liblzma - OSS and Backdoors

Exploring the xz-utils Backdoor, its Emergence and its Impact on FOSS and OSS

9525469
xnacly.me
Applied Computer Science
DHBW Mosbach

July 2, 2024

Abstract

In recent years, several vulnerabilities in the open-source software supply chain were discovered. The most recent being the intentionally placed backdoor in the compression library named *liblzma*. This paper aims to explore the implementation of said backdoor while highlighting the insertion of the backdoor and the inserters use of social engineering enabling their placement in the leadership of the project. Furthermore ways of preventing similar attacks are presented and evaluated on the example of the *liblzma* situation.

1 Introduction

FOSS[1] is generally defined as software the user can *"[...] run, copy, distribute, study, change and improve [...]"* [2]. This requires the source to be available and enables the dependence of other software on subsets or the entirety of the code. On the other hand, source available or OSS[2] are distinct from FOSS software. Some licenses do not require the resulting product to be licensed under the same license as its dependencies, such as the MIT license[3]. It therefore differs from the GPL[4] and software licensed with the MIT-Licence can therefore not be referred to as free open-source software, but rather as open-source software.

Most OSS-projects accept contributions from individuals and enterprises. This is wanted and required to support the actuality of said software. Most OSS projects accept changes matching their pre-defined contribution guidelines and credit the contributor for their addition. These contributors often use the software they are contributing to and therefore make changes they care for, such as adding drivers for new devices to the Linux kernel [5].

However, other independent OSS contributors are abusing the contribution system by exploiting the trust the unpaid maintainers have in the quality of the submitted changes. Specifically, this refers to manipulating maintainers and inserting oneself into the group of by applying pressure on said group. As was the case with `liblzma` or the `xz-utils` OSS library.

[1] Free and Open-Source Software [1]

[2] Open-Source Software

[3] Requires the license to be present in *"all copies or substantial portions of the Software"* [3]

[4] Requires all copies of the software to be licensed as GPL [4]

1.1 Dependence on FOSS and OSS

Open source software is often divided into reusable components, such as libraries or toolkits implementing a specific feature, and built upon by other software. The goal is to use tried and tested components in the creation of new OSS, thus building on field tested and established software found in the OSS community.

Not only does OSS depend on other libraries from the OSS ecosystem. Proprietary software also makes use of said OSS components, while being forced to adhere to their terms, as declared in their respective licenses [6].

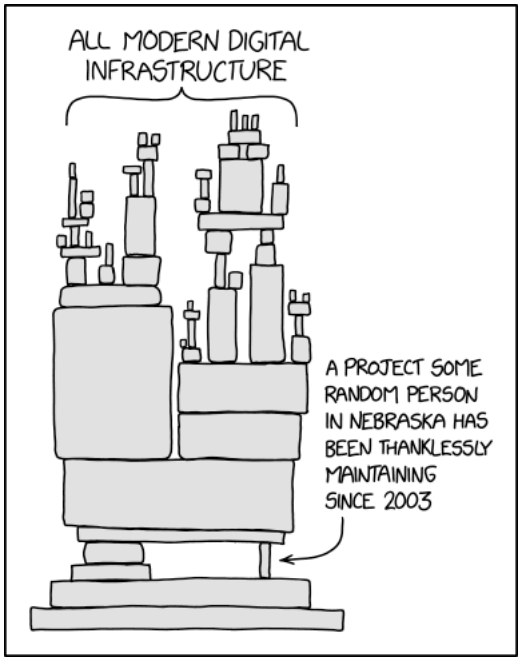

Figure 1: Dependency [7]

Commonly used examples for said libraries are `libcurl`, which provides multi protocol file transfers [8], `raylib` which is a library for video games programming [9] and the `sqlite` library, that implements a in process database [10]. These library examples are so widely used, a vulnerability in them would impact the security of the whole software space.

1.2 Supply Chain Security

Supply chain security is referring to the fact of ensuring the dependencies of a given software are to be considered secure and making sure this property can be accessed to be true. This can be achieved by keeping the development toolchain used for creating said software up to date, thus patching and removing found vulnerabilities [11]. Other ways of establishing the security of a dependency is to manually evaluate the source code of the given dependency.

However most largely used open source software is thoroughly tested and famously so. `sqlite` prides itself as being the *"most used and deployed database engine"* [12] and *"[...] the project has 590 times as much test code and test scripts [as lines of source code]"* [13].

Considering the large amount of memory safety errors[5], often caused by invalid or maliciously crafted input, projects entirely focussed around detecting said vulnerabilities, such as `OSS-Fuzz` [15], saw their inception.

1.3 xz-utils and liblzma

`xz-utils` refers to a c implementation of the `xz` compression algorithm and format using the Lempel–Ziv–Markov-Chain. It is written to comply with the C99 standard and consists of several components. One of these components is, as previously introduced, a library providing an API for compression and decompression, named `liblzma` [16]. According to the components documentation, its API is based on the `lzma` SDK but includes heavy modifications necessary for the `xz-utils` suite [17].

[5] 70% according to [14]

2 Backdoor Exploration

According to [18], a backdoor refers to a hidden method of gaining entry to a system bypassing security measures, such as biometric or password based authentication. They can be implemented in cryptographic algorithms, on the hardware level or in an application. Backdoors can be used to remotely access systems and are often hidden inside commonly used non-malicious software.

2.1 Implementation

`CVE-2024-3094` was assigned to the `libzlma` backdoor with the highest possible CVSS Score: 10.0. This assessement was made due to the severness of the included remote code execution [19]. The sophistication of this backdoor suggests a highly proficient attacker known by several confirmed aliases, such as *Jia Tan* (`JiaT75`), *Jigar Kumar*, `krygorin4545`, `misoeater91` and *Hans Jansen* [20].

2.2 Social Engineering & Pressure on OSS Maintainer

The process of injecting oneself into the group of maintainers of a open source software project is a complicated and tedious one. OSS-projects are often led by a close group of individuals, which often proved themselves by contributing constructive additions over a long period of time.
A threat actor, whether state-sponsored or a group of individuals, most often do not take this approach to tampering with software for the purpose of implementing a backdoor. However this particular backdoor was implemented with said path over the course of three years.
Specifically the threat actor abused the mental state of the lead maintainer by applying pressure on them via sock puppet accounts[6] and depicting the project changes made by the lead maintainer as slow and not begin up to date enough. Using said pressure in combination with the maintainers mental health issues enabled the threat actor to gain the trust and therefore the co-maintainer position [20].
This position allowed the threat actor to make changes to the build-pipeline, test files and to sign-off and release versions of the software itself to the public.

2.3 Build Pipeline Manipulation

As introduced before, the attacker uses this privileged position in the maintainer group to make changes to the test files by introducing a file called `build-to-host.m4`[7] [8]. This file is included in the package release, but not in the version controlled repository. Furthermore the threat actor embeds obfuscated and encrypted stages of the backdoor in two test files called `bad-3-corrupt_lzma2.xz` and `good-large_compressed.lzma`.

Byte	ASCII	Substitution
0x09	"\t"	0x20
0x20	" "	0x09
0x2d	"-"	0x5f
0x5f	"_"	0x2d

Figure 2: Substitution table according to [20]

The aforementioned macro is executed during the build process and substitutes characters in the `bad-3-corrupt_lzma2.xz` file, as shown in Section 2.3. Upon having performed the substitution, the `.xz` file is decoded and ready for the first stage of the backdoor.

[6] false online identity created and used specifically for deceptive purposes
[7] unix macro processor used in `autoconf` [21]
[8] produces scripts to configure software [22]

2.4 IFUNC & CPU Features

`IFUNC`, short for GNU indirect function is a mechanism for resolving a function call to an implementation. This is done by invoking the resolver of said function upon the functions initial invocation. Marking a function with `ifunc` allows for the symbol value resolution at load time, for instance of a shared object. This process is made possible by using the `STT_GNU_IFUNC` symbol type ELF standard extension. This functionality is used to replace a generic function implementation with a platform and architecture specific and often optimized implementation. However there are some requirements and safe guards for using `ifunc`: firstly the attributed function can not be marked as `weak`[9], the resolver and the indirect function have to be defined in the same translation unit and `glibc` is required [23].

```
#include <stdio.h>
#include <stdlib.h>

void generic_function_linux() {
↪  puts("linux"); }

void generic_function_windows() {
↪  puts("windows"); }
```

Listing 1: `ifunc` platform specific function

Would one require a function to be executed on specific runtime conditions, such as conditionally use a faster implementation for the current architecture, one could make use of the `ifunc` attribute as follows. `ifunc` requires an implementation, as shown in Section 2.4. And a resolver, see Listing 2.4. The compiler requires the function implementation signatures to match the function marked with the attribute.

[9]declare symbol as weak and not global, allows for overriding

```
void (*select_generic_function())()
↪  {
#ifdef __linux__
  return generic_function_linux;
#elif _WIN32
  return generic_function_windows;
#else
  return NULL;
#endif
}
```

Listing 2: `ifunc` function resolver

As introduced before, the usage of `ifunc` introduces symbols with the `STT_GNU_IFUNC` in the ELF symbol table in the resulting binary. These entries point to their respective resolver functions.

```
void generic_function()
↪  __attribute__(
(ifunc("select_generic_function")));

int main() {
  generic_function();
  return EXIT_SUCCESS;
}
```

Listing 3: `ifunc` function stub and function usage

The dynamic linker resolves the symbols at load time by calling the resolver function and patching the symbol with the correct address.

```
void generic_function_malicious() {
    puts("linux");
    system("whoami");
}
```

Listing 4: `ifunc` malicious function

While there are a number of legitimate uses of this feature the threat actor used `ifunc` to overwrite `RSA_public_decrypt()`[10] of the `OpenSSL`

[10]used for implementing RSA decryption [24]

project with a call to `system()`[11], executing injected shell code after successful authentication, thus effectively having introduced a remote code execution vulnerability. Considering the example laid out in Listing 2.4, Section 2.4 and Listing 2.4, one can observe a severely simplified implementation of said backdoor in Section 2.4. The simplified backdoor, see Listing 2.4 and Listing 2.4, mimics the way `liblzma` was included in the resulting binary by inserting the call to the malicious implementation thereby replacing the resolver logic and enabling the execution of shell code. While this allows for a reduced overview over the inner workings of the backdoor, the complexity of said backdoor is significantly larger than shown here.

```
void (*select_generic_function())()
{
  return generic_function_malicious;
#ifdef __linux__
  return generic_function_linux;
#elif _WIN32
  return generic_function_windows;
#else
  return NULL;
#endif
}
```

Listing 5: manipulated `ifunc` function resolver definition

As introduced before, the first stage of the backdoor is to manipulate the build pipeline to produce a malicious bash file for producing a shared object containing the backdoor. Specifically this includes the changes from Section 2.3 and an other file introduced in Section 2.3, `good-large_compressed.lzma`.

[11] run sub programs [25]

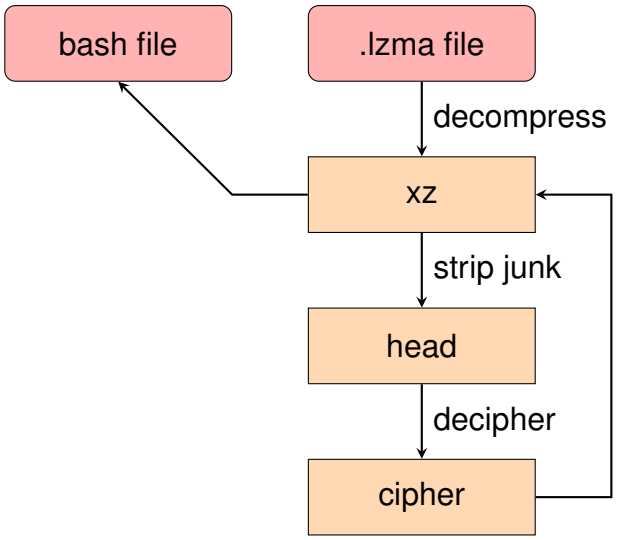

Figure 3: Stage 1: backdoor prep

Firstly, said file is decompressed via the `xz` command line application. The resulting blob is then stripped of junk data with the `head` unix tool, while a portion is discarded and subsequently deciphered using a custom substitution cipher. Once the data is deciphered, the `xz` suite is used to decompress the resulting blob. See Listing 2.4.

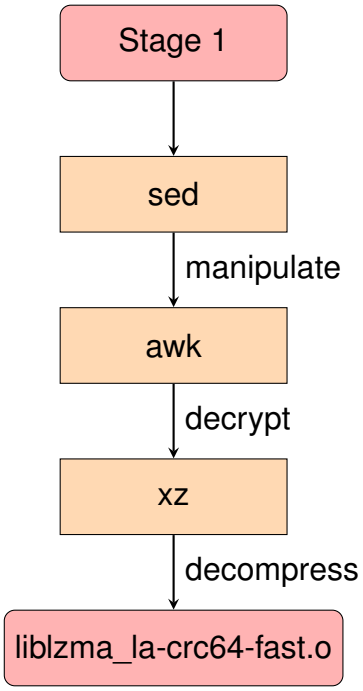

Figure 4: Stage 2: creation of `liblzma_la-crc64-fast.o`

As Figure 2.4 visualises, The resulting text is used as the starting point for the second stage of the backdoor, specifically this stage encloses the extraction of the backdoor. Stage two at first sets the `LC_ALL` environment variable to `C` for the sake of ensuring consistent behaviour by using ASCII. The output of the previous stage is manipulated with the `sed` unix package and specifically moving every byte encountered to its own line. Once this reformatting is performed, the data is decrypted with a stream cipher and decompressed with the `xz` command line application [20]. The resulting shared object is part of the compilation and linking process and is therefore included in the resulting binary.

2.5 Indirect Dependence on liblzma

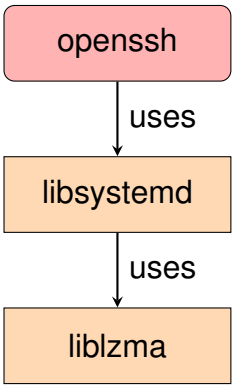

Figure 5: `openssh` dependency graph

`openssh` does not directly depend on the `libzlma` library, however a number of systems patch `openssh` to integrate `libsystemd` for the purpose of supporting `libsystemd` notifications[12]. This indirect dependencies is dangerously obscure due to the indirection. The indirection is not only obscure, but also enables the loading of shared objects of every node in the cascading dependence chain.

[12]used to notify the `systemd` about status changes

In this chain of shared objects, the threat actor decided to attach the `liblzma` backdoor. Specifically the backdoor is active once `openssh` loads the `libsystemd` dependency. Once this dependency is attempted to be loaded, it starts to load its own dependencies, here the backdoored `liblzma`. In this linking stage the before explained hijacking of the `RSA_public_decrypt()` function is performed.
The result of this process is a remote code execution in said function.

2.6 Prerequisites for the Backdoor

- built with `GCC` and `glibc`
- shared object is opened on `x86-64`
- built by `dpkg` or `rpm`

3 Response

3.1 Detection

- Detected by Andres Freund on 29th March 2024
- PostgreSQL developer
- noticed high CPU usage and Valgrind errors when attempting to connect via SSH
- compromised version not yet deployed to production systems but dev builds

3.2 Software Distributors Reactions

- Red Hat, SUSE and Debian downgraded `libzlma` to its previous version

- Ubuntu held the beta release of Ubutun 24.04 LTS back by a week and rebuild all binaries of the distros packages

3.3 Supply Chain Security

- should critical parts of the software supply chain depend on unpaid volunteers
- keeping the supply chain secure

4 Conclusion

In conclusion the `xz-utils/libzlma` backdoor is the result of years of work by a highly sophisticated threat actor. The backdoor implementation shows the deep knowledge of low level programming, obscure compiler features, execution formats, the open source software supply chain and its security while exhibiting the ability for total control over the threat actors operational security. No group or individual have taken credit or were accredited to have engineered this backdoor.

4.1 Funding FOSS and OSS

4.2 Vetting Dependency

As shown in Section 2.5, not only the direct dependencies of a given software project have to be secure, but their dependencies too. To ensure this, a large effort has to be made to vet this source code.
Making sure source code is secure is a combination of static and dynamic analysis, testing, fuzzing and human analysis. Ignoring the last method, all can be largely automated, human analysis however is a time consuming process and requires manual labor.

References

[1] R. M. S. (RMS). (2021), [Online]. Available: `https://www.gnu.org/philosophy/pragmatic.html` (visited on 06/04/2024).

[2] F. S. Foundation. (2024), [Online]. Available: `https://www.gnu.org/philosophy/free-sw.html` (visited on 06/04/2024).

[3] Opensource.org. (2024), [Online]. Available: `https://opensource.org/license/MIT` (visited on 06/04/2024).

[4] Opensource.org. (2024), [Online]. Available: `https://opensource.org/license/gpl` (visited on 06/04/2024).

[5] T. kernel development community. “Linux - introduction - device drivers.” (2021), [Online]. Available: `https://linux-kernel-labs.github.io/refs/heads/master/lectures/intro.html#device-drivers` (visited on 06/10/2024).

[6] D. Stenberg. “Companies using curl in commercial environments.” (2024), [Online]. Available: `https://curl.se/docs/companies.html` (visited on 06/09/2024).

[7] xkcd. “Dependency.” (), [Online]. Available: `https://xkcd.com/2347/` (visited on 06/09/2024).

[8] D. Stenberg. “Libcurl - the multiprotocol file transfer library.” (2024), [Online]. Available: `https://curl.se/libcurl/` (visited on 06/09/2024).

[9] raysan5. “Raylib is a simple and easy-to-use library to enjoy videogames programming.” (2024), [Online]. Available: `https://www.raylib.com/` (visited on 06/09/2024).

[10] D. R. Hipp. “What is sqlite?” (2024), [Online]. Available: https://www.sqlite.org/index.html (visited on 06/09/2024).

[11] M. Stapelberg. “Supply chain security with go.” (2024), [Online]. Available: https://media.ccc.de/v/gpn22-438-supply-chain-security-with-go (visited on 06/07/2024).

[12] J. M. Dwayne Richard Hipp Dan Kennedy. “Most widely deployed and used database engine.” (2022), [Online]. Available: https://www.sqlite.org/mostdeployed.html (visited on 06/20/2024).

[13] J. M. Dwayne Richard Hipp Dan Kennedy. “How sqlite is tested.” (2022), [Online]. Available: https://www.sqlite.org/testing.html (visited on 06/20/2024).

[14] T. C. Project. “Memory safety.” (), [Online]. Available: https://www.chromium.org/Home/chromium-security/memory-safety/ (visited on 06/10/2024).

[15] G. O. Source. “Oss-fuzz.” (2024), [Online]. Available: https://google.github.io/oss-fuzz/ (visited on 06/20/2024).

[16] L. Collin. “Xz utils.” (2024), [Online]. Available: https://tukaani.org/xz/#_introduction (visited on 06/21/2024).

[17] cy. “Common code <> different backdoors.” (2024), [Online]. Available: https://media.ccc.de/v/gpn22-304-common-code-different-backdoors (visited on 06/05/2024).

[18] C. Wysopal and C. Eng. “Static detection of application backdoors.” (2007), [Online]. Available: https://www.veracode.com/sites/default/files/Resources/Whitepapers/static-detection-of-backdoors-1.0.pdf (visited on 06/24/2024).

[19] R. Hat. “Cve-2024-3094.” (2024), [Online]. Available: https://access.redhat.com/security/cve/CVE-2024-3094 (visited on 06/24/2024).

[20] D. Goodin. “What we know about the xz utils backdoor that almost infected the world.” (2024), [Online]. Available: https://arstechnica.com/security/2024/04/what-we-know-about-the-xz-utils-backdoor-that-almost-infected-the-world/ (visited on 06/27/2024).

[21] E. B. Gary V. Vaughan. “Gnu m4.” (2021), [Online]. Available: https://www.gnu.org/software/m4/ (visited on 06/30/2024).

[22] E. B. Paul Eggert. “Autoconf.” (2020), [Online]. Available: https://www.gnu.org/software/autoconf/autoconf.html (visited on 06/30/2024).

[23] G. Team. “6.30 declaring attributes of functions.” (2012), [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/Function-Attributes.html (visited on 07/01/2024).

[24] O. P. Authors. “Rscrpt.c.” (2022), [Online]. Available: https://github.com/openssl/openssl/blob/master/crypto/rsa/rsa_crpt.c#L51 (visited on 07/01/2024).

[25] I. Free Software Foundation. “26.1 running a command.” (2023), [Online]. Available: https://www.gnu.org/software/libc/manual/html_node/Running-a-Command.html (visited on 07/01/2024).

Appendix