# Keyloggers - DLL Injection based

## What is a keylogger?

A keylogger is a tool that is used to record a user's keystrokes, most of the time without the consent of the user. They are used to capture sensitive information like passwords, payment details, chat logs, etc.

Keyloggers are one of the most dangerous threats due to their ability to operate silently while capturing user input. Unlike more conspicuous malware like ransomware or worms that generally cause immediate system disruption, keyloggers are silent, often remaining undetected for extended periods whilst collecting data. This nature makes them particularly dangerous for both individual users and enterprise environments, as they can compromise credentials, financial information, and confidential communications without triggering standard security alerts. They present many security challenges, consisting of bypassing traditional antivirus protections, hijacking legitimate system processes to hide their activity, and utilising fileless execution methods

## DLL Injection

Before understanding what DLL injection is and how it works, it's important to understand what a Dynamic Link Library (DLL) is. A DLL is a type of file that contains code and data which can be used by multiple programs simultaneously. Rather than each app having its own copy of a function, they can call functions stored in a DLL. For example, imagine a hotel's laundry service. Each room could have its own washing machines, however the hotel has one central operation. The whole idea behind using DLLs is about sharing common functions across programs so that you save memory since they are loaded into memory once and then shared, redundant code is reduced since you won't need multiple copies of the same functions, and that it is easier for developers to update since they won't need to recompile the whole app.

There are two main ways programs use DLLs, either through implicit or explicit linking. With implicit linking, the program is compiled knowing it will use a specific DLL so the OS loads the DLL automatically when the program starts. The downside to this is that if the DLL is missing, then the program will not run. Whereas with explicit linking, it allows the program to load a DLL when it is needed, rather than at startup. Whilst it is more flexible, it is generally more difficult to implement.

DLL injection is a technique where you force a running process to load and execute malicious code from a DLL it wasn't originally meant to use. Once it has been injected, it runs inside the

program's memory so it can see what the program sees, record its activity, etc. If you have an app running, it will only load the DLLs that it requires in order to function, however with injection, there will be another program that forces the target app to load a different DLL.

# Use Cases

## Legitimate

**Security Software**

Security software like your antivirus will utilise DLL injection for monitoring and protection purposes. Endpoint detection and response (EDR) systems like CrowdStrike Falcon inject monitoring DLLs into processes to detect malicious activity in real time. By doing this, they can hook network communication functions so that they can monitor for data transfers or even newly created processes. These implementations will typically sign their DLLs with trusted certificates and maintain boundaries in order to avoid disrupting host processes.

**Game Mods**

In almost all games, you will see ethical use of DLL injection in order to create mods. There are graphical mods e.g. Reshade which injects a DLL into the game process to intercept DirectX calls which allows them to add things like post processing effects like ambient lighting that were not in the original game. There are also multiplayer game server emulators which bypass official servers and enable private modded gameplay. While these uses are often tolerated in single player contexts, multiplayer game developers are working on anti cheat systems like EasyAntiCheat and BattlEye that detect and block unauthorized DLL injections.

## Malicious

**Surveillance**

Some spyware will implement keylogging and screen capture capabilities by hijacking input handling functions. For example FormBook which is considered an infostealer will inject into browser processes in order to intercept form submissions and saved passwords before encryption occurs. They do this by hooking functions that handle database operations or intercepting APIs. These are dangerous because they operate at such a low level that even encrypted communication and multi factor authentication can be bypassed once compromised.

**Game Cheats**

Game cheats can use injection to modify in memory structures that store player coordinates or health values, which creates unfair advantages in competitive multiplayer games. For example, a common cheat is a wallhack, where the DLL hooks into the game's rendering functions and modifies how objects are drawn so that players can see enemies through walls. Another example is an aimbot, where the DLL accesses player position data from the game's memory and automatically adjusts the aim to target enemies. Since these are running inside the game, it gives them the same level of access as the original game code.

# How DLL Injection works in a nutshell

1. The injector will open the target process
2. Allocate memory inside the process
3. Write the path to DLL into process
4. Load the DLL with a remote thread
5. Done, the DLLMain will run

# Evasion Methods

**Alternate APIs**

Avoiding well known APIs that are heavily monitored by security solutions is common. For example, many keyloggers rely on SetWindowsHookEx or GetAsyncKeyState, which are often flagged. To bypass this, keyloggers can use less obvious or lower level APIs, or even undocumented system calls to capture input. Some may directly read from device drivers which intercepts raw keyboard inputs at the kernel level, bypassing higher level user mode logging APIs entirely. This reduces the chance of detection by behavioral monitors.

**Encryption**

One of the most common evasion methods is to muddy up the code. Instead of storing strings like 'keylog' or 'SetWindowsHookEx' in plaintext, attackers encrypt or encode them. At runtime, the keylogger decrypts them before execution. This prevents signature based tools from identifying the binary through known strings or byte patterns.

**Polymorphic and Metamorphic Code**

Polymorphic code changes its appearance while retaining its original functionality. Imagine a disguise, the person underneath is still the same, but their outfit and hair are constantly changing. They will utilise encryption techniques that transform the binary signature whilst maintaining the same core functionality. They contain a polymorphic engine. After execution, the engine alters the decryption function and potentially other non functional parts of the keylogger's code which generates a new, unique signature for each payload, ensuring that no two infections share identical signatures. This makes it difficult for traditional antivirus software as they generally rely on signature. Even if one variant is identified and blacklisted, the next variant will appear as a completely new one. The encryption layers will typically incorporate environmental keying, where decryption functions will only activate under specific system conditions (e.g correct OS version, absence of debugger or sandbox). More advanced versions may implement multi stage decryption with filler code insertion and instruction reordering, where each stage of the decryption reveals part of the payload whilst simultaneously removing the previous decryption stub. The Storm Worm used polymorphic techniques to mutate its binary on each infection, making blacklist based detection nearly useless.

Metamorphic keyloggers will not only change its appearance, it will also rewrite its code each time it runs. They can replace an instruction with an equivalent (1 + 1 is the same as 4 - 2). The flow of execution can be changed without altering the inherent logic, for example you could add unnecessary jumps. They may also rename the registers.

**Fileless**

Generally, traditional malware uses an executable file on disk, which makes it easier to detect via file system monitoring. Fileless keyloggers avoid this by existing only in memory and are often delivered through Powershell. Being fileless, it allows keyloggers to inject themselves directly into a running process without touching the disk, avoiding antivirus scans, file integrity checks, and system snapshots.

**Process Hollowing**

It creates a suspended instance of a legitimate process, unmaps its memory, and then replaces it with the malicious code which allows it to appear as a trusted process. With this, it also maintains its process relationships and inherits the security context of the hijacked process. It can continue running indefinitely and can also be made to persist after reboot if linked with other techniques.

# Methods of detection

**API Monitoring**

Keyloggers will generally use Windows API functions in order to intercept keystrokes. Microsoft Defender uses user mode hooking to monitor calls to SetWindowsHookEx and GetAsyncKeyState which are commonly abused by malware. Security tools like Microsoft Defender and EDRs (Endpoint Detection and Response systems) will monitor user mode and kernel mode calls to these functions in order to identify suspicious behavior.

In legitimate applications like Notepad or accessibility tools, keyboard hooks are expected. However, if an unusual process such as clock.exe or a randomly named binary calls SetWindowsHookEx, it raises flags. Establishing what is considered normal behaviour helps security tools learn what normal API usage looks like for each process, and anything deviating is flagged.

In some cases, they can trace the callback functions used in the hook. If the hook callback is pointing to an unknown memory region through dynamically allocated memory, it can signal malicious activity as most legitimate hooks reside in properly loaded modules.

**Network Traffic Monitoring**

Generally, keyloggers will transfer data back to the attacker silently in the background. The logs could be sent in small batches using HTTP POST requests in order to avoid detection and reduce memory footprint. They are typically just a few bytes, frequent, sent to unusual domains and may lack typical browser headers. Some keyloggers avoid HTTP and use DNS tunneling instead to bypass firewalls. Most networks allow DNS traffic, making it a more stealthy channel of transfer as opposed to HTTP.

**Signature Based Detection**

This method uses databases of known malware signatures including hashes, strings and byte patterns to detect malicious files. Anti virus software will compare files, memory segments, and behaviors to these known signatures. While this method is effective against older and unchanged keyloggers, it is now easily bypassed by polymorphic and metamorphic malware that jumble their code each time it is run.

**Machine Learning**

Machine learning models, often trained on behavioral datasets, can detect anomalies in process execution and typical user inputs. Unlike signature detection, ML doesn't require predefined rules. Instead, it generalises patterns across malware families. Keyloggers can be detected by ML models based on typing cadence since humans have irregular typing delays, while automated keyloggers generate input at consistent intervals. Another pattern would be through process behavior as it is abnormal for explorer.exe to spawn injector.exe, which then runs powershell.exe. ML models learn such behavior trees and will flag deviations from typical

activity. Sudden unexplained memory usage or abnormal API usage patterns e.g. excessive calls to GetAsyncKeyState may raise alerts.

**Memory Analysis**

Keyloggers will generally leave traces in memory or in the behaviour or processes. Filesless keyloggers will inject into legitimate processes and avoid writing any files to disk in order to bypass signature detection and hook into input APIs. Of these traces in memory, there may be missing digital signatures, relationships that don't make sense i.e. notepad.exe spawned by powershell.exe, or unknown process names. They will also look out for DLLs that are loaded into a process. With this anything loaded from suspicious paths or is missing their on disk counterpart will be flagged.

**Honeytokens**

Honeytokens involve planting decoy data like fake credentials, fake documents, or intentionally placed text input. For example, a user might create a file named "passwords.txt" on the desktop containing dummy credentials. If those credentials are used in an authentication attempt or observed leaving the network, it serves as a strong indicator that a keylogger is active on the system. Similarly, typing fake credentials into login forms can serve as bait. This approach doesn't stop keyloggers directly but provides an alert system for detecting ongoing attacks.

**Rule Based Detection**

This identifies malicious behavior by looking for suspicious patterns or combinations of actions that are typical of keyloggers rather than relying on exact code signatures. For instance, a process that uses SetWindowsHookEx to hook into keyboard input but is missing a visible user interface is highly suspicious as this API is used to intercept keystrokes. Similarly, if a process is using GetAsyncKeyState to log input and also opens network connections it would raise flags. These kinds of behavior based rules allow security tools to catch previously unknown or polymorphic keyloggers that evade traditional signature based scanning by slightly altering their code.

# Reflection

I have come to appreciate the technical aspect of these tools and the mitigation strategies of such. In my opinion, I think they fall among some of the most invasive and dangerous forms of malware simply because of how stealthy they can be and the resulting magnitude of the potential damage can be catastrophic if not dealt with quickly. A well built and placed keylogger can compromise your personal identity, banking information and even access pretty much anything the target can.

Back in high school, like many others who didn't want to pay full price for a popular AAA game, I downloaded a cracked version from a sketchy torrent site. It worked. Or so I thought. A few days later, I noticed something strange. My email was accessed from an unknown location. Then my game accounts started getting locked. Then, a week later, my PayPal account flagged an unauthorised purchase. That's when I realised something was wrong. I ran a scan and found the culprit. It was some kind of keylogger, deeply embedded into my system. It had been quietly logging everything I typed from passwords, chats, searches since the moment I launched that pirated installer. It was humbling and terrifying. I wiped my PC and changed every password I could remember, and spent the next few weeks living with the unease of 'what else did they get?'

Nowadays, I'm much more careful when it comes to downloading files online. So I revisited keyloggers, but I wanted to learn about the inner workings and the different methods of evasion, to which I stumbled upon DLL injection.

My infection happened not because the malware was good, but because I was careless. I disabled my antivirus, trusted a shady website, and ran unknown code with administrative privileges. Social engineering and unsafe behavior is the reality of most infections today, not the style of hacking you see in movies.

Working on this keylogger made me think about the ethics behind it. It was just a learning exercise on the surface and that I wasn't doing anything malicious with it. However knowing that this could very easily be reproduced by someone with different intentions. One of the main ethical concerns is violating ones privacy. Since keystrokes reveal everything from passwords to messages etc, they can be easily abused. Now even in cases where they are used for legitimate reasons like parental control or company monitoring, there is a line between surveillance and transparency that is quite blurry.

Once it has been written and published, even a keylogger built for educational purposes can be weaponised quickly. If I was to upload it to Github as a tool to educate others, it could just as easily be used as a malicious payload in a phishing attack. This raises the question on whether it is more harmful to publish the keylogger even if it was for educational purposes.

I decided to use C++ for this since Windows OS and its APIs are primarily written in C and C++ so when interacting with the APIs, it is in its native language without the need for wrappers or runtime environments. In addition, you can directly include Windows API headers, so by linking these libraries, you don't need managed runtimes or wrappers.

My injected keylogger didn't show up in the Task Manager. There was no new process name, no flashy GUI window, no suspicious pop up asking for administrative privileges. To the average user, nothing at all had changed. All I had done was hijack an existing trusted process. The logging system was also quite subtle. I had it write the keystrokes to a file in C:\Windows\Temp which is already cluttered with temporary files and logs.

It was interesting to see that at this point for the user, nothing seemed wrong or broken. Everything still responded as normal, there weren't any crashes and the computer was still performing as usual.

DLL injection can be considered a dual use technology. They can be used by security researchers to test their defenses, or to create malware. The same line of code that strengthens a red team penetration test can also power spyware software.

Building a keylogger was actually a really fun technical exercise. It taught me about DLL injection, memory management, API hooking, and stealth operations.

In terms of the actual keylogger itself, it was a difficult project since I had worked with C++ before, however I had never used the Windows API. With this, I initially thought it wouldn't be too difficult to understand how to use, however there is minimal documentation for edge cases.

Initially I had thought that the injector would be more difficult, however it turned out that it was much more complicated than the injection. Major issue that I had encountered was how it handled special keys and chars like (shift, ctrl, !, #, etc). Using the hook WH_KEYBOARD_LL, it actually intercepts key events rather than actual characters, so when you press something like 'Shift + 1', the logger won't log '!', rather it will see VK_SHIFT and VK_2 being pressed. In order for me to properly log the intended character, I had to account for the current keyboard layout as well as the state of the modifier keys such as shift, ctrl etc. Another extending issue was that if someone was using a different keyboard layout e.g. French, since the layout is different, it would interpret VK_Q as VK_A

Debugging was also something I looked past initially and didn't think it would take as long as it did. Traditional debugging tools and methods would not work since it is working with memory spaces of processes rather than developing a calculator app. I did initially try using breakpoints, however this would often crash or freeze the target process. I ended up using printf logs and occasionally would use messagebox popups. With DLL injection, there are many reasons why it fails, silently at that so it was always unclear why it was failing. Some include cases where the target process doesn't allow you to run WriteProcessMemory or CreateRemoteThread, or Windows Defender silently blocking without any notification. I wasted many hours as a result of this, since at first I assumed that it was due to a bug in my code, many of which were. So I learned to log return values of all system calls as well as calling GetLastError.
Even after successful injection, installing a keyboard hook like SetWindowsHookExW had its own issues. For example, if the target process didn't have a message loop, the hook would never fire. If the DLL didn't export the proper hook function correctly, or if the calling convention was wrong, the hook silently failed. If Windows Defender flagged the hook, the whole injection was blocked. I had to ensure the process I was injecting into was GUI based and ran a proper message loop. Eventually, I used notepad.exe for testing because it was stable and simple, although it crashed repeatedly in the process.

I learned the hard way that once the DLL is loaded, it will receive DLL_PROCESS_ATTACH, and if I spawned new threads or called functions that resulted in LoadLibrary or other locking

calls from within DllMain, I risked a deadlock. That's when I found DisableThreadLibraryCalls, which prevents the loader from calling DLL_THREAD_ATTACH/DETACH and makes your DLL more stable inside foreign processes. Without this, I crashed Notepad over a dozen times just by trying to log keystrokes or spawn a logging thread.

This project challenged me both technically and ethically. Writing the keylogger pushed me to understand how the Windows API works under the hood, how keystrokes can be intercepted, and how system level operations are handled. It also forced me to think about the risks that come with this knowledge. If this took me about a month from scratch to understand and build, imagine a professional who has extensive knowledge and experience. Having been a victim of a keylogger in the past, I understand how it feels to have your privacy violated. To an extent, I approached this project not with the intent to simply build something malicious, but to understand the inner workings of these threats so that I can better protect myself against them.