SAFE LOADER

Roey Gross, Yair Lasri
Instructor: Arie Haenel

2024 – 2nd Sem.

Jerusalem College Of Technology (JCT)
Computer Science

Table of contents

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: DLL Hijacking - 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 (Reflective Loading).

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.

DLL Injection

DLL injection is a technique used to execute arbitrary code in the address space of another process by forcing it to load a DLL. This method allows a program to manipulate the behavior of another process, making it a powerful tool debugging, monitoring, and extending the functionality of existing applications. We will use this technique as a base for executing IAT Hooking later. The DLL injection is done by these steps:

1. Find the target process ID by its name
2. Open the target process with the necessary permissions to allow memory allocation and code execution.

3. Allocate memory in the target process's address space to store the path of the DLL, then write the path in the allocated memory.
4. Obtain the address of the LoadLibraryA function from KERNEL32.dll, which is responsible for loading DLLs.
5. Create a remote thread in the target process that starts executing LoadLibraryA with the DLL path as its argument, causing the target process to load the specified DLL.
6. After the DLL is loaded, we can execute the DllMain function. The DllMain is responsible for the next phase – the IAT Hooking.

IAT Hooking

The IAT, Import Address Table, is a table used by Windows applications to manage function calls to external libraries (DLLs). When a program is compiled and linked, the linker generates an Import Table (which includes the IAT) in the executable file. This table lists the DLLs the executable will use and the functions it will call from those DLLs. When the executable is loaded into memory by the Windows loader, the loader reads the Import Table and populates the IAT with the addresses of the imported functions. This is done by resolving the function addresses from the DLLs at runtime.

This allows the program to call functions in the DLL by looking up their addresses in the IAT. By altering these addresses, we can intercept and modify the behavior of specific function calls, allowing us to monitor, modify, or extend the functionality of the target application without changing its source code.

In our case, we will change the address of the LoadLibraryA function inside the IAT, to our function, that will verify the DLL before really loading it. When the EXE will try to load a DLL via LoadLibraryA, it will access the IAT at the KERNEL32.dll entry, and find the place of address of LoadLibraryA function – but we changed this place to point our function, so our function will be called instead of LoadLibraryA.

The IAT Hooking is done by these steps:

1. Locate the target process and its base address in memory.
2. Parse the PE (Portable Executable (EXE)) and find the Import Address Table within the structure of the target process.
3. Search the IAT for the entries corresponding to the functions to be hooked.
4. Store the original function addresses for later use.
5. Create a replacement function that will be called instead of the original function.
6. Modify the memory protection of the IAT to allow writing.
7. Replace the address of the original function in the IAT with the address of the replacement function.
8. Restore the original memory protection of the IAT.
9. Implement the replacement function to perform desired actions (DLL Safe Loading).

IAT Hooking modifies the Import Address Table (IAT) in to intercept and alter function calls to DLLs. By redirecting function addresses in the IAT to custom implementations, we can control and extend application behavior. For example, replacing the `LoadLibraryA` function address with a custom function allows us to verify DLLs before loading them.

Verification

Before loading and executing DllMain, and using the DLLs functions, we need to check whether the DLL is verified or not. By loading the DLL `AS DATAFILE`, the DLL's entry-point function (`DllMain`) is not executed, and the functions addresses aren't resolved, therefore the application can't use the DLL's functionality as intended. It prevents any initialization code or other functions from running, thus mitigating the risk of running malicious code from an improperly loaded or hijacked DLL.

We receive an HMODULE to the DLL, copy it to a file with auto unique name, and verify its signature. If the signature isn't verified, or doesn't exist, we free the DLLs virtual memory and close the process. If the signature succeeds, we change the DLL which we loaded to memory as datafile, to be executable. Its entry-point function (DllMain) is executed the function addresses are resolved, and the application can use the DLL's functionality as intended. This way the DLL is executable only if its verified, preventing rogue DLL attacks.

Loading

If the DLL is successfully verified, the project takes several important steps to manually load and manage the DLL. New memory is allocated for the DLL in the target process, separate from its normal loading location, because the necessary memory protections and address space mappings for executing code were not established (because we loaded it as data file). The Loading is done by these steps**:**

1. Section Copying: The DLL's sections (code, data, etc.) are copied into the newly allocated memory.
2. Relocation: The DLL's base address is relocated to match its new location in memory.
3. Import Resolution: The DLL's import table is manually resolved, linking it to other required libraries.
4. Export Processing: The DLL's export table is processed, making its functions available for use.
5. DllMain Execution: The DLL's entry point (DllMain) is called with DLL_PROCESS_ATTACH to initialize the DLL.
6. Tracking: A separate thread is started (using the function TrackAndFreeDLLs) to track the usage of the manually loaded DLL. It periodically checks if the loaded DLL is still in use. If a DLL is found to be no longer in use, the thread performs cleanup: It calls the DLL's DllMain function with DLL_PROCESS_DETACH. It then frees the memory allocated for the DLL using VirtualFree.
7. GetProcAddress resolve: Because the manually loaded DLL is not in the standard locations where Windows would normally look for it, we hook also the GetProcAddress function, replace it with a custom GetExportedFunction. When a program tries to get the address of a function from the manually loaded DLL, GetExportedFunction is called instead of the original GetProcAddress. It searches the export table of the manually loaded DLL for the requested function and, if found, returns the correct address of the function within the manually loaded DLL. This approach allows the program to correctly find and use functions from the manually loaded DLL, maintaining functionality while operating within this custom loading system.
   (This hooking is done at the same time as LoadLib hook. Didactically, it is more understandable here).

These mechanisms work together to provide a high degree of control over the loaded DLLs, allowing for custom security measures, resource management, and function access control, all while maintaining the functionality of the loaded libraries.

Future Work and Improvement

Future development efforts will focus on addressing the current limitations related to the automatic execution of code by DLLs through their DllMain entry point and the subsequent dynamic loading of additional libraries by verified DLLs.

Specifically, we aim to enhance our solution to monitor and validate any DLLs that are dynamically loaded by a verified DLL after the initial load. This will involve implementing runtime checks and validation mechanisms to ensure that any libraries loaded at runtime by a verified DLL also undergo rigorous security verification.

By extending our validation process to cover these scenarios, we will close the security gap, ensuring that the entire chain of loaded DLLs maintains the same level of trust and integrity as the initially verified DLL. This comprehensive approach will significantly bolster the security of our system, preventing potential exploitation through dynamically loaded libraries.

In future work, we aim to eliminate the need for a blocking function in the executable and implement an automatic process finder. Instead of relying on the executable to be in a blocked state, we will develop a mechanism that continuously monitors the system for the creation of the target process. This can be achieved by employing busy waiting techniques, where our injector program repeatedly checks for the existence of the target process in a loop. As soon as the target process is detected, our injector will immediately inject the custom DLL into the newly created process, ensuring that the LoadLibrary function is hooked from the very beginning of the process lifecycle. This approach will provide a more seamless and automated experience, as the injection will occur instantaneously upon the creation of the target process, without requiring any manual intervention or blocking mechanisms in the executable itself. By implementing this automatic process finder, we can streamline the injection process and make it more efficient and user-friendly.

Choices Made

At first, for the initial proposal we thought about creating our own CA to be a root of trust. The CA will be Client-Server framework with secure communication. The server will store data on signed DLLs and their public keys. The client will generate and sign DLLs with the private key. The signed DLL and public key will be sent to the server, which will store this information. For Loading we offered IAT Hooking.

Thankfully our instructor, Prof. Arie Haenel, read our proposal and pointed us to implement a signer program, without a server. We learned how windows files are signed and how the root of trust works, and chose to use this mechanism in our project. He also told us about the option of loading a DLL as data file, which really helped us. Then we chatted about the project with our Operating Systems Prof. Barak Gonen, and he led us to reflective DLL injection sources.
With Prof. Haenel's guidance, we were able to refine our approach and enhance our project's security mechanisms. His insights on implementing a signer program and the option of loading a DLL as a data file were invaluable, enabling us to develop a more robust and efficient solution!

User Guide

Signing

To sign your own DLLs, you need to create an authorized certificate first.

Download windows SDK Signing Tools for Desktop Apps, run cmd as administrator, execute this and fill <YOUR_PASSWORD> with your password and <FILE_PATH> with the DLL's path:

```
cd C:\Program Files (x86)\Windows Kits\10\bin\10.0.19041.0\x64
```

```
makecert -sky signature -r -n "CN=RootCert" -pe -a sha256 -len 2048 -
ss MY -cy authority -sv RootCert.pvk RootCert.cer
```

```
makecert -pe -n "CN=CodeSignCert" -ss my -sr LocalMachine -a sha256 -
sky signature -cy end -ic RootCert.cer -iv RootCert.pvk -sv
CodeSignCert.pvk CodeSignCert.cer
```

```
pvk2pfx -pvk CodeSignCert.pvk -spc CodeSignCert.cer -pfx
CodeSignCert.pfx -po <YOUR_PASSWORD>
```

```
signtool sign /f CodeSignCert.pfx /p <YOUR_PASSWORD> /t
http://timestamp.digicert.com <FILE_PATH>
```

Then import this unauthorized certificate to the authorized certificates file by:

WIN+R, MMC, FILE, ADD SNAP-IN, ADD CERTIFICATES, Right Click on Authorized CAs, All Tasks, import, choose RootCert.cer.

Now the file is signed by the authorized certificate!

Running

Put your signed DLLs in the EXE's folder alongside `Injected.dll`. Run your exe. Run cmd as administrator and execute `Injection.exe <Insert your EXE name>`. Now you can run your EXE safely. If an unverified DLL has been loaded, it will be detected and not used.

Work organization

Throughout this project, we investigated various attacks and began developing our solution. Yair was responsible for the DLL Injection and IAT Hooking part as well as the reflective DLL loader. Roey was responsible on the signing part and verification of the DLLs. Once both parts were completed, they have been integrated into a completed project containing verification and hooking functionality. After debugging the project, our solution was successfully finalized. Lastly, we wrote this report and prepared a presentation.

Bibliography

Reflective DLL Injection:

Explanation:

https://www.ired.team/offensive-security/code-injection-process-injection/reflective-dll-injection#implementing-reflective-dll-injection

https://otterhacker.github.io/Malware/Reflective%20DLL%20injection.html

https://www.youtube.com/watch?v=jg0CmrwEcNs&ab_channel=RITSEC

https://www.digitalwhisper.co.il/files/Zines/0x76/DW118-2-ReflectiveDLLInjection.pdf
https://www.digitalwhisper.co.il/files/Zines/0x0D/DW13-1-CodeInjection.pdf

Implementation – Stephen Fewer:
https://github.com/stephenfewer/ReflectiveDLLInjection/tree/master

IAT Hooking:

Explanation:

https://digitalwhisper.co.il/files/Zines/0x12/DW18-3-IAT_Hooking.pdf

https://data.cyber.org.il/os/os_book.pdf, Pages 227-239.

Implementation:

https://github.com/roeygross/Remote_IAT_Hooking

Dll Injection:

Explanation:

https://data.cyber.org.il/os/os_book.pdf, Pages 172 – 189

Signing:
https://stackoverflow.com/questions/252226/signing-a-windows-exe-file

https://knowledge.digicert.com/solution/how-to-import-intermediate-and-root-certificates-using-mmc

https://www.digitalwhisper.co.il/files/Zines/0x7A/DW122-2-X.509.pdf

https://www.mercurymagazines.com/pdf/072_WP_CodeSigningBestPracticeGuide_1013%5B1%5D.pdf

https://learn.microsoft.com/en-us/azure/vpn-gateway/vpn-gateway-certificates-point-to-site-makecert

Verification:

https://learn.microsoft.com/he-il/windows/win32/api/wintrust/nf-wintrust-winverifytrust?redirectedfrom=MSDN

https://www.youtube.com/watch?v=pjRapiIXi-U&ab_channel=NielsenNetworking

https://msrc.microsoft.com/update-guide/vulnerability/CVE-2013-3900

DLL Hijacking:

https://book.hacktricks.xyz/windows-hardening/windows-local-privilege-escalation/dll-hijacking

https://attack.mitre.org/techniques/T1574/001/

https://support.microsoft.com/en-us/topic/secure-loading-of-libraries-to-prevent-dll-preloading-attacks-d41303ec-0748-9211-f317-2edc819682e1