Unveiling the critical attack path for implanting backdoors in supply chains: Practical experience from XZ

Mario Lins$^{1\,[0000-0003-1713-3347]}$, René Mayrhofer$^{1\,[0000-0003-1566-4646]}$, and Michael Roland$^{1\,[0000-0003-4675-0539]}$

Johannes Kepler University Linz, Institute of Networks and Security, Linz, Austria
`{lins,rm,roland}@ins.jku.at`

Abstract. Recently, a previously unseen supply chain attack due to a backdoor in XZ Utils has been identified by A. Freund. This particular attack leverages highly sophisticated attack techniques, starting with social engineering attacks against the open source community up to implanting a backdoor in obfuscated binary blobs. This malware is designed to empower the attacker(s) to remotely run commands on vulnerable servers utilizing SSH evading authentication. A reverse dependency analysis revealed that the affected library is used by almost 30,000 packages in Debian and Ubuntu—the same order of magnitude as the GNU standard C library (`glibc`/`libc6`), a dependency for roughly 50,000 packages on these systems. This fact highlights the severity of such supply chain attacks and raises concerns about further backdoored packages still undetected in the wild. This paper identifies the critical attack path for successful implantation of such a backdoor and abstracts general key takeaways for future detection and mitigation of similar attacks. We also present *SketchyCrawler*, an open-source tool prototype designed to illustrate crawling repositories to reveal 'sketchy' signs of potential backdoor implantation attempts.

Keywords: Open source · Backdoor · Remote code execution · Supply chain attack

1 Introduction

We were extraordinarily lucky this time. On Friday, March 29th, 2024, Andres Freund ignited newspapers, social media channels, and security experts around the world when he revealed [13] a critical supply chain attack that introduces a backdoor in the widely used XZ Utils (CVE-2024-3094). He began investigating the XZ Utils package after noticing a slight 500 ms delay, an increase in CPU consumption, and some *valgrind* errors during test cases utilizing a remote SSH server. The affected package, XZ Utils, is a set of open source components, including `xz` and `lzma`, used for data compression. An attacker, using this backdoor, is able to execute arbitrary code remotely on vulnerable machines utilizing

SSH without prior authentication. The seriousness of this security incident is underscored by the widespread usage of XZ Utils on a significant number of Linux and other Unix-based systems, and the resulting CVSS rating of 10.0. Both the level of organizational attack complexity and the high code sophistication that were applied to compromise a widely distributed open source project with the aim of attacking a specific supply chain (and various other aspects) make this security incident particularly worthy of in-depth study. It highlights the need for careful consideration, both in addressing similar attacks in the future and in detecting potential active backdoors for which we were not yet as lucky to have become aware of. In this paper, we address relevant practical experiences with a focus on supply chain attacks by making the following theoretical and practical contributions:

- We analyze the malicious version of the XZ Utils package to identify and emphasize properties that could be relevant to other potential targets of interest for performing such a supply chain attack.
- We identify the essential stages of the critical attack path for implanting and activating a backdoor in an open source project.
- We generalize from this particular case to identify key takeaways as categories of important signals—both technical and social—to consider within the open source community in order to prevent similar attacks in the future.
- We implement *SketchyCrawler* as a first prototype to identify GitHub repositories with sketchy signs based on our evaluated takeaways.

2 Preliminaries

Both the infected package XZ Utils and the attack itself leverage various tools, functionalities, and well-established techniques. This section introduces them as essential preliminaries relevant to describe the critical attack path.

2.1 XZ Utils

XZ Utils is a widely distributed set of open source packages used for lossless data compression on Unix-like operating systems, including Linux. Beside supporting compression with the *xz file format*, XZ Utils also supports the legacy *lzma* format [41]. Because of its high compression ratio, in the last decade it has been included in many basic system components such as archival/compression tools, package installers like `dpkg` and `rpm`, the squashfs filesystem,[1] and even the Linux kernel itself for decompression of the kernel image and initramfs.[2]

2.2 Distribution systems

Linux distributions, such as Debian, contain thousands of packages that work together to ensure seamless functionality. These packages are often developed by

[1] `https://docs.kernel.org/filesystems/squashfs.html`
[2] `https://docs.kernel.org/staging/xz.html`

multiple open source contributors and distributed either as already pre-existing components of the OS through package managers, like Advanced Package Tool (APT), or Dandified YUM (DNF). To include an open source package in their distribution system, it is a common practice to provide the related source code files. According to e.g., the "UpstreamGuide" of Debian [11], it is considered best practice to provide only source code files that make a rebuild possible. However, they also acknowledge that the rule "rebuild everything" is applied inconsistently, particularly for packages utilizing `autoconf` and `automake`. Thus, the current toolchain of Debian requires source code files to be submitted within a tarball. As a result, it is quite common to find a source tarball on the release pages of various open source projects available on VCS, like GitHub. Some open source projects also include pre-built binaries on these release pages. However, these are typically not used by package distribution systems, which build the final artifact from source instead. Another particularly relevant aspect of such package distribution systems is that the upstream sources are also expected to include test suites as part of the source tarball, which will be executed as part of the *Continuous Integration* (CI) pipeline to verify the intended behavior.

2.3 GNU M4

GNU `m4` [14] is used as an implementation of a macro processor standardized by POSIX including some additional built-in functions, such as running shell commands. Additionally, it is used by packages built with GNU `autoconf` for generating the `configure` scripts that customize the Makefile for compilation.

2.4 GNU indirect function support (IFUNC)

Indirect function support (IFUNC) [30] is a feature of the GNU toolchain allowing developers to define multiple function implementations and to choose one of them at runtime. For selecting the proper implementation, the developer uses a resolver function. The dynamic loader calls the resolver function during startup and loads the corresponding implementation.

2.5 Google OSS-Fuzz

Google's OSS-Fuzz [16] is a continuous fuzzing tool for open source software to detect programming errors, like buffer overflows. According to the documentation [15] only projects with a significant user base and/or critical projects are included in the fuzzing process. To include a new project, it is necessary to open a pull request with information about the project (e.g., repository URL, primary contact).

3 Preliminary considerations for target selection

First and foremost, XZ Utils is considered a well-known and widely distributed package used by various other packages and is shipped as an integral component of many operating systems, such as Debian. Due to its continuous development and long-standing presence, it has been highly trusted within the open source community for many years. We also believe that the attacker(s) may have chosen the XZ Utils project because they were looking for an open source project with a single maintainer who could become overwhelmed when increasing the workload. Another key observation relates to evasion, emphasizing that the actual target—servers utilizing OpenSSH—do not have a direct dependency on XZ Utils. Instead, some distributions utilize `systemd` in combination with `libsystemd` to manage services like OpenSSH. Notably, `libsystemd` depends on the library `liblzma`, which is part of the XZ Utils project. Considering further evasion and obfuscation signs with regards to the XZ project, the test files may play a crucial role as well. These files, being binary blobs, are not easily readable yet appear legitimate within the context of a data compression utility.

We outline the (supposed) key reasons for choosing an open source project like XZ Utils to strategically plan such a supply chain attack:

– Well-known package with sufficiently long history
– Trusted within the open source community
– Widely distributed package to increase attack surface
– Single maintainer, easily overwhelmed
– Actual target does not directly depend on infected package, making detection harder
– Using binary blobs that appear legitimate, but can be used to implant malicious code

Another important consideration when selecting the ideal target for such an attack is to determine how widely distributed the infected XZ Utils package actually is. To address this, we conduct a recursive reverse dependency analysis of the affected `liblzma` and a set of other libraries for comparison. We chose that

Table 1. Results from a reverse dependency analysis[*] for a set of selected libraries on 2024-09-15

	reverse-depends count on	
Library	Ubuntu 24.04	Debian 12.7
libc6	51,209	45,344
libzstd1	32,249	27,548
liblzma5	**29,133**	**27,129**
libssl3(t64)	24,361	23,178
libbz2-1.0	24,871	21,246
liblz4-1	15,170	12,840
libsystemd0	14,579	12,040
libjpeg(62)-turbo(8)	11,102	11,124
libssh2-(4|1)	6,082	5,484
libcurl4(t64)	3,125	3,832

[*]Using:
```
apt-rdepends -r <library-package> | \
       sed 's/^Reverse Depends: //;s/ (.*)$//' | sort -u | wc -l
```

set as an exemplary representation of libraries (a) providing similar functionality (i.e., other compression libraries), (b) providing essential functionality that we expected to be widely used in general (i.e., GNU standard C library) and particularly in applications with network communication (e.g., OpenSSL's `libssl3`), (c) a randomly-picked image processing library (as an example for (in-)famous bugs leading to exploitable RCE on both consumer devices and server-side applications), and (d) libraries related to the exploitation target chosen in the XZ Utils incident. Table 1 summarizes the results of our analysis. As shown, `liblzma` is used by nearly 30,000 other packages. Compared to the ubiquitous glibc (`libc6`), we see the same order of magnitude. When compared to another popular, and potentially even more well-known tool, such as `curl`, `liblzma`, while supposedly more obscure to both users and administrators, shows a usage rate that is 10 times higher. These facts underscore the significant severity in having malicious code (e.g., a backdoor) hidden in a library with such widespread distribution.

4 Attack analysis

This section describes relevant stages, including both human/social and technical aspects, that lead to the compromise of the XZ C library by incorporating a backdoor. We start with an analysis of the currently known facts [1,4,7,8,10,13] around the security incident and derive a critical attack path as shown in Fig. 1 by categorizing the available information (including the temporal sequence of the events) into individual stages.

4.1 Stage 1: Building trust

The XZ security incident did not begin with fancy hacking tools or with sophisticated code; it all started with a hacking technique known as social engineering, which prays on humans' emotions, trust, and pressure. In case of XZ, especially pressure plays a crucial role in the social engineering attack targeting the open source community. This section summarizes insights into the social engineering tactics employed by the attacker(s) to gain necessary privileges on the repository.

In 2005 [42], a small group including Lasse Collin, started work on a project called LZMA Utils, which was later renamed XZ Utils. In 2021, "Jia Tan" (presumably a pseudonym) started to support the XZ Utils project by regularly contributing code improvements via the XZ developers mailing list. At the time

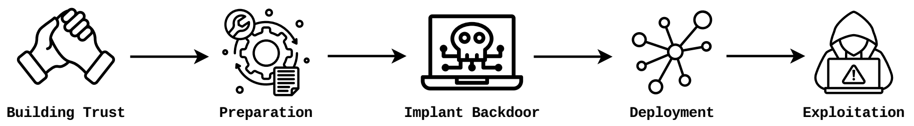


Fig. 1. Critical Attack Path

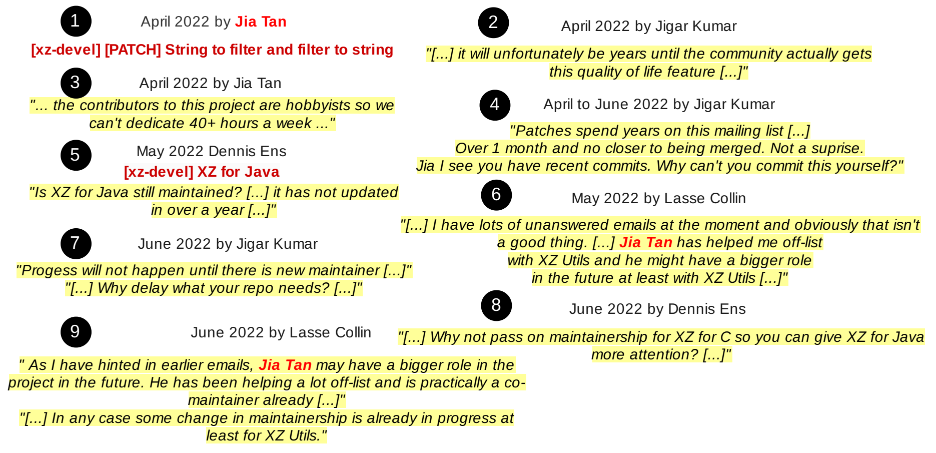
of writing this paper, we are not aware of any malicious content in the first contributions of Jia Tan. Apparently, Jia Tan seems to be a friendly, interested, and supportive contributor of the open source project. However, some years later it became clear that Jia Tan had different intentions with the XZ Utils project. It appears that the initial intention of Jia Tan was to build up trust within the open source project, which plays a significant role with regards to the upcoming attack path. The first suspicious commit [22] involving Jia Tan under the username "JiaT75" dates back to November 2021. The commit was done within another project called "libarchive" and replaced the `safe_fprintf` function with `fprintf`, the less secure variant of this functionality. This commit has been approved and merged to the main branch of that project without noticing that this might be a suspicious change. At the time of writing this paper, it remains uncertain whether this commit is relevant to the XZ backdoor case.

A few months later, in April 2022, user JiaT75 tried to submit another patch via the XZ developers mailing list [37]. According to the conversation log, there was suddenly another pseudonym, called "Jigar Kumar" involved, who describes this patch as "*quality of life feature*" [27] and complains about the slow release schedule of the project maintainer. As one of the initial responses to this complaint, Jia Tan took on the role of a supportive, friendly, and loyal defender of the original maintainer, Lasse Collin, pointing out that the contributors are hobbyists and that can't dedicate 40+ hours a week. The conversation continued with putting more and more pressure on the original maintainer Lasse Collin. Meanwhile, in a separate project focused on XZ for Java, a new participant, Dennis Ens, enters the scene. This pseudonym also began in a similar manner, adding extra pressure on the original maintainer by, for example, inquiring whether the project was still maintained. In this particular conversation, Lasse Collin mentioned that the project is an unpaid hobby and that he is already in contact with Jia Tan who "will have a bigger role in the future". We detected the first commit [23] solely merged by Jia Tan from December 2022. Thus, we infer that Jia Tan obtained the required permissions for the repository and may have been promoted to co-maintainer of the project.

Based on these conversations and the similarities between these different types of individuals, we assume that the attacker(s) have set up these pseudonyms, commonly referred to as "sock puppet" accounts. More insights on the conversations can be found in Fig. 2. We explicitly note that attribution, i.e., trying to estimate the real source of these (supposedly coordinated) pseudonyms and tracing back to an attacking organization and/or geo-political region is outside the scope of this paper.

4.2 Stage 2: Preparation

In the second stage of the critical attack path we outline the various actions taken by the attacker(s) to prepare for the actual supply chain attack. A key aspect of this stage is that all their actions and steps remain deniable. Thus, if any malicious activity had been detected at this stage, the attacker would still have a valid reason to deny any malicious intent.



1 April 2022 by Jia Tan
[xz-devel] [PATCH] String to filter and filter to string

2 April 2022 by Jigar Kumar
"[...] it will unfortunately be years until the community actually gets this quality of life feature [...]"

3 April 2022 by Jia Tan
"... the contributors to this project are hobbyists so we can't dedicate 40+ hours a week ..."

4 April to June 2022 by Jigar Kumar
"Patches spend years on this mailing list [...]
Over 1 month and no closer to being merged. Not a suprise.
Jia I see you have recent commits. Why can't you commit this yourself?"

5 May 2022 Dennis Ens
[xz-devel] XZ for Java
"Is XZ for Java still maintained? [...] it has not updated in over a year [...]"

6 May 2022 by Lasse Collin
"[...] I have lots of unanswered emails at the moment and obviously that isn't a good thing. [...] Jia Tan has helped me off-list with XZ Utils and he might have a bigger role in the future at least with XZ Utils [...]"

7 June 2022 by Jigar Kumar
"Progess will not happen until there is new maintainer [...]"
"[...] Why delay what your repo needs? [...]"

8 June 2022 by Dennis Ens
"[...] Why not pass on maintainership for XZ for C so you can give XZ for Java more attention? [...]"

9 June 2022 by Lasse Collin
" As I have hinted in earlier emails, Jia Tan may have a bigger role in the project in the future. He has been helping a lot off-list and is practically a co-maintainer already [...]"
"[...] In any case some change in maintainership is already in progress at least for XZ Utils."

Fig. 2. Conversation highlights **(1)** Jia Tan wants to provide a patch via mailing list. **(2)** First signs of pressure by Jigar Kumar. **(3)** Support from Jia Tan. **(4)** Increasing pressure from Jigar Kumar. **(5)** Pressure from another pseudonym, Dennis Ens, on a different project. **(6)** First signs of being overwhelmed by Lasse Collin. **(7)** Attacker(s) want to become co-maintainer. **(8)** Again pressure to pass maintainership to attacker(s). **(9)** Finally, the attacker(s) were successful.

In March 2023, the primary contact mail for a component in the testing infrastructure, Google OSS-Fuzz, was changed from `lasse.collin@tukaani.org` to `jiat0218@gmail.com` by opening a merge request on OZZ-Fuzz. That change of contact was manually approved by Lasse Collin as the original maintainer, confirming that Jia Tan is an official co-maintainer of the XZ project [25]. After some months, a new contributor called “Hans Jansen” provided patches to the XZ Utils project that use the GNU indirect function (IFUNC) feature [19, 20]. After some reworks and discussions, Jia Tan merged that functionality to the repository. Only later, it turned out that these patches are essential for the attack, because IFUNC is the function hooking mechanism eventually used by the backdoor code. However, the use of IFUNC cannot be fuzzed by OSS-Fuzz, due to compatibility issues. We assume that the adversary expected errors resulting from fuzzing with OSS-Fuzz after enabling IFUNC. Consequently, Jia Tan opened a pull request [24] within the OSS-Fuzz project, to disable IFUNC for the fuzzed XZ Utils build. In this request, Jia Tan mentioned that IFUNC is not compatible with “`-fsanitize=address`” that is also known as *AddressSanitizer* (ASan) and used to detect memory errors. The OSS-Fuzz repository incorporates a GitHub action to automatically label pull requests as “ready to merge” if they originate from the maintainer of a fuzzed project by verifying that the committer has a contribution history for that project and is one of the responsible project contacts. As both applied to Jia Tan, the pull request was labeled as *ready to merge* by the bot, and then quickly approved and merged into OSS-Fuzz.

```
1  gl_[$1]_config='sed \"r\n\" $gl_am_configmake |
   eval $gl_path_map |
   $gl_[$1]_prefix -d 2>/dev/null'

2  gl_am_configmake=`grep - aErls "#{4}[[:alnum:]]{5}#{4}$"
   $srcdir/ 2>/dev/null`

3  gl_path_map='tr "\t \-_" " \t_\-"'

4  gl_[$1]_prefix=`echo $gl_am_configmake | sed "s/.*\.//g"`

5  cat ./tests/files/bad-3-corrupt_lzma2.xz |
   tr "\t \-_" " \t_\-" | xz -d
```

Fig. 3. Obfuscated command hidden in `build-to-host.m4` file. **(1)** Obfuscated command including three variables: `gl_am_configmake`, `gl_path_map`, `gl_[$1]_prefix`. The `sed` command is actually a hidden `cat` command. **(2)** Extracts the relative path of the `bad-3-corrupt_lzma2.xz` test file. **(3)** `tr` command to uncorrupt the content of the test file. **(4)** Echoes the relative path of the test file and removes characters except the file ending: xz. **(5)** Unveiled command that fetches content of test file, pipes it into `tr` command to uncorrupt it, and afterwards extracts the hidden shell script by decompressing it with `xz`.

4.3 Stage 3: Implanting backdoor

This stage consolidates the steps required for implanting the backdoor into an open source project, ensuring it stays hidden even in a public repository. Compared to the previous stage, the actions taken by the attacker(s) in this stage are no longer deniable.

On Sunday, February 23rd, 2024, Jia Tan added various lzma test files [38,39] to the project. Later, it turned out that the uploaded test files, called `bad-3-corrupt_lzma2.xz` and `good-large_compressed.lzma` contained the binary backdoor and some other obfuscated code necessary to inject the backdoor code. At this stage, the hidden backdoor is already part of the project, but is not yet deployed. Another essential preparation step was to add the file `build-to-host.m4` to the `.gitignore` file of the source code repository so that its source code does not get pushed to the repository. However, a modified version of the `build-to-host.m4` file was still part of the source tarball that gets fetched by the build system to build the final artifact. Comparing a previous version of the `build-to-host.m4` file with that one stored in the source tarball clearly yields some differences. We focus on one of these to show how the attacker(s) were able to obfuscate the malicious parts finally executed on the build machine. Fig. 3 shows the obfuscated command hidden in the `build-to-host.m4` file.

4.4 Stage 4: Deployment

The following day (Monday, February 24th, 2024), Jia Tan released version 5.6.0 of XZ Utils via GitHub. In addition to the auto-generated source archives in

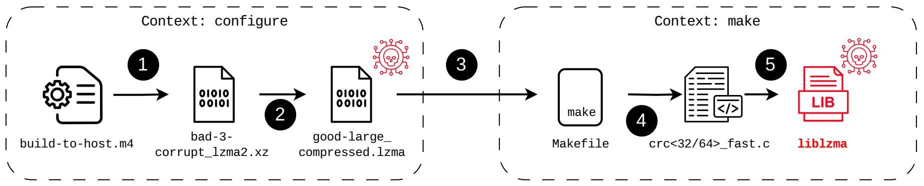


Fig. 4. Abstract overview of the remaining steps to implant the backdoor in the final artifact. The dashed lines are used to visualize the different build contexts. **(1)** The `build-to-host.m4` file initiates the implanting process by "repairing" the `bad-3-corrupt_lzma2.xz` file and executing a script hidden in there. **(2)** This script runs additional commands hidden in another test file called `good-large_compressed.lzma`. This test file is properly compressed with XZ, but contains junk data that gets removed by various operations, such as using the `head` command to get specific bytes. The final result is again piped into shell and gets executed. **(3)** Besides some checks on the environment, the actual binary backdoor also hidden in the `good-large_compressed.lzma` test file, but with another offset, gets extracted into `liblzma_la-crc64-fast.o`. Additionally, the `Makefile` is patched to incorporate the malicious backdoor in the compiled library. **(4)** After some preparation, verification, and cleanup steps, the patched `Makefile` modifies the code of the files `crc64_fast.c` and `crc32_fast.c`. This step is important for the exploitation stage, as the contained function `crc64_resolve` provides the actual ifunc resolver, which is used to call the `_get_cpuid` function that is provided by the backdoor. **(5)** Finally, the backdoor hidden in the `liblzma_la-crc64-fast.o` file including the ifunc resolver gets compiled in the liblzma library.

`.zip` and `.tar.gz` form, XZ Utils releases include four additional, manually uploaded source archives with corresponding signature files. The manually uploaded source archives, which were uploaded to GitHub for this release, also include the modified `build-to-host.m4` file, which is not part of the repository itself to evade potential discovery through the repository change history. The malicious m4 file extracts the backdoor from the test files when building the deb or rpm packages.

The next step was to convince Linux distributions to include the latest, but malicious, release of XZ Utils in their package repositories. For that, the pseudonym Hans Jansen started to contact and pressure some Linux distributions (e.g. Debian [21]). Once the package maintainer upgraded to the new version by fetching the source code tarball, the malicious `build-to-host.m4` file gets integrated into the package build process. The malicious build script then initiates the steps required to extract the backdoor hidden in the test files. After several obfuscated modification, decompression, and execution steps on the malicious test files, as illustrated in Fig. 4, an object file named `liblzma_la-crc64_fast.o`, which contains the backdoor code, is linked into the `liblzma` library via setting the respective pointer which gets resolved by the IFUNC functionality. Additional information can also be found in [10].

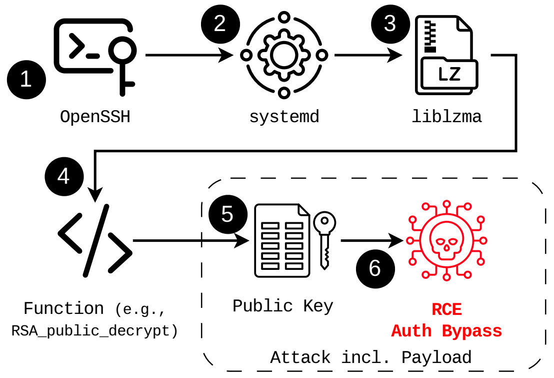


Fig. 5. Illustration of the actual exploitation of the backdoor hidden in XZ Utils. **(1)** The actual targets are vulnerable servers utilizing OpenSSH. The attack gets initiated by providing a specific OpenSSH certificate to these servers. **(2)** OpenSSH does not directly depend on the malicious liblzma library, but is managed on some systems via systemd. **(3)** systemd is used to load the malicious library liblzma. **(4)** The malicious library providing the ifunc resolver gets called before the GOT and PLT table are read-only. This allows the backdoor to rewrite the tables to hijack function calls, such as `RSA_public_decrypt`, which is used by SSH. **(5)** The SSH certificate provided by the attacker contains the payload. **(6)** The payload gets executed so the attacker is able to run commands remotely without prior authentication.

4.5 Stage 5: Exploitation

The final stage is the exploitation stage (Fig. 5) in which the backdoor gets triggered on the infected system. Current investigations show that a pointer of a function used by `RSA_public_decrypt` for OpenSSH (more specifically for any process of the executable located at `/usr/bin/sshd`) is changed to point on a function owned by the backdoor. The malicious function extracts a command from the certificate provided by the SSH client. This is only initiated for a specific private key which makes the backdoor only available to attackers who possess the corresponding key. Afterwards, the command is passed to the `system()` function, which executes it. Thus, an attacker who controls the private SSH key can send arbitrary commands to affected SSH servers which are executed remotely without ever completing the authentication to the SSH server.

5 Takeaways

We now try to generalize from the specific XZ incident and derive more abstract takeaways that we argue are applicable to different open source projects under the category of aspects described in Section 3.

5.1 Signs of social engineering

Lively discussions within an open source project are generally seen as positive and encouraged. However, if there are suddenly more and more messages of offensive or otherwise problematic nature such as increasing pressure on specific contributors, it might be a sign that should be taken seriously. Especially time pressure seems to be effective in this context and could be a sign for social manipulation as part of an attack. Other signs of social engineering are extending or transfer of maintainer status for a project or components, as has happened in this case, but also in many others with malicious intent (e.g., in the various NodeJS and Python cases [3,31,32]). The **first takeaway (T1)** for mitigating comparable future cases is therefore an increased focus on social engineering of project maintainer control.

5.2 Small projects with large reverse dependencies

The XZ Utils project was, in hindsight, an excellent target due to its relatively small code base and clear focus, but with a surprisingly large reverse dependency tree.

This **second takeaway (T2)** is potentially the lowest hanging fruit for future research by prioritizing independent analysis on and adding mitigations to exactly such projects matching this filter criterion—which are far fewer than the total number of packages in most Linux distributions or programming language package/module systems. Conversely, we can derive a **third takeaway (T3)** for critical software projects (such as OpenSSH) to particularly focus on their own dependency tree in terms of such small, potentially vulnerable projects they might depend on. This is not easy if the dependency tree is large or includes dynamic indirections,[3] potentially even crossing language/platform boundaries. Nonetheless, the same filter criterion can be applied to prioritize analysis efforts.

5.3 Potential for bugdoors

Brittle code in the sense of source code or build system level meta code that, structurally or based on bad code quality, makes it easy to make "mistakes" with security-critical consequences, should be analyzed for its potential of introducing *bugdoors*.[4] Ideally, such brittle code should not be merged into projects,

[3] One of the changes after the XZ Utils incident is that `libsystemd` now loads compression libraries through `dlopen`, which makes this indirect dependency even harder to follow (cf. `https://github.com/systemd/systemd/issues/32028` and `https://github.com/systemd/systemd/pull/31550`).

[4] The portmanteau "bugdoor" describes a backdoor—typically implanted in the source code—that can plausibly be argued to have been a genuine bug in the relevant code. It is unclear when and by whom this term was coined, but early sources include an analysis on embedded systems firmware from 2014 [40]. The concept itself is generic and not limited to pure memory safety issues, as formalized (but not explicitly named as potential bugdoor vectors) in the seminal "Weird Machines" work [12].

although that might happen in combination with social engineering as mentioned above. Additionally, unnecessary code or architectural complexity can be a sign of preparing for a future bugdoor. In the case of XZ, existing use of `autoconf` (with a high level of build system complexity with embedded arbitrary code execution on multiple layers) and the (presumably) maliciously introduced use of IFUNC indirection are, in hindsight, good signals for the potential of introducing bugdoors. Especially adding the whole IFUNC infrastructure was technically unnecessary for the existing code base, and should probably have been rejected without a clear architectural need that could not have been handled with simpler code. Unfortunately, the prevalence of both memory-unsafe languages such as C and (most of) C++ and build systems with arbitrary code execution instead of declarative and side-effect free (also called hermetic [36]) build statements makes this the hardest call for action. Detecting the actual bugdoor itself is, by definition, hard, as they are designed to look like benign human mistakes in another reasonable interpretation.

The **fourth takeaway (T4)** is therefore to avoid unnecessary complexity, both in structure/architecture and in actual source or build system meta code. That includes moving to strongly typed, memory safe languages (which do not make it impossible, but arguably harder to write bugdoors) and declarative build systems whenever possible (particularly for the filtered projects mentioned above), but at least avoiding complex indirections and layers of abstractions not strictly required in those legacy code bases that cannot (yet) be switched.

5.4 Signs of obfuscation

The final, **fifth takeaway (T5)** is perhaps the clearest one: be alert to signs of obfuscation, including explicit ones (such as excerpted in Fig. 3) and implicit, unexplained binary blobs—which are much more common in many projects. While binary blobs acting as input for training or used for “error” test cases have their purpose, we should now start to demand reproducibility of all such binary blobs. That is, binary blobs should, e.g., be generated by explainable helper scripts checked into the repository. While there can be no guarantee that such helper scripts are not in themselves bugdoors in the sense of generating malicious, dual-purpose binary blobs, we argue that it lowers the probability of occurrence. Remaining input for (regression) test cases without reproducible generation processes that have, e.g., been submitted by users as part of bug reports, should only be used or executed within tight sandboxes during the build process.

6 SketchyCrawler

We implemented *SketchyCrawler*, an open-source tool prototype used to crawl GitHub repositories to detect sketchy signs for backdoor potential in open source projects. The main objective is to support software development teams in auditing and assessing such projects before they incorporate them in their own software project. Although there are established, well-known practices for mitigating certain risks in the supply chain, many challenges in this emerging threat landscape remain unresolved as demonstrated with the XZ incident. Established practices involve vulnerability scans, Software Composition Analysis (SCA), or manual auditing of the respective component and its source code. While auditing third party components before integrating them in a product is valuable, it is often impractical to (thoroughly) audit every component or update.

Our objective is to demonstrate how software development teams can leverage such a tool to prioritize their (ideally regular) audits and assess third party repositories, following the five key takeaways outlined in Section 5. First, we explain how *SketchyCrawler* addresses these takeaways. Then, we illustrate some of the information gathered by the first prototype. Finally, we discuss relevant deployment considerations including our assessment of the results.

6.1 Implementation

SketchyCrawler is a set of Python and Shell scripts used to crawl GitHub repositories. Currently, users of our tool simply need to provide their crawling targets and some additional information in a CSV file to collect relevant assessment information automatically. This includes the repository URL, owner, repository name, the time range for the analysis, currently trusted maintainers, potential sketchy files and types, and the package name. Below we describe the main features of *SketchyCrawler*:

Signs of social engineering (T1, cf. Section 5.1) Trust is essential not only within the open source community but also from a supply chain perspective when using third party components in software projects. In case of XZ, the primary objective of the adversary was, as outlined in Section 4.1, to build sufficient trust to gain commit permissions within the open source project. Based on this, one feature of *SketchyCrawler* is about detection of new maintainers that are not marked as trustworthy yet. Since auditing every update of the incorporated third party components requires significant effort, we believe that focusing on major changes (e.g., change of maintainers) offers a valuable advantage. Using a third party component requires some sort of trust in the maintainer anyway. However, when the previously trusted maintainers change, it may be a sign for a necessary re-audit of this component to ensure valid and conscious trust assumptions.

Reverse dependency analysis (T2, cf. Section 5.2) *SketchyCrawler* also includes a module for analyzing a package's reverse dependencies required at runtime. The main objective here is to highlight packages with a high number of reverse dependencies as these can increase the attack surface. In case of XZ, the adversaries planted a vulnerability into SSH servers not by compromising OpenSSH directly, but one of its (indirect) dependencies. The reverse dependency analysis is recursive so we also count the dependencies of the dependencies, and so on.

Forward dependency analysis (T3, cf. Section 5.2) The third feature is similar to the previous one, but addresses forward dependencies at runtime and build time. From an adversary point of view, forward dependencies may be interesting for critical software projects (such as OpenSSH) as the attack surface increases for every dependency that could be vulnerable too. We showcase the analysis of build dependencies by leveraging the *cargo tree* command to count the dependencies recursively. However, *SketchyCrawler* currently focuses on forward package dependencies and does not yet detect dependencies that are dynamically loaded during runtime, e.g. through `dlopen`.[5]

Identify bugdoors (T4, cf. Section 5.3) Our fourth takeaway addresses the potential for bugdoors in open source projects. Thus, *SketchyCrawler* incorporates a module that allows defining a list of potential sketchy files and crawls the repository if those files exist. A good example is the `build.rs` file used in various Rust packages; legitimately, this can, e.g., be used to compile third-party C libraries before building the actual Rust package [35], but potentially for attacking the build process or host itself. Rust projects with a `build.rs` file cause Cargo to execute its content before building the Rust package. A report published by the Phylum Research team [33], dedicated to identifying software supply chain attacks, highlights an attack targeting Rust developers. They observed a case where an adversary initially tried to typosquat a package (e.g., name it postgress instead of postgres) containing almost no code. This was followed by a series of small, seemingly non-suspicious code changes. However, at some point the adversary included a `build.rs` file that was used to establish a communication channel to the adversary. This example illustrates why Rust projects using a `build.rs` file may require a closer look.

Additionally, we check whether sketchy files have been deliberately added to the `.gitignore` file. However, this specific crawling feature only highlights files that are not expected in the `.gitignore` file and adds a more critical sign when combined with the results of T5 (e.g., hidden file is part of the source tarball).

Signs of obfuscation (T5, cf. Section 5.4) In this category we try to identify sketchy mime types that may reveal attempts to obfuscate code (e.g., by incorporating binary blobs). A possible example for a mime-type is *application* that is typically used for a binary file rather than a text file. Another indication of obfuscation is the presence of discrepancies between the actual source code repository and the published source tarball. In case of XZ, the adversaries tried to hide the malicious `build-to-host.m4` file by adding it to the `.gitignore` file. However, the source tarball, used to build the actual package, contained the malicious file.

[5] Work is independently underway to improve tooling for these dynamically loaded dependencies to be listed in the ELF header (cf. `https://github.com/systemd/systemd/blob/main/docs/ELF_DLOPEN_METADATA.md`). Future versions of *Sketchy-Crawler* should use this data when more broadly available.

6.2 Evaluation

We evaluate *SketchyCrawler* by crawling GitHub repositories of Debian packages to identify sketchy signs for a potential backdoor. To select our test examples, we use Debian PopCon,[6] a list of the most popular Debian packages. Currently, we support open-source projects hosted on GitHub only, but we believe that with some engineering effort, integrating additional version control systems would be straightforward. We select the top-three packages on Debian PopCon with the most installations that are also available on GitHub. This results in `libpam-modules`, `libblkid1`, and `zlib1g`. Although `liblzma5`, with 250,211 installations, ranked 95th out of 200,909 packages only, we included it in our evaluation to discuss the findings in case of XZ. Additionally, we also include two packages written in Rust in our evaluation as this ecosystem focuses on secure software development. We crawl 6,369 packages to find at least two Rust packages—`mdevctl` on rank 4,097 and `ripgrep` on rank 6,369. The final selection of packages can be found in Table 2. The crawler results are illustrated in Table 3. T3 includes the forward package- and build dependencies in the format x/y, where x represents the number of forward package dependencies and y represents the number of build dependencies. T5 includes the number sketchy file types and differences in the format x/y, where x represents the number of detected sketchy file types and y represents the number of differences between the source tarball and the repository for a given release tag. For simplicity, and to provide an initial indication of sketchy signals, we display only the number of differences that occur in either the source tarball or the repository. The specific list of differences can be printed to a log output. A full run, including fetching commits within the given time range, cloning the repository, performing the dependency-, file-, and file type analysis including the differences between the source tarball and the repository took 503.70 seconds on an Intel Core i7-1185G7 @ 3.00GHz CPU with 32GB of RAM.

Table 2. Selected Debian packages

Package	# Inst.	Since	Until	Trusted maintainer	Sketchy files	File types
liblzma5	250150	2022-01-01	2023-01-02	Larhzu, web-flow	configure.ac	application
libpam	251820	2022-01-01	2023-01-02	ldv-alt, thkukuk	configure.ac	application
libblkid1	251817	2022-01-01	2023-01-02	karelzak	configure.ac	application
zlib1g	251815	2022-01-01	2023-01-02	madler	configure.ac	application
mdevctl	14072	2024-01-01	2025-01-02	jonner	build.rs	application
ripgrep	6167	2024-01-01	2025-01-02	BurntSushi	build.rs	application

[6] `https://popcon.debian.org/main/by_inst`

Table 3. *SketchyCrawler* results

Package	T1	T2	T3	T4	T5
liblzma	5	33436	4/80	1	99/38
libpam	8	5704	32/74	2	199/28
libblkid1	26	14448	4/85	3	422/45
zlib1g	0	41461	4/75	5	650/60
mdevctl	0	1	53/87	1	758/14
ripgrep	21	1	5/84	3	869/25

6.3 Discussion

We believe that tools such as *SketchyCrawler* can be utilized as part of a risk management process to gather metrics about sketchy findings and include them in further risk assessments. For example, a package maintained by many developers is likely more difficult to verify than one with a single maintainer (e.g., `zlib1g` or `mdevctl`), and can therefore be assigned to a higher risk category. In Table 3, `liblzma5`—relevant in case of XZ—is listed with 5 commits made by an untrusted maintainer, specifically Jia Tan, the adversary. Additionally, *SketchyCrawler* highlights packages with a high number of reverse dependencies (e.g., `liblzma5` and `zlib1g`), which may be a valuable target for an adversary, as demonstrated in the XZ case. Next, we identify packages with a high number of forward dependencies, which may increase the likelihood of a vulnerability. Currently, dynamically loaded dependencies during runtime are beyond the scope of the current implementation. However, there are already discussions [2,34] within the open source community going on concerning dependencies loaded dynamically by `dlopen()`. One potential risk mitigation—particularly on projects with a high number of dependencies—is that software developers may consider additional security controls before publicly exposing such components. As expected, the number of sketchy files represented in T4 is low, since we use only the `configure.ac` and `build.rs` files to showcase the functionality of our crawler. However, in projects like XZ, where sketchy files such as `build-to-host.m4` are included in the `.gitignore` file, we expect to see an increased number here. T5 illustrates sketchy types, especially binary files with regards to our sample set. This can certainly be legitimate; however, it is a sign that there is also a lot of potential to obfuscate malicious code such as a backdoor. Finally, T5 also highlights the number of differences between the source tarball and the actual repository for a specific tag. While the results show significant differences, most of the files, such as the `.gitignore` and the `.git` directory, are expected not to be part of the source tarball.

6.4 Deployment consideration

Typically, Secure Software Development Life Cycles involve license clearing and component approval before third party libraries are integrated into a software product. As a result, software development teams may audit those third party components before adding them. However, since those components are often updated over time, it can be challenging for developers to audit every new version. We believe that integrating a tool such as *SketchyCrawler* into a *Continuous Integration/Continuous Deployment* (CI/CD) pipeline to identify sketchy signs allows development teams to focus first on critical parts in their supply chain. Particularly within the Rust ecosystem with its focus on secure software development, future integration with development team processes using the `cargo vet`[7] and/or `cargo geiger`[8] tools might be helpful to better manage audit resources.

7 Related work

The challenge of describing different supply chain attacks to develop effective mitigation techniques has been explored in several academic papers. Naik et al. [29] compare a range of various attacker models on a more abstract level. Several works focus on generalizing security frameworks (e.g., MITRE ATT&CK [9] or the Diamond Model [6]). Additionally, there is also research [17, 31] focusing on analyzing malicious software packages in general. Moreover, specific research efforts have investigated particular attacks, such as the SolarWinds attack presented by Martínez and Durán [28]. The variety of research tracks in the field of software supply chain security highlights the difficulty in mitigating such attacks properly and the XZ incident serves as a clear example that we still face open issues. Specifically concerning the XZ incident, there are existing attempts to counter such attacks, such as differential property monitoring [5] or a fuzzing approach to detect backdoor triggers at runtime [26]. However, our paper investigates emerging supply chain attacks such as the XZ incident, aiming to identify new methods in every stage by which adversaries can still successfully implant a backdoor in an open source project. In comparison with existing research, we outline a new critical attack path including corresponding mitigation techniques to support mitigation for similar attacks in the future. We abstract the critical attack path observed in the XZ incident, and believe this attack strategy could be applied to analogous supply chain attacks. During the course of our work, Hamer et al. [17] also conducted a study providing recommendations for preventing such supply chain attacks. However, their perspective focuses more on existing frameworks—suggesting the use of starter kits, improvements to current frameworks, and continued research into threat intelligence. In contrast, we analyze the critical attack path of a software supply chain attack based on new attacker strategies used in the XZ incident, highlight the main key takeaways, and provide a tool prototype to showcase how sketchy signs in open source projects can be identified.

[7] `https://mozilla.github.io/cargo-vet/`
[8] `https://github.com/geiger-rs/cargo-geiger`

8 Conclusion

The XZ Utils backdoor is another example of open source software that has become a central component of (too?) many foundational services, and therefore of critical Internet infrastructure components. Through the complexity and duration of the full attack path, we can speculate that the threat actor(s) are part of a highly sophisticated organization with stable funding. This example can be used as an excellent teaching opportunity for practical applications, both in terms of human/social/organizational and technical aspects. We have briefly summarized what we consider to be the critical components of the attack path, referencing many more detailed resources for specific parts of the attack analysis. We acknowledge that this is an empirical study for a single incident; however, as with social science, security incidents are rarely reproducible.

From this attack path, we derive five more generic takeaways for open source projects to mitigate future attacks on them or their (direct or indirect, forward or reverse) dependency tree. These should be seen as calls for action, but not as quick fixes. Particularly the complexity of old build toolchains around programming languages without strong type systems and memory safety will continue to leave open many different attack paths [18] that are hard to find, difficult to audit, and provide the cover of plausible deniability for attackers in the form of exploitable "bugdoor" attacks. Additionally, we illustrate how to support software development with a tool-based approach to identify new signs that may result from the continuous emerging threat landscape in the field of software supply chains. Our prototype tool, *SketchyCrawler*, demonstrate how such a solution can be built and utilized to support auditing of third party components by revealing sketchy signs.

One additional takeaway is a strengthened belief voiced by many of the experts involved with analyzing this case that similar attacks are possible on proprietary, closed source codebases, and that, as a security community, we have to assume other attacks with comparable complexity and sophistication to be actively deployed in multiple projects/products. It is now our shared task to find them, and we hope that this analysis and our prototype tool can help with a few steps on this difficult endeavor.

Availability

We provide an open-source repository of *SketchyCrawler* on `https://github.com/linsm/sketchy-crawler`.

Acknowledgments

This work has been carried out within the scope of Digidow, the Christian Doppler Laboratory for Private Digital Authentication in the Physical World and has partially been supported by the LIT Secure and Correct Systems Lab. We gratefully acknowledge financial support by the Austrian Federal Ministry of Economy, Energy and Tourism, the National Foundation for Research, Technology and Development, the Christian Doppler Research Association, 3 Banken IT GmbH, ekey biometric systems GmbH, Kepler Universitätsklinikum GmbH, NXP Semiconductors Austria GmbH & Co KG, Österreichische Staatsdruckerei GmbH, and the State of Upper Austria.

References

1. Akamai Security Intelligence Group: XZ Utils Backdoor — Everything You Need to Know, and What You Can Do (Apr 2025), `https://www.akamai.com/blog/security-research/critical-linux-backdoor-xz-utils-discovered-what-to-know`. Last accessed April 2025
2. Alden, D.: Identifying dependencies used via dlopen() (2024), `https://lwn.net/Articles/969908/`. Last accessed July 2025
3. Birsan, A.: Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies. Medium (Feb 2021), `https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610`. Last accessed December 2024
4. Boehs, E.: Everything I know about the XZ backdoor (Mar 2024), `https://boehs.org/node/everything-i-know-about-the-xz-backdoor`. Last accessed April 2024
5. Brechelmacher, O., Ničković, D., Nießen, T., Sallinger, S., Weissenbacher, G.: Differential property monitoring for backdoor detection. In: Ogata, K., Mery, D., Sun, M., Liu, S. (eds.) Formal Methods and Software Engineering. pp. 216–236. Springer Nature Singapore, Singapore (2024)
6. Caltagirone, S., Pendergast, A., Betz, C.: The diamond model of intrusion analysis. Threat Connect **298**(0704), 1–61 (2013)
7. Coldwind, G.: xz/liblzma: Bash-stage Obfuscation Explained (2024), `https://gynvael.coldwind.pl/?lang=en&id=782`. Last accessed April 2024
8. Collin, L.: XZ Utils backdoor (2024), `https://tukaani.org/xz-backdoor/`. Last accessed April 2024
9. Corporation, T.M.: ATT&CK (2025), `https://attack.mitre.org/`. Last accessed April 2025
10. Cox, R.: The xz attack shell script (2024), `https://research.swtch.com/xz-script`. Last accessed April 2024
11. Debian: UpstreamGuide (2025), `https://wiki.debian.org/UpstreamGuide`. Last accessed April 2025
12. Dullien, T.: Weird Machines, Exploitability, and Provable Unexploitability. IEEE Transactions on Emerging Topics in Computing **8**(2), 391–403 (2020). `https://doi.org/10.1109/TETC.2017.2785299`
13. Freund, A.: backdoor in upstream xz/liblzma leading to ssh server compromise. Post on mailing list oss-security@openwall (2024), `https://openwall.com/lists/oss-security/2024/03/29/4`. Last accessed April 2024
14. GNU Project: GNU M4 1.4.19 macro processor (2024), `https://www.gnu.org/software/m4/manual/m4.html`. Last accessed December 2024
15. Google: Accepting New Projects (2025), `https://google.github.io/oss-fuzz/getting-started/accepting-new-projects/`. Last accessed April 2025
16. Google: OSS-Fuzz: Continuous Fuzzing for Open Source Software (2025), `https://github.com/google/oss-fuzz`. Last accessed February 2025

17. Hamer, S., Bowen, J., Haque, M.N., Hines, R., Madden, C., Williams, L.: Closing the Chain: How to reduce your risk of being SolarWinds, Log4j, or XZ Utils. Computing Research Repository (CoRR), arXiv:2503.12192v1 [cs.SE] (Mar 2025). https://doi.org/10.48550/arXiv.2503.12192
18. Horn, J.: How a simple Linux kernel memory corruption bug can lead to complete system compromise. Project Zero (Oct 2021), https://googleprojectzero.blogspot.com/2021/10/how-simple-linux-kernel-memory.html. Last accessed April 2024
19. Jansen, H.: Add ifunc check to CMakeLists.txt (2023), https://git.tukaani.org/?p=xz.git;a=commitdiff;h=b72d21202402a603db6d512fb9271cfa83249639. Last accessed April 2024
20. Jansen, H.: Add ifunc check to configure.ac (2023), https://git.tukaani.org/?p=xz.git;a=commitdiff;h=23b5c36fb71904bfbe16bb20f976da38dadf6c3b. Last accessed April 2024
21. Jansen, H.: xz-utils: New upstream version available. Debian Bug report #1067708 (2024), https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=1067708. Last accessed April 2024
22. JiaT75: Added error text to warning when untaring with bsdtar. Pull request #1609 (2021), https://github.com/libarchive/libarchive/pull/1609. Last accessed April 2024
23. JiaT75: CMake: Update .gitignore for CMake artifacts from in source build (2022), https://github.com/tukaani-project/xz/commit/8ace358d65059152d9a1f43f4770170d29d35754. Last accessed April 2024
24. JiaT75: xz: Disable ifunc to fix Issue 60259. Pull request #10667 (2023), https://github.com/google/oss-fuzz/pull/10667. Last accessed April 2024
25. JiaT75: XZ updates. Pull request #9960 (2023), https://github.com/google/oss-fuzz/pull/9960. Last accessed April 2024
26. Kokkonis, D., Marcozzi, M., Decoux, E., Zacchiroli, S.: ROSA: Finding Backdoors with Fuzzing. In: 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE). pp. 720–720. IEEE (2025). https://doi.org/10.1109/ICSE55347.2025.00183
27. Kumar, J.: Re: [xz-devel] [PATCH] String to filter and filter to string. Reply on mailing list xz-devel (2022), https://www.mail-archive.com/xz-devel@tukaani.org/msg00555.html. Last accessed April 2024
28. Martínez, J., Durán, J.M.: Software supply chain attacks, a threat to global cybersecurity: SolarWinds' case study. International Journal of Safety and Security Engineering **11**(5), 537–545 (2021)
29. Naik, N., Jenkins, P., Grace, P., Song, J.: Comparing Attack Models for IT Systems: Lockheed Martin's Cyber Kill Chain, MITRE ATT&CK Framework and Diamond Model. In: 2022 IEEE International Symposium on Systems Engineering (ISSE). pp. 1–7 (2022). https://doi.org/10.1109/ISSE54508.2022.10005490
30. O'Donell, C.: GNU_IFUNC (2024), https://sourceware.org/glibc/wiki/GNU_IFUNC. Last accessed April 2024
31. Ohm, M., Plate, H., Sykosch, A., Meier, M.: Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In: Detection of Intrusions and Malware, and Vulnerability Assessment. LNCS, vol. 12223, pp. 23–43. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-52683-2_2
32. Parilli, A., Maclachlan, J.: No Unaccompanied Miners: Supply Chain Compromises Through Node.js Packages. Google Cloud Blog (Dec 2021), https://cloud.google.com/blog/topics/threat-intelligence/supply-chain-node-js/. Last accessed December 2024

33. Phylum Research: Rust Malware Staged on Crates.io (2023), `https://blog.phylum.io/rust-malware-staged-on-crates-io/`. Last accessed April 2025
34. Poettering, L.: Expose dlopen() dependencies in an ELF section, and add spec for it (2024), `https://github.com/systemd/systemd/pull/32234`. Last accessed July 2025
35. Rust: The Cargo Book (2025), `https://doc.rust-lang.org/cargo/reference/build-scripts.html`. Last accessed April 2025
36. Schwaighofer, M., Roland, M., Mayrhofer, R.: Extending Cloud Build Systems to Eliminate Transitive Trust. In: Proceedings of the 2024 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED '24). pp. 45–55. ACM, Salt Lake City, UT, USA (Oct 2024). `https://doi.org/10.1145/3689944.3696169`
37. Tan, J.: [xz-devel] [PATCH] String to filter and filter to string. Post on mailing list xz-devel (2022), `https://www.mail-archive.com/xz-devel@tukaani.org/msg00553.html`. Last accessed April 2024
38. Tan, J.: Tests: Add a few test files (2024), `https://git.tukaani.org/?p=xz.git;a=commitdiff;h=cf44e4b7f5dfdbf8c78aef377c10f71e274f63c0`. Last accessed April 2024
39. Tan, J.: Tests: Update two test files (2024), `https://git.tukaani.org/?p=xz.git;a=commitdiff;h=6e636819e8f070330d835fce46289a3ff72a7b89`. Last accessed April 2024
40. Tan, S.J., Bratus, S., Goodspeed, T.: Interrupt-oriented bugdoor programming: a minimalist approach to bugdooring embedded systems firmware. In: Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC '14). pp. 116–125. ACM, New Orleans, Louisiana, USA (2014). `https://doi.org/10.1145/2664243.2664268`
41. The Tukaani Project: LZMA Utils (2024), `https://tukaani.org/lzma/`. Last accessed April 2024
42. Watson, R.: History of LZMA Utils and XZ Utils (2013), `https://github.com/kobolabs/liblzma/blob/87b7682ce4b1c849504e2b3641cebaad62aaef87/doc/history.txt`. Last accessed April 2024