Safe Loader

Roey Gross, Yair Lasri



Instructor: Arie Haenel

2024 – 2nd Sem.  
Jerusalem College Of Technology (JCT)

Computer Science

Table of content:

[Abstract 2](#_Toc172234483)

[Introduction 2](#_Toc172234484)

[Threat Models: 3](#_Toc172234485)

[Security Assumptions 4](#_Toc172234486)

# Abstract

Dynamically linked libraries (DLLs) are critical components that enable executables (EXEs) to extend their functionality by loading additional code at runtime. Sometimes a high-authorization EXE can load DLLs without adequately verifying their signatures. The EXE is assumed to be trusted, but the DLLs aren’t. An attacker can exploit this weakness by replacing legitimate DLLs with malicious ones, compromising the integrity of the signed EXE.

To counter this threat, we develop a robust framework for signing DLLs and ensuring that an EXE validates these signatures at load time, before any library code is executed.

Our solution involves a signer program and a validating program. The signer program signs the DLLs. The validating program uses IAT hooking to intercept the EXE's DLLs loading proccess, verify the DLL's signature, and load only trusted DLLs.

We will demonstrate that our approach ensures that even if an attacker can add or replace DLLs, only verified DLLs will be executed, maintaining the integrity and security of the system, and completing the chain of trust.

# Introduction

Imagine a scenario where a meticulously signed executable, trusted by its users and the operating system, unwittingly becomes a gateway for malicious actors. This threat unfolds when attackers exploit a critical vulnerability: the unchecked validation of dynamically linked libraries (DLLs) loaded by the executable.

In this typical case, the attacker gains access to the directory housing the signed executable. Here, they discreetly substitute genuine DLLs with their malicious counterparts, designed to execute unauthorized commands or steal sensitive information. Despite the executable being rigorously authenticated through Windows' signing mechanisms, these rogue DLLs evade detection, leveraging the executable's implicit trust to operate within the system undetected.

To further complicate matters, attackers can also exploit the system's search paths for DLLs. By placing a rogue DLL with the same name in a directory early on the search path, they ensure that their malicious DLL is loaded instead of the legitimate one intended for use by the executable. This stealthy maneuver exploits the executable's reliance on system search paths, effectively bypassing the intended security measures based on executable signing alone.

While users diligently verify the integrity of their executables, often utilizing User Account Control (UAC) to elevate privileges only for signed and validated executables, the oversight lies in the unchecked DLLs. These secondary components, essential for extending an executable's functionality, lack the same level of scrutiny, creating a vulnerable entry point into the system.

This project intervenes precisely at this juncture of vulnerability. By developing a comprehensive framework for DLL signing and validation, we aim to fortify executables against such malicious DLL manipulations. Through innovative approaches like DLL injection and IAT hooking, our solution ensures that only authenticated DLLs accompany trusted executables, safeguarding system integrity from DLL-based exploits.

# Threat Models:

In this project, our primary focus is on safeguarding the load-time loading process of DLLs within the context of executable (EXE) files. Our goal is to protect against specific threat models where attackers exploit vulnerabilities in the DLL loading process to compromise system integrity.

An ongoing example of this threat is prevalent in installers. Installers play a crucial role in software deployment, ensuring that applications are correctly installed on users' systems. However, they also pose significant security risks, particularly concerning the handling of DLLs. Installers typically request User Account Control (UAC) elevation from users to install software, assuming administrative privileges for the installation process. They trust all DLLs in the installation directory or system path. DLLs loaded during installation inherit these elevated privileges, potentially allowing malicious DLLs to execute privileged operations. Such attacks can lead to system compromise, unauthorized access to sensitive information, or manipulation of system settings, leveraging the installer's elevated permissions.

**Threat Model 1: Substituting Safe DLL in the EXE's Directory with rogue one**

* **Scenario Overview:** Attackers gain access to a directory containing a signed executable and DLLs it uses, and substitute legitimate DLLs with malicious ones. When the trusted signed executable is activated, it loads the substituted malicious DLLs, allowing attackers to exploit the executable's trust and execute unauthorized code.
* **Details:** When the trusted signed executable is activated, it loads the substituted malicious DLLs, which the system does not initially verify. By replacing DLLs with rogue counterparts, attackers exploit the assumption that all files within the directory are trustworthy due to the executable’s validation.

**Threat Model 2: Placing Rogue DLLs Earlier in the Loading Path**

* **Scenario Overview:** Attackers place a rogue DLL with the same name as a legitimate DLL earlier in the system's DLL search path. As the executable attempts to load the DLL, it inadvertently loads the rogue version instead of the intended legitimate DLL.  
  This scenario exploits the system's DLL search order, where the first DLL matching the requested name is loaded. By strategically placing a rogue DLL in a directory earlier in the search path, attackers ensure that their malicious DLL is loaded before the legitimate one.
* **Details:** The DLL loading path is the sequence of directories that the system searches to locate the necessary DLLs when an executable requests them. The typical search order is:
  + The directory from which the application is loaded.
  + The system directory (usually C:\Windows\System32).
  + The 16-bit system directory.
  + The Windows directory (usually C:\Windows).
  + The current directory.
  + Directories listed in the system PATH environment variable.

By strategically placing a rogue DLL in a directory earlier in the search path, attackers ensure that their malicious DLL is loaded before the legitimate one.

# Security Assumptions

Our security assumptions are foundational to understanding the scope and limitations of our project.

* **Load-Time Loading Focus:** We are focusing solely on the run-time loading process of DLLs and not on load-time loading. Our solution is designed to validate DLLs only when they are initially loaded by the executable.
* **Trust in Verified DLLs:** We assume that the verified signed DLLs are trustworthy and non-malicious. Conversely, any signed DLLs that fail our verification process are considered malicious.
* **Private Key Security:** The private key used for signing DLLs remains secure and is not accessible to attackers.
* **Administrative Privilege Escalations:** Scenarios involving administrative privilege escalations post-installation are not addressed by our solution.
* **Executable Safety:** The executable (EXE) is assumed to be safe, properly validated, and has passed User Account Control (UAC) checks.
* **RAM Integrity:** We assume that the attacker cannot alter the RAM. Our mitigation strategies focus on preventing Time-of-Check-to-Time-of-Use (TOCTOU) attacks, ensuring that once DLLs are verified and loaded, their integrity remains intact throughout their execution.

**What is Not Covered:**

* **Verified DLLs Loading DLLs:**  Some DLLs execute code automatically when loaded, using the DllMain entry point. These can pose additional challenges due to their automatic execution. Our solution does not address scenarios where a verified DLL, once loaded, dynamically loads another library. This limitation means that while the initial run-time validation is secure, subsequent runtime loading initiated by verified DLLs remains unprotected.

With these assumptions in mind, we now turn to the implementation details, focusing on how the solution is designed and developed to achieve our security goals.

# Solution Design

Our project is structured in two main phases, each addressing a critical aspect of ensuring the integrity and security of DLLs used by trusted executables.

1. **Signing:** In this phase, we will sign the DLLs to guarantee their authenticity and integrity. We will use PKI infrastructure and Certificates for signing the DLLs.
2. **Validation:** This phase focuses on validating the signatures of dynamically loaded DLLs at runtime, ensuring that only verified DLLs are executed.  
   We will employ DLL injection combined with Import Address Table (IAT) hooking to intercept and control the DLL loading process, then loading the verified DLLs successfully.

# Solution Implementation

**Signing:**

The signing process involves creating and applying a digital signature to executable files to ensure their integrity and authenticity.   
We chose to work with Windows SDK tools for signing and certificate management. Windows SDK tools are specifically designed to integrate seamlessly with Windows operating systems and security mechanisms. Their standardization and close alignment with Windows security protocols help maintain a high level of trust and compatibility, making them more reliable and secure for managing digital signatures compared to third-party tools that may not be as well integrated with the Windows ecosystem.  
There are 3 file types we need to know for the signing process. CER – stores certificate, PVK – stores private key, and PFX (Personal Information Exchange File) - combines the private key and the code-signing certificate into a single, secure file, ready for signing. The PVK and PFX demand a password since they contain the user's private key which will be used for signing DLLs, but the Certificate has the public key only, which will be used for verification.  
  
This signing process begins with generating and self-signed root certificate and a code-signing certificate which is signed by the root certificate using Windows SDK signing tools. We chose to hash with sha-256 and sign with RSA-2048. SHA-256 provides a fixed-size, compact hash that efficiently represents data, reducing the size of the data to be signed.   
RSA-2048 offers strong encryption with a manageable key size, ensuring secure digital signatures while keeping computational overhead reasonable. The certificates contain this signature's information.

For each certificate created there is a PVK file generated that has its private key. To sign we need a PFX file (Personal Information Exchange File) which is generated by combining the code sign certificate (CER) with the private key file (PVK).  
The signtool utility is employed to sign the executable file with this PFX file, applying a timestamp to maintain the validity of the signature even if the certificate expires.   
It hashes the EXEs data with SHA-256 and encrypts the hash with the private key. The certificate (which includes the public key and the issuer’s details) is appended to the EXE as a separate file alongside the data.   
Finally, the own created root certificate is imported into the Authorized Certificates Store to establish root of trust. This ensures that the executable file is properly signed and recognized as valid by the system.