Construire du code sécurisé : à quoi sensibiliser nos étudiants ?

Marie-Laure Potet
Vérimag - Université Grenoble-Alpes - France
Email: Marie-Laure.Potet@univ-grenoble-alpes.fr

Laurent Mounier
Vérimag - Université Grenoble-Alpes - France
Email: Laurent.Mounier@univ-grenoble-alpes.fr

I. INTRODUCTION

Nous sommes en construction d'un cours, décliné à plusieurs niveaux, qui permet de sensibiliser les étudiants aux problèmes de développer du code non vulnérable, avec plusieurs objectifs, et en particulier non pas former les hackers d'aujourd'hui mais les développeurs responsables du futur.

Nous présentons notre expérience avec les objectifs de discussions suivants :

- Que doivent être les objectifs d'un cours Software Security, comment combiner au mieux attaque et défense ?
- Que serait un syllabus complet d'un tel cours ?
- Comment préparer au mieux les étudiants à la construction de solutions sécurisées et aux outils d'analyse ?
- Comment les sensibiliser au développement de futurs outils (langage, smart compilateur ...) qui permettront à terme de développer de manière sûre ?

Plus concrètement nous sommes intéressés pour partager du matériel pédagogique : contenu de cours, exemples, TP mais aussi les sites intéressants, proposant des guides et des solutions. Un cours de longue date avec des objectifs similaires, et bien détaillé : http://www.cs.ru.nl/E.Poll/ss/.

Le cours que nous proposons est décliné dans plusieurs parcours, avec différentes variantes, et pas seulement pour des parcours spécialisés en Cybersécurité. D'ailleurs une partie du contenu est plutôt du niveau M1 et concerne tout le monde. On évangélise donc un peu partout dès que possible [1] (- :

- Master Cybersecurity commun G-INP/UGA
- Formation en apprentissage M2 CSI
- Cursus Ensimag (filières Informatique niveaux M1 et M2)
- Séminaires à l'Esisar pour le parcours labellisé SecNumedu

1. Ecole d'été Cyber in Berry par exemple

II. UN CONTENU POSSIBLE

Ci-dessous les différents thèmes abordés, dont les objectifs sont détaillés dans les sous-sections suivantes :

- les vulnérabilités classiques des programmes
- les propriétés des langages aidant la sécurité
- les outils d'analyse existants
- les différentes formes de propriétés de sécurité
- les problèmes ouverts (ce sera à eux de proposer des solutions dans le futur !)

A. *les vulnérabilités classiques des programmes*

C'est ici une partie très classique d'un tel cours, on montre des exemples de vulnérabilités, mais aussi les raisons : les comportements non définis par exemple mais aussi les choix de sémantique qui peuvent poser problème aux programmeurs (par exemple les entiers qui " wrappent " [2]).

On montre aussi les sites qui expliquent les problèmes et proposent des solutions comme les règles de codage du CERT [3]. En Annexe A nous déclinons un petit exemple pour illustrer la démarche des aspects sémantiques des langages aux corrections et aux outils qui peuvent nous aider (voir sous-section II-C).

On parle aussi des différents niveaux de protection (analyse de code, monitoring à l'exécution, protection de la plate-forme). On ne résiste pas non plus à un petit

2. la sémantique en C pour les non signés est une sémantique de modulo, par exemple l'expression syntaxique $a + b$ s'évalue en $(a + b)\ mod\ 2^n$ avec n la taille du type. Dans le cas des signés le résultat est non défini.

3. https://www.securecoding.cert.org/confluence/display/seccode/SEI+CERT+Coding+Standards

TP buffer overflow (- : si on a du temps, sinon on fait cela au niveau du code source en regardant l'assembleur et la pile.

Messages qu'on veut faire passer : le hacking est peut être un art mais la défense aussi ! Connaitre les sites expliquant les vulnérabilités, chercher des solutions existantes, ne pas réinventer. Se poser des questions sur les langages utilisés (hors la facilité de développer).

B. les propriétés des langages aidant la sécurité

Ici l'idée est de prendre un peu de hauteur pour se poser des questions sur ce que serait un bon langage, et surtout en comprendre les forces et faiblesses. On présente donc les propriétés de haut-niveau de *type safety* (les valeurs en mémoire à l'exécution ne peuvent que être des représentations des valeurs du type calculé), *memory safety* (pas d'accès à des mémoires non attribuées) et *control flow safety*, ainsi que les mécanismes classiques de typage et de gestion mémoire. On insiste ici sur ce qui peut être pris en charge statiquement ou dynamiquement, comment et avec quelles garanties [4]. Exemples :

	memory safety	type safety
C, C ++	-	-
Java, C#, Rust	+	+

On peut aussi s'intéresser aux mécanismes proposés par les langages qui imposent des disciplines de programmation afin d'offrir des garanties (contrôle d'accès, immutabilité, etc). Deux exemples intéressants à étudier :

- Java qui offre les bonnes propriétés précédentes (revérifiées sur le byte-code) et permet ainsi de mettre en place du contrôle d'accès portant sur l'appel de méthodes.
- Rust qui offre une gestion fine de la mémoire tout en évitant les références pendantes en particulier.

Néanmoins aucun langage/plate-forme n'est parfait et il faut en maitriser/comprendre les faiblesses. Voir par exemple l'étude Javasec [5].

Messages qu'on veut faire passer : choisir un langage/API/plate-forme en se posant les bonnes questions en terme de sécurité et mettre en place les protections/vérifications adaptées à ce choix.

4. notions de correction et de complétude, rappel du théorème de Rice.
5. https://www.ssi.gouv.fr/agence/publication/securite-et-langage-java/

C. les outils existants

Un des objectifs de cet enseignement est également de montrer l'usage possible des différentes techniques classiques d'analyse de programme dans notre contexte d'aide à la production de code sécurisé. Il s'agit donc de sensibiliser les étudiants à l'existence de ces outils et de leur permettre de comprendre les apports et les limites de chaque technique. L'accent est donc mis ici davantage sur la détection de comportements non sûrs que sur la validation de propriétés fonctionnelles (c.f. Annexe A).

Une séquence pédagogique type consiste à introduire [6] en cours les notions sous-jacentes à différentes techniques d'analyse de code puis à faire expérimenter les étudiants avec les outils associés sur des (petits) exemples orientés " sécurité ".

Nous abordons donc classiquement :

1) le fuzzing : c'est une technique " incontournable " en sécurité ; on insiste ici sur l'automatisation et le passage à l'échelle de l'analyse, mais aussi sur le caractère partiel et assez aléatoire des résultats susceptibles d'être obtenus. Nous utilisons AFL [7] pour l'expérimentation.

2) l'exécution symbolique ou dynamique-symbolique : présentée comme une forme " avancée " de fuzzing, cela nous permet d'illustrer l'intérêt des techniques SMT pour la génération de séquences de tests difficilement atteignables par fuzzing. Concrètement l'utilisation d'outils (non dédiés sécurité) comme PathCrawler [8] ou Klee [9] permet de montrer facilement la détection de vulnérabilités qui échappent à AFL.

3) l'interprétation abstraite : le développement de versions orientées ' sécurité " d'outils commerciaux conçus initialement pour la vérification fonctionnelle reflète une évolution importante dans ce domaine ; l'analyse statique de code fait maintenant partie de la panoplie du développeur " sécurisé " et il convient donc de montrer à nos étudiants les forces et faiblesses de cette approche. Nous utilisons notamment Frama-C [10], en particulier les *plugins* RTE (pour la génération automatique d'assertions sur les erreurs possibles à l'exécution), EVA (analyse de valeurs) et WP (calul de plus faible précondition).

Messages qu'on veut faire passer : une bonne compréhension de l'offre existante des différentes ap-

6. ou ré-introduire, selon les cursus
7. http://lcamtuf.coredump.cx/afl/
8. http://pathcrawler-online.com:8080
9. https://klee.github.io
10. https://frama-c.com/

proches et des différents cas d'usage (quel outil pour quels objectifs dans le processus de développement sécurisé ?).

https://fr.mathworks.com/help/bugfinder/
defects-runtime-checks-misra-coding-rules.html

D. les différentes formes de propriétés de sécurité

Vérifier que les programmes ne contiennent pas de vulnérabilités directement exploitables ne se réduit pas aux comportements non définis. Avoir du code sécurisé consiste à vérifier différents aspects, dépendant des fonctionalités attendues, de l'environnement (quelles garanties offre-t-il, quelles sont les possibilités d'interaction) et du modèle d'attaquant qu'on veut prendre en compte (que peut-il faire avec quel degré d'expertise et de connaissance de la cible).

On peut classifier les propriétés en plusieurs classes :

1) Propriétés fonctionnelles et règles de codage: Dans ce cadre on veut s'assurer que les programmes font bien ce qu'il devraient faire vis-à-vis des fonctions implémentées, du contrôle d'accès …. Ceci correspond à des propriétés classiques de sûreté.

Il existe aussi en sécurité un certain nombre de règles de codage soit sur les langages, soit sur la programmation [11], comme les fonctions bannies, les chemins en erreur qui peuvent finir l'exécution en mode privilégié ou la création de fichiers temporaires qui peuvent faire fuir de l'information.

Par exemple la norme ISO/IEC TS 17961 est un standard international pour le langage C et qui vise spécifiquement les problèmes de sécurité, pour du C standardisé, et qui peuvent être vérifiés en analyse statique. L'outil BugFinder de Mathworks implémente cette norme en proposant un grand nombre d'analyses dédiées [12].

2) Propriétés sur les flux: En dehors des vulnérabilités possibles, on peut s'intéresser à l'exploitabilité de ces vulnérabilités par un attaquant. Une analyse classique est le calcul de teinte qui permet de calculer les objets qu'un attaquant peut contrôler de l'extérieur. La teinte se généralise à la non-interférence qui permet de calculer si des informations secrètes peuvent fuir à travers le flot de données et de contrôle (confidentialité) ou bien si des données sensibles peuvent être modifiées (intégrité). Vérifier ces propriétés demande des analyses adaptées, en particulier qui portent sur des ensembles de traces.

Hormis la cryptographie, des applications importantes sont les preuves d'isolation, qui permettent par exemple de montrer que deux applications hébergées sur une même plate-forme (ou normalement isolées par virtualisation) ne peuvent exploiter ou modifier leurs données respectives.

3) Propriétés structurelles: En particulier dans le domaine d'application gérant des services de sécurité (authentification, chiffrement, protection des données, …) des propriétés structurelles ou des règles de codage particulières peuvent être imposées. On trouve par exemple le nettoyage effectif de la mémoire (remise à 0) ou l'exécution en temps constant [13] pour éviter des attaques en brute force. Le site cryptocoding.net [14] liste un ensemble de bonnes pratiques pour ce type d'applications. Les propriétés structurelles posent deux types de problèmes : elles ne sont généralement pas préservées par un processus non maitrisé de compilation et leur vérification demande des analyses ad-hoc.

Messages qu'on veut faire passer : garantir la sécurité du code ne se réduit pas à la recherche de vulnérabilités classiques mais nécessite de faire un cahier des charges précis, en identifiant ce qui doit être protégé et contre quoi, et en prenant en compte le contexte d'utilisation (d'où peuvent provenir les attaques).

E. les problèmes ouverts (où ils devront trouver eux les solutions)

On liste ici quelques problèmes ouverts qui les concerneront (liste non complète).

1) Evaluer/caractériser l'attaquant: La caractérisation de l'attaquant contre lequel on veut se protéger est un point important pour mener une analyse de sécurité. Les attaques " standards " consistent généralement à exploiter les E/S du programme en fournissant des entrées malveillantes mais on peut aussi vouloir se protéger contre des attaques physiques ou bien contre des applications malveillantes, hébergées sur une même plate-forme d'exécution (mobile par exemple) ayant potentiellement des failles exploitables. Construire des modèles d'attaquant potentiellement actif pour le code, et apprendre à raisonner avec, comme ceci est fait pour les modèles Dolev-Yao dans le domaine de la vérification de protocoles, est un domaine ouvert.

11. https://security.web.cern.ch/security/recommendations/en/checklist_for_coders.shtml
12. https://fr.mathworks.com/help/bugfinder/defects-runtime-checks-misra-coding-rules.html
13. ou plus finement en temps constant vis-à-vis des données secrètes
14. https://cryptocoding.net/index.php/Coding_rules

2) Des chaines de développement qui prennent en compte la sécurité : Les langages et surtout les compilateurs ne nous aident actuellement pas pour garantir des propriétés de " sécurité " sur les programmes. En particulier les compilateurs n'ont pas été conçus pour prendre en compte la sécurité : en plus des propriétés non préservables par compilation (temps constant d'exécution par ex.) les optimisations peuvent éliminer des protections vues comme du code mort (mise à 0, contre-mesures logicielles prévues par le développeur, ...). Faut-il repenser les compilateurs ?

La défense en profondeur consiste à écrire du code offrant de bonnes garanties mais aussi à combiner ces garanties avec de l'instrumentation à l'exécution (compilation ou plate-forme d'exécution) et, dans une moindre mesure, avec les mécanismes de protection de la plate-forme, ceci en fonction des possibilités de l'attaquant (par ex. observation ou modification de l'exécution). Définir des processus permettant d'analyser globalement les garanties sur un programme est un problème ouvert.

F. Cycle de vie du logiciel sécurisé

Le logiciel assure de plus en plus de services de sécurité (cryptographie, protection des données, identification/authentification, ...) dans des composants de plus en plus accessibles et donc de plus en plus attaquables. La mise à jour de ces programmes et l'application de patchs sont des opérations difficiles et sensibles. Gérer le cycle de vie et la sécurité des logiciels, ainsi que des programmes effectuant les mises à jour (firmware update), est un problème ouvert.

Messages qu'on veut faire passer : de l'art de l'attaque on est passé durant la dernière décennie à un début d'ingénierie pour la défense (pratiques de bonne programmation, outils ...), sensibiliser les étudiants qu'il reste beaucoup de choses à faire/repenser.

III. Conclusion

La section ci-dessus montre un aperçu des messages que nous voulons faire passer et permet, en plus d'étudier la construction de codes ayant de bonnes propriétés en sécurité, d'aborder des problématiques générales en sécurité. Par exemple la notion de contrôle d'accès peut être abordé en système mais aussi dans les programmes et la non-interférence est la version programme des modèles de politiques de contrôle de flux à la Bell-Lapadula ou Biba. De la même manière nous abordons le problème du modèle d'attaquant (que peut-il observer/contrôler), problématique généralement plutôt vue pour les primitives et protocoles cryptographiques, et en lien avec ce modèle, l'élaboration précise d'un cahier des charges en sécurité. Aborder ces notions dans différents cours (ou sous différents angles de vues) renforce l'aspect important et transverse de ces questions et, voire, permet de décloisonner l'enseignement de la sécurité (cours très pratiques / cours très théoriques).

Certains points ne sont pas abordés ici et constituent plutôt un cours avancé en sécurité du logiciel (par exemple pour les masters spécialisés) : protection du code, offuscation/déoffuscation, technique d'analyse de binaire et reverse ...

En conclusion ce qui nous intéresse de discuter avec la communauté :

- Etablir ensemble une liste des compétences visées
- Partager du matériel pédagogique (sujets mais aussi objectifs visés)
- Décliner ces cours sous différentes formes pour différents publics
- Avoir des avis/contributions de nos collègues (- :

Annexe A
Un exemple illustratif de la démarche

```
int g(int a, int b)
{   if (a < 0)
       {printf("arguments pas bons \n");
       return (-1) ;
    if (b <=0)
       {printf("arguments pas bons \n");
       return (-1) ;
     if (a+b < 0)
       {printf("erreur de débordement \n");
       return (-1) ;
       printf("pas d'erreur détectée \n");
      return(a+b) ;
}
```

Si on considère l'appel `g(2, INT_MAX)` on peut avoir des résultats différents ("erreur de débordement" ou "pas d'erreur détectée") suivant les compilateurs et les options utilisées (pas d'option, -02, -fno-strict-overflow). On tombe dans les 203 comportements non définis du langage C (- :!!! Des outils comme RTE et EVA vont détecter le problème comme illustré Figure 1. On peut alors utiliser les bons conseils du CERT (mettre la règle exacte) pour réécrire le test sans passer par un comportement non défini. Nous obtiendrons alors à l'exécution le résultat prévu indépendemment des optimisations et RTE et EVA pourront valider le code proposé (Figure 2).

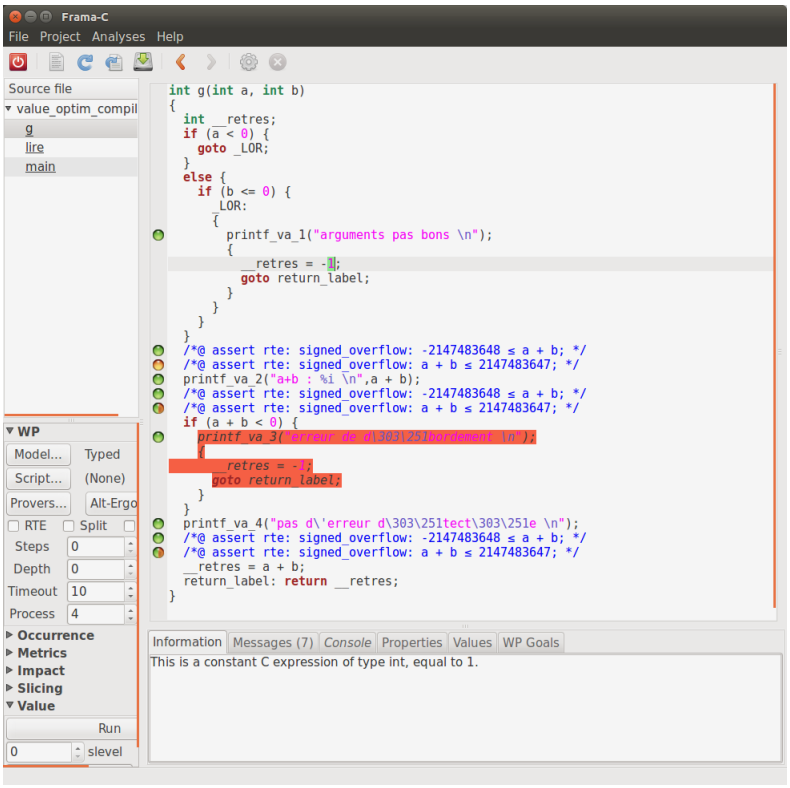

FIGURE 1. Détection de code mort, on ne peut pas avoir a+b négatif

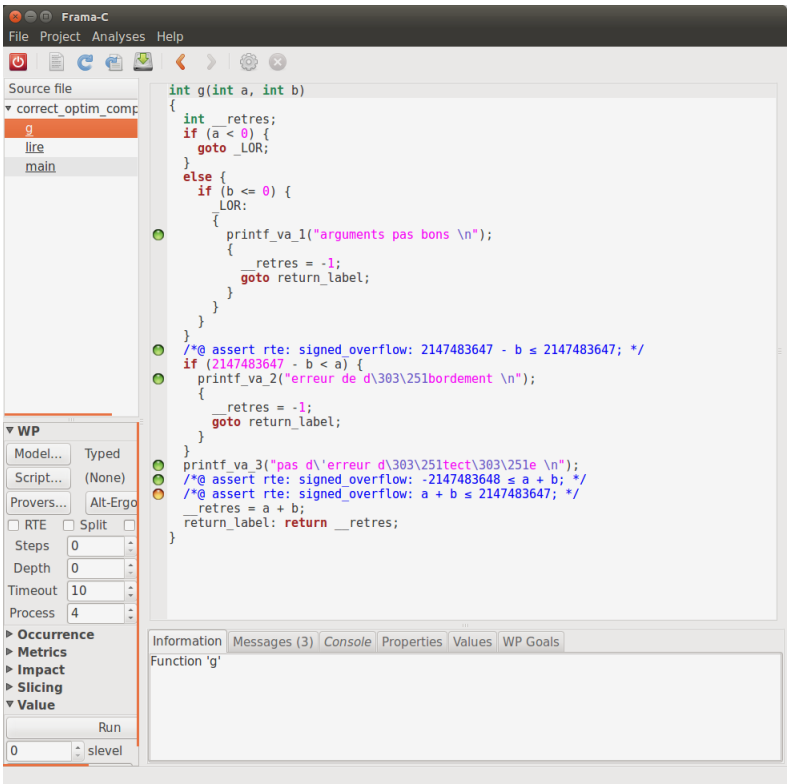

FIGURE 2. Et on est content !