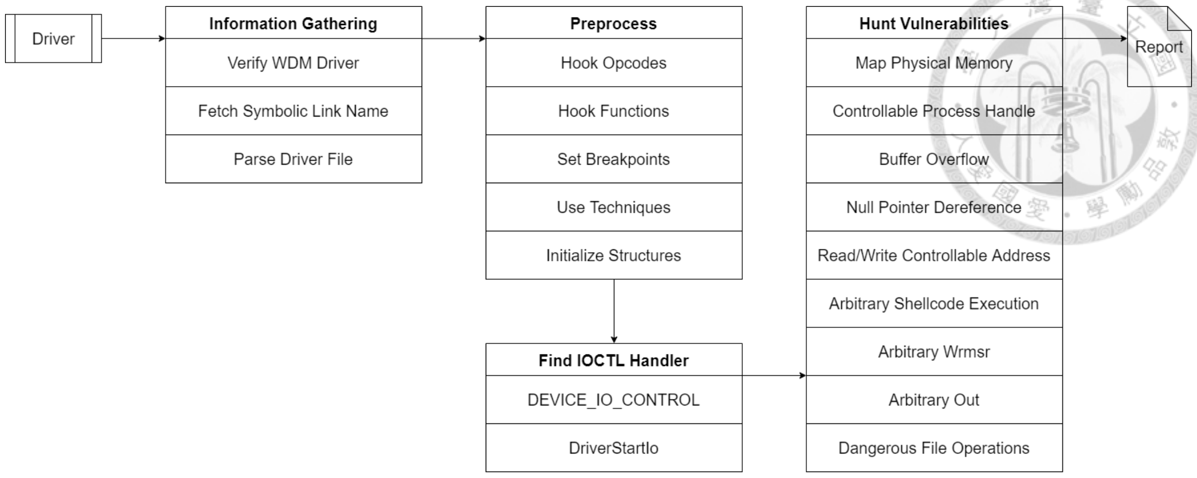

Figure 3.1: IOCTLance Design

dresses, addresses of the indirect jump, etc. With the information, we can know where to hook and avoid to improve the performance of symbolic execution.

3.2 Preprocess

Prior to simulation, IOCTLance prepares the symbolic execution engine environment to ensure smoother simulation. This involves opcode and function hooking as well as breakpoint setting.

3.2.1 Hook Opcodes

To enhance the quality of symbolic execution, IOCTLance hooks specific opcodes that may hinder the simulation. Additionally, in some cases, symbolic variables are concretized for certain instructions to prevent issues that may cause the simulation to get stuck.

Moreover, IOCTLance also hooks opcodes that fall under the target vulnerability types in our scope. To minimize the occurrence of false positives and false negatives, we

doi:10.6342/NTU202302404

have incorporated validation within the hook function. This involves setting constraints on operands and checking if the state is satisfiable.

3.2.2 Hook Functions

Angr does not implement most of the kernel APIs, but instead, it summarizes the functions and returns a symbolic variable. This may lead to a loss of semantics during symbolic execution and restricts code coverage because some parameters of kernel APIs are passed by reference and get their values inside the APIs. As a result, there is a possibility of encountering false positives and false negatives. To address this issue, IOCTLance hooks these kernel APIs and provides appropriate values or symbolic variables to the parameters that are expected to be assigned inside the APIs and are passed by reference.

Furthermore, many vulnerabilities stem from kernel APIs. In order to detect these types of vulnerabilities, IOCTLance hooks the kernel APIs that are of interest to us. The system then verifies the safety of the usage of a kernel API within the hook function.

Moreover, certain functions like memset and memcpy, which are built-in during code writing, become highly complex after compilation, leading to performance degradation during symbolic execution. In order to enhance performance without compromising the original program's semantics, IOCTLance hooks such functions and implements them internally.

3.2.3 Set Breakpoints

IOCTLance sets breakpoints on mem_read and mem_write to trace the tainted buffer and the buffer it points to. This helps detect some target vulnerability types, such as

doi:10.6342/NTU202302404

null pointer dereference and read/write controllable address. In the next section, ”Find IOCTL Handler,” the mem_write breakpoint is also used to identify the target value in the DRIVER_OBJECT structure when it is written into the function pointer.

IOCTLance has implemented a custom function that is executed whenever the symbolic execution engine simulates the call instruction, which is triggered by a breakpoint. This breakpoint is useful for detecting situations where the target function address is either a symbolic variable or a tainted buffer that is controlled by an attacker. In such scenarios, the call breakpoint can detect arbitrary shellcode execution by handling these situations appropriately.

3.2.4 Use Techniques

IOCTLance offers various options for customizing the analysis of drivers. These options can be adjusted to better suit different situations. Limits can be set for the length of instructions that a state can execute, the number of times a state can repeat a loop and the number of times a state can recurse. There are also limits on the total time allowed for symbolic execution and the time allowed for each IoControlCode in the ”Hunt Vulnerabilities” phase.

3.2.5 Initialize Structures

IOCTLance simplifies access to common kernel structures in WDM drivers by registering them. These structures must be initialized before symbolic execution. In the first phase of the analysis, called Find IOCTL Handler, IOCTLance symbolizes two parameters, namely DRIVER_OBJECT and REGISTRY_PATH, in DriverEntry. This allows the

doi:10.6342/NTU202302404

tool to bypass parameter checks and obtain the function pointer of the IOCTL handler. In the second phase, known as Hunt Vulnerabilities, IOCTLance initializes parameters such as DeviceObject and IRP in the IOCTL handler. To trace and identify vulnerabilities, IOCTLance symbolizes certain members, such as SystemBuffer, Type3InputBuffer, UserBuffer, InputBufferLength, OutputBufferLength, and IoControlCode, as tainted buffers. In addition, some other members are also initialized to a concretized value to pass certain checks in the program. The data section can also be symbolized on a case-by-case basis, as needed.

3.3 First Phase - Find IOCTL Handler

IOCTLance begins searching for the IOCTL handler after setting up the symbolic engine environment, starting from DriverEntry. The simulation begins by examining the mem_write action on MajorFunction's DEVICE_IO_CONTROL in DRIVER_OBJECT. Additionally, some drivers have their logic in DriverStartIo [41], so we also inspect the mem_write action on DriverStartIo in DRIVER_OBJECT. After locating the IOCTL handler, IOCTLance searches for a state that returns from DriverEntry with NT_SUCCESS, which becomes the base state for identifying vulnerabilities.

3.4 Second Phase - Hunt Vulnerabilities

The next step for IOCTLance is to search for vulnerabilities starting from the IOCTL handler, which was identified in the previous step. The symbolic execution engine will analyze the program and execute the hooked functions and opcodes, breaking on the inspected actions. When a vulnerability condition is satisfied, IOCTLance will generate a

doi:10.6342/NTU202302404

JSON-formatted report containing details including the vulnerability's title, description, state when it was detected, parameters, and return address. This information is logged to help with manual vulnerability confirmation.

To obtain a better understanding of the vulnerability-hunting process, IOCTLance evaluates the values of SystemBuffer, Type3InputBuffer, UserBuffer, InputBufferLength, and OutputBufferLength, which are all potentially controlled by a user mode process. Additionally, we consider the constraints imposed by ProbeForRead [34], ProbeForWrite [35], and MmIsAddressValid [57] to minimize the likelihood of false positives.

3.5 Report

Once IOCTLance has generated a list of vulnerabilities, it applies deduplication to eliminate duplicates. The IoControlCode and vulnerability type are used to determine if a vulnerability is unique or not. Additionally, IOCTLance calculates the time taken and memory consumed for finding the IOCTL handler and hunting vulnerabilities, respectively. To calculate the code coverage, it unionizes the history of all states and gets the length of unique executed addresses.

The information on the hunted vulnerabilities is added to a list and included in the report, containing the title of the vulnerability type, the description of the vulnerability, the evaluation of the input buffer, the address where the vulnerability occurs, the limitation set by kernel API, and the return address. Any errors that occurred during the program simulation are also logged in the report to facilitate debugging and adjustment of IOCTLance's symbolic engine.

doi:10.6342/NTU202302404

Chapter 4 Target Vulnerability Types

The following section outlines the vulnerability types within the scope of our investigation, including ”map physical memory”, ”controllable process handle”, ”buffer overflow”, ”null pointer dereference”, ”read/write controllable address”, ”arbitrary shellcode execution”, ”arbitrary wrmsr”, ”arbitrary out”, and ”dangerous file operation”. Figure 4.1 shows the relationships between tainted buffer and target vulnerability types.

4.1 Map Physical Memory

IOCTLance intercepts the kernel APIs MmMapIoSpace [58], MmMapIoSpaceEx [31], and ZwMapViewOfSection [62], which are used by drivers to map physical memory for hardware register access, memory optimization, and direct memory access. The symbolic execution engine simulates these functions to detect vulnerabilities.

4.1.1 MmMapIoSpace and MmMapIoSpaceEx

The vulnerability type we are focusing on concerns the parameters PhysicalAddress and NumberOfBytes of the MmMapIoSpace and MmMapIoSpaceEx kernel APIs. The PhysicalAddress parameter specifies the physical address to be mapped, while NumberOfBytes parameter specifies the length of the physical memory to be mapped. If an attacker

doi:10.6342/NTU202302404

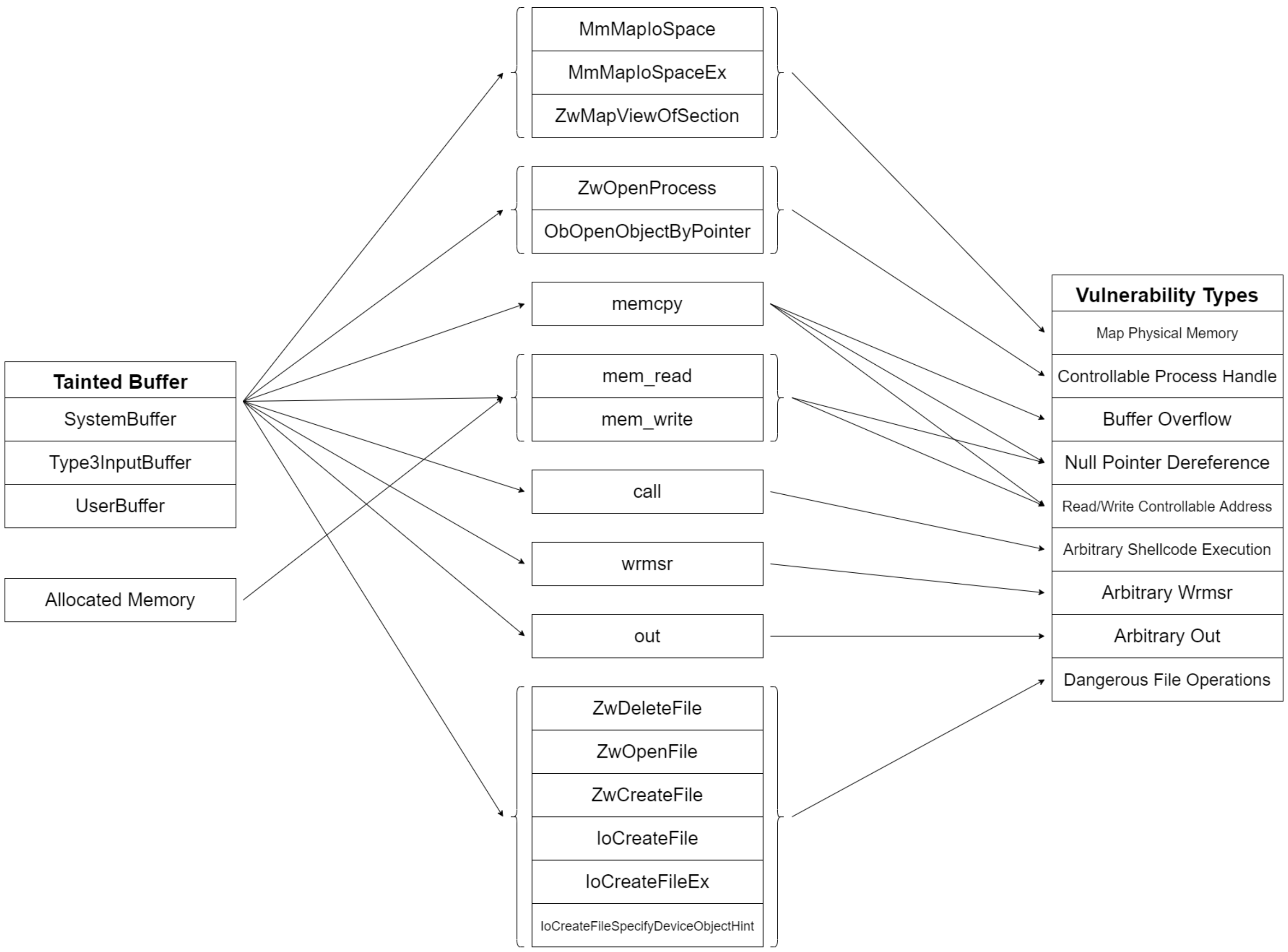

Figure 4.1: Relationships between Target Vulnerability Types and Tainted Buffer

doi:10.6342/NTU202302404

can control either of these parameters, it may allow for arbitrary kernel memory mapping and result in an elevation of privilege vulnerability. Listing 4.1 illustrates an instance that is deemed vulnerable by IOCTLance, since an attacker can manipulate the PhysicalAddress parameter in MmMapIoSpace.

```
VOID Vuln_MmMapIoSpace(PVOID inbuf)
{
    if (!inbuf) return;
    MmMapIoSpace(*(PHYSICAL_ADDRESS *)inbuf, 4, MmNonCached);
}
```

Listing 4.1: Map Physical Memory - MmMapIoSpace

4.1.2 ZwMapViewOfSection

ZwMapViewOfSection is another kernel API that is inspected by this type of vulnerability. It takes four parameters: SectionHandle, BaseAddress, CommitSize, and ViewSize. Among these, the vulnerability focuses on SectionHandle, BaseAddress, CommitSize, and ViewSize. SectionHandle is the handle of the section to be mapped, while BaseAddress is the starting virtual address for the allocated view. CommitSize indicates the committed region of the view, and ViewSize is the size of the view. If an attacker can control SectionHandle or use \Device\PhysicalMemory and can also control BaseAddress or CommitSize and ViewSize, they can map any arbitrary physical memory address, potentially leading to privilege escalation. Listing 4.2 portrays a scenario that is considered vulnerable by IOCTLance due to the attacker's ability to manipulate the SectionHandle and BaseAddress parameters in MmMapIoSpace.

```
VOID Vuln_ZwMapViewOfSection(PVOID inbuf)
{
    if (!inbuf) return;
    ZwMapViewOfSection(
```

doi:10.6342/NTU202302404

```
            *(HANDLE *)inbuf,
            ZwCurrentProcess(),
            *(PVOID **)(inbuf + 8),
            0,
            1000,
            0,
            1000,
            (SECTION_INHERIT)1,
            MEM_RESERVE,
            PAGE_READWRITE
        );
}
```

Listing 4.2: Map Physical Memory - ZwMapViewOfSection

4.2 Controllable Process Handle

The target process handle is a parameter passed before executing process operations. If an attacker can control the process handle passed as a parameter, it could result in potential vulnerabilities. IOCTLance intercepts the kernel APIs ZwOpenProcess [39] and ObOpenObjectByPointer [32] which can be used to get process handle.

4.2.1 ZwOpenProcess

This type of vulnerability focuses on the parameters of ZwOpenProcess, namely ObjectAttributes, and ClientId. ObjectAttributes is used to specify the attributes to apply, while ClientId identifies the thread whose process is to be opened. If these parameters can be controlled by an attacker, it may lead to vulnerabilities.

When the member Attributes in ObjectAttributes is not OBJ_FORCE_ACCESS_CHECK

doi:10.6342/NTU202302404

and ProcessId in ClientId is manipulable, it becomes possible for an attacker to return an arbitrary process handle, which can result in unauthorized access control. The example in Listing 4.3 is considered vulnerable by IOCTLance, as the attacker can manipulate the UniqueProcess in CLIENT_ID parameter of ZwOpenProcess. Moreover, the ObjectAttributes parameter is not set to OBJ_FORCE_ACCESS_CHECK.

```
VOID Vuln_ZwOpenProcess(PVOID inbuf)
{
    if (!inbuf) return;
    OBJECT_ATTRIBUTES objAttr;
    CLIENT_ID clientId;
    HANDLE processHandle;
    InitializeObjectAttributes(
        &objAttr,
        NULL,
        OBJ_KERNEL_HANDLE,
        NULL,
        NULL
    );

    clientId.UniqueProcess = *(HANDLE *)inbuf;
    clientId.UniqueThread = (HANDLE)0;
    ZwOpenProcess(
        &processHandle,
        PROCESS_ALL_ACCESS,
        &objAttr,
        clientId
    );
}
```

Listing 4.3: Controllable Process Handle - ZwOpenProcess

doi:10.6342/NTU202302404

4.2.2 ObOpenObjectByPointer

If the parameter ”Object” passed to ObOpenObjectByPointer is a pointer to EPROCESS and can be controlled by an attacker, it may result in the attacker obtaining an arbitrary process handle and may lead to unauthorized access control. IOCTLance considers Listing 4.4 as vulnerable because an attacker can manipulate the parameter ProcessId in PsLookupProcessByProcessId and pass it into ObOpenObjectByPointer.

```
VOID Vuln_ObOpenObjectByPointer(PVOID inbuf)
{
    if (!inbuf) return;
    PEPROCESS p;
    HANDLE h;
    PsLookupProcessByProcessId(*(HANDLE *)inbuf, &p);
    ObOpenObjectByPointer(
        p,
        OBJ_KERNEL_HANDLE,
        NULL,
        KEY_ALL_ACCESS,
        (POBJECT_TYPE)PsProcessType,
        0,
        &h
    );
}
```

Listing 4.4: ObOpenObjectByPointer

4.3 Buffer Overflow

The parameter of interest for this type of vulnerability in memcpy is the destination buffer size. If an attacker can control the size of the destination buffer, it may cause a buffer overflow, resulting in a denial of service and elevation of privilege. Listing 4.5

doi:10.6342/NTU202302404

demonstrates a vulnerability identified by IOCTLance where an attacker can manipulate the size parameter in memcpy.

```
VOID Vuln_BufferOverflow(PVOID inbuf)
{
    if (!inbuf) return;
    memcpy(dest, src, *(SIZE_T *)(inbuf));
}
```

Listing 4.5: Buffer Overflow

4.4 Null Pointer Dereference

IOCTLance carefully analyzes all memory read and write operations to monitor specific buffers that do not undergo null validation. We focus on two types of buffers, namely tainted buffers and allocated memory.

4.4.1 Tainted Buffer

During symbolic execution, IOCTLance inspects each memory read and write operation on tainted buffers. An attacker can invoke DeviceIoControl with SystemBuffer set as NULL. If the driver fails to validate the input received from a user-mode process and accesses it, a null pointer dereference can occur. Our system verifies if the buffer accessed is controlled by the attacker and is possibly NULL to determine whether the operation is vulnerable. The example shown in Listing 4.6 is considered vulnerable by IOCTLance because the program fails to verify if the input buffer has a NULL value or not.

```
VOID Vuln_NullPointerDereference_TaintedBuffer(PVOID inbuf)
{
    *(CHAR *)inbuf = 0;
```

doi:10.6342/NTU202302404

```
}
```

Listing 4.6: Null Pointer Dereference - Tainted Buffer

4.4.2 Allocated Memory

IOCTLance examines each memory read and write operation on buffers that are allocated using kernel APIs that return NULL when there is not enough memory in the free pool to satisfy the request. If a program fails to check whether the return value of these APIs is NULL or not, an attacker can cause null pointer dereference by intentionally making the kernel API fail. Our system verifies whether the buffer being accessed is likely to be null to determine whether the operation is vulnerable. Listing 4.7 demonstrates a situation that IOCTLance considers vulnerable, as the program fails to verify the return value of MmAllocateNonCachedMemory.

```
VOID Vuln_NullPointerDereference_AllocatedMemory()
{
    memory = MmAllocateNonCachedMemory(0x1000);
    *(CHAR *)memory = 0;
}
```

Listing 4.7: Null Pointer Dereference - Allocated Memory

4.5 Read/Write Controllable Address

IOCTLance verifies each read and write operation during its search for vulnerabilities. If an attacker can control the destination address for the read or write operation, they can make the program read from or write to unintended addresses.

doi:10.6342/NTU202302404

4.5.1 Tainted Buffer

If a program reads from or writes to a tainted buffer, it means that an attacker can manipulate the address that the program accesses, which could potentially result in denial of service and elevation of privilege. Listing 4.8 presents an example that is considered vulnerable by IOCTLance because it allows an attacker to write NULL into a controllable address.

```
VOID Vuln_ReadWriteControllableAddress_TaintedBuffer(PVOID inbuf)
{
    if (!inbuf) return;
    **(char **)inbuf = 0;
}
```

Listing 4.8: Read/Write Controllable Address - Tainted Buffer

4.5.2 memcpy

When using the memcpy function, the dest and src parameters are of particular interest in terms of vulnerability. The dest parameter is a pointer to the destination array where the data will be copied. The src parameter is a pointer to the source of the data to be copied.

If either src or dest parameter is under an attacker's control, it could allow the attacker to read from or write to arbitrary memory addresses. This could result in denial of service and even elevation of privilege. Listing 4.9 exemplifies a case that IOCTLance considers vulnerable, as an attacker can manipulate both the destination and source addresses in the memcpy function.

```
VOID Vuln_ReadWriteControllableAddress_memcpy(PVOID inbuf)
{
```

doi:10.6342/NTU202302404

```
    if (!inbuf) return;
    memcpy(*(CHAR *)inbuf, *(CHAR *)(inbuf + 8), 4);
}
```

Listing 4.9: Read/Write Controllable Address - memcpy

4.6 Arbitrary Shellcode Execution

If an attacker controls the address that a program uses to call a function, they can execute arbitrary code and potentially elevate their privileges and achieve arbitrary kernel execution. Listing 4.10 depicts an example of the vulnerability identified by IOCTLance, where the program calls a function whose address can be manipulated by an attacker, allowing for arbitrary shellcode execution.

```
VOID Vuln_ArbitraryShellcodeExecution(PVOID inbuf)
{
    if (!inbuf) return;
    ((VOID (*)())inbuf)();
}
```

Listing 4.10: Arbitrary Shellcode Execution

4.7 Arbitrary Wrmsr

The wrmsr [24] function is a kernel function in Windows that enables writing data to a Model-Specific Register (MSR) located in the CPU. MSRs control hardware features and performance data. This function is commonly used in device drivers for hardware interaction and system behavior control.

By invoking the SYSCALL function, the OS system-call handler at privilege level

doi:10.6342/NTU202302404

0 is activated. RIP is loaded from the IA32_LSTAR MSR, while the address of the instruction following SYSCALL is saved into RCX. RFLAGS is saved into R11 and then masked using the IA32_FMASK MSR.

The _xeroxz/msrexec [12] open-source project created by IDontCode exploits the wrmsr function to execute arbitrary kernel code. The wrmsr function takes operands Address and Value, where Value is written into Address. If an attacker can control both operands, it could lead to arbitrary kernel execution. Listing 4.11 demonstrates an instance that is considered vulnerable by IOCTLance, as an attacker can manipulate the Address and Value operands in the opcode wrmsr.

```
VOID Vuln_ArbitraryWrmsr(PVOID inbuf)
{
    if (!inbuf) return;
    __writemsr(*(int *)inbuf, *(unsigned __int64 *)(inbuf + 8));
}
```

Listing 4.11: Arbitrary Wrmsr

4.8 Arbitrary Out

The Windows kernel utilizes the out [22] opcode to transfer data from the CPU to an I/O device, often by writing to an I/O port. This instruction requires two operands: the first operand, Port, specifies the I/O port number, and the second operand, Value, is the data to be written.

When developing kernel drivers, the out opcode may be utilized to transfer data to hardware devices such as sound cards or network adapters or to communicate with I/O controllers.

doi:10.6342/NTU202302404

However, if an attacker can manipulate the Port and Value operands, they can set Port to 0xcf9 and Value to 6 to shut down the system, or set Port to 0xcf9 and Value to 0xe to restart the system, potentially causing a denial of service. Listing 4.12 depicts an example that is considered vulnerable by IOCTLance, as an attacker can manipulate the Port and Value operands in the opcode out.

```
VOID Vuln_ArbitraryOut(PVOID inbuf)
{
    if (!inbuf) return;
    __outbyte(*(USHORT *)inbuf, *(UCHAR *)(inbuf + 8));
}
```

Listing 4.12: Arbitrary Out

4.9 Dangerous File Operation

IOCTLance examines the parameters passed to kernel APIs such as ZwDeleteFile [38], ZwOpenFile [63], ZwCreateFile [61], IoCreateFile [46], IoCreateFileEx [47], and IoCreateFileSpecifyDeviceObjectHint [48], which use ObjectAttributes [59] to specify the file to operate on. ObjectName in ObjectAttributes holds the file name, while Attributes holds the bitmask of flags that determine the object's attributes.

When an attacker can control ObjectName and Attributes is not OBJ_FORCE_ACCESS_CHECK, calling these kernel APIs may allow performing operations on arbitrary files and lead to unauthorized access control. ZwDeleteFile is a kernel API used to delete the file specified by ObjectName in ObjectAttributes. The other APIs generate a file handle with the specified name in ObjectName in ObjectAttributes. Listing 4.13 demonstrates an example considered vulnerable by IOCTLance due to the ability of an attacker to manipulate the ObjectName parameter in ObjectAttributes, while Attributes in ObjectAttributes does not

doi:10.6342/NTU202302404

have the OBJ_FORCE_ACCESS_CHECK flag.

```
VOID Vuln_DangerousFileOperation(PVOID inbuf)
{
    if (!inbuf) return;
    OBJECT_ATTRIBUTES ObjectAttributes;
    InitializeObjectAttributes(
        &ObjectAttributes,
        (PUNICODE_STRING)inbuf,
        OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE,
        NULL,
        NULL
    );
    ZwDeleteFile(&ObjectAttributes);
}
```

Listing 4.13: Dangerous File Operation

doi:10.6342/NTU202302404

doi:10.6342/NTU202302404

Chapter 5 Implementation

Our system, IOCTLance, is built on top of the POPKORN framework and leverages the angr [68] symbolic execution engine. We have customized and enhanced the engine to make it more effective in detecting vulnerabilities specifically in WDM drivers.

5.1 Hook Opcodes

IOCTLance employs objdump [16] to identify opcode addresses in the driver file. This is done by creating a subprocess to generate disassembly and then capturing the standard output. We subsequently compare each line of output with the opcode names that we are targeting, to locate their respective addresses.

5.1.1 wrmsr

The Windows kernel function wrmsr is used to write data to a Model-Specific Register (MSR) on the CPU and is classified as a vulnerability type in IOCTLance. The function accepts two arguments, Address and Value, which are specified by registers ecx and edx:eax, respectively. When both arguments can be controlled by an attacker through the input buffer, arbitrary kernel execution can be achieved.

doi:10.6342/NTU202302404

To detect the controllability of wrmsr's operands, IOCTLance hooks the function and evaluates whether both registers are tainted by the input buffer during execution. If Address can be set to 0xC0000082, which contains the address of the x64 system call handler and it may result in arbitrary kernel execution, the function is considered vulnerable and inserted into the report.

```
void __writemsr(
    unsigned long Register,
    unsigned __int64 Value
);
```

5.1.2 out

The opcode ”out” copies a value to the I/O port and has two operands, Port and Value, which are specified by registers edx and eax, respectively. If an attacker can control both operands, it may cause a denial of service resulting in a BSOD.

To detect the controllability of its operands, IOCTLance hooks ”out”. If both operands Port and Value are tainted by the input buffer that is controlled by an attacker during the instruction execution, and if Port can be 0xcf9 and Value can be 0xe, it will be marked as vulnerable and added to the report.

```
void __outbyte(
    unsigned short Port,
    unsigned char Data
);
```

doi:10.6342/NTU202302404

5.1.3 rep

The rep instruction is used to repeat a series of instructions, such as the rep movsb opcode which copies values from register rsi to rdi for the length specified by ecx. However, when ecx is a symbolic variable or too long, the operation can be time-consuming. Therefore, IOCTLance hooks instructions with rep, such as rep movsw, rep movsd, rep stosb, rep stosw, rep stosd, etc., and implements them in our system.

All these instructions use register ecx to specify the length of the operation. However, if ecx is symbolic, determining the length can be complex and time-consuming. To address this issue, IOCTLance evaluates a minimum value for the length inside the hook function. If the length is zero, it is set to one to avoid missing read/write controllable addresses that could result in false negatives. Conversely, if the length is too large, causing cumbersome memory operations, the length is set to a maximum value of 0x1000.

5.1.4 indirect jump

An indirect jump is a type of computer programming instruction that determines the destination of a jump at runtime, rather than directly specifying it. The instruction contains a reference to a memory address or register that holds the address of the target instruction.

During program execution, the indirect jump reads the memory address or registers specified in the instruction and jumps to the instruction located at that address. This allows for more flexibility and dynamic behavior in the program's logic, as the destination of the jump can be determined based on the program's state or input.

Indirect jumps are commonly used in programming languages like C and C++ to

doi:10.6342/NTU202302404

implement function pointers and virtual method tables. In assembly language, they can be achieved using instructions like jmp and call with a memory or register operand containing the target address.

IOCTLance hooks every indirect jump and evaluates the target addresses. If there are multiple evaluations of the target addresses, it copies the current state, adds a constraint to set the target address to one of the evaluations, and appends this state to the simulation manager [2].

5.2 Hook Functions

By default, angr's summary of kernel APIs returns a symbolic variable, and some of the API parameters passed by reference may not be properly assigned. IOCTLance checks for tainted parameters in kernel APIs to identify target vulnerabilities. Although memset and memcpy are built-in functions, they are not compiled as imported functions but rather have complicated implementations that significantly impact performance during symbolic execution. To resolve these issues, IOCTLance hooks these functions and implements them internally.

5.2.1 memset and memcpy

The functions memset and memcpy are commonly used to perform memory operations on a buffer, with the former copying memory from a source address to a destination address and the latter setting memory to a specified value. However, these functions are often compiled as functions that are not in the imported function table, which can complicate their implementation during symbolic execution.

doi:10.6342/NTU202302404

To address these issues, IOCTLance hooks both functions and replaces memset with the one implemented by angr. When the symbolic execution engine encounters the hook function for memcpy, we check whether the parameters dest and src are tainted or not. This indicates whether an attacker can control the destination or source address, which could result in a read/write controllable address vulnerability.

To avoid false positives, we also check whether the addresses are restricted by ProbeForRead and ProbeForWrite. If dest is restricted by ProbeForWrite and src is restricted by ProbeForRead, then the vulnerability is not considered vulnerable. Another vulnerability that is checked for is buffer overflow, which can occur if the size parameter of memcpy is controlled by an attacker.

In order to optimize performance, we have established a threshold of 0x1000 for the size parameter when it is either too large or symbolic. However, it is important to note that if the size is evaluated as 0, a false negative may occur due to the mem_read or mem_write breakpoint not being triggered. This could result in a failure to detect null pointer dereference vulnerabilities. To address this, we set the size to 1 in these cases and check whether the state is satisfiable [5]. Overall, IOCTLance leverages memcpy and memset hooks to enhance performance during symbolic execution and identify vulnerabilities.

5.2.2 Imported Kernel APIs

Windows kernel APIs are essential components of the operating system that are imported from a built-in library. However, angr does not have a built-in mechanism to handle these APIs, and instead, it returns a symbolic variable, which may result in lost output that should have been set as parameters from the APIs. This can impact the program's behav-

doi:10.6342/NTU202302404

ior, especially if the output is necessary for executing certain code, leading to decreased code coverage during simulation.

Furthermore, there are many vulnerabilities in the code that can be exploited by attackers due to a lack of parameter validation in the APIs. To identify such vulnerabilities, IOCTLance hooks APIs that are commonly targeted by attackers or are prone to common kernel exploits. In the detoured function, the system checks whether the code is vulnerable by evaluating the parameters of the API.

5.2.2.1 Restricted Address

Several kernel APIs in the Windows kernel, such as MmIsAddressValid, ProbeForRead, and ProbeForWrite, are used to validate addresses. MmIsAddressValid checks whether a given virtual address will cause a page fault during a read or write operation. ProbeForRead checks whether a user-mode buffer is located in the correct portion of the address space and is properly aligned, while ProbeForWrite checks whether a user-mode buffer is located in the user-mode portion of the address space, is writable, and is properly aligned.

To avoid generating false positives, IOCTLance keeps a record of the addresses that have been validated by these kernel APIs. This helps prevent issues like null pointer dereference or read/write controllable address.

5.2.2.2 Unicode String

The structure UNICODE_STRING is utilized to define Unicode strings. By registering a type for UNICODE_STRING, IOCTLance is able to conveniently access the

doi:10.6342/NTU202302404

member within the structure.

The kernel API RtlInitUnicodeString is used to initialize a counted Unicode string, with DestinationString and SourceString as its parameters. The former is the buffer for the initialized counted Unicode string, while the latter is a pointer to a null-terminated Unicode string. By implementing the API, if SourceString is tainted, IOCTLance saves the DestinationString to the tainted Unicode strings list.

The kernel API RtlCopyUnicodeString is used to copy a source string to a destination string, with both parameters being pointers to Unicode strings. IOCTLance implements the API and if SourceString is tainted or in the tainted Unicode strings list, the DestinationString is saved to the tainted Unicode strings list as well.

5.2.2.3 ObjectAttributes

The OBJECT_ATTRIBUTES is a data structure in the Windows kernel that specifies the attributes of a kernel object, including its name, security descriptor, and other control attributes during object creation or manipulation.

To access the members inside ObjectAttributes, IOCTLance registers a type for OBJECT_ATTRIBUTES. We specifically resolve ObjectName and Attributes members and check if ObjectName is tainted and the value of Attributes. To avoid false positives, we add a constraint that Attributes should not be 1024, which indicates that it is not necessarily OBJ_FORCE_ACCESS_CHECK. We then check if the state is satisfiable.

```
typedef struct _OBJECT_ATTRIBUTES {
  ULONG           Length;
  HANDLE          RootDirectory;
  PUNICODE_STRING ObjectName;
  ULONG           Attributes;
```

doi:10.6342/NTU202302404

```
    PVOID               SecurityDescriptor;
    PVOID               SecurityQualityOfService;
} OBJECT_ATTRIBUTES;
```

5.2.2.4 Map Physical Memory

Several kernel APIs are available in the Windows kernel to map physical memory, including MmMapIoSpace, MmMapIoSpaceEx, and ZwMapViewOfSection.

When it comes to MmMapIoSpace and MmMapIoSpaceEx, their parameters PhysicalAddress and NumberOfBytes are checked by IOCTLance. If either of these parameters is tainted, it can be used by attackers to map an arbitrary physical memory address, leading to insertion into the report.

```
PVOID MmMapIoSpace(
    PHYSICAL_ADDRESS     PhysicalAddress,
    SIZE_T               NumberOfBytes,
    MEMORY_CACHING_TYPE  CacheType
);
```

For ZwMapViewOfSection, if SectionHandle is either \Device\PhysicalMemory or is tainted, and BaseAddress or CommitSize and ViewSize are also tainted, it can be used by attackers to map arbitrary physical memory addresses. To determine whether SectionHandle is tainted or \Device\PhysicalMemory, IOCTLance hooks ZwOpenSection and creates a symbolic variable to store the information. Then, we can identify the parameter SectionHandle in ZwMapViewOfSection.

```
NTSYSAPI NTSTATUS ZwMapViewOfSection(
    HANDLE          SectionHandle,
    HANDLE          ProcessHandle,
    PVOID           *BaseAddress,
    ULONG_PTR       ZeroBits,
```

doi:10.6342/NTU202302404

```
    SIZE_T            CommitSize,
    PLARGE_INTEGER    SectionOffset,
    PSIZE_T           ViewSize,
    SECTION_INHERIT   InheritDisposition,
    ULONG             AllocationType,
    ULONG             Win32Protect
);
```

5.2.2.5 Allocated Memory

Several kernel APIs are available in Windows to allocate kernel memory, such as ExAllocatePool [42], ExAllocatePool2 [43], ExAllocatePool3 [27], MmAllocateNonCachedMemory [56], ExAllocatePoolWithTag [44], and MmAllocateContiguousMemorySpecifyCache [55]. IOCTLance hooks these APIs and returns a symbolic variable with a recognizable name. This name is then used to validate whether the allocated memory is likely to be NULL, by setting mem_read and mem_write breakpoints. If it is likely to be NULL, this indicates that the program doesn't check the value returned by the kernel API, which could lead to null pointer dereference vulnerabilities.

5.2.2.6 Process Operation

IOCTLance monitors the ZwOpenProcess and ObOpenObjectByPointer kernel APIs to ensure that attackers cannot control the accessibility to the process and the process ID.

The CLIENT_ID structure is used in ZwOpenProcess, which takes ObjectAttributes and ClientId as parameters. IOCTLance registers a type for CLIENT_ID, which allows access to the member inside the structure. If either the ClientId or UniqueProcess is tainted, and the resolved ObjectAttributes attribute is not OBJ_FORCE_ACCESS_CHECK, it is

doi:10.6342/NTU202302404

considered vulnerable.

```
NTSYSAPI NTSTATUS ZwOpenProcess(
    PHANDLE             ProcessHandle,
    ACCESS_MASK         DesiredAccess,
    POBJECT_ATTRIBUTES  ObjectAttributes,
    PCLIENT_ID          ClientId
);
```

ObOpenObjectByPointer is another kernel API that returns a process handle. IOCTLance hooks this function against process operation, which takes a parameter Object and outputs a process handle. To identify if Object is an EPROCESS object and is tainted, IOCTLance also hooks PsLookupProcessByProcessId. This routine accepts a process ID and returns a referenced pointer to the EPROCESS structure. If the ProcessId is tainted, it is put into the tainted list. If Object passed to ObOpenObjectByPointer is in the tainted list, it is considered vulnerable.

```
NTSTATUS ObOpenObjectByPointer(
    PVOID            Object,
    ULONG            HandleAttributes,
    PACCESS_STATE    PassedAccessState,
    ACCESS_MASK      DesiredAccess,
    POBJECT_TYPE     ObjectType,
    KPROCESSOR_MODE  AccessMode,
    PHANDLE          Handle
);
```

5.2.2.7 File Operation

There are several kernel APIs used for file operations in Windows, such as ZwDeleteFile, ZwOpenFile, ZwCreateFile, IoCreateFile, IoCreateFileEx, and IoCreateFileSpecifyDeviceObjectHint. One of the parameters they all take is ObjectAttributes, which is of

doi:10.6342/NTU202302404

interest to us.

We can resolve ObjectName and Attributes parameters in ObjectAttributes. If ObjectName in ObjectAttributes is tainted and the value of Attributes is not definitively OBJ_FORCE_ACCESS_CHECK, then an attacker can perform file operations on any file without verifying the required privilege. In such a scenario, it will be considered vulnerable.

5.2.2.8 Other

To avoid listing all the hooked Windows kernel APIs, here's a summary of some important ones that IOCTLance uses.

The IoStartPacket [21] function is hooked to create a call state and insert it into the simulation manager, based on the current state and calling DriverStartIo defined in DriverEntry.

IoCreateDevice is hooked to initialize the DeviceObject as a symbolic variable and create DeviceExtension with the size specified by the parameter DeviceExtensionSize.

RtlGetVersion [37] is hooked to pass conditions of the version set by some WDM drivers. IOCTLance registers a type for RTL_OSVERSIONINFOW [33] and RTL_OSVERSIONINFOEXW [60], and symbolizes the dwMajorVersion, dwMinorVersion, and dwBuildNumber in lpVersionInformation, to overcome version validation from the program.

```
typedef struct _OSVERSIONINFOW {
    ULONG dwOSVersionInfoSize;
    ULONG dwMajorVersion;
    ULONG dwMinorVersion;
    ULONG dwBuildNumber;
    ULONG dwPlatformId;
```

doi:10.6342/NTU202302404

```
    WCHAR szCSDVersion[128];
} OSVERSIONINFOW, *POSVERSIONINFOW, *LPOSVERSIONINFOW,
    RTL_OSVERSIONINFOW, *PRTL_OSVERSIONINFOW;
```

Some programs use PsGetVersion [36] to check the version and execute corresponding codes. IOCTLance symbolizes the parameters MajorVersion, MinorVersion, BuildNumber, and CSDVersion to pass the conditions set by WDM drivers.

```
BOOLEAN PsGetVersion(
    PULONG          MajorVersion,
    PULONG          MinorVersion,
    PULONG          BuildNumber,
    PUNICODE_STRING CSDVersion
);
```

MmGetSystemRoutineAddress [30] and FltGetRoutineAddress [19] are hooked to return a function pointer of a SimProcedure that returns 0 and avoid running into an unconcretized address. By resolving the API name from the parameter, IOCTLance can hook the target API with a specified SimProcedure to produce more accurate results.

5.3 Set Breakpoints

IOCTLance utilizes APIs to set breakpoints on specific operations for taint analysis and to handle situations that affect symbolic execution. IOCTLance sets breakpoints on mem_read and mem_write to trace tainted buffers and identify vulnerability types. Additionally, the breakpoint for the call instruction is implemented to handle situations where the target function address is either a symbolic variable or a tainted buffer that is controlled by an attacker, allowing for the detection of arbitrary shellcode execution.

doi:10.6342/NTU202302404

5.3.1 mem_read and mem_write

Taint analysis is crucial in identifying vulnerabilities in our system. In addition to symbolizing the buffer of interest, we also examine the memories of concretized addresses to identify tainted buffers. IOCTLance inspects memory read and write to determine if the target address is tainted and not yet concretized. If so, we concretize the target address and symbolize the memory it points to. We use this approach to detect target values written into a structure and find the IOCTL handler. We inspect various target addresses, including SystemBuffer, Type3InputBuffer, UserBuffer, and memory allocated by different functions.

To identify null pointer dereference vulnerabilities, we impose a constraint on the allocated memory to check if the target address can be NULL. If the state is satisfiable, it indicates that the program does not validate the success of kernel APIs, potentially allowing an attacker to cause null pointer dereference.

In the case of SystemBuffer, Type3InputBuffer, and UserBuffer, they are input buffers that can be manipulated by attackers. At the initialization stage, IOCTLance generates symbolic variables for these input buffers. During breakpoints, we set constraints to verify that SystemBuffer is NULL, InputBufferLength is 0, and OutputBufferLength is 0. If it is possible for these to be NULL and not restricted by MmIsAddressValid, this implies that the program does not validate whether SystemBuffer is NULL, which can lead to null pointer dereference. Because Type3InputBuffer or UserBuffer does not copy the contents to other memory locations like SystemBuffer does, if the program fails to validate these buffers, attackers may be able to manipulate read/write addresses. We examine the vulnerability by setting Type3InputBuffer or UserBuffer to any non-null value, and if the state

doi:10.6342/NTU202302404

is satisfiable and the target address is not limited by ProbeForRead or ProbeForWrite, this implies that the vulnerability exists, and the attacker can manipulate the address to read or write.

In addition to SystemBuffer, Type3InputBuffer, and UserBuffer, IOCTLance also focuses on the pointers contained within these buffers. We use concrete addresses to represent these buffers and store our own symbolic variables at these addresses. Because we can identify these buffers, we validate them when performing read/write operations by adding a constraint to set them to any non-null value. If the state is satisfiable and the target address is not restricted by ProbeForRead or ProbeForWrite, it indicates that an attacker can manipulate the address to perform read/write operations, indicating the presence of a vulnerability.

5.3.2 Call

IOCTLance places breakpoints on all function calls to handle situations where the address of the called function is symbolized. This occurs when a program calls a function pointer that has not been successfully assigned, such as when the function pointer is assigned in other IoControlCodes but used in others. This creates an error state because angr is unable to determine the target address to call, resulting in decreased code coverage. Therefore, in the call breakpoint, if the target function address is symbolic, we manually set the instruction pointer to a SimProcedure that executes nothing.

Furthermore, we verify whether the target address is tainted to confirm if there is an arbitrary shellcode execution vulnerability. If the target function address is either SystemBuffer, Type3InputBuffer, or UserBuffer, IOCTLance includes the vulnerability in the

doi:10.6342/NTU202302404

report.

5.4 Use Techniques

ExplorationTechnique [3] is a feature provided by angr that allows users to customize the behavior during symbolic execution. In order to prevent recursion, IOCTLance checks the call stack of each state to ensure that a function address does not appear more than once. If a recursive call is detected, the state is terminated. In addition, WDM drivers often have multiple IoControlCodes that perform different tasks. To prevent the analysis of a particularly complex IoControlCode from affecting the overall analysis, IOCTLance sets a timeout for each individual IoControlCode. The starting time of each IoControlCode is recorded when it is evaluated in each state. If the time taken to analyze an IoControlCode exceeds its timeout, all states associated with that IoControlCode are terminated.

IOCTLance provides various options for controlling the symbolic execution of drivers using angr. One such option is Timeout [4], which enables the user to set a maximum time limit for the analysis to finish. If the analysis exceeds this limit, it is terminated. Another option offered by angr is LengthLimiter, which allows the user to restrict the length of instructions that a state can execute. This helps to minimize the number of states and prevent state explosion. In addition, IOCTLance utilizes Angr's LoopSeer and LocalLoopSeer techniques to constrain the number of times a state can loop. This can also decrease the number of states and mitigate state explosion.

doi:10.6342/NTU202302404

5.5 Initialize Structures

During the vulnerability hunting process, IOCTLance employs taint analysis. To begin with, we construct the kernel structures ourselves and generate symbolic variables for the specific members that our system targets. IOCTLance employs two phases for conducting symbolic execution, namely, finding the IOCTL handler and hunting vulnerabilities.

During the first phase - find IOCTL handler, IOCTLance sets addresses and stores the symbolic variables we created into those addresses to represent DriverObject and RegistryPath, respectively. This phase starts from DriverEntry, which takes the parameters DriverObject and RegistryPath.

The second phase - hunt vulnerabilities, begins from the IOCTL handler, which takes parameters DeviceObject and IRP. To properly execute this phase, DeviceObject needs to be built in IoCreateDevice and initialized in DriverEntry when finding the IOCTL handler. However, if our system fails to find the entire initialization of kernel structures within the timeout, we will proceed to hunt vulnerabilities from a blank state and create a symbolic variable for DeviceObject. In IRP, several members may be under the control of an attacker. Hence, IOCTLance sets MajorFunction to 14, which represents DEVICE_IO_CONTROL, and sets RequestorMode to 1, which represents user mode. We also create symbolic variables for SystemBuffer, InputBufferLength, OutputBufferLength, Type3InputBuffer, UserBuffer, and IoControlCode.

Programs often use global variables in the data section to determine whether to execute a statement. By default, angr initializes the data section with NULL, which can

doi:10.6342/NTU202302404

decrease code coverage during symbolic execution. For instance, if a global variable is set in an IoControlCode, and another reads the global variable, some code may not be simulated, leading to false positives. To address this issue, we have the option to symbolize the data section with a specified length. The decision of how long to symbolize is a tradeoff between solving false negatives and generating new false positives. We will discuss the impact of symbolizing the data section in the Evaluation section.

doi:10.6342/NTU202302404

doi:10.6342/NTU202302404

Chapter 6 Evaluation

In this chapter, we assess the performance of IOCTLance and compare it to POPKORN using both known and unknown drivers. Our experiments were conducted on a four-core Intel i7-3770K 3.40GHz CPU and 32GB of RAM. The evaluation process was divided into two phases: finding the IOCTL handler and hunting vulnerabilities.

We obtained 185 vulnerable WDM drivers from namazso/physmem_drivers [65] and CaledoniaProject/drivers-binaries [6] for the known driver dataset. Through manual reverse-engineering, we deduplicated the set to 104 unique drivers, with the largest file being 1.3 MB and the smallest being 10.1 KB. On average, the drivers in the known dataset had a size of 62.4 KB.

For the unknown driver dataset, we manually downloaded and installed various software and collected 1959 drivers. We first removed non-WDM drivers and duplicates, resulting in 342 unique drivers. Next, we removed drivers that were solely used to export functions for other drivers and those without an IOCTL handler, leaving us with 318 unique drivers. The largest file in this dataset was 71.1 MB, while the smallest was 7.2 KB, with an average size of 630.0 KB.

doi:10.6342/NTU202302404

6.1 Known Drivers

To evaluate IOCTLance, we conducted manual reverse-engineering on each vulnerable driver and used the results as a benchmark for comparison. We set a total timeout of 30 minutes for both phases and 40 seconds for each IoControlCode, and were able to successfully analyze 103 known drivers. Of these, IOCTLance was able to identify the IOCTL handler within 30 seconds in the first phase for 100 known drivers. The second phase, which involved hunting for vulnerabilities, was completed in 20 minutes for 98 known drivers. Table 6.1 compares the number of vulnerabilities in the known WDM drivers found by IOCTLance with the ground truth. v1, v2, v3, v4, v5, v6, v7, v8, and v9 represent ”map physical memory”, ”controllable process handle”, ”read/write controllable address”, ”buffer overflow”, ”null pointer dereference”, ”arbitrary shellcode execution”, ”arbitrary wrmsr”, ”arbitrary out”, and ”dangerous file operation” respectively.

The number of vulnerabilities found in the experiment is lower than the ground truth due to the following reasons. Despite the occurrence of some false negatives, we were able to mitigate the issues by customizing options for IOCTLance.

1. Timeout. To locate the IOCTL handler, a total timeout of 30 minutes was set, and IOCTLance was able to find IOCTL handlers in 103 out of 104 known drivers. However, in three drivers, the IOCTL handler was located within the timeout period, but the initialization step from DriverEntry was not completed, leading to the hunting of vulnerabilities with a blank state. For the last driver, the issue was caused by long-running concrete loops, which we solved by manually reverse-engineering and finding the address of the IOCTL handler ourselves. If an analysis lacks the necessary context to hunt vulnerabilities, it may cause false negatives. But in our experiment, the lack of context did not affect the number

doi:10.6342/NTU202302404

of vulnerabilities much.

To hunt vulnerabilities, a total timeout of 30 minutes and an IoControlCode timeout of 40 seconds were set. This means that in the worst case, only 45 IoControlCodes can be analyzed by IOCTLance. Some drivers, such as rtkiow10x64.sys with 192 IoControlCodes, have many IoControlCodes, and in our experiment, six out of 104 known drivers required a longer total timeout due to the large number of IoControlCodes. Additionally, 40 seconds for each IoControlCode may not be enough, as some drivers implement several features within an IoControlCode, leading to false negatives when IOCTLance reaches the timeout for IoControlCodes.

2. Data Section. Global variables are used as conditions to decide whether to walk into logic in some vulnerabilities. Although global variables belong to the data section, they are not symbolized by default. Therefore, if global variables are not correctly initialized, false negatives may occur. IOCTLance provides an option for users to symbolize an editable length of the data section to address this issue. In our experiment, one driver required symbolizing 0x3000 bytes in the first phase, and three out of 104 known drivers required symbolizing 0x1000 bytes of the data section to analyze properly in the second phase. By symbolizing the customized length of the data section, we were able to successfully find the correct number of vulnerabilities.

Table 6.1: Compare ground truth with experiment by the number of the vulnerability.

Result	v1	v2	v3	v4	v5	v6	v7	v8	v9
Ground Truth	221	12	104	65	818	8	49	179	6
Experiment	217	12	103	65	813	8	49	172	4

doi:10.6342/NTU202302404

6.2 Unknown Drivers

In this section, we present an evaluation of IOCTLance using a set of 318 unknown drivers. To analyze these drivers, we set a total timeout of 30 minutes for both phases, with each IoControlCode given 40 seconds to execute. During the first phase, the IOCTL handler of 220 unidentified drivers can be detected within 30 seconds, and 259 unknown drivers can be detected before the timeout. The remaining drivers are manually tuned by setting the length limit, the bound limit, and the length of the data section to symbolize. In the second phase, which aims to identify vulnerabilities, 282 unknown drivers can be analyzed within 20 minutes and 287 can be examined before the timeout. For the remaining drivers, the timeout of the second phase is extended to ensure their complete analysis.

We consider a vulnerability to be unique if it differs in either the IoControlCode or the vulnerability type. Using this criterion, we discovered 117 previously unknown vulnerabilities in 26 unique drivers, resulting in the identification of 41 CVEs, including 5 CVEs from Advanced Micro Devices and 2 CVEs from Microsoft. Some of these vulnerabilities are currently being addressed. The number of real-world vulnerabilities found by IOCTLance in unknown WDM drivers is presented in Table 6.2, while the vulnerabilities that have been assigned CVE numbers at the time of writing this paper are listed in Table A.1. v1, v2, v3, v4, v5, v6, v7, v8, and v9 represent ”map physical memory”, ” controllable process handle”, ”read/write controllable address”, ”buffer overflow”, ”null pointer dereference”, ”arbitrary shellcode execution”, ”arbitrary wrmsr”, ”arbitrary out”, and ”dangerous file operation” respectively.

doi:10.6342/NTU202302404

Table 6.2: Vulnerabilities hunted by IOCTLance.

v1	v2	v3	v4	v5	v6	v7	v8	v9
19	1	18	5	55	1	2	12	4

6.3 Comparison with POPKORN

This section provides a comparison between IOCTLance and POPKORN, using the ucsb-seclab/popkorn-artifact [17] open-source project. POPKORN is a lightweight tool that utilizes symbolic execution and taint analysis to identify vulnerabilities related to MmMapIoSpace, ZwMapViewOfSeciton, and ZwOpenProcess. It prevents state explosion by monitoring the number of states and terminating symbolic execution once it exceeds a specific threshold. However, POPKORN lacks optimization, leading to poor performance and numerous false negatives. On the other hand, IOCTLance surpasses POPKORN in detecting various vulnerability types. It also enhances performance by hooking functions and opcodes to bypass certain checks in the target programs.

When analyzing parameters in the target function, POPKORN may generate false positives. Listing 6.1 shows an example. if a parameter is tainted, but our input buffer only serves as an index of a global array, POPKORN can mistake it for being controllable by an attacker, and thus mark it as vulnerable. To illustrate, suppose the parameter PhysicalAddress is tainted by the input buffer controlled by an attacker, but it can only affect the index of a global variable. In this scenario, POPKORN would misidentify it as tainted and deem it vulnerable. However, IOCTLance avoids this problem by scrutinizing the combination of a tainted symbolic variable, thus preventing false positives.

```
void FalsePositive_MmMapIoSpace(PVOID inbuf)
{
    PHYSICAL_ADDRESS global_array[4] = {global_pa1, global_pa2,
```

doi:10.6342/NTU202302404

```
        global_pa3 , global_pa4};
    if (inbuf && *(int *)inbuf < 4 && *(int *)inbuf >= 0)
    {
        // false positive: map physical memory
        MmMapIoSpace(global_array[*(int *)inbuf], 4, MmNonCached);
    }
}
```

Listing 6.1: False Positive For POPKORN: Tainted Buffer

Upon examining the source code of POPKORN, we discovered that it only examines the state of the target function via a single path. Consequently, it may fail to detect vulnerabilities in other IoControlCodes that use the same target function. On the other hand, IOCTLance intercepts the target functions and comprehensively analyzes each IoControlCode from the IOCTL handler. This approach ensures that no vulnerabilities are overlooked. In our dataset of 104 known drivers, we confirmed that every vulnerability detected by POPKORN was also detected by IOCTLance.

Table 6.3 shows the number of IOCTL handlers in the known drivers and the average time taken by IOCTLance and POPKORN. We conducted a comparison of the time taken by IOCTLance and POPKORN to find the IOCTL handler. It was observed that POPKORN was only able to find the handler within the 30-minute timeout limit. In contrast, IOCTLance demonstrated significantly better performance in this aspect, indicating that we have effectively improved the efficiency of the first phase.

Table 6.3: Performance for IOCTLance and POPKORN.

Tool	IOCTL Handler Found	Average Time (s)
POPKORN	95	82
IOCTLance	103	3

doi:10.6342/NTU202302404

6.4 False Positive

Even though we have implemented checks on MmIsAddressValid, ProbeForRead, and ProbeForWrite to reduce false positives, there are other kernel APIs that can verify whether the parameters are NULL or not. Listing 6.2 shows an example. When the program invokes RtlInitUnicodeString with the SourceString as the input buffer, if DestinationString is not NULL, the program writes into the address pointed to by the input buffer. This could lead to a null pointer dereference vulnerability being reported, even though it should only occur if DestinationString is not NULL. However, since SystemBuffer is not NULL when DestinationString is not NULL, this condition should also be checked.

```
void FalsePositive_NullPointerDereference(PVOID inbuf)
{
    DestinationString = NULL;
    RtlInitUnicodeString(&DestinationString, inbuf);
    if (DestinationString)
    {
        // false positive: null pointer dereference
        *inbuf = 1;
    }
}
```

Listing 6.2: False Positive: Kernel APIs

Certain programs use the try-except mechanism to handle exceptions. This technique can prevent some vulnerabilities from being triggered under the circumstances. For instance, when dealing with null pointer dereference, we expect the system to crash when the program accesses a null pointer. However, if the exception is caught by an exception handler, the program may return from the function without crashing. Since IOCTLance

doi:10.6342/NTU202302404

does not support detecting this kind of mechanism, we have to manually validate the vulnerabilities found by IOCTLance.

6.5 False Negative

Not symbolizing the data section may result in false negatives. For instance, a global variable is initialized in one IoControlCode and checked in another to determine whether to perform a task. Even when symbolizing the data section, we must allocate sufficient length. Balancing false positives, false negatives, time, and memory is crucial.

Discontinuing the state of recursion may lead to some false negatives. However, we haven't found any actual scenarios where vulnerabilities were missed due to this.

While IOCTLance has significantly improved symbolic execution for WDM drivers, it can still be challenging for them to navigate the target address of complex programs, leading to path explosion. This is a severe impediment to symbolic execution.

doi:10.6342/NTU202302404

Chapter 7 Discussion

In this chapter, we explore some specific cases encountered during our analysis and demonstrate the various options available to customize the analysis, and offer guidance on selecting appropriate options for optimal results. The evaluation results are presented, providing further insight into IOCTLance's operation. Additionally, this chapter highlights certain limitations that we have identified.

7.1 Customization

IOCTLance offers users several options to customize their analysis based on specific cases. The following section outlines how to adjust these features based on our experience.

1. Length Limit: By default, IOCTLance does not set a length limit since it depends on the complexity and size of the target driver. Although there is no definite correlation between the driver size and length limit, larger drivers usually require a larger limit. In general, a larger length limit can be set for bigger drivers, but a precise limit can be obtained by reverse-engineering the target driver to analyze the length of the DriverEntry and IOCTL handler.

2. Loop Bound: By default, IOCTLance does not set a loop bound since a con-

doi:10.6342/NTU202302404

crete loop that a program must pass through can cause false negatives if not accounted for. Many WDM drivers use loops to assign function pointers to correspond to IRP operations, usually iterating 28 times. A loop bound of 30 can bypass such situations, but unpredictable situations may affect the setting. Reverse-engineering the target driver to analyze the iteration is crucial for determining a proper loop bound.

3. Total Timeout: IOCTLance sets the total timeout to 30 minutes by default, which has proven successful in detecting vulnerabilities completely in 98 out of 104 known drivers. If the time taken to hunt for vulnerabilities exceeds the total timeout, a larger value should be set. If it takes longer to find the IOCTL handler than the total timeout, two possible reasons exist. Either it needs more time to obtain the IOCTL handler, or it runs into complex codes. In the former case, a longer total timeout is needed, and in the latter, a proper length limit or loop bound should be set after manual analysis.

4. IoControlCodes Timeout: By default, IOCTLance sets the timeout to 40 seconds since the majority of vulnerabilities in our dataset can be detected with this setting. However, some drivers implement many features in an IoControlCode, leading to insufficient time to go deeper. Under such circumstances, a larger timeout for IoControlCodes is recommended. If it runs into complex codes, a proper length limit or loop bound should be set after manual analysis.

5. Recursion: By default, IOCTLance kills the recursion states since it critically affects the performance of symbolic execution. While there are no known vulnerabilities in recursion among our 104 known drivers due to IOCTLance's focus, a rare case may arise. Therefore, the option is available to users for customization.

6. Symbolize Data Section: By default, IOCTLance does not symbolize the data sec-

doi:10.6342/NTU202302404

tion. In our experiment, only 3 out of 104 known drivers required symbolizing 0x1000 to get the correct result, and only 1 out of 104 known drivers required symbolizing 0x3000. However, symbolizing the data section incurs overhead, with symbolizing 0x1000 data section taking over 27 seconds and consuming more than 96KB of memory on average to hunt for vulnerabilities in the 104 known drivers. In addition, The act of symbolizing the data section can potentially result in false positives. Listing 7.1 shows an example. If a program has a condition that restricts a global variable to an tainted buffer, IOCTLance may perceive it as a vulnerability where the address is read from or written into. For instance, if global_var is constrained to SystemBuffer and the program writes into the address pointed by global_var, IOCTLance will report it as a read/write controllable address vulnerability. However, this may actually be a function pointer stored in a global variable, where the program is comparing the input buffer with it. Users must decide how long they need to symbolize the data section while analyzing the target driver.

```
void FalsePositive_ReadWriteControllableAddress(PVOID inbuf)
{
    if (inbuf && global_var == inbuf[0])
    {
        // false positive: read/write controllable address
        *global_var = 1;
    }
}
```

Listing 7.1: False Positive: Symbolize Data Section

7.2 Bugs Not Vulnerabilities

The vulnerabilities detected by IOCTLance possible to turn out to be mere bugs under certain conditions. For instance, the null pointer dereference caused by accessing allo-

doi:10.6342/NTU202302404

cated memory without validating the return value is one such type of vulnerability that can sometimes be just a bug. If we cannot find a way to deplete the kernel memory resources to make the APIs used to allocate memory fail, it is less possible to cause a BSOD, and hence it is more unlikely to be considered a vulnerability.

Moreover, certain WDM drivers or their functionalities can only be accessed by processes with Administrator privileges or even other kernel drivers. In such cases, an attacker cannot exploit the vulnerabilities from a user-mode process with unprivileged access. Given the controversy surrounding the boundary between Administrator and kernel privileges, many vendors do not consider it a vulnerability if Administrator privilege or access from other kernel-mode drivers is required. At present, IOCTLance lacks the capability to automatically identify whether a specific privilege is required to exploit a vulnerability. As a result, we have to manually analyze the drivers that appear to be vulnerable.

7.3 Limitations

IOCTLance currently supports only x64 WDM drivers, and it is weak to analyze packed drivers. To analyze a packed driver, it has to analyze complicated code and may need to interact with OS to perform correctly. In addition, if a driver imports functions from another driver, IOCTLance currently can not simulate the imported function, and thus it causes some false negatives under the condition.

doi:10.6342/NTU202302404

Chapter 8 Conclusion

WDM drivers can pose a significant security risk to a system because of their kernel-level privileges. The proposed solution, IOCTLance, utilizes symbolic execution and taint analysis to hunt for vulnerabilities. Vulnerability types targeted by IOCTLance include " map physical memory", "controllable process handle", "buffer overflow", "null pointer dereference", "read/write controllable address", "arbitrary shellcode execution", "arbitrary wrmsr", "arbitrary out", and "dangerous file operation". By tracing the target input buffer and marking them as tainted, IOCTLance hooks kernel APIs and opcodes and sets breakpoints to detect vulnerabilities, reducing both false positives and false negatives while improving symbolic execution performance. By testing IOCTLance on 104 known vulnerable WDM drivers and 318 unknown WDM drivers, we demonstrate that IOCTLance can effectively analyze WDM drivers. Compared to POPKORN, IOCTLance has superior performance and lower rates of false positives and false negatives. The discussion delves into the customizations made to IOCTLance for analyzing WDM drivers on a case-by-case basis, as well as the controversial vulnerabilities and limitations encountered during the hunting process. As of the writing of this paper, IOCTLance has successfully uncovered 117 vulnerabilities that were previously unknown, spread across 26 drivers, resulting in the assignment of 41 CVEs.

doi:10.6342/NTU202302404

doi:10.6342/NTU202302404

References

[1] Andreas Klopsch. Remove all the callbacks –blackbyte ransomware disables edr via rtcore64.sys abuse. Technical report, Sophos News, Oct. 2022.

[2] angr documentation. Simulation managers, 2023.

[3] angr documentation. Source code for angr.exploration_techniques.tech_builder, 2023.

[4] angr documentation. Source code for angr.exploration_techniques.timeout, 2023.

[5] angr documentation. Symbolic expressions and constraint solving, 2023.

[6] CaledoniaProject. drivers-binaries, 2022.

[7] Debasish Mandal. debasishm89/iofuzz, 2014.

[8] Dmytro Oleksiuk. Cr4sh/ioctlfuzzer, 2011.

[9] eclypsium. Screwed-drivers, 2019.

[10] R. Gupta, L. Dresel, N. Spahn, G. Vigna, C. Kruegel, and T. Kim. POPKORN: Popping Windows Kernel Drivers At Scale. PhD thesis, UC Santa Barbara, USA, 2022.

doi:10.6342/NTU202302404

[11] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. PhD thesis, VU University Amsterdam, USA, 2013.

[12] IDontCode. _xeroxz/msrexec, 2021.

[13] Jan Vojtěšek. The return of candiru: Zero-days in the middle east. Technical report, Avast, July 2022.

[14] S. Y. Kim, S. Lee, I. Yun, W. Xu, B. Lee, Y. Yun, and T. Kim. CAB-FUZZ: Practical Concolic Testing Techniques for COTS Operating Systems. PhD thesis, Purdue University, USA, 2017.

[15] koutto. Ioctlbf, 2017.

[16] Linux manual page. objdump.

[17] Lukas Dresel and Rajat Gupta. ucsb-seclab/popkorn-artifact, 2022.

[18] Microsoft. Device input and output control (ioctl), 2021.

[19] Microsoft. Fltgetroutineaddress function (fltkernel.h), 2021.

[20] Microsoft. Introduction to wdm, 2021.

[21] Microsoft. Iostartpacket function (ntifs.h), 2021.

[22] Microsoft. __outbyte, 2021.

[23] Microsoft. Windows kernel-mode i/o manager, 2021.

[24] Microsoft. __writemsr, 2021.

[25] Microsoft. Writing dispatch routines, 2021.

doi:10.6342/NTU202302404

[26] Microsoft. devtest/ioattack, 2022.

[27] Microsoft. Exallocatepool3 function (wdm.h), 2022.

[28] Microsoft. Iocreatesymboliclink function (wdm.h), 2022.

[29] Microsoft. Irp structure (wdm.h), 2022.

[30] Microsoft. Mmgetsystemroutineaddress function (wdm.h), 2022.

[31] Microsoft. Mmmapiospaceex function (wdm.h), 2022.

[32] Microsoft. Obopenobjectbypointer function (ntifs.h), 2022.

[33] Microsoft. Osversioninfow structure (wdm.h), 2022.

[34] Microsoft. Probeforread function (wdm.h), 2022.

[35] Microsoft. Probeforwrite function (wdm.h), 2022.

[36] Microsoft. Psgetversion function (wdm.h), 2022.

[37] Microsoft. Rtlgetversion function (wdm.h), 2022.

[38] Microsoft. Zwdeletefile function (ntifs.h), 2022.

[39] Microsoft. Zwopenprocess function (ntddk.h), 2022.

[40] Microsoft. Driver_object structure (wdm.h), 2023.

[41] Microsoft. Driver_startio callback function (wdm.h), 2023.

[42] Microsoft. Exallocatepool function (wdm.h), 2023.

[43] Microsoft. Exallocatepool2 function (wdm.h), 2023.

[44] Microsoft. Exallocatepoolwithtag function (wdm.h), 2023.

doi:10.6342/NTU202302404

[45] Microsoft. Iocreatedevice function (wdm.h), 2023.

[46] Microsoft. Iocreatefile function (wdm.h), 2023.

[47] Microsoft. Iocreatefileex function (ntddk.h), 2023.

[48] Microsoft. Iocreatefilespecifydeviceobjecthint function (ntddk.h), 2023.

[49] Microsoft. Irp major function codes, 2023.

[50] Microsoft. Irp_mj_close, 2023.

[51] Microsoft. Irp_mj_create, 2023.

[52] Microsoft. Irp_mj_device_control, 2023.

[53] Microsoft. Irp_mj_read, 2023.

[54] Microsoft. Irp_mj_write, 2023.

[55] Microsoft. Mmallocatecontiguousmemoryspecifycache function (wdm.h), 2023.

[56] Microsoft. Mmallocatenoncachedmemory function (ntddk.h), 2023.

[57] Microsoft. Mmisaddressvalid function (ntddk.h), 2023.

[58] Microsoft. Mmmapiospace function (wdm.h), 2023.

[59] Microsoft. Object_attributes structure (ntdef.h), 2023.

[60] Microsoft. Osversioninfoexw structure (wdm.h), 2023.

[61] Microsoft. Zwcreatefile function (wdm.h), 2023.

[62] Microsoft. Zwmapviewofsection function (wdm.h), 2023.

[63] Microsoft. Zwopenfile function (wdm.h), 2023.

doi:10.6342/NTU202302404

[64] D. Molnar, X. C. Li, and D. A. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. PhD thesis, Peking University, USA, 2009.

[65] namazso. physmem_drivers, 2019.

[66] S. Sidiroglou-Douskos, E. Lahtinen, N. Rittenhouse, P. Piselli, F. Long, D. Kim, and M. Rinard. Targeted Automatic Integer Overflow Discovery Using Goal-Directed Conditional Branch Enforcement. PhD thesis, USA, 2015.

[67] T. Wang, T. Wei, Z. Lin, and W. Zou. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution. PhD thesis, Peking University, PRC, 2009.

[68] Yan Shoshitaishvili and Ruoyu (Fish) Wang and Audrey Dutcher and Lukas Dresel and Eric Gustafson and Nilo Redini and Paul Grosen and Colin Unger and Chris Salls and Nick Stephens and Christophe Hauser and John Grosen. angr/angr, 2023.

doi:10.6342/NTU202302404

doi:10.6342/NTU202302404

Appendix A — IOCTLance CVEs

Table A.1 presents the CVEs discovered by IOCTLance as of the paper's writing. Out of the 41 assigned CVEs, only 38 have been released and made public.

Table A.1: CVEs found by IOCTLance.

Vendor	Driver	Vulnerability	Disclosure
FabulaTech	ftwebcam.sys	v5	CVE-2023-1186
FabulaTech	ftwebcam.sys	v5	CVE-2023-1188
WiseCleaner	WiseFs64.sys	v5	CVE-2023-1189
Watchdog	wsdk-driver.sys	v9	CVE-2023-1453
WiseCleaner	WiseUnlock64.sys	v9	CVE-2023-1486
WiseCleaner	WiseHDInfo64.dll	v5	CVE-2023-1487
WiseCleaner	WiseHDInfo64.dll	v8	CVE-2023-1488
WiseCleaner	WiseHDInfo64.dll	v7	CVE-2023-1489
Max Secure	SDActMon.sys	v9	CVE-2023-1490
Max Secure	MaxCryptMon.sys	v9	CVE-2023-1491
Max Secure	MaxProc64.sys	v5	CVE-2023-1492
Max Secure	MaxProctetor64.sys	v5	CVE-2023-1493

Continued on next page

doi:10.6342/NTU202302404

Table A.1 – continued from previous page

Vendor	Driver	Vulnerability	Disclosure
JiangMin	kvcore.sys	v4	CVE-2023-1629
JiangMin	kvcore.sys	v5	CVE-2023-1630
AMD	AMDPowerProfiler.sys	v5	CVE-2023-20556
AMD	AMDCpuProfiler.sys	v5	CVE-2023-20561
AMD	AMDCpuProfiler.sys	v3	CVE-2023-20562
AMD	AMDRyzenMasterDriver.sys	v5	CVE-2023-20560
AMD	AMDRyzenMasterDriver.sys	v1	CVE-2023-20564
IObit	ImfRegistryFilter.sys	v5	CVE-2023-1638
IObit	ObCallbackProcess.sys	v5	CVE-2023-1640
IObit	ObCallbackProcess.sys	v5	CVE-2023-1641
IObit	ImfHpRegFilter.sys	v5	CVE-2023-1643
IObit	IMFCameraProtect.sys	v5	CVE-2023-1644
IObit	IMFCameraProtect.sys	v4	CVE-2023-1645
IObit	IMFCameraProtect.sys	v4	CVE-2023-1646
DriverGenius	mydrivers64.sys	v7	CVE-2023-1676
DriverGenius	mydrivers64.sys	v5	CVE-2023-1677
DriverGenius	mydrivers64.sys	v8	CVE-2023-1678
DriverGenius	mydrivers64.sys	v1	CVE-2023-1679
Microsoft	pgodriver.sys	v5	CVE-2023-28262
Microsoft	pgodriver.sys	v3	CVE-2023-28263
EnTech Taiwan	Se64a.sys	v8	CVE-2023-2870

Continued on next page

doi:10.6342/NTU202302404

Table A.1 – continued from previous page

Vendor	Driver	Vulnerability	Disclosure
FabulaTech	ftusbbus2.sys	v5	CVE-2023-2871
FLEXIHUB	fusbhub.sys	v5	CVE-2023-2872
Twister Antivirus	filppd.sys	v3	CVE-2023-2873
Twister Antivirus	filppd.sys	v5	CVE-2023-2874
eScan	PROCOBSRVESX.SYS	v5	CVE-2023-2875

doi:10.6342/NTU202302404