

## TP — Introduction à la sécurisation du code C

Ce TP montre un certain nombre de vulnérabilités classiques du C et leurs dangers. Et aussi les protections contre ces vulnérabilités ajoutées dans l'OS et dans gcc. Ici, on joue justement à désactiver certaines protections par défaut pour voir leur intérêt...

Voilà des outils à utiliser pendant ce TP :

**Plugin Eva de Frama-C** pour détecter les vulnérabilités dans les sources C. Pour l'utiliser, il faut commencer par initialiser Frama-C sur votre session Ensimag avec  
`source /matieres/4MMACSS/init-frama-c.sh`  
Puis lancer “`frama-c -eva`” ou “`frama-c-gui -eva`” (avec votre fichier C sur la ligne de commande)

**Ghidra** désassemblage & décompilation des binaires (<https://ghidra-sre.org/>). Pour l'utiliser :

- Lancer `ghidraRun &` (installation école `/opt/ghidra/ghidraRun`)
- Créer un nouveau projet : “File” puis “New Project” puis “Non-Shared Project”
- Donner un nom à “Project Name” (ex : TP-secu-acss)
- Puis cliquer sur codebrowser (l'icône en forme de dragon)
- Importer le fichier exécutable souhaité au projet : “File” puis “import file” puis nom-du-binaire.
- Lancer l'analyse du fichier et dans Symbol Tree puis functions, chercher la fonction que vous voulez analyser.

**Exercice 1 (optimise).** Regarder le code source de `optimise.c`.

▷ **Question 1.** Tester le avec `frama-c-gui -eva`. Quel est le problème ?

▷ **Question 2.** Compiler et exécuter pour chacun des cas ci-dessous. Analyser dans chaque cas le binaire généré avec `ghidraRun` : regarder à la fois la décompilation de la fonction `g` et de `main`. Qu'est-ce qu'on observe ? Quel est l'effet des différentes options de gcc ?

**no option :**

```
gcc -o optimise1 optimise.c
```

**optimisation option :**

```
gcc -O2 -o optimise2 optimise.c
```

**overflow detection option :**

```
gcc -fno-strict-overflow -o optimise3 optimise.c
```

**optimisation and overflow detection option :**

```
gcc -O2 -fno-strict-overflow -o optimise4 optimise.c
```

▷ **Question 3.** Proposer une solution pour rendre cette fonction `g` sécurisée en utilisant les préconisations données ici : <https://www.securecoding.cert.org/confluence/display/c/INT32-C.+Ensure+that+operations+on+signed+integers+do+not+result+in+overflow>. Vérifier qu'elle est correcte avec FRAMA-C.

**Exercice 2 (winloose).** Regarder le code du programme `winloose.c`. Ce programme prend 2 entiers sur la ligne de commande, (`argv[1]` et `argv[2]`). Compiler ce programme avec la commande suivante :

```
gcc -fno-stack-protector -o winloose1 winloose.c
```

Ce programme peut mener à différents résultats :

- print "You loose"
- infinite loop
- crash
- etc.

▷ **Question 1.** Trouver la vulnérabilité avec `frama-c-gui -eva`. Quel est le nom de ce type de vulnérabilité ?

▷ **Question 2.** Trouver les entrées qui permettent d'imprimer "You win" ou de provoquer une boucle infinie !

▷ **Question 3.** Regarder le code assembleur dans le fichier `winloose1.s` produit par `gcc -fno-stack-protector -S -o winloose1.s winloose.c` Regarder le code du main et retrouver le placement des variables locales dans la pile. On pourra aussi éventuellement s'aider de Ghidra sur l'exécutable `winloose1`.

**Indication :** dans une architecture X86 64-bits sous Linux le pointeur de début de la stackframe s'appelle `rbp` et le pointeur de sommet de pile `rsp`. Les deux premiers paramètres s'appellent respectivement `rdi` et `rsi`. Plus généralement, voir les conventions de liaison de la plateforme (ou ABI pour *Application Binary Interface*)<sup>1</sup>.

Expliquer les résultats obtenus en dessinant la pile à l'exécution et en particulier en identifiant les adresses des variables locales et des paramètres dans la pile.

▷ **Question 4.** Compiler maintenant le programme `winloose.c` avec l'option "stack protection" :

```
gcc -fstack-protector -o winloose2 winloose.c
```

Exécuter le programme avec les entrées précédentes. Que se passe t'il ?

▷ **Question 5.** En analysant l'assembleur, expliquer comment le mécanisme de protection de la pile est implémentée.

**Exercice 3 (Un bonjour malheureux).** Dans cet exercice, on positionne la variable `PATH` à export `PATH=.:${PATH}`. Compilez `bonjour.c` avec `gcc bonjour.c -o bonjour` et exécutez `bonjour`.

▷ **Question 1.** Regardez ensuite le code source et proposez une attaque qui fait qu'au prochain lancement de `bonjour`, vous verrez `*** T'as été piraté ***` et un interpréteur `python3` s'ouvrira dans la ligne de commande. Vous n'avez pas le droit de recompiler `bonjour`.

▷ **Question 2.** Est-ce que `frama-c -eva` peut aider pour détecter ce type de vulnérabilité ?

1. Voir [https://wiki.osdev.org/System\\_V\\_ABI#x86-64](https://wiki.osdev.org/System_V_ABI#x86-64).

**Exercice 4 (UAF & Uninit).**

- ▷ **Question 1.** Comprendre le programme `uaf.c` : il contient un use-after-free. Le voyez-vous ? Lancer `frama-c-gui -eva` pour vérifier.
- ▷ **Question 2.** Suivre les instructions données en début de fichier. Expliquer pourquoi et comment un attaquant peut faire exécuter son propre code....
- ▷ **Question 3.** Refaire les questions précédentes pour le fichier `uninit.c` : quelle est la vulnérabilité que l'attaquant peut exploiter ici ?

**Exercice 5 (Tester un mot de passe).**

- ▷ **Question 1.** Regarder en détail la spécification de `fgets` : <http://www.cplusplus.com/reference/cstdio/fgets/>. Quels sont les différents comportements possibles ?
- ▷ **Question 2.** Tester le programme `fgets.c` et trouver une attaque qui permet de s'authentifier sans rentrer 2 fois le bon mot de passe. Rappel : au terminal on peut encoder la fin de fichier par contrôle-D. Que se passe t-il pour que l'attaque fonctionne ?
- ▷ **Question 3.** Corriger le programme en prenant en compte les cas d'erreurs. On pourra s'inspirer du site suivant : <https://wiki.sei.cmu.edu/confluence/display/c/ERR33-C.+Detect+and+handle+standard+library+errors>
- ▷ **Question 4.** Une préconisation en sécurité consiste à nettoyer les mémoires. Compléter votre solution. Est-elle robuste aux optimisations ?
- ▷ **Question 5.** Regarder le programme `exploit2-fixed-fgets.c` qui teste un appel à `IsPasswordOK`. Montrer que ce programme est potentiellement vulnérable en changeant la valeur de `SIZE`.

Un site qui explique les problèmes de lecture : <https://openclassrooms.com/fr/courses/19980-apprenez-a-programmer-en-c/16993-securisez-la-saisie-de-texte>

**Exercice 6 (bufInit).** Vérifier les analyses faites en TD sur `bufInit.c` à l'[exo 1 de la feuille de TD 6](#). Corriger le programme et tester avec le plugin EVA de FRAMA-C qu'il n'y a plus d'UB.