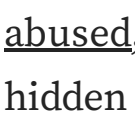
Code Signing Certificate Cloning Attacks and Defenses



Matt Graeber · Follow
Published in Posts By SpecterOps Team Members · 11 min read · Dec 22, 2017

Before reading this post, ponder the following question: “What does it *actually* mean to you for something to be signed by Microsoft (or any vendor for that matter)?”

Introduction: SOC Analyst Autoruns Baselining Scenario

Imagine you’re working in a SOC and you’re tasked with baselining persistence entries across 40,000 hosts. You’re tasked specifically with inspecting run key persistence. You have Sysinternals deployed across the enterprise, you run Autoruns across every system, and forward the results to a Splunk dashboard that allows you to easily interpret the results. The smart SOC analyst you are knows that signed Microsoft applications can be abused, so you make sure that “Microsoft” and “Windows” entries are not hidden when running autorunsc.exe. You cluster all of the common results together and start focusing on outliers in the data set. You find the following outlier on

6 systems out of 40,000:

```
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
   SecurityAudit
     C:\Windows Defender\MpCmdRun.exe
     Microsoft Malware Protection Command Line Utility
     (Verified) Microsoft Corporation
     4.12.16299.15
     c:\windows defender\mpcmdrun.exe
     11/25/1912 5:39 AM
```

You apply the following process to determine whether the entry is benign or suspicious:

1. You note that the binary is a verified “Microsoft Corporation” binary. Knowing that it is signed by Microsoft allows you to scrutinize it less since this particular signed binary is not known to have been abused by attackers.

2. You Google MpCmdRun.exe and confirm that it is indeed associated with Windows Defender.
3. You enabled VirusTotal integration with Autoruns (assuming your organization has accepted that risk) and it comes back with 0 positive AV hits.
4. You're still unsure as to why it's an outlier but your enterprise is a large, heterogenous environment where the concept of a baselined gold image does not exist.
5. You accept that it's an outlier but you are confident that MpCmdRun.exe isn't being abused in the wild and you subsequently filter future hits of this hash. After all, you have many more outliers to wade through.

Does this scenario sound familiar to anyone? Unfortunately, as much as I hate to say it, that Autoruns entry was positive evidence of compromise and you overlooked it and decided to overlook it in the future as well.

. . .

Certificate Chain Cloning and Cloned Root Trust Attacks

What our SOC analyst failed to pick up on was the fact that MpCmdRun.exe was signed using a cloned Microsoft certificate chain where the attacker also trusted their cloned root certificate on the compromised victim systems. How might an attacker go about performing such an attack? The steps can be summarized as follows:

1. Export all certificates in a legitimate certificate chain to disk. These certificates are what you'll be using as a template for your own cloned certificate chain.
2. Build a cloned certificate chain using the chain that was exported to disk. The New-SelfSignedCertificate cmdlet in PowerShell has very convenient "-CloneCert" and "-Signer" parameters to enable this. Upon cloning the chain, you will be able to sign malicious code with the cloned certificate chain.
3. You'll also want to export the cloned root certificate as you will need to trust this certificate on the victim system in order for any of your signed, malicious code to verify properly and blend in with many security tools.

The following video shows the manual process of exporting the certificate chain used to sign kernel32.dll:

Manually exporting a legitimate Microsoft certificate chain to disk to use as a template for cloning.

Now that the Microsoft certificate chain has been exported to disk, you can now use it as a template for building a spoofed Microsoft certificate chain. The following code was used to achieve this:

The following video demonstrates running the code above:

So why does this attack work? Well, at a high level, digital signature validation relies upon the following:

1. Integrity validation — Does the hash of the file match the signed hash in the signature? If not, the integrity of the file has been compromised and it should not be trusted.
2. Certificate chain validation — Was each certificate in the chain properly issued by its parent?
3. Certificate validity check — If each certificate in the chain is not timestamped, is each certificate within its stated validity time frame? If the digital signature is timestamped, validate the timestamping certificate counter-signature chain.
4. Revocation check — Are any of the certificates in the chain revoked or explicitly untrusted by an administrator?
5. Root CA validation — Is the root certificate in the signer chain a trusted certificate?

Technically, our cloned certificate chain passes all of these checks so any tool that performs signature validation (sigcheck, autoruns, procexp, AV?, etc.) will likely be fooled.

You may have noticed in the video, upon installation of the root certificate in the “CurrentUser” certificate store, a dialog popped up asking if you trust the certificate. If running in an elevated context, that popup will not occur. Why non-admin users are able to trust root CA certificates is beyond my comprehension. That should not be permitted in any organization.

. . .

Attack Weaponization

The video above showed a demo of how to create and trust a cloned root certificate locally. Ideally, in a real-world attack scenario, you wouldn't clone a certificate chain and sign your malicious file on a compromised system. Rather, you would build the cloned chain and sign your malicious code on an attacker system. Now, the problem remains however of how you would realistically trust the cloned CA certificate on the victim system. You could probably get away with dropping it to disk and installing it but if you wanted to be a bit stealthier, as an admin, you could install and trust the certificate directly in the registry. The following is an example of how you could use WMI to remotely install and trust a cloned root CA certificate:

In this example, $EncodedCertBlob is just the contents of the exported cloned root CA .cer file base64-encoded. $CertThumbprint is the thumbprint value (i.e. SHA1 hash of the certificate). So, upon installation of that certificate, any code signed with a certificate from that CA will properly validate. In this particular case, the code will additionally give the appearance of being Microsoft-signed code.

. . .

Detecting Malicious Root CA Certificate Installation

Considering the root of this attack involves installation of a root CA certificate, this action will be the focus of building a detection. The

installation of root CAs should be sufficiently uncommon such that a high-fidelity alert should be possible by monitoring the registry. Sysmon serves this purpose really well and what follows is an ideal config for catching root certificate installation:

When an event fires, it would look like the following:

```
Registry value set:
EventType: SetValue
UtcTime: 2017-12-20 17:12:11.999
ProcessGuid: {7ed59fb9-99eb-5a3a-0000-00102ab1af06}
ProcessId: 4404
Image: C:\WINDOWS\system32\wbem\wmiprvse.exe
TargetObject:
HKLM\SOFTWARE\Microsoft\SystemCertificates\ROOT\Certificates\1F3D3
8F280635F275BE92B87CF83E40E40458400\Blob
Details: Binary Data
```

Using this rule set, you will likely get a lot of CreateKey event false positives. The high-fidelity events to pay attention to are SetValue events where the TargetObject property ends with “<THUMBPRINT_VALUE>\Blob” as this indicates the direct installation or modification of a root certificate binary blob. Unfortunately, as of this writing, Sysmon configurations don’t allow sufficient granularity to constrain a set of registry events to a specific EventType nor are wildcards permitted in rule entries.

So the next question to ask yourself would be, “how do I know if this root certificate installation is ‘malicious’?” A logical first step would be to investigate the contents of the certificate to see if anything stands out. PowerShell makes inspecting certificates really easy.

```
Get-ChildItem -Path Cert:\ -Recurse | Where-Object { $_.Thumbprint
-eq '1F3D38F280635F275BE92B87CF83E40E40458400' } | For
mat-List *
```

The result of running this command might produce the following output:

```
PSPath                   :
Microsoft.PowerShell.Security\Certificate::LocalMachine\Root\1F3D3
8F280635F275BE92B87CF83E40E40458400
PSParentPath             :
Microsoft.PowerShell.Security\Certificate::LocalMachine\Root
PSChildName              :
1F3D38F280635F275BE92B87CF83E40E40458400
PSDrive                  : Cert
PSProvider               :
Microsoft.PowerShell.Security\Certificate
PSIsContainer            : False
EnhancedKeyUsageList     : {}
DnsNameList              : {Microsoft Root Certificate Authority
2010}
SendAsTrustedIssuer      : False
EnrollmentPolicyEndPoint :
Microsoft.CertificateServices.Commands.EnrollmentEndPointProperty
EnrollmentServerEndPoint :
Microsoft.CertificateServices.Commands.EnrollmentEndPointProperty
PolicyId                 :
Archived                 : False
Extensions               : {System.Security.Cryptography.Oid,
System.Security.Cryptography.Oid,
                           System.Security.Cryptography.Oid}
FriendlyName             :
IssuerName               :
System.Security.Cryptography.X509Certificates.X500DistinguishedNam
e
NotAfter                 : 11/30/2042 9:06:37 PM
NotBefore                : 12/1/2017 1:55:14 PM
HasPrivateKey            : False
PrivateKey               :
PublicKey                :
System.Security.Cryptography.X509Certificates.PublicKey
RawData                  : {48, 130, 5, 219...}
SerialNumber             : 52761736EEA4458142453E2D73FA89B2
SubjectName              :
System.Security.Cryptography.X509Certificates.X500DistinguishedNam
e
SignatureAlgorithm       : System.Security.Cryptography.Oid
Thumbprint               :
1F3D38F280635F275BE92B87CF83E40E40458400
Version                  : 3
Handle                   : 1849876297952
Issuer                   : CN=Microsoft Root Certificate Authority
2010, O=Microsoft Corporation, L=Redmond, S=Washington, C=US
Subject                  : CN=Microsoft Root Certificate Authority
2010, O=Microsoft Corporation, L=Redmond, S=Washington, C=US
```

To any observer, this certificate definitely has the “look and feel” of a legitimate certificate but what is it exactly that makes a certificate

"legitimate" or trusted? That process will be described in the last section of the post.

. . .

Preventing Malicious "CurrentUser" Root CA Certificate Installation

In the video demonstrating the root CA installation, it was performed in the current user context. While there may not be strong preventative mitigations for certificate installation as an admin, it is possible to prevent root certificate installation in the current user context by setting the following registry value:

```
HKLM\SOFTWARE\Policies\Microsoft\SystemCertificates\Root\Protected
Roots - Flags (REG_DWORD) - 1
```

While this registry key is not well documented online, wincrypt.h in the Windows SDK provides some contextual clues regarding the options available to set in the "Flags" value. The following relevant flag values are documented in the header file:

```
// Set the following flag to inhibit the opening of the
CurrentUser's
// .Default physical store when opening the CurrentUser's "Root"
system store.
// The .Default physical store open's the CurrentUser
SystemRegistry "Root"
// store.
#define CERT_PROT_ROOT_DISABLE_CURRENT_USER_FLAG    0x1

// Set the following flag to inhibit the adding of roots from the
// CurrentUser SystemRegistry "Root" store to the protected root
list
// when the "Root" store is initially protected.
#define CERT_PROT_ROOT_INHIBIT_ADD_AT_INIT_FLAG     0x2
```

After setting this key, you will get an access denied error when attempting to install a root CA to the CurrentUser Root certificate store.

So while not the most robust of preventative techniques, preventing non-admin users from trusting their own root CAs is certainly a strong policy to enforce in your organization.

As with any enforced preventative measure, an admin will need to consider "what might this break in my environment?" As with any preventative measure, it is important to roll them out in phases across an environment. If for whatever reason there is a business justification for permitting any user to trust a root certificate, you accept that an attacker or rogue software can trust arbitrary root certificates as well. Windows administrators will always have the ability to push trusted root certificates via Group Policy. A recent case where software installed its own root certificate without alerting the user was a Savitech audio driver. In this case, you would have needed to be admin to trust this root certificate but arbitrary root certificates have no basis for the establishment of trust compared to the arduous steps required to get your root certificate trusted by Microsoft.

. . .

Proper Validation of Root CA Trust

Until recently, I had actually never considered the way in which the trust of certificates could be validated until version 2.60 of sigcheck was released and the introduced the -v switch for use with -t or -tu:

```
-t[u][v]   Dump contents of specified certificate store ('*' for
all stores). Specify -tu to query the user store (machine store is
the default). Append '-v' to have Sigcheck download the trusted
Microsoft root certificate list and only output valid certificates
not rooted to a certificate on that list. If the site is not
accessible, authrootstl.cab or authroot.stl in the current
directory are used instead, if present.
```

Here is some example output:

```
sigcheck64.exe -tuv -nobanner

User\Root:
  Microsoft Root Certificate Authority 2010
     Cert Status:    Valid
     Valid Usage:    All
     Cert Issuer:    Microsoft Root Certificate Authority 2010
     Serial Number:  52 76 17 36 EE A4 45 81 42 45 3E 2D 73 FA 89
B2
     Thumbprint:     1F3D38F280635F275BE92B87CF83E40E40458400
     Algorithm:      sha256RSA
     Valid from:     1:55 PM 12/1/2017
     Valid to:       9:06 PM 11/30/2042
```

So why should this entry not be trusted? What is Microsoft's basis for trust? The answer to that is authroot.stl — a signed, ASN.1 encoded file consisting of the root certificates that Microsoft has deemed to be trustworthy. This is equivalent to the set of root CAs that come installed by default in the operating system. Occasionally, Microsoft may update this list though (whether through addition or revocation) and distribute updates via this link.

Wanting to understand the STL file format better and not necessarily wanting to rely upon sigcheck for performing root CA trust validation, I wrote a crude parser that extracts all of the trusted certificate thumbprint values so that I could perform similar validation in a PowerShell script. In the screenshot below, you will see the "malicious" cloned root CA certificate highlighted:

It is also possible to parse authroot.stl with certutil.exe:

```
certutil -dump authroot.stl
```

Through parsing authroot.stl, you can also easily determine which Microsoft-specific roots are trustworthy for code signing:

```
PS> ls Cert:\LocalMachine\Root\ | Where-Object {
($TrustedRootHashes -contains $_.Thumbprint) -and
($_.Subject.StartsWith('CN=Microsoft Root')) }

Thumbprint : CDD4EEAE6000AC7F40C3802C171E30148030C072
Subject    : CN=Microsoft Root Certificate Authority,
DC=microsoft, DC=com

Thumbprint : A43489159A520F0D93D032CCAF37E7FE20A8B419
Subject    : CN=Microsoft Root Authority, OU=Microsoft
Corporation, OU=Copyright (c) 1997 Microsoft Corp.

Thumbprint : 8F43288AD272F3103B6FB1428485EA3014C0BCFE
Subject    : CN=Microsoft Root Certificate Authority 2011,
O=Microsoft Corporation, L=Redmond, S=Washington, C=US

Thumbprint : 3B1EFD3A66EA28B16697394703A72CA340A05BD5
Subject    : CN=Microsoft Root Certificate Authority 2010,
O=Microsoft Corporation, L=Redmond, S=Washington, C=US
```

So the way in which Microsoft-signed code should ideally be validated (versus simply pulling the publisher name and validating that it chains to a

"trusted" root) is to perform the following:

1. Validate that the integrity of the binary has not been compromised.
2. Validate that each certificate in the chain is valid.
3. Validate that the root certificate has one of the trusted thumbprints present in authroot.stl (listed above). An alternative to validating against authroot.stl is to call the CertVerifyCertificateChainPolicy function passing it the CERT_CHAIN_POLICY_MICROSOFT_ROOT value. Deep inside this function is basically the same array of certificate thumbprints that the root certificate will be validated against.

One notable omission from authroot.stl is the Microsoft flight root certificate (thumbprint: F8DB7E1C16F1FFD4AAAD4AAD8DFF0F2445184AEB) — the issuer of certificates for Windows Insider Preview builds. Worth noting is the absence of a timestamp countersignature meaning that the signature will fail to validate beyond the certificate validity period. This was likely an intentional decision on the part of Microsoft. The lack of a MSFT timestamp would potentially make the Microsoft flight certificate chain a more viable candidate for cloning assuming a defender isn't aware of what would be considered a "trusted" root certificate thumbprint. Here's an example of kernel32.dll signed by a certificate issued by the Microsoft flight root:

. . .

Conclusion

Hopefully, by now you have a better appreciation of how attackers can appear to originate from the code signer of their choosing. This isn't the only signing attack that would permit this, however. I've also published related

research on how to hijack Subject Interface Packages that effectively allows you to apply legitimate digital signatures to malicious code that passes integrity validation checks.

The purpose of all of this research is twofold: to help encourage defenders and security vendors to challenge assumptions made in their investigative processes but to also educate on the importance of proper code signing validation for the purposes of determining whether any given signed code *actually* originates from who it claims to originate from.

Lastly, an astute reader will have noted that there may have been additional anomalies associated with the cloned certificate chain and signed code. I'll leave discussion of these anomalies for another blog post. See you in 2018!

Security Code Signing

-- 2

Written by Matt Graeber

635 Followers · Writer for Posts By SpecterOps Team Members

Threat Researcher

Follow