Fiches pédagogiques sur la sécurité des logiciels

CyberEdu

CyberEdu

La sécurité par l'enseignement supérieur des NTIC

Ce document pédagogique a été rédigé par un consortium regroupant des enseignants-chercheurs et des professionnels du secteur de la cybersécurité.

Version 1.3 — Février 2017

Table des matières

Introduction

Les fiches pédagogiques présentées dans ce guide ont pour objectif de mettre en avant les éléments fondamentaux de la sécurité du développement logiciel qui peuvent être présentés à des étudiants de l'enseignement supérieur non-spécialistes du domaine. Les fiches apportent à l'enseignant des repères pédagogiques mais ne peuvent constituer à elles seules un support d'apprentissage pour l'enseignant.

Prérequis pour les étudiants

Chacune des quatre fiches indique les prérequis nécessaires à sa compréhension. Les prérequis portent sur les connaissances fondamentales en développement logiciel : les vulnérabilités logicielles standard, la sécurité des différents langages, les bonnes pratiques de développement et les outils d'analyse de vulnérabilités.

Prérequis pour les formateurs

Les fiches apportent des repères pédagogiques aux enseignants, en présentant de manière structurée et concise les sujets importants de la sécurité du développement logiciel.

Ces fiches ne constituent pas un cours complet sur la sécurité du développement logiciel. Il n'est pas demandé à l'enseignant de parfaitement maîtriser le domaine de la sécurité, mais il devra se renseigner sur les sujets présentés pour pleinement exploiter les fiches pédagogiques. Une bonne maîtrise des différents langages de programmation étudiés est fortement conseillée.

Utilisation du guide pédagogique

Ce document contient quatre fiches pédagogiques à destination des enseignants en développement logiciel dans l'enseignement supérieur. Chaque fiche permettra à l'enseignant d'illustrer son cours de développement avec des notions de sécurité. Typiquement, l'enseignant consacrera une trentaine de minutes à la sécurité à la fin de chacun de ses chapitres. Les fiches peuvent être présentées en tout ou partie, dans l'ordre approprié à l'enseignement et aux étudiants visés.

Numéro	Sujet	Durée (min)	Prérequis
Fiche 1	Vulnérabilités logicielles standard - Corruption de mémoire par débordement de pile d'appel - Injection SQL - Programmation défensive et conception orientée sécurité	30-45	Compilation Architecture des ordinateurs
Fiche 2	Sécurité intrinsèque des différents langages de programmation - Différences entre programmation défensive et mécanismes de sécurité - Mécanismes de sécurité des différents langages - Notion de programme « défini »	45-60	Programmation (niveau intermédiaire)
Fiche 3	Pratiques et méthodologies pour le développement sécurisé - Surface d'attaque d'un logiciel - Principe des moindres pivilèges - Paramétrage des environnements d'exécution et de compilation	15-30	Développement logiciel (notions)
Fiche 4	Outils d'analyse de vulnérabilités - Recherche de vulnérabilités logiciels par *fuzzing* - Analyse de teintes - Types d'alertes en analyse statique	30-45	Compilation

1 Fiche 1 : Vulnérabilités logicielles standard

1.1 Thématique

Thématique	Vulnérabilités logicielles standard	**Numéro de fiche**	1	**Mise à jour**	05/01/2016

Ce cours aborde les sujets suivants :
– la corruption de la mémoire par débordement de pile d'appel ;
– l'injection SQL ;
– la programmation défensive et conception orientée sécurité.

1.2 Thèmes des cours visés

– Programmation
– Compilation
– Architecture des ordinateurs

1.3 Volume horaire

Entre 30 et 45 minutes.

Deux versions restreintes sont envisageables en évoquant uniquement la corruption de la mémoire par débordement de pile d'appel ou uniquement les injections SQL. Les contre-mesures de programmation défensive devront être évoquées dans les deux cas.

1.4 Prérequis / corequis

Cette activité s'intègre dans un cours de programmation ou en complément d'un cours de compilation, voire d'architecture des ordinateurs. Dans ces derniers cas, on pourrait supprimer l'étude d'une injection SQL et étoffer la présentation d'une corruption de la mémoire avec l'utilisation d'un débogueur.

Si la fiche est utilisée dans un cours de programmation, il faudra veiller à ce que la notion de convention d'appel ait été préalablement présentée lors d'un cours d'architecture ou de compilation.

1.5 Objectifs pédagogiques

L'objectif pédagogique de cette activité est de sensibiliser les étudiants aux vulnérabilités logicielles les plus classiques, en insistant sur les erreurs de conception qui peuvent mener à de telles vulnérabilités.

1.6 Conseils pratiques

Un partie des mots clés de cette fiche sont parfois connus des étudiants informaticiens mais nous conseillons de prendre le temps de bien faire comprendre les mécanismes mis en jeux. La respons-

abilité du programmeur à utiliser des contre-mesures adaptées doit aussi être soulignée.

1.7 Description

L'enseignant précise au préalable le modèle d'attaquant considéré ici : le programme de l'utilisateur est exécuté dans un environnement hostile où l'attaquant peut soumettre des entrées dans le but de pousser le programme à la faute.

Deux exemples classiques d'attaques pourront être proposés. Le premier, une **corruption mémoire par débordement de tampon** (*buffer overflow*), est une des attaques les plus classiques. On peut par exemple l'illustrer en prenant un petit programme C.

```
#include <stdio.h>
#include <string.h>

int verifie_mot_de_passe() {
        int res = 0;
        char tampon[10];
        scanf("%s", &tampon);
        if (strcmp (tampon, "secret") == 0)
                res = 1;
        return res;
}

int main() {
        if (verifie_mot_de_passe ()) {
                // code privilégié
        }
        return 0;
}
```

En exécutant le programme et en fournissant une entrée suffisamment longue, il est facile de provoquer une corruption de la mémoire *(segmentation fault)*. En tapant « abcdefghijklmn », un attaquant peut même changer la valeur de retour sans connaître le mot de passe. Attention, le changement du flot d'exécution ne pourra pas forcement être observé, car il dépend à la fois des options de compilation (nous conseillons l'option `-O0` pour réaliser cette expérience) et de l'architecture matérielle considérée. Pour faire comprendre le phénomène, il convient de rappeler les bases des conventions d'appels en génération de code. Un schéma de l'état de la pile d'appel après l'appel de fonction `scanf` permettra de faire comprendre le risque d'écraser l'adresse de retour de l'appel de `verifie_mot_de_passe()` et les variables locales de la fonction. En fonction du volume horaire alloué, une utilisation d'un débogueur pourra être proposée pour suivre pas-à-pas une attaque par débordement de tampon menant à une exécution de code arbitraire [1] (voir notamment la section 3.12 intitulée *Out-of-Bounds Memory References and Buffer Overflow*).

Une protection efficace contre un tel débordement est l'option `-fstack-protector-all` de gcc, qui implémente les canaris (une case mémoire insérée dans la pile pour vérifier que celle-ci n'a pas été modifiée[1]).

1. Il est intéressant de noter que certaines distributions Linux l'activent par défaut, rendant la démonstration impossible. Pour la désactiver *le temps du TP uniquement*, il faut ajouter `-fno-stack-protector`.

La deuxième vulnérabilité qui pourra être présentée est une corruption d'argument de type **injection SQL** dans un script PHP. Cette attaque est très connue et largement documentée. L'attaquant soumet une chaîne de caractères sur un formulaire et cette chaîne est ensuite utilisée pour construire une requête SQL sur une base de donnée.

Supposons que la requête soit de la forme :

```
SELECT * from table WHERE login='$login' AND password='$pwd'
```

avec `$login` et `$pwd` des entrées directement fournies sous forme de chaînes de caractères par un attaquant potentiel. Si la première chaîne est `' OR 1=1#`, puisque le symbole # indique le début d'un commentaire SQL, la requête transmise à la base de données sera simplement :

```
SELECT * from table WHERE login='' OR 1=1
```

Cette requête devient triviale : elle ignore la vérification de mot de passe et interroge tous les noms de *login* de la base.

Il conviendra de donner un exemple classique comme celui-ci mais surtout d'insister sur les leçons à tirer de cette vulnérabilité. Ceci fait l'objet de la deuxième partie de la fiche. Nous recommandons de ne pas dissocier les deux parties. La liste des attaques par injection SQL et les dégâts occasionnés sont immenses (vol, altération et/ou destruction de données). Le cours pourra s'orienter vers l'histoire des failles de type *Cross-site scripting* pour prolonger l'étude de ce problème [2]. Pour illustrer des injections reposant sur d'autres langages, il est aussi possible de présenter des vulnérabilités reposant sur des langages de script comme bash (voir par exemple la vulnérabilité CVE-2010-3088).

Face à ces vulnérabilités, la première contre-mesure est de penser, très tôt durant la conception, au modèle d'attaquant. Si certaines données du programme proviennent d'un environnement potentiellement hostile, le programmeur doit défendre son programme vis-à-vis de ces entrées. Les hypothèses réalisées sur les entrées sensibles doivent non seulement être explicités, mais testées à l'exécution grâce à une programmation défensive. Chaque langage de programmation possède des contre-mesures spécifiques, par exemple les bibliothèques C gérant explicitement la taille des chaînes de caractères, des compilateurs insérant des détecteurs de corruption de pile comme les canaris [1] ou les bibliothèques PHP nettoyant les entrées sensibles. Dans le cas des requêtes SQL, on peut citer le mécanisme des requêtes préparées *(prepared statements)*.

La programmation *défensive* peut parfois s'avérer délicate. On pourra ainsi mentionner les attaques de type TOCTOU [3] (*Time of check to time of use*) où une donnée sensible est testée par le programmeur avant son utilisation mais l'attaquant réussit à modifier la donnée entre ces deux étapes. La forme la plus courante de cette vulnérabilité est un programme défensif où la validité du fichier que l'on souhaite modifier est testée avant son usage :

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
        FILE *fd;
        if (access(filename, W_OK) == 0) {
                printf("access␣granted.\n");
                fd = fopen(filename, "wb+");
                if (fd != NULL) {
                        /* écriture dans le fichier */
                        fclose(fd);
                }
```

```
        }
        ...
        return 0;
}
```

Un tel programme tournant sous l'identité `root` permet en apparence de vérifier (à l'aide de l'appel système `access`) que le fichier est inscriptible avant de l'ouvrir (appel à `fopen`)[2]. Cependant, si le fichier cible est substitué (par le jeu d'un lien symbolique par exemple), entre le moment de la vérification et le moment de la modification, l'attaquant réussira à provoquer la modification sur lequel il n'avait pas nécessairement lui même les droits en écriture. Là encore, la capacité de l'attaquant à interagir avec le programme utilisateur doit être anticipée. Pour cet exemple particulier, il est possible d'explorer les options des fonctions d'ouverture de fichier telles que `O_NOFOLLOW` ou encore de travailler sur la gestion des identités et privilèges. L'enseignant souhaitant présenter d'autres exemples de vulnérabilité TOCTOU pourra s'appuyer sur le site CWE-367 [3].

1.8 Matériels didactiques et références bibliographiques

Afin de préparer son cours, l'enseignant pourra s'appuyer sur l'ouvrage de Bryant et O'Hallaron [1] qui comporte de nombreuses illustrations liées à la sécurité logicielle. Cette fiche ne présente qu'un aperçu partiel des vulnérabilités classiques. D'autres types de vulnérabilités, accompagnés d'exemples, sont disponibles sur la base de donnée CWE [4].

[1] Randal E. Bryant, David R. O'Hallaron, *Computer Systems : A Programmer's Perspective*, 2nd Edition, Prentice Hall, 2011.

[2] *CWE-79 : Improper Neutralization of Input During Web Page Generation (Cross-site Scripting)*, `https://cwe.mitre.org/data/definitions/79.html`

[3] *CWE-367 : Time-of-check Time-of-use (TOCTOU) — Race condition*, `https://cwe.mitre.org/data/definitions/367.html`

[4] *2011 CWE/SANS Top 25 Most Dangerous Software Errors*, `https://cwe.mitre.org/top25/index.html`

2. En effet, les privilèges de super-utilisateurs lui permettent d'ouvrir des fichiers pour lesquels la permission en écriture n'est pas accordée.

2 Fiche 2 : Sécurité intrinsèque des différents langages de programmation

2.1 Thématique

Thématique	Sécurité intrinsèque des langages de programmation	**Numéro de fiche**	2	**Mise à jour**	05/01/2016

Ce cours aborde les sujets suivants :
- les différences entre programmation défensive à la charge du programmeur et mécanisme de sécurité inhérent au langage de programmation utilisé ;
- l'étude des distinctions entre les langages proposant des mécanismes de sécurité robustes et ceux qui sont plus fragiles ;
- la notion de programme « bien défini » et compréhension des libertés que s'autorisent les compilateurs pour compiler les programmes « mal définis ».

2.2 Thèmes des cours visés

- Programmation
- Génie logiciel

2.3 Volume horaire

Entre 45 et 60 minutes.

2.4 Prérequis / corequis

Cette activité s'intègre idéalement dans un cours de programmation intermédiaire ou avancé, quand les étudiants ont déjà été sensibilisés à plusieurs langages de programmation.

Un cours de méthode formelle abordant le typage et la sémantique dans les langages de programmation, permettra de prendre davantage de recul dans cette étude multi-langage.

2.5 Objectifs pédagogiques

L'objectif pédagogique de cette activité est de présenter aux étudiants les mécanismes de sécurité inhérents à certains langages de programmation.

2.6 Conseils pratiques

L'étude simultanée de plusieurs langages de programmation n'est pas un exercice pédagogique simple. Il convient de ne pas rentrer dans une discussion stérile sur l'existence d'un hypothétique langage surpassant tous les autres.

2.7 Description

Cette fiche pédagogique prend résolument le parti de comparer des langages de programmation. C'est un exercice difficile et il est important de rappeler en préambule que l'objectif n'est pas de critiquer tel ou tel langage de programmation, mais plutôt de comparer dans un même cours, les contributions et les limites de certains langages pour le développement des logiciels dans un contexte de cybersecurité.

2.7.1 Modèle d'attaquant

Dans le modèle de cette fiche, une partie du programme de l'utilisateur peut contenir du code malveillant. Ce modèle couvre le cas d'une utilisation d'une bibliothèque « piégée » par l'attaquant. Cette fiche propose aussi des éléments de discussion pour identifier des erreurs de programmation pouvant mener à des vulnérabilités. Dans tous les cas, il est nécessaire d'effectuer une relecture de code minutieuse et de comprendre la portée des interactions avec une bibliothèque.

2.7.2 Garanties offertes par un système de type

Chaque langage suit une discipline de typage plus ou moins forte, en offrant plus ou moins de garantie à la compilation et à l'exécution. En fonction du langage enseigné, il est intéressant de commenter les choix suivis par ce langage vis-à-vis des problématiques de sécurité.

Langage *faiblement typé / vérification statique* (C, C++) :

Les annotations de type permettent de lever les ambiguïtés de certaines opérations polymorphes (la division entière/flottante, ou encore le type des accès mémoires) mais le compilateur fait confiance au programmeur et ne vérifie pas (ni statiquement, ni dynamiquement) la cohérence globale des annotations. Une erreur de type (transtypage par exemple) peut donner lieu à un dysfonctionnement, voire une vulnérabilité silencieuse (sans aucune remontée d'alerte, ni à la compilation ni à l'exécution). Par exemple la fonction C suivante [1] a pour but de réaliser une écriture d'une valeur `val` dans le tableau `tab` de taille `size`, à l'indice `ind`, mais en vérifiant au préalable que l'indice réside dans les bornes du tableaux.

```
void safewrite(int tab[], int size, signed char ind, int val) {
        if (ind < size) {
                tab[ind] = val;
        } else {
                printf("Out␣of␣bounds\n");
        }
}
```

Malheureusement, un appel comme `safewrite(tab,120,128,1)` ne va pas générer de message d'erreur, bien que 128 soit plus grand que la taille 120. Cette erreur inattendue s'explique par la conversion implicite qui est réalisée lors de l'appel de la fonction, afin de transtyper 128 vers un `signed char`.

Plus généralement, les programmes qui ont un comportement « indéfini » vis-à-vis de la norme du C, laissent une grande liberté au compilateur. Ce dernier peut librement transformer une petite erreur de programmation en faille de sécurité. Le code suivant [1], qu'on trouvait il y a quelques années dans les sources du noyau Linux, illustre un de ces comportements indéfinis :

```
struct tun_struct *tun = __tun_get(tfile);
struct sock *sk = tun->sk;
if (!tun)
   return POLLERR;
/* utilisation de *sk pour des écritures mémoires */
```

Le code de la fonction commence par déréférencer le pointeur `tun`. Cette opération étant *indéfinie* si le pointeur est nul, le compilateur en déduit que le pointeur n'est pas nul (sans quoi le programme n'a pas de sens). De cela, le compilateur déduit ensuite que le test de la ligne suivante (le pointeur est-il nul ?) est toujours faux, ce qui permet une optimisation, la suppression pure et simple du bloc `if/return`. Ce code précis a donné lieu à une vulnérabilité en 2009 (CVE-2009-1897).

L'article [2] propose de nombreux exemples de programmes C dont le comportement indéfini autorise le compilateur à effectuer des optimisations contre-intuitives pour le programmeur, voire dangereuses pour le programme compilé. L'exemple suivant provient également du noyau Linux. Les entrées `offset` et `len` de cette fonction doivent être validées : les deux entiers doivent être positifs et leur somme ne doit pas provoquer de débordement. Le test `(offset + len < 0)` a pour but de tester si la somme des deux entiers a débordé (ce qui conduit sur certaines architectures à produire un nombre négatif). Malheureusement, un compilateur comme GCC peut déduire de l'échec du premier test `(offset < 0 || len <= 0)` que les deux variables sont des valeurs positives et que la somme est donc nécessairement positive, c'est-à-dire que le test suivant `(offset + len < 0)` échouera : le compilateur peut donc optimiser le code en retirant le second test. En effet, la norme C indique que le débordement d'entiers signés n'est pas défini. Le résultat est que la fonction devient vulnérable si on lui soumet deux entiers dont la somme provoque un débordement.

```
int do_fallocate(..., loff_t offset, loff_t len)
{
  struct inode *inode = ...;
  if (offset < 0 || len <= 0)
    return -EINVAL;
  /* Check for wrap through zero too */
  if ((offset + len > inode->i_sb->s_maxbytes)
      || (offset + len < 0))
    return -EFBIG;
  ...
}
```

En réalité, le noyau Linux est compilé avec l'option `-fno-strict-overflow` pour éviter ce genre d'optimisations. Ce type d'exemple demande une grande attention pour être bien compris. Il convient d'éviter (ou d'expliquer) la réaction simpliste qui consiste à penser que le compilateur à mal agi. Ce n'est pas le compilateur qui est en faute, c'est le programmeur qui ne maîtrise pas les subtilités de la norme du langage C.

***Langage faiblement typés / vérification dynamique* (Python, PHP, JavaScript) :**

Dans ce cas, un script demande peu d'annotations au programmeur et l'interpréteur n'effectue que très peu de vérifications. Les valeurs manipulées à l'exécution sont étiquetées avec leurs types, et des vérifications de type sont réalisées pendant l'exécution. Même si elles sont retardées, les vérifications de type ont lieu, ce qui permet de rattraper certaines erreurs de programmation. Cependant, la revue de code peut être passablement entravée par le manque d'annotations de type. L'exemple suivant [1] est un fragment de programme JavaScript illustrant des phénomènes de surcharges contre-intuitifs.

```
a=1; b=2; c='Foo';
print(a+b+c); print(c+a+b); print(c+(a+b));
```

En effet, son exécution provoque les affichages 3Foo, Foo12 and Foo3. L'opérateur + est tantôt interprété comme une concaténation de chaînes, tantôt comme une addition d'entiers. Ce type de langage fournit généralement peu de mécanismes permettant d'assurer les propriétés d'encapsulation et d'immuabilité présentées dans la suite de cette fiche.

***Langage fortement typé / vérification statique* (Java, Scala, ML) :**

Les données doivent être déclarées avec des types relativement rigides mais ces annotations favorisent une vérification des types à la compilation. Les programmes qui sont déclarés bien typés ont tous une sémantique bien définie et le compilateur s'engage à la préserver. La revue de code est facilitée et les valeurs n'ont plus besoin d'être entièrement étiquetées avec leurs types à l'exécution (gain de performance). À noter, cependant que la sécurité de ces langages reposent sur la fiabilité de l'algorithme de vérification de type. Une erreur dans ce dernier peut laisser « carte blanche » à un programme malveillant, car certains types ne sont plus présents à l'exécution. Il convient aussi d'utiliser avec grande précaution (voire d'éviter) les mécanismes *unsafe* de certains de ces langages car ils contournent la vérification de type.

2.7.3 Encapsulation

Les langages qui permettent ce mécanisme (via la programmation par objet ou un système de module par exemple), facilitent grandement la relecture de code et réduisent les risques de corruption d'une information transmise vers des bibliothèques tierces (notamment en permettant d'implémenter une propriété d'immuabilité ou des invariants permettant de garantir la cohérence interne). Une grande prudence est néanmoins nécessaire pour comprendre les limites des mécanismes réellement proposés par les différentes plateformes d'exécution, car l'encapuslation est avant tout pensée comme un mécanisme de sûreté logicielle, et non un mécanisme de sécurité.

On peut en particulier citer l'annotation `private` d'un champ en Java. L'annotation ne peut, à elle seule garantir la confidentialité de la valeur de ce champs (tout dépend des accesseurs proposés). De plus, l'introspection Java, associée à une configuration trop faible du mécanisme de contrôle d'accès de Java, peut permettre la manipulation de ces champs par des classes tierces, annulant toutes les propriétés de cette annotation. L'exemple suivant [1] illustre ce phénomène.

```
import java.lang.reflect.*;

class Secret { private int x = 42; }

public class Introspect {
  public static void main (String[] args) {
    try { Secret o = new Secret();
          Class c = o.getClass();
          Field f = c.getDeclaredField("x");
          f.setAccessible(true);
          System.out.println("x="+f.getInt(o));
        }
    catch (Exception e) { System.out.println(e); }
  }
 }
```

2.7.4 Immuabilité

L'échange de valeurs immuables est fortement conseillé pour assurer l'intégrité des programmes et éviter des attaques par effet de bord (dans le style des attaques TOCTOU, présentées dans la fiche *Vulnérabilités logicielles standard*). C'est un style de programmation plébiscité [5] pour la programmation concurrente afin d'éviter les situations de compétitions (*data races*). Il s'agit d'une propriété forte, à la charge du programmeur et chaque langage propose des mécanismes plus ou moins robustes pour assurer cette propriété. Les langages fonctionnels assurent cette propriété par défaut, mais même pour cette famille de langage, il faut rester vigilant sur certains détails d'implémentation. C'est ainsi que le langage OCaml propose des chaînes de caractères mutables, contrairement à Java. La fonction OCaml suivante illustre le problème.

```
let f (s:string) =
  if validate s then begin bad(); use(s); end
```

Dans cet exemple, la fonction `bad()` peut tout à fait modifier par effet de bord la valeur de la chaîne `s` et ainsi contourner la validation de la ligne qui précède son appel. C'est par exemple le cas si la fonction `f` est appelée sur une variable globale telle qu'une des chaînes du tableau `Sys.argv`.

2.7.5 Surcharge

Ce mécanisme a pour but de simplifier l'écriture de programme, en laissant le compilateur ou l'interpréteur réaliser les bonnes conversions de type pour lever l'ambiguïté sur certaines opérations, ou permettre un certain polymorphisme. Malheureusement, assez peu de programmeurs maîtrisent suffisamment les finesses d'un langage de programmation (quel qu'il soit) pour ne pas être surpris par certains comportements. La plus grande prudence est donc conseillée.

Un panorama des « faux-amis » de chaque langage pourra finir de convaincre les étudiants. L'exemple précédent de la surcharge de l'opérateur + en JavaScript, s'applique encore ici. Par ailleurs, en Java, la surcharge d'une méthode peut par exemple rendre certains appels de méthode difficiles à comprendre, en présence de transtypages implicites.

```
class Confuser {
  static void A(short i) { System.out.println("Foo"); }
  static void A(int i) { System.out.println("Bar"); }

  public static void main (String[] args) {
    short i = 0;
    A(i);     // affiche "Foo"
    A(i+i);   // affiche "Bar"
    A(i+=i); // affiche "Foo"
}}
```

Pour le deuxième appel, le résultat de l'addition a le type `int`. Pour le troisième appel, la même addition est effectuée mais son résultat est ensuite implicitement reconverti vers le type `short` pour permettre l'affectation de la variable `i`, déclarée avec le type `short`.

En conclusion, cette fiche doit encourager l'étudiant à rester critique sur les langages de programmation qu'il utilise. Une bonne connaissance de leurs forces et de leurs faiblesses est bien plus utile que la recherche d'un hypothétique langage de programmation sans faiblesses.

2.8 Matériels didactiques et références bibliographiques

[1] Éric Jaeger, Olivier Levillain, *Mind your languages(s) – A discussion about languages and security*, 2014 IEEE Security and Privacy Workshops

[2] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, M. Frans Kaashoek, *Undefined Behavior : What Happened to My Code ?*, APSys'12

[3] John Mitchell, *Concepts in Programming Languages*, Cambridge Press

[4] Joshua Bloch, Neal Gafter, Java *Puzzlers : Traps, Pitfalls and Corner Cases*, Addison-Wesley Professional

[5] Joshua Bloch, *Effective* Java, Addison-Wesley Professional

3 Fiche 3 : Pratiques et méthodologies pour le développement sécurisé

3.1 Thématique

Thématique	Pratiques et méthodologies pour le développement sécurisé	**Numéro de fiche**	3	**Mise à jour**	05/01/2016

Ce cours aborde les sujets suivants :
- la notion de surface d'attaque d'un logiciel ;
- le principe des moindres privilèges ;
- le paramétrage des environnements d'exécution et de compilation.

3.2 Thèmes des cours visés

- Programmation
- Génie logiciel

3.3 Volume horaire

Entre 15 et 30 minutes.

3.4 Prérequis / corequis

Cette activité s'intègre idéalement dans un cours de génie logiciel standard.

3.5 Objectifs pédagogiques

L'objectif premier de cette fiche est de faire comprendre que la problématique de sécurité d'un logiciel doit se poser dès sa conception et non uniquement après. Les techniques standards d'ingénierie logicielles doivent être mises en pratique en pensant aussi aux risques et menaces qui pèseront sur le logiciel une fois déployé.

3.6 Conseils pratiques

Cette fiche sensibilise l'étudiant avec les possibilités de vérifications de programme offertes par les compilateurs. Il est fortement conseillé d'utiliser ces options de compilation dans les projets de programmation, tout au long du cursus, afin de se familiariser avec eux le plus tôt et le plus longtemps possible.

3.7 Description

La phase de spécification d'un logiciel doit s'appuyer sur une étude de la **surface d'attaque** du système à construire. Quels seront les points d'interactions avec l'environnement potentiellement hostile ? Il convient de penser aux entrées utilisateurs sous forme de chaînes de caractères, mais aussi aux différents arguments d'un service s'il peut être appelé par une tierce-partie. Une attaque peut aussi provenir d'un canal auxiliaire (le temps d'exécution, la gestion de la mémoire...). Si toutes les possibilités sont difficiles à prévoir, il convient en tout cas d'en imaginer un certain nombre le plus tôt possible et sans se restreindre à des points d'interactions trop évidents.

Pour réduire cette surface, le principe de **moindre privilège** prévaut. En s'appuyant notamment, sur les mécanismes d'encapsulation de la fiche 2, le logiciel développé peut minimiser les services présentés à l'environnement extérieur. Cela réduit les risques d'attaques et simplifie le travail de défense, puisqu'il y a alors moins d'éléments critiques à défendre. Pour autant, le principe de **défense en profondeur** s'applique aussi au développement logiciel et sécuriser un composant interne d'un système reste très utile.

La pratique du développement sécurisé passe aussi par une bonne utilisation de l'**environnement de développement**. Les bonnes pratiques de génie logiciel restent d'actualité mais il faut par contre bien veiller à maîtriser les paramètres d'exécution de ces plateformes, afin d'éviter qu'un programme soit mis au point et testé dans un contexte trop différent de celui dans lequel il sera déployé. En Java, l'option `-enableassertions` (ou `-ea`) active la vérification des assertions, comme illustré par l'exemple suivant :

```
import java.util.Scanner;

  public class AssertTest
  {
     public static void main( String args[] )
     {
        Scanner input = new Scanner( System.in );

        System.out.print( "Entrer␣un␣nombre␣entre␣positif~:␣" );
        int number = input.nextInt();

        // vérifie que l'entrée est positive
        assert ( number >= 0 )~: "entrée␣invalide~:␣" + number;
        System.out.printf( "Vous␣avez␣entré␣%d\n", number );
     }
  }
```

Sans cette option, ces assertions sont justes ignorées. Aucun test ne sera effectué sur ces instructions. Notez que l'exemple précédent change donc de comportement avec ou sans l'option `-ea`. L'usage des assertions doit être réservé à la vérification de propriétés internes du programme (des propriétés toujours vraies, si l'implémentation du programme est correcte, indépendamment des hypothèses sur l'environnement extérieur). Ici, la lancée d'une exception est plus adaptée, car elle permet de traiter le cas d'erreur dans tous les cas.

Dans le cas de la mise au point de programmes concurrents, l'usage des assertions est encore plus subtil car il peut ajouter, en mode *debug*, des synchronisations qui cachent certains comportements qui seront pourtant visibles en mode *production* (en présences de *race conditions*).

Sous Eclipse, le mode est activé en sélectionnant un projet, puis « *Run as* », puis « *Run...* ». La boite de dialogue qui s'ouvre alors, comporte un onglet « *Arguments* » où il est possible d'ajouter `-ea` dans le boîte « *VM arguments* ».

Les **compilateurs** fournissent parfois des options permettant d'augmenter le nombre de vérifications effectuées à la compilation, ou de générer des programmes avec des défenses dynamiques contre certaines attaques. Ces mécanismes sont fortement conseillées mais nécessitent une bonne compréhension des options disponibles dans le compilateur [1, 2, 3].

Il convient de se méfier des groupements d'options de type `-Wall` qui contiennent rarement l'ensemble des options possibles, pour des raisons de compatibilité arrière. Parmi les options intéressantes à activer dans un compilateur C, nous pouvons citer :

- L'option `-Wconversion` de gcc permet détecter certaines conversions de types implicites dangereuses.
  Exemple : une instruction `unsigned int x = -1;` génère une conversion implicite à l'exécution du programme. Cette option permet de détecter cette situation à la compilation. Si la conversion est souhaitée par le programmeur, une conversion explicite devra être favorisée.
- L'option `-Wformat` de gcc permet d'éviter certaines vulnérabilités sur les chaînes de format des instructions `printf`.
  Exemple : un simple appel comme `fprintf(s);` peut provoquer une manipulation illicite de la pile d'appel si la variable `s` contient le symbole `%`. Cette option permet de détecter à la compilation les erreurs de ce type.
- L'option `-Wwrite-strings` de gcc permet de détecter une écriture dans une chaîne de caractères placée dans une mémoire en lecture seule. Cette option permettra par exemple de rejeter le programme suivant :

```
int main (void) {
  char* s = "Hello";
  printf ("%s\n",s);
  s[0] = 'h';
  printf ("%s\n",s);
  return 0;
}
```

- L'option `-fstack-protector` de gcc protége la pile contre l'écrasement de l'adresse de retour d'une fonction par débordement de tampon.
- L'option `-Werror` assure que le programme ne sera pas compilé tant que des messages d'avertissement subsistent. C'est une façon simple et radicale d'assurer que tous les avertissements seront sérieusement étudiés par le programmeur. Son utilisation peut avantageusement être proposée (voire imposée) aux étudiants pour leurs projets.

Il convient enfin de rappeler que ce type d'option de compilation existe dans d'autres langages que le C. En Java (à partir de la version 7), on peut ainsi utiliser l'option `-Werror`, comme en C [4]. En OCaml, c'est l'option `-warn-error +a` (en complément de `-w +a`) qui joue un rôle similaire. Enfin, en Python, il faut utiliser l'option `-W error` de l'interpréteur.

3.8 Matériels didactiques et références bibliographiques

[1] *Options to Request or Suppress Warnings*, `https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html#Warning-Options`

[2] *Wiki Debian - Hardening*, `https://wiki.debian.org/Hardening`

[3] *C-Based Toolchain Hardening*, `https://www.owasp.org/index.php/C-Based_Toolchain_Hardening`

[4] *javac - Java programming language compiler*, `http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javac.html`

[5] Éric Jaeger, Olivier Levillain, *Mind your languages(s) – A discussion about languages and security*, 2014 IEEE Security and Privacy Workshops

4 Fiche 4 : Outils d'analyse de vulnérabilités

4.1 Thématique

Thématique	Outil d'analyse de vulnérabilités	**Numéro de fiche**	4	**Mise à jour**	05/01/2016

Ce cours aborde les sujets suivants :
- la recherche de vulnérabilités logicielles par *fuzzing* ;
- l'analyse de teintes ;
- l'analyse des fuites mémoire ;
- les différents types d'alarmes d'un outil statique.

4.2 Thèmes des cours visés

- Compilation
- Génie logiciel

4.3 Volume horaire

30 à 45 minutes.

4.4 Prérequis / corequis

Cette activité s'intègre idéalement dans un cours de validation logicielle par test (*fuzzing*) ou en complément d'un cours de compilation (analyse statique).

Un cours de méthodes formelles pourra compléter les techniques de validation abordées dans cette fiche. Des notions de calculabilité permettront d'évoquer les limites théoriques pour l'analyse automatique des programmes.

4.5 Objectifs pédagogiques

L'objectif pédagogique de cette activité est de présenter les grandes familles d'outils permettant de détecter certaines erreurs de programmation.

4.6 Conseils pratiques

Les éléments de cette fiche seront idéalement présentées à travers des séances de travaux pratiques. Il est possible de s'initier au fuzzing avec le logiciel American fuzzy lop développé par Michal Zalewski [11]. Pour l'analyse statique, nous recommandons Frama-C [9] (plugin *valueanalysis*) pour l'analyse de programmes C et Findbugs [8] pour l'analyse de programmes Java.

4.7 Description

Une première technique de recherche de vulnérabilité consiste à tester le logiciel. Le **fuzzing** est une technique très populaire de recherche de vulnérabilités dans un logiciel. Le principe est de générer au hasard des données plus ou moins cohérentes pour le système testé, dans le but de mettre en défaut le programme (par exemple, de le faire échouer). Dans ces versions les plus simples, cette technique procède par force brute.

D'autres approches, plus sophistiquées, restreignent les données générées pour qu'elles respectent un certain format. L'intérêt de cette technique est de pouvoir s'appliquer à tout type de logiciel, quel que soit son environnement, que le code source soit disponible ou pas. Par exemple, si un programme prend en entrée un entier, et si cet entier est enregistré sur un octet, une technique de **fuzzing** permettra de tester le comportement du système si on lui soumet une entrée en dehors des valeurs représentables par cet encodage. L'approche reste limitée : il s'agit de découvrir des dysfonctionnements, et non pas de vérifier le bon fonctionnement d'un logiciel. Cette technique aide néanmoins à identifier les parties du logiciel les plus vulnérables.

Une deuxième technique consiste à instrumenter le logiciel pour lui faire réaliser des vérifications pendant son exécution et l'empêcher d'atteindre une faille de sécurité. Les mécanismes de **teinte** (*tainting*) permettent ainsi de suivre la propagation des valeurs en fonction de leurs origines. Il est alors possible d'empêcher un appel de fonction critique (comme une requête SQL) de s'appliquer à un argument qui pourrait dépendre de l'environnement extérieur, sans avoir été traitée au préalable par une procédure de nettoyage (*sanitizing*). Un tel mécanisme est disponible en Ruby et en Perl [1, 2]. Le programme suivant illustre ce mécanisme en Perl :

```
# lecture du premier argument du script: arg1 est teintée
my $arg1 = shift;
# arg2 est elle aussi teintée
my $arg2 = substr($arg1,0,2);
```

La teinte, issue d'une valeur provenant de l'environnement extérieur (fichier, entrée standard), se propage dans les calculs qui utilisent une valeur teintée. Certaines fonctions sensibles, comme les appels de fonctions système, refusent de s'exécuter sur des valeurs teintées. Pour enlever une teinte, on peut la filtrer avec une expression régulière et récupérer une sous-expression non-teintée pour la placer dans la variable d'origine :

```
if ($arg2 =~ /^(\w{4})$/) {
    # expression régulière des mots comportant
    # exactement 4 caractères alphanumérique (ou _)

    $arg2 = $1;
    # arg2 n'est plus teintée
} else {
    ... # le nettoyage de la teinte échoue
}
```

C'est au programmeur de s'assurer que l'algorithme de nettoyage est correct. L'activation du suivi des teintes est optionnelle en Perl (option `-T`).

Il est aussi indispensable dans certains langages de programmation (comme le C) de bien vérifier les allocations et libérations de la mémoire utilisée. Cela permet d'éviter d'éventuelles fuites mémoire produisant une saturation partielle ou totale d'un système. Voici un exemple (en C) d'une fonction

ne libérant pas la mémoire allouée (à la variable « tmp ») :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

size_t getLength(int num1, int num2) {
        char *tmp = (char *)calloc(sizeof(char), (sizeof(int) *
           2) + 1);

        if (tmp != NULL) {
                sprintf(tmp, "%d%d", num1, num2);
                return strlen(tmp);
        }
        return 0;
}

int main() {
        printf("%ld\n", getLength(42, 10000));
        return 0;
}
```

Chaque appel à la fonction getLength alloue un espace pour la variable tmp, qui n'est jamais libéré. Il y a donc une fuite mémoire, proportionnelle au nombre d'appels de la fonction, dont la conséquence peut être la terminaison inopinée du programme.

Afin de tester les fuites de mémoire d'un programme en C on peut utiliser le logiciel Valgrind très simplement avec la commande valgrind <le_programme_a_tester>, qui produit un rapport sur les fuites mémoire d'un code source.

Si on souhaite avoir plus de détails on peut aussi rajouter l'option --leak-check=full, ce qui donne le résultat suivant avec le programme décrit ci-dessus :

```
==7750== Memcheck, a memory error detector
[...]
==7750== HEAP SUMMARY:
==7750==     in use at exit: 9 bytes in 1 blocks
==7750==   total heap usage: 1 allocs, 0 frees, 9 bytes
   allocated
==7750==
==7750== 9 bytes in 1 blocks are definitely lost in loss record
   1 of 1
==7750==    at 0x4C2CC70: calloc (in /usr/lib/valgrind/
   vgpreload_memcheck-amd64-linux.so)
==7750==    by 0x400629: getLength (in /home/user/TEST)
==7750==    by 0x400678: main (in /home/user/TEST)
[...]
==7750== For counts of detected and suppressed errors, rerun
   with: -v
==7750== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0
   from 0)
```

Il est souhaitable de détecter les vulnérabilités avant le déploiement et l'exécution d'un logiciel. L'**analyse statique** a pour but de diagnostiquer d'éventuelles erreurs d'exécutions, sans exécuter le logiciel sous étude. Elle s'appuie pour cela sur des techniques d'analyse de flots de données ou d'inférence de types pour calculer automatiquement des informations sémantiques [3, 4].

Par nature (pour des raisons d'indécidabilité), une analyse statique réalise des approximations. Il n'est pas possible, dans tous les cas, de calculer le diagnostic le plus précis possible sur un programme. Pour cette raison fondamentale, une analyse pourra lever une alerte sur un programme, alors que le chemin n'est pas faisable : c'est une *fausse alarme*. Par exemple, l'expression :

```
1 / ((x*x - x + 1) mod 2)
```

ne réalise pas de division par zéro, quelle que soit la valeur de la variable x, mais la plupart des analyseurs ne seront pas capables de le comprendre et lèveront une fausse alarme.

L'utilisateur doit inspecter l'alarme pour comprendre si elle est fausse ou pas. Les garanties sont surtout apportées sur les portions de programme sans alarme : l'analyse garantie qu'elle ne contient pas d'erreurs. Il faut cependant modérer ce discours quelque peu théorique : en pratique, très peu d'analyses statiques déployées dans des outils de production sont totalement correctes. Il arrive alors parfois que l'une de ces analyses ne trouve pas d'exécution dangereuse alors qu'il en existe : c'est un *faux négatif*. Il s'agit souvent d'un choix de conception : plutôt que de submerger le programmeur d'alertes, certaines analyses ne signalent qu'une partie des erreurs possibles [5]. Le développement d'analyseurs statiques corrects sur des langages réalistes (et subtils) comme C ou Java reste un sujet de recherche actif [6, 7] .

Malgré ces imprécisions, les analyseurs modernes sont capables de rapporter des erreurs de façon extrêmement efficace. On pourra par exemple utiliser l'analyseur libre FindBugs [8] pour expérimenter la recherche d'erreurs (pas uniquement des vulnérabilités) dans les programmes Java. Le plugin *valueanalysis* de Frama-C [9] permet quant à lui d'effectuer des vérifications statiques sur les programmes écrits en C. Ce dernier peut par exemple détecter des erreurs comme la lecture de portions mémoires non initialisées ou l'utilisation de pointeurs invalides (précédemment libérés) :

```
int *f(int c) {
        int r, t, *p;
        if (c) r=2;
        t = r + 3;       // alarme Frama-C~: lecture d'une
                         // variable non initialisée
        return &t;
}

int main(int c) {
  int *p;
  p = f(c);
  return *p; // alarme Frama-C~: déréférencement
             // d'un pointeur invalide
}
```

Le *fuzzing* et l'analyse statique sont des techniques complémentaires. L'outil Frama-C essaiera, par exemple, de *prouver* qu'un accès tableau `[i]` respecte les bornes mémoire de l'espace pointé par la variable `t`. Il calcule pour cela un intervalle de variation de la variable `i` puis vérifie qu'il est inclus dans l'intervalle `[0, length(t)-1]`. Le fuzzing ne cherche pas à prouver l'*absence* de faille, il cherche à prouver l'*existence* d'une faille. C'est une différence fondamentale. À l'inverse, sauf

dans quelques cas spécifiques, le *fuzzing* ne permet pas de prouver l'absence de faille et l'analyse statique ne permet pas de prouver l'existence d'une faille.

4.8 Matériels didactiques et références bibliographiques

[1] Dan Ragle, *Introduction to Perl's Taint Mode*, `http://www.webreference.com/programming/perl/taint/index.html`

[2] *The Pragmatic Programmer's Guide : Locking Ruby in the Safe*, `http://ruby-doc.com/docs/ProgrammingRuby/html/taint.html`

[3] Andrei Sabelfeld, Andrew C. Myers, *Language-based information-flow security*, IEEE Journal on Selected Areas in Communications, 2003

[4] Flemming Nielson, Hanne Riis Nielson, Chris Hankin, *Principles of Program Analysis*, Springer 2005

[5] David Hovemeyer, William Pugh, *Finding more null pointer bugs, but not too many*, PASTE 2007 : 9-14

[6] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, Xavier Rival, *A static analyzer for large safety-critical*, software. PLDI 2003 : 196-207

[7] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie, *A formally-verified C static analyzer. In 42nd symposium Principles of Programming Languages*, pages 247–259. ACM Press, 2015

[8] *Site officiel de FindBugs*, `http://findbugs.sourceforge.net`

[9] *Value analysis plug-in*, `http://frama-c.com/value.html`

[10] *Magasine MISC*, n°39, Fuzzing - Injectez des données et trouvez les failles cachées

[11] *Magasine MISC*, n°11, Hors Série - Outils de sécurité

Ce document pédagogique a été rédigé par un consortium regroupant des enseignants-chercheurs et des professionnels du secteur de la cybersécurité.

Version 1.3 — Février 2017