Secteur Tertiaire Informatique

Filière « Etude et développement »

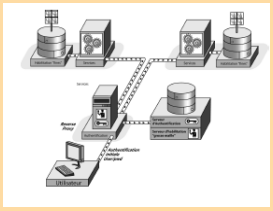
Séquence « Développer une interface utilisateur de type Desktop»

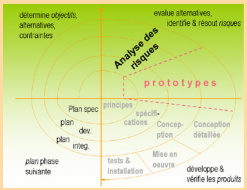
Identifier les spécificités de sécurité des langages et les attaques classiques

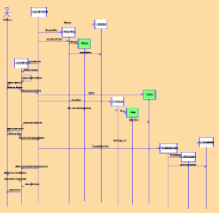
Apprentissage

Mise en situation

Evaluation







Version	Date	Auteur(s)	Action(s)
1.0	15/06/16	Lécu Régis	Création du document

TABLE DES MATIERES

Objectifs

A l'issue de cette séance, le stagiaire sera capable de :

• Connaître les atouts et les inconvénients des différentes catégories de langage, par rapport à la sécurité : mécanisme de protection de la mémoire, mécanisme d'exception, détection des débordements de tampon (*buffer overflow*), typage fort ou faible.

• Connaître les attaques classiques « bas niveau » (*buffer overflow*, écrasement de pile, *integer overflow*) et des exemples d'exploitation de ces attaques (injection de code).

• Associer les catégories de langage et les attaques possibles (le *buffer overflow* est possible en langage C mais pas en Java ou en C#) pour pouvoir sécuriser son programme avec efficacité.

• Choisir au mieux son langage et son compilateur pour le niveau de sécurité demandé.

• Activer les options de compilation pour détecter les opérations dangereuses ou « durcir » le code de manière générique.

Pré requis

Cette séance intervient après l'apprentissage d'un langage du type Java ou C# et du développement objet : les mécanismes de protection mémoire et d'exception en cas de débordement de tableau ont été utilisés, mais leur fonctionnement interne n'est pas nécessairement connu. La connaissance préalable du langage C n'est pas nécessaire. Cette séance ne prétend pas être un cours de langage C, mais nous donnerons des indications de lecture pour faciliter la compréhension des exemples.

Méthodologie

Ce document peut être utilisé en présentiel ou à distance (pour un stagiaire connaissant un IDE intégrant le langage C, tel que Visual Studio). Ce document précise la situation professionnelle visée par la séance, la resitue dans la formation, et guide le stagiaire dans son apprentissage et ses recherches complémentaires.

Mode d'emploi

Symboles utilisés :

 Renvoie à des supports de cours, des livres ou à la documentation en ligne constructeur.

 Propose des exercices ou des mises en situation pratiques.

 Point important qui mérite d'être souligné !

Ressources

- Projet CyberEdu, préconisations de l'ANSSI :

  CyberEdu_developpement_09_2015.pdf, Fiche 1 « Vulnérabilités standard » et fiche 2 « Sécurité intrinsèque des différents langages de programmation »
- Initiation au langage C pour des développeurs Java, document Afpa : DuJavaAuC.doc
- Bonnes pratiques de sécurité en Java :

  JavaSec-Recommandations.pdf, JavaSec-Execution.pdf, JavaSec-Langage.pdf
- Tutorial Oracle sur le développement sécurisé en Java :

  http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/SecureJavaCodingGuidelines/player.html

1. INTRODUCTION

Les langages ne sont pas égaux par rapport à la sécurité du développement, et aucun d'entre eux n'est parfait. Pour faire le bon choix selon le type d'application à réaliser, le développeur doit donc connaître les spécificités des langages, et le niveau de sécurité qu'ils fournissent. Il y a toujours un prix à payer, et chaque avantage est en général accompagné d'un inconvénient.

Le langage de haut niveau que nous avons pratiqué dans les séances précédentes (Java ou C#) est bien adapté à des applications complexes de gestion, de calcul scientifique, d'application mobile. Nous allons voir qu'il intègre de nombreux mécanismes de sécurité, qui facilitent la sécurisation d'une application. Mais les contrôles effectués en standard augmentent la taille du code généré et ralentissent l'exécution du programme. Ces deux inconvénients limitent l'utilisation de ces langages pour les applications proches du matériel, qui doivent être rapides et prendre peu de place : driver, application embarquée, robotique etc.

A l'autre extrémité de la gamme des langages, on trouve l'assembleur et le langage C. Ces langages sont rapides, compacts et donc bien appropriés pour le matériel ; mais ils effectuent peu de contrôles internes, et sont donc à l'origine de nombreuses vulnérabilités logicielles, qui rendent possibles des attaques.

Dans cette séance, que vous pouvez suivre de façon autonome, nous allons d'abord observer ces différences de comportement entre les langages Java et C sur un cas d'école : le *buffer overflow*. Cette comparaison nous permettra de saisir tout l'intérêt des mécanismes d'exception, de protection des tableaux et des types en Java qui vous sont maintenant familiers. Ces mécanismes qui paraissent naturels au développeur objet, demandent beaucoup de travail à la compilation et à l'exécution, et ils n'existent pas dans les langages proches du matériel, comme le C. Pour que cette séance soit accessible à tous, nous commenterons les extraits en langage C, et en particulier l'utilisation des pointeurs.

Un bug est un plantage involontaire du programme, sans qu'il y ait forcément une intention malveillante de l'utilisateur. Mais un bug introduit une « vulnérabilité » qui pourra ensuite être « exploitée » par un utilisateur malveillant, pour mener une attaque. Dans la deuxième partie de la séance, nous allons donc appliquer le principe de sécurité « connaître les attaques pour se défendre efficacement », en montrant comment exploiter les vulnérabilités du langage C pour mener des attaques. Les précautions à prendre pour prévenir ces attaques seront présentées ultérieurement, dans la séance « Coder de façon défensive en suivant les bonnes pratiques de sécurité ».

Le langage Java contribue largement à la sécurité, mais il n'est pas parfait pour autant ! Dans la troisième partie de la séance, nous commenterons donc certains aspects du langage Java par rapport à la sécurité.

2. JAVA ET C FACE AU DEBORDEMENT DE TAMPON

Nous allons commencer par observer les réactions des langages Java et C face à une tentative de *buffer overflow*. Dans ce premier exemple, il s'agit d'un simple bug (erreur de codage sans intention malveillante).

2.1 LE DEBORDEMENT DE TAMPON (BUFFER OVERFLOW) EN LANGAGE C

Fichier complet fourni dans le dossier « sources » : **bufferoverflow.c**

Commentaires de l'extrait de code qui suit :

(1) Les tableaux PRENOM et TABBUG sont des variables locales déclarées dans la fonction main. Comme en Java, elles sont donc placées dans la Pile.

La pile « remonte » des adresses mémoire hautes vers les adresses basses : le tableau TABBUG est donc placé en mémoire avant le tableau PRENOM.

A noter en C : les tableaux C sont statiques et doivent être déclarés avec une taille fixe.

(2) Le programme affiche les adresses des deux variables, leur taille et le nombre d'octets qui les sépare en mémoire.

A noter en C :

- le *printf* (équivalent C du *System.out.println*) utilise des formats d'affichage : *%d* pour un entier, *%s* pour une *String*, *%p* pour un pointeur ou l'adresse d'une variable.

Syntaxe : **printf ("format d'affichage", variable1, variable2 etc.)**

📖 http://fr.wikipedia.org/wiki/Printf

- **sizeof** (variable) ou **sizeof** (untype) donne le nombre d'octets de la variable ou du type

(3) En dépassant les limites de TABBUG, on écrase donc le début du tableau PRENOM qui vaut à la fin "#regis". On constate qu'aucune exception n'est levée par le C.

(4) Même principe avec des variables statiques TABBUG2 et NOM. Ces variables existent pendant toute la durée du programme, contrairement aux variables locales dans la pile qui sont détruites en fin de bloc. Normalement, sans optimisation particulière du compilateur, les variables statiques sont réservées en mémoire dans l'ordre où on les déclare. On place donc cette fois TABBUG2 avant NOM pour réaliser le débordement de tableau.

(5) En C, les « chaines », *String*, ne sont pas des objets comme en Java mais de simples tableaux de caractères, terminés par le nombre zéro (noté 0 et surtout pas '0' qui désigne le caractère Zéro comme en Java).

Un cas particulièrement grave de débordement : si l'on arrive à écraser le 0 terminateur. Toutes les fonctions qui utilisent des chaines (donc **printf** avec **%s**) parcourent la mémoire jusqu'à ce qu'elles trouvent un 0 terminateur, au risque de provoquer de nombreuses violations mémoire.

(6) En règle générale, de nombreuses fonctions C de manipulation des String (bibliothèque *string.h*) ne sont pas sécurisées et constituent un véritable « musée des horreurs » à éviter (il existe une version sécurisée de cette bibliothèque).

Dans l'exemple, **strcpy** recopie les caractères du tableau droit sur le tableau gauche, sans vérifier la taille du tableau cible.

scanf est la fonction de lecture clavier standard du C (équivalent de **System.in.read** en Java). Utilisée avec le format de saisie *"%s"*, elle n'offre aucune sécurité : l'utilisateur malveillant peut entrer autant de caractères qu'il le souhaite, et écraser le tampon de lecture, en débordant sur les variables qui le suivent, dans la zone statique ou dans la pile.

(7) Pour être complet, tentons maintenant un *buffer overflow* sur des variables allouées dynamiquement dans le « tas » (*heap*). Le C utilise la fonction **malloc** (*memory allocation*) qui est l'équivalent rustique du **new** de Java.

Les pointeurs en C (équivalents des références d'objet ou de tableau en Java) permettent de stocker les adresses de variables statiques, automatiques (dans la pile) ou dynamiques (dans le tas) et de les manipuler.

📖 Sur les pointeurs en C, vous pouvez lire le document afpa : DuJavaAuC.doc, pp 11-19, 28-47

Contrairement à Java, le C est un langage **faiblement typé** qui ne fait aucun lien entre la taille de la mémoire demandée et le type du pointeur qui reçoit l'adresse. **malloc**

renvoie un pointeur non typé (**void ***) dont le *cast* est à la charge du développeur. Toutes les erreurs sont donc possibles ! Pour être le plus près possible de Java, l'écriture conseillée pour allouer n éléments de type **UnType** serait :

```
UnType * p = (UnType *) malloc(n * sizeof(UnType) );
```

Dans l'exemple :

```
char * p1 = (char *) malloc(8 * sizeof(char) );
```

(8)

Pour réaliser le *buffer overflow*, le programme calcule la distance en octets entre les deux blocs alloués dans le tas, et l'utilise comme index (**offset**) pour écraser le début du deuxième bloc pointé par p2 :

```
p1 [offset] = '#';
```

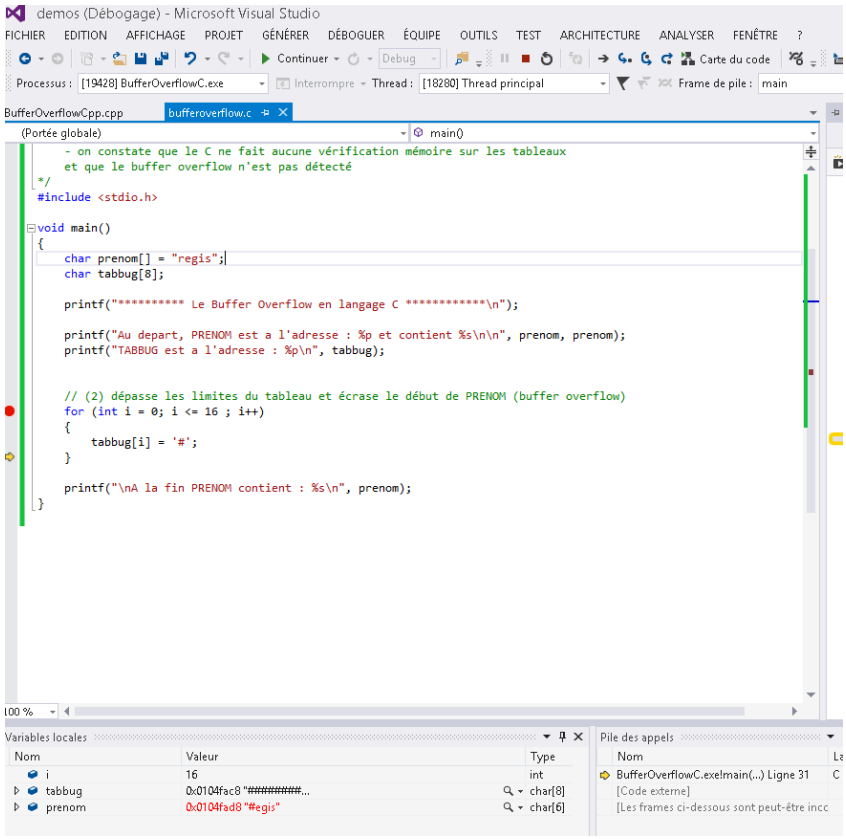
Extrait du programme **bufferoverflow.c**

(4)

```
static char tabbug2[4];
static char nom[8];

void main()
{
(1)     char prenom[] = "regis";
        char tabbug[8];

(2)     printf("TABBUG est a l'adresse:%p, de taille:%d\n", tabbug, sizeof(tabbug));
        printf("PRENOM est a l'adresse:%p, de taille:%d et contient au debut %s\n\n",
                prenom, sizeof(prenom), prenom);
        printf("Les deux variables sont distantes de %d octets en memoire", prenom-tabbug);

(3)     int i;
        for (i = 0; i <= 16; i++)
        {
               tabbug[i] = '#';
        }
        printf("\n\nApres le buffer overflow, PRENOM contient : %s\n", prenom);

(4)     printf("TABBUG2 est a l'adresse:%p, de taille:%d\n", tabbug2, sizeof(tabbug2));
        printf("NOM est a l'adresse:%p, de taille:%d\n", nom, sizeof(nom));
        printf("Les deux variables sont distantes de %d octets en memoire", nom - tabbug2);

(5)     for (i = 0; i < 7 ; i++)
        {
           nom[i] = 'a';
        }
        nom[i] = 0;
        printf("\nAu debut, NOM vaut: %s\n", nom);
        tabbug2[4] = '#';
        printf("\n\nApres le buffer overflow, NOM contient : %s\n", nom);
(6)
        strcpy(nom, "aaaaaaa");
        printf("%s", "Rentrer un texte (merci de saisir trois caracteres au plus) :");
        scanf("%s", tabbug2);
        printf("Apres le buffer overflow, NOM vaut:%s\n", nom);

        strcpy(nom, "aaaaaaa");  // réinialisation du tableau NOM (pour le test suivant)
```

```
        printf("%s", "Rentrer un texte (trois caracteres au plus, pas le choix!) :");
        scanf("%3s", tabbug2);
        printf("Apres le buffer overflow, NOM vaudra toujours :%s\n", nom);
(7)
        char * p1 = (char *)malloc(8 * sizeof(char) );
        char * p2 = (char *)malloc(8 * sizeof (char) );
        printf("Le pointeur p1 vaut %x, p2 vaut %x\n", (int) p1, (int) p2);
        printf("Les deux blocs memoire sont distants de %d\n", (int)p2-(int)p1);
        strcpy(p2, "regis");
        printf("Au depart, p2 pointe sur:%s\n", (char *)p2);
(8)
        int offset = ((int)p2 - (int)p1);
        p1 [offset] = '#';
        printf("\nApres le buffer overflow, p2 pointe sur:%s\n", p2);
    }
```

Faites tourner ce programme C dans votre IDE habituel (Eclipse, NetBeans ou Microsoft Visual Studio). Nous l'avons testé sous Visual Studio.



Observez le comportement du programme, en mode « Debug » et « Release ».

En mode « *Debug* », le debugger détecte le débordement du tableau TABBUG, et arrête le programme avec le message *« Run-Time Check Failure #2 - Stack around the variable 'tabbug' was corrupted ».*

En mode « *Release* », le débordement n'est pas détecté, et on écrase le début du tableau PRENOM.

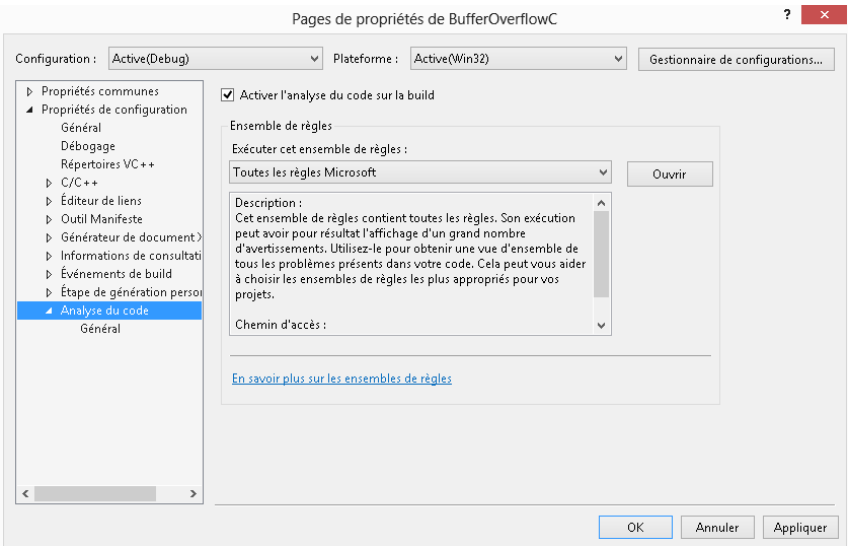
Conclusion : il faut donc systématiquement faire tourner les programmes en mode « *Debug* » pour détecter les débordements, avant de les générer en mode « *Release* » pour la livraison chez le client.

Sous debugger, visualisez les variables locales qui sont affichées selon leur ordre réel en mémoire.

Faites tourner le programme en pas à pas, pour observer l'écrasement du tableau PRENOM (les valeurs modifiées apparaissent en rouge).

Procédez de même pour observer les débordements dans la zone statique et dans le tas.

Dans les propriétés du projet, activez l'analyse de code, pour rechercher les violations mémoires :



En regénérant le projet, l'analyse statique de code détecte l'erreur suivante, qui est pertinente :

```
C6201 L'index dépasse la taille maximale autorisée par la mémoire tampon allouée par la pile.
L'index '16' est en dehors de la plage d'index valide '0' à '7' pour la mémoire tampon 'tabbug' allouée sans doute par la pile.
```

Cliquez sur chaque erreur pour visualiser la ligne correspondante dans le code.

Cliquez sur le nom de la règle de sécurité (par exemple *C6201*) pour obtenir l'aide en ligne Microsoft sur cette règle et des propositions de correction.

Conclusion

- Pour sécuriser nos développements dans des langages de bas niveau comme le C, nous utiliserons fréquemment ces deux moyens complémentaires :
  - L'exécution du programme en debug
  - L'analyse statique du code source
- La plupart des débordements de tampon qui viennent d'être vus, résultent d'erreurs de programmation qu'il suffit de corriger.
- D'autres sont plus délicats puisqu'ils résultent de vulnérabilités internes au langage C : mauvaise gestion des chaînes de caractères dans des bibliothèques non sécurisées (en particulier les entrées-sorties dans **stdio.h**, et la manipulation des chaines dans **string.h**).
- Certaines fonctions sont à proscrire : par exemple, la fonction **gets** qui remplit une zone mémoire, sans possibilité de préciser une taille max : `char *gets(char *s);`

- Les problèmes résultent souvent du manque d'intérêt des développeurs pour le développement sécurisé, car les mêmes bibliothèques fournissent en général des fonctions sécurisées, qui permettent d'éviter le débordement de tampon :

```
char *fgets(char *str, int n, FILE *stream)
(le paramètre n désigne la taille max du tampon, pointé par str)
```

2.2 L'EXCEPTION « ARRAY OUT OF BOUNDS » EN JAVA

Il est possible de sécuriser le code en C, en utilisant des versions sécurisées des bibliothèques *string.h* et *stdio.h*, ou en se limitant à certaines fonctions sures dans les bibliothèques existantes. Mais cela demande une attention constante du développeur, avec un risque important d'introduction de vulnérabilités dans des gros projets.

Ce type de vulnérabilité a donc été assez vite pris en compte, dans des langages comme Java qui lève une exception pendant l'exécution, pour stopper la tentative de *buffer overflow*.

Ce comportement vous est déjà familier (séances sur le développement objet), mais le passage par le langage C permettra de mieux comprendre l'importance de ces mécanismes pour le développement sécurisé.

Commentaires de l'extrait de code qui suit :

(1) Voici une version Java du programme précédent (*demodebordement.java*), qui tente d'écrire au-delà de la taille du tableau TABBUG, en confondant la taille du tableau et son dernier indice (taille-1). Pour éviter cette erreur classique, il faut éviter les constantes numériques dans les tests et se baser sur la propriété **length** du tableau (*tabbug.length*).

(2) Le code généré automatiquement par le compilateur détecte la tentative de débordement et lève une exception **ArrayIndexOutOfBoundsException** avant l'écrasement, qui est interceptée par le *catch*.

```
public class DemoDebordement
{
public static void main(String[] args)
{
        String prenom = "regis";
        char tabbug[] = new char[8];
        try
        {
                System.out.println("Au départ, PRENOM contient " + prenom);
    (1)         for (int i = 0; i <= 8; i++)
                {
                    tabbug[i] = '#';
                }
            }
    (2)     catch (Exception e)
            {
                System.out.println("Exception :"  + e);
            }
            System.out.println("Après l'exception, PRENOM contient toujours : " + prenom);
        }
    }
```

3. LES ATTAQUES CLASSIQUES

Comment passer du simple bug à une attaque qui « exploite » ce bug, pour en tirer profit : se loguer indument sur un système, récupérer des données cachées, prendre le contrôle du système ?

3.1 L'ATTAQUE PAR « BUFFER OVERFLOW » EN LANGAGE C

L'attaque par *buffer overflow* suffit à remettre en cause les quatre critères de sécurité DICP : disponibilité, intégrité, confidentialité, preuve.

Dans le programme qui suit, nous allons exploiter la vulnérabilité vue précédemment de la fonction **scanf** pour se loguer de façon inconditionnelle, sans avoir besoin du mot de passe.

Commentaires de l'extrait de code :

(1) Les variables TAMPON et OK sont des variables statiques, qui se suivent en mémoire. Un débordement sur le tableau TAMPON va donc écraser la variable OK, qui est utilisé comme booléen (le langage C ne possède pas de type **boolean** comme Java ou C++, donc on utilise des entiers, avec 0 pour FAUX et différent de 0 pour VRAI)

(2) La fonction *verifie_mot_de_passeBad()* est vulnérable au *buffer overflow*, car elle utilise la fonction *scanf*, sans préciser les limites du tableau à lire (3). En entrant suffisamment de caractères, l'utilisateur peut donc écraser OK qui est ensuite testé par le *main*, pour déterminer si la connexion a réussi (4).

(5) La fonction *verifie_mot_de_passeGood()* prévient le risque de *buffer overflow* en utilisant la forme sécurisée du *scanf*, qui précise la taille max du tableau :

```
scanf("%9s", tampon);
```

Pour les curieux, voir la méthode pour sécuriser un tampon dans le cas général, en générant un format "%n-1s" pour un tampon de taille n, avec la fonction *sprintf* : DuJavaAuC, page 34.

Extrait du programme fourni, **Attaquebufferoverflow.c**

```
(1)
static char tampon[10];
static int ok;
(2)
void verifie_mot_de_passeBad()
{
   ok = 0;
   printf("Votre mot de passe:");
(3) scanf("%s", tampon);
   if (strcmp(tampon, "secret") == 0)
      ok = 1;
}
(5)
void verifie_mot_de_passeGood()
{
   ok = 0;
   printf("Votre mot de passe:");
   scanf("%9s", tampon);
   if (strcmp(tampon, "secret") == 0)
         ok = 1;
}
void main()
{
   verifie_mot_de_passeBad();
   (4)
   if (ok)
```

```
    {
        printf("=> Vous etes peut-etre connecte en administrateur\n");
    }
    else
    {
        printf("xxxx Utilisateur inconnu\n");
    }
    verifie_mot_de_passeGood();
    (4)
    if (ok)
    {
        printf("=> Cette fois, vous etes VRAIMENT connecte en administrateur \n");
    }
    else
    {
        printf("xxxx Utilisateur inconnu\n");
    }
}
```

Faites tourner ce programme C dans votre IDE habituel et observez son comportement en mode « *Debug* » et « *Release*».

Si vous utilisez l'IDE Visual Studio, lancez l'analyse de code, localisez le code à l'origine des avertissements, et leurs descriptions en ligne (*C6031* etc.)

3.2 L'ATTAQUE PAR DEPASSEMENT D'ENTIER (« INTEGER OVERFLOW ») EN C

Revenons au langage C pour montrer que le *buffer overflow* n'est pas un problème isolé, mais l'une des conséquences du « typage faible ». Le C ne contrôle pas plus la cohérence des types de données qu'il ne contrôle la taille des tableaux.

Soit le plus grand entier court non signé : `unsigned short int max = 65535 ;`

En ajoutant 1 à max, on obtient 0, sans erreur d'exécution. Ce sera donc au développeur de vérifier qu'une opération arithmétique est possible pour un entier donné et ne provoque pas d'*overflow*.

Comme le *buffer overflow*, *l'integer overflow* est souvent un simple bug. Mais il peut être exploité dans des attaques, où il peut être combiné avec le *buffer overflow*.

Commentaires de l'extrait de code qui suit :

(1) Démonstration simple de *l'integer overflow*. 65535 + 1 = 0.

Il y a en plus une erreur de format : il faudrait utiliser **"%hd"** pour un entier court, pour éviter un *buffer overflow* dans l'appel à *scanf*. Le *scanf* avec **"%d"** écrit son résultat sur 4 octets alors que l'entier court ne fait que 2 octets, ce qui écrase les deux octets suivant. (Ce problème n'apparait pas dans l'exemple, car le code généré par le compilateur Microsoft réserve en réalité 4 octets pour un entier court)

(2) Ebauche d'attaque par exploitation d'*integer overflow*.

Le programme gère des données (représentées par le tableau TAB de taille MAX), avec une partie accessible à tous les utilisateurs (à partir de l'indice NBPRIV) et une partie réservée aux utilisateurs privilégiés (de 0 à NBPRIV-1). Pour la démonstration, les cases accessibles par tous les utilisateurs contiennent un 'U', les cases réservées au mode privilégié contiennent un 'P'.

(3) L'extrait de code traite la partie Utilisateur. Il saisit un indice compris entre 0 et (MAX - NBPRIV-1) qui sert d'offset pour indexer la deuxième partie du tableau. L'indice saisi est testé correctement.

(4) Mais le calcul de l'indice final provoque un *Integer Overflow* lorsque nombre > (65535 - NBPRIV). Dans ce cas, l'indice résultant après *overflow* (entre 0 et 65535) permet d'accéder à la partie basse du tableau, normalement réservée aux utilisateurs privilégiés (on affiche alors la lettre P)

```
int main()
{
 (1)   printf("%s", "Nombre ? (entre 0 et 65535)");
       unsigned short int nombre = 0;
       scanf("%d", &nombre);
       printf("Votre nombre est en realite : %d\n", nombre);

 (2)
       #define MAX          70000
       #define NBPRIV       30000
       char tab[MAX];
       int i;
       for (i = 0; i < NBPRIV; i++)
              tab[i] = 'P';
       for (i = NBPRIV; i < MAX; i++)
              tab[i] = 'U';

(3)    scanf("%d", &nombre);
       if (nombre >= MAX-NBPRIV)
              printf("Indice errone:%d\n", nombre);
       else
       {
(4)           unsigned short int i = NBPRIV + nombre;
              printf("L'utilisateur de base consulte la donnee numero : %d\n", i);
              printf("La donnee demandee est: %c\n", tab[i]);
       }
   }
```

 Faites tourner ce programme C dans votre IDE et observez son comportement en mode « *Debug* » et « *Release*». Si vous utilisez l'IDE Visual Studio, lancez l'analyse de code, localisez le code à l'origine des avertissements, et leur description en ligne (C6031 etc.)

Situez l'erreur de conception dans le programme fourni, qui est à l'origine de l'overflow, et corrigez-la. La valeur de MAX est imposée par le client et doit être conservée.

3.3 L'ATTAQUE PAR DEPASSEMENT D'ENTIER EN JAVA

Nous l'avons dit dès le début, aucun langage n'est parfait du point de vue de la sécurité, et le développeur devra sécuriser son code, quelque soit le langage choisi.

A la compilation, Java oblige le développeur à plus de rigueur qu'en C, mais les contrôles syntaxiques sont parfois surprenants.

A l'exécution, Java détecte systématiquement les tentatives de *buffer overflow*, mais il n'existe pas d'exception *Integer Overflow*.

Il existe une `ArithmeticException` mais elle n'est pas levée automatiquement : comme en C ou C++, c'est au développeur de tester la faisabilité d'une opération avant de l'effectuer et de lever (*throw*) une `ArithmeticException` pour éviter un *overflow*.

(Nous verrons dans une autre séance sur la sécurité, comment tester la faisabilité d'un calcul).

Commentaires de l'extrait de code qui suit :

(1) Chaque type, défini par une classe, définit ses bornes Min et Max que l'on pourra tester dans les calculs. Notons que Java ne possède pas de type « *unsigned short* » comme le langage C, tous les nombres sont signés.

(2) Dans les bibliothèques de conversion, tout va bien : la méthode *Short.valueOf(s)* lève l'exception *NumberFormatException* si la *String* ne représente pas un *short*. Mais ce n'est pas une exception spécifique à *l'overflow* : n'importe quelle chaine incorrecte provoque la même exception. Essayez des nombres inférieurs et supérieurs à 32767 et observez l'exception.

(3) Le compilateur Java est plus « fortement typé » que le C, ce qui exige plus de cohérence dans la manipulation des types : l'addition d'un *short* et d'un *int* donne un *int*, qui ne peut donc être rangé dans un *short* (risque de perte d'information). Mais la syntaxe à la ligne suivante passe (car cette fois, il n'y a pas de variable intermédiaire de type *int*)

(4) Dans les calculs, aucun *overflow* n'est testé : c'est au développeur de tester les overflow, comme en C. En saisissant par exemple *32000*, on obtient l'overflow *32000 + 10000 = -23536*

(5) Même principe d'attaque que dans l'exemple précédent en C. Les 10000 premières cases du tableau sont en mode « privilégié », marquées par la lettre 'P'. Les cases suivantes sont accessibles à tous les utilisateurs, marquées par la lettre 'U'. Le programme boucle, puisque l'indice i, du fait de l'overflow, n'atteint jamais la taille du tableau *tab.length*. Si l'on décommente la ligne suivante qui affiche les cases du tableau, on sort sur l'exception *ArrayOutofBounds*.

```
public class DemoNumericException
{
 public static void main(String[] args)
 {
    (1)   System.out.println("Nombre entre "+Short.MIN_VALUE+"et "+Short.MAX_VALUE + ":");
        short nombre =0;
        try
        {
            String s = Lire.Chaine();
     (2)    nombre = Short.valueOf(s);
            System.out.println("Votre nombre vaut : " + nombre);
        }
        catch (NumberFormatException e)
        {
            System.out.println(e);
        }
        short result = nombre;
(3)     // result = result + 10000;
(4)     result += 10000;
        System.out.println(nombre + " + 10000 = " + result);

(5)
   char tab[] = new char[40000];
   int j ;
   for (j = 0; j < 10000 ; j++)
           tab[j] = 'P';
   for (j = 10000; j < tab.length; j++)
            tab[j] = 'U';
```

```
        short i = result;
        while (i < tab.length)
        {
           System.out.println("Indice de consultation: " + i );
           // System.out.println( "valeur case = " + tab[i] );
           i++;
         }
     }
}
```

📖 Pour approfondir : le site du CERT sur le code sécurisé en Java « CERT Oracle Coding Standard for Java » décrit le *buffer overflow* en Java et les méthodes pour le prévenir.
Ces méthodes seront présentées dans la séance « Coder de façon défensive en suivant les bonnes pratiques de sécurité »
https://www.securecoding.cert.org/confluence/display/java/NUM00-J.+Detect+or+prevent+integer+overflow

3.4 UN EXEMPLE D'EXPLOITATION DU « STACK OVERFLOW » EN C : L'ATTAQUE PAR INJECTION DE CODE

La mise en œuvre complète de cette attaque demande une bonne connaissance des mécanismes mémoire de la Pile pour un système donné, d'un compilateur C et de ses options, voire de l'assembleur, ce qui dépasse le cadre de cette séance (=> voir un cours de sécurité avancée ou de *hacking*).

Nous allons cependant présenter le principe de cette attaque, et du bug qu'elle exploite, pour montrer les risques auxquels conduit un *buffer overflow* sur la pile d'exécution (dit « *stack overflow »)*

3.4.1 Le principe de l'injection de code

Lorsqu'il se produit dans la Pile, le *buffer overflow* fait courir un risque plus grand que l'écrasement des données : il permet de détourner l'exécution normale du programme vers du code malveillant, que l'on « injecte » dans la pile.

Contrairement à un simple tableau de données, la pile d'exécution est utilisée par le processeur lors de l'appel d'une fonction, pour passer les paramètres, stocker les variables locales et l'adresse de retour de la fonction.

Normalement, cette adresse pointe sur l'instruction qui suit l'appel de la fonction, dans le *main* ou la fonction appelante. L'injection de code est une exploitation possible du *buffer overflow* qui va corrompre cette adresse, en lui substituant l'adresse du code malveillant.

L'exemple suivant, fourni par l'ANSSI n'est pas un code d'attaque. C'est un programme bogué (et particulièrement obscur) qui va nous permettre de comprendre le fonctionnement de la corruption de la pile d'exécution (*stack smashing*), avec détournement du programme vers du code existant non désiré, ou du code injecté par l'attaquant. Ce bug crée une vulnérabilité qui pourrait être exploitée pour construire une attaque.

Commentaires de l'extrait de code :

Ce programme provoque un *Stack Overflow* qui écrase l'adresse de retour de la fonction **set** en lui substituant l'adresse de la fonction **bad** : cette fonction affiche **"***Bad things happen***"** et termine le programme par un appel à la fonction **exit**, alors que la suite du main, immédiatement après l'appel de la fonction **set,** aurait du afficher *"Hello world"***.**

Détaillons les étapes du code :

(1) A l'appel de la fonction **set**, le **main** empile les paramètres dans l'ordre inverse : d'abord **v**, puis **s,** puis il empile l'adresse de retour (adresse du *printf ("hello word")* dans le main). Comme la pile remonte en mémoire des adresses hautes vers les adresses basses, **s** est donc placé dans la pile avant **v**, et après l'adresse de retour (4 octets après).

(2) Détaillons la seule instruction (obscure) de cette fonction :

- **&s** pointe sur le paramètre s dans la Pile, c'est donc l'adresse immédiatement après l'adresse de retour
- le paramètre **s** vaut 1 (dans le cas du bug) donc **&s-s** vaut **&s - 1**
- **&s-1** pointe donc sur l'adresse de retour (attention : -1 décrémente l'adresse de 4 octets car c'est un pointeur d'int)
- le paramètre **v** de type int contient en réalité l'adresse de la fonction **bad**

(grâce au **cast** dans l'appel de la fonction **set** dans le main : *set (1, (int) bad);*

- donc **(&s - s) = v;* écrase l'adresse de retour avec l'adresse de la fonction **bad.** Au retour de fonction, le branchement se fera vers la fonction **bad** au lieu de revenir au main.

```
void set (int s, int v)
{
(2) *(&s - s) = v;
}
void bad()
{
   printf("Bad things happen!\n");
   exit(0);
}
int main (void)
{
(1) set(1, (int) bad);
    printf("Hello world\n");
    return 0;
}
```

Compilez ce programme en mode *release* et en mode *debug*.
Exécutez le sans déboguer dans les deux modes de compilation et observez les différences.

Suivez le ensuite sous debugger.

3.4.2 Les étapes de l'injection de code, sur un exemple [1]

```
// Programme injection.c
#include <string.h>
void foo (char *bar)
{
   char c[12];
   strcpy(c, bar);  // pas de test sur les bornes du tableau
}
int main (int argc, char * argv[])
{
```

[1] Adapté de Wikipedia : https://en.wikipedia.org/wiki/Stack_buffer_overflow

```
    foo(argv[1]);
}
```

Un programme C peut recevoir des paramètres de la ligne de commande dans le tableau de pointeurs **argv** (**argument values**), à partir de l'indice 1. **argv[0]** contient toujours le nom de l'exécutable, **argc** est le nombre de paramètres passés (commande comprise)

Exemple de commande : **injection hello**

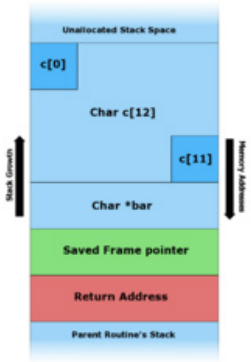
argv[0] = "injection", argv[1] = "hello", argc = 2

Commentaires de l'extrait de code :

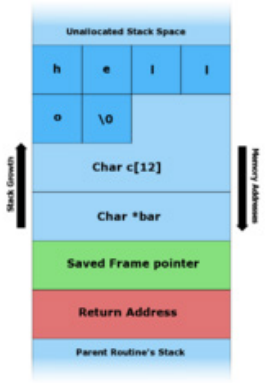
Le programme copie son paramètre (pointé par **argv[1]**) dans le tableau **c** de la pile, par l'appel à la fonction *strcpy* qui n'est pas sécurisée et ne teste pas les bornes du tableau cible.

Ceci ne pose pas problème, si le paramètre passé fait au plus 11 caractères (12 octets avec le terminateur de chaine '\0'), mais dans le cas contraire, la fonction *strcpy* va écraser la pile, par débordement du tableau c.

La pile avant la copie des données

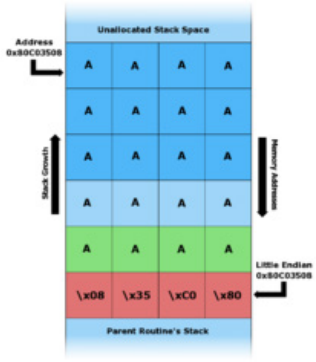


La pile après la copie des données ("hello")



Dans l'appel de fonction **foo**, le **main** empile d'abord l'adresse de retour (**Return Address**), puis le paramètre **bar** et les variables locales (tableau char **c [12]).**

Si la taille des données dépasse 11 caractères, il y écrasement du ***Saved Frame Pointer*** et surtout de l'adresse de retour.



Exemple avec le paramètre : "AAAAAAAAAAAAAAAAAAAA\x08\x35\xC0\x80"

Dans l'exemple 0x80C03508 est l'adresse du sommet de pile, où sont stockées les données injectées par l'attaquant : « AAA etc. ». En modifiant l'adresse de retour en Pile, l'attaquant va détourner le programme lors du retour de la fonction **foo**, vers un extrait de code malveillant (dans la vraie attaque, ce code sera injecté à la place de la chaine « AAA etc. » de notre exemple).

4. LES SPECIFICITES DE SECURITE DU LANGAGE JAVA

Nous avons commencé cette séance par l'observation et la pratique. Avant de présenter les spécificités de sécurité du langage Java, nous allons le resituer parmi les langages existants, en précisant la notion de typage et son intérêt pour le développement sécurisé.

4.1 DES LANGAGES INEGAUX FACE A LA FIABILITE ET LA SECURITE

Nous avons défini dans la première séance sur le développement sécurisé, la différence entre les notions de « sûreté de fonctionnement » et de « sécurité » d'une application. Mais elles se rejoignent souvent dans la pratique.

Le langage ADA, dérivé du langage Pascal, reste sans doute un des langages les plus adaptés aux développements fiables et sécurisés, ce qui justifie son choix pour des applications critiques, militaires ou industrielles. Pour le dire avec humour, c'est un des rares langages où il est parfois difficile de compiler, mais où l'application une fois compilée a de très fortes chances de fonctionner du premier coup, en conformité avec le cahier des charges !

📖 Pour les curieux, présentation d'ADA : https://fr.wikipedia.org/wiki/Ada_%28langage%29

Cette efficacité repose bien sûr sur la clarté et la non ambigüité de la syntaxe, et principalement sur la notion de « typage fort ».

4.1.1 Le typage fort, en Pascal/Ada

Au sens strict, un langage est dit « fortement typé » si :

- chaque objet appartient à un seul type
- le type d'une expression peut être déterminé syntaxiquement à la compilation (typage « statique »)
- il n'y a aucune conversion implicite.

En Pascal et Ada, les variables et les constantes ne sont pas de simples zones mémoire, comme en C. Elles appartiennent à un TYPE, c'est-à-dire à un ensemble qui partage une sémantique et des règles d'utilisation. Par exemple, on peut additionner deux entiers, ou afficher un caractère, mais on ne peut pas additionner un entier et un caractère comme en C, comparer un booléen à un entier ou une chaîne de caractères comme en JavaScript etc. Les nombres ont une arithmétique, mais pas les caractères. On ne peut comparer que deux éléments de même type, ou de types compatibles dans des cas très précis.

Cette logique ensembliste rend les programmes plus faciles à comprendre et plus prédictifs, voire démontrables : cela ne sert pas à grand chose de démontrer un algorithme par récurrence, si on le code ensuite dans un langage où ses données peuvent être écrasées de façon aléatoire par une autre partie du programme (*buffer overflow*).

Dans ces langages, le typage fort est un « typage statique » :

- les variables et les constantes sont typées dès leur déclaration
- la compatibilité entre les types et la cohérence des opérations sont vérifiées dès la phase de compilation.
- pour compléter les types prédéfinis du langage, on définit des « types utilisateur » : par exemple un tableau sera défini par son nom de type utilisateur, sa taille et son type de données.
- le compilateur interdit d'affecter un tableau d'un certain type à un tableau d'un autre type, ou d'affecter n'importe quoi à une case d'un tableau.
- pour ces langages, un tableau n'est pas seulement une zone mémoire, avec une adresse de début : il a une sémantique, que l'on peut reconnaître à la déclaration par le nom du type (*les_mesures*). Exemple, en langage Pascal :

```
type
  liste_entiers = array [1..10] of integer;
  les_mesures = array [1..20] of real;
var
  nombres: liste_entiers;
  mesures: les_mesures;
begin
   mesures := nombres;
```

Le pseudo-langage utilisé dans la séance *« Ecrire un algorithme »*, est une version francisée du langage Pascal. Cette séance vous a habitué à définir des constantes typées, des types utilisateur, et à attribuer ces types aux variables en respectant des règles sémantiques : l'âge du capitaine et la vitesse du vent sont des nombres, mais ne doivent pas se comparer ou s'additionner !

Pour les curieux, présentation du Pascal :

https://fr.wikipedia.org/wiki/Pascal_%28langage%29

Ces bonnes pratiques algorithmiques contribuent évidemment au bon fonctionnement du programme, à sa « sûreté » ou « fiabilité », mais également à sa « sécurité » : rappelons qu'il n'y a pas de « sécurité par l'obscurité ». Un programme complexe et mal écrit sera peut-être plus difficile à déchiffrer par l'attaquant, mais au final il contiendra presque toujours des vulnérabilités qui rendront possibles les attaques.

Les attaques vues précédemment reposent toutes sur une mauvaise conception initiale, et un mauvais typage des variables :

- *buffer overflow* en C : mauvaise définition de l'intervalle décrit par l'indice du tableau. Il faut vérifier cet intervalle, par des tests ou mieux par un type « intervalle » (en Pascal ou ADA)

- *integer overflow* en C et Java : mauvais choix de typage (entier court pour un tableau de taille > 65536)

En complément des vérifications à la compilation, un compilateur peut aussi protéger dynamiquement le typage en générant du code, qui teste automatiquement les bornes des tableaux à l'exécution et lève des exceptions, lorsque les règles d'indexation sont enfreintes.

4.1.2 Le typage dynamique en C++/Java

Si l'on suit la définition stricte, les langages orientés objet comme C++ ou Java ne sont pas « fortement typés », puisque le type effectif d'un objet ne peut souvent être résolu qu'à l'exécution du programme » : le compilateur ne peut souvent déterminer que la classe ancêtre et le choix de la classe dérivée qui va être instanciée n'est pas prédictible.

Cette « liaison dynamique » rend possible le polymorphisme, qui est un mécanisme essentiel de la programmation objet. Mais elle complique l'approche sécurité, en complexifiant le code et en permettant l'introduction de nouvelles vulnérabilités : sans précaution particulière, un attaquant peut par exemple créer une nouvelle classe dérivée malveillante, ou charger une classe en l'appelant depuis un autre package hostile. C'est un risque de sécurité directement lié à la programmation objet, que l'on peut résoudre en interdisant l'héritage sur certaines classes critiques (attribut **final** sur la classe) :

📖 Pour les curieux, lisez sur ce sujet : **sealing.pdf**

Les aspects sécurité liées à la programmation objet seront détaillés dans la suite de cette séance. Retenons pour le moment que la programmation objet, malgré ses atouts, peut aussi être à l'origine de failles de sécurité spécifiques.

Certains auteurs classent Java dans les langages « fortement typés ». Java est un langage correct du point du typage et de la sécurité, qui est intermédiaire entre Pascal/ADA et C/JavaScript/PHP :

- Java est plus rigoureux que le C dans la manipulation des types prédéfinis, mais il ne comporte pas d'instruction TYPE pour créer de nouveaux types utilisateur, et enrichir les types existants reconnus par le compilateur. Un tableau Java ne peut pas être affecté à un autre par l'opérateur = comme en Pascal ou en ADA.

  Vous avez rencontré ces difficultés de traduction du pseudo-langage à Java, dans la séance « Coder un algorithme ».

- Il conserve des conversions implicites

  https://docs.oracle.com/javase/specs/jls/se7/html/jls-5.html

- les types réels des instances de classes dérivées sont déterminés par « liaison dynamique » et ne sont pas connus à la compilation.

4.1.3 Le typage faible du langage C

A l'opposé de Pascal et Ada, le langage C est faiblement typé : les variables et les constantes sont typées à la déclaration, mais le compilateur C fait très peu de vérifications sur la cohérence des types.

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.

Avec les options par défaut, le compilateur C de Microsoft accepte par exemple :

```
int k = 'A';                    // affectation d'un caractère à un entier
```

```
float r = "regis";          // affectation d'une adresse dans un reel
char tableau[] = "toto";
if (tableau == 1) ;         // comparaison entre adresse et entier
                            // et instruction vide (permise)
```

Une erreur de type (transtypage par exemple) peut donner lieu à un dysfonctionnement, voire à une vulnérabilité silencieuse, comme le *buffer overflow*, sans aucune remontée d'alerte, ni à la compilation ni à l'exécution. Ce danger est bien illustré par l'exemple suivant, fourni par l'ANSSI.

Commentaires de l'extrait de code :

(1) La fonction C **safewrite** devrait écrire une valeur **val** dans le tableau **tab** de taille **size**, à l'indice **ind**, en vérifiant au préalable que cet indice appartient aux bornes du tableau. Elle semble donc suivre les règles du développement sécurisé, puisqu'elle contrôle ses paramètres en entrée, considérés comme non sûrs.

(2) Mais dans le deuxième appel, ce contrôle échoue à cause d'un problème de transtypage : le paramètre **ind** étant défini en *signed char*, la constante **128** est convertie en -1 (< MAX) et le *buffer overflow* a lieu.

(3) Attention à la définition d'interface maladroite, qui rend inefficaces les tests des paramètres en entrée.

```
(1)                                          (3)
void safewrite(int tab[], int size, signed char ind, int val)
{
   if (ind < size)
          tab[ind] = val;
   else
          printf("==> Hors limite !\n");
}
int main(void)
{
   #define MAX 120
   int tab[MAX];
   printf("ecriture a l'indice 127");
   safewrite(tab, MAX, 127, 1);
   printf("ecriture a l'indice 128\n");
(2)   safewrite(tab, MAX, 128, 1);
   return 0;
}
```

En conclusion, le but de cette mise au point n'est pas de vous dissuader de pratiquer le langage C, qui est très utile dans certains domaines, mais de vous préparer à mieux sécuriser vos programmes dans un environnement vulnérable.
A la défense du C :

- le laxisme du compilateur se justifie parfois, pour des applications embarquées et temps réel, où il faut aller vite et être le plus proche possible du matériel (adressage absolue, registres etc.)
- Il existe des options de compilation pour durcir le code et sortir au moins des *warning*.
- L'analyse statique du code identifie beaucoup de vulnérabilités.

4.1.4 Les parents pauvres de la sécurité : JavaScript, PHP

Une dernière catégorie de langages, très utilisés, ne précise pas les types dans la déclaration des variables, et il n'est pas obligatoire de déclarer une variable avant son utilisation. Les variables sont typées dynamiquement à l'initialisation.
Par exemple, en JavaScript [2]:

```
a = 1;
b = 2;
c = 'Foo';
```

a est un entier, b un entier et c une chaine.

Ce typage implicite des variables nuit à la lisibilité des programmes et facilite l'introduction de bugs et de vulnérabilités.

Autre inconvénient : la surcharge des opérateurs, par exemple l'opérateur +, et les règles d'associativité du langage, aboutissent à des résultats peu intuitifs et difficiles à prédire. Par exemple :

`alert(a + b + c);` affiche 3Foo

JavaScript l'interprète comme (a + b) + c. Le premier + est l'addition entière, le deuxième + la concaténation. Dans l'ordre, l'interpréteur additionne a et b (3), convertit le résultat en chaine ('3'), et concatène la chaine **'Foo'** au résultat intermédiaire ('3Foo').

`Alert (c + a + b);` affiche Foo12
JavaScript l'interprète comme (c + a) + b. Le deux + sont des opérateurs de concaténation. Dans l'ordre, l'interpréteur convertit a en chaine qu'il concatène avec c ('Foo1'), puis il convertit b en chaine, qu'il concatène au résultat intermédiaire (**'Foo12'**).

```
alert(c + (a + b));
```

Qu'est-ce qui va être affiché et pourquoi ?

Vérifiez en lançant la page de démonstration : **DemoJavaScript.html**

D'autres bizarreries de l'interpréteur JavaScript rendent les programmes peu prédictibles et conduisent à des vulnérabilités. Par exemple, un test rigoureux sur les opérateurs, effectué par l'ANSSI, montre que :

null <= *false* est vrai, alors que

null == *false* est faux

null < *false* est faux.

Le JavaScript est donc par nature difficile à sécuriser, comme la majorité des langages interprétés : nous reviendrons sur cette question dans la séquence « Développer des Pages Web en lien avec une base de données », en exposant les bonnes pratiques d'écriture en JavaScript.

PHP est également très difficile à sécuriser, et des vulnérabilités PHP sont parfois présentes dans les FrameWorks ou les CMS eux-mêmes, ce qui les étend automatiquement à de nombreux sites Web basés sur ces outils.

---

[2] Exemple fourni par l'ANSSI

📖 Pour approfondir (si vous n'avez pas peur de l'anglais), lisez l'article de l'ANSSI sur la sécurité comparée des langages de programmation :

Mind_Your_Languages.pdf

4.2 DEVELOPPEMENT OBJET ET SECURITE, EN JAVA

Java propose des mécanismes a priori intéressants en sécurité, par exemple pour cloisonner les informations et les traitements. Comme ils permettent d'améliorer la qualité du code, ces mécanismes contribuent globalement à sa sécurisation. Mais :

- ce sont souvent de simples outils d'ingénierie logicielle, qui se révèlent fragiles et ne suffisent pas à garantir la sécurité du code (exemple : l'encapsulation).
- ils peuvent être facilement détournés de leur objectif, et introduire de l'obscurité dans le code, voire augmenter les vulnérabilités, faute d'un minimum de compréhension du développeur.

Nous allons parcourir plusieurs cas d'école fournis par l'ANSSI, en partant d'exemples bogués qui démontrent une méconnaissance profonde de Java de la part du développeur, pour aller vers des cas moins évidents, qui exigent soit de renoncer à certaines particularités du langage, soit de bien paramétrer l'environnement de développement.

📖 Pour chaque exemple, on pourra consulter les documents de l'ANSII sur la sécurité du langage Java, en français :

JavaSec-Recommandations.pdf, JavaSec-Langage.pdf, JavaSec-Execution.pdf

4.2.1 Bien comprendre les mécanismes objet pour sécuriser ses programmes

Aucun compilateur ni outil ne pourra sécuriser à lui seul un développement, à la place du développeur. Placée dans de mauvaises mains, l'approche objet obscurcit le code, en multipliant les bugs et en facilitant les attaques.

Voici un premier exemple, fourni par l'ANSSI, qui était proposé sur un forum de développeurs :

http://thedailywtf.com/articles/Java-Destruction

Commentaires de l'extrait de code :

(1) La méthode *delete* est bien sûr boguée pour une raison simple : un objet est passé en paramètre par référence et la méthode appelée peut donc modifier son contenu par des *set*, mais elle ne peut pas mettre à *null* l'objet lui-même (car ce serait alors la copie de la référence dans la pile qui serait mise à *null* et pas la référence elle-même). Les réponses sur le forum sont nombreuses et très inquiétantes pour les développeurs, pas pour Java !

```
public class Destruction
{
(1)       public static void delete (Object object)
          {
             object = null;
          }
}
```

4.2.2 Limiter sa confiance et valider les bibliothèques externes

La première qualité exigible pour pratiquer le développement sécurisé est donc d'être le plus compétent possible dans son langage, et de bien comprendre les concepts du développement objet.

La deuxième est d'accorder sa confiance au minimum, aux classes et aux bibliothèques tierces. L'approche objet conduit à réutiliser le plus possible du code existant, en évitant de réinventer la poudre. Le développement sécurisé ne contredit pas cet objectif mais il le nuance, en changeant de perspective : le développeur logiciel, qui est souvent un assembleur de briques logicielles existantes, doit se montrer aussi responsable du code qu'il intègre que du code qu'il produit directement. Il doit soumettre les bibliothèques tierces à des procédures de validation pour tester leur sécurité, et éviter d'introduire dans l'application des chevaux de Troie, des *backdoor,* des *malware.*

Cette méthodologie de validation avec une approche sécurité sera développée dans d'autres séances. Mais nous allons illustrer le principe de sécurité « accorder sa confiance au minimum », en mettant en évidence les dangers qui peuvent se cacher dans des classes tierces, y compris dans une utilisation apparemment anodine.

4.2.3 Une classe peut faire plus qu'on ne croit et autre chose que ce que l'on voudrait

Le développement sécurisé nous demande de changer de point de vue sur le code, en considérant qu'il n'y RIEN d'anodin et qu'il faut TOUT valider.

Dans l'exemple suivant, fourni par l'ANSSI, on consulte une variable *pi* fournie par une classe tierce *Mathf.* En apparence, cette classe ne devrait pas nous causer de problème, puisque :

- On n'appelle aucune de ses méthodes
- On ne crée pas d'objets de cette classe (et donc pas d'appel à un constructeur)

Mais il ne faut pas oublier que les classes Java peuvent contenir des « **blocs d'initialisation statiques** » qui sont exécutés à la première utilisation de la classe, et donc à la consultation de la variable *pi*.

Commentaires de l'extrait de code :

(1) La classe tierce *Mathf* (et non pas la classe standard *Math* de Java), possède un bloc d'initialisation statique, appelé au premier chargement de la classe, qui exporte une valeur incorrecte de pi et peut faire « ce qu'il veut ».

(2) Le *main* qui utilise simplement la classe tierce, en consultant la variable *Mathf.pi,* peut donc faire l'objet d'une attaque, insérée dans les blocs d'initialisation statique.

```
public class Mathf {
    public static double pi;

    (1)  static
    {
        System.out.println("Bloc d'initialisation de Mathf)");
        pi = Math.PI + 0.2;
        System.out.println("Ici, je fais ce que je veux (sans rapport avec pi");
    }
}
```

```
public class StaticInit
{
    public static void main(String[] args)
    {
    (2)  if (Mathf.pi - 3.1415 < 0.0001) {
```

```
            System.out.println("Hello world : pi=" + Mathf.pi);
        }
        else
        {
            System.out.println("Hello strange universe : pi=" + Mathf.pi );
        }
    }
```

Deuxième exemple sur le même thème, où le source de la classe tierce *Friend* n'est pas connu (elle est dans un jar) et ses données sont sérialisées.

Commentaires de l'extrait de code :

(1) Comme la classe tierce *Mathf*, la classe *Friend* possède un bloc d'initialisation statique, qui sera appelé lorsque le *main* qui l'utilise va désérialiser le premier objet de cette classe.

(2) Que font réellement le constructeur et la méthode *getSomething*, dans une classe *Friend* qui « se veut votre amie », mais dont vous ne possédez pas le source ?

(3) Deux risques chez l'appelant, si la sécurité du code de la classe tierce n'est pas validée : un risque visible dans l'appel de la méthode *getSomething()*, et un risque caché dans la désérialisation de l'objet **f**, qui entraîne l'exécution des blocs d'initialisation statique.

```
public class Friend implements Serializable
{
    private String s = null;
    (1)  static
    {
          System.out.println("et là, je suis méchant !");
    }
    public Friend()
    {
   (2) // Do Something : code inconnu ?
        s = "something";
    }
    public String getSomething ()
    {
      (2)  return s;
    }
```

```
public class Deserial
{
    public static void main(String[] args)
            throws FileNotFoundException, IOException,
            ClassNotFoundException
    {
        FileInputStream fis = new FileInputStream("friend");
        ObjectInputStream ois = new ObjectInputStream(fis);
    (3) Friend f = (Friend) ois.readObject();

    (3)     System.out.println("Je récupère bien : " + f.getSomething() );
    }
}
```

4.2.4 Connaître les limites du compilateur Java

Comparé aux langages interprétés comme JavaScript ou à un langage compilé comme le C, l'analyseur syntaxique de Java est très fiable : mais posons-nous la question de ce qui se passe avant l'analyse syntaxique et après.

Avant l'analyseur syntaxique ?

En standard, le compilateur Java n'accepte pas seulement les caractères ASCII mais l'encodage UTF-8.

📖 Sur l'encodage UTF-8, voir https://fr.wikipedia.org/wiki/UTF-8

Cet encodage est très délicat à traiter. Un exemple ANSSI montre comment les caractères UTF-8 sont traités par un préprocesseur, avant l'analyse syntaxique proprement dite. Ce préprocesseur traite les caractères UTF-8 de la même manière, indépendamment de leur signification pour le langage Java : dans le code lui-même, dans les strings, dans les commentaires. L'utilisation de caractères UTF-8 dans les commentaires est à proscrire, car elle interdit toute compréhension intuitive du code source.

Commentaires de l'extrait de code :

(1) Dans le commentaire en rouge, les codes UTF-8 sont préprocessés, AVANT la compilation, en ignorant le commentaire :

- le caractère UTF-8 \u00a est un LINE FEED qui va à la ligne suivante et sort du commentaire //

- le caractère \u007d est une accolade fermante } qui termine donc le bloc du if

- le caractère UNICODE \u007b est une accolade ouvrant {

(2) Ce que le compilateur va traiter. L'affichage se fait donc de façon inconditionnelle.

```
public class Preprocess
{    public static void ma\u0069n(String[] args)
    {
       (1)
        if (false == true)
        { // \u000a\u007d\u007b
            System.out.println("Bad things happen!");
        }
    }
}
```

```
public class Preprocess
{    public static void main (String[] args)
    {
        if (false == true)
        {  //
        }
        {  System.out.println("Bad things happen!");
        }
    }
}
```

Et après l'analyseur syntaxique ?

On sait que le compilateur Java ne produit pas du code natif, comme le langage C, mais *du bytecode*, code intermédiaire qui va être interprété et exécuté par la machine virtuelle Java.

Les règles du *bytecode* ne sont pas exactement les mêmes que celles du source Java : certains contrôles sur le code source peuvent disparaître dans le *bytecode* (voire dépendre d'un paramétrage).

L'exemple ANSSI qui suit, illustre cette difficulté, par le cas des classes internes (*inner class*) en Java. Dans le code source, une classe interne peut utiliser l'annotation *private* pour limiter son accès. Mais cette annotation disparaît dans le *bytecode*, et on peut de fait, utiliser les variables annotées *private*, comme si elles étaient *public*.

Commentaires de l'extrait de code :

(1) Traits du source non préservés dans le *bytecode* : les classes internes.

Java permet de définir des classes internes, mais elles ne sont pas supportées en *bytecode*. Le compilateur remet tout à plat … et retire les *private*.

(2) La méthode *println* de la classe *Innerinner* affiche l'attribut privé ***a*** de la classe principale *Innerclass*

(3) Inversement, le *main* de la classe principale affiche l'attribut privé ***b*** de la classe interne *Innerinner*

```
public class Innerclass
{
    private static int a = 42;

(1) static public class Innerinner
    {
        private static int b = 54;
        public static void print()
        {
 (2)        System.out.println(Innerclass.a);
        }
    }
    public static void main(String[] args)
    {
        System.out.println(Innerinner.b);
 (3)    Innerinner.print();
    }
}
```

4.2.5 Bien choisir ses types pour vraiment sécuriser ses interfaces

Le compilateur Java est moins permissif que le C ou JavaScript, sur les conversions implicites entre les types, mais il faut l'aider un peu : ne le forcez pas à effectuer des conversions explicites absurdes, car vous en prenez la responsabilité ; le compilateur ne vérifie pas leur cohérence.

L'exemple suivant reprend en Java la fonction *safewrite* vue en C : on tente de sécuriser l'interface d'une fonction, en vérifiant que ses entrées sont dans les limites souhaitées, pour éviter le *buffer overflow*. Mais on ne tient pas compte des problèmes de typage, ce qui fait échouer les tests.

Commentaires de l'extrait de code :

(1) L'erreur de typage par inadvertance est détectée par le compilateur qui refuse d'affecter une constante entière (65534) à un *char :*

« incompatible types: possible lossy conversion from int to char »

Remarquons que dans ce cas, la conversion n'était pas risquée, car le type *char* en Java va de 0 à 65535.

(2) Si l'on insiste, par un « *cast* » explicite, Java accepte de compiler, que ce soit licite ou pas (erreur de transtypage avec 65538, qui dépasse la limite haute du type *char*).

(3) *L'integer overflow* se produit et on n'accède pas à la bonne case (2 au lieu de 65538)

```
public class RisqueTypages
{
    private static void safewrite(int tab[], int size, char ind, int val)
    {
      (3)  System.out.println("safewrite => i vaut : " + (int) ind);
        if (ind < size)
        {
            tab[ind] = val;
            System.out.println("==>Ecriture réussie \n");
        }
        else
        {
            System.out.println("==> Hors limite  \n" );
        }
    }
    public static void main(String[] args)
    {
        final int  MAX= 120;
        int tab[] = new int [MAX];
        System.out.println("ecriture a l'indice 65534");
   (1)    safewrite (tab, MAX, 65534, 1);

   (2)  safewrite (tab, MAX, (char) 65534, 1);
        System.out.println("ecriture a l'indice 65538\n");
   (2)  safewrite(tab, MAX, (char) 65538, 1);
    }
}
```

4.2.6 Se méfier des optimisations et ne pas parier sur elles

Tous les compilateurs modernes génèrent du code optimisé, en privilégiant la taille du code, sa vitesse, ou en faisant un compromis. Les optimisations peuvent surprendre, et être à l'origine de bugs et de vulnérabilités, si on fait des paris (souvent à son insu), sur ce qui sera effectivement généré. Il faut au contraire écrire du code « correct sur papier », le plus indépendant possible des optimisations.

L'exemple suivant, fourni par l'ANSSI, conduit à un résultat surprenant, en combinant une maladresse du développeur et une optimisation du compilateur.

Commentaires de l'extrait de code :

(1) Le partage maximal (*hash consing*) et ses surprises.

Principe de cette optimisation : Java optimise la déclaration de ses constantes en les partageant en mémoire dans un cache. Il ne faut surtout pas baser son raisonnement sur cette optimisation qui cache des pièges.

(2) Le mécanisme *d'AutoBoxing* permet à Java de ranger un type simple (*int*) dans l'objet *wrapper* correspondant (*Integer*). Le mécanisme inverse *d'UnBoxing* consiste à extraire le type simple de son objet *wrapper*.

(3) Pour respecter le principe de partage maximal, Java range les objets *wrapper* dans un cache mémoire, et tente de réutiliser le même *wrapper* pour une même constante numérique. Dans l'exemple, ***a2*** ne référence pas un nouveau *wrapper*, il pointe sur ***a1*** qui contient l'entier ***42***. Mais le cache a une taille limitée : le compilateur réserve donc un *Integer* ***b2*** distinct de **b1**. Ils contiennent tous les deux la même constante ***1000*** mais sont bien des objets distincts.

(4) Donc, dans cet exemple très mal pensé et faux sur papier, le programmeur compare les références des *Integer* par l'opérateur ==, au lieu de comparer les valeurs des deux entiers, comme il conviendrait (5). A cause du cache, la première comparaison est vraie, puisque le compilateur réserve un seul *wrapper* pour a1 et a2, mais la deuxième est fausse, puisque le compilateur utilise un *wrapper* b1 pour la première constante et un autre *wrapper* pour la deuxième. Du fait de cette mauvaise écriture, tout devient sujet à vulnérabilités : que se passerait-il si par introspection, on accédait à ce cache pour le corrompre ?

(5) Ce qu'il aurait fallu écrire, pour comparer correctement les deux entiers !

```
(1)
public class IntegerBoxing
{
    public static void main(String[] args)
    {
        (2) Integer a1 = 42;
        (3) Integer a2 = 42;
        if (a1 == a2)
        {
            System.out.println("a1 == a2");
        }
        Integer b1 = 1000;
        Integer b2 = 1000;
        if (b1 == b2)
        {
            System.out.println("b1 == b2");
        }
  (5)   if (b1.intValue() == b2.intValue())
        {
           System.out.println("En réalité, b1 == b2");
        }
    }
}
```

4.2.7 Encapsulation et introspection

L'encapsulation est très utile pour gérer l'abstraction et améliorer la qualité du code : chaque classe garantit sa cohérence interne, respecte des règles métier (comme vous l'avez découvert dans les séances sur le développement objet).

Rappelons d'abord que l'efficacité de ce mécanisme repose entièrement sur le développeur : quels contrôles effectue-t-on dans les méthodes *set* ? Si l'on n'effectue pas les bons contrôles sur le type et la taille des paramètres, un attribut *private* ne sera pas davantage sécurisé qu'un attribut *public*.

Mais de toute façon, l'encapsulation N'EST PAS une véritable protection contre les attaques. C'est une convention de bonne conduite entre l'appelant et la classe, qu'un appelant peut changer en utilisant « l'introspection » (package *java.lang.reflect*). Comme son nom l'indique, l'introspection permet d'interroger la classe, d'obtenir la liste de ses méthodes, de ses attributs et de changer leur niveau d'accessibilité.

L'exemple ci-dessous, fourni par l'ANSSI, utilise l'introspection pour changer une annotation *private* en *public* et accéder à un attribut (en principe privé) sans passer par un *get.*

Commentaires de l'extrait de code :

(1) La classe **Secret** contient un attribut privé **x** (normalement non accessible d'une classe tierce)

(2) Le *main* crée une instance de la classe Secret, récupère le type de la classe **c** par la méthode **getClass**

(3) Obtient un objet **f** de la classe **reflect.Field** qui va permettre d'agir sur le champ **x** d'une instance de la classe **Secret**.

(4) Modifie l'accessibilité du champ x et affiche sa valeur

(5) Ce n'est pas un objet particulier qui est modifié, mais l'accessibilité de la classe, et de toutes ses instances : on affiche aussi le champ **x** de l'objet **o2**

(6) On peut remettre l'accessibilité à son niveau d'origine (idéal pour une attaque discrète !)

(7) Le programme finit donc en exception, en tentant à nouveau d'afficher le champ privé x

```
java.lang.IllegalAccessException: Class demointrospection.Introspect can not access a
member of class demointrospection.Secret with modifiers "private"
```

```
import java.lang.reflect.Field;
 (1) class Secret
 {
     private int x = 42;
 }
public class Introspect
 {
  public static void main(String[] args)
  {
      try
      {
            Secret o = new Secret();
        (2) Class c = o.getClass();
        (3) Field f = c.getDeclaredField("x");
        (4) f.setAccessible(true);
            System.out.println("x=" + f.getInt(o));

        (5) Secret o2 = new Secret();
            System.out.println("x=" + f.getInt(o2));
        (6) f.setAccessible(false);
        (7) System.out.println("x=" + f.getInt(o));
         }
         catch (Exception e)
         {
        (7)  System.out.println(e);
         }
    }
}
```

5. CONCLUSION

Cette séance peut vous paraître inquiétante car elle révèle beaucoup de problèmes, en donnant seulement quelques conseils et des ébauches de solution. Celles-ci seront vues en détail dans la séance suivante sur la sécurité « Coder de façon défensive en suivant les bonnes pratiques de sécurité ».

CRÉDITS

OEUVRE COLLECTIVE DE L'AFPA

Sous le pilotage de la DIIP
et du centre sectoriel Tertiaire

EQUIPE DE CONCEPTION

Chantal PERRACHON – IF Neuilly-sur-Marne
Régis Lécu – Formateur AFPA Pont de Claix