Empirical Software Engineering manuscript No.
(will be inserted by the editor)

An Empirical Study of DLL Injection Bugs in the Firefox Ecosystem

Le An · Marco Castelluccio · Foutse Khomh

Received: date / Accepted: date

Abstract DLL injection is a technique used for executing code within the address space of another process by forcing the load of a dynamic-link library. In a software ecosystem, the interactions between the host and third-party software increase the maintenance challenges of the system and may lead to bugs. In this work, we empirically investigate bugs that were caused by third-party DLL injections into the Mozilla Firefox browser. Among the 103 studied DLL injection bugs, we found that 93 bugs (90.3%) led to crashes and 57 bugs (55.3%) were caused by antivirus software. Through a survey with third-party software vendors, we observed that some vendors did not perform any QA with pre-release versions nor intend to use a public API (WebExtensions) but insist on using DLL injection. To reduce DLL injection bugs, host software vendors may strengthen the collaboration with third-party vendors, *e.g.*, build a publicly accessible validation test framework. Host software vendors may also use a whitelist approach to only allow vetted DLLs to inject.

Keywords DLL injection · Software ecosystem · Mining software repositories

1 Introduction

Firefox, since its inception, has always provided APIs to extend the functionality of the browser. There has been an evolution of methods to extend the functionality towards safer and more stable methods (starting from plugins such

Le An[†] and Marco Castelluccio[‡] (joint first authors, contributed equally)
[†] Polytechnique Montreal, Canada
[‡] Mozilla Corporation, United Kingdom and University of Napoli Federico II, Italy
E-mail: le.an@polymtl.ca, mcastelluccio@mozilla.com

Foutse Khomh
Polytechnique Montreal, Canada
E-mail: foutse.khomh@polymtl.ca

as Flash, moving to XUL/XPCOM extensions, then ending with JavaScript/HTML WebExtensions). While Firefox and other equivalent browsers provide public APIs for extending functionality, a lot of *third-party software* (*i.e.*, software that adds code into another software) still employ DLL injection techniques, *i.e.*, techniques that forces *host software* (*i.e.*, software that allows other software to extend its functionality) to run arbitrary code by making it load a dynamic-link library (DLL). By injecting arbitrary code, third-party software can extend the functionality of the host software without limits. However, injecting arbitrary code, while it is a very powerful technique, can easily cause severe bugs, such as crashes, in the host software. As can be seen in [23], bugs arising from injection can be indeed severe and widespread as to delay or cause revisions of entire software releases.

To the best of our knowledge, there has not been an empirical study towards understanding the DLL injection landscape, why third-party software vendors still employ these techniques despite the availability of safer alternatives, the root causes of DLL injection bugs, and proposing solutions to reduce them. This motivated us to conduct this work, in which we analyzed DLL injection bugs that occurred from July 2015 to August 2017 in the Firefox ecosystem. In particular, our study aims to answer the following three research questions:

RQ1: What are the characteristics of the bugs caused by DLL injections?

We observed that most of the DLL injection bugs led to severe problems. Out of the 103 studied bugs, 93 bugs (90.3%) caused crashes (among them, 47 bugs (45.6%) crashed Firefox while the browser was starting) and four bugs (3.9%) made the browser hang (*i.e.*, losing responses from users' requests). By analyzing the types of the third-party software, we found that 57 bugs (55.3%) derive from antivirus software, 19 from hardware vendor drivers, and 10 from malware.

RQ2: Which factors triggered the DLL injection bugs?

To further understand the root causes of DLL injection bugs, we surveyed third-party vendors who caused the bugs. From their responses, we learnt that third-party software uses a variety of techniques (including standard Windows DLL injection techniques and proprietary techniques) to inject DLLs into the host software. DLL injection bugs can be triggered by injection engine errors, compiler/runtime incompatibility, or version incompatibility between the host and third-party software.

RQ3: What would be the potential solutions to reduce such DLL injection bugs?

In the survey, we also asked questions about the potential solutions that could reduce DLL injection bugs. From the answers, we realized that DLL injection should not be outright blocked from the ecosystem because it could be useful under certain circumstances, *e.g.*, when antivirus software intercepts suspicious processes. Host and third-party software vendors

should strengthen their collaboration. Host software vendors should extend the features of the extension API (as a safer alternative to DLL injection) and can build a publicly accessible validation test framework.

The rest of the paper is organized as follows. Section 2 provides background knowledge on the Firefox ecosystem as well as the risks and countermeasures of DLL injection in the system. Section 3 describes the design of the case study. Section 4 shows and analyzes the results of the case study. Section 5 discusses the implications of our findings. Section 6 discusses the threats to the validity of our study. Section 7 summarizes related work, and Section 8 draws conclusions.

2 Background

2.1 Firefox Ecosystem

There are several ways third-party developers have been able to extend the functionality of Firefox: a) themes; b) plugins; c) extensions; d) DLL injection.

Themes are only allowed to change UI elements of the browser, thus they are very limited.

The API used to build plugins, NPAPI (Netscape Plugin Application Programming Interface), has been introduced by Netscape in 1995, and later adopted by most major browsers. NPAPI plugins declared content types that they could handle. When the browser was not natively able to handle that content type, it would load the appropriate plugin and let it run. NPAPI plugins are binary plugins, and they have been slowly deprecated for security reasons (*e.g.*, Chrome dropped NPAPI plugins in September 2015, Firefox dropped all NPAPI plugins except Flash in March 2017 and will drop Flash too in 2019).

Since its inception, Firefox has also allowed third-party developers to extend the functionality of the browser through JavaScript/HTML APIs by writing extensions. Extensions are either self-hosted, or hosted on a Mozilla website called AMO (addons.mozilla.org). When hosted on AMO, they undergo code review by Mozilla employees and/or volunteers. Since Firefox 44 (released in January 2016), Mozilla introduced a signing requirement where all extensions (either self-hosted or hosted on AMO) must be signed by Mozilla in order to be installable in Firefox (with the objective of reducing malware). This means that all extensions since Firefox 44 undergo code review.

Initially, extensions had access to browser internals (using XUL/XPCOM APIs); meaning that they could introduce technical debt into Firefox itself, as Mozilla developers could not easily modify Firefox internal code that was being used by extensions.

To ease development and to make extensions higher level (which would allow Mozilla to change their internal APIs without breaking existing extensions), Mozilla later introduced an extension SDK (JetPack). Behind the hood, JetPack extensions were still using XUL/XPCOM APIs.

A new set of APIs, the WebExtensions API [25], was later introduced in alpha state in November 2015, then in stable state since August 2016. Since November 2017, following a major rewrite of the browser which would have made many extensions incompatible, all extensions are required to use the WebExtensions API, which is an API supported by many major browsers (Firefox, Edge, and Chromium-based browsers). The advantage of such a common API is that developers only need to write a single extension and it will (modulo implementation differences) work on multiple browsers seamlessly, much like the web. The WebExtensions API is more restrictive than the old APIs, but also more secure and stable, and with better performance characteristics [24] [22]. Moreover, since these extensions are not allowed to use Firefox internal APIs, they cannot introduce technical debt as the old extension APIs used to do.

Another way that third-party developers use to extend the functionality of the browser (and of other software) is DLL injection.

2.2 Risks of DLL Injection and Countermeasures

By employing DLL injection, third-party developers are able to inject in the Firefox process any type of code, whose behaviour was not intended nor anticipated by Mozilla developers.

DLL injection is a powerful technique as it allows third-party developers to extend the functionality of the host software however they want, but it can be very risky. The injected code can, for example, use internal functions of the host software, without the knowledge of the host software developers, thus causing crashes or other problems when the host software removes or changes the behaviour of those functions. In order to use internal functions of the host software, some injected code depends on the binary layout of the host software, which changes for every specific build. If there are no mitigations in place, the injected code can cause crashes for every new release of the host software.

Figure 1 shows an excerpt of some buggy code injected in Firefox by a software using an open source library, EasyHook[1]. This is one of the few examples that can be shown, as usually the injection techniques are proprietary. In this example, Firefox is the host software (whose functionality is extended) and the software using the EasyHook library is the third-party software (which injects its code into Firefox). The process of the third-party software used the `CreateRemoteThread` function[2] to create a thread that runs in the Firefox process address space. The thread would call the `Injection_ASM_x86` function, which first loads the library to inject (line 11), then tries to find the entry point of the library using the `GetProcAddress` function (`AcLayers!NS_Armadillo::APIHook_GetProcAddress()`, from the Windows DLL: `AcLayers.dll`) (line

[1] `https://github.com/EasyHook/EasyHook`

[2] `https://docs.microsoft.com/en-us/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createremotethread`

```
1  public Injection_ASM_x86@0
2    Injection_ASM_x86@0 PROC
3    ; no registers to save, because this is the thread main function
4    ; save first param (address of hook injection information)
5
6      mov esi, dword ptr [esp + 4]
7
8    ; call LoadLibraryW(Inject->EasyHookPath);
9      push dword ptr [esi + 8]
10
11     call dword ptr [esi + 40] ; LoadLibraryW@4
12     mov ebp, eax
13     test eax, eax
14     je HookInject_FAILURE_A
15
16   ; call GetProcAddress(eax, Inject->EasyHookEntry);
17     push dword ptr [esi + 24]
18     push ebp
19     call dword ptr [esi + 56] ; GetProcAddress@8
20     test eax, eax
21     je HookInject_FAILURE_B
```

Fig. 1 An example of DLL injection performed by RoboSizer

19). This is where the crash occurs: the address to the `GetProcAddress` function was retrieved by the third-party software in its process, but then called in the Firefox process, expecting it to have the same function and at the same address. Since Firefox does not load `AcLayers.dll`, this function does not exist in its process. EasyHook later fixed the bug by retrieving the address of the function from the remote process, rather than the process doing the injection.

Other software employed a very similar technique to the one used by EasyHook, but using `apphelp!StubGetProcAddress()` instead (from the Windows DLL `apphelp.dll`. Again, the technique is not used by Firefox). `AcLayers.dll` and `apphelp.dll` are both part of Windows, providing fixes for backward compatibility. `GetProcAddress` is usually part of `kernel32.dll` (which is loaded in every process), but for such software, Windows was probably shimming the API for compatibility, redirecting to `apphelp.dll` or `AcLayers.dll`.

Mozilla later totally blocked this kind of injection mechanism which uses `CreateRemoteThread` (ironically, the code blocking this kind of injection mechanism triggered a bug in another third-party software, an antivirus, which was later fixed by the vendor).

Using public APIs rather than DLL injection is preferable. Besides the aforementioned examples, there are other reasons:

1. Since the WebExtensions API is supported by multiple browsers, the extension code only needs to be written once but can be deployed to different major browsers;
2. The public API is controlled by the browser vendor, who has information on the API's usage and can decide when to deprecate it (and when not to);

3. The extensions are written in JavaScript and HTML, just like normal web pages, which implies a very reduced chance of crashing the browser compared to the binary code that is injected with DLL injection;
4. Should an extension cause a problem, the browser can easily recover (*e.g.*, by reloading the extension). Instead, when an injected DLL causes a problem, it will likely lead to an unrecoverable situation.

Mozilla has been applying a blocklisting policy to react to bugs caused by third-party DLLs [27]. If a DLL causes a severe and–or widespread bug (such as an easily reproducible startup crash), Mozilla will, in parallel: a) try to contact the vendor of the third-party DLL and ask them to solve the problem; b) start preparing a blocklisting addition to block the DLL; c) attempt to reproduce the problem with its own quality assurance (QA) resources, if the third-party software is publicly available.

In order to solve the problem, third-party vendors usually request crash dumps from Mozilla, which often cannot be shared with external people for privacy reasons (the dumps might contain personal information of Firefox users). Mozilla may share crash dumps with third-party vendors only in the two following situations: 1) when Mozilla's QA manages to reproduce the crash; 2) when Mozilla manages to get in contact with users who can reproduce the crash (users can optionally leave their contact details when they submit a crash via Socorro, *i.e.*, Mozilla's automated crash reporting system) and the users agree to the sharing of crash dumps.

If the third-party software is publicly available, Mozilla will prepare modified Firefox builds that block the offending DLLs. Sometimes blocking a DLL is not easily feasible, as some DLL injection techniques operate at the kernel level. Sometimes blocking DLLs can cause more severe problems than the ones caused by the DLL itself. Hence, the blocklisting addition has to be tested first. If blocklisting works and does not cause regressions, Mozilla will apply the blocklisting patch, uplift it (*i.e.*, publish the patch ahead of the normal release cycle [6]), and, if the problem is widespread enough, generate a new release build to ship to users.

3 Case Study Design

In this section, we describe the data collection, design of the survey, and analysis approaches that we used to answer our three research questions.

3.1 Data Collection

From the Mozilla bug tracking system, Bugzilla [33], we searched bug reports that were created between July 2015 and August 2017. We chose this time window because the WebExtensions API was introduced in September 2015, and our study started in August 2017. In this work, we did not limit the analysis on already resolved bugs, because some bugs were closed as `WONTFIX`

or `WORKSFORME`, for example, if a DLL injection bug was deemed too hard to fix for very little benefit or if the influence of a DLL injection bug drastically decreased after the opening of the bug. From all the bugs in the studied time period, we selected the ones that matched at least one of the following rules:

- the Bugzilla component of the bug is the one Mozilla uses to track bugs caused by third-party software ("External Software Affecting Firefox::Other");
- the title of the bug contains one of the keywords: ".dll", "virus", "malware" or "adware";
- the whiteboard of the bug contains the text "AV", which Mozilla uses to mark some bugs caused by antiviruses.

We then manually analyzed the results of the search to filter out false positives, obtaining 103 bugs caused by external software through DLL injection.

The AV- and malware-specific rules only helped increasing our dataset slightly (5 out of 103 bugs), so our results should not be biased towards those kinds of software. Within the results from the other generic rules, we also found AV- and malware-specific bugs.

3.2 Data Processing

We manually identified a series of characteristics from the 103 bugs obtained in Section 3.1. Table 1 shows the names and the descriptions of the characteristics. To reduce biases in the manual identification, two of the authors separately collected the characteristics before comparing their results together. They created an online document to discuss any divergence until reaching an unanimous decision. In addition, we wrote scripts to automatically extract some other characteristics as shown in the bottom of Table 1.

3.3 Survey

To further understand the root cause of the DLL injection bugs and how the bugs were resolved, we designed a survey intended for the 58 vendors who caused these bugs. However, we could not find the contact information of 14 vendors (including the malware producers) from Bugzilla or through an online search. Hence, we ended up contacting only 44 vendors. Among them, 12 vendors answered all or part of our questions, which corresponds to a response rate of 27%. As we aim to propose potential solutions to reduce this kind of bugs, we also asked these software vendors questions on improving the reliability when adding their code into Firefox.

In our survey, we only used open questions. Participants could choose all or a part of the questions to answer. Our questions were designed to better understand the DLL injection landscape: what techniques are used, what kinds of bugs can arise, why DLL injection is still used as an extension mechanism despite the presence of safer techniques. Here are the questions we used in the survey:

Table 1 Characteristics of the bugs caused by third-party software.

Characteristic	Description
Manually collected characteristics	
Bug impact	Whether a bug broke the functionality of the browser, caused a crash (or startup crash), or caused a hang.
Software name	Name of the software that caused a bug. If no software name is mentioned in a bug report, we marked as "unknown".
Software type	Type of the external software, *e.g.*, antivirus, malware, and hardware vendor driver.
How resolved	How a bug is resolved, *e.g.*, fixed by the vendor, or blocked by Mozilla.
Reproducibility	Whether a bug can be reproduced by the QA of Mozilla or third-party vendors.
Automatically collected characteristics	
Percentage of DLL users	Percentage of Firefox users who also have the third-party software.
Fixing time	How many days it took for a bug to be fixed since its first occurrence. We cannot retrieve the first occurrence date for some bugs, we have to use the time period from the creation date until the fixed date to estimated these bugs' fixing time.
Tracked or blocking	Whether a bug was ever tracked for a release or was blocking a release. More information about Mozilla tracking flags and how they are used in the release management process can be found in [26].

Q1. What is the injection mechanism that you used?
Q2. Do you know the root cause of this bug?
Q3. If the bug is resolved from your part, do you remember the way by which you resolved this bug?
Q4. Since Mozilla is encouraging other organizations to produce their software as an extension, is there any specific reason why you are still using the way of DLL injection to add functionalities into Firefox?
Q5. Would you be open to switching to an extension-based solution if Mozilla gave you the API you needed?
Q6. Do you run QA with pre-release versions of Firefox (*e.g.*, Firefox Beta)?
Q7. Do you have any suggestions to improve the Mozilla API extension?

A possible approach to mitigate the DLL injection issues is to adopt a whitelist solution. Instead of reacting to DLL injection issues by blocklisting misbehaving DLLs, Mozilla could proactively block all DLLs except "good" ones. The vendors in the whitelist would need to be more careful and perform QA in order to be in the whitelist. Once a whitelisted DLL causes a problem, it will be removed from the whitelist. Also, developers using the WebExtensions API would effectively be exempt and would always be in the whitelist. Besides reducing bugs, Mozilla expects that this mechanism can push third-party software vendors to use the WebExtensions API, which can also avoid crashes in the third-party code taking down Firefox [23].

To evaluate how this solution would be received by third-party vendors, we asked additional questions to the vendors who have answered our initial questions. During this work, we consulted some Mozilla developers by email and added these follow-up questions based on their suggestions.

Table 2 Impact of the DLL injection bugs (some bugs have more than one impact)

Bug impact	Occurrence	Proportion
startup crash	47	45.6%
crash (unknown)	25	24.3%
crash	21	20.4%
broken functionality	8	7.8%
hang	4	3.9%
plugin crash	2	1.9%

Q8. In your opinion, what would be a solution to allow for an effective integration of third-party code into software like Firefox?
Q9. Some software vendors are moving to instruct users to uninstall third-party software after a crash, what do you think of such practice?
Q10. When Firefox rolls out new content security features, it often runs into compatibility issues with third-party suites that leverage injection. What steps do you think Firefox should take to prevent these issues with your product(s) in the future?
Q11. What support might you be willing to provide to avoid these issues in the future?
Q12. If Firefox blocks third-party injection associated with your product, what side effects do you anticipate? Would this potentially break your software product(s)? Could this break Firefox?
Q13. Some vendors are considering introducing a whitelist that only allows "reliable" DLLs to be installed. Would the whitelist be an incentive to adopt the cross-browser WebExtensions API? (products using the extension API are always whitelisted)
Q14. Would the existence of a whitelist be an incentive for your company to do more QA with Firefox?
Q15. Would your company try to circumvent the whitelist? If yes, how would you do it?

4 Case Study Results

We present the results of our case study and discuss the implications of these results.

4.1 (RQ1) What are the characteristics of the bugs caused by DLL injections?

According to Mozilla telemetry[3], large shares of Firefox users are also users of software employing DLL injection to extend Firefox functionality. Each major third-party software can be installed on between 1% and 15% of Firefox users' machines. Severe bugs affecting a DLL from a third-party software that is

[3] `https://wiki.mozilla.org/Telemetry`

Table 3 Types of the DLL injection software

Software type	Occurrence	Proportion
antivirus	57	55.3%
hardware vendor driver	19	18.4%
malware	10	9.7%
multimedia tool	4	3.9%
screen reader	3	2.9%
other	3	2.9%
IME	2	1.9%
download manager	2	1.9%
desktop customization	1	1.0%
file hosting service	1	1.0%
accessibility	1	1.0%

Table 4 How the DLL injection bugs were fixed (some bugs were fixed by more than one resolution)

Resolution	Occurrence	Proportion
fixed by the vendor	24	23.3%
worksforme	18	17.5%
not yet resolved	18	17.5%
blocklisted	16	15.5%
duplicate	12	11.7%
wontfix	8	7.8%
workaround	5	4.9%
invalid	2	1.9%
fixed by switching to WebExtension	2	1.9%
fixed bug in firefox	1	1.0%

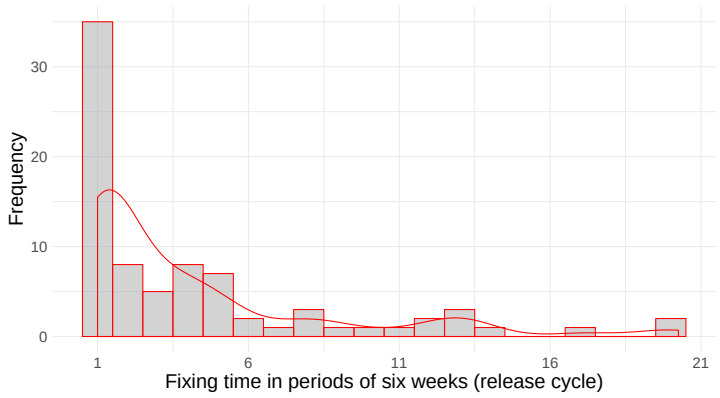

Fig. 2 Distribution of the bug fixing time. Each bin represents a period of six weeks, *e.g.*, the first bin means bugs fixed within six weeks (*i.e.*, one release cycle).

installed on 15% of users' machines (or even 1%) can be very concerning for Mozilla.

Table 2 shows the distribution of the impact of the DLL injection bugs. Out of the 103 studied bugs, 93 bugs (90.3%) caused browser crashes, *i.e.*, the browser unexpectedly terminates. Among them, 47 bugs (45.6%) caused crash during the browser startup (the most severe type); 21 (20.4%) crashed

while the browser was running; we could not deduct the type of crash from the other 25 bugs (24.3%) (*i.e.*, uptime unknown). Besides, two bugs (1.9%) crashed a browser plugin. In addition, four bugs (3.9%) caused hangs, *i.e.*, the browser does not respond to users' requests. Only eight bugs (7.8%) have lower severity. They break the browser's expected functionality. The overall impact of the DLL injection bugs are severe, which can negatively affect users' trustfulness on the quality of the browser. From the side of users, they may not know whether the severe problems (such as crashes) are caused by the host software itself (Firefox in this case) or by its interaction with third-party software (usually they will just assume it is the host software, since that is the one which crashes, even if the crash stems from injected code). If the problems are kept unresolved for a long time, users may switch to other equivalent products. Especially for startup crashes, where users cannot use the browser at all, nor automatically update it to a newer version when a fix is released by Mozilla. The only options for them are to manually reinstall Firefox after a fix is released, wait for an update of the third-party software, or switch to use another browser.

Table 3 shows the types of the DLL injection software. More than half of the bugs (57, *i.e.*, 55.3%) are from antivirus software, 19 (18.4%) are from hardware vendor drivers, 10 (9.7%) are from malware, and 17 (16.5%) are from other software, including multimedia tools, screen readers, input method tools (IME), and download managers. Overall, except for a small amount of malware and purpose-unidentified software, most bugs are derived from DLLs that provide useful features to users.

Table 4 shows how the DLL injection bugs were resolved (or not resolved). 58 bugs (56.3%) were not actually resolved by the time of this study. Some of the bugs were closed with a label as "WORKSFORME" (bugs can no longer be reproduced), "INVALID" (bugs are in the third-party software and with low enough severity), "WONTFIX" (due to low or decreased volume of impact), or "DUPLICATE" (duplicate of another resolved bug). Unfortunately, the labels are not always used consistently (for example, bugs with very low impact are sometimes resolved as INVALID and sometimes as WONTFIX). Besides, five bugs (4.9%) were fixed by employing workarounds (temporary and ugly solutions). For the bugs that were actually resolved, 16 (15.5%) were fixed by Mozilla by blocklisting the offending DLLs; 24 (23.3%) of them were fixed from the vendor side. Only two bugs (1.9%) were resolved by switching to using Mozilla's WebExtension API as recommended. Merely one bug (1%) was not due to the DLL vendors but due to defects of Firefox. From the result, we observe that a weak percentage of the bugs can be resolved by the host software itself (Firefox). Third-party vendors' efforts and collaboration are important to keep the Firefox ecosystem healthy. Moreover, few third-party vendors have adopted Mozilla's recommendation of using the WebExtensions API.

Figure 2 depicts the time period (in six weeks periods) during which the DLL injection bugs were resolved. In this figure, we only considered the 81 bugs that were closed by the time of this study. 40 bugs were fixed within a period of six weeks; meaning that nearly half of the DLL injection bugs

can be fixed before the next release. 55 bugs were resolved within 18 weeks, a full release cycle from Nightly to Release. End users can benefit from the resolution of these bugs within three releases (a new version is released every six weeks). However, we also observed 10 bugs that were not resolved for more than one year. Moreover, 22 other bugs have never been resolved until the writing of this paper. Long resolution time of DLL injection bugs challenges users' trustfulness not only to the third-party software, but also, and in many cases even more, to the host software. To maintain the health of the ecosystem, both sides of the host and third-party software need to actively and effectively discover and resolve bugs. We found that some bugs, such as Bug #1268470, were resolved late because at the time of reporting the bug, it affected only a small number of users. When the bug started affecting more users, it attracted Mozilla's attention.

Although Bugzilla has priority/importance fields, they are used inconsistently by different developers and different teams, thus cannot be relied upon to infer the importance of a given bug. In order to evaluate the actual severity of the bugs, we analyzed the Bugzilla tracking flags that are used by Release Managers during the release process [26]. We found that 32 bugs (31.1%) were tracked or blocking for a release at least once. These kinds of bugs are particularly important because they either have been closely monitored by release managers for possible resolution in a Firefox release (tracked bugs: 24, 23.3%) or have been marked as blocking (**must be fixed before shipping**) a Firefox release (blocking bugs: 8, 7.8%). To put it into perspective, we can compare these percentages with the overall ones: 3390 tracked bugs (around 0.037%) and 165 blocking bugs (around 0.002%). This means that DLL injection bugs, even though expectedly rarer than other bugs, are often more severe than other bugs. We also compared the fixing times of DLL-injection blocking/tracked bugs with those of generic blocking/tracked bugs. In addition, we found that the average fixing time is around 3.4 times higher for DLL-injection tracked bugs than generic tracked bugs (for blocking bugs the average is 2.8 times higher). However, the differences are not statistically significant based on the Mann-Whitney U test [14]. One reason is that there are too few samples in our dataset.

Finally, 26 (25.2%) of the DLL injection bugs could be reproduced by Mozilla or third-party vendor's QA, four (3.9%) of the bugs could not be reproduced, and we cannot identify whether the rest 73 bugs (70.9%) could be reproduced or not. For bugs that were reproducible, additional QA performed by either Mozilla or the third-party vendors before a Firefox release could have prevented the bug from hitting users. Among the aforementioned eight blocking bugs (account for 7.8%), five of them could be reproduced by Mozilla or third-party QA, one of them could not be reproduced, and we cannot identify the reproducibility for the remaining two bugs. If more in-depth QA was part of the envisioned whitelist policy of Mozilla, many of these blocking bugs could have been resolved before they became blocking.

Table 5 Statistics on the survey participants (all participants are from different vendors)

Question	Participants	Software type (and its frequency)
1	12	antivirus (7) screen reader (1) unknown (1) internet downloader (1) media recorder (1) hardware vendor driver (1)
2	12	antivirus (7) screen reader (1) unknown (1) internet downloader (1) media recorder (1) hardware vendor driver (1)
3	10	antivirus (5) screen reader (1) unknown (1) internet downloader (1) media recorder (1) hardware vendor driver (1)
4	11	antivirus (7) screen reader (1) unknown (1) hardware vendor driver (1)
5	7	antivirus (3) screen reader (1) unknown (1) hardware vendor driver (1)
6	6	antivirus (2) screen reader (1) unknown (1) hardware vendor driver (1)
7	5	screen reader (1) unknown (1) hardware vendor driver (1) media recorder (1)
8	5	antivirus (3) media recorder (1)
9	4	antivirus (3) media recorder (1)
10	4	antivirus (3) media recorder (1)
11	4	antivirus (3) media recorder (1)
12	4	antivirus (3) media recorder (1)
13	4	antivirus (3) media recorder (1)
14	5	antivirus (3) media recorder (1)
15	4	antivirus (3) media recorder (1)

93 bugs (90.3%) of the DLL injection bugs led to crashes. 57 bugs (55.3%) of the bugs are from antivirus software, 19 (18.4%) of them from hardware vendor drivers, and 10 (9.7%) from malware. 1% to 15% of Firefox users also have some of the software that caused these bugs.

4.2 (RQ2) Which factors triggered the DLL injection bugs?

Firefox is an open source browser. Its crash and bug reports are also open to the public. Developers and researchers can leverage these resources to understand the root causes of most bugs. However, through our manual analysis, none of the DLL injection software that caused bugs in Firefox is open source. Thus, we cannot understand the root causes of these bugs from source code. As we observed in **RQ1**, many subject bugs, which were eventually resolved, were fixed by the software vendors or blocked by Mozilla. In both cases, Mozilla did not know the triggers. Although the third-party vendors knew the triggers of the bugs they resolved, they rarely mentioned them in the bug reports. In other words, bug reports cannot help us to understand the bugs' root causes either. Therefore, to answer this research question, we decided to ask the software vendors themselves. In the rest of this section, we will show the vendors' responses to the corresponding survey questions and discuss these responses. Table 5 shows statistics on the participants for each survey question. In this table, we respectively provided the total number of participants who answered a

question, types of these participants' software, and number of participants for each type of software. All the reported responses are from closed source software vendors. Due to privacy reasons, we may have hidden some confidential details.

DLL injection mechanisms used by the software vendors (Q1). We received 12 responses to the question related to the injection mechanisms used on Firefox. Two general kinds of mechanisms can be identified from the responses: standard Windows techniques and proprietary techniques. Among the eight responses on the standard techniques, seven participants explained the detail of their technique, one participant only mentioned that their DLL injection technique is standard for the Windows OS. Here we quote our participants' answers to this question: *"It's just a standard Shell Extension that runs when folks use the open/save dialogues." "We use SetWinEventHook [29] from user32.dll." "We used a general mechanism (SetWindowsHookEx [28]) to inject other processes in order to be able to influence window creation flags in case the user decides to not be disturbed in Game Mode / Do Not Disturb Mode." "AppInit_dll [2] registry entry." "CreateRemoteThread+LoadLibrary [8, 21]".*

Three participants said that they used proprietary techniques, but none of them revealed details. Two other participants did not directly answer this question but said that the injection mechanism is irrelevant to the bugs. Overall, **third-party software uses a variety of techniques to inject DLLs into the host software**.

Root causes of DLL injection bugs and resolution mechanisms (Q2, Q3). Our second and third questions concerned the root causes of the bugs and how the bugs were resolved. Nine participants explained the root causes of the bugs caused by their injected software. 10 participants explained the resolution process of the bugs caused by their injected code. Some bugs were caused by the injection engine. The participants said: *"Bug in hook engine. Legacy code not covered by automatic tests."*, *"Problem was internal to the hooked functionality and likely not dependent on Firefox code"*. The DLL vendors resolved the bugs by fixing their injection code.

Compiler or runtime incompatibility is another cause mentioned: *"Our compiler wasn't C++ 11 compliant and therefore introduced a race initialization of a mutex." "(Our DLL) was incompatible with C++ runtime, shipped with Windows 8.0 x64. It is not depend of upgrade or clear installation of FF (Firefox). In addition, it should not depend from browser, for crash it is enough Windows 8.0 x64 C++ runtime and any browser."*. Participants did not provide detailed information about the resolution of this problem. We suppose that upgrading the compiler would address the bugs.

Some other bugs were due to generic programming mistakes, which were later resolved and made the DLL work again. One participant explained: *"It was a mistake regarding 64 and 32 bit values in our code base." "bad_alloc wasn't caught in our code."*

In addition, bugs can also occur when *"users forcibly loaded old extensions to newer versions of Firefox and disabled compatibility checks ... (Old versions of Firefox) missed a check for NULL on one of interface queries. The issue started to persist after significant changes in Mozilla interfaces."* To reduce this kind of bugs, the host software can alert users to upgrade their old version of the third-party software, and warn them of the potential consequences of the incompatibilities on the host/third-party software versions.

Based on our observations, **most bugs are due to injection engine problems, compiler/runtime incompatibility, or version incompatibility between the host and third-party software**. This finding corroborates what we found in RQ1: most bugs are in third-party software's code and thus cannot directly be fixed by Mozilla.

In many cases, DLL injection bugs are triggered by injection engine errors, compiler/runtime incompatibility, or version incompatibility between the host and third-party software.

4.3 (RQ3) What would be the potential solutions to reduce such DLL injection bugs?

Unreliability challenges all software ecosystems. To reduce potential crashes caused by third-party software, from September 2018, Chrome will try to block most third-party software that injects code into it [7] (Chrome developers claim that "users with software that injects code into Windows Chrome are 15% more likely to experience crashes"). The organization hopes third-party software can switch to use the recommended WebExtensions API to run code inside Chrome processes. Mozilla is also trying to reduce bugs caused by third-party software, while avoiding outright blocking, by introducing a whitelist to allow only DLLs, which are proved reliable, to inject code into Firefox. With the same expectation as Chrome, Mozilla hopes that this measure can make third-party software vendors switch from DLL injection to WebExtensions, which is considered as a more reliable way to interact with Firefox. In this paper, by analyzing survey participants' answers, we want to discuss whether the whitelist is the best solution to reduce bugs from third-party software, and whether there are better alternatives to it.

Reasons provided for not adopting WebExtensions (Q4).
First, we wanted to know the reasons why many third-party vendors are still using the way of DLL injection, although WebExtensions have been available for a while (in alpha state since Firefox 42, released in 2015-11-03; in a stable state since Firefox 48, released in 2016-08-02). This corresponds to Question #4 in the survey. 11 participants answered this question. Multiple participants mentioned that their DLL is not specifically designed for Firefox but is also being used for other host software, *e.g.*, *"Our software is not just used*

for FF (Firefox). It is a general purpose audio recorder. Users choose which application they wish to target." For these vendors, migrating to WebExtensions would not be interesting because it requires extra efforts to refactor the existing code.

Another reason is that some vendors cannot use WebExtensions to achieve their goal, *e.g.*, *"We must be able to gather content from Firefox. The most efficient way being to inject. Extensions are not suitable for Screen Reading software such as ours"*. An antivirus vendor said: *"We provide secure input feature in our product, which means that no one can intercept symbols, which user input in browser fields. The task could not be done on Windows OS without kernel driver and injected dll in browser"*. Another antivirus vendor explained: *"As hackers always inject, while we are reducing to minimize our injections, we cannot totally eliminate them"*. This would partially explain why a big percentage of DLL injection bugs derive from antivirus software. Due to the above two reasons, if a host software banned DLL injections, the vendors will have to find other feasible hosts.

Moreover, some participants indicated the disadvantages of WebExtensions, *e.g.*, *"The main disadvantage we find is that WebExtensions can be easily disabled (for a user with admin-rights, and in a Windows workgroup environment). We had taken this route of injecting a DLL to enforce URL filtering even in such environments"*. Again, DLL injection is currently the most suitable way for such vendors.

Only one participant is willing to accept WebExtensions, but they also said that WebExtensions cannot fulfill some particular purposes, which is inline with the aforementioned observations.

In general, **some DLL vendors do not want to adopt WebExtensions, because they do not target for one specific host software, and the features currently offered by the WebExtensions API are still limited for some purposes**. One participant told us that their organization has thoroughly analyzed the pros and cons about using WebExtensions. However, they still keep using DLL injection because they *"don't see any way how and why to stop injecting there (in order to protect our users, which is our business)"*. We cite their analysis here and hope that host software organizations can take this as a reference to improve the extension API and–or communicate better about their advantages.

"In comparison with injection, extension has much worse deployment possibilities – the installation process is cumbersome (you can't install the extension silently without user interaction which is a major UX problem, you can't protect the extension from uninstalling, you'd need to check for browser reinstalls and install again etc).
Also, it's possible to write the extension, but since the API is limited (everyone saw the 2/3 of extensions being removed from new Firefox because of API problems) and the model is also asynchronous, which kinda gets in a way what would AV product need. And the next point against extensions is a need for three different extensions for three browsers – although they all use WebExten-

sions, they're quite different. And MSIE is still there, with stronger presence than Edge."

Migration from code injection to WebExtensions (Q5).
Q5 is about whether third-party vendors are open to switch to WebExtensions if Mozilla gave them the needed API. Seven participants answered this question. One participant, who is the one saying that WebExtensions can be easily disabled, simply said Yes. Those vendors targeting multiple hosts answered No, because *"Mozilla doesn't control the surface area we modify"*.

A participant suggested that if different host software organizations can standardize their APIs, third-party vendors will be more willing to migrate. *"It depends on the functionality and if there are general, OS runtime based standard mechanisms already available. It makes no sense to have two different implementations of the same functionality."*

Other participants' attitude is rather open, but they doubt whether Mozilla can provide the specific API they require. For example, *"I doubt that the extension mechanism would be sufficient for our requirements. However, we, Mozilla, and other vendors are actively considering other ways that software such as ours would not have to inject to gather this content."*

"We are combatting malware and exploits though, which work in a low-level way, directly manipulating Firefox code and interacting with the operating system. It is quite unlikely that a high-level extension (i.e., JavaScript) can be used to detect and mitigate all those threats reliably."

"Actually, we prefer to use 'standard' means whenever possible ... The main concern is, how do you expose the API without any malicious software using it."

Overall, **although some third-party vendors are open to adopt WebExtensions API, they doubt whether the API can fulfill their requirements**.

Quality assurance of injected code (Q6).
Six participants answered whether they run QA with pre-release versions of Firefox. Four participants said Yes, one of them further explained: *"but not as often as we would like"*. The other two said No. In our opinion, running QA against each version of the host software is necessary. The vendors who neglect this process may miss bugs in the ecosystem. In this case, the whitelist would be an effective measure to penalize the vendors who do not test their software well and frequently have bugs.

Suggested improvements to the WebExtensions API (Q7).
Q7 encourages participants to suggest improvements for the WebExtensions API. One participant wished that *"(Mozilla) can provide a mean to get the HWND [37] of a window from within the extension"*. This suggestion is in line with the doubts on the functionality offered by the WebExtensions API.

Another suggestion is about the reliability of the API itself: *"Some of the mechanisms (of WebExtensions) do not work ... We opened a bug (on this problem)"*. Therefore, completely blocking DLL injection may not be the

best solution because if a third-party vendor can neither use DLL injection nor program against an available/reliable API, they have to give up the host software and find other platforms. However, if all browsers move to reduce DLL injection, third-party software will be forced to gradually transition to WebExtensions.

To further discuss the solutions of reducing DLL injection bugs, we will analyze the answers on the follow-up questions. Some of the questions are targeted for the upcoming whitelist by Mozilla. Only five participants answered these questions. Their answers may not be representative, but can be used as a reference for host and third-party software to improve the reliability of an ecosystem. In the following of this section, we will cite their answers and discuss the implications.

Allow an effective integration of third-party software into another software (Q8).

Our follow-up questions start by how to allow an effective integration of third-party software into another software. Our participants answered as follows: *"Certainly the most common extensions can and should be handled by a plugin API like WebExtensions. Additionally, having a link to AMSI (Anti-Malware Scan Interface) by Microsoft would make sense. But generally, what Windows supports should be also supported by Firefox, which also includes code injection. For monitoring the process state on a system level, sometimes there are no other options that would come to my mind."*

"Use of extensions is the most effective method. However, in enterprise environment, admin would want to enforce use of certain extensions (without allowing a user to disable it). Browsers allow enforcing certain extension through group policy in domain environment. However, we have a lot of SMB (small and midsize business) customers who don't have domain-network environments. Solving that requirement is tricky."

"There (should be) an extensive QA verification process in place that includes Firefox test scenarios and a working collaboration with Mozilla. One proven approach to improve the code quality of external components is to establish a publicly accessible validation test framework that provides the test scenarios an extension has to pass and where test scenarios are updated, based on observances with field issues."

"If they can provide an API (e.g., callback) that will be available only for registered whitelisted DLLs, we can move to that model instead of our current model and reduce even more compatibilities issues."

Based on their answers, besides the extension API, third-party software vendors believe that DLL injection should also be kept as an option since it is legally supported by the operating system. The collaboration between host and third-party software is necessary to ensure the quality of an ecosystem. Particularly, a publicly accessible validation test framework can help standardize the QA for both parties. Moreover, the upcoming whitelist seems to be a favourable solution for some third-party vendors.

Whether suggest users to uninstall third-party software after a crash (Q9).
We then were curious to know the opinions of third-party vendors on the practice that some host software (*e.g.*, Chrome [7]) will suggest users to uninstall third-party software after a crash. We received a favourable opinion *"If an app crashes on your machine then sure uninstall it. Makes complete sense. Not all machines are created equal."* versus multiple against opinions *"I consider this generally to be a bad practice, especially when a crash can't be clearly attributed to a particular third-party software – which is usually not possible in an automated way." "They put their customers at risk, since the legitimate (e.g., antivirus) will be removed ... If I were malware, I will use this functionality to ask users to remove any 3rd party mechanisms that prevent me from doing whatever I need." "Uninstalling third-party solution isn't a long term solution."*

From the answers, we can see that this is a complex problem. First, such suggestions may become false alarms to users because a host vendor cannot simply decide whether a crash is due to the third-party or the host software itself. Second, in the Mozilla ecosystem, many crashes are caused by antivirus software. If such antivirus software is uninstalled, malware may take advantage of this. **Facing a third-party software related crash, we suggest that host vendors warn users about the potential risks of running the third-party software (*e.g.*, by showing the number of crashes) but also remind them of the risks of removing it.** Besides, host vendors should investigate whether the crash happens with other equivalent host software. Moreover, host vendors should always make efforts to improve the reliability of their platform if necessary, because if users value the importance of the third-party software and find it working well with other hosts, they may uninstall the host software instead.

Incompatibilities between host and third-party software (Q10, Q11).
Q10 and Q11 are about the way to prevent incompatibilities between host and third-party software when the host software rolls out new content security features. Our participant suggested: *"Notify us like they did when there is an issue. Worked well last time. We have a fix rolled out very quickly when we were made aware of the issue."*

"Browser vendors can closely work with security vendors to bring about more stable, secure browser ecosystem."

"A preview of such functionality to test it in our labs will be highly appreciated (with enough leeway and documentation to have the time for the vendors to adapt their code)."

In the meanwhile, the participants told us that they are willing to take the following measures from their part. *"We always try and fix any issues with our software when they are reported to us. We do this as soon as we were alerted to the problem."*

"Regular compatibility testing of latest aurora/beta releases of various browsers from our side along with our product and addresses any issues found."

"We are willing, and already testing, any beta and post beta releases. But if we can get documentation and enough time, we can commit to have our code ready and tested by the release date (or if push comes to shove, temporarily some remove functionality to accommodate browsers releases)."

Overall, we learnt that many third-party vendors are making efforts on compatibility testing and bug fixing for each (pre-) release. **A good communication between host and third-party software would help to reduce incompatibilities due to new security features. Mozilla can provide some preview and necessary documentation of the new features to the trusted (*i.e.*, whitelisted) vendors (for compatibility testing) before the features are released to users.**

Blocking of third-party DLLs (Q12).
Blocking third-party DLLs is one the of measures host software is using. Let us look at the potential side effects analyzed by third-party vendors.

"Our users would not be able to target FireFox ... and would probably use another browser."

"Practically I wouldn't anticipate any side effects, although theoretically it could affect the stability of Firefox, our software products or even the whole operating system."

"It will break our protections and cause frauds associated with the removed protections, can crash our browser components and probably Firefox as well."

"This will break our ability to scan HTTPS URLs for malware/phishing links."

Again, according to the respondents, blocking DLLs would not be the best way to resolve DLL bugs. Before doing this, host software vendors should be aware of any potential and serious side effects. This is the reason why in Mozilla's blocklisting policy the blocks are always applied after careful consideration and testing, and also why outright blocking might pose problems if not handled well.

Enforcing a whitelist (Q13, Q14).
Some host software vendors are considering to put the DLLs into the whitelist if the DLL software is also using the standard extension API.

On the one hand, some third-party vendors agreed that such whitelist bonus is an incentive for them to adopt the extension API, but these vendors have already considered/started to migrate to the API. *"Yes ... (the whitelist bonus will be) along with the ability to enforce addons in certain scenarios." "We already adapting to the best of our ability the WebExtension API. We also moved to that methods on other browsers."*

On the other hand, some others are not interested in this bonus because *"I am unaware that we can extract audio from a browser using this API"* and *"The WebExtensions API has simply different use cases than the ones we are currently implementing. Therefore I don't think it makes sense to mix that up"*. The benefit of the whitelist bonus still needs to be verified in the future.

Some participants agreed that the existence of a whitelist will be an incentive for them to do more QA. For the two participant who did not agree,

one thought that their *"current QA processes are sufficient"*. The other one absolutely denied potential benefits from the whitelist: *"A whitelist approach is inferior as it holds back the extension ecosystem overall, in my opinion. A proactive approach providing extensive and frequently updated test scenario framework support covering known problematic techniques is superior."* Therefore, we also need future evidences to answer this question.

Bypassing the whitelist (Q15).
About our last question, no participant plans to circumvent the whitelist, even for the vendors who insist to use DLL injection.

"No, because it won't be a long term solution."

"We would not for legal reasons. We do not circumnavigate anything." *"This question is quite hypothetical right now. Likely we would respect Firefox's policy and not try to actively circumvent anything like this by technical means, but instead we may notify our users about this and suggest to move to another browser. Depending on the exact method of implementation, it's questionable if we'd be affected by such a whitelist though."*

"If we will be on the white list, why should we (circumvent it)?"

However, we do not know whether malware producers would try to circumvent the whitelist (our guess is that they probably would), since we are not able to contact any of them. Also, we cannot be sure that the answers to this question are actually honest, given that circumventing the whitelist might be illegal and would be a direct challenge against Mozilla. Clearing out this doubt will be a part of our future work, once we collect enough field data on the whitelist.

Completely blocking DLL injection might not be the best strategy to reduce bugs caused by third-party software. Instead, host software vendors should strengthen their collaboration and communication with third-party vendors, and build a publicly accessible validation test framework. To attract third-party vendors to use the standard extension API, host software should improve the API's reliability and functionality (i.e., available functions). A whitelist might be beneficial, but more empirical evidences are needed to support this claim.

5 Discussion

In a software ecosystem, pursuing user satisfaction is one of the most important goals for both host and third-party vendors. However, to achieve this goal, some host and guest vendors are taking conflicting measures. In the previous section, we have observed that, on the one hand, some host vendors are (even completely) blocking third-party software added through DLL injection and are suggesting users to uninstall unreliable software. On the other hand, some third-party vendors are not willing to adopt host vendors' advices and new

solutions because once their extensions cannot work with the host software, they claim that they will suggest users to migrate to another host. We believe that in an ecosystem, host and third-party vendors should not consider their benefit as a zero-sum game, but a win-win game. To satisfy and hold their common users, host and third-party vendors should strengthen their collaboration along all aspects of the development of the ecosystem, including (but not limited to) testing, bug fixing, feature introducing, and API evolution.

In this work, we choose DLL injection as subject because some host software vendors realize that this technique often caused bugs (even crashes) and can be exploited by attackers. However, besides DLL injection and a standard extension API, there are other ways to add third party code into another software, such as Flash. As a resource consuming and outdated technique, Flash has been made "click-to-play" in both Firefox and Chrome since 2017, and will be completely blocked in all browsers by 2019 (2020 for Firefox ESR), so we do not study it in our work. Comparing the reliability among different extension techniques will be a part of our future work.

6 Threats to Validity

Construct validity threats are concerned with the relationship between theory and observation. Studying DLL injection bugs in an ecosystem is a new research topic. As far as we know, there has not been a theory behind this. However, before conducting the empirical study, we learnt some assumptions through our contact with Mozilla developers, but observed opposing results. For example, some Mozilla developers thought that the WebExtensions API can fulfill most of the purposes. They guessed that some third-party vendors are not willing to migrate to the API because the vendors do not want to spend time to modify their existing code. However, multiple of our survey participants indicated that their purposes cannot be satisfied by the current WebExtensions API. Moreover, to reduce DLL injection bugs, host vendors are taking measures, *e.g.*, blocking DLL injection, suggesting users to uninstall "unreliable" extensions. By analyzing feedback from third-party vendors, we realize that many of these measures could be harmful for end users and even the host vendors themselves.

Internal validity threats concern factors that may affect a dependent variable and were not considered in the study. Some of our observations derived from the 12 survey responses. Although these responses cannot represent all third-party vendors' opinions, they provided us valuable information to understand the root causes of the DLL injection bugs and to propose potential solutions to reduce the bugs occurrence. The most important reason is that such information cannot be discovered from any open source repositories, such as Mozilla bug reports, crash reports, or commit logs. Besides, we studied all the 103 DLL injection bugs reported during the past two years. These bugs were caused by 58 different vendors, among which, 44 vendors were contacted. 12

survey participants represent a 21% coverage of all subject third-party vendors and 27% survey response rate (which is higher than the average response rate in questionnaire-based software engineering studies, *i.e.*, 5%, according to Singer et al.'s finding [30]).

Conclusion validity threats concern the relationship between the treatments and the outcome. When investigating the characteristics of the DLL injection bugs, we manually classified DLL bugs into different categories. To reduce any biases during this process, we did not predefine any category. For each characteristic, two of the authors independently made their classification before comparing their results and resolving each of the discrepancies. Despite this, we cannot guarantee a 100% accuracy on our classification result. To help future studies validate our result, we share our dataset online at: `https://github.com/swatlab/dll_injection`. Some of the important observations are based on the survey responses. To reduce any possible biases, besides our discussion and analyses, we cited participants original answers. Readers can use this information to validate our conclusion and discover more insight. When compiling the survey responses, we hid some details due to privacy reasons. For example, we did not make a table showing which participant answered which question because this way may disclose information that participants do not wish to publish. In the survey, we only use open questions, because first, our subject problem has not been empirically studied before, *i.e.*, there is no reference to help us predefine options for the answers. Second, predefined answers may bias and limit participants' judgement. In this work, we are open to receive any unexpected ideas that can lead us to a better understanding of the subject problem.

External validity threats are concerned with the generalizability of our results. In this work, we choose Mozilla Firefox as subject ecosystem because other equivalent ecosystems either lack relevant data or will try to completely block DLL injection soon (*e.g.*, Chrome). We believe that Firefox is a large-scale representative ecosystem, which contains various and diverse DLL software (refer to the software types discussed in **RQ1**). In addition, Firefox possesses some public resources that we cannot benefit from other host vendors, such as bug reports, where we can also often see decision processes in play, and third-party vendors' contacts. Nevertheless, the results and conclusion of our work may not be generalized to other environments. Future studies are required to validate and complement our findings. Researchers can also use our shared dataset to replicate this study: `https://github.com/swatlab/dll_injection`.

7 Related Work

7.1 Software Ecosystems

When a software organization increasingly allows other software to join and extend its software platform, an ecosystem is gradually formed. Many software

organizations have realized that either creating or joining into such an ecosystem can be beneficial because they no longer have to produce an entire system but only need to work for a part of it. Recently, we have seen an increase in the number of software ecosystems and the number of research studies that have focused on them. Bosch [4] observed the emerging trend of the transition from traditional software product lines to software ecosystems and proposed actions required for this transition. He also discussed the implications of adopting a software ecosystem approach on the way organizations build software. Hanssen [13] conducted an empirical study of the CSoft system, which transitioned from a closed and plan-driven approach towards an ecosystem. He observed that transitioning to a software ecosystem improved the cross organizational collaboration and the development of a shared value (*i.e.*, technology and business) in the collaboration. Jansen et al. [17] discussed the challenges of software ecosystems at the levels of software ecosystems themselves, software supply network, and software vendors. This early work provided a guideline for software vendors to make their software adaptable to new business models and new markets, and help them to choose appropriate strategy to succeed in an ecosystem. Later on, Van Den Berk et al. [32] built models to quantitatively assess the status of a software ecosystem as well as the success of decisions taken by the host vendors in the past.

Researchers have also empirically studied various popular open source ecosystems, including Linux kernel (*e.g.*, [31]), Debian distribution (*e.g.*, [10, 12]), Eclipse (*e.g.*, [34, 5]), and R (*e.g.*, [11]) ecosystems. The host software in these ecosystems are respectively operating system, integrated development environment, and mathematical software. However, as far as we know, there is no previous study that empirically investigates a browser-based open source ecosystem (*e.g.*, Firefox, Chrome). Although Liu et al. [20] studied the extension security model of Chrome and Karim et al. [18] studied the Jetpack Extension Framework of Mozilla, their research focused on the extension techniques rather than on the ecosystems. We contribute to filling this gap by conducting an empirical study of DLL injection bugs in the Firefox ecosystem. Another difference between our work and these previous works [20, 18] is that DLL injection is completely arbitrary, *i.e.*, a third-party software can execute whatever it requires; while the extension API can constrain third-party software's behaviour.

7.2 DLL Injection

DLL injection is one of the popular ways to insert code into other software. It can force a process to load external code in a manner that the author of the process does not anticipate or intend. Leveraging the DLL injection technique, Andersson et al. [1] proposed a framework to detect code injection attacks [35]. Lam et al. [19] proposed an approach that uses DLL injection to isolate the execution of the incoming email attachments and web documents on a physically separate machine rather than on the user machine. Their approach can help

reduce the risk that user machines are attacked. Berdajs et al. [3] analyzed the limitations of multiple existing DLL injection techniques (including CreateRemoteThread, proxy DLL, Windows hooks, using a debugger, and reflective injection) and introduced a new approach that combines DLL injection and API hooking (a technique by which we can modify the behaviour and flow of an API call [15]). The improved approach can inject code even when the application is not fully initialized.

As DLL injection allows a program to inject arbitrary code into arbitrary processes [36], malware producers can also take advantage of this technique to exploit computers. Jang et al. [16] proposed an approach to help identify malicious DLLs in Windows. Windows maintains a list of all loaded modules, including DLL modules. Some software checks this list to detect DLLs injected from another process and take corresponding measures, *e.g.*, block it if a DLL is suspicious. However, an approach called Reflective injection [9] can inject DLLs in a stealthy manner, which increases the difficulty of detecting suspicious DLLs.

Like a double-edged sword, DLL injection is a useful (even indispensable) programming technique, but can also cause severe damages due to its arbitrary nature. To the best of our knowledge, we are not aware of any existing work that empirically studied the root causes and counterplans of the bugs or defects caused by DLL injection. Particularly, in a software ecosystem, this kind of bugs can hardly be predicted but can affect a large number of users. To help software practitioners understand the root causes of DLL injection bugs and propose solutions to reduce them, we conduct this case study on the Firefox ecosystem.

8 Conclusion

In a software ecosystem, DLL injection allows third-party software to forcibly load arbitrary code into the host software. This technique may cause severe problems, such as crashes and hangs. In this paper, we quantitatively and qualitatively studied DLL injection bugs in the Firefox ecosystem. We found that: most of the subject bugs (93 bugs, *i.e.*, 90.3%) led to crashes, and 57 (55.3%) of them were caused by antivirus software (**RQ1**). Various DLL injection mechanisms were applied by third-party vendors; the triggers of the bugs can be engine errors, compiler/runtime incompatibility, or version incompatibility between the host and third-party software (**RQ2**). Completely banning DLL injection might not be the best strategy because some software (*e.g.*, antivirus) relies on this technique. Collaboration between host and third-party software vendors could help reduce DLL injection bugs; host software vendors should extend the features of the extension API (as a safer alternative of adding functionalities onto the host software) and build a publicly accessible validation test framework (**RQ3**). In the future, we plan to investigate whether the upcoming whitelist can further help reduce DLL injection bugs.

References

1. Andersson S, Clark A, Mohay G, Schatz B, Zimmermann J (2005) A framework for detecting network-based code injection attacks targeting windows and unix. In: Computer Security Applications Conference, 21st Annual, IEEE, pp 10–pp
2. AppInitDLLs (2018) AppInit_DLLs in Windows 7 and Windows Server 2008 R2. `https://msdn.microsoft.com/en-us/library/windows/desktop/dd744762(v=vs.85).aspx`, online; Accessed April 12th, 2018
3. Berdajs J, Bosnić Z (2010) Extending applications using an advanced approach to dll injection and api hooking. Software: Practice and Experience 40(7):567–584
4. Bosch J (2009) From software product lines to software ecosystems. In: Proceedings of the 13th international software product line conference, Carnegie Mellon University, pp 111–119
5. Businge J, Serebrenik A, van den Brand M (2010) An empirical study of the evolution of eclipse third-party plug-ins. In: Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE), ACM, pp 63–72
6. Castelluccio M, An L, Khomh F (2018) An empirical study of patch uplift in rapid release development pipelines. Springer, pp 1–37
7. Chromium Blog (2017) Reducing Chrome crashes caused by third-party software. `https://web.archive.org/web/20180728201546/https://blog.chromium.org/2017/11/reducing-chrome-crashes-caused-by-third.html`, online; Accessed August 1st, 2018
8. CreateRemoteThread (2018) CreateRemoteThread function. `https://msdn.microsoft.com/en-us/library/windows/desktop/ms682437(v=vs.85).aspx`, online; Accessed April 12th, 2018
9. Fewer S (2008) Reflective dll injection. Harmony Security, Version 1
10. German DM, Gonzalez-Barahona JM, Robles G (2007) A model to understand the building and running inter-dependencies of software. In: Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on, IEEE, pp 140–149
11. German DM, Adams B, Hassan AE (2013) The evolution of the r software ecosystem. In: Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on, IEEE, pp 243–252
12. Gonzalez-Barahona JM, Robles G, Michlmayr M, Amor JJ, German DM (2009) Macro-level software evolution: a case study of a large software compilation. Empirical Software Engineering 14(3):262–285
13. Hanssen GK (2012) A longitudinal case study of an emerging software ecosystem: Implications for practice and theory. Journal of Systems and Software 85(7):1455–1466
14. Hollander M, Wolfe DA, Chicken E (2013) Nonparametric statistical methods, 3rd edn. John Wiley & Sons
15. InfoSec Institute (2014) API hooking. `http://resources.infosecinstitute.com/api-hooking`, online; Accessed April 12th,

2018
16. Jang M, Kim H, Yun Y (2007) Detection of dll inserted by windows malicious code. In: Convergence Information Technology, 2007. International Conference on, IEEE, pp 1059–1064
17. Jansen S, Finkelstein A, Brinkkemper S (2009) A sense of community: A research agenda for software ecosystems. In: Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on, IEEE, pp 187–190
18. Karim R, Dhawan M, Ganapathy V, Shan Cc (2012) An analysis of the mozilla jetpack extension framework. In: European Conference on Object-Oriented Programming, Springer, pp 333–355
19. Lam Lc, Yu Y, Chiueh Tc (2006) Secure mobile code execution service. In: LISA, pp 53–62
20. Liu L, Zhang X, Yan G, Chen S, et al (2012) Chrome extensions: Threat analysis and countermeasures. In: NDSS
21. LoadLibrary (2018) LoadLibrary function. `https://msdn.microsoft.com/en-us/library/windows/desktop/ms684175(v=vs.85).aspx`, online; Accessed April 12th, 2018
22. Mozilla Add-ons Blog (2018) Advantages of WebExtensions for Developers. `https://blog.mozilla.org/addons/2016/03/14/webextensons-whats-in-it-for-developers/`, online; Accessed April 16th, 2018
23. Mozilla Add-ons Blog (2018) Preventing Add-ons And Third-party Software From Loading DLLs Into Firefox. `https://blog.mozilla.org/addons/2017/01/24/preventing-add-ons-third-party-software-from-loading-dlls-into-firefox/`, online; Accessed November 11th, 2018
24. Mozilla Add-ons Blog (2018) The Future of Developing Firefox Add-ons. `https://blog.mozilla.org/addons/2015/08/21/the-future-of-developing-firefox-add-ons/`, online; Accessed April 16th, 2018
25. Mozilla Wiki (2017) WebExtensions API. `https://wiki.mozilla.org/WebExtensions`, online; Accessed April 12th, 2018
26. Mozilla Wiki (2018) Mozilla Release Management Tracking Rules. `https://wiki.mozilla.org/Release_Management/Release_Process`, online; Accessed March 28th, 2018
27. Mozilla Wiki (2018) Mozilla's blocklisting policy. `https://wiki.mozilla.org/Blocklisting`, online; Accessed April 16th, 2018
28. SetWindowsHookEx (2018) SetWindowsHookEx function. `https://msdn.microsoft.com/en-us/library/windows/desktop/ms644990(v=vs.85).aspx`, online; Accessed April 12th, 2018
29. SetWinEventHook (2018) SetWinEventHook function. `https://msdn.microsoft.com/en-us/library/windows/desktop/dd373640(v=vs.85).aspx`, online; Accessed April 12th, 2018
30. Singer J, Sim SE, Lethbridge TC (2008) Software engineering data collection for field studies. In: Guide to Advanced Empirical Software Engineering, Springer, pp 9–34

31. Tu Q, et al (2000) Evolution in open source software: A case study. In: Software Maintenance, 2000. Proceedings. International Conference on, IEEE, pp 131–142
32. Van Den Berk I, Jansen S, Luinenburg L (2010) Software ecosystems: a software ecosystem strategy assessment model. In: Proceedings of the Fourth European Conference on Software Architecture: Companion Volume, ACM, pp 127–134
33. WebExtensions (2017) Bugzilla@Mozilla. https://bugzilla.mozilla.org, online; Accessed April 12th, 2018
34. Wermelinger M, Yu Y (2008) Analyzing the evolution of eclipse plugins. In: Proceedings of the 2008 international working conference on Mining software repositories, ACM, pp 133–136
35. Wikipedia (2018) Code injection. https://en.wikipedia.org/wiki/Code_injection, online; Accessed April 12th, 2018
36. Wikipedia (2018) DLL injection. https://en.wikipedia.org/wiki/DLL_injection, online; Accessed April 12th, 2018
37. WindowsDataTypes (2018) Windows Data Types. https://msdn.microsoft.com/en-us/library/windows/desktop/aa383751(v=vs.85).aspx, online; Accessed April 12th, 2018