# Overview

I have found three vulnerabilities in both IOBitUnlocker and IOBitMalwareFighter. All these vulnerabilities lead to Local Privilege Escalation and will likely receive a high severity CVSS score of around 7 – 8. This whitepaper will first explain the vulnerabilities, then display proof of concepts for each one.

All screenshots of decomplication have been heavily edited and renamed. These functions and variable names were based on my best guesses of decompiled code and may not perfectly reflect real code and naming structure.

All programs have been tested with the most recent build of products at time of submission. Malware Fighter was on version 13.2.0, and Unlocker was on version 1.3.0.

## Table of Contents

# IOBitUnlocker Driver Vulnerability

## Introduction

The program IOBitUnlocker includes a driver, IOBitUnlocker.sys. This driver can be used to force unlock and move files through the kernel. When the IOBitUnlocker program is running, this driver is loaded. This driver can be communicated with from any process from any privileged, allowing for a medium integrity process to use all functions that IOBitUnlocker provides.

## Reverse Engineering

Decomplication also displays that this device validates the calling process through two checksum checks, which can easily be bypassed. By ensuring the proof-of-concept process passes these two checks, this allows for this proof of concept to directly issue commands to the driver without requiring high integrity.

Analysis shows that this driver is created insecurely. IoCreateDeviceSecure is not issued, meaning there is no restriction on the integrity of the process that interacts with the drivers.

```
local_30 = L"\\Device\\IObitUnlockerDevice";
local_28 = 62;
local_26 = 64;
local_20 = L"\\DosDevices\\IObitUnlockerDevice";
uVar2 = IoCreateDevice(param_1,0,&local_38,34,0,0,local_res18);
if (-1 < (int)uVar2) {
  uVar1 = IoCreateSymbolicLink(&local_28,&local_38);
  uVar2 = (ulonglong)uVar1;
  if ((int)uVar1 < 0) {
    IoDeleteDevice(local_res18[0]);
  }
```

This allows an unprivileged user mode process to directly send IRP requests to the driver.

However, just the ability to send requests does not mean that this driver is particularly vulnerable yet. Later in the decomplication process it was discovered that this driver verifies the process is IOBitUnlocker through two checksum checks. These values are hardcoded into the actual program. The below screenshot is a result of annotated driver code decompiled through Ghidra. The xorsum check functions by taking every byte of the file and performs an XOR with the next byte of the file. The second check is the alternating sum check. This check functions by adding the bytes. If the byte index is even, it adds a

byte to the total. If the byte index is odd, it subtracts the byte value from the total. The byte index is the specific position of a byte within the file.

```
    sigchecker(filename,alternatingsum,xorsum);
    filename[4] = 0;
    local_1e = 0;
    local_1a = 0;
    filename[0] = 0;
    filename[1] = 0;
    ExFreePoolWithTag(puVar3,1768776039);
    if ((alternatingsum[0] == 43) && (xorsum[0] == 11044471)) {
      return 0;
    }
  }
  return 3221225473;
```

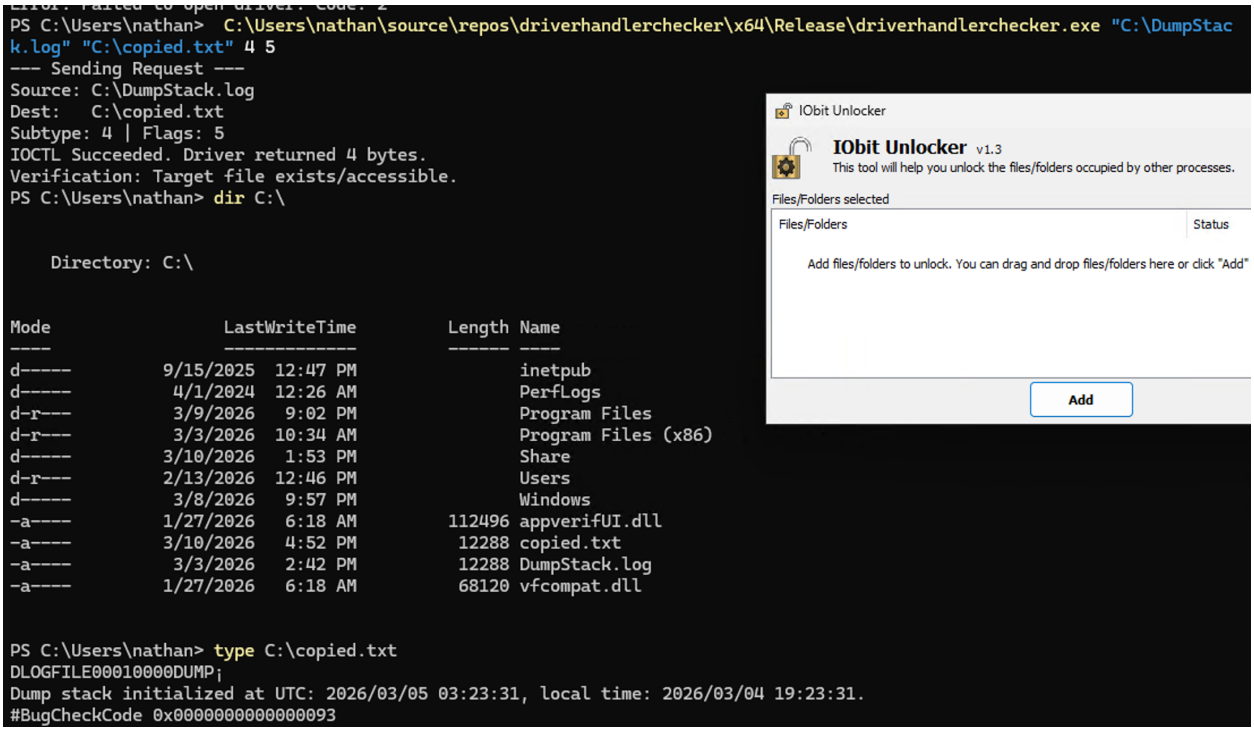
This checking system is insufficient to secure a device driver. Instead, simply ensure the device is created securely.

To exploit this, I created a program to add these hardcoded checksums to any program. This executable is included in the Unlocker folder. Below is a screenshot of how to call and run it.

```
PS C:\Users\nathan\source\repos\driverhandlerchecker\x64\Release> .\checksumpatcher.exe .\driverhandlerchecker.exe
Original File Size: 235449 bytes
Current Alternating Sum: 11044471
Current XOR Sum: 43
File already has the correct checksums. No changes needed.
PS C:\Users\nathan\source\repos\driverhandlerchecker\x64\Release>
```

The included proof of concept is called driverhandlerchecker. Both the executable and source code are included in the Unlocker folder. I have added functionality for both move and copy operations, and flags can be specified to decide what degree of forcing will be done.

In the below example, a file in the root of the C drive was copied ot the root of the C drive under a different name. The permissions are not retained after the copy.




This allows any user to copy or move any file they wish. This can be used alongside DLL sideloading for local privilege escalation or can be used to steal passwords and other protected files on a computer.

### Patching

To patch this vulnerability, ensure that the driver is created securely so only processes with high integrity can interact with it.

## Malware Fighter Driver Vulnerability

### Introduction

This device exposes a driver labeled IMFForceDelete123. The driver for this file is located at "C:\Program Files (x86)\IObit\IObit Malware Fighter\Drivers\win10_amd64\IMFForceDelete.sys”

The referenced driver can be fully communicated with and the delete function can be called by a medium integrity process. By deleting a DLL file, the attacker can place their own malicious DLL within a writeable directory with a proxied version of the same DLL. With this example, the DLL chosen was a DLL within the IObit Malware Fighter program, however, many other DLL’s on the system can be chosen for this local privilege escalation. The referenced driver runs in the background while the computer remains on.

## Reverse Engineering

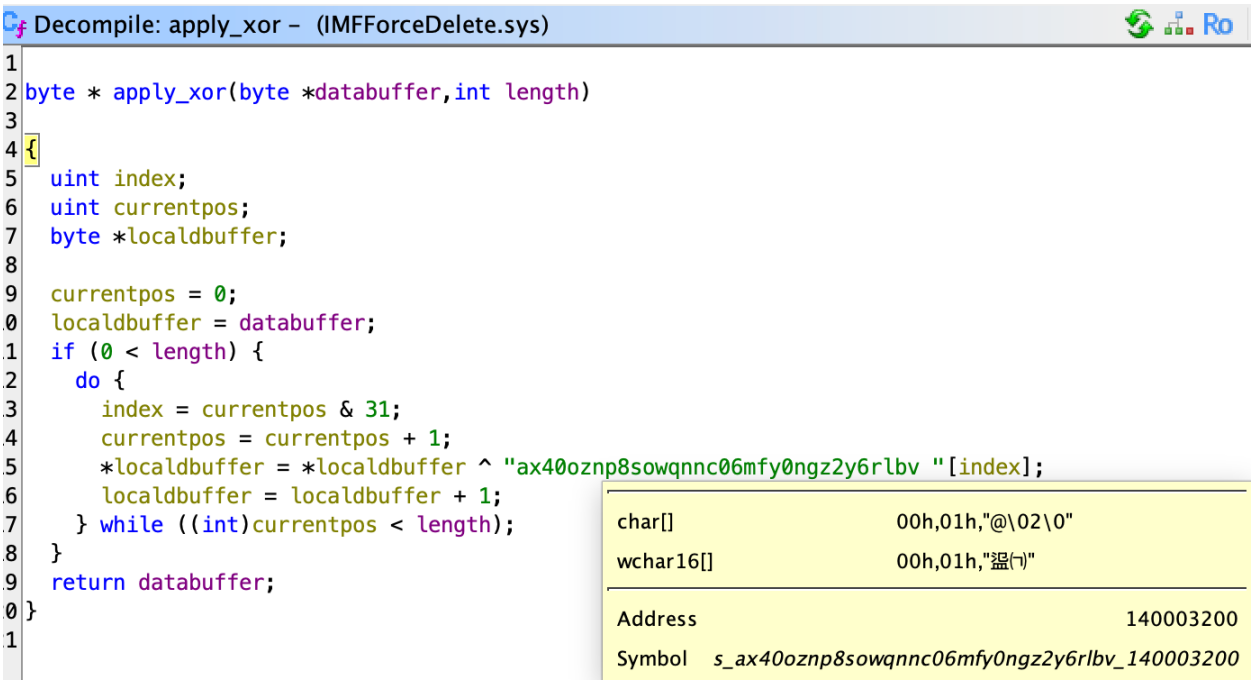
Similarly to the last driver, this is all possible because the device is created without IOCreateDeviceSecure and a security string. Any medium integrity process may interact with this driver.

```
local_res18[0] = 0;
RtlInitUnicodeString(local_28,L"\\Device\\IMFForceDelete123");
RtlInitUnicodeString(local_18,L"\\DosDevices\\IMFForceDelete123");
uVar2 = IoCreateDevice(param_1,0,local_28,34,256,0,local_res18);
if (-1 < (int)uVar2) {
  uVar1 = IoCreateSymbolicLink(local_18,local_28);
  uVar2 = (ulonglong)uVar1;
  if ((int)uVar1 < 0) {
    IoDeleteDevice(local_res18[0]);
  }
  else {
```

Further analysis of the available function calls in this driver show how this driver handles interfacing. This driver reverses a string, decodes it through base64, and applies an xor operation to that string with a hardcoded key.

```
if (-1 < filepathlength) {
  filepath = stringreverser((longlong)kernelstringloc,(uint)(ushort)stringstruct,filepatharr
                            );
  if ((char)filepath != '\0') {
    filepath = base64decoder((longlong)filepatharr,(uint)(ushort)stringstruct,
                             (longlong)filepatharr);
    filepathlength = (int)filepath;
    if (filepathlength - 1U < 2047) {
      filepatharr[filepathlength] = 0;
      apply_xor(filepatharr,filepathlength);
      memset(outbuffer,0,(undefined1 *)2048);
      errorcode = RtlUTF8ToUnicodeN(outbuffer,2047,local_res18,filepatharr,filepathlength);
      goto LAB_140001401;
    }
```

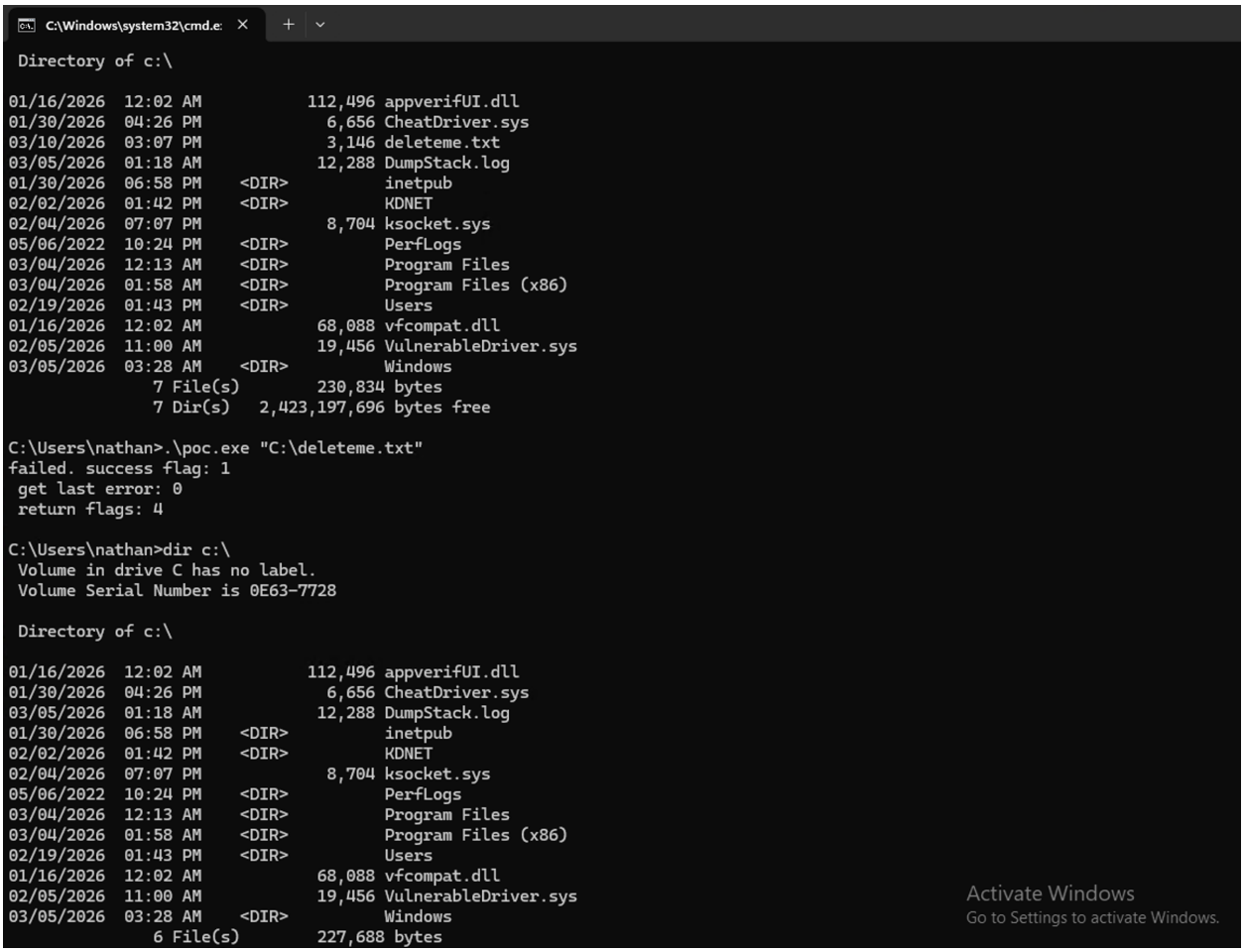
Decompiling of this function displays the hardcoded key within the program used for encoding the file path.




Therefore, to exploit this program, an attacker must encode their string with the above key, encode it with base64, then reverse the string. Finally, it must be sent to the driver with the corresponding IOCTL code. This is the code that determines the correct operation to continue with. Further decomplication displayed the code required.

## Proof of Concept

A proof of concept was written, and the source code and compiled version is attached in the MalwareFighterDriver folder. To use this proof of concept, put the file that must be deleted as a command line argument.




## DLL Proxying to Local Privilege Escalation

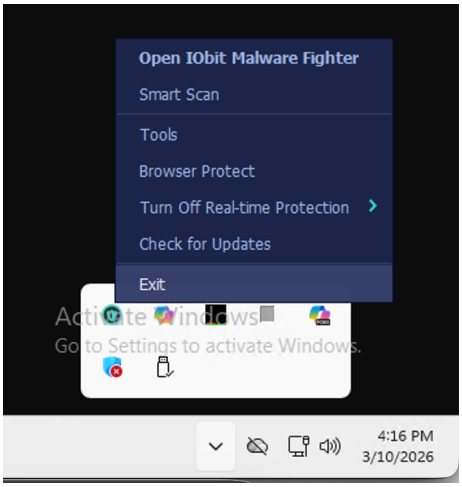
If LoadLibrary is called without specifying a full path of a DLL, deleting a DLL will cause windows to search through different search locations, eventually searching through the User's Path. By default, this Path folder is writeable without elevated privileges, and is set to "C:\users\username\Appdata\local\Microsoft\WindowsApps". The DLL search order can be further explained in this Microsoft Blog [1].

The ProductNews2 library was chosen and deleted from "C:\Program Files (x86)\IObit\IObit Malware Fighter\ProductNews2.dll" to start.

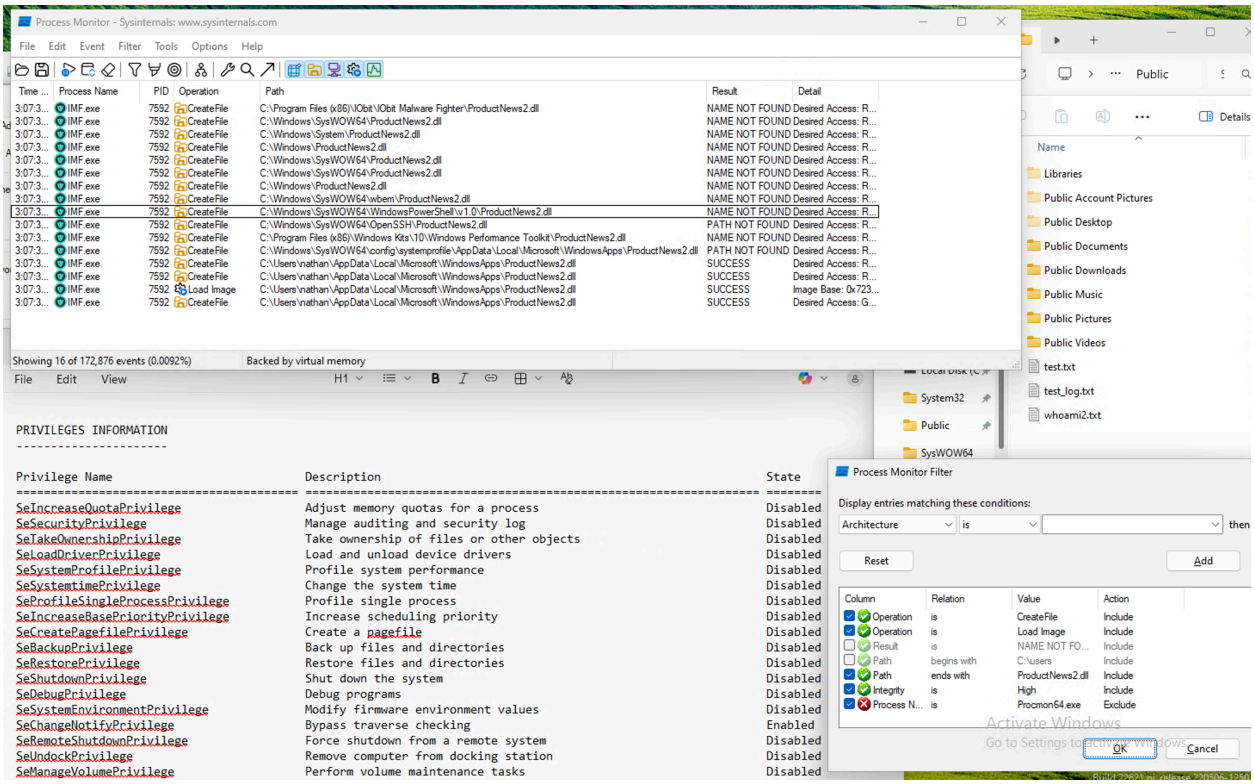
By deleting or moving a DLL file from its typical location, this allows an attacker to place their own DLL with the exact same name as the deleted DLL in their Path folder. However, the DLL will not load unless all functions are proxied to a copy of the original DLL. To automate this proxying, the tool SharpDllProxy [2] was used. This tool creates a copy of the original DLL, and generated boilerplate code that proxies' functions to that DLL. Below is the example of this call.

```
PS C:\Users\nathan> C:\Users\nathan\source\repos\SharpDllProxy\SharpDllProxy\bin\Debug\netcoreapp3.1\SharpDllProxy.exe -
dll "C:\Program Files (x86)\IObit\IObit Malware Fighter\ProductNews2.dll"
[+] Reading exports from C:\Program Files (x86)\IObit\IObit Malware Fighter\ProductNews2.dll...
[+] Redirected 15 function calls from ProductNews2.dll to tmp7C92.dll
[+] Exporting DLL C source to C:\Users\nathan\output_ProductNews2\ProductNews2_pragma.c
PS C:\Users\nathan> dir
```

A new DLL project was created with the contents of ProductNews2_Pragma.c. The source code for this project is available, along with a compiled executable, in the MalwareFighterDriver folder. The current source code displays the privileges of the current process in C:\users\public\whoami2.txt. This exploit only triggers on a restart, or if the Malware Fighter programmed is restarted. To restart this program, a user may select exit in the Windows application menu.




Through selecting exit, and relaunching, this exploit can be triggered. Both the ProductNews2.dll and the temporary DLL must be copied to the user's Path folder. Below is a screenshot of the exploit functioning, with the proxied DLL loading with high integrity.




## Patching

To patch this vulnerability, ensure that the driver is created securely so only processes with high integrity can interact with it.

# Malware Fighter DLL Vulnerability

## Introduction

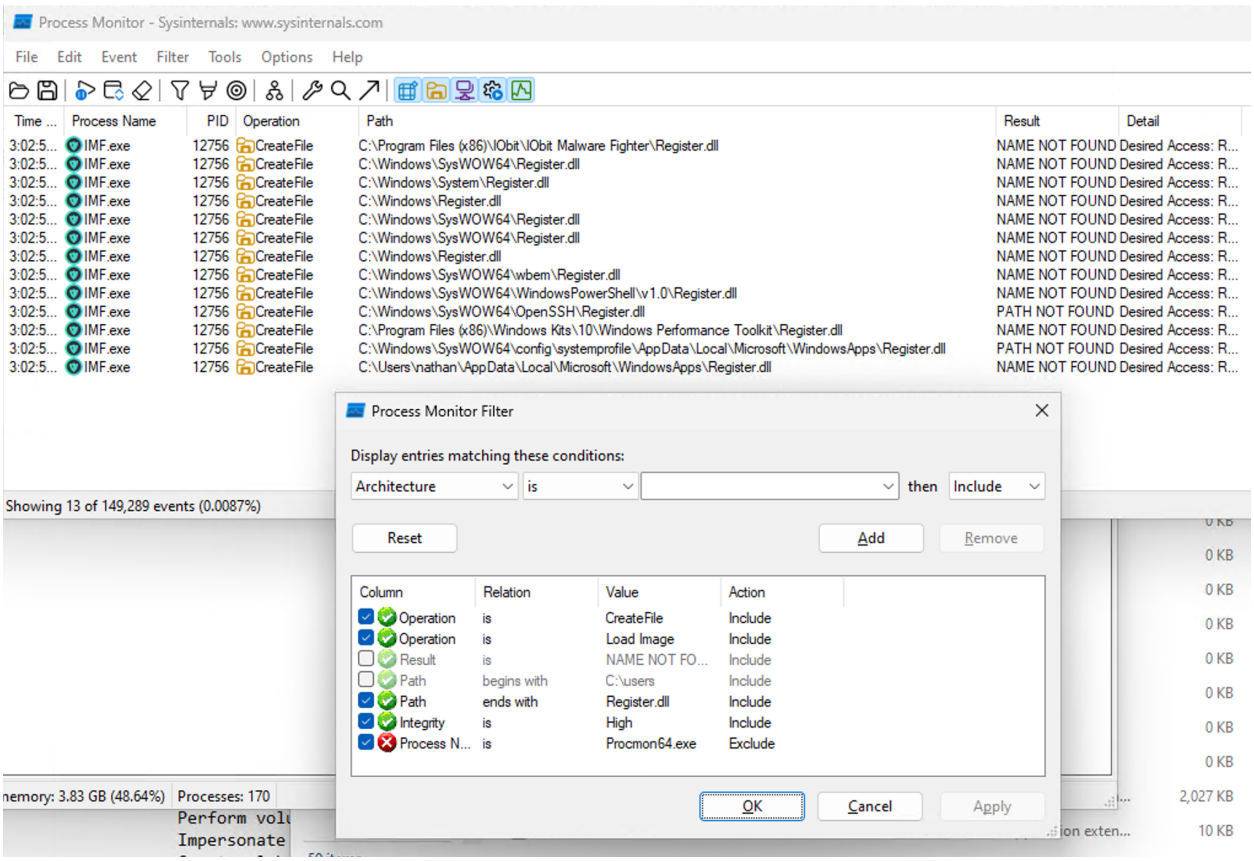
Malware Fighter, on startup, searches for a DLL that does not exist. Due to the DLL search order, which has previously been explained in this paper, an attacker can simply place their DLL in their user's Path folder. This folder is writeable by default. An attacker can simply place a DLL in their Path with the correct name, and this DLL will now run with High integrity when Malware Fighter is re-launched, or the computer is restarted.

Unlike the last vulnerability, to trigger this vulnerability the application's GUI simply needs to be opened.

## Analysis

Process Monitor by Sysinternals [3] was used to discover this vulnerability. The filter applied is visible in this screenshot. This filter searches for all DLL's that were not found but

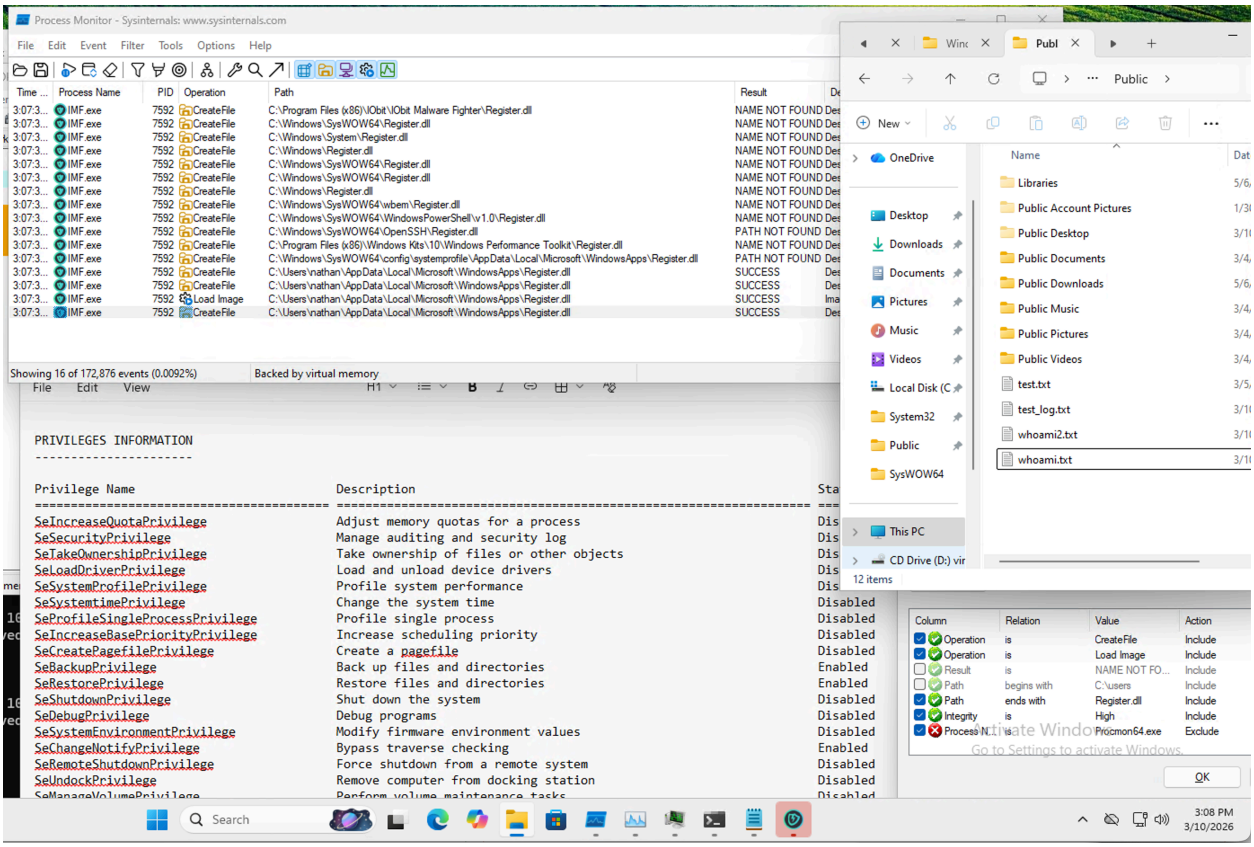
referenced and searched for by the program under high integrity. The DLL search order is visible here.




The last path checked on the DLL search order is the user's local AppData folder. This is last in the DLL search order, because the user can write any file to this path without requiring excess permissions. To complete this exploit, I compiled my own DLL and named it Register.dll. Once this DLL was placed in the displayed AppData path, it loaded with high integrity.

To display proof of exploitation, I compiled a DLL and placed a function in DLLMain, a function which executes when a DLL is loaded into a process's memory. This function ran whoami /priv and redirected the output to the C:\users\public\whoami.txt public folder. The source code for this DLL, and a compiled version, is attached in MalwareFighterDLL folder. The below folder shows my Register.dll being called after Malware Fighter is opened. The integrity is High, and this results in a local privilege escalation, as shown in the whoami

output.




This exploit can be launched by launching Malware Fighter even if the service is already running.

## Patching

To patch this vulnerability, remove insecure load library call to Register.dll. Call this DLL through its full path. This will completely remove this vulnerability.

# Works Cited

[1] Microsoft, "Dynamic-link library search order," 8 2023. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/dlls/dynamic-link-library-search-order.

[2] M. L, "SharlDllProxy," 07 2020. [Online]. Available: https://github.com/Flangvik/SharpDllProxy.

[3] M. Russinovich, "Process Monitor v4.01," 06 2024. [Online]. Available: https://learn.microsoft.com/en-us/sysinternals/downloads/procmon.