Microsoft Windows

Code Signing Best Practices

July 25, 2007

Abstract

This paper provides information about code signing for the Windows® family of operating systems. It provides guidelines for:

- Chief technology officers (CTOs) or chief security officers (CSOs) who deploy the code-signing infrastructure.
- Test lab managers, IT professionals, and developers who manage the process of test signing and verifying applications.
- Build engineers who sign and verify applications for use with Windows.

This information applies for the following operating systems:

Windows Vista®
Windows Server® 2003
Microsoft Windows XP

Future versions of this preview information will be provided in the Windows Driver Kit.

The current version of this paper is maintained on the Web at:
http://www.microsoft.com/whdc/winlogo/drvsign/best_practices.mspx

References and resources discussed here are listed at the end of this paper.

Contents

Disclaimer

This is a preliminary document and may be changed substantially prior to final commercial release of the software described herein.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Introduction

Protecting personal and corporate data remains a top concern for consumer and enterprise users of Windows® around the world. Microsoft is committed to implementing new ways to help restrict the spread of malicious software. Digital signatures for software are an important way to ensure robustness and integrity on computer systems. Digital signatures allow the administrator or end user who is installing Windows-based software to know which publisher has provided the software package and whether the package has been tampered with since it was signed.

Earlier versions of Windows used digital signatures to discourage users from installing anonymous ActiveX® controls, download packages, executable files, and drivers from untrusted sources. In Windows Vista®, new features are taking advantage of code-signing technologies.

What's New in Windows Vista

There are several new uses of code-signing technology in Windows Vista.

- User Account Control (UAC) verifies signatures and prompts users before allowing executables to run with or exercise administrator privileges.
- To ensure access to certain types of next-generation premium content, all kernel-mode components must be signed and components in the Windows Vista Protected Media Path (PMP) must be signed for PMP.
- Installation packages and executables that are downloaded with Microsoft Internet Explorer have a default setting of "don't run" when a Microsoft Authenticode® signature is missing or cannot be verified.
- Boot-start drivers must contain an embedded signature.
- All kernel-mode components on x64 versions of Windows Vista must be digitally signed or they will not load.

Note: A boot-start driver is one that is loaded by the Windows Vista operating system loader. Boot-start drivers can be identified as follows:

- The driver's INF specifies the start type as "Start=0".
- A kernel service that is configured with a ServiceType of "Kernel Driver" or "File System Driver" and a StartMode of "boot".

This paper is an overview of code-signing and related technologies for Windows Vista and provides a set of best practice guidelines for operating a code-signing infrastructure.

Who Should Read this Paper

This paper is intended for:

- Chief technology officers (CTOs), chief security officers (CSOs), public key infrastructure (PKI) administrators, and code-signing operations personnel who intend to deploy a code-signing infrastructure.
- Information technology (IT) professionals, network managers, test lab managers, developers, or build engineers who are responsible for signing and verifying applications for use with Windows.

Table 1 gives the intended audience for each section.

Table 1. Intended Audience

Section name	Intended reader
Code-Signing Basics	All readers
Signing Technologies in Windows	All readers
Digital Signatures in Windows	All readers
Code Signing during Software Development	CTOs, CSOs, and code-signing operations personnel
Code-Signing Service Best Practices	CTOs, CSOs, and code-signing operations personnel
Code-Signing Service Example Topologies	CTOs, CSOs, and code-signing operations personnel
Code Signing for Managed Networks	CTOs, CSOs, and code-signing operations personnel
Resources	All readers
Appendices	All readers

Code-Signing Basics

Code signing employs PKI technologies such as keys, certificates, and digital signatures to ensure the identity and integrity of software. This section is an introduction to the basics of code signing. It does not provide a detailed description of cryptography or digital signatures. For more information on that topic, see "Resources" at the end of this paper.

Uses of Code Signing

Code-signing techniques use digital signatures to provide identity and integrity for software applications. A valid digital signature:

- Identifies the software's publisher.
- Confirms the integrity of the software by verifying that the software has not been modified since it was signed.

Digitally signed software, distributed over the Internet, is thus no longer anonymous. By verifying the identity of the software publisher, a signature assures users that they know who provided the software that they are installing or running. Digital signatures also assure users that the software they received is in exactly the same condition as when the publisher signed it.

Note: Code signing does not necessarily guarantee the quality or functionality of software. Digitally signed software can still contain flaws or security vulnerabilities. However, because software vendors' reputations are based on the quality of their code, there is an incentive to fix these issues.

Digital Signatures

A digital signature binds the publisher's identity to the data that they have published and provides a mechanism to detect tampering. Digital signature algorithms use cryptographic hash functions and asymmetric—or public/private key pair—encryption algorithms. Some digital signatures also take advantage of digital certificates and PKI to bind public keys to the identities of software publishers.

To create a digital signature, one must first acquire a key pair. This consists of a public key and a private key, with the following characteristics:

- The private key is known only to its owner; the public key can be distributed to anyone.
- The private key is used to sign the data; the public key is used to verify the signature on the data.
- The private key cannot practically be calculated from the public key.

In practice, using public-key algorithms to directly sign files is inefficient. Instead, typical code-signing procedures first create a cryptographic hash of the data in a file—also known as a digest—and then sign the hash value with a private key. The signed hash value is then used in the digital signature. The digital signature can be packaged with the data or transmitted separately. A separately transmitted digital signature is known as a *detached signature*.

A cryptographic hash function is a one-way operation that cannot be easily reversed. If the hash function is sufficiently strong, currently known techniques cannot alter an arbitrary set of data without changing the associated hash value. To verify a file's integrity, a user calculates the hash value of the current contents of the file with the same algorithm that was used to create the digest in the signature. If the resulting hash value does not match the digitally signed digest, then the signature, the contents of the file, or some combination thereof have been altered. This renders the signature invalid and the file should not be installed or run.

Figure 1 shows a simplified procedure for signing a file:

1. A hashing algorithm creates a hash value from the data in the file.
2. The hash value is signed with the publisher's private key.
3. The data and the signed hash are published.

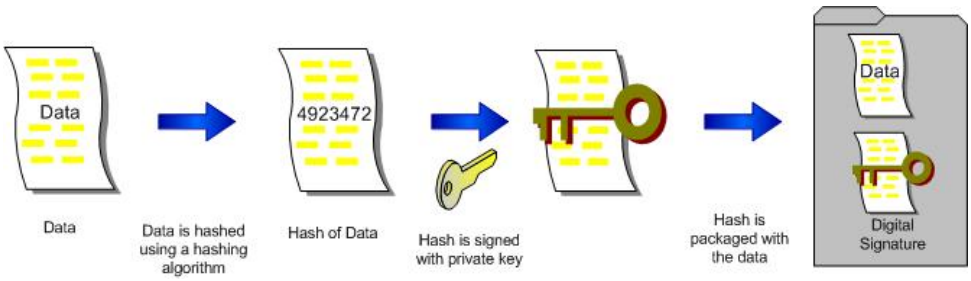

Figure 1. Creating a digitally signed file

Figure 2 shows a simplified process for verifying a signed file:

1. The originator’s public key is used to obtain the original hash value from the signed hash value that is packaged with the data.
2. A new hash value for the data is calculated by using the same algorithm that was used to create the original signed hash value.
3. The hash value that was calculated in step 2 is compared with the hash value from step 1.
4. If the two hash values are identical, the data has not been altered and the digital signature is valid.

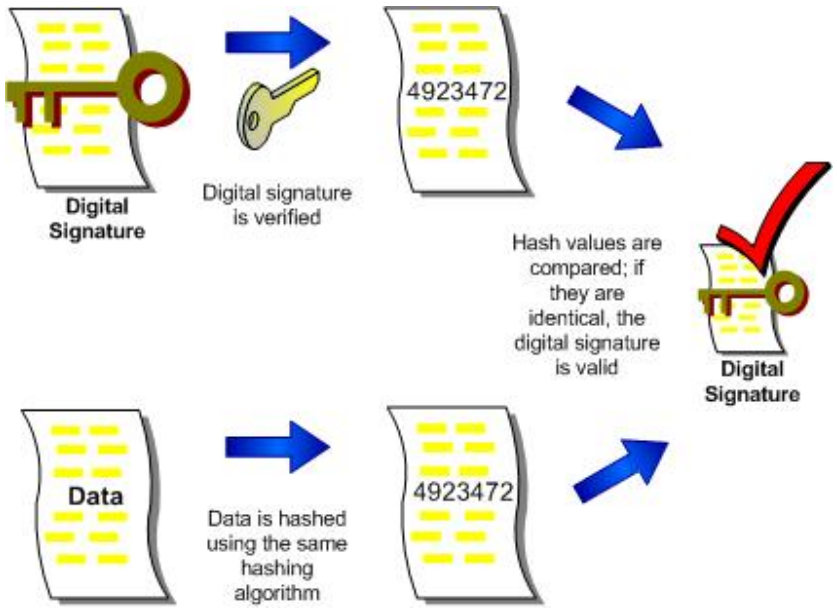

Figure 2. Verifying a digital signature

Digital Certificates

Digital certificates bind an entity, such as an individual, organization, or system, to a specific public and private key pair. Digital certificates can be thought of as electronic credentials that verify the identity of an individual, system, or organization.

Various types of digital certificates are used for a variety of purposes. Examples include:

- Secure Multipurpose Internet Mail Extensions (S/MIME) for signing e-mail messages.
- Secure Sockets Layer (SSL) and Internet Protocol security (IPSec) for authenticating network connections.
- Smart cards for logging on to personal computers.

Windows code-signing technologies use X.509 code-signing certificates, a standard that is owned by the Internet Engineering Task Force (IETF). For more information, see "Resources" at the end of this paper.

Code-signing certificates allow software publishers or distributors to digitally sign software. Certificates are often contained in digital signatures to verify the origin of the signature. The certificate owner's public key is in the certificate and is used to verify the digital signature. This practice avoids having to set up a central facility for distributing the certificates. The certificate owner's private key is kept separately and is known only to the certificate owner.

Software publishers must obtain a certificate from a certification authority (CA), which vouches for the integrity of the certificate. Typically, a CA requires the software publisher to provide unique identifying information. The CA uses this information to authenticate the identity of the requester before issuing the certificate. Software publishers must also agree to abide by the policies that are set by the CA. If they fail to do so, the CA can revoke the certificate.

Identity and Policy

Digital signatures that include digital certificates allow users to make trust decisions by cryptographically verifying the identity of software publisher whose certificate was used to sign the code. Code-signing policies enable users, administrators, and the Windows platform to make trust decisions in a consistent way.

For a list of examples of uses and policies that are supported by Windows Vista, see "Digital Signatures in Windows" later in this paper.

Roles within the Code-Signing Ecosystem

The code-signing ecosystem consists of several roles. In smaller organizations or in test environments, the same individual might fulfill several roles. In larger organizations or in production code-signing environments, each role is typically fulfilled by a separate individual or team. The common high-level roles within a code-signing ecosystem are:

- Certification authority (CA)
- Software publisher
- Software distributor
- Software consumer

Figure 3 shows how the different roles fit together.

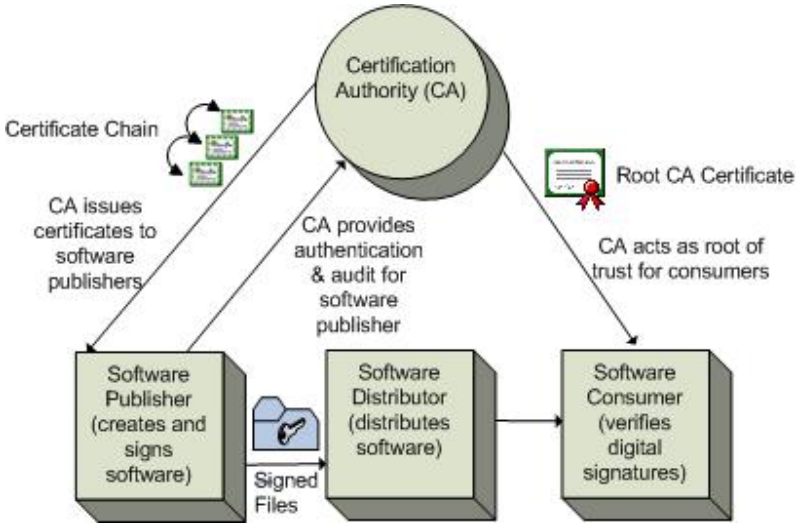

Figure 3. Code-signing ecosystem

Certification Authorities

CAs act as trust brokers. They issue certificates to software publishers, binding the publisher's public code-signing key to an identity that software consumers can verify. The policies and operating procedures of CAs vary greatly. However, in general, a CA:

- Verifies the identity of the software publisher according to the CA's issuer statement.
- Issues certificates to software publishers.

- Revokes certificates if a software publisher's private key is compromised or the publisher violates the CA's usage practices. CAs revoke certificates by issuing certificate revocation lists (CRLs) or an online certificate status protocol (OCSP).
- Protects the CA's private keys, which are used to sign certificates that are issued to software publishers and to sign revocation information.
- Publishes a root certificate or *root of trust* that consumers can use to verify the identity of the entities to which the CA has issued certificates.

For commercial and government applications, CAs typically deploy a certificate chain, in which each certificate is digitally signed by the certificate above it. A certificate chain normally traces up to a CA's root certificate, creating a *hierarchy of trust*.

Several commercial CAs issue code-signing certificates whose root certificates ship with the Windows platform and are updated through the Microsoft Root Certificate program.

As an alternative, software publishers and enterprises can use the Microsoft Windows Certificate Server—a feature of some versions of Windows Server®—to issue certificates for use within a managed network or test environment.

- For managed network scenarios, an internal CA is useful for signing internal line-of-business (LOB) applications, scripts, or macros.
- For test environment scenarios, an internal CA is useful for testing prerelease software without requiring code to be publicly signed. Additionally, test code-signing certificates can be issued to individual members of development teams to sign private prerelease versions of software with unique certificates.

The manager of the network or test environment uses tools that are built into Windows or distributed with the Windows Software Developers Kit (SDK) to configure the clients on the managed network or test lab to trust the root certificate of the internal CA. In these cases, the code-signing certificates are trusted only within the lab or network environment.

Note: Earlier than Windows Vista, the Windows SDK was known as the Platform SDK (PSDK).

Software Publisher

A software publisher creates and digitally signs software with a code-signing certificate. The software publisher registers with a CA, requests a code-signing certificate, and agrees to act in accordance with the CA's policies. Software publisher is a broad term and includes a large range of software development activities. Some examples include:

- Independent software vendors (ISVs) that produce Windows applications.
- Independent hardware vendors (IHVs) that produce Windows drivers.
- Web developers who create ActiveX controls for internal or public applications.
- IT administrators who sign third-party drivers and applications for use in a managed network.

A software publisher must protect its private code-signing key and set up internal code-signing policies and procedures to ensure that only approved code is signed. For more information on this process, see "Code-Signing Service Best Practices" later in this paper.

If a software publisher signs a malicious program, it must revoke the compromised code-signing certificate. To do so, the software publisher informs the CA that issued the certificate. The CA revokes the compromised certificate by issuing a CRL or updating the certificate's OCSP status. The software publisher must then request a new code-signing certificate. Software consumers can optionally check the revocation status of a code-signing certificate when verifying digital signatures; revoked certificates do not verify.

Software Distributor

A software distributor publishes software to users. Digital signatures normally ensure the integrity of the software regardless of the distribution method. The distribution mechanism varies. However, if a file is properly signed, it does not matter from a security point of view whether the file is distributed via a Web site, ftp site, file share, optical media (CD or DVD), or e-mail.

In the absence of an Authenticode signature, the security of the distribution mechanism that is used for strong name–signed .NET assembly is a factor when deciding whether to trust a strong name–signed package. For example, an authenticated file server on a private network provides users a higher degree of certainty about the origin of the strong name–signed software. For further information on strong name signatures, see "Strong Name Signatures" later in this paper.

Software Consumer

A software consumer is any user or application that verifies the digital signatures on software. Without the identity and integrity information in the digital signature, the software consumer cannot make an informed decision about whether to install or run the software.

Authenticode policy verifies only signatures that are signed with a code-signing certificate that was issued by a CA whose root certificate is in the Trusted Root Certification Authorities certificate store. By default, the Trusted Root Certification Authorities certificate store contains a set of commercial CAs that have met the requirements of the Microsoft Root Certificate Program. Administrators can configure the default set of trusted CAs and install their own private CA for verifying internal software. Note that a private CA is not likely to be trusted outside the administrator's network environment.

A valid digital signature assures the software consumer that an application is strongly identified and has not been tampered with. However, a valid signature does not necessarily mean that the end user or administrator should implicitly trust the software publisher. Users or administrators must decide whether to install or run an application on a case-by-case basis, based on their knowledge of the software publisher and the application.

Test Signing versus Release Signing

Microsoft recommends that commercial software publishers establish a separate test code-signing infrastructure to test-sign prerelease builds of their software. Test certificates must chain to a completely different root certificate than the root certificate that is used to sign publicly released products. This precaution helps ensure that test certificates are trusted only within the intended test environment.

Using test certificates during development has several advantages:

- Code signing during testing provides more authentic test coverage by exercising signing-related code paths. It also checks for any footprint or timing-related issues.

- Test signing is much more efficient than release signing for daily use during development. In particular, members of the development team can have much greater access to test certificates and private keys than they do to release certificates and private keys.
- Using test certificates during testing ensures that if prerelease code is accidentally released to the public, it is not signed with the organization's production certificate.
- Test certificates can be deployed broadly to an organization's developers or contractors to identify the creator of software. This provides accountability in the development process and a measure of software integrity.

This paper recommends reserving production certificates that are issued by commercial CAs for signing only public beta and final releases of software and internal line-of-business software. If the best practices for managing keys—discussed later in this paper—are followed, the private keys that are used for release signing will be behind a security boundary. The private keys will be accessible to only a few trusted individuals and will require an approval process before the private keys can be used to sign code.

Test certificates, on the other hand, are valid only in a test environment, so access can be granted more broadly. Using test certificates to sign code does not require the same sort of controlled process as production signing. Test signing can be supported on a small scale with tools that are provided with the Windows Software Development Kit (WinSDK) or Windows Driver Kit (WDK). For more information on test-signing see "Code Signing during Software Development" later in this paper.

Note: If code is signed automatically as part of a build process, it is highly recommended that any code that is submitted to that build process be strongly authenticated.

Signing Technologies in Windows

Windows supports a number of code-signing technologies. Authenticode and strong name signing are the two technologies that are most relevant to a wide range of scenarios in Windows Vista. This section covers these two technologies and their related tools.

Authenticode

Authenticode is a Microsoft code-signing technology that identifies the publisher of Authenticode-signed software and verifies that the software has not been tampered with since it was signed. Authenticode uses cryptographic techniques to verify identity. It combines digital signatures with an infrastructure of trusted entities, including CAs, to assure users that an application originates from the stated publisher. Authenticode allows users to verify the identity of the software publisher by chaining the certificate in the digital signature up to a trusted root certificate. Authenticode code signing is different from strong name signing in that it uses a hierarchical system of CAs to verify the identity of the software publisher.

Authenticode code signing does not alter the executable portions of an application:

- With embedded signatures, the signing process embeds a digital signature within a nonexecution portion of the file.
- With signed catalog (.cat) files, the signing process requires generating a file hash value from the contents of a target file. This hash value is then included in a catalog file. The catalog file is then signed with an embedded signature. Catalog files are a type of detached signature.

Authenticode allows software vendors to embedded-sign a number of Windows file formats, including:

- Portable executable (PE).
  An executable file format that includes .exe, .dll, .sys, and .ocx files.
- Cabinet (.cab).
  A file that stores multiple compressed files in a file library. The cabinet format is an efficient way to package files because compression is performed across file boundaries, significantly improving the compression ratio.
- Windows Installer (.msi and .msp).
  A file format for software packages or patches that are installed by the Windows Installer.

Embedded Signatures

Embedded signing protects an individual file by inserting a signature into a nonexecution portion of the file. The advantage of using an embedded signature is that the required tool set is simpler and the signature is in the file itself.

With embedded signatures, Windows features such as the **File Download – Security Warning** dialog box, the UAC prompt, and the Windows loader can easily find the signature. However, not all operating system features check for embedded signatures. For example, Device Management and Installation (DMI) recognizes only signed catalog files. For more information, see "User Account Control" and "Third-Party Signing for Plug and Play Driver Installation" later in this paper.

By default, embedded signatures are supported for only the limited set of file formats that were listed in the previous section. All other formats are supported through signed catalog files, which are discussed in the following section. However, the Authenticode infrastructure has a provider model to support embedded signing and verification for other file formats. An Authenticode provider is called a subject interface package (SIP) and must be installed separately by the application that requires the new file format.

Signed Catalog Files

A catalog file contains the hash values for a set of files. A signed catalog file acts as a signature for all the files whose hash values are in the catalog. Unlike embedded signatures, a signed catalog file is a type of detached signature. Signed catalog files allow developers to store signed hashes separately from the hashed files' content. For example, a driver package could include a catalog file that contains hash values for the driver's .sys and .inf file. Digitally signing a catalog file protects the hash values, allowing customers to verify the integrity of the individual files.

The advantage of using signed catalog files is that they can support all types of file formats, especially file formats that cannot accommodate an embedded signature. In addition, catalog files are more efficient if a software publisher must digitally sign many files because the entire set of files requires only a single signing operation.

A file can be both embedded and catalog signed. For example, the Windows loader requires boot-start drivers to have an embedded signature. However, the Plug and Play installation process requires a signed catalog file to verify the driver. Authenticode can simultaneously accommodate both types of signatures. To see how Plug and Play installation displays digital signature information, see "Third-Party Signing for Plug and Play Driver Installation" later in this paper.

The Windows Catalog Database

A signed catalog file must be added to the Windows Catalog Database for Windows features such as UAC and the x64 Windows kernel, to discover it. There are several ways to handle this procedure:

- For Plug and Play drivers, the catalog file is automatically added to the database as part of the installation process.
- Non–Plug and Play drivers, third-party applications, or installation programs use **CryptCATAdminAddCatalog** to add catalog files to the database. For more information, see the reference documentation in the MSDN® library.

Timestamps

Certificates normally expire after a period of time, such as one year. However, software is typically designed to operate for many years. If the certificate that was used to sign the code expires, the signature cannot be validated and the software might not install or run. To avoid this issue, Microsoft recommends that software publishers timestamp their digital signatures.

A timestamp is an assertion from a trusted source, called a timestamp authority (TSA), that the digital signature's signed hash was in existence when the timestamp was issued. If the signing certificate was valid at that time, Windows considers the signature to be valid even if the certificate has since expired. If a signature is not timestamped, when the certificate used to sign the software expires, the signature simply becomes invalid.

Timestamps also play a role when checking for revoked certificates. If a digital signature was timestamped before the certificate was revoked, the signature remains valid. Timestamping thus allows a company to revoke a certificate and start signing with a new certificate without any risk of invalidating previously signed software applications.

Currently, a few CAs offer Authenticode timestamp services. The timestamps are valid for all Windows platforms that are configured with the issuing CAs root certificate. However, network or test lab managers might not want to use third-party timestamps for internal applications or prerelease software. Windows does not include a timestamping service, but nCipher offers a code-signing timestamp server that can be deployed as an in-house service. For more information, see the white paper titled "Deploying Authenticode with Cryptographic Hardware for Secure Software Publishing."

Strong Name Signatures

Strong name signatures differ from other types of code signing in that the public/private key pair is not associated with a certificate. In addition, strong name signing is used exclusively with .NET assemblies.

Strong names are used to provide unique identities to .NET assemblies. A strong name consists of:

- An RSA public key
- A simple name, usually the name of the file that contains the assembly without the extension
- A four-part version number
- An optional culture
- An optional processor architecture

The build process creates a strong name assembly by signing it with the private key that corresponds to the public key in the strong name. The strong name signature can then be verified by using the RSA key that is part of the strong name.

An assembly must use the strong name to reference another strong name assembly. When the .NET Framework loads a strong name assembly, it normally verifies the strong name signature. If verification fails, the .NET Framework does not load the assembly.

This rule is not applied to assemblies from the global assembly cache (GAC), which is a computer-wide cache that stores assemblies that are intended to be shared by several applications on the computer. Assemblies in the GAC are verified when they are installed. Because the GAC is a locked-down store that can be modified only by an administrator, GAC assemblies are not verified each time they are loaded.

Strong Name Best Practices

Strong name signatures are more specific in function and in the security assurance they provide than Authenticode signatures are. These differences require another set of best practices, in addition to general code-signing guidelines.

Strong name signatures prevent attackers from tampering with an assembly in an undetected fashion. However, the security assurance of a strong name signature depends on two factors:

- The signer’s private key remaining private.
- The signer’s public key being securely distributed to developers whose assemblies reference the signed assembly.

Strong name signatures are particularly vulnerable to compromised keys because strong name keys cannot expire or be revoked. In addition, strong name signatures are self issued. There is no verification or registration procedure, which means that a strong name provides no inherent way to tie securely the identity of the assembly's publisher to the private key that is used to sign the assembly. For this reason, strong name signatures are often coupled with Authenticode signatures because Authenticode provides support for certificate expiration and revocation, plus a PKI hierarchy to bind the Authenticode signature's public key to the software publisher.

End users who install strong name–signed assemblies should have additional ways to verify the assembly or its publisher. Network managers should take extra precautions when choosing to trust public keys that originate from outside their organization because they do not necessarily know what level of security an external organization has used to secure its private keys or signing process. To mitigate the risks, strong name signatures should be used in conjunction with other signing or distribution mechanisms. For example:

- Some organizations trust strong name–signed assemblies if they are also Authenticode signed from a trusted software publisher. The Authenticode signature should be applied only after the assembly is strong name signed.
- Some enterprises trust managed assemblies that are strong name signed with trusted public/private key pairs and are distributed from a trusted source such as an authenticated file server.

After an assembly is strong name signed, all referenced assemblies must also be strong name signed. This ensures a consistent level of integrity protection within a strong named–signed assembly, provided that strong key management practices are maintained for each key.

Code-Signing Tools

Authenticode and strong name tools are available from several sources:

- Code-signing tools for Windows Vista are distributed with the Windows SDK, which is available to members of the Windows Vista Beta Program.
- The Platform SDK for Windows Server contains the required information and tools to develop 32-bit and 64-bit Windows-based applications for earlier versions of Windows. This SDK is available as a free download.
- The WDK contains the information and tools for developing Windows drivers. It includes the Hardware Compatibility Test (HCT) kits and the tools that Microsoft uses to build and test the stability and reliability of the Windows operating system. The WDK is available to members of the Windows Vista Beta Program.
- The .NET Framework SDK contains the required information and tools to develop managed code applications. Like the Platform SDK, it is available as a free download.

For further information on these tools, see "Resources" at the end of this paper.

Note: The tools in the Platform SDK and the WDK are not distributable. For more information, see the tools' end-user license agreements (EULAs).

The following sections provide brief descriptions of the tools that are used for code signing. For further information, see "Resources" at the end of the paper. For a detailed walkthrough of the kernel-mode code-signing process, see the white paper titled “Kernel-Mode Code Signing Walkthrough.”

MakeCat

The MakeCat utility is a code-signing tool that creates an unsigned catalog file that contains the hashes of a specified set of files and their associated attributes. A catalog file allows an organization to sign a single file—the catalog—instead of signing numerous individual files.

After the catalog file is signed, the software consumer can hash the target files, compare the hashes of the existing files to the original hashes within the catalog file, and verify the signature on the catalog file. This process verifies that the hashed files are free of tampering.

Before using MakeCat, the user must use a text editor to create a catalog definition file (.cdf), which contains the list of files to be cataloged and their attributes. The MakeCat tool:

- Scans the .cdf file and verifies the attributes for each listed file.
- Adds the listed attributes to the catalog file.
- Hashes each of the listed files and stores the hash values in the catalog file.

MakeCat does not modify the .cdf file.

For more information on MakeCat, see "Using MakeCat," in the MSDN library.

Certification Creation Tool (MakeCert)

MakeCert is a certificate-creation tool that generates certificates for test signing. MakeCert creates a code-signing certificate that is either self signed or signed by the Microsoft Root Agent key. The certificate can be placed in a file, a system certificate store, or both. If the test environment requires authentication, only self-signed certificates should be used.

For more information on MakeCert, see Appendix 1 of this paper and "Certification Creation Tool (Makecert.exe)" in the MSDN library.

Sign Tool (SignTool)

SignTool is a command-line tool that signs, verifies, and timestamps all file formats that Authenticode supports including PE, .cat, .cab, and any format for which a SIP has been installed. SignTool also can verify whether a signing certificate was issued by a trusted CA, whether a signing certificate has been revoked and, optionally, whether a signing certificate is valid for a specific policy.

Other uses of SignTool include:

- Verifying the files with catalog signatures.
- Verifying signatures against different Authenticode policies.
- Displaying a signature's certificate chain.
- Displaying the SHA1 hash value of a file.
- Displaying errors for files that did not verify.
- Adding and removing catalog files from the catalog database.

For more information on SignTool, see Appendix 5 of this paper or "SignTool (Signtool.exe)" in the MSDN library.

Certificate Manager Tool (CertMgr)

CertMgr manages certificates and CRLs. The tool has three functions:

- Displaying certificates and CRLs.
- Adding certificates and CRLs from one certificate store to another.
- Deleting certificates and CRLs from a certificate store.

For more information on CertMgr see Appendix 2.3 of this paper or "Certificate Manager Tool (Certmgr.exe)" in the MSDN library.

PVK2PFX

A .pvk file is a deprecated file format for storing keys. The use of .pvk files is discouraged because .pvk is a less secure file format than the Personal Exchange Format (.pfx), but is still used by some CAs. PVK2PFX is a Windows SDK tool that moves certificates and private keys that are stored in .spc and .pvk files to .pfx files. When possible, the preferred approach is to then store private keys in hardware.

For more information on PVK2PFX see "Appendix 5. Signing with PVK and PFX Files" in this paper.

Inf2Cat

Inf2Cat is a Winqual submission tool that replaces the functionality provided by Signability. For driver vendors, Inf2Cat verifies driver packages and uses the information in a driver's INF file to create an unsigned catalog file.

Note: Inf2Cat is not currently part of the WDK tools; it is installed with the Winqual Submission Tools. When the Winqual Submission Tools package is installed, Inf2Cat is placed in the Program Files (x86)\Microsoft Winqual Submission Tool folder.

Signability

Signability is a WDK tool for Plug and Play drivers that verifies the contents of a driver package and creates an unsigned catalog file. For driver vendors, this tool is easier to use than MakeCat because Signability does not require a separate .cdf file. It instead gets the information it needs from the driver's INF file. Signability is also available from the Microsoft Winqual Web site.

Strong Name Tool (Sn.exe)

The .NET Framework SDK's Strong Name Tool is used to strong name sign assemblies. It supports options for key management, signature generation, and signature verification and supports strong name test signing, delay signing, and release signing. For more information, see "Strong Name Tool (Sn.exe)" in the MSDN library.

Digital Signatures in Windows

Microsoft recommends that vendors who publish Windows software provide a better overall user experience by digitally signing their code. Previous versions of Windows have used digital signatures to provide a more secure download and installation experience for Windows components, drivers, installation packages, and executables.

In Windows Vista, Authenticode signatures are a requirement for the Windows Vista Logo for Software program. Windows Vista enhances existing uses of digital signature verification with stricter policies and stronger messages to discourage users from running or installing unsigned code. In addition, Window Vista also includes new features that verify signatures on applications that require administrator access, drivers that load within the windows kernel, and drivers that access PMP.

Note: Some Windows Vista features accept only embedded signatures, catalog signatures, or a specific combination of both. Different Windows Vista features place different constraints on the issuer of the code-signing certificate. Software publishers are advised to research the requirements of the features that they are targeting.

Existing Uses of Digital Signatures on Windows

The following are some examples of the use of digital signatures by Windows:

- **Windows Setup**. The Windows installer uses signed Authenticode catalog files to verify all Windows components and software updates. Windows Setup installs only files that are digitally signed by Microsoft.
- **Windows Update**. Software update packages from Windows Update are protected by signed catalog files. The installation fails if the Windows Update client detects that any of the files have been modified since they were signed.

  Windows Server Update Services (WSUS) allows enterprises to control which updates are downloaded to users' systems. WSUS also uses digital signatures to verify the integrity of the package.
- **Windows Installer (.msi)**. The Windows installer verifies signatures on .msi packages. If a package has an invalid signature, the installer warns users before it installs the package.
- **.NET assemblies.** A digitally signed .NET assembly has a strong name, which uniquely identifies the assembly and helps prevent spoofing. Users should not make security decisions based solely on a strong name. For more information on strong name signing, see "Strong Name Signatures" earlier in this paper.

- **Software restriction policies (SRPs)**. These policies allow administrators to identify what software is running in their environment and control the ability of that software to execute, thereby protecting their environments from malicious code. For more information, see "Resources" at the end of this paper. With software restriction policies, an administrator can:
  - Control which programs run on a computer.
  - Permit users to run only specific files on multiple-user computers.
  - Specify who can use Group Policy to add trusted publishers to the computer.
  - Control whether software restriction policies affect all users on a computer or just certain users.
  - Prevent files that are not explicitly signed by the administrator from running on the local computer, the organizational unit, the site, or the domain.

Enhanced Use of Digital Signatures in Internet Explorer Windows on Windows Vista

This section contains two examples of how Microsoft Internet Explorer on Windows Vista has enhanced the use of digital signatures.

The Internet Explorer Download Experience

In Windows XP and earlier versions of Windows, Internet Explorer checks the signatures on downloaded executables and installation packages and warns users if the content is unsigned or if the digital signature is invalid before installing or running the software. With Windows XP SP2, Internet Explorer always checks the signatures on downloaded software before the program is run, even if the program is launched after Internet Explorer has closed.

With Windows Vista, Internet Explorer has implemented even stricter signature checking for downloaded files. After a file is downloaded, the signature is checked and one of the following dialog boxes appears to the user. If the digital signature is valid and signed with a certificate that was issued from a trusted CA, Internet Explorer displays a **Security Warning** dialog box that shows the publisher and cautions the user about downloading files, as shown in Figure 4.

Figure 4. Internet Explorer security warning: digital signature

If the digital signature does not verify or if the software is not signed, Internet Explorer presents a second **Security Warning** dialog box that warns the user that the signature was not valid and the publisher is unknown, as shown in Figure 5.

Figure 5. Internet Explorer security warning: digital signature is invalid or not present

ActiveX Controls

ActiveX controls are software packages that Internet Explorer can automatically download and run. Microsoft recommends that all ActiveX controls be digitally signed so that Internet Explorer can identify the control's publisher. If a user attempts to download a signed ActiveX control that has not yet been installed, Internet Explorer displays the information bar shown in Figure 6 to ask the user if he or she wants to continue. Using the information bar instead of a dialog box for this purpose reduces the likelihood of a user inadvertently installing a control.

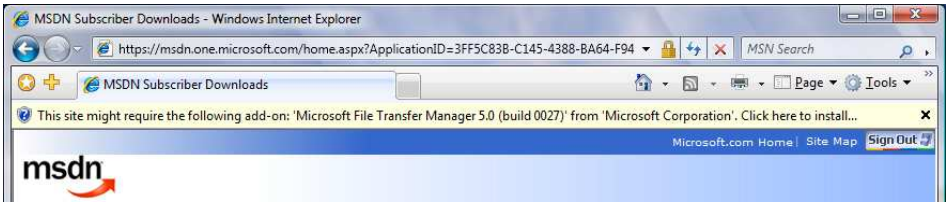

Figure 6. Internet Explorer Information Bar

Depending on the security zones settings, Internet Explorer might not download an unsigned ActiveX control at all. Figure 7 shows the security level settings for a typical system's Internet zone. Note that Internet Explorer does not download unsigned ActiveX controls from this zone.

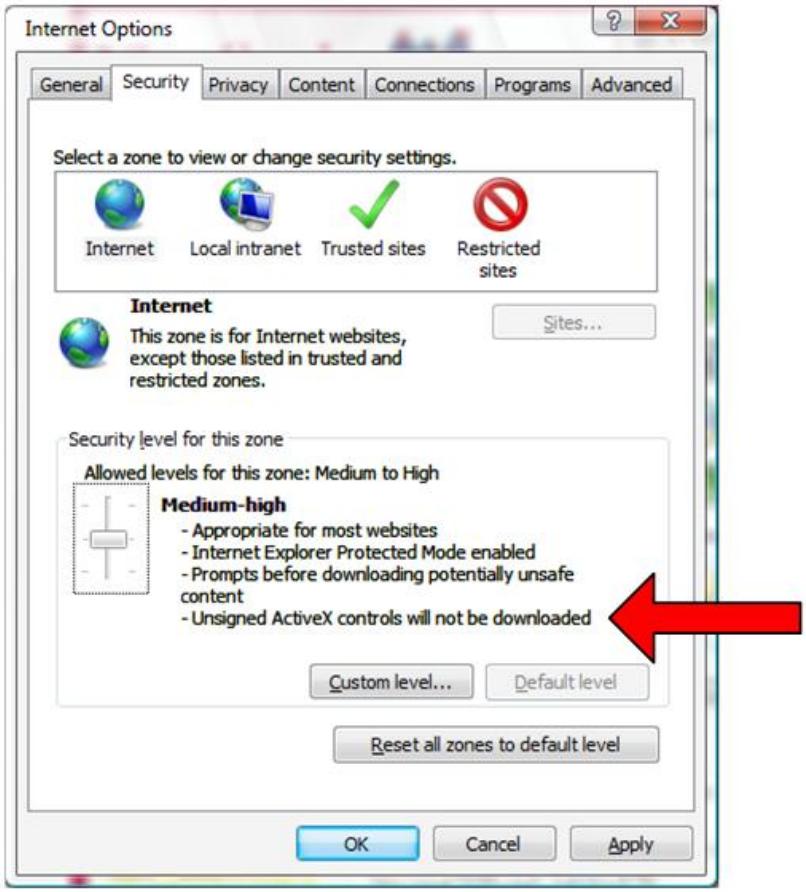

Figure 7. Typical system: security levels

New Uses of Digital Signatures in Windows Vista

Windows Vista uses code signing more widely than earlier versions of Windows. This section contains several examples of new uses of code signing by Windows Vista.

User Account Control

The Windows Vista UAC feature enhances the traditional user-privilege model that prevents users from accidentally running unsafe programs. UAC enables users to run at low privilege most of the time, but allows them to elevate the application's privileges when necessary. In addition, Windows Vista has a security policy option that can be set to allow only signed executables and Windows components to run with elevated privilege. A user receives a different prompt, depending on whether the executable is a Windows component, signed third-party software, or unsigned software.

Figure 8 shows an example of a **User Account Control** dialog box that asks the user whether it should elevate an application's privileges.

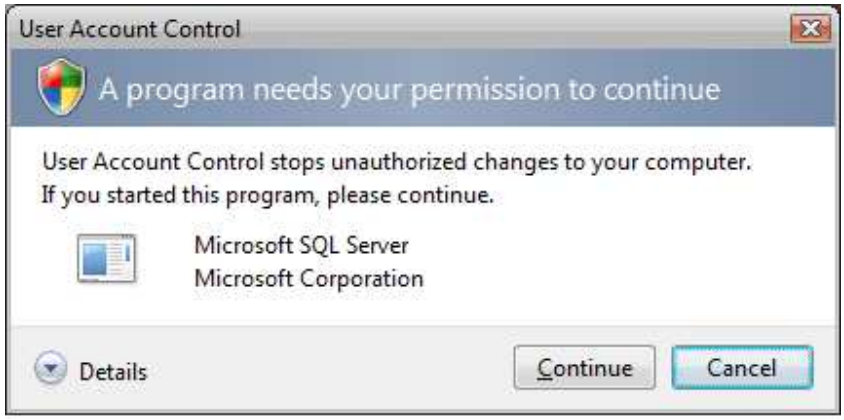

Figure 8. UAC: An application requesting privilege elevation

Figure 9 shows an example of the **User Account Control** dialog box asking for user consent before running an unidentified application.

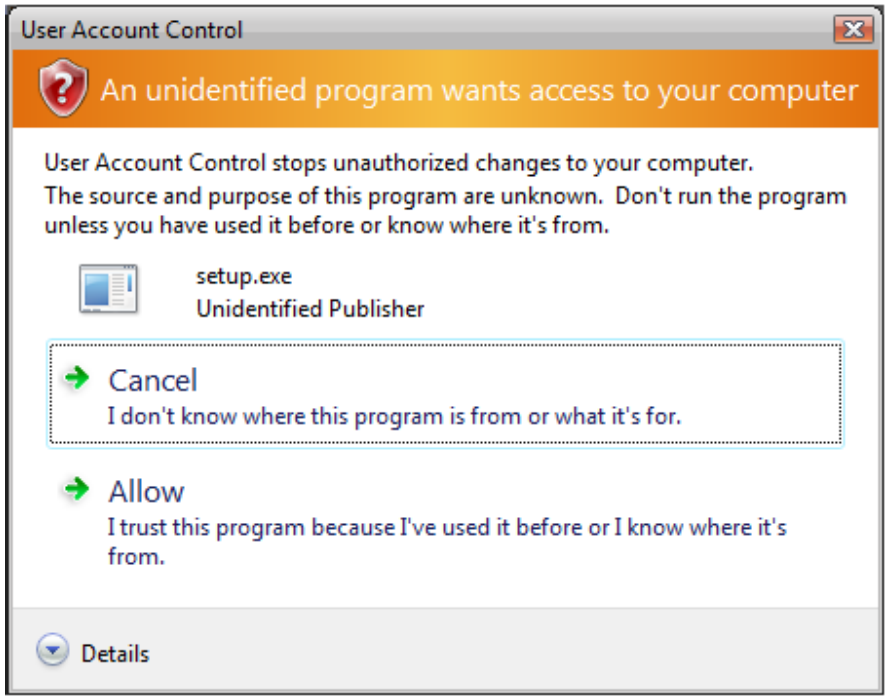

Figure 9. UAC: application without valid signature

Kernel-Mode Driver Signature Enforcement

Windows Vista confirms that the kernel is in an *identifiable* state by verifying the digital signatures for all kernel-mode driver image files as they are loaded into memory.

- The x64 versions of Windows Vista and Windows Server do not allow unsigned kernel-mode drivers to load, except on test computers. The operating system loader also enforces digital signature verification for drivers that are loaded into memory at boot time.

- All kernel modules must be code signed for a PC system to be able to play back next-generation premium content such as HD DVD and other formats that are licensed under the Advanced Access Content System (AACS) specification.

For further information on these issues, see the white papers titled “Code Signing for Protected Media Components in Windows Vista” and “Digital Signatures for Kernel Modules on x64-based Systems Running Windows Vista.” Signing kernel-mode drivers for use on x64 versions of Windows Vista requires the use of cross-certificates. For more information, see the white paper titled “Microsoft Cross-certificates for Windows Vista Kernel Mode Code Signing.”

Third-Party Signing for Plug and Play Driver Installation

The Plug and Play driver signing behavior in Windows XP makes it difficult for device vendors and corporate IT departments to deploy drivers in some situations. Windows Vista addresses these problems by allowing vendors and IT departments to sign and publish drivers themselves.

IT departments can configure Windows to treat drivers that they have signed as equivalent to drivers that a Windows publisher has signed. They can then silently deploy drivers across their corporation, including updates of in-box drivers. This feature also allows driver vendors to deploy emergency fixes quickly to their customers without having to wait for a Microsoft Windows Hardware Quality Lab (WHQL) Logo Program signature from Microsoft.

Third-party signatures complement rather than replace the signatures that the WHQL provides.

- Third-party signatures guarantee the publisher's identity and driver integrity, but do not provide any information about the quality of the driver.
- WHQL signatures are provided as part of the Windows Logo Program. They not only verify identity and integrity but also indicate that the driver has met or exceeded the quality standard that the WHQL tests define.

Windows Vista contains several other changes and new features that are related to allowing third-party signatures:

- Administrators can control which driver publishers Windows Vista trusts. Windows Vista installs drivers from trusted publishers without prompting. It never installs drivers from publishers that the administrator has chosen not to trust.
- Driver-signing policy is always set to Warn, eliminating the Block and Ignore options that were available in earlier versions of Windows.
- All device setup classes are treated equally. Certclas.inf no longer exists.
- When there are several compatible drivers to choose from, the ranking algorithm that Windows Vista uses to pick the best driver includes drivers with third-party signatures. By default, Microsoft signatures take priority over third-party signatures, but IT departments can configure them to be equivalent.

Windows Vista verifies the digital signatures of driver packages during installation. Plug and Play installation accepts only signed catalog files because .inf files do not support embedded signatures. Depending on the results of signature verification, Plug and Play might display a dialog box. The behavior of the dialog box depends on the driver package's digital signature, as detailed in the following list. Note that standard users can install drivers from the first two categories on the list but only administrative users receive the prompts that allow them to install drivers from the remaining categories. For nonadministrative users, Windows Vista silently refuses to install drivers from those categories.

- **Signed by a Windows publisher.** Plug and Play does not display a dialog box for driver packages that are signed and distributed as part of Windows Vista or that are signed by the Windows Logo Program.
- **Signed by a trusted publisher.** No dialog box appears if a driver package is signed by a trusted publisher. A publisher is trusted when its certificate has been added to the Trusted Publishers certificate store.
- **Signed with a publisher of unknown trust.** Plug and Play displays the dialog box shown in Figure 10 if a driver package is signed with a certificate that is not in the Trusted Publishers certificate store or the Untrusted Publishers certificate store. An administrator can choose to install the driver package. An administrator can also select the **Always trust software from *PublisherName*** option. This adds the publisher's certificate to the Trusted Publishers certificate store.

Figure 10. Publisher of unknown trust dialog box

- **Unsigned or altered:** Plug and Play displays the dialog box shown in Figure 11 for two scenarios:
  - The driver package does not contain a signed catalog file. A driver package is also considered unsigned if the certificate was not issued by a trusted CA or the signature is not valid.
  - One or more of the files in the driver package have been altered since the package was signed.

Figure 11. Unsigned driver dialog box

For more information on driver package integrity for third-party developers, see the white paper titled “Driver Package Integrity during Plug and Play Device Installs in Windows Vista.” There is also related information in the WDK under the sections titled "How Setup Selects Drivers" and "Signing Drivers for Development and Test." For additional information on driver signing for x64 versions of Windows Vista, see the white paper titled “Digital Signatures for Kernel Modules on x64-based Systems Running Windows Vista.”

Protected Media Path (PMP)

Windows Vista enables the playback of next-generation premium content such as HD DVD and other formats that are licensed under the AACS specification. To ensure access to this new content, Windows Vista systems must support the requirements that are defined by the AACS specification and by content owners. Windows Vista fulfills one of these requirements through code signing, a process that provides benefits to consumers such as increased security against malware and better driver reliability to improve system stability.

System and device manufacturers must follow new code-signing requirements for Windows Vista systems that will support the playback of premium content. These requirements include the following:

- To ensure access to premium content, components that run within the Windows Vista PMP must be signed for PMP.
- Display device drivers must include an embedded certificate that verifies a robust pipeline throughout the video processing engine.

Tools and guidelines for code signing are provided in the WDK and Windows SDK. Submission guidelines for driver signing under the WHQL testing program are available on the Microsoft Web site. For more information on PMP requirements, see the white paper titled “Code Signing for Protected Media Components in Windows Vista.”

Windows Defender

Windows Defender is a feature of Windows Vista that protects Windows computers against spyware and malicious code. One of its components, Software Explorer, provides detailed information about the software that is running on a system, such as the program's name, publisher, and whether the software is digitally signed. Microsoft recommends when users should obtain more information about the software publisher before running the software. A user can then make a more informed decision about whether to trust a specific piece of software. For more information on Windows Defender, see “Windows Defender.”

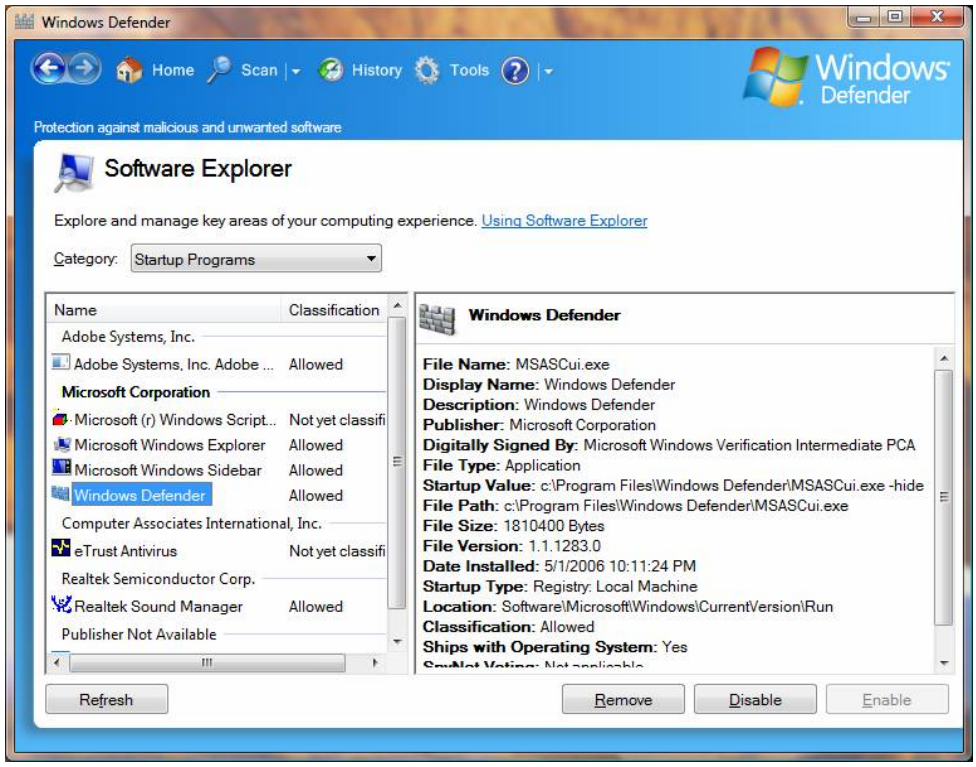

Figure 12. Windows Defender: Software Explorer

Code Signing during Software Development

Previous sections of this paper discussed the basics of code signing and how end users and Windows features use digital signatures to authenticate and ensure the integrity of publicly released applications. Application vendors should verify that the application installs and functions properly with a signature, before the application is publicly released. However, the private keys that are used to test-sign prerelease builds of an application should be different from the keys that are used to release-sign an application.

This section discusses why test signing is important and describes how developers and testers can *test-sign* prerelease builds of an application. This process ensures that the software behaves as expected on a test computer or in a test environment before signing the application for public release. This section includes discussions of several scenarios for test signing during the development cycle, including:

- Test signing by individual developers
- Test signing as part of a build process

- Configuring a test computer or environment
- Test-signing operations

The topics in this section are accompanied by appendixes that contain the required detailed procedural information to implement each phase of the test-signing process.

What Test Signing Is

Test signing is the act of digitally signing an application with a private key and corresponding code-signing certificate that is trusted only within the confines of a test environment. Test signing ensures that an application functions correctly during installation and run time before it is signed with the publisher's release code-signing certificate for public distribution. The procedures are essentially similar to those that are used to release-sign an application. However, there a number of reasons for having a separate test-signing process and reserving release signing only for publicly released builds such as public betas and final releases.

- It is much easier for organizations to recover from a compromised test certificate than from a compromised release certificate because test certificates are trusted only within the confines of a test environment.
- Using test certificates during testing ensures that accidentally leaked prerelease applications are not signed with the organization’s release certificate.
- Using the release certificate’s private key exclusively for release signing minimizes the chances of accidentally signing code that should not be signed.
- Development teams can have much greater access to test certificates than release certificates. Easier access to test-signing keys means greater efficiency for the development team.
- For most software development environments, virus scanning may not be required for prerelease builds before test signing because the code is not trusted outside the test environment.
- Test certificates—other than WHQL test certificates—are normally created in-house. In the event of a problem, such as a virus or compromised test certificate, organizations can simply reset the test environment and issue a new test certificate. This is significantly easier than issuing a security bulletin to customers and publicly revoking a release certificate.

A test-signing environment can be implemented very cost effectively by using free utilities or features that are built into the Windows. Tasks such as issuing test certificates, configuring test computers, or even test signing builds can be automated, depending on an organization’s level of investment.

Test Signing by Individual Developers

During the earlier phases of the development cycle, it is useful for developers to test-sign their binaries by using their own test-signing certificates. There are two primary ways to generate such test-signing certificates:

- Developers can generate their own *self-signed* certificates.
- Certificates can be issued from a test-signing CA.

For either option, test-signing certificates should be clearly identified as appropriate only for testing purposes. For example, the word "test" could be included in the certificate subject name and additional legal disclaimers could also be included in the certificate

Self-Signed Test Certificates

Developers can use the MakeCert tool to generate self-signed test certificates for signing their code. In this context, a self-signed certificate is a certificate that is signed by the certificate's own private key, which is not issued by a CA. For a detailed discussion of the procedure, see Appendix 1 of this paper.

The advantage of self-signed certificates is three-fold:

- Self-signed certificates require no overhead or infrastructure.
- The necessary tool is available free as part of the WinSDK or WDK.
- The developer can operate independently of any infrastructure components in the development environment.

The main drawback of using self-signed certifications is that the developer must spend some time acquiring and learning to use MakeCert. Additionally, developers must configure their test computers to recognize their self-signed certificates as “trusted” before the test computers will be useful for testing test-signed applications. This can be done manually or with scripting. The most convenient options for configuring test computers are the Microsoft Management Console (MMC) Certificates snap-in—discussed in Appendix 2.2 of this paper—and the CertMgr command line tool—discussed in Appendix 2.3 of this paper. For more information on using self-signed certificates, see "Configuring a Test Computer/Environment" later in this paper.

In theory, it is possible to use Group Policy to configure a self-signed certificate on multiple computers. However:

- Using Group Policy to configure multiple certificates can create manageability issues.
- Group Policy is the only centralized way to revoke self-signed certificates. This could be problematic if one or more computers is disconnected from the network for an extended period.

Test Certificates Issued by a Certification Authority

Another option for issuing test certificates to developers is to have a domain administrator or network manager issue test certificates through a centrally managed CA, such as a Microsoft Certificate Server. This paper highly recommends that test certificates chain up to a separate root certificate. For more details on this option, see Appendix 4 of this paper.

Deploying Microsoft Certificate Server is most effective in a domain environment with an Active Directory. This environment is not required to deploy a Microsoft Certificate Server, but a properly configured domain, CA, and Active Directory largely eliminate certificate management tasks for developers who only need to sign prerelease applications with individual test certificates.

CA administrators have several models that they can use to issue test certificates:

- The CA administrator automatically issues certificates to all members of a specified domain, typically developers and testers.
- CA clients make certificate enrollment requests. There are two ways to handle the approval process:
  - The CA administrator designates a specific individual to manually approve each request.
  - The CA automatically evaluates each request based on access control lists (ACLs), previously configured by the CA administrator.
- The CA administrator designates specific individuals to make certificate enrollment requests on behalf of the members of the organization.

In addition to centralizing the process, a CA makes other aspects of certificate management simpler including expiration, revocation, and reissuance.

Test computers can be configured through Active Directory or with Group Policy. If these techniques are not desirable, other options include manual configuration with the MMC snap-in or command line scripts that use CertMgr. For more information, see "Configuring a Test Computer/Environment" later in this paper.

Integrating Test Signing into the Build Environment

Integrating test signing into an automated build process can save the development team considerable time and effort. Source control and build processes are largely beyond the scope of this paper. For the purposes of this paper:

- A build process is a centralized, automated process that compiles both release and prerelease software from source files that are stored in a source control system.
- The source control system ensures that only approved source files are compiled, tracks submissions from individual developers, and allows code changes to be rolled backward or forward.
- Developers and testers typically develop against or test with the most recent build. The productivity of an entire software team may depend on getting the latest build as quickly as possible.

Given these characteristics, it makes sense for many development teams to integrate test signing into their build process:

- Build integration saves time because the signing is part of an automatic, centralized process. Developers do not have to concern themselves with the mechanics of code signing.
- Build integration can be very helpful for signing software packages that have complex signing requirements. For example, an x64 kernel-mode boot start driver might require both an embedded signature for load-time verification and a catalog signature for verification during Plug and Play installation. Furthermore, the driver package might be packaged in a self-extracting executable file, which must also be signed.

Here are some things to remember when integrating test signing into a build procedure:

- Test signing a build can be performed with a self-signed certificate or a CA-issued certificate.
- The test-signing certificate must be configured to be trusted on each computer on which binaries are tested. This can make a CA-issued certificate a more

practical choice for larger environments. For information on configuring test computers, see "Configuring a Test Computer/Environment" later in this paper.

- Test signing must occur after the binaries are in their final form. Procedures such as rebasing or localizing a signed file invalidate the signature.
- In a well-architected build process, the only difference between test and release signing is whether the code is signed locally with a test certificate or submitted to a code-signing service for release signing.
- If many files must be submitted to a code-signing service, consider using signed catalog files. It requires less network bandwidth to transfer a single catalog to the code-signing service, and the signing operation requires less time for a single file.
- Automatically release signing the output of a build process requires a carefully considered security model. "Code-Signing Service Best Practices" later in this paper, discusses additional security requirements for automatically release signing submissions from a build server.

Configuring a Test Computer or Environment

Test computers must be configured to verify test-signed applications by adding the test root certificate and test certificate to the appropriate system certificate stores. A system certificate store is a logical data store that can be configured with an MMC snap-in, CertMgr, Group Policy, or certificate auto-enrollment, which is described in more detail in Appendixes 2 through 4 of this paper. For more information on certificate stores, see “Certificate Revocation and Status Checking.”

Important: When adding a certificate to a test computer's certificate stores, do not export the private key from the computer that handles the signing process. Exporting a private key compromises its security, and the private key is not necessary for signature verification.

Having test and test root certificates in the appropriate certificates stores affects Windows behavior and policy decisions for the following features:

- Plug and Play device installation
- User Account Control
- Internet Explorer download experience and ActiveX control installation
- Windows Defender Software Explorer

Note: x64 kernel-mode software and PMP software require additional configuration. For information on configuring x64 systems for test signing, see the white paper titled “Digital Signatures for Kernel Modules on x64-based Systems Running Windows Vista. For more information on PMP test signing, see the white paper titled “Code Signing for Protected Media Components in Windows Vista.”

The remainder of this section discusses the different security stores that may have to be configured, depending on the testing scenario.

Trusted Root Certification Authorities

The Trusted Root Certification Authorities certificate store contains the root certificates of all CAs that Windows trusts. For a Windows test computer to recognize a test signature as valid, the Trusted Root Certification Authorities certificate store must contain the test root certificate of the CA that issued the code-signing certificate in the signature.

Many commonly used commercial CAs' root certificates are in the Trusted Root Certification Authorities certificate store by default. However, if an application is signed with a self-signed certificate, that certificate must be added to the Trusted Root Certification Authorities certificate store. Otherwise, Windows will not successfully verify the test signature.

Trusted Publisher

For default Authenticode policy—used by Plug and Play, Internet Explorer, and others —if a publisher's code-signing certificate is in the Trusted Publishers certificate store, the user or administrator has agreed to allow Windows to install or run the publisher's software without displaying security warnings or dialog boxes. The Trusted Publishers certificate store is different from the Trusted Root Certification Authorities certificate store in that only *end-entity* certificates can be trusted. In other words, adding a test CA root certificate to the Trusted Publishers certificate store does not configure all certificates that this CA issued as trusted. Each certificate must be added separately.

Configuring certificates in the Trusted Publishers certificate store is not necessary—nor in most cases desirable—for manual testing of test-signed applications. For scenario testing, it is generally desirable for a tester to replicate the end-user experience of installing the application, with the Trusted Publishers certificate store in its default state. In that state the software vendor's code-signing certificate is not configured in the Trusted Publishers certificate store. However, scripting an automated test to respond to a trust dialog box may not be possible. In these cases, it is appropriate to add the test-signing certificate to the Trusted Publishers certificate store.

Machine and User Certificate Stores

Two types of system certificate stores are relevant to configuring a test computer: user certificate stores and machine certificate stores. There is one set of machine certificate stores per computer and one set of user certificate stores per user account. Note that all user certificate stores inherit the contents of the machine-level certificate stores. If a certificate is added to the Trusted Root Certification Authorities machine certificate store, all Trusted Root Certification Authorities user stores will also contain the certificate.

The required level of privilege to configure a certificate store depends on the type of store. Users with administrator privilege can configure the machine store and their own user stores. Users with lower privileges can configure only their own user stores.

Note: Different Windows features make decisions based on different certificate stores. In general, processes that are running under LocalSystem, LocalService, or NetworkService contexts trust only certificates in the machine certificate stores. Applications that run in a user's context trust that user's certificate stores.

For example, test-signing verification by a Plug and Play installation requires that test certificates be located in a machine certificate store. Test-signing applications for the UAC, Internet Explorer, and Windows Defender scenarios only require test certificates to be present in the user's certificate store.

Test owners can differentiate between configuring the user and machine stores in all but the first of the configuration tools that are described in Appendix 2 of this paper.

Test Computers versus Test Environments

The best way to configure a test computer depends on its broader environment:

- A single test computer is most easily configured by double-clicking the certificate's .cer file, launching the MMC Certificates snap-in, or running CertMgr from the command line. For details, see Appendixes 2.1, 2.2, and 2.3, respectively, of this paper.
- A larger test network that is not joined to a domain can be configured by creating client side scripts that call CertMgr.
- A larger test network that is joined to a domain can use either of the preceding mechanisms. However, this paper recommends configuring such computers with either Group Policy, or Microsoft Certificate Server with Active Directory. For further information, see "Certificates from an Internal CA" later in this paper or Appendix 3 of this paper.

Test-Signing Operations

Test signing can be performed by individual developers or by a centralized build process. The procedure is relatively simple. To set up a test-signing operation, a developer must:

- Acquire a test code-signing certificate with a private key.
- Configure the code-signing certificate.
    - If the code-signing certificate is issued by a CA, the CA must be configured in either the user or machine Trusted Root Certification Authorities certificate store. This ensures that the tool that it used for signing will recognize that the code-signing certificate is valid.
    - If the test certificate is self-signed, then the self-signed certificate itself must be located in the Trusted Root Certification Authorities certificate store.
- Acquire a copy of SignTool. This tool is included with the WinSDK and WDK and runs from their command line environments. If developers want to use SignTool outside those command line environments, they must make sure that CAPICOM.dll—version 2.1.0.1 or higher—has been copied to the directory where SignTool is located. CAPICOM is also redistributed through MSDN. For more information, see "Resources" at the end of this paper.
- Decide whether to timestamp the application. Timestamping is discussed in the next section.

Timestamping

A timestamp is an assertion from a trusted source—called a timestamp authority (TSA)—that the digital signature's signed hash was in existence when the timestamp was issued. If the signing certificate was valid when the signature was timestamped, Windows considers the signature to be valid, even after the certificate has expired. If a signature is not timestamped, Windows considers the signature invalid after the certificate has expired.

There are three options for timestamping prerelease software, as discussed in the following sections.

Option 1: No timestamp

This choice is appropriate if the expiration date of the code-signing certificate significantly exceeds the expected lifetime of the test code. For example, a self-signed certificate that expires in one year should be more than sufficient to sign a

prerelease build of an application that is expected to be used for only a few days. If the test certificate is part of an automated process, be sure to refresh the certificate periodically, well before the certificate expires. This ensures that prerelease applications are not accidentally signed with a certificate that is close to expiring. The build process designer should also ensure that the release-signing process still timestamps the signature.

Option 2: Timestamp against a public timestamp authority
This is the recommended approach for all public and long-lasting releases of applications. Timestamping is currently available through Verisign, free of charge. Other commercial CAs may also offer timestamping services. Accessing the timestamping service requires Internet connectivity. For more information, see "Appendix 4. SignTool" of this paper and "SignTool (Signtool.exe)" in the MSDN library.

Option 3: In-house timestamps
Software development organizations that must perform code signing and timestamping without sending traffic outside a private network should consider deploying an in-house timestamping service. Currently nCipher offers a hardware-based Authenticode timestamping solution. However, if a timestamp does not chain to a root certificate by default in the Trusted Root Certifications Authorities certificate store, the timestamp is trusted only within the managed network environment where the computers are explicitly configured to trust the TSA. For further information, see the white paper titled “Deploying Authenticode with Cryptographic Hardware for Secure Software Publishing.”

If Option 2 or 3 is chosen, the code signer must decide whether to sign and timestamp in the same operation or to perform the tasks on separate computers. The rationale behind using two computers is that timestamping requires an Internet-facing computer. Signing requires access to a private signing key, and an organization might not want to risk having the computer used for release signing connected to the Internet. Using a separate computer for timestamping helps maintain the security of the private key. For test-signing purposes, however, a single operation is probably more appropriate because the value of the test-signing private key is limited and can easily be reissued.

Using SignTool
After the signing environment has been configured, it is easy to use SignTool to sign, timestamp, or sign and timestamp. For examples of command line syntax and additional guidance on the operation of SignTool, see Appendix 5 of this paper.

Code-Signing Service Best Practices
The cryptographic keys that are the core of the code-signing process must be well protected and treated with the same care as a company’s most valuable assets. These keys represent a company’s identity. Any application that is signed with these keys appears to Windows as bearing a valid digital signature that can be traced to the company. If a key is stolen, the thief could, for example, use it to fraudulently sign malicious code and deliver an application that contains a Trojan or virus that appears to originate from a legitimate publisher.

It is therefore prudent for software publishers to implement policies defining:

- How private keys are to be protected.
- Who approves code-signing operations.
- How to ensure that only high-quality, virus-free code is signed and released to the intended customer.

These policies should be in place before acquiring a code-signing certificate.

This section describes the best practices for implementing test and production code-signing services. These recommendations are general principles to guide the decision-making of organizations that want to implement a code-signing infrastructure. Following the best practices in this document will defend against compromising the private key, signing prerelease software with a release signing key, or allowing malicious software to be signed.

Conforming to the best practices that are described below does not necessarily require a large expenditure or time commitment. Organizations might find it useful to perform a formal security analysis to understand what measures are appropriate to protect their identity and reputation. The controls put in place to secure a code-signing infrastructure can be tailored to the circumstances of the code signer and the value of the private signing key. There is no one-size-fits-all solution; each organization must determine the right balance between security, cost, and impact on development processes.

Cryptographic Key Protection

Microsoft recommends that software publishers store cryptographic keys for code signing in a secure tamper-proof hardware device. Keys that are stored in software are more susceptible to compromise than keys that are stored in hardware. For example, if a software key is leaked, the key or file that contains the key is typically copied, making it difficult to detect the breach. Keys that are stored on hardware are less vulnerable to undetected compromise.

Three types of hardware devices are typically used to protect code-signing keys:

- Smart cards.
- Smart card-type devices such as Universal Serial Bus (USB) tokens. The best practices that are discussed in this paper for smart cards also apply to smart card–type devices.
- Hardware security modules (HSMs).

HSMs are dedicated hardware devices that store and protect cryptographic keys. They are typically used by medium to large organizations with substantial code-signing requirements. Although HSMs offer more features than smart cards, smart cards and smart card–type devices might be sufficient for smaller organizations with fewer code-signing requirements.

Organizations should consider the criteria in Table 2 when evaluating smart cards against HSMs.

Table 2. Key Protection Evaluation Criteria

Criteria	Smart card	Hardware security module
Certification: FIPS 140-2 Level 3	Generally, no	Yes
Key generation in hardware	Maybe	Yes
Key backup	No	Yes
Multifactor authentication	No	Maybe
Speed	Slower	Faster
Separation of roles	No	Yes
Automation	No	Yes
Destruction	Yes	Yes

The following list explains the terms and issues that are summarized in Table 2.

- **Certification.** Federal Information Processing Standard (FIPS) 140-2 is a U.S. Government standard that provides criteria for evaluating cryptographic modules. As a best practice, Microsoft recommends that vendors use FIPS 140-2 Level 3 certified products. Industrial HSMs normally have this certification, but smart card devices usually do not.
- **Key generation**. When keys are generated, they should never be exposed as plain text outside the hardware device. Industrial HSMs provide key generation in the hardware. Smart card implementations might support this feature.
- **Key backup**. Keys should be backed up for disaster-recovery purposes and to mitigate the risk of a hardware failure. It is important that keys not be exposed in plain text outside the HSM during backup operations. Industrial HSMs support secure key backup from one FIPS 140-2 Level 3 certified device to another. If the keys are not backed up and the hardware device fails, new keys must be issued.
- **Multifactor authentication.** Organizations should require more than one approver for a code-signing operation. This is referred to as "k of n," where a specific number of authorizations must be present to perform any cryptographic operations. For example, three trusted individuals out of seven must be present to digitally sign software.

  Some HSMs support multifactor authentication. A multiple-approver requirement can be met with a smart card, but it must be accomplished in other ways. For example, a smart card could be stored in a safe that requires multiple keys or one officer could know the safe's combination while another knows the smart card's personal identification number (PIN).
- **Speed.** Smart cards generally perform cryptographic operations more slowly than industrial HSMs. HSMs contain dedicated cryptographic accelerators, which makes them better suited to high-throughput environments. Any organization that requires more than 100 signings per week should consider using an HSM.
- **Separation of duties.** Administrative and operational roles should be segregated to avoid relying upon a single individual or team.
- **Automation**. HSMs support automated signing. With smart cards, someone must manually enter a PIN for each signing operation. High-throughput or time-sensitive environments that require automated signing should deploy an HSM.
- **Destruction**. HSMs support the complete key life cycle including key destruction. If a private key becomes invalid for any reason, it is important that the key be destroyed. This can be accomplished by overwriting the key on the HSM or smartcard or by physically destroying the device.

Signing Environment

The signing environment must be physically secure. Otherwise, there is no way to prevent smart cards or HSMs from being lost or used to sign inappropriate content. The value and availability requirements of the code-signing infrastructure dictate the physical and logical barriers that must be implemented. For example, for a low-throughput infrastructure it may be sufficient to keep an HSM in a locked room that is accessible to only a limited set of individuals. In contrast, a high-throughput production infrastructure may require a high-security data center.

This paper does not attempt to provide a comprehensive description of how to physically secure a code-signing environment. In general, Microsoft recommends that software publishers apply the principles of defense-in-depth by using multiple security techniques for different layers of their code-signing system. Organizations should consider the following physical security best practices:

- Access controls such as locks, keycard cards, or biometrics to restrict access to the code-signing infrastructure
- Restricted physical access to HSMs or smart cards
- Security cameras
- Onsite security guards
- Visitor logging
- A tools-resistant server safe for online code-signing servers
- Logical and physical segregation of the code-signing infrastructure from the production network
- Periodic audits of the code-signing infrastructure to ensure compliance with documented practices and design

For non-networked signing services, the signing computer or smart card should be kept offline in a tool-resistant safe when it is not in use. The following procedures are highly recommended for networked code-signing systems:

- Keep the signing service behind a firewall or on a separate network. Consider Microsoft Internet Security and Acceleration Server or Server and Domain Isolation with IPSec.
- Deploy network authentication and integrity protocols as part of the submission and approval process. Password authentication is generally not recommended. Instead, consider certificate-based authentication that uses certificates that are issued from an internal CA or domain authentication that uses Kerberos. IPSec with mutual authentication is also a viable option.
- Consider using a separate drop-off-and-pick-up server as an intermediary between the corporate and signing service networks.
- Consider deploying network intrusion-detection systems to detect network-based hacking attempts.

Code-Signing Submission and Approval Process

Microsoft recommends implementing a code-signing submission and approval process to prevent the signing of unapproved or malicious code. At a minimum, the approval process should consist of three roles:

- **Submitter.** The submitter is typically a developer.
- **Approvers.** The approvers should understand software development but also be somewhat independent of the submitter to increase objectivity during the approval process. Requiring multiple approvers reduces the chance of accidentally signing software and mitigates the risk of a single employee signing inappropriate content. Face-to-face meetings to review code for approval are also encouraged.
- **Signers.** The individuals who actually sign should be independent of the development and approval process. For example, an operations team could be responsible for the actual code signing.

The approval and submission process itself should be protected against compromise. Organizations should consider the following measures to ensure that submissions and approvals are genuine.

- If the service is not networked, consider face-to-face submissions between trusted individuals.
- For networked approval, consider:
  - Digitally signed e-mails from trusted individuals.
  - Signoff through authenticated Web forms.
- For networked code submission, consider:
  - Authenticated network access for code submission.
  - Digital signatures for code submission.

Auditing Practices

Microsoft recommends that all code-signing activities be logged for auditing purposes. In a small organization, auditing could be handled with an electronic log or physical log book that is stored in a secure location. A medium to large organization might record every transaction in a code-signing database. The auditing process should record data such as the name of the file, the name of the submitter, the name of the approver, the hash of the file, and the file before and after signing. This data ensures that code-signing activity can be examined and individuals can be held accountable if unintended or malicious code is signed.

If an audit server is used, it is important to ensure that the audit server is hardened. Microsoft Baseline Security Analyzer (MBSA) is a free tool that helps organizations detect common security misconfigurations, such as unnecessary active services, and identify missing security updates. Run MBSA regularly to ensure a sound security posture.

Virus Scanning

It is critical that signed code not contain viruses. Microsoft strongly recommends that software publishers check all submitted files for viruses as part of the release process. Organizations should use at least two antivirus programs before signing any public release. Virus scanning should be part of the code-signing process and should be used in addition to standard desktop virus scanning by developers. Best-practice recommendations include the following:

- Ensure that the virus-scanning system is hardened and is operating in a known good state. Harden the virus scan server and run MBSA regularly.
- Ensure that the virus-scanning software is operating in a known good state. This is accomplished by updating the virus definitions and testing the virus-scanning application with a known test file before scanning submissions. This ensures that the virus-scanning engine and virus-scanning definitions are up to date and working properly.
- Logically or physically separate the virus-scanning server from the code-signing server. Ensure separation of duties by giving two separate individuals or teams access to and responsibility for maintaining the virus-scanning servers.

Test Signing

Microsoft recommends that organizations use separate test-signing procedures to sign prerelease code, not the release-signing process. The use of test signing has several advantages:

- Test signing limits the possibility that prerelease code could be signed with the organization's production keys.
- Test-signing keys are not valid outside the test environment, so they require less stringent security policies.
- Test-signing keys are less valuable than production-signing keys. so they can be distributed more freely to members of a software development team. For example, a developer can be granted access to an individual test key, reducing the required time for seeking approval.
- Access to the release code-signing infrastructure can be far more controlled than for test signing because the release keys are required less frequently.
- Test-signing keys and certificates can be generated in-house with freely available tools.

For further information on test signing, see "Code Signing during Development" earlier in this paper.

Release Signing

Microsoft strongly recommends that all publicly released code be signed with a certificate from a CA that is present in the trusted CA root store. This recommendation includes public betas as well as final releases. Software for internal use can be signed by a certificate that is issued by either a commercial CA or an internal CA that the organization manages. Major milestones might also be release signed to ensure that the release-signing process is functioning properly. However, such internal submissions should be confined to the organization and protected from public release.

How to Acquire a Certificate from a Commercial CA

When obtaining a certificate from a commercial CA to sign publicly released code, the following key generation best practices should be followed:

- A minimum of two highly trusted individuals should witness the request to the CA for a certificate and the creation of the key.
- The keys should be generated in hardware on an HSM.
- After creation, the keys should be securely backed up and stored in a secure offsite.

For more information, see "Key Backup" under "Cryptographic Key Protection" earlier in this paper.

Revocation

If a private signing key is compromised, the publisher must contact the issuing CA and have the certificate revoked. Revocation invalidates the certificate and adds it to the CRL. If signatures are timestamped, those files that were signed before the certificate was revoked still have valid signatures. Otherwise, revocation invalidates all files that are signed by using the revoked certificate. Note that revoked certificates should remain in the CRL because a timestamped signatures that was created with certificates that were later revoked can remain valid indefinitely.

Note: If legitimately signed applications are signed and timestamped after the malware is signed and timestamped, revoking the compromised certificate invalidates the signatures on the legitimately signed applications in addition to the malware.

The most effective way to limit the risk of a compromised key is to implement good security policies and procedures.

Automation

Automation, in the context of this paper, means a process for automatically approving and signing code that is submitted to a code-signing service without requiring manual approval. This approach can shorten the turnaround time for builds and increase the available time for development. However, automation extends the trust boundary from the code-signing service to the build process itself. Organizations that use automation should implement a layered security approach to ensure that only approved code can be submitted to the build process and subsequently signed.

Some additional recommendations for a robust automation process include the following:

- Authenticate and audit the process for approving code that is submitted to the build process.
- Authenticate submissions from the build computer to the code-signing service.
- Consider using strong network security for both the build and code-signing networks, including:
  - Requiring strong authentication and message integrity.
  - Maintaining separate networks for build, code signing, and general use.
  - Installing intrusion-detection and network-monitoring software.

Separation of Duties

Organizations should use separation of duties to ensure that administrative and operational roles are separated within the code-signing environment and the automated build process. This approach prevents organizational pressures from influencing the overall signing process. For example, one way to implement separation of duties is to have:

- One group that is responsible for the approval process.
- A second group that controls access to the HSM.
- A third group that is responsible for auditing and virus scanning.

Staffing Requirements

Only highly trusted employees should have a code-signing role. Organizations should consider, for example, having such employees undergo background or credit rating checks. Other factors that might influence staffing decisions could include seniority, performance, and job satisfaction.

Timestamping

As previously discussed, certificates normally expire after a period of time, such as one year. However, software is typically designed to operate for many years. If the certificate that was used to sign the code expires, the signature cannot be validated

and the software might not install or run. To avoid this issue, Microsoft recommends that software publishers timestamp their digital signatures.

The digital signature of production code should be timestamped to ensure that the code is still valid after the certificate that was used to sign the code has expired. Timestamping is available through Verisign. Other commercial CAs may also offer timestamping services. Alternatively, an internal hardware-based timestamping service can be deployed. However, the timestamps that an internal TSA provides will be trusted only within the internal network. For more information, see "Timestamps" under "Signing Technologies in Windows" earlier in this paper.

Code-Signing Service Example Topologies

The purpose of this section is to map the code-signing best practices and general requirements that were discussed earlier in this paper to specific deployment scenarios. It includes several examples of possible code-signing topologies to which CTOs and CSOs can refer when designing their organization's code-signing infrastructure.

Because every organization has different requirements, these topologies provide general guidelines and are not meant to be overly prescriptive. Organizations should consider conducting a formal security analysis to understand what measures are appropriate to protect their identity and reputation. Code-signing service architects must then research and define the specific details of the code-signing infrastructure**.**

This section includes three topology diagrams that illustrate some typical scenarios for signing applications:

- Offline manual signing
- Manual submission with automated signing
- Automated submission from a build computer with automated signing

As shown in the diagrams, all code-signing infrastructures contain several common procedures, The details of these procedures were discussed earlier in this paper. Common operations include:

- Approval process
- Submission process
- Virus scanning
- Logging
- Code signing

These procedures are somewhat independent of each other. For example, manual submission could be combined with automated logging and virus scanning. Alternatively, automated submission, virus scanning, and logging processes could be combined with a manual signing process.

Each topology includes a different set of security boundaries that are based on the characteristics of the signing environment and the requirements of the signing service. Security boundaries, in the context of this paper, are physical or logical barriers that control access to sensitive assets and processes. Security boundaries are used to limit access to a small group of highly trusted accountable individuals and trusted infrastructure components. When possible, separation of duties inside security boundaries helps to mitigate reliance on any single individual or team.

Note: Several topics that were discussed in the previous section are not included in these example topologies: certificate acquisition, internal CA deployment, test-signing architecture, key backup, and key destruction. However, code-signing service architects must address these issues when designing their specific code-signing infrastructures.

Offline Manual Signing Topology

Offline manual signing is the simplest topology and is best suited to organizations where release signing is relatively infrequent. Such organizations should consider using a batch approach, which runs the signing process only when a preset limit of submitted jobs is reached.

Because this process is manual, several actors must perform the various roles:

- **Approver.** An individual who is responsible for approving submissions to the build and signing process.
- **Developer.** An individual who is responsible for implementing code and submitting it to the build and signing process.
- **Neutral third party.** A trusted individual or small group of individuals that is somewhat removed from the development process. For example, they could be employees from another part of the organization, such as the legal or finance department. The neutral third-party brings objectivity and separation of duties to the code-signing process.
- **Operations signer.** A trusted individual or set of individuals who are responsible for code-signing operations.
- **Virus scanner.** A trusted individual who is responsible for scanning submissions and maintaining the virus-scanning computer. To enforce separation of duties, a separate individual or team should handle the virus-scanning operation.

An offline manual signing process may have the following characteristics:

- Signings are low frequency.
- The code-signing server is a laptop that is stored in a safe.
- Private signing keys are stored on a smart card.
- The smart card is stored in a safe. The neutral third party, such as a member of the finance or legal department, can access the safe.
- Every signing operation requires that a PIN be entered to unlock the smart card.
- Separation of duties restricts knowledge of the smart card's PIN to trusted operations signers.
- A log book is used to manually record time, date, submitter, approvers, and a hash of the data that was signed.

For more information on separation of duty and virus scanning best practices, see "Code-Signing Service Best Practices" earlier in this paper.

Figure 13 depicts a typical offline manual signing topology and the following sections describes the workflow.

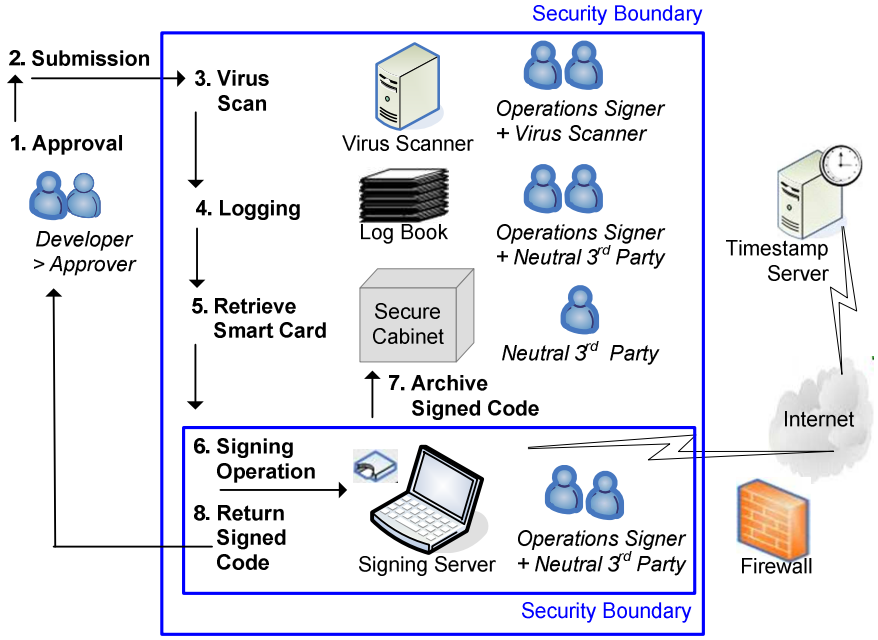

Figure 13. Offline manual signing topology

Step 1: Approval

Offline Manual Signing

Who: *Developer; Approvers*
Reference: "Code-Signing Submission and Approval Process"

The code is approved before it can be submitted to the signing process.

- The developer asks the approvers to approve the code for submission.
- Approval requires at least two individuals in addition to the developer.

Step 2: Submission

Offline Manual Signing

Who: *Operations Signer*
Reference: "Certificates from an Internal CA"

After the code is approved, it is submitted to the signing system.

- All processing—other than signing—is complete. Changing the submission later, by processes such as rebasing files or localizing resource files, invalidates the signature.
- Approved code is transported to the signing system by removable media such as a CD, DVD, or USB storage drive, not over a network connection.
- A face-to-face meeting authenticates the submitter to the operations signer.
- Optionally, the developer digitally signs the code to further authenticate the submission. The submitter's certificate is issued by an internal CA, which is not shown in Figure 13.

Step 3: Virus Scan *Offline Manual Signing*

Who: *Operations Signer; Virus Scanner*

Released applications must be free of viruses.

- At least two commercial-grade antivirus applications are used to check submitted code for malware.
- To enforce separation of duties, the operations signer and the virus scanner are different individuals.
- The virus scanning server is physically and logically separate from the code-signing server.
- To enforce separation of duties, different individuals operate the virus-scanning server and the code-scanning server.
- The virus-scanning computer is maintained according to the best practice recommendations:
  - Run MBSA regularly.
  - Ensure that virus signatures are up to date and are known to be in a good state.

Step 4: Logging *Offline Manual Signing*

Who: *Operations Signer; Neutral Third Party*

The entire code-signing operation is logged for auditing purposes.

- A manual log book is used to log submissions.
- Access to the log book is restricted to the operations signer and the neutral third party.
- The log book is stored in a secure cabinet or safe that is accessible only to the neutral third party.

Step 5: Retrieve Smart Card *Offline Manual Signing*

Who: *Neutral Third Party*

The private keys are retrieved and given to the operations signer.

- The private signing keys are stored on a smart card that is owned and securely stored by a neutral third party.
- Only the neutral third party has access to the smart card. This provides separation of duties from the operations signer.
- To begin the signing operation, the neutral third party retrieves the smart card from storage and meets the operations signer within the security boundary of the signing environment.

Step 6: Signing Operation *Offline Manual Signing*

Who: *Operations Signer; Neutral Third-party*
Reference: “Signing Environment”

After the smart card is present, the signing operation commences.

- The code-signing server is in a logically and physically secure room that is isolated from the other components of the code-signing infrastructure.

- Logical and physical access to the code-signing server is tightly controlled.
- A neutral third party witnesses the code-signing operation.
- The operations signer signs the submission by using the signing keys from the smart card and the code-signing software on the signing server.
- The operations signer timestamps the submission. This step is optional but highly recommended for code that is expected to function beyond the lifetime of the certificate.

  The options for timestamping a public release are:

  - Timestamp the signed file on the code-signing server. Use a global TSA over a separate dedicated Internet connection that is behind a strict firewall to ensure that the signing server is logically isolated from the internal network and the Internet.
  - Copy the signed file to a separate server that is used only for timestamping. This ensures that the signing server is never exposed to the Internet or corporate network.

  The options for an internal release are:

  - Use a hardware-based timestamp solution that is within the boundaries of the signing environment.
  - Use a global TSA, as described in the previous list.
- The operations signer returns the smart card to the neutral third party, who returns it to its secure storage.
- If the offline mode is used, the operations signer returns the signing server to its secure storage.
- The results of the operation are logged for auditing purposes.

Step 7: Archive Signed Code *Offline Manual Signing*

Who: *Operations Signer*

The signing procedure is archived, to allow auditing.

- The operations signer archives the signed files to removable archive media.
- Optionally—as a space-saving measure—the hash value of the submission is archived instead of the entire submission.
- The archive medium, like the virus scanning server, is logically and physically secure.
- Access to the archive medium is tightly controlled and restricted to a small group of highly trusted users. Separation of duties is used whenever possible to avoid reliance upon a single individual or team.
- When not in use, the archive medium is stored in a secure cabinet or safe.

Step 8: Return Signed Code *Offline Manual Signing*

Who: *Operations Signer*

There are multiple options for returning the signed package because it can be easily authenticated after it is signed.

- The data is returned by placing it on a file server, as long as it does not appreciably increase the attack surface of the code-signing service. Any such file server is behind a firewall, authenticates anyone requesting access, and

grants only read access to anyone who is not specifically tasked with posting the signed file.

- The signed data can also be returned on removable media such as a CD-ROM or USB drive.

Online Signing with Manual Approval

This example topology uses a manual submission process and an automated signing process. It is generally appropriate for medium and large organizations that develop complex software. As mentioned above, the operations that are illustrated in these topologies are somewhat independent of each other. Designers of code-signing infrastructures should consider combining different elements to create code-signing services that meet their requirements.

The online manual signing process has the following characteristics:

- Signings have a medium to high frequency.
- Developers are authenticated by a Web front end.
- Multiple approvers manually approve the submission. Approvers are also authenticated by the Web front end.
- A staging server automatically routes the submission through the virus-scanning, logging, and code-signing process.
- Private signing keys are stored on an HSM.
- The process uses dedicated code-signing, virus-scanning, and archive servers.

.

Figure 14 depicts a typical online manual topology, and the following sections describe the workflow.

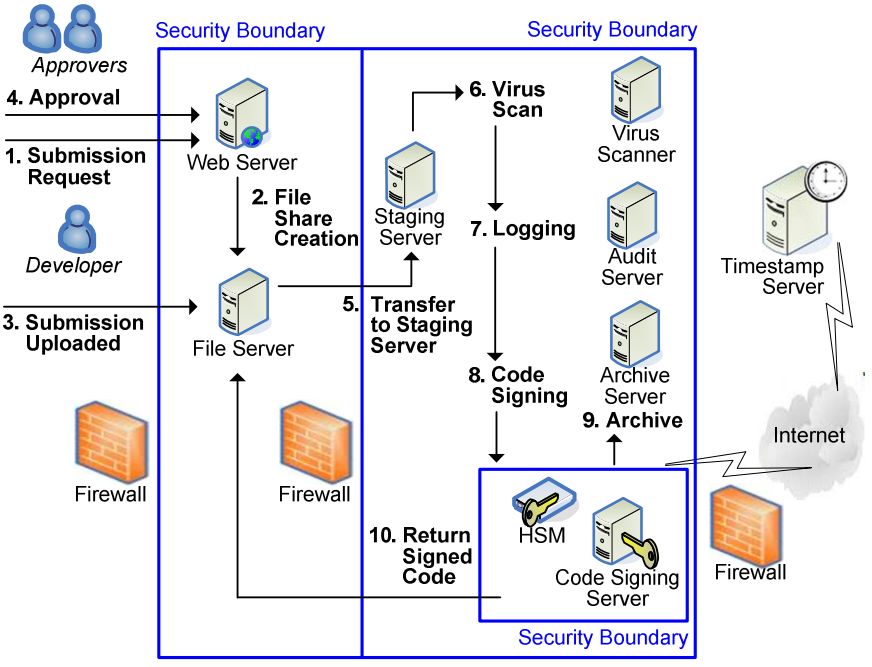

Figure 14. Online signing with manual approval topology

1. Submission Request *Online Signing with Manual Approval*

Who: *Developer*

The developer initiates the submission request by using a secure Web front end.

- The developer is authenticated by using SSL or IPSec.
- Only developers or build services have access to the Web front end.

2. File Share Creation *Online Signing with Manual Approval*

Who: *Web Front End*

The secure web form dynamically creates a file share to hold the submitted files.

- Only the Web front end can create this share.
- The Web front end configures the access control list for the file share so that only the submitter can upload files to it.
- Approvers have read access to the file share so that they can review the submission.

3. Submission Uploaded *Online Signing with Manual Approval*

Who: *Developer*

The developer uploads the unsigned files to the secured file share.

- All processing—other than signing—is complete. Changing the submission later, by processes such as rebasing files or localizing resource files, invalidates the signature.

4. Approval *Online Signing with Manual Approval*

Who: *Approver(s)*
Reference: "Code-Signing Submission and Approval Process"

Approvers review the submission.

- Approvers authenticate themselves to the Web front end by using SSL or IPSEC.
- There are at least two approvers according to best practices.

5. Transfer to Staging Server *Online Signing with Manual Approval*

Who: *File Server*

The submitted files are transferred to a staging server.

- The submission crosses a logical security boundary. This boundary is secured by using technologies such as firewalls, IPSec, and server-to-server authentication that uses digital certificates.

6. Virus Scan

Online Signing with Manual Approval

Who: *Staging Server; Virus-Scanning Server*

The submission is scanned for viruses.

- Scanning uses at least two commercial-grade antivirus applications.
- The virus-scanning server is physically and logically separated from the code-signing server.
- The virus-scanning server is operated by a separate individual to enforce separation of duties.
- The virus-scanning computer is maintained according to best practices:
  - MBSA is run regularly.
  - Virus signatures are up to date and are known to be in a good state.

7. Logging

Online Signing with Manual Approval

Who: *Staging Server; Audit Server*

The entire code-signing operation is logged for auditing purposes.

- The log is maintained in a SQL Server database on a separate audit server.
- The audit server is maintained according to best practices. In particular, MBSA is run regularly.
- The audit server is physically and logically separate from the code-signing server.
- To enforce separation of duties, the audit server is operated by a separate individual.

8. Code Signing

Online Signing with Manual Approval

Who: *Staging Server; Code-Signing Server*

The submission is sent online to the code-signing server for signing.

- The code-signing server is in a logically and physically secure room that is isolated from the other components of the code-signing infrastructure.
- Logical and physical access to the code-signing server is tightly controlled.
- The code-signing server is maintained according to best practices. In particular, MBSA is run regularly.
- The submission is signed by using signing keys on the HSM and the code-signing software on the signing server.
- The submission is timestamped. This step is optional but highly recommended for code that is expected to function beyond the lifetime of the certificate.

  The options for timestamping a public release are:

  - Timestamp the signed file on the code-signing server. Use a global TSA over a separate dedicated Internet connection that is behind a strict firewall to ensure that the signing server is logically isolated from the internal network and the Internet.
  - Copy the signed file to a separate server that is used only for timestamping. This ensures that the signing server is never exposed to the Internet or corporate network.

The options for an internal release are:

- Use a hardware-based timestamp solution within the boundaries of the signing environment.
- Use a global TSA, as described in the previous list.

9. Archive *Online Signing with Manual Approval*

Who: *Staging Server; Archive Server*

The signing procedure is archived, to allow auditing.

- Optionally—as a space-saving measure—the hash value of the submission is archived instead of the entire submission.
- The audit server, like the virus-scanning server, is logically and physically secure.
- Access to the audit server is controlled and restricted to a small group of highly trusted users. Separation of duties is used whenever possible.
- The audit server's data is backed up regularly and, along with the archive medium, stored in a secure location.

10. Return Signed Code *Online Signing with Manual Approval*

Who: *Staging Server; File Server*

The signed submission is returned to the file server.

- The file server is behind a firewall, authenticates anyone requesting access, and grants only read access to any entity that is not specifically tasked with posting the signed file.

Online Signing with Automated Approval

This example topology uses an automated submission process that is combined with an automated signing process. A fully automated signing infrastructure is required when the latency of the signing process has a significant impact on development times.

Deploying an automated submission and signing process extends the security boundary to include the entity that automatically submits the code. This entity is normally a build computer. Automated signing requires a layered security approach to ensure that only approved code is submitted to the build computer and from there to the signing service. For more information, see "Code-Signing Service Best Practices" earlier in this paper.

As mentioned in the preceding examples, the operations that are illustrated in these topologies are somewhat independent of each other. Designers of code-signing infrastructures should consider combining different elements to create code-signing services that meet their requirements

The online automated signing process has the following characteristics:

- High frequency of signings.
- Developers that authenticate to a dedicated build server.
- Automated approval.
- Build server that signs submissions by using signing keys that are stored on an HSM.

- Private signing keys that are stored on an HSM.
- Dedicated servers that are used for logging, virus scanning, and code signing.

Figure 15 depicts a typical online automated topology, and the following sections describe the workflow.

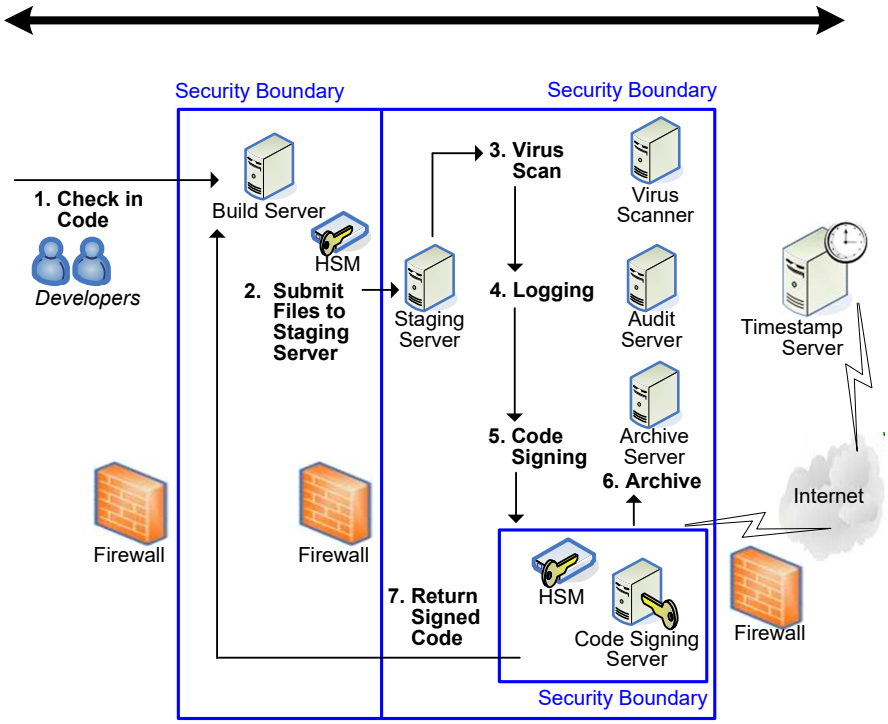

Figure 15. Online signing with automated approval topology

1. Check in Code *Online Signing with Automated Approval*

Who: *Developer; Build Server*

The developer submits source code to the build service.

- The application's source code is submitted to the build server.

  Note: Source control is a topic outside the scope of this paper.
- The developer is authenticated to the build server with SSL or IPSec.
- Access to the build server is restricted to developers.
- The build server then compiles the source into an application that will eventually be signed by the signing service.
- Optionally, the application can be signed by the build server before submission by using a private code-signing certificate and key that are stored in hardware, such as an HSM. The public key is known to the staging server and is used to authenticate the submission.

2. Submit Files to Staging Server *Online Signing with Automated Approval*

Who: *Build Server; Staging Server*

- The build server automatically submits the compiled application to the staging server.

- If the submission was signed, the staging server uses the public key to authenticate the submission.
- The submission crosses a logical security boundary that is secured by using technologies such as firewalls and IPSec. Optional additional network security features include:
  - An intrusion detection system
  - An isolated network link
- All processing—other than signing—is complete. Changing the submission later, by processes such as rebasing files or localizing resource files, invalidates the signature.

3. Virus Scan *Online Signing with Automated Approval*

Who: *Staging Server; Virus-Scanning Server*

- The submission is scanned for viruses.
- Scanning uses at least two commercial-grade antivirus applications.
- The virus-scanning server is physically and logically separated from the code-signing server.
- The virus-scanning server is operated by a separate individual to enforce separation of duties.
- The virus-scanning computer is maintained according to best practices:
  - MBSA is run regularly.
  - Virus signatures are up to date and are known to be in a good state.

4. Logging *Online Signing with Automated Approval*

Who: *Staging Server; Audit Server*

The entire code-signing operation is logged for auditing purposes.

- The log is maintained in a SQL Server database on a separate audit server.
- The audit server is maintained according to best practices. In particular, MBSA is run regularly.
- The audit server is physically and logically separate from the code-signing server.
- The audit server is operated by a separate individual to enforce separation of duties.

5. Code Signing *Online Signing with Automated Approval*

Who: *Staging Server; Code-Signing Server*

The submission is sent online to the code-signing server for signing.

- The code-signing server is in a logically and physically secure room that is isolated from the other components of the code-signing infrastructure.
- Logical and physical access to the code-signing server is tightly controlled.
- The code-signing server is maintained according to best practices. In particular, MBSA is run regularly.

- The submission is signed by using signing keys on the HSM and the code-signing software on the signing server.
- The submission is timestamped. This step is optional but highly recommended for code that is expected to function beyond the lifetime of the certificate.

  The options for timestamping a public release are:

  - Timestamp the signed file on the code-signing server. Use a global TSA over a separate dedicated Internet connection that is behind a strict firewall to ensure that the signing server is logically isolated from the internal network and the Internet.
  - Copy the signed file to a separate server that is used only for timestamping. This ensures that the signing server is never exposed to the Internet or corporate network.

  The options for an internal release are:

  - Use a hardware-based timestamp solution within the boundaries of the signing environment.
  - Use a global TSA, as described in the previous list.

6. Archive *Online Signing with Automated Approval*

Who: *Staging Server; Archive Server*

The signing procedure is archived, to allow auditing.

- Optionally—as a space-saving measure—the hash value of the submission is archived instead of the entire submission.
- The audit server, like the virus-scanning server, is logically and physically secure.
- Access to the audit server is controlled and restricted to a small group of highly trusted users. Separation of duties is used whenever possible.
- The audit server's data is backed up regularly and, along with the archive medium, stored in a secure location.

7. Return Signed Code *Online Signing with Automated Approval*

The signed submission is returned to the file server.

- The file server is behind a firewall, authenticates anyone requesting access, and grants only read access to any entity that is not specifically tasked with posting the signed file.

Code Signing for Managed Networks

Code signing is useful for deploying internal software applications such as installation packages, line-of-business applications, and ActiveX controls for use in a managed network environment. Network managers can use code signing to verify the integrity of signed applications during deployment and ensure that end users' installation and run-time experiences are free of unnecessary security warnings and dialog boxes. For details, see "Digital Signatures in Windows" earlier in this paper.

As with signing software for public release, signing software for deployment on a managed network should use a trusted code-signing service that performs all code-signing and related operations. This helps to ensure the integrity of the private keys

that are used for release signing and prevents the signing of applications from unintended sources. For more information regarding code-signing processes and the protection of private keys, see "Code-Signing Service Best Practices" earlier in this paper.

Network managers configure which applications are trusted within managed network environments by adding the code-signing certificates of trusted software publishers to the certificate stores of the computers that they manage. The trusted code-signing certificates can come from any combination of trusted third-party software publishers, an internal CA that the organization manages, or a certificate that a commercial CA issues to an organization.

Certificates from Trusted Third-Party Software Publishers

Network managers can choose to trust code-signing certificates that are owned by the software publishers whose applications are deployed on the managed network. A third-party software publisher's code-signing certificate is considered trusted within a managed network when it meets both of the following criteria:

- The code-signing certificate of the third-party software publisher is issued by a commercial CA that is configured in the Trusted Root Certificate Authorities certificate store.
- The software publishers' certificates are configured in either the user or machine Trusted Publishers certificate store of the managed computers. For more information on this procedure, see "Appendix 2.3. Certificate Manager Tool (CertMgr)" or "Appendix 2.4. Group Policy" of this paper.

This approach makes the default installation and run-time experience of trusted applications easier for end users without requiring an organization to deploy a CA. However, it has some limitations:

- There may be multiple software applications at issue, only some of which might be trusted by the network administrator and signed by the third-party software publisher's code-signing certificate.
- The security of the managed network becomes dependent on how well the software publisher protects its signing environment. If any of the publisher private keys are compromised, a window of opportunity could be created for an attacker to distribute code that appears to be legitimately signed.
- This solution applies only to signed third-party applications: it offers no mechanism for signing applications that are developed in-house.

Certificates from an Internal CA

The primary benefit of deploying an internal CA is that the CA can be used to sign internally developed applications. Trusted internal CA certificates are also more easily managed than third-party certificates. Network managers can use Group Policy or auto-enrollment with Active Directory to configure the Trusted Root Certification Authorities certificate store of the managed environment with the root certificate of the internal CA.

In addition, the network manager can configure specific *end-entity* code-signing certificates—issued by a third-party or internal CA—to be trusted by adding those certificates to the Trusted Publishers certificate store of trusted computers. For more information on certificate configuration mechanisms, see Appendix 2 and for more information on deploying Microsoft Certificate Server, see Appendix 4 of this paper.

Another benefit of deploying an internal CA is to maintain a higher granularity of control over which applications are trusted within the network. By trusting a third-party software publisher's certificate on a managed network, the network manager has chosen to trust all of applications that are signed by using that certificate. However, a network manager can limit trust to only a specified subset of these applications by using a certificate from the internal CA to sign them. Only these explicitly re-signed applications are treated as trusted.

Note: It is important that organizations only re-sign applications for use within the organization's managed network. Microsoft does not recommend re-signing another software publisher's software with the intent of publicly releasing the software without the consent of the original publisher.

Certificates from a Commercial CA

Another option for signing applications for internal use is to acquire a code-signing certificate that was issued by a third-party commercial CA. The advantage of this approach is that it does not require the organization to deploy an internal CA. In addition, applications that are signed by using a certificate from a commercial CA can be trusted outside the managed network. This is relevant if the organization is sharing applications externally.

The disadvantages of using certificate from a commercial CA are:

- The organization must depend on the commercial CA for new certificates and updates.
- Commercial CAs require a comprehensive verification process before issuing code-signing certificates. This process can be time consuming.
- The quality and cost of service depend on the third-party vendor.
- Certain classes of commercial CA certificates are available only in specific geographic regions.

Note: Windows Vista x64 kernel-mode software must be signed by using a certificate that a trusted Commercial CA issued. For further information, see "Digital Signatures for Kernel Modules on x64-based Systems Running Windows Vista."

Software Restriction Policies for Managed Networks

If a network manager wants to restrict the software that network users download and run to only applications that are signed by trusted publishers, the manager can deploy a software restriction policy (SRP) in conjunction with a code-signing infrastructure. With an SRP, administrators can control which programs run on a computer and decide who can add trusted publishers to the computer. For more information about software restriction policies, see "Resources" at the end of this paper.

Resources

The following links provide further information about Windows code-signing and related topics.

Introduction

Digital Signatures for Kernel Modules on x64-based Systems Running Windows Vista
http://www.microsoft.com/whdc/system/platform/64bit/kmsigning.mspx

Cryptographic Services
http://go.microsoft.com/fwlink/?LinkId=95779

Code-Signing Basics

Introduction to Code Signing
http://go.microsoft.com/fwlink/?LinkId=95781

Authenticode
http://www.microsoft.com/technet/archive/security/topics/secaps/authcode.mspx?mfr=true

Public-Key Infrastructure (X.509) Certificates
http://www.ietf.org/html.charters/pkix-charter.html

Signing Technologies in Windows

Deploying Authenticode with Cryptographic Hardware for Secure Software Publishing
http://go.microsoft.com/fwlink/?LinkId=70870

Using MakeCat
http://go.microsoft.com/fwlink/?LinkId=95790

Sign Tool (Signtool.exe)
http://go.microsoft.com/fwlink/?LinkId=70873

Certificate Manager Tool (Certmgr.exe)
http://go.microsoft.com/fwlink/?LinkId=95775

Security Briefs: Strong Names and Security in the .NET Framework
http://go.microsoft.com/fwlink/?LinkId=95783

Using Catalog Files
http://go.microsoft.com/fwlink/?LinkId=95788

Certificate Creation Tool (Makecert.exe)
http://go.microsoft.com/fwlink/?LinkId=70872

CLR Inside Out: Using Strong Name Signatures
http://go.microsoft.com/fwlink/?LinkId=95776

Timestamping references:
Signing and Checking Code with Authenticode:
http://go.microsoft.com/fwlink/?LinkId=95785

GeoTrust Code Signing Credentials for Java and Microsoft Authenticode:
http://www.geotrust.com/codesigning/java_ms_authenticode/faq.htm

Inf2Cat
https://winqual.microsoft.com/

Strong Name Tool (Sn.exe)
http://go.microsoft.com/fwlink/?LinkId=95787

Microsoft Windows XP: Using Software Restriction Policies to Protect Against Unauthorized Software
http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/rstrplcy.mspx

Kernel-Mode Code Signing Walkthrough
http://www.microsoft.com/whdc/winlogo/drvsign/kmcs_walkthrough.mspx

Digital Signatures in Windows

Windows Vista Logo for Software
https://partner.microsoft.com/global/30000104

Microsoft Windows XP: Using Software Restriction Policies to Protect Against Unauthorized Software
http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/rstrplcy.mspx

Code Signing for Protected Media Components in Windows Vista
http://go.microsoft.com/fwlink/?LinkId=70876

Driver Signing Requirements for Windows
http://www.microsoft.com/whdc/winlogo/drvsign/drvsign.mspx

Driver Package Integrity during Plug and Play Device Installs in Windows Vista
http://www.microsoft.com/whdc/winlogo/drvsign/pnp-driver.mspx

Digital Signatures for Kernel Modules on x64-based Systems Running Windows Vista
http://www.microsoft.com/whdc/system/platform/64bit/kmsigning.mspx

Microsoft Cross-Certificates for Windows Vista Kernel Mode Code Signing
http://go.microsoft.com/fwlink/?LinkId=66583

Output Content Protection and Windows Vista
http://www.microsoft.com/whdc/device/stream/output_protect.mspx

Windows Defender
http://www.microsoft.com/athome/security/spyware/software/default.mspx

Code Signing during Software Development

Certificate Manager Tool (Certmgr.exe)
http://go.microsoft.com/fwlink/?LinkId=70874

Windows XP Professional: Certificate Revocation and Status Checking
http://go.microsoft.com/fwlink/?LinkId=70869

Digital Signatures for Kernel Modules on x64-based Systems Running Windows Vista
http://www.microsoft.com/whdc/system/platform/64bit/kmsigning.mspx

MakeCert
http://go.microsoft.com/fwlink/?LinkId=95782

Driver Signing Requirements for Windows
http://www.microsoft.com/whdc/winlogo/drvsign/drvsign.mspx

Strong Name Tool (Sn.exe)
http://go.microsoft.com/fwlink/?LinkId=95787

Timestamping references
Signing and Checking Code with Authenticode:
http://go.microsoft.com/fwlink/?LinkId=95785

GeoTrust Code Signing Credentials for Java and Microsoft Authenticode:
http://www.geotrust.com/codesigning/java_ms_authenticode/faq.htm

SignTool (Signtool.exe)
http://go.microsoft.com/fwlink/?LinkId=95784

Code-Signing Service Best Practices

Deploying PKI Inside Microsoft
http://www.microsoft.com/technet/itsolutions/msit/security/deppkiin.mspx

The Antivirus Defense-in-Depth Guide
http://www.microsoft.com/technet/security/topics/serversecurity/avdind_3.mspx

Security Content Overview
http://www.microsoft.com/technet/security/bestprac/overview.mspx

Security Architecture Blueprint
http://www.microsoft.com/technet/itsolutions/wssra/raguide/ArchitectureBlueprints/rbabsa.mspx?mfr=true

Public Key Infrastructure for Windows Server 2003
http://www.microsoft.com/windowsserver2003/technologies/pki/default.mspx

Microsoft Internet Security and Acceleration Server 2006
http://www.microsoft.com/isaserver/default.mspx

Server and Domain Isolation
http://www.microsoft.com/technet/itsolutions/network/sdiso/default.mspx

Best Practices for Implementing a Microsoft Windows Server 2003 Public Key Infrastructure
http://technet2.microsoft.com/WindowsServer/en/Library/091cda67-79ec-481d-8a96-03e0be7374ed1033.mspx?mfr=true

Windows Server 2003 PKI Operations Guide
http://technet2.microsoft.com/WindowsServer/en/Library/e1d5a892-10e1-417c-be13-99d7147989a91033.mspx?mfr=true

Code-Signing Service Example Topologies

Microsoft Security Baseline Analyzer
http://www.microsoft.com/technet/security/tools/mbsahome.mspx

The Antivirus Defense-in-Depth Guide
http://www.microsoft.com/technet/security/topics/serversecurity/avdind_3.mspx

Server and Domain Isolation
http://www.microsoft.com/technet/itsolutions/network/sdiso/default.mspx

Windows Server 2003 Security Guide
http://www.microsoft.com/downloads/details.aspx?familyid=8A2643C1-0685-4D89-B655-521EA6C7B4DB&displaylang=en

Code Signing for Managed Networks

Windows XP: Using Software Restriction Policies to Protect Against Unauthorized Software
http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/rstrplcy.mspx

Introduction to Group Policy in Windows Server 2003
http://www.microsoft.com/windowsserver2003/techinfo/overview/gpintro.mspx

Certificate Manager Tool (Certmgr.exe)
http://go.microsoft.com/fwlink/?LinkId=95775

MakeCert
http://go.microsoft.com/fwlink/?LinkId=95782

Designing a Public Key Infrastructure
http://technet2.microsoft.com/WindowsServer/en/Library/b1ee9920-d7ef-4ce5-b63c-3661c72e0f0b1033.mspx?mfr=true

Others

Add a trusted root certification authority to a Group Policy object
http://technet2.microsoft.com/WindowsServer/en/Library/4b7ea7f9-311a-479b-aecc-c856165b97c11033.mspx?mfr=true

Certificate stores
http://technet2.microsoft.com/WindowsServer/en/Library/1c4d3c02-e996-450a-bf4f-9a12d245a7eb1033.mspx

Strong Name Tool (Sn.exe)
http://msdn2.microsoft.com/en-us/library/k5b5tt23(vs.71).aspx

Implementing and Administering Certificate Templates in Windows Server 2003
http://technet2.microsoft.com/WindowsServer/en/Library/c25f57b0-5459-4c17-bb3f-2f657bd23f781033.mspx

Platform SDK Redistributable: CAPICOM
http://www.microsoft.com/downloads/details.aspx?FamilyID=860ee43a-a843-462f-abb5-ff88ea5896f6&DisplayLang=en

Appendix 1. Generating Test Certificates with MakeCert

MakeCert is a simple command-line tool that generates test code-signing certificates that are either self signed or signed by the Root Agency certificate.

Note: This paper recommends that code signers use self-signed certificates for test signing, when possible. Anyone, including malicious code writers, can use MakeCert to create certificates that are signed by the Root Agency certificate. Adding the Root Agency certificate to the system certificate stores can reduce the security of test computers because doing so allows Root Agency–issued certificates to be used for purposes other than verifying the signatures of test-signed applications.

To create a self-signed test certificate, run MakeCert from the command line as follows:

```
MakeCert.exe –r –pe –sr CurrentUser –ss TestCertStore –n
"CN=TestCert" TestCertFile.cer
```

MakeCert arguments:

- **-r**: Specifies that the certificate is self signed, that is, the certificate is a root certificate.
- **-pe**: Specifies that the private key that is associated with the certificate can be exported. This is useful if it is necessary to migrate the private key to another computer.
- **–sr CurrentUser**: Specifies that the test certificate is created in the CurrentUser certificate store. To add the certificate to the machine store, specify **LocalMachine.** Note that adding certificates to a machine store requires administrator credentials.
- **-ss *TestCertStore***: Specifies the name of the certificate store that contains the test certificate as TestCertStore.
- **-n** "**CN=*TestCert***": Specifies the subject name of the certificate as TestCert. The subject name should clearly identify the certificate as a test certificate, for example, "Contoso Testing Cert - for in-house use only."
- ***TestCertFile.cer***: The name of the file that contains a copy of the test certificate. It can be used to configure a test computer as described in "Appendix 2. Configuring System Certificate Stores" of this paper. The file does not contain the private key.

For more information on MakeCert, see “Resources” for links to the MSDN documentation.

Appendix 2. Configuring System Certificates Stores

This section discusses several different ways to configure system certificate stores:

- The Certificate Import Wizard and MMC Certificates snap-in are best used for individual computers.
- The CertMgr command-line tool can be used for an individual computer or can be scripted to configure a managed network.
- Group Policy can be used only by a network administrator in a domain environment.

For more information on system certificate stores including Trusted Root Certification Authorities, Trusted Publishers, and machine and user certificate stores, see "Configuring a Test Computer/Environment" earlier in this paper.

Appendix 2.1. Certificate Import Wizard

To add a certificate to a system certificate store with the Certificate Import Wizard:

1. Launch Windows Explorer.
2. Navigate to the directory that contains the desired certificate file. Supported file types include .cer, spc, and .pfx.
3. Right-click the certificate file and click **Install Certificate** on the shortcut menu to launch the Certificate Import Wizard. Click **Next** to go to the second page.
4. From the list, select **Place all certificates in the following store** and click **Browse**.
5. Select the desired certificate store, such as **Trusted Root Certification Authorities** or **Trusted Publishers.**
6. Click **Okay**, click **Next,** and click **Finish** to complete the procedure.

If the certificate must be placed in multiple certificate stores, repeat the procedure for each store.

Note: This procedure can be used only to add a certificate to the user's certificate store. It cannot be used to add a certificate to a machine store.

Appendix 2.2. MMC Certificates Snap-in Wizard

To import a test certificate into a system certificate store, use the MMC certificates snap-in, as follows:

1. On the **Start** menu, click **Run**, and enter "mmc" to launch MMC.
2. On the MMC window's **File** menu, click **Add\Remove Snap-in**.
3. On the **Standalone** tab of the **Add/Remove Snap-in** dialog box, click **Add**.
4. From the drop-down list, select **Certificates** and click **Add** to launch the Certificates Snap-in Wizard.
    - To add the certificate to a user store, select **My user account** and click **Finish**.
    - To add the certificate to a machine certificate store, select **Computer Account** and click **Next**. In the **Select Computer** dialog box, select **Local Computer** and then click **Finish** to add the certificate to the machine certificate store.

5. In the **Add Standalone Snap-in** dialog box , click **Close.** In the **Add/Remove Snap-in** dialog box, click **OK**.
6. In the left pane of the **MMC** window, expand **Certificates**. and then click the desired certificate store, such as **Trusted Root Certification Authorities** or **Trusted Publishers**.
7. On the **Action** menu, click **All Tasks**, and then click **Import** to launch the Certificate Import Wizard. Then click **Next** to go to the second page.
8. In the **File Name** edit box, enter the name of the file that contains the certificate and click **Next**. An alternative is to click **Browse** and navigate to the file.
9. Click **Place all certificates in the following store,** and then click **Browse** to open the **Select Certificate Store** dialog box. From the drop-down list, select **Trusted Root Certification Authorities** and click **OK**.
10. Click **Next,** and then click **Finish**.

Note: Before a test certificate can be added to the Trusted Root Certification Authorities certificate store, the system displays a **Security Warning** dialog box. Click **Yes** to allow the certificate to be added.

Appendix 2.3. Certificate Manager Tool (CertMgr)

CertMgr is a command-line utility that is distributed as part of the WinSDK and WDK. In the context of code signing, it is particularly useful for configuring certificates in nondomain-joined test environments. This section gives several typical examples of how to use CertMgr.

Note: CertMgr refers to system certificate stores by different names than are used elsewhere in this paper or by the Certificates MMC snap-in. For successful results with CertMgr, be sure to use the certificate store names in the following examples.

- Add a certificate to the Trusted Root Certification Authorities machine certificate store:

```
certmgr.exe –add TestCertFile.cer –s –r localMachine root
```

- Add a certificate to the Trusted Publishers machine certificate store:

```
certmgr.exe –add TestCertFile.cer –s –r localMachine
trustedpublisher
```

- Add a certificate to the Trusted Root Certification Authorities user certificate store:

```
certmgr.exe –add TestCertFile.cer –s –r currentUser root
```

- Add a the certificate to the Trusted Publishers user certificate store:

```
certmgr.exe –add TestCertFile.cer –s –r currentUser
trustedpublisher
```

CertMgr arguments:

- **-add**: Adds the certificate in the specified certificate file to the specified certificate store.
- **-s**: Specifies that the certificate store is a system store.

- **-r** *SystemStore*: Specifies the system certificate store location. The default location is "currentUser." Note that adding certificates to the localMachine certificate store requires administrator credentials.
  - For the user store, use "currentUser".
  - For the machine store, use "localMachine".
- CertificateStore: The certificate store.
  - For Trusted Publisher, use "trustedpublisher".
  - For Trusted Root Certification Authorities, use "root".

For more information on CertMgr, see "Resources" at the end of this paper.

Appendix 2.4. Group Policy

Domain administrators use Group Policy settings to define specific configurations for groups of users and computers. Domain administrators can use Group Policy settings to add certificates to the system certificate stores for computers that are members of a domain. This can be accomplished by configuring the default domain security policy or by configuring a Group Policy object (GPO). With a GPO, Group Policy can be applied to a site, an entire domain, or an organizational unit.

The certificate stores of the computers in the domain are updated each time the Group Policy is applied. This happens approximately every 90 minutes and each time a computer is restarted. In addition, a computer can force an update of the default domain policy by using the `gpupdate /force` command.

Administrators edit policy settings with the MMC Group Policy Object Editor snap-in. To use the snap-in to deploy code-signing certificates to the system certificate stores of multiple users:

1. On the **Start** menu, click **Run**, and enter "mmc" to launch MMC.
2. On the **MMC** window's **File** menu, click **Add\Remove Snap-in**.
3. On the **Standalone** tab of the **Add/Remove Snap-in** dialog box, click **Add**.
4. From the list of available snap-ins, select **Group Policy Object Editor**, and click **Add**.
   - To edit the Group Policy object for the local computer, click **Finish** and then click **Close**.
   - To edit the Group Policy object for a domain, click **Browse** and either select **Default Domain Policy** or specify a domain. Then click **Finish**.
5 If there are no more snap-ins to add to the console, click **OK**.
6. In the left pane of the **Console** window, expand **Local Security Policy** or **Default Domain Policy**, depending on the choice made in Step 4. Then expand **Computer Configuration, Windows Settings, and Security Settings** and click **Public Key Policies**.
7. Right-click the desired system certificate store, such as **Trusted Root Certification Authorities** or **Trusted Publishers**.
8. On the shortcut menu, click **Import** to launch the Certificate Import Wizard and use it to import the certificates.

Appendix 3. Microsoft Certificate Server Deployment

The details of deploying and operating Microsoft Certificate Server as an internal CA are already well described by a substantial set of existing documentation. The scope of this paper is restricted to describing how CAs are useful to an organization within the context of code signing. For detailed information about deploying and operating a Microsoft Certificate Server, see the following white papers:

Best Practices for Implementing a Microsoft Windows Server 2003 Public Key Infrastructure
http://technet2.microsoft.com/WindowsServer/en/Library/091cda67-79ec-481d-8a96-03e0be7374ed1033.mspx

Microsoft Public Key Infrastructure for Windows Server 2003
http://www.microsoft.com/pki/

Deployment Planning (Best Practices for Implementing a Microsoft Windows Server 2003 Public Key Infrastructure)
http://technet2.microsoft.com/WindowsServer/en/Library/c3f67fb4-a1ae-43ed-b30e-fe1b183a553d1033.mspx

Implementing and Administering Certificate Templates in Windows Server 2003
http://technet2.microsoft.com/WindowsServer/en/Library/c25f57b0-5459-4c17-bb3f-2f657bd23f781033.mspx

Certificate Autoenrollment in Windows Server 2003
http://www.microsoft.com/technet/prodtechnol/windowsserver2003/technologies/security/autoenro.mspx

General CA Deployment Considerations

An organization can deploy a Microsoft Certificate Server as an internal CA to issue certificates for code signing. Microsoft Certificate Server is included in some versions of Microsoft Windows Server. A Microsoft Certificate Server is typically most effective when deployed with domain and Active Directory services.

Deploying a CA offers the following advantages:

- Test signing is easier because each developer in the organization can be automatically given his or her own test-signing certificate and private key.
- Code-signing certificates can be easily issued, renewed, and revoked by CA administrators and delegates.
- Test signing enables code signing of in-house applications for secure deployment and enhanced user experience within a managed network.
- Test signing eliminates dependencies on third-party commercial CAs for deploying signed applications to a managed network.
- Test signing provides more granular control over which third-party software publishers' applications are trusted in the network environment.
- Combined with SRPs, an internal CA can provide a granular application trust policy that prevents the use of unsigned applications within the managed environment.

Best Practices for Deploying an Internal CA for Code Signing

The PKI and Certificate Server documentation listed above discusses the basics of installation, configuration, and best practices. The following topics are relevant when using a Microsoft Certificate Server for code signing.

Automated Deployment by Using Group Policy and Active Directory

Microsoft recommends the use of Group Policy and Active Directory because this combination supports automated wide-scale deployment of certificates.

Group Policy supports automatic certificate enrollment for Active Directory users. Users can receive certificates of various types—such as smartcard user and code signing—when they first log on to a domain. Auto-enrollment allows administrators to deploy certificates throughout the enterprise while requiring no user interaction. Additional auto-enrollment behaviors can also be defined for individual templates such as users not receiving a certificate if they already have a valid certificate published in Active Directory.

For domain-joined computers, auto-enrollment automatically downloads and manages trusted root certificates, cross-certificates, and NTAuth certificates from Active Directory into the local machine registry. All users who log on to the computer inherit the trust and downloaded certificates that are managed by auto-enrollment.

Note: Users and CAs must be in the same Active Directory forest.

Code-Signing Certificate Templates

All certificates that are issued from the Microsoft Certificate Service are based on certificate templates. The templates reside in Active Directory and define the parameters of what is in a certificate and who is entitled to receive it. Typically, the templates pull subject information directly from Active Directory. In doing so, the templates normally require enrollees to have an associated Active Directory account.

Certificate requesters can—depending on their access rights—select from a variety of certificate types that are based on associated templates, such as exchange user or code signing.

Organizations can create and customize certificate templates to meet specific requirements such as code signing. If necessary, certificate templates are easily replaced or updated. Updated certificate templates supersede existing templates.

Revocation

In most PKI deployments, only certificates that have not yet expired are placed on the CRL. This means that when a revoked certificate expires, it is eventually removed from the CRL. However, timestamping allows Authenticode signatures to remain valid indefinitely. As a result, revoked code-signing certificates should not be removed from the CRL. This paper recommends always revoking compromised certificates that were used to sign publicly released software and maintaining these entries in the CRL indefinitely.

Microsoft Certificate Server enables administrators to configure the CA to keep all revoked certificates in the CRL. However, the CA does not distinguish between code-signing certificates and other certificates. For this reason, administrators should dedicate a CA for the sole purpose of handling code-signing production certificates. This paper makes no recommendation on revoking test code-signing certificates because the certificates are trusted only within the test environment.

Organizations must resolve this question themselves based on the value of the required test certificates and resources to revoke all test certificates generated by the test CA.

Appendix 4. Sign Tool (SignTool)

SignTool is a command-line tool that is used to sign, verify, and timestamp files. This appendix describes how to use SignTool for these purposes. The “Resources” section contains a link to MSDN that describes additional SignTool functionality such as:

- Advanced certificate and private key selection
- Catalog database operations
- Description information

Note: Signtool.exe depends on Capicom.dll version 2.1.01, which is distributed with the WDK, Platform SKD, and WinSDK. If signing outside these development kits' command-line environments, be sure to copy the correct version of capicom.dll into the directory in which Signtool.exe is located. Capicom.dll is also redistributed through MSDN. For details, see "Resources" at the end of this paper.

Note: x64 kernel-mode code signing and signature verification require different files and command-line syntax for embedded signing drivers. For details, see “Digital Signatures for Kernel Modules on x64-based Systems Running Windows Vista.”

SignTool Sign and SignTool Timestamp

These examples demonstrate how to test-sign files with the self-signed certificate that was generated in Appendix 1. The certificates are selected in the examples below by specifying the certificate's subject name in the user's system certificate stores. Other certificate selection command-line arguments that are not discussed here include:

- /SHA1: Selects a certificate by its SHA1 hash value.
- /F: Specifies a PFX file.
- /P: Specifies a PFX file’s password.

Note: Signing with a private key that is stored in a hardware device is more secure than signing with a PFX file. For more discussion of this topic, see "Code-Signing Services Best Practices" earlier in this paper. However, for scenarios where a hardware solution is not available or practical, PFX is the only file format that is supported by the SignTool command-line syntax. SPC and PVK files are deprecated for signing purposes. If storing the private key in software is necessary, use PVK2PFX to convert SPC and PVK files to a password-protected PFX format.

In the examples in this appendix, the file to be signed can be any file type that supports a native Authenticode signature, including catalog and binary files. For more information, see "Signing Technologies in Windows" earlier in this paper.

The examples also show variations in the syntax that is used for timestamping signatures, as discussed in "Timestamping Operations" earlier in this paper. The timestamping URL that was used here belongs to Verisign. However, the URL of any Authenticode TSA can also be used.

Note: There are actually two Verisign DLLs for timestamping: timestamp.dll and timstamp.dll. They differ only in name, so either can be used.

Signing a target file

The following command line signs a target file with a specified certificate.

```
Signtool sign /v /s TestCertStore /n TestCert targetfile
```

SignTool signing arguments:

- **sign**: Signs the specified target file.
- /**v**: Specifies the verbose option, which displays successful execution, warning messages, and the certificate chain.
- /**s**: *CertificationStoreName*: Specifies the certificate store that contains the test certificate.
- /**n**: *TestCertificateName*: Specifies the test certificate with the subject name.
- ***FileName***: Specifies the name of the target file to be timestamped.

Timestamping a target file

The following command-line timestamps a signed target file.

```
Signtool timestamp /v /t
    http://timestamp.verisign.com/scripts/timestamp.dll targetfile
```

SignTool timestamp arguments:

- **timestamp**: Timestamps the specified file.
- /**v**: Specifies the verbose option, which displays successful execution and warning messages.
- /**t *URL***: Specifies a digital signature, timestamped by the TSA that the URL indicated.
- ***FileName***: The name of the target file to be signed.

Signing and Timestamping a Target File

The following command line signs and timestamps a target file as a single operation. For an explanation of the arguments, see the previous two examples.

```
Signtool sign /v /s TestCertStore /n TestCert /t
    http://timestamp.verisign.com/scripts/timestamp.dll targetfile
```

Signature Verification with SignTool

The SignTool verify option checks the validity of the signature and its certificate chain. The root certificate of the CA that issued the test certificate must be configured in the Trusted Root Certification Authorities user or machine certificate store before verification. For further information on installing the test CA certificate, see Appendix 2 of this paper.

The following command line verifies the signature in the specified target file.

```
Signtool verify /pa /v targetfile
```

SignTool arguments:

- **verify**: Verifies the signature in the specified file.
- /**pa**: Uses the Authenticode verification policy when verifying the signature.
- /**v**: Specifies the verbose option, which displays successful execution and warning messages.
- ***FileName***: Specifies the name of the binary file that contains the signature to be verified,

Appendix 5. Signing with PVK and PFX files

Although Microsoft suggests that ISVs release-sign software with private keys that are stored in hardware, some ISVs choose to use keys that are stored in software instead. This section describes how to use private keys that are stored in software for code signing with Windows.

Appendix 5.1. Converting PVK to PFX files with PVK2PFX

Some certification authorities use the PVK file format to store the digital certificate's private key. This format uses a pair of files. The public part of the digital certificate is stored in an .spc or .cer file. The private key is stored in a .pvk file. If you have a Verisign Class 3 certificate, these digital certificates are currently provided to you in a pair of .pvk and .spc files. Before you can use the certificate for code signing, you must convert the .pvk and .spc files into the PFX format.

You can use the Windows Platform SDK tool PVK2PFX to convert these files to .pfx format. The syntax for using PVK2PFX is:

```
pvk2pfx.exe –pvk MyPrivateKey.pvk –pi PVK_password –spc
    MyCertificate.spc –pfx MyPFX.pfx –po PFX_password
```

PVK to PFX arguments:

- **-pvk *MyPrivateKey.pvk***: The name of the PVK file that contains the private key that is used for signing the software. If this file is password protected, the **-pi** argument might be required.
- **-spc *MyCertificate.spc***: The name of the SPC file that contains the public code-signing certificate that is associated with the PVK file.
- **-pfx *MyPFX.pfx***: The name of the PFX file to be generated by PVK2PFX. Note: if the **-pfx** argument is not used, PVK2PFX displays an export wizard. In this case, the **-po** and **-f** arguments are ignored.
- -**pi *PVK_password***: The password that is used to access the private key of a password-protected PVK file.
- **-po *PFX_password***: The PFX password that is used to encrypt the private key in the PFX file. If this argument is omitted, the tool uses the password that was specified by the **-pi** argument.

An additional argument, not shown in the example:

-f: Forces PVK2PFX to overwrite an existing PFX file with the same name.

Appendix 5.2. Importing PFX Files

The PFX format is used to store a certificate and a password-protected private key. SignTool is designed to accept .pfx files for code signing. The SignTool command options /**f** *SignCertFile* and /**p** *Password* enable code signing by using a certificate and private key from a .pfx file.

Note: Developers who sign x64 kernel-mode drivers cannot use the SignTool /f argument for release signing because the /f and /ac arguments are currently mutually exclusive. This functionality will be supported in a future version of SignTool. The workaround is to import the certificate from the PFX file into a local Personal certificate store:

1. Launch Windows Explorer and double-click the PFX file to open the Certificate Import Wizard.
2. To go to the **File to Import** page, click **Next**.

3. Examine the string in the **File name** edit box to confirm that the file path is correct, and then click **Next** to go to the **Password** page.
4. Enter the PFX file's password, select the **Mark the key as exportable** check box, and click **Next** to go to the **Certificate Store** page.

   Important: If you do not select the check box, it will be more difficult to delete the private key later.
5. To add the certificate to the Personal store and go to the wizard's completion page, click **Next** . Otherwise, specify a store and click **Next**.
6. Examine the settings on the completion page and click **Finish** to complete the wizard.

Appendix 5.3. Removing Certificates and Private Keys from Windows

When you have finished code signing, you might want to remove the certificate and private key from the code-signing computer. This can be accomplished through the MMC Certificates snap-in that is described in the following steps:

Note: CertMgr can be used for this purpose, but it removes only the certificate from the system certificate stores. The private key container remains on the system. Use the following method to remove the certificate and private key at the same time.

The first step is to launch MMC and open the certificate store that contains the certificate:

1. On the **Start** menu, click **Run**, and enter "mmc" to launch MMC.
2. In the MMC window's **File** menu, click **Add\Remove Snap-in**.
3. On the **Standalone** tab of the **Add/Remove Snap-in** dialog box, click **Add**.
4. From the drop-down list, select **Certificates**, and click **Add** to launch the Certificates Snap-in wizard.
   - To add the certificate to a user store, select **My user account** and click **Finish**.
   - To add the certificate to a machine certificate store, select **Computer Account** and click **Next**. In the **Select Computer** dialog box, select **Local Computer,** and then click **Finish** to add the certificate to the machine certificate store.
7. To close the **Add Standalone Snap-In** dialog box, click **Close.** To close the **Add/Remove Snap-In** dialog box, click **OK** .
8. In the left pane of the **MMC** window , expand **Certificates**, and then click the desired certificate store, such as **Trusted Root Certification Authorities** or **Trusted Publishers**.

To delete the certificate and key:

1. Click the desired certificate. On the **Action** menu, click **All Tasks,** and then click **Export** to launch the Certificate Export Wizard.
2. To go to the wizard's **Export Private Key** page, click **Next**, select **Yes, export the private key**, and then click **Next** to go to the **Export File Format** page.

   Note: if the **Yes, export the private key** option is not available, reimport the PFX file as described in the previous section. Be sure to mark the private key as exportable and try these steps again.

3. Select the **Personal Information Exchange – PCKS #12 (.PFX))** check box. Under that check box, select the **Enable strong protection** and **Delete the private key if the export is successful** check boxes. To go to the **Password** page, click **Next**.
4. Enter a strong password in both edit boxes, and click **Next** to go to the **File to Export** page.
5. Click **Browse** and navigate to a location that can be easily remembered and is accessible to a limited set of users. Assign a name to the exported PFX file and click **Close** to create the file and return to the wizard. Then click **Next** to go the wizard's completion page.
6. Examine the data on the completion page and click **Finish** to complete the wizard. Click **OK** on the subsequent message box stating that the export was successful.
7. Return to the MMC certificates snap-in and right-click the certificate. On the shortcut menu, click **Delete,** and then click **Yes** on the **Certificates** message box to delete the certificate.
8. Navigate Windows Explorer to the PFX file that was exported earlier in this procedure. Select the file, hold down the SHIFT key to prevent the deleted file from being placed in the Recycle Bin, and press the DELETE key. An alternative approach is to delete the file normally and then delete it from the Recycle Bin.