

TP — Recherche de vulnérabilités avec KLEE

Les options comme `-fsanitize` des compilateurs (qui ajoutent des tests défensifs à l'exécution) ou les outils comme `valgrind` permettent de détecter des vulnérabilités dans les programmes **à l'exécution**.

Pour les utiliser, encore faut-il disposer de jeux de tests avec une "bonne" *couverture* du code. C'est ce que propose l'outil **KLEE** qui génère des tests visant à couvrir l'ensemble des chemins d'exécution (par des techniques combinant exécution symbolique, SMT-solving, exécution concrète, etc). Ce TP a été testé avec KLEE 3.0 compilé pour clang+llvm 13.

Principes d'utilisation de Klee pour ce TP Voilà la suite de commandes qu'on va exécuter sur les exemples du TP comme `optim_klee.c`. En fait, on semi-automatise cette suite de commandes en lançant plutôt `./imag_klee.sh optim_klee.c` (après avoir éventuellement fait `chmod u+x imag_klee.sh`) qui exécute les étapes 1+2 ci-dessous.

1. compiler (dans la représentation intermédiaire **LLVM-IR**).

```
clang -I /usr/local/include/klee -emit-llvm -c -g -O0 \
  -Xclang -disable-O0-optnone -DKLEE optim_klee.c
llvm-dis optim_klee.bc
```

2. lancer la génération de cas de tests

```
klee optim_klee.ll
```

Pour chaque chemin d'exécution trouvé par KLEE, on peut observer les affichages produit par ce chemin d'exécution (l'ordre des chemins d'exécution correspond à l'ordre de numéro des fichiers de tests générés par KLEE, cf. ci-dessous).

3. Si KLEE signale une erreur (ce qui n'est pas le cas sur `optim_klee.c`) alors, on examine les cas d'erreurs dans le répertoire `klee-last`. Chaque erreur détecté est signalée par un fichier `*.err` comme `test0000NN.ptr.err` ou `test0000NN.assert.err`. L'entrée provoquant cette erreur est donnée par le fichier `test0000NN.ktest`. Typiquement, `*ptr.err` signale un accès invalide à un pointeur. On **vérifie** alors que l'erreur se produit en compilant un exécutable avec par exemple l'option `-fsanitize=address`.
4. Dans tous les cas, on examine la couverture des tests

```
ktest-tool klee-last/*.ktest | less
```

On peut rejouer ces cas de tests en compilant (voir le détail sur chaque exemple).

Explications : KLEE produit des fichiers `*.ktest` qui sont sauvegardés dans un répertoire `klee-out-NN/` (généré à chaque lancement de `klee` avec des numéros croissants). Le lien symbolique `klee.last` pointe sur le dernier répertoire produit.

A FAIRE EN FIN DE TP (pour nettoyer votre répertoire)

```
rm -rf klee-* grub linear-search mdp-klee optim_klee
```

Exercice 1 (fichier `optim_klee.c`).

▷ **Question 1.** Appliquer les consignes ci-dessus. A quoi correspondent les cas de tests trouvés par KLEE?

▷ **Question 2.** Compiler `optim_klee.c` avec

```
gcc optim_klee.c -fsanitize=undefined -o optim_klee
```

puis essayer (e.g. avec `./optim_klee; echo $?`) les différents tests proposés par KLEE en ligne de commande. Qu'en conclure?

Exercice 2 (fichier `mdp-klee.c`).

▷ **Question 1.** Faire tourner KLEE. Que pensez-vous des tests fournis? A quoi correspondent-ils?

▷ **Question 2.** Que se passe-t-il si on change le `size=10` en `size=8`?

Exercice 3 (fichier `linear-search.c`).

▷ **Question 1.** Ce programme contient une erreur. La voyez-vous? Faire tourner KLEE et justifier le nombre de tests produits par Klee.

▷ **Question 2.** Compiler avec

```
gcc linear-search.c -fsanitize=address -o linear-search
```

et faire tourner le code avec la valeur fournie pour le cas d'erreur. Puis recommencer sans le sanitizer. Qu'observe t-on? Est-ce normal?

▷ **Question 3.** Corriger l'erreur dans le fichier fournie et refaites passer KLEE et les tests...

Exercice 4 (fichier `grub.c`). Ce exercice est tiré d'une vraie CVE qui permettait de s'authentifier sans mot de passe en mode root...

▷ **Question 1.** Faites tourner KLEE.

1. Utiliser `man ascii` pour comprendre les tests produits par KLEE.
2. Vérifier que l'erreur détectée se produit à l'exécution.
3. Comprendre et corriger l'erreur.

▷ **Question 2.** Expliquer pourquoi l'accès `buf[cur_len]` ne provoque pas d'erreur.

▷ **Question 3.** Dans le programme original de cette CVE (voir URL ci-dessous), il y avait un `while (1)` au lieu du `for (int i=0; i<6 ; i++)`. Comment l'exécution de KLEE se comporte si on remet le `while (1)` original?

Pour finir lire comment cette vulnérabilité a pu être transformée en attaque sur [la page de cette CVE](#).