TP fautes au niveau binaire

Master *Advanced Security* 2024

Damien Couroussé, CEA-List Grenoble – damien.courousse@cea.fr

November 27, 2024

Ce TP a pour but d'aborder quelques aspects pratiques liés à l'analyse de vulnérabilité aux injections de fautes sur le code binaire, et à l'impact du compilateur sur des contre-mesures logicielles contre les attaques par injection de fautes.

1 Préliminaires

1.1 Nom de l'image docker

Dans tout ce document, on suppose que l'image docker utilisée en TP s'appelle `TP_IMAGE`. Le nom de l'image à utiliser vous sera communiqué en séance.

1.2 Notations

Commande lancée dans le terminal de la machine hôte :

```
$ commande
```

Commande dans un conteneur docker :

```
(...)# commande
```

1.3 Préparation de l'environnement de TP

Sur votre machine de TP, se trouve une image `docker` nommée `TP_IMAGE`, qui contient l'ensemble des outils et des fichiers dont vous aurez besoin.

→ Vérifiez que l'image est installée avec la commande :

```
$ docker images
```

→ Se placer ensuite dans l'environnement de travail du TP, en créant un conteneur `docker` à partir de cette image :

```
$ docker run --rm -it TP_IMAGE bash
```

Vous pouvez observer le contenu suivant :

```
root@db6875599ef7:/homedir# pwd
/homedir
root@db6875599ef7:/homedir# ls
Dockerfile  Makefile  README.org  TP.Dockerfile  apps  fistic  platforms  src
```

1.4 Utilisation de l'environnement de TP

Le contenu de l'image `docker` vous permet de compiler des programmes binaires pour un système embarqué ARM de type STM32, et d'émuler leur exécution à l'aide de QEMU.

Dans le conteneur, le répertoire `/homedir` contient les sources utiles pour ce TP.

- `platforms/stm32-qemu.conf` : fichier de configuration pour la compilation et l'émulation de programmes ARM pour STM32/QEMU.
- `apps` : code source des programmes à analyser
- `src/common.mk` : ce fichier contient quelques variables utiles à la compilation,
  - notamment la variable `OPT_LEVEL` permet de configurer le niveau de d'optimisation.
    - Valeurs supportées : `-O0`, `-O1`, `-O2`, `-O3`, `-Os`.

* Sa valeur peut être modifiée :
  - dans le `Makefile` du programme cible. e.g. dans `apps/verifypin0/Makefile` :
    `OPT_LEVEL = -Os`
  - dans la ligne de commande, lors de l'appel à `make` :
    `# OPT_LEVEL = -Os make clean dump`
  - Veillez à bien nettoyer tous les fichiers de build (`make clean`) après chaque modification des options de compilation pour que les changements soient répercutés à tous les fichiers compilés !

1.4.1 configuration

→ Avant toute utilisation du conteneur, il faut charger la configuration de compilation :

```
# source platforms/stm32-qemu.conf
```

1.4.2 compilation

Le `Makefile` à la racine (i.e., dans `/homedir`) apporte en particulier les règles de build suivantes :

- `make`, ou `make show-targets` : afficher la liste des programmes supportés (cf. `apps`)
- `make elf` : compiler tous les programmes
- `make dump` : produit un *dump* de chaque binaire sous la forme d'un fichier texte (e.g. `AES.dump`)

Chaque sous-répertoire de `apps` (e.g., `apps/verifypin0`) supporte entre autres les règles de build suivantes :

- `make` : compilation du programme binaire.
- `make dump` : produit un *dump* du programme binaire sous la forme d'un fichier texte (e.g. `AES.dump`)
- `make run` : émulation du programme binaire avec QEMU.
- `make gdb` : lancement d'une session de debug avec `gdb`.

1.4.3 simulation de fautes par mutation de code

L'outil `fistic`, contenu dans le conteneur docker, permet d'appliquer quelques modèles de fautes de niveau ISA par mutation de code. Le point d'entrée principal de l'outil est le programme `fistic-core`.

→ Parcourez les options de l'outil (commande `fistic-core --help`).

L'analyse en fautes avec `fistic` consiste à générer un nouveau programme binaire (appelé *mutant*), copie du programme original, qui est légèrement modifié pour représenter l'effet de la faute qu'on souhaite simuler. Par exemple, un modèle de faute de *saut d'instruction* est simulé par le remplacement de l'instruction machine ciblée par une instruction `nop`.

Pour chaque instance possible du modèle de faute (instance du modèle de faute, point d'injection de la faute ou combinaison de points d'injection en fautes multiples), `fistic` produit un mutant qui est ensuite exécuté (ici avec QEMU).

Chaque mutant est ensuite exécuté pour simuler l'effet des fautes. Le code du programme à analyser doit être instrumenté pour aider à l'analyse. En fin d'exécution, `fistic` attend l'affichage d'une des deux chaînes de caractères suivantes :

- `==VERDICT== OK` : le programme se comporte normalement.
- `==VERDICT== FAIL` : objectif d'attaque atteint.

L'exécution de `fistic` produit un rapport d'analyse (format texte, dans le terminal), où chaque ligne correspond à l'exécution d'un programme mutant, i.e. à la simulation d'un programme sur lequel ont été appliqués un ou plusieurs modèles de fautes. Le rapport d'analyse considère les cas suivants :

- `valid` : comportement normal.
- `invalid` : attaque réussie.
- `timeout` : la simulation du programme analysé ne s'est pas terminée, la simulation est interrompue.
- `failure` : la simulation du programme analysé s'est terminée, mais ne produit aucune des deux chaînes de caractères attendues.

2 Prise en mains

→ Démarrez le conteneur `docker`, et sourcez l'environnement de configuration.

2.1 Compilation

Pour le programme `verifypin0` :

- → compilation, exécution du binaire,
- → analyse du dump programme: identifiez le point de démarrage du programme, les composantes de la fonction VerifyPIN.

2.2 Simulation de fautes

À partir du répertoire `apps/verifypin0`, lancez `fistic` :

```
# fistic-core -e qemu -t 10000 \
    --fault-model skip \
    --functions verifyPIN_A byteArrayCompare \
    -n 1 -b VerifyPIN_0.elf
```

Cette analyse applique des fautes par saut d'instruction sur chaque instruction des fonctions `verifyPIN_A` et `byteArrayCompare`.

2.3 Rejeu de programmes fautés

`fistic` conserve tous les programmes dans le répertoire nommé par défaut `faulted.bin` (nom configurable).

Il est possible de lancer le programme binaire compilé avec la commande :

```
# make run
qemu-system-arm -machine lm3s6965evb -cpu cortex-m3 -nographic -monitor null -serial null -semihosting
(...)
Not Authenticated [AUTH]
==VERDICT== OK
```

... cette commande lance QEMU avec les paramètres adéquats pour simuler un programme compilé pour une plateforme d'exécution de type STM32. Il est possible de ré-utiliser cette commande pour lancer un mutant produit par `fistic`, par exemple :

```
# qemu-system-arm (...) -kernel faulted.bin/f36.bin
(...)
Authenticated [AUTH]
ORACLE: ATTACK SUCCESS
==VERDICT== FAIL
```

3 Analyse de vulnérabilité aux fautes par saut d'instruction

3.1 verifyPin0, $N = 1$ fautes

→ Analysez une attaque réussie, expliquez comment la faute injectée permet une authentification réussie.

→ Pourquoi est-ce que, avec les options de compilation proposées par défaut, les fautes injectées dans la fonction `byteArrayCompare` ne donnent lieu à aucune exploitation ?

→ Modifiez les sources du programme (fichier `code.c`), pour que la fonction `byteArrayCompare` ne soit pas inlinée dans la fonction `verifyPIN_A`. Recompilez et relancez l'analyse de sécurité. Que concluez-vous ?

→ Recompilez les sources pour chaque niveau d'optimisation et dressez une synthèse du nombre de comportements inattendus dans chaque cas.

3.2 verifyPin5

3.2.1 $N = 1$ fautes

→ Analysez les différences avec le verifyPin0, sur la base des sources et des codes compilés (`.dump`). Analysez les protections ajoutées et leur effet attendu.

→ Dans la fonction `byteArrayCompare`, vous pouvez remarquer que la version compilée ne contient plus la vérification de la valeur du compteur `i` en sortie de boucle. Pourquoi ?

→ Expliquez pourquoi, pour ce programme, les résultats d'injections de fautes dans la fonction `byteArrayCompare` sont à prendre avec précautions.

3.2.2 $N = 2$ fautes

Dans la suite, les fautes sont seulement injectées dans la fonction `verifyPIN_5`.

Lancez une analyse avec $N = 2$ fautes.

→ Pourquoi le temps d'analyse est-il considérablement allongé ? Vérifiez le nombre de mutants testés.

→ Faites l'analyse de quelques vulnérabilités identifiées.