A SLAHP in the face of DLL Search Order Hijacking

Antonin Verdier, Romain Laborde, Mohamed Ali Kandi, Abdelmalek Benzekri

A SLAHP in the face of DLL Search Order Hijacking

Antonin Verdier, Romain Laborde, Mohamed-Ali Kandi, and Abdelmalek Benzekri

IRIT, Université de Toulouse, CNRS, Toulouse INP, UT3, Toulouse, France
{antonin.verdier, romain.laborde, Mohamed-Ali.Kandi, abdelmalek.benzekri}@irit.fr

Abstract. DLL Search Order Hijacking (also known as DLL Hijacking or DLL planting) is a problem that is generally overlooked by software developers even though its existence has been known for over a decade. While Microsoft has designed and implemented mitigations to reduce the feasibility and the impact of DLL Search Order Hijacking, this issue is worth being brought back up due to the recent adoption of user-writable directories as potential, and sometimes default, software installation paths (in lieu of directories like "Program Files" which require administration privileges by default) in order to improve installation success rates. We conducted a study on 48 different software programs (Top software on Sourceforge across 4 different categories and the 4 major web browsers) and found that more than 88% of them were vulnerable to some form of DLL Search Order Hijacking. To alleviate this issue, we propose SLAHP, a novel way of preventing DLL Search Order Hijacking exploitation in the form of a proof-of-concept implementation that is both easy to integrate with new and existing products by software developers and users. It is invisible to end users while still allowing the usage of previously insecure installation locations. To further demonstrate the usability of our solution, we conducted performance tests and found that its impact is mostly negligible.

Keywords: DLL hijacking · DLL sideloading · Shared libraries

1 Introduction

The principle of DLL Search Order Hijacking (DSOH) is to confuse the Windows OS into loading a malicious DLL (Dynamic-Link Library) file instead of a genuine one by taking advantage of the way Windows locates DLL files. This attack enables attackers to hide malicious behaviour behind genuine applications by making trusted processes execute their code. While its existence has been known for over a decade [17, 18] and is well documented by reputable sources [15], it still is regularly used by threat actors [3, 6] as a mean to obtain persistence or to perform privilege escalation. Indeed, up to 20% of criminal and APT attacks in 2019 [2] used DSOH. In addition, as of 2021, no EDR was able to detect the

attack mechanism itself [9]. Back in 2003, Microsoft added a mitigation that fixed most DSOH vulnerabilities, this mitigation was effective mainly because most software programs were installed in the `C:\Program Files` directory, which requires administrative privileges to be written into. However, this hypothesis does not hold anymore and the past decade has seen more and more software developers choosing different installation directories like `AppData` to facilitate the installation process. This directory, being located in the user directory, is writable by an unprivileged user. This enhances the user experience and the installation success rate. However, it also simplifies the task of attackers to place their malicious DLL in the targeted software installation directory. Furthermore, DSOH's underlying issue can also be exploited by attackers in order to gain initial access to a system: this is known as DLL Sideloading [11,16], which is sometimes considered to be a technique of its own. While DLL Sideloading isn't the main topic of this research, its similarity with DSOH makes it a secondary target.

In this article, we present SLAHP (for Shared Library Anti Hijack Protector), a solution against DSOH and similar techniques that is easy to integrate with new and already existing software as well as unnoticeable by end-users. These characteristics should facilitate the adoption of our solution by both developers and end-users, as well as allow advanced users to protect programs without any intervention from the original software developer. Our solution can be added as a static library during development or in the form of a custom launcher after compilation and/or distribution.

This paper is organised as follows. In section II, we explain how the way the Windows operating system performs DLL loading can be exploited by threat actors. Section III contains the results of a vulnerability survey we conducted on how widespread the issue of DSOH is. Then, we present our solution and evaluate its performance in concordance with our objectives. Next, we discuss related works before finally providing a summary of our contributions and discussing ideas to overcome our solution's limits in future works.

2 What is DLL Search Order Hijacking?

In this section, we introduce the Windows DLL loading process exploitation and the current mitigations available.

2.1 Exploitation

DLLs are Windows shared libraries that can be referenced by name only. To accomplish this, the Windows OS employs a search order that specifies the list of directories it should sequentially inspect for libraries. It starts with the libraries listed in the `KnownDLLs` registry key, followed by the Application Installation directory, the System directories (e.g., `C:\Windows\System32`), the current working directory (CWD) and finally the directories listed in the `PATH` variable.

However, the simplicity of the DLL search order mechanism has its flaws. If we consider the hypothetical case where a threat actor is able to place an

arbitrary file in an application's installation directory, the application could end up loading and executing malicious code against its will. This is because the Windows OS only looks at file names to locate DLLs. Thus, a malicious DLL file with the same name as the genuine system file could be loaded in lieu of the real one. This exploitation process is illustrated in Figure 1.

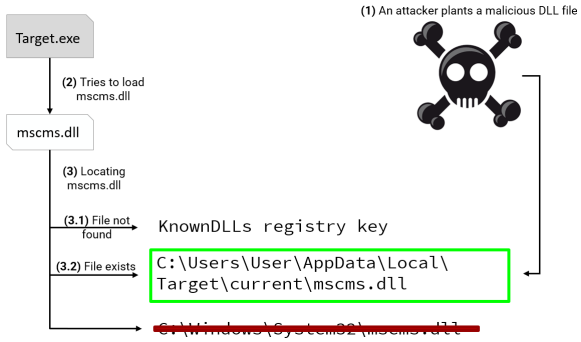

Fig. 1. Diagram of a search-order-hijacked DLL loading

However, such an attack requires two conditions: i) the threat actor must be able to write to the application's installation directory and ii) the malicious DLL must either be optional to the target program or be able to emulate the features of the usurped DLL, which is known as a proxy DLL. As mentioned earlier, the first condition is becoming more and more prevalent as applications seek out ease of installation by choosing directories that can be written to without administrative privileges as their installation location; while some software such as Google Chrome or Mozilla Firefox use these directories as a failsafe, more and more software — mostly observed amongst software using the Electron and Chromium Embedded Framework (CEF) — use these directories as their default installation location, sometimes without user consent. As for the second condition, it is only a matter of time and expertise for the threat actor to meet it.

2.2 Mitigations proposed by the Windows OS

DllSearchMode Windows 2000 SP3 [13] introduced an optional feature called `DllSearchMode` that moved the CWD further down the DLL search order (it was initially inspected before the system directories). This mitigation has been enabled by default since Windows XP SP2 in August 2004 and reduced the potential of DSOH exploitation, as software programs were generally installed in the `C:\Program Files` directory, which requires administrative privileges to write to.

PreferSystem32 As part of the 2017 Windows 10 Creators Update, Microsoft introduced a similar mitigation called `PreferSystem32`. This mitigation works by inverting the application installation directory and the system directories in the DLL search order. While this approach can prevent attacks such as the one described in section 2.1, it also prevents by design a developer from loading a custom DLL that shares its name with a DLL present in the system directories. Surprisingly, the amount of documentation surrounding this mitigation is very poor and sometimes incorrect. Indeed, the Powershell documentation [21] describes the existence of a Powershell command that can be used to enable that mitigation on an executable, or system-wide. While that command exists, we found that an executable that is supposed to be protected is still vulnerable to the exploitation of DSOH vulnerabilities. The most complete piece of documentation we could find [4] is 4 sentences long. Nonetheless, we found out that the Mozilla Firefox browser uses `PreferSystem32` in order to protect itself from DSOH vulnerabilities. By reading Firefox's source code and experimenting with the OS-provided APIs, we observed that the only way we could make the mitigation work was by having a process acting as a launcher enable the mitigation, then start the program that we want to protect. In conclusion, `PreferSystem32` is just a partial mechanism to mitigate DSOH vulnerabilities. We believe that the complexity brought by its implementation is the main reason behind its observable absence in software development.

3 Survey of software currently vulnerable to DLL Search Order Hijacking

In order to better understand how widedespread the issue of DSOH is, we developed a tool that allowed us to look for DSOH vulnerabilities in applications with as little human interaction as possible. We used a sample of 48 different programs, including programs from the top education, text editors and file-sharing software on Sourceforge, as well as the four major web browsers (Google Chrome, Mozilla Firefox, Microsoft Edge & Opera).

In order to detect DSOH vulnerabilities, our tool enumerates the dependencies of the program we are analysing and tries to usurp the name of each dependency, one by one. The detected vulnerabilities can be qualified as low-hanging, as our tool is not designed to mimic any of the features of the original DLLs. It only contains code that is executed on loading and creates a temporary file whose existence constitutes proof of successful exploitation.

We have observed that some programs refuse to start when some of their dependencies at start-up do not contain the expected DLL function exports; thus, the amount of vulnerable programs we have detected may be underestimated. Another consequence is that our tests induced apparent misbehaviour (e.g. black screen) from the programs we analysed. However, we do not believe that this would be a problem for an actual threat actor, as the creation of a malicious proxy DLL is a task that can be accomplished with open-source tools [12].

The results of the vulnerability tests we conducted are summarised in Table. 1. We categorised the analyzed software according to their default installation directory. The motivation behind this partition is the fact that these installation directories largely represent the main difficulty of DSOH-based persistence attacks. Indeed, directories such as `C:\Program Files` can only be written to with administrative privileges making the task of a threat actor harder, while the `C:\%USERNAME%\AppData` directory can be written to by its owner, `%USERNAME%`. Portable applications were assigned their own category because the choice of their installation directory is up to the user. However, we can speculate that the users are more likely to install them in directories that do not require administrative privileges as it is the easiest solution.

Table 1. DSOH vulnerability survey results

Location	Vulnerable	Non-vulnerable	Total
AppData	15	3	18
Portable	6	1	7
Program Files	21	1	22
Total	42	5	47

We can see that most of the software we have tested is vulnerable to DSOH, regardless of the category. However, the programs installed into `Program Files` directory seem to exhibit the highest vulnerability proportion. It should be noted that the relatively small amount of programs we analyzed may not accurately represent the totality of computer software programs, and consequently our estimated total proportion of 89% vulnerable programs is only intended to highlight that this security issue is common for some of the most popular programs. Therefore, this justifies research in mitigating DSOH attacks.

We hypothesise that there are two reason explaining the amount of programs currently vulnerable to DSOH. We believe the first one is the general ignorance of its existence [19] by software developers and the lack of easily integrable mitigation mechanisms. The second possible reason is the nonchalant attitude adopted by some major software editors, such as Google [1] ignoring this security issue, arguing the fact that the exploitation of DSOH largely depends on a threat actor having had prior access to the targeted computer. Their reasoning is that Chrome cannot do anything against a threat actor having access to the computer. We know this rationale is not entirely correct, as Mozilla Firefox implements security features that are specifically designed to combat DSOH.

4 The Shared Library Anti Hijack Protector

4.1 General principle

Our objective is to protect software programs against DSOH. Considering the problems of DSOH unawareness and lack of attention, we have tried to make

our solution as easy to implement as possible to facilitate general adoption by developers. Furthermore, our solution is intended to be used by advanced users to protect vulnerable programs when the software editor is not interested in fixing vulnerabilities in its own software, for instance. The source code of SLAHP is available on Github[1].

Our solution is designed to filter DLL loading attempts based on a security policy. Thus, the first step consists of loading the security policy, either from a remote HTTP server or from a cached file. To prevent threat actors from tampering with the security policy configuration file, either by using man-in-the-middle network attacks or simply by modifying it during initial access, we provide the option of digitally signing it. Once the authenticity and integrity of the security policy has been verified, SLAHP will intercept every attempt to load a DLL, i.e. every absolute path from which Windows tries to load a DLL, and allow or deny it based on whether or not the DLL complies with the security policy. An overall framework is illustrated in Fig. 2.

We used the Microsoft Detours library [8] to intercept DLL loading attempts. SLAHP specifically hooks the `NtQueryAttributesFile` function, a general-purpose semi-documented function of the Windows API used to retrieve information about files; this function also allows the DLL loading system to determine whether or not a DLL file with the desired name exists at each location of the search order. While using this unspecialised function as a way to intercept DLL loading attempts is not ideal, it gives us the ability to pretend the file does not exist by modifying the return value of the function. This way, our solution "blocks" DLL loading attempts that do not conform to the defined security policy. This approach allows potentially under-attack programs to continue operating normally: if the DLL does not exist from the program's point of view, the rest of the search order is inspected until a compliant DLL is found and loaded.

When SLAHP denies the loading of a DLL, it can either automatically attempt to fetch a potentially newer security policy from the software developer's remote server or simply prompt the end-user for the action they want to take, while informing them of the potentially dangerous situation: they can ignore the issue, abort execution or try to update the security policy in case a DLL got updated to a version that was not allowed by the cached security policy. An important aspect of this choice is that none of the provided options can have a negative security impact. Hence, an inattentive or frustrated user cannot be used as a vector for acquiring higher attack capabilities.

Finally, SLAHP has been designed to be able to protect programs that employ a multi-process architecture; this is often the case with web browsers and software using the Electron Framework or the Chromium Embedded Framework. This is accomplished by using inter-process communication operating system features such as events, shared memory and mutexes.

[1] `https://github.com/lacaulac/SLAHP`

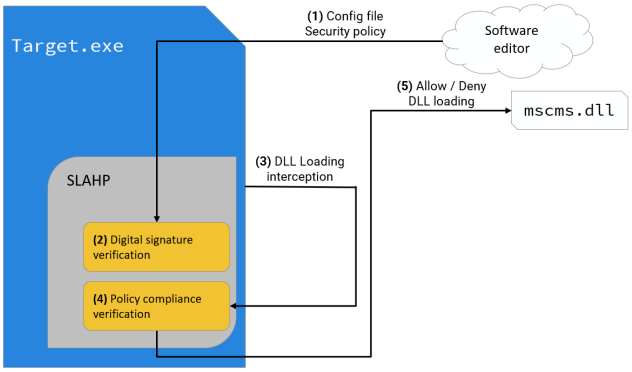

Fig. 2. Overall framework

4.2 Security policy

The security policy for a given SLAHP-protected program is specified in a configuration file named `policy.cfg`. Its optional digital signature is contained by the `policy.cfg.sig` in base64 form. An example of such a configuration file is shown in figure 3. A configuration file is divided into two parts : the protection options and a list of allowed DLLs.

Every line of the allow-list defines a mapping between the name of an allowed DLL and one or more of its versions, each represented by a hash. In the example depicted in figure 3, two versions of `test_dll.dll` are allowed while only one version of `other_dll.dll` is permitted. By default any DLL not included in this list is denied.

In addition to the list of hashes of the allowed DLL, the current version of SLAHP provides three different protection options to change the default behavior to make it easier to write SLAHP configuration files. The first option is called `allowunspecified` and allows DLLs that are not explicitly listed in the list of allowed hashes to be accepted. In the case of figure 3, `test_dll.dll` and `other_dll.dll` should still match with the specified hashes. However, if the program requires `unspecified_dll.dll`, no constraint will apply to it. This option is interesting when the hash of a DLL is not known in advance which is the case for plugins or optional third-party dependencies. The potential vulnerability introduced by enabling this option can be mitigated by using the `unspecifiedcantbeinlocaldirectory`, which prevents unknown DLLs to be loaded from the installation directory of the program, enforcing DLLs to be loaded only from files listed in the KnownDLLs registry key and files within the system directories, PATH directories and current working directory. Finally, the `signatureallowsbypass` configuration option can allow any DLL to be loaded regardless of the security policy as long as the DLL is signed using a valid code signing certificate. In the extreme case where an application and all of its de-

pendencies are signed with a valid certificate, using this option would negate the need to define DLL hashes; however, this is far from the norm.

```
allowunspecified:yes
signatureallowsbypass:yes
unspecifiedcantbeinlocaldirectory:yes
test_dll.dll:[SHA-256],[SHA-256]
other_dll.dll:[SHA-256]
ENDCONFIG
```

Fig. 3. Example config file

```
InitProtector(L"SLAHP-UserAgent", L"soft-editor.net",
    L"/path/to/policy.cfg", useHttps, ignoreHttpsErrors,
    pubKey, useCache, useMultiProcess, hideFromUser);
```

Fig. 4. Calling InitProtector

4.3 Configuration options

SLAHP also provides configuration options regarding how the security policy is obtained and the usage of certain specific behaviour options.

The first configuration options are the domain name and path to the security policy configuration file on the remote HTTP server and the desired user agent. Then, the developer / user can choose if they want SLAHP to communicate with the server using HTTPS and whether or not SLAHP should ignore HTTPS security errors (e.g. invalid X.509 chain of trust). If the security policy is signed, the issuer's public key must also be provided so that the integrity of the security policy configuration file can be verified. In order to avoid systematically relying on a remote server to obtain the security policy, which could lead to increased start-up delays as well as denial-of-service if the remote server can't be reached, the configuration file and its digital signature can optionally be stored locally.

Finally, the developer / user can also set whether or not the prompting for human decisions and the support for software using a multi-process architecture that are described in section 4.1 should be enabled.

4.4 Integration possibilities

SLAHP can integrate with new programs as a static library that can be added to the project during development (static integration) and with applications that are already compiled as a custom launcher (dynamic integration).

Static integration SLAHP is available as a static library (see Figure 5) that can be added to any C/C++ project. Once this static library has been added to the project, SLAHP can be easily initialised by calling the `InitProtector` function, which takes as parameters the configuration options detailed in section 4.3 as can be seen in Fig. 4. We recommend calling `InitProtector` as early as possible, as the filtering of DLL loading attempts only happens once SLAHP is initialised. If the developer is using the Microsoft Visual Studio compiler toolchain, using the delayed-loading [22] linking option wherever possible will help prevent DLLs from being loaded before SLAHP is initialised.

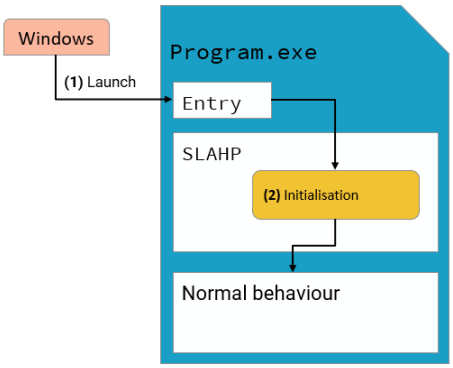

Fig. 5. Static library integration

Dynamic integration Our solution can also be used when the source code of the program is not available. We call this approach the dynamic integration, allowing virtually anyone to protect the software installed on their computer. This is also useful when the project is written in a language that does not provide any Foreign Function Interface that can work with our static library integration, or when SLAHP cannot be initialised before potentially vulnerable DLLs are loaded (e.g. packaged python software, Electron applications, etc.).

To accomplish this, the dynamic integration is separated in two components, as shown in Fig. 6. The first component is a SLAHP-protected launcher application that starts the target program in a suspended state (i.e. the program gets loaded into memory but no code execution takes place since all threads are suspended) and performs DLL Injection [20] on the target process, causing it to load the `ProtectorAgent`. Upon loading, the DLL will initialise SLAHP and resume the execution of the targeted process's thread(s).

We provide a secured launcher creator program whose task is to create a `Launcher.exe` & `ProtectorAgent.dll` file pair based on the configuration options described in section 4.3. We decided to make the launcher creator work by patching the desired configuration options into pre-compiled binaries, making the presence of a compiler toolchain unnecessary to create a Launcher and a Protector Agent. This strategy makes our solution easier to use by non-developers.

A video demonstrating the dynamic integration of SLAHP with Microsoft Teams is available for watching[2].

4.5 Solution evaluation

Security analysis Once it is initialised, SLAHP is able to successfully block any attempts of loading unauthorised DLLs. If we refer to the example described in section 2.1, simply disallowing the loading of unspecified files in the security

[2] `https://youtu.be/sb-lZN37tCg`

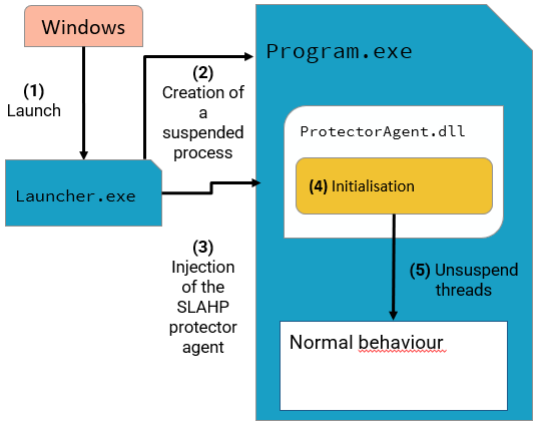

Fig. 6. Dynamic launcher-based integration

policy would mean the malicious file located in the installation folder does not comply with the policy. The DLL loading algorithm would thus ignore it and continue to inspect the next directories in the search order, where the genuine DLL should be found and loaded from.

As our solution aims at protecting software against DSOH attacks, we paid particular attention to minimise the attack surface of SLAHP. Since SLAHP depends on several libraries, we have built in security measures against DSOH. Firstly, the DLLs used by SLAHP are only loaded from system directories. We chose to trust the contents of these directories, because we are convinced that if an attacker were to gain the privileges to modify these directories, he would have access to much more advanced persistence capabilities. We also chose to load its dependencies using the delayed-loading [22] method to ensure the DLL origin restriction has been applied before any DLL can be loaded.

The integrity of the security policy is paramount, as its content dictates what DLLs can and cannot be loaded. An attacker may try to make SLAHP load an ineffective policy through Man-in-the-middle attacks when using HTTP(S) or by modifying the cached policy. However, our implementation provides an option to digitally sign the security policy and check its integrity before loading it.

Finally, regarding the integrity of SLAHP's code (i.e. the initialisation function or the launcher & protector agent library), our solution does not provide any protection against the modification of its compiled code by itself. Nonetheless, Code signature using Windows code sign certificates can overcome this issue.

Performance evaluation To assess the overall impact our solution can have on the performance of a protected software product, we considered the time taken by SLAHP to initialize and the overhead caused by the filtering of DLL loading attempts. For the initialisation time evaluation, we decided to consider only the cache-based loading, as retrieving a file from a remote server could introduce

Table 2. Performance impact - Startup time

	Unsigned security policy	Signed security policy
Time	1.01 ms	1.39 ms

Table 3. Performance impact - DLL load time

Policy situation	**DLLs, by implicit dependencies**	
	0	**6**
Unprotected (a)	0.16 ms	0.99 ms
Allow (hash-based) (b)	3.04 ms	3.4 ms
Disallow (hash-based) (c)	60.04 ms	59.32 ms
Allow (digital signature-based) (d)	11.88 ms	14.55 ms
Disallow (digital signature-based) (e)	67.58 ms	71.13 ms

delays through no fault of SLAHP. As detailed in Table. 2, the average added delay is less than 2ms, with little impact from verifying the digital signature of the security policy.

Regarding the evaluation of the efficiency of the DLL filtering mechanism, we decided to evaluate how fast DLLs were loaded in 5 different cases: (a) Without the protector, (b) Loading allowed if the DLL was compliant with the security policy because of its hash - no digital signature checks, (c) Loading blocked if the DLL was non-compliant with the security policy because of its hash - no digital signature checks, (d) Loading allowed if the DLL was compliant with the security policy because of its valid digital signature, but with an invalid hash, (e) Loading blocked if the DLL was non-compliant with the security policy because of both its hash and invalid digital signature.

An important aspect to consider in DLL loading is that any DLL can optionally depend on one or more DLL(s). In our performance tests, we only validate implicit (loaded before start) dependencies. Indeed, if a DLL needs to load other DLLs in order to complete its own loading, a higher number of dependencies will result in a longer overall load time. This is especially the case here, as each of these dependencies have to be verified by SLAHP; note that our tests always made the dependencies have a valid hash, so as not to skew the "block" results.

Table. 3 shows the average load time for each of the 5 cases applied to two DLLs: one with no implicit dependencies and one with 6 implicit dependencies. Each test case was run 500 times for each DLL. Because the DLLs used to test the performance of our solution were written specifically for this task, they're only present in the installation directory of the performance test binary. Thus if a loading is blocked, looking for the file further down the search order will not yield any conclusive results but will still take a significant amount of time. This would generally not be the case when dealing with real-world DLLs, as most of the dependencies we've observed are already either in the application installation directory or in the system directories.

While the overhead introduced by our solution is proportionally significant, we can observe that during normal operation (i.e. no DLL gets blocked) the load time is always well under 50ms, which would not be noticeable by the end-user.

5 Related works

Min and Varadharajan [14] proposed a cross-verification mechanism which can be described as a bi-directional trust relationship between a caller and a callee (e.g. an executable and a DLL). The identity of a file is based on its digital signature, more precisely on the entity that signed the program. However, establishing mutual trust relationship at scale "may be hard", as per the author's own words.

Gates *et al.* [5] suggested adding an *installation mode* to Windows that requires a system reboot to be enabled. Only the binary files (e.g. .exe, .dll, etc.) that were created in the installation mode would be allowed to have their code executed. Nonetheless, it ultimately places the burden on users to avoid introducing malware on their computer. This is not a reasonable expectation, as social-engineering attacks are the easiest way for an APT to establish a foothold [10].

Wu and Yap [23] introduced the concept of "domains" for identifying binary files, with rules making domain-less binary files non-executable and to prevent files from different domains to tamper with each other. Their solution also employed an "install mode" mechanism that when enabled marked newly created files with the domain of the program that wrote them (as well as a special "temporary trusted" mode designed for software development and packed binary execution). The prerequisite for enabling the *install mode* is user authentication, with the same implications of potential social-engineering described before. However, correctly signed executables can be exempted which reduces the frequency of authentication, thus making users less tempted to authenticate without thinking about the implications.

Finally, Halim *et al.* [7] introduced a system-wide list of trusted software that covers all the file types containing executable code. This approach assigns identifiers to all binary files based on information such as their cryptographic hash, location, existing digital signature, etc. However, this system ultimately requires a qualified individual or entity to maintain the global allow list, which makes this approach incompatible with consumer use.

Conversely, our solution is easy to integrate, scalable thanks to per-program security policies (no administrator needed) and prevents social-engineering attacks, as user interactions cannot result in dangerous behaviour.

6 Conclusion & Future work

We proposed SLAHP, a security solution against DLL Search Order Hijacking which can be easily and quickly integrated with both new and already existing software products. We conducted a survey on widely used programs that showed many of them are still vulnerable to DSOH. SLAHP provides Windows developers with a complete framework to protect their software products against DSOH.

Additionally, they can continue choosing unsecure installation directories such as `AppData` to facilitate the installation process of their program without introducing new security issues. In addition, advanced users can also protect themselves should the editors of vulnerable software be indifferent. We assess the security as well as the performance, proving the feasibility of our solution.

There still are areas of our solution that could greatly benefit from further development and research. Indeed, we have observed that in certain cases, DLL loading attempts were made before SLAHP's initialisation which results in potential vulnerability; we believe the use of the `PreferSystem32` mitigation in SLAHP could solve this issue. As SLAHP is a proof-of-concept, some features are not currently as secure (e.g. IPC communications) or as configurable (i.e. No fine-grained configuration of `signatureallowsbypass`) as one would expect from a commercial software product.

In terms of long term research, we will extend the DLL Search Order Hijacking topic to low footprint attacks, in that they hide their malicious behaviour on infected systems behind the execution of trusted software, resulting in a very limited amount of indicators (e.g. dropped files). This is also the case of Living-off-the-Land (LotL) attacks, where threat actors abuse the features of trusted software components, chaining them together to accomplish their malicious activities without having to introduce new software components that could be subject to detection. In our ongoing research, we are delving deeper into the matter of LotL attacks and how we could detect them.

Acknowledgments

This work was partially supported by the European research projects H2020 CyberSec4Europe (GA 830929), Horizon Europe DUCA (GA 101086308), and CNRS EU-CHECK.

References

1. Chromium Docs - Chrome Security FAQ, `https://chromium.googlesource.com/chromium/src/+/master/docs/security/faq.md`
2. CrowdStrike: 2020 Global threat report (Mar 2020), `https://go.crowdstrike.com/rs/281-OBQ-266/images/Report2020CrowdStrikeGlobalThreatReport.pdf`
3. Faou, M.: Turla Crutch: Keeping the "back door" open (Dec 2020), `https://www.welivesecurity.com/2020/12/02/turla-crutch-keeping-back-door-open/`
4. Galvan, A., Nagaraju, S.S.: Triaging a DLL planting vulnerability | MSRC blog | microsoft security response center, `https://msrc.microsoft.com/blog/2018/04/triaging-a-dll-planting-vulnerability/`
5. Gates, C., Li, N., Chen, J., Proctor, R.: CodeShield: towards personalized application whitelisting. In: Proceedings of the 28th Annual Computer Security Applications Conference on - ACSAC '12. p. 279. ACM Press, Orlando, Florida (2012)
6. Gatlan, S.: Realtek Fixes DLL Hijacking Flaw in HD Audio Driver for Windows (Feb 2020), `https://www.bleepingcomputer.com/news/security/realtek-fixes-dll-hijacking-flaw-in-hd-audio-driver-for-windows/`

7. Halim, F., Ramnath, R., Sufatrio, Wu, Y., Yap, R.H.C.: A Lightweight Binary Authentication System for Windows. In: Karabulut, Y., Mitchell, J., Herrmann, P., Jensen, C.D. (eds.) Trust Management II, vol. 263, pp. 295–310. Springer US, Boston, MA (2008)
8. Hunt, G., Brubacher, D.: Detours: Binary interception of win32 functions. In: Third USENIX windows NT symposium. p. 8. USENIX (Jul 1999), https://www.microsoft.com/en-us/research/publication/detours-binary-interception-of-win32-functions/
9. Karantzas, G., Patsakis, C.: An Empirical Assessment of Endpoint Detection and Response Systems against Advanced Persistent Threats Attack Vectors. Journal of Cybersecurity and Privacy **1**(3), 387–421 (Jul 2021)
10. Krombholz, K., Hobel, H., Huber, M., Weippl, E.: Advanced social engineering attacks **22**, 113–122
11. Lechtik, M., Rascagnères, P., Kayal, A.: LuminousMoth APT: Sweeping attacks for the chosen few, https://securelist.com/apt-luminousmoth/103332/
12. Malura, M.: Dll proxy generator, https://github.com/maluramichael/dll-proxy-generator, original-date: 2018-09-29T20:51:52Z
13. Microsoft: Windows 2000 security hardening guide: Security configuration, https://web.archive.org/web/20080323071041/https://www.microsoft.com/technet/security/prodtech/windows2000/win2khg/05sconfg.mspx\#E6JBG
14. Min, B., Varadharajan, V.: Rethinking Software Component Security: Software Component Level Integrity and Cross Verification. The Computer Journal **59**(11), 1735–1748 (Nov 2016)
15. MITRE: Hijack Execution Flow: DLL Search Order Hijacking, Sub-technique T1574.001 - Enterprise | MITRE ATT&CK®, https://attack.mitre.org/techniques/T1574/001/
16. MITRE: Hijack Execution Flow: DLL Side-Loading, Sub-technique T1574.002 - Enterprise | MITRE ATT&CK®, https://attack.mitre.org/techniques/T1574/002/
17. National Vulnerability Database: NVD - CVE-2010-3129, https://nvd.nist.gov/vuln/detail/CVE-2010-3129
18. National Vulnerability Database: NVD - CVE-2010-3139, https://nvd.nist.gov/vuln/detail/CVE-2010-3139
19. Oliveira, D., Rosenthal, M., Morin, N., Yeh, K.C., Cappos, J., Zhuang, Y.: It's the psychology stupid: how heuristics explain software vulnerabilities and how priming can illuminate developer's blind spots. In: Proceedings of the 30th Annual Computer Security Applications Conference. pp. 296–305. ACM
20. Richter, J.: Load your 32 bit dll into another process's address space using injlib. Microsoft Systems Journal-US Edition pp. 13–40 (1994)
21. Wheeler, S., Sherer, T.: Set-ProcessMitigation (ProcessMitigations), https://learn.microsoft.com/en-us/powershell/module/processmitigations/set-processmitigation
22. Whitney, T., Robertson, C., Sharkey, K., nxtn, MSDN.WhiteKnight, Jones, M., Blome, M., Hogenson, G., Cai, S.: Linker support for delay-loaded DLLs, https://learn.microsoft.com/en-us/cpp/build/reference/linker-support-for-delay-loaded-dlls
23. Wu, Y., Yap, R.H.C.: Simple and Practical Integrity Models for Binaries and Files. In: Damsgaard Jensen, C., Marsh, S., Dimitrakos, T., Murayama, Y. (eds.) Trust Management IX, vol. 454, pp. 30–46. Springer International Publishing, Cham (2015)