BOHOPS

A blog about cybersecurity research, education, and news

WRITTEN BY BOHOPS

OCTOBER 15, 2020

EXPLORING THE WDAC MICROSOFT RECOMMENDED BLOCK RULES: VISUALUIAVERIFYNATIVE

QUICK LINKS

- Leveraging INF-SCT Fetch & Execute Techniques For Bypass, Evasion, & Persistence (Part 2)
- Abusing .NET Core CLR Diagnostic Features (+ CVE-2023-33127)
- Abusing the COM Registry Structure (Part

INTRODUCTION

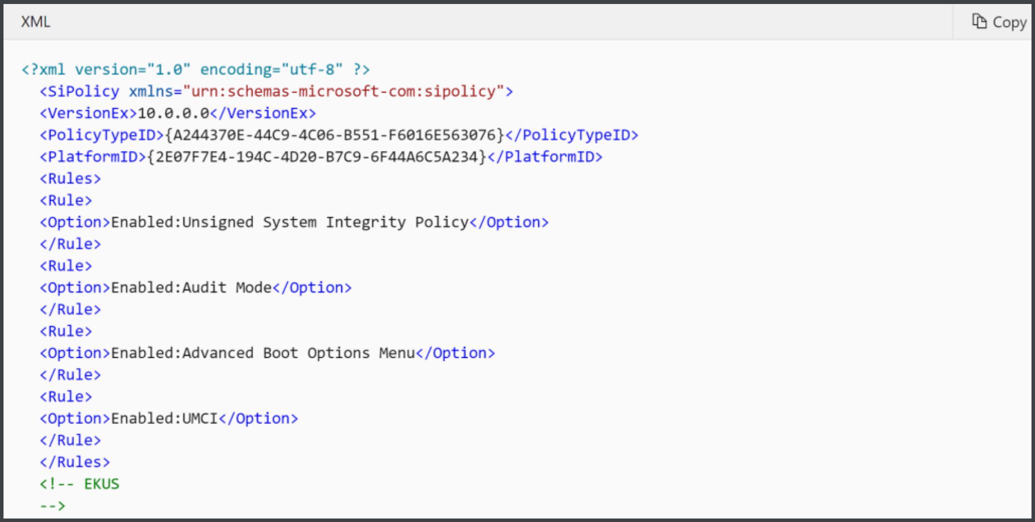
If you have followed this blog over the last few years, many of the posts focus on techniques for bypassing application control solutions such as Windows Defender Application Control

2): Hijacking & Loading Techniques
- Leveraging INF-SCT Fetch & Execute Techniques For Bypass, Evasion, & Persistence
- Abusing the COM Registry Structure: CLSID, LocalServer32, & InprocServer32
- WS-Management COM: Another Approach for WinRM Lateral Movement
- DiskShadow: The Return of VSS Evasion, Persistence, and Active Directory Database Extraction
- Analyzing and Detecting a VMTools Persistence Technique
- Investigating .NET CLR Usage Log Tampering Techniques For EDR Evasion (Part 2)
- Executing Commands and Bypassing AppLocker with PowerShell Diagnostic Scripts

(WDAC)/Device Guard and AppLocker. I have not been blogging as much lately but wanted to get back into the rhythm and establish a similar theme for at least the next few posts by exploring the 'forgotten' lolbins on the WDAC Microsoft Recommended Block Rules page.

MICROSOFT BLOCK RULES PRIMER

If you are familiar with WDAC, you likely have come across the Recommended Block Rules page at some point and have noticed the interesting list of binaries, libraries, and the never ending XML formatted WDAC block rules policy. Microsoft recommends merging the block rule policy with your existing policy if your IT organization uses WDAC for application control.



```xml
<?xml version="1.0" encoding="utf-8" ?>
  <SiPolicy xmlns="urn:schemas-microsoft-com:sipolicy">
  <VersionEx>10.0.0.0</VersionEx>
  <PolicyTypeID>{A244370E-44C9-4C06-B551-F6016E563076}</PolicyTypeID>
  <PlatformID>{2E07F7E4-194C-4D20-B7C9-6F44A6C5A234}</PlatformID>
  <Rules>
  <Rule>
  <Option>Enabled:Unsigned System Integrity Policy</Option>
  </Rule>
  <Rule>
  <Option>Enabled:Audit Mode</Option>
  </Rule>
  <Rule>
  <Option>Enabled:Advanced Boot Options Menu</Option>
  </Rule>
  <Rule>
  <Option>Enabled:UMCI</Option>
  </Rule>
  </Rules>
  <!-- EKUS
  -->
```

WDAC is considered a formal security boundary. Novel circumvention of an enforced code integrity policy (e.g. executing unsigned arbitrary code) may result in a CVE from Microsoft if patched or a few added deny rules within the block rules policy (dedicated in your honor, of course).

The decision making process for deciding wither to either service a bypass vulnerability or add a policy mitigation is something that I do not fully understand. Speculatively, the decision tree is complex, and I'd imagine that decision usually boils down to impact, cost (level of effort), and time. Regardless, the block rules policy is essential for mitigating the residual risk of discovered WDAC bypass techniques…especially those caused by pesky lolbins.

THERE WAS SOMETHING MENTIONED ABOUT *FORGOTTEN*…

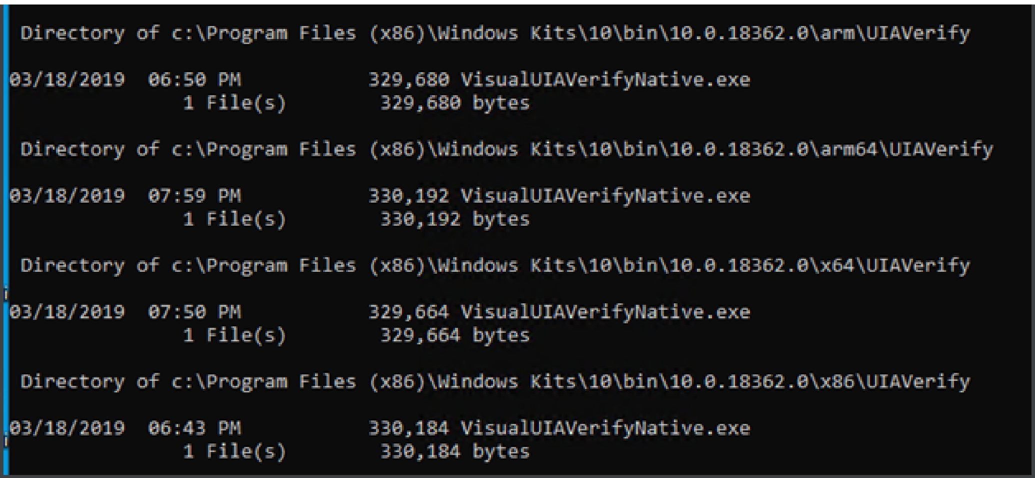
In the last few years, there have been a lot of great posts and presentations about WDAC internals and circumvention. I decided to centralize the various public write-ups for easier accessibility in a common place (which is a work in progress) as well as (somehow) unravel the mysteries behind the publicly undocumented techniques of those lolbins on the block rules page, which still may have utility for a variety of use cases (e.g. app control policy oversight). My simple notes on the matter can be accessed here until a better solution is adopted. Now, Let's take a look at one of these interesting lolbins: VisualUiaVerifyNative.

CIRCUMVENTING WDAC WITH VISUALUIAVERIFYNATIVE

While going through the 'undocumented' candidates on the WDAC list, VisualUiaVerifyNative stuck out to me because I recalled seeing it somewhere while learning more about WDAC a few years ago. It turns out that it was actually mentioned within this pull request for RunScriptHelper by Matt Graeber (@mattifestation) in 2017. The original discovery was made by Lee Christensen (@tifkin_), whose discovery methodology for finding bugs is likely way more polished than my own :). Fortunately for us, we already know which lolbin to examine, so the hardest part is done. Let's dive in to see if we can make sense of how VisualUiaVerifyNative can be used to bypass WDAC.

VISUALUIAVERIFYNATIVE BACKGROUND

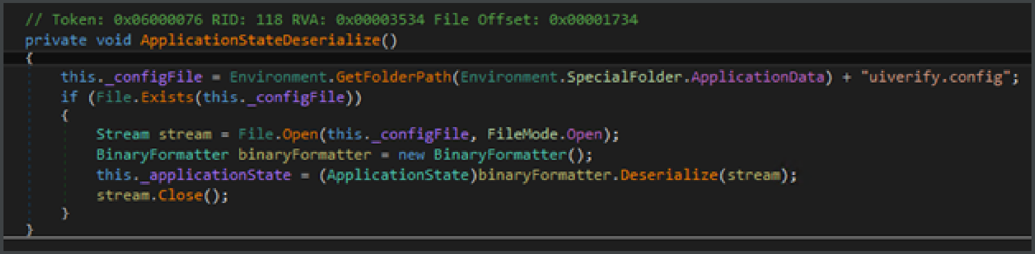
VisualUiaVerifyNative (visualuiaverifynative.exe) is the GUI executable binary for UI Automation Verify, a "testing framework for manual and automated testing of a control's or application's implementation of Microsoft UI Automation" (Microsoft Docs). VisualUiaVerifyNative is included with the Windows Software Development Kit (SDK). On our WDAC test machine, various instances of the binary are located in the following depicted paths:



```
 Directory of c:\Program Files (x86)\Windows Kits\10\bin\10.0.18362.0\arm\UIAVerify

03/18/2019  06:50 PM           329,680 VisualUIAVerifyNative.exe
               1 File(s)        329,680 bytes

 Directory of c:\Program Files (x86)\Windows Kits\10\bin\10.0.18362.0\arm64\UIAVerify

03/18/2019  07:59 PM           330,192 VisualUIAVerifyNative.exe
               1 File(s)        330,192 bytes

 Directory of c:\Program Files (x86)\Windows Kits\10\bin\10.0.18362.0\x64\UIAVerify

03/18/2019  07:50 PM           329,664 VisualUIAVerifyNative.exe
               1 File(s)        329,664 bytes

 Directory of c:\Program Files (x86)\Windows Kits\10\bin\10.0.18362.0\x86\UIAVerify

03/18/2019  06:43 PM           330,184 VisualUIAVerifyNative.exe
               1 File(s)        330,184 bytes
```

VISUALUIAVERIFYNATIVE ANALYSIS

We quickly discover that VisualUiaVerifyNative is a .NET application. Using dnSpy to 'decompile' to source code, we come across a very interesting function called *ApplicationStateDeserialize()* in the *MainWindow* class. At first glance, a configuration file with a suffix of *uiverify.config* appears to be loaded and deserialized via *BinaryFormatter.Deserialize()*:



```
// Token: 0x06000076 RID: 118 RVA: 0x00003534 File Offset: 0x00001734
private void ApplicationStateDeserialize()
{
    this._configFile = Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "uiverify.config";
    if (File.Exists(this._configFile))
    {
        Stream stream = File.Open(this._configFile, FileMode.Open);
        BinaryFormatter binaryFormatter = new BinaryFormatter();
        this._applicationState = (ApplicationState)binaryFormatter.Deserialize(stream);
        stream.Close();
    }
}
```

The first time VisualUIAVerifyNative is executed, the configuration file does not exist but the program will attempt to locate it anyway. If not found, the file is simply created :

In the previous screenshot, we can see that the file is actually named *Roaminguiverify.config* and stored in the user's \AppData directory:

Interestingly, the naming of the file appears to be as a simple oversight as it missing an escaped "\". The file should reside in the user's \AppData\Roaming directory (if I had to guess).

Regardless, *Roaminguiverify.config* is successfully read into the program if it exists as verified by the following Procmon ETW trace:

Furthermore, A simple screen print of the *Roaminguiverify.config* contents shows that it is in a serialized format:

Returning the attention to VisualUIAVerifyNative, we note that the application is Microsoft signed. As such, it will likely be able to run when a WDAC policy is enforced at (least) the PcaCertificate level:

To test simple exploitation, a serialized payload is built with the Ysoserial.net project as follows:

```
ysoserial.exe -f BinaryFormatter -g
TextFormattingRunProperties -o raw -c "notepad" >
Roaminguiverify.config
```

The following screenshot verifies payload creation:

In this case, the YoSoSeial.Net *TextFormattingRunProperties* gadget is leveraged for simplicity as it builds a XAML command payload with the .NET Process class (via System.Diagnostics) to execute a command (notepad.exe in this case). After replacing *Roaminguiverify.config* with our same named serialized file, payload execution is successful when visualuiaverifynative.exe is launched:

> *Note: An exception pops up in this case because it is not the expected data.

Excellent! We have an eligible Microsoft signed binary and a deserialization primitive that we may be able to abuse for circumventing WDAC. Let's put it to the test…

WDAC CONFIGURATION

To test the use of VisualUIAVerifyNative deserialization as a vector for application control bypass, a WDAC/Device Guard Code Integrity policy is configured based on the directions located here. For this configuration, a scan policy is created at the PCA certificate level. However, we do not merge the Block Rules policy as stated in the directions so that VisualUIAVerifyNative has a chance to execute

:). The following screenshot demonstrates how a WDAC policy is enforced and loaded (following a reboot):

After rebooting and logging in as a low privileged user, we can validate whether the policy is enforced by checking MSInfo32.exe:

Before validating the bypass, let's perform a quick test to run something that should fail. In this test case, we'll run a simple Jscript payload with cscript.exe. As expected, the COM object cannot be created via WDAC Code Integrity policy enforcement:

Next, we copy our serialized payload to the path we expect VisualUIAVerifyNative to read the serialized payload file. In this case, the following path is expected under the lowpriv account:

```
C:\Users\lowpriv\AppData\Roaminguiverify.config
```

Lastly, we launch VisualUIAVerifyNative and see that the serialized payload is executed accordingly. Fantastic!

DEFENSIVE CONSIDERATIONS

- Although not all bypass techniques have been disclosed for block list lolbins, there is still residual risk for opportunistic abuse.
- If you deploy WDAC within your environment, consider merging the block rules with your current WDAC policy. If you prefer to go the EDR route, consider integrating analytics/queries to observe blocklist lolbin behavior.
- If enforcement policies are not ideal for your environment, consider using WDAC in audit mode for added visibility and telemetry to compliment other security solutions.
- For a more interesting overview of Application Control solutions (including WDAC) and links to other great researcher resources, refer to this post.

CONCLUSION

Thanks for taking the time out of your busy day to read this post. I plan to follow up with a few similar posts unless others beat me to the punch (which is very much welcome).

Take Care,

~ Bohops

SHARE THIS:

Twitter Facebook

Loading...

ONE THOUGHT ON "EXPLORING THE WDAC MICROSOFT RECOMMENDED BLOCK RULES: VISUALUIAVERIFYNATIVE"

Pingback: Exploring the WDAC Microsoft Recommended Block Rules (Part II): Wfc.exe, Fsi.exe, and FsiAnyCpu.exe – | bohops |

Comments are closed.

PREVIOUS POST

NEXT POST