La sécurité des applications Les débordements de tampon mémoire

V. Nicomette, E. Alata

INSA Toulouse

Introduction

Rappels sur les appels de fonction

Les attaques de type “buffer overflow”

La technique

Les `shellcode`

Les défenses actuelles

Détection automatique

Introduction

Rappels sur les appels de fonction

Les attaques de type “buffer overflow”

La technique

Les `shellcode`

Les défenses actuelles

Détection automatique

Introduction

- Un programme informatique, lorsqu'il s'exécute, intéragit avec son environnement
- Chaque point d'interaction ouvre la voix à des menaces potentielles
- Les menaces peuvent être locales ou distantes :
  - Menaces locales via : accès disque, accès mémoire, communications inter-processus (IPC), etc.
  - Menaces distantes via : utilisation de sockets, RPC, RMI (java), etc.

Introduction

- Un programme doit anticiper les actions malveillantes pouvant se produire lors de ces interactions :
  - Entrées du programme vérifiées et assainies
  - Contrôle des appels de programmes externes
  - Utilisation correcte de fonctions sûres
    par exemple `strncpy` *et* `strncat` *au lieu de* `strcpy` *et* `strcat` 1
    - `strcat(dst, src);` ?

Introduction

- Un programme doit anticiper les actions malveillantes pouvant se produire lors de ces interactions :
  - Entrées du programme vérifiées et assainies
  - Contrôle des appels de programmes externes
  - Utilisation correcte de fonctions sûres
    par exemple `strncpy` *et* `strncat` *au lieu de* `strcpy` *et* `strcat` 1
    - `strcat(dst, src);` ?
    - `strncat(dst, src, sizeof(dst));` ?

Introduction

- Un programme doit anticiper les actions malveillantes pouvant se produire lors de ces interactions :
  - Entrées du programme vérifiées et assainies
  - Contrôle des appels de programmes externes
  - Utilisation correcte de fonctions sûres
    par exemple `strncpy` *et* `strncat` *au lieu de* `strcpy` *et* `strcat` 1
    - `strcat(dst, src);` ?
    - `strncat(dst, src, sizeof(dst));` ?
    - `strncat(dst, src, sizeof(dst) - strlen(dst));` ?

Introduction

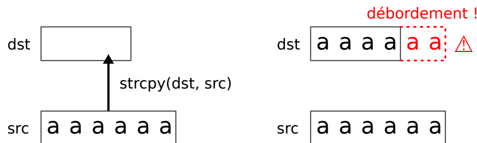
- Un programme doit anticiper les actions malveillantes pouvant se produire lors de ces interactions :
  - Entrées du programme vérifiées et assainies
  - Contrôle des appels de programmes externes
  - Utilisation correcte de fonctions sûres
    par exemple `strncpy` *et* `strncat` *au lieu de* `strcpy` *et* `strcat` [1]
    - `strcat(dst, src);` ?
    - `strncat(dst, src, sizeof(dst));` ?
    - `strncat(dst, src, sizeof(dst) - strlen(dst));` ?
    - `strncat(dst, src, sizeof(dst) - strlen(dst) - 1);` ?

Introduction

- Un programme doit anticiper les actions malveillantes pouvant se produire lors de ces interactions :
  - Entrées du programme vérifiées et assainies
  - Contrôle des appels de programmes externes
  - Utilisation correcte de fonctions sûres
    par exemple `strncpy` *et* `strncat` *au lieu de* `strcpy` *et* `strcat` 1
    - `strcat(dst, src);` ?
    - `strncat(dst, src, sizeof(dst));` ?
    - `strncat(dst, src, sizeof(dst) - strlen(dst));` ?
    - `strncat(dst, src, sizeof(dst) - strlen(dst) - 1);` ?
- Malheureusement, la majorité des attaques sur les applications proviennent du fait que les programmes sont peu scrupuleusement écrits
  les conseils précédents sont tout bonnement ignorés

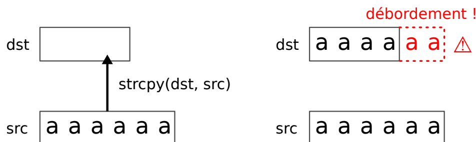
Un avant-goût des buffer overflow 1/2

- Accès à l'origine : (lecture) ou (éctriture ●)
  Focus sur les accès en écriture
- Causes : mauvais indice d'un tableau, `strcpy` mal maîtrisé, etc.



- Analyse de la situation :
  - *Sur quoi on déborde ? Qu'est ce que c'est ?*
  - *Comment peut-on le modifier de manière cohérente ?*
  - *Est-ce que le débordement est limité en taille ?*

Un avant-goût des buffer overflow 2/2



- Les possibilités/conséquences dépendent du lieu du débordement
  - pile : données de contrôle d'éxécution ou variables du programme
  - tas : données gérées par la `libc` ou variables du programme
  - data : variables du programme
  - code : pas en écriture ! (cf. pagination)
- L'exploitation nécessite de connaitre la sémantique des données écrasées
  - données de contrôle d'éxécution ⇒ spécification du processeur
  - données gérées par la `libc` ⇒ implémentation de `malloc`
  - variables du programme ⇒ sémantique du programme (rétro-conception...)
- Conséquences de l'exploitation : déni de service (`segmentation fault`), changement de comportement du processus vulnérable, escalade de privilèges, etc.

Introduction

Rappels sur les appels de fonction

Les attaques de type “buffer overflow”

La technique

Les `shellcode`

Les défenses actuelles

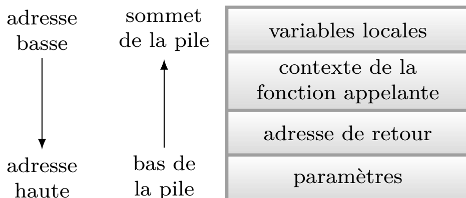
Détection automatique

Les outils à connaître

- `gcc` – pour la compilation
- `ddd` – pour le *debuggage*
- `gdb` – pour le *debuggage*
- `objdump` – pour le désassemblage

Appel de fonction

- Lors de l'appel d'une fonction en langage C, on empile dans l'ordre :
  - Les paramètres de la fonction invoquée
  - L'adresse de retour de la fonction appelante
    adresse de l'instruction qui suit l'invocation
  - Une sauvegarde du contexte d'exécution de la fonction appelante
    dépend de l'architecture matérielle (état de la pile, etc.)
  - Les variables locales




```
void f(int a, char* str) {
  char ch[8];
  int var;
}
void main(int argc, char** argv) {
  f(4, argv[1]);
}
```

Appel de fonction

- Lors de l'appel d'une fonction en langage C, on empile dans l'ordre :
  - Les paramètres de la fonction invoquée
  - L'adresse de retour de la fonction appelante
    adresse de l'instruction qui suit l'invocation
  - Une sauvegarde du contexte d'exécution de la fonction appelante
    dépend de l'architecture matérielle (état de la pile, etc.)
  - Les variables locales

- Lors du retour, on dépile dans l'ordre :
  - Les variables locales
  - La sauvegarde du contexte d'exécution de la fonction appelante
    le contexte est restauré
  - L'adresse de retour de la fonction appelante
    retour au niveau de l'instruction qui suit l'invocation
  - Les paramètres de la fonction invoquée

Appel de fonction sur x86 1/3

- Le registre `esp` est le sommet de la pile
  - Il évolue avec les `push`/`pop`
  - `push` : un élément est ajouté en sommet de pile et `esp` est décrémenté
  - `pop` : un élément est retiré du sommet de pile et `esp` est incrémenté
  - ⇒ la taille de la pile évolue au cours de l'exécution des fonctions
  - NB `push` *et* `pop` *peuvent être remplacés par des* `mov`, `add` *et* `sub`
- Le registre `ebp` est le pointeur de base de pile
  - Difficile de localiser les paramètres avec `esp` (taille de pile variable)
  - Utilisation d'un registre qui pointe sur le début d'une zone de la pile réservée à la fonction en cours d'exécution
  - Lors de l'appel d'une fonction, ce registre doit donc être sauvegardé pour permettre à la fonction appelée de disposer de sa propre zone de pile
- La pile *grandit* vers le bas, en opposition au tas qui *grandit* vers le haut

Appel de fonction sur x86 2/3

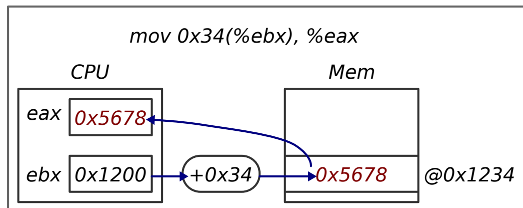
Décompilation du binaire : `objdump -d program` 2

```
f :         08048cb0  push   %ebp
            08048cb1  mov    %esp,%ebp
            08048cb3  sub    $0x10,%esp
            08048cb6  leave
            08048cb7  ret
main :  →   08048cb8  push   %ebp
            08048cb9  mov    %esp,%ebp
            08048cbb  sub    $0x8,%esp
            08048cbe  mov    Oxc(%ebp),%eax
            08048cc1  add    $0x4,%eax
            08048cc4  mov    (%eax),%eax
            08048cc6  mov    %eax,Ox4(%esp)
            08048cca  movl   $0x4,(%esp)
            08048cd1  call   8048cb0
            08048cd6  leave
            08048cd7  ret
```

Appel de fonction sur x86 2/3

Décompilation du binaire : `objdump -d program` 2

```
f :           08048cb0  push   %ebp
              08048cb1  mov    %esp,%eb
              08048cb3  sub    $0x10,%es
              08048cb6  leave
              08048cb7  ret
main :  →     08048cb8  push   %ebp
              08048cb9  mov    %esp,%eb
              08048cbb  sub    $0x8,%esp
              08048cbe  mov    Oxc(%ebp
              08048cc1  add    $0x4,%eax
              08048cc4  mov    (%eax),%eax
              08048cc6  mov    %eax,Ox4(%esp)
              08048cca  movl   $0x4,(%esp)
              08048cd1  call   8048cb0
              08048cd6  leave
              08048cd7  ret
```



Appel de fonction sur x86 2/3

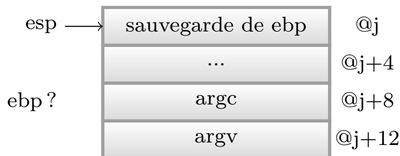
Décompilation du binaire : `objdump -d program` 2

```
f :        08048cb0  push  %ebp
           08048cb1  mov   %esp,%ebp
           08048cb3  sub   $0x10,%esp
           08048cb6  leave
           08048cb7  ret
main :  →  08048cb8  push  %ebp
           08048cb9  mov   %esp,%ebp
           08048cbb  sub   $0x8,%esp
           08048cbe  mov   Oxc(%ebp),%eax
           08048cc1  add   $0x4,%eax
           08048cc4  mov   (%eax),%eax
           08048cc6  mov   %eax,Ox4(%esp)
           08048cca  movl  $0x4,(%esp)
           08048cd1  call  8048cb0
           08048cd6  leave
           08048cd7  ret
```

Appel de fonction sur x86 2/3

Décompilation du binaire : `objdump -d program` 2

```
f :       08048cb0  push  %ebp
          08048cb1  mov   %esp,%ebp
          08048cb3  sub   $0x10,%esp
          08048cb6  leave
          08048cb7  ret
main :    08048cb8  push  %ebp
      →   08048cb9  mov   %esp,%ebp
          08048cbb  sub   $0x8,%esp
          08048cbe  mov   Oxc(%ebp),%eax
          08048cc1  add   $0x4,%eax
          08048cc4  mov   (%eax),%eax
          08048cc6  mov   %eax,Ox4(%esp)
          08048cca  movl  $0x4,(%esp)
          08048cd1  call  8048cb0
          08048cd6  leave
          08048cd7  ret
```



Appel de fonction sur x86 2/3

Décompilation du binaire : `objdump -d program` 2

```
f :      08048cb0  push  %ebp
         08048cb1  mov   %esp,%ebp
         08048cb3  sub   $0x10,%esp
         08048cb6  leave
         08048cb7  ret
main :   08048cb8  push  %ebp
         08048cb9  mov   %esp,%ebp
      →  08048cbb  sub   $0x8,%esp
         08048cbe  mov   Oxc(%ebp),%eax
         08048cc1  add   $0x4,%eax
         08048cc4  mov   (%eax),%eax
         08048cc6  mov   %eax,Ox4(%esp)
         08048cca  movl  $0x4,(%esp)
         08048cd1  call  8048cb0
         08048cd6  leave
         08048cd7  ret
```

Appel de fonction sur x86 2/3

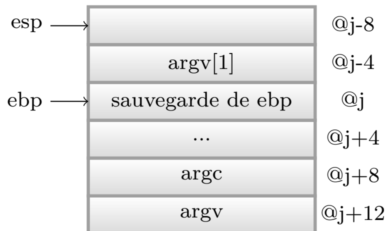
Décompilation du binaire : `objdump -d program` 2

```
f :      08048cb0  push   %ebp
         08048cb1  mov    %esp,%ebp
         08048cb3  sub    $0x10,%esp
         08048cb6  leave
         08048cb7  ret
main :   08048cb8  push   %ebp
         08048cb9  mov    %esp,%ebp
         08048cbb  sub    $0x8,%esp
      →  08048cbe  mov    Oxc(%ebp),%eax
      →  08048cc1  add    $0x4,%eax
      →  08048cc4  mov    (%eax),%eax
      →  08048cc6  mov    %eax,Ox4(%esp)
         08048cca  movl   $0x4,(%esp)
         08048cd1  call   8048cb0
         08048cd6  leave
         08048cd7  ret
```

Appel de fonction sur x86 2/3

Décompilation du binaire : `objdump -d program` 2

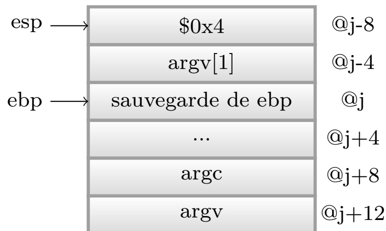
`f :`	08048cb0	push	%ebp
	08048cb1	mov	%esp,%ebp
	08048cb3	sub	$0x10,%esp
	08048cb6	leave	
	08048cb7	ret	
`main :`	08048cb8	push	%ebp
	08048cb9	mov	%esp,%ebp
	08048cbb	sub	$0x8,%esp
	08048cbe	mov	Oxc(%ebp),%eax
	08048cc1	add	$0x4,%eax
	08048cc4	mov	(%eax),%eax
	08048cc6	mov	%eax,Ox4(%esp)
→	08048cca	movl	$0x4,(%esp)
	08048cd1	call	8048cb0
	08048cd6	leave	
	08048cd7	ret	



Appel de fonction sur x86 2/3

Décompilation du binaire : `objdump -d program` 2

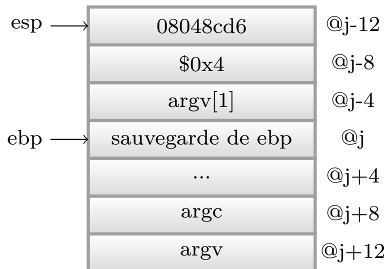
```
f :      08048cb0  push   %ebp
         08048cb1  mov    %esp,%ebp
         08048cb3  sub    $0x10,%esp
         08048cb6  leave
         08048cb7  ret
main :   08048cb8  push   %ebp
         08048cb9  mov    %esp,%ebp
         08048cbb  sub    $0x8,%esp
         08048cbe  mov    Oxc(%ebp),%eax
         08048cc1  add    $0x4,%eax
         08048cc4  mov    (%eax),%eax
         08048cc6  mov    %eax,Ox4(%esp)
         08048cca  movl   $0x4,(%esp)
      →  08048cd1  call   8048cb0
         08048cd6  leave
         08048cd7  ret
```



Appel de fonction sur x86 2/3

Décompilation du binaire : `objdump -d program` 2

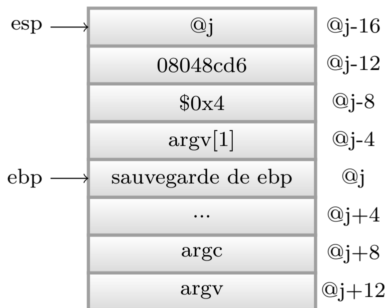
```
f :     → 08048cb0  push  %ebp
          08048cb1  mov   %esp,%ebp
          08048cb3  sub   $0x10,%esp
          08048cb6  leave
          08048cb7  ret
main :    08048cb8  push  %ebp
          08048cb9  mov   %esp,%ebp
          08048cbb  sub   $0x8,%esp
          08048cbe  mov   Oxc(%ebp),%eax
          08048cc1  add   $0x4,%eax
          08048cc4  mov   (%eax),%eax
          08048cc6  mov   %eax,Ox4(%esp)
          08048cca  movl  $0x4,(%esp)
          08048cd1  call  8048cb0
          08048cd6  leave
          08048cd7  ret
```



Appel de fonction sur x86 2/3

Décompilation du binaire : `objdump -d program` 2

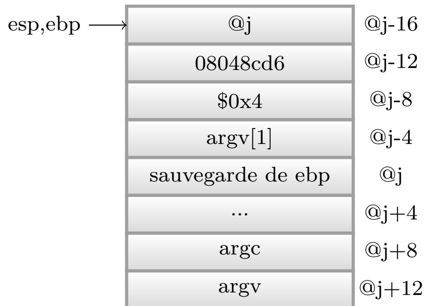
```
f :       08048cb0  push  %ebp
       →  08048cb1  mov   %esp,%ebp
          08048cb3  sub   $0x10,%esp
          08048cb6  leave
          08048cb7  ret
main :    08048cb8  push  %ebp
          08048cb9  mov   %esp,%ebp
          08048cbb  sub   $0x8,%esp
          08048cbe  mov   Oxc(%ebp),%eax
          08048cc1  add   $0x4,%eax
          08048cc4  mov   (%eax),%eax
          08048cc6  mov   %eax,Ox4(%esp)
          08048cca  movl  $0x4,(%esp)
          08048cd1  call  8048cb0
          08048cd6  leave
          08048cd7  ret
```



Appel de fonction sur x86 2/3

Décompilation du binaire : `objdump -d program` 2

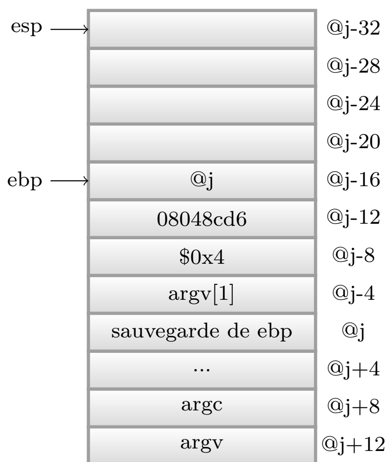
```
f :       08048cb0  push  %ebp
          08048cb1  mov   %esp,%ebp
      →   08048cb3  sub   $0x10,%esp
          08048cb6  leave
          08048cb7  ret
main :    08048cb8  push  %ebp
          08048cb9  mov   %esp,%ebp
          08048cbb  sub   $0x8,%esp
          08048cbe  mov   Oxc(%ebp),%eax
          08048cc1  add   $0x4,%eax
          08048cc4  mov   (%eax),%eax
          08048cc6  mov   %eax,Ox4(%esp)
          08048cca  movl  $0x4,(%esp)
          08048cd1  call  8048cb0
          08048cd6  leave
          08048cd7  ret
```



Appel de fonction sur x86 2/3

Décompilation du binaire : `objdump -d program` 2

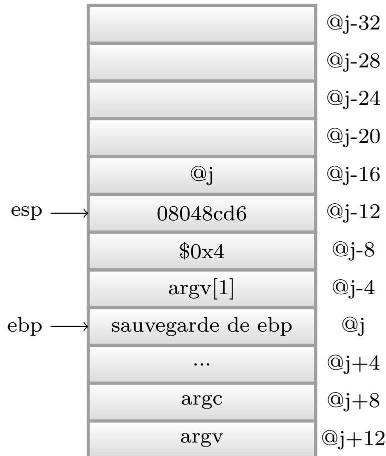
```
f :       08048cb0  push  %ebp
          08048cb1  mov   %esp,%ebp
          08048cb3  sub   $0x10,%esp
       →  08048cb6  leave
          08048cb7  ret
main :    08048cb8  push  %ebp
          08048cb9  mov   %esp,%ebp
          08048cbb  sub   $0x8,%esp
          08048cbe  mov   Oxc(%ebp),%eax
          08048cc1  add   $0x4,%eax
          08048cc4  mov   (%eax),%eax
          08048cc6  mov   %eax,Ox4(%esp)
          08048cca  movl  $0x4,(%esp)
          08048cd1  call  8048cb0
          08048cd6  leave
          08048cd7  ret
```



Appel de fonction sur x86 2/3

Décompilation du binaire : `objdump -d program` 2

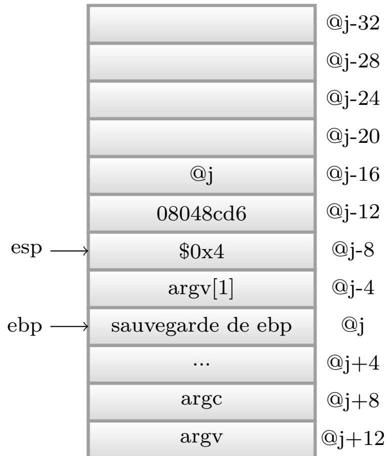
```
f :        08048cb0  push  %ebp
           08048cb1  mov   %esp,%ebp
           08048cb3  sub   $0x10,%esp
           08048cb6  leave
       →   08048cb7  ret
main :     08048cb8  push  %ebp
           08048cb9  mov   %esp,%ebp
           08048cbb  sub   $0x8,%esp
           08048cbe  mov   Oxc(%ebp),%eax
           08048cc1  add   $0x4,%eax
           08048cc4  mov   (%eax),%eax
           08048cc6  mov   %eax,Ox4(%esp)
           08048cca  movl  $0x4,(%esp)
           08048cd1  call  8048cb0
           08048cd6  leave
           08048cd7  ret
```



Appel de fonction sur x86 2/3

Décompilation du binaire : `objdump -d program` 2

```
f :      08048cb0  push  %ebp
         08048cb1  mov   %esp,%ebp
         08048cb3  sub   $0x10,%esp
         08048cb6  leave
         08048cb7  ret
main :   08048cb8  push  %ebp
         08048cb9  mov   %esp,%ebp
         08048cbb  sub   $0x8,%esp
         08048cbe  mov   Oxc(%ebp),%eax
         08048cc1  add   $0x4,%eax
         08048cc4  mov   (%eax),%eax
         08048cc6  mov   %eax,Ox4(%esp)
         08048cca  movl  $0x4,(%esp)
         08048cd1  call  8048cb0
     →   08048cd6  leave
         08048cd7  ret
```



Appel de fonction sur x86 3/3

- Etat de la pile avant et durant l'invocation de la fonction

```
void f(int a, char* str) {
   char ch[8] ;
   int var ;
}
void main(int argc, char** argv) {
   f(4, argv[1]) ;
}
```

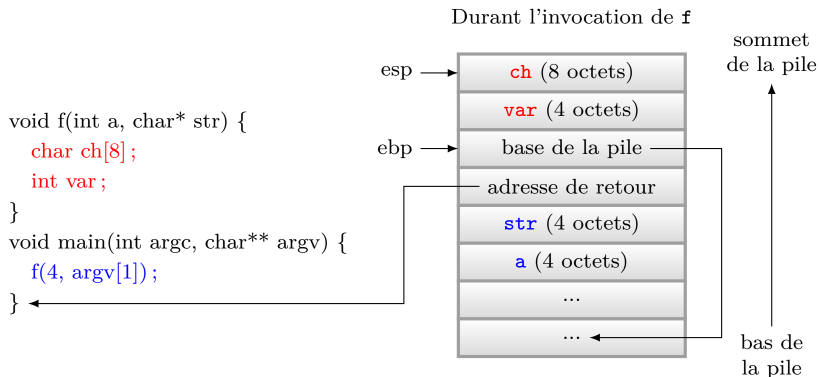
Avant l'invocation de **f**

sommet de la pile

bas de la pile

Appel de fonction sur x86 3/3

- Etat de la pile avant et durant l'invocation de la fonction



Récapitulatif

- Une fonction a besoin d'un espace pour stocker ses variables locales
- Une fonction peut être récursive (même indirectement)

⇒ A un instant, il peut y avoir deux contextes pour cette fonction

- L'espace dédié aux variables locales doit être propre à chaque exécution de la fonction
- Une fonction doit pouvoir localiser ses paramètres fournis par la fonction appelante
- A la fin d'une fonction, le processeur doit pouvoir déterminer l'adresse à laquelle poursuivre l'exécution – dans la fonction appelante
- Une fonction peut être invoquée par différentes fonctions appelante ⇒ l'adresse de retour n'est pas unique
- Une fonction doit être indépendante des implémentations des autres fonctions
- Une fonction doit savoir comment invoquer les autres fonctions sans connaître leur implémentation
- Une fonction doit pouvoir récupérer ses paramètres sans connaître l'implémentation des fonctions appelantes

Introduction

Rappels sur les appels de fonction

Les attaques de type “buffer overflow”

La technique

Les `shellcode`

Les défenses actuelles

Détection automatique

La vulnérabilité

- Vulnérabilité très répandue
- Ecriture dans un tableau d'une donnée de taille supérieure à celle du tableau
- La donnée écrite déborde du tableau
- Entraîne souvent, à l'exécution, un message d'erreur du type `Segmentation fault`, lors du débordement sur un segment mémoire n'appartenant pas au processus 3
- Risque d'écrire dans une zone mémoire qui contrôle l'exécution du programme
- Possibilité pour un attaquant d'exploiter ce risque pour détourner l'exécution du programme voire lui faire exécuter un code arbitraire
- Tous les langages de programmation ne permettent pas ce genre de débordement
  le langage C oui !

Principe de l'exploitation

- En général, le code arbitraire est un invité de commande (Shell sous Unix)
- Ce code arbitraire est exécuté avec les privilèges du processus ciblé
  intérêt du principe de moindre privilèges
- Si le programme attaqué est exécuté avec les privilèges `root`, l'intérêt est bien supérieur
  $\Rightarrow$ *contrôle de la machine*
- Ce débordement peut avoir lieu dans la pile (premières attaques publiées) mais aussi dans le tas
- Nous n'aborderons dans ce cours que les attaques dans la pile

Exemple de fonction vulnérable 1/5

- Soit le programme `program.c` suivant :

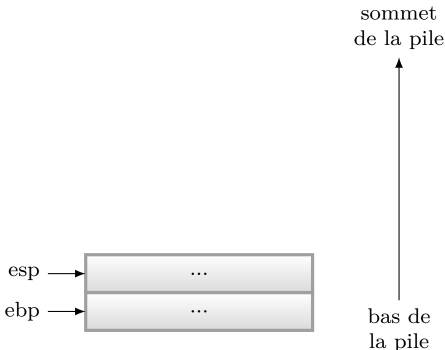
```
void f(int a, char* str) {
  char ch[8];
  int var;
  // Copie dangereuse !
  strcpy(ch, str);
}
void main(int argc, char** argv) {
  f(4, argv[1]);
}
```

- `str` est une donnée fournie par l'utilisateur
  `./program truc`
- La fonction `strcpy(ch, str)` copie le contenu de `str` à l'adresse `ch` *sans vérifier que la place mémoire est suffisante*

Exemple de fonction vulnérable 2/4

Si la taille de `str` est inférieure à 8, pas de problème
écrasement de `ch` uniquement
Commande : `./program XXXX 4`

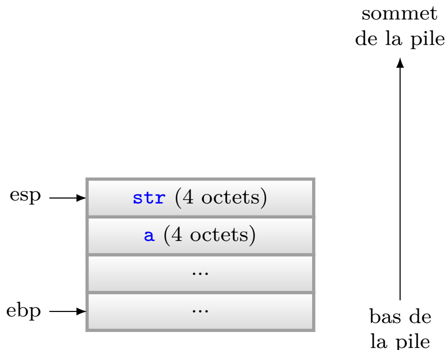
```
void f(int a, char* str) {
   char ch[8];
   int var;
   // Copie dangeureuse !
   strcpy(ch, str);
}
void main(int argc, char** argv) {
   f(4, argv[1]);
}
```



Exemple de fonction vulnérable 2/4

Si la taille de `str` est inférieure à 8, pas de problème
écrasement de `ch` uniquement
Commande : `./program XXXX` 4

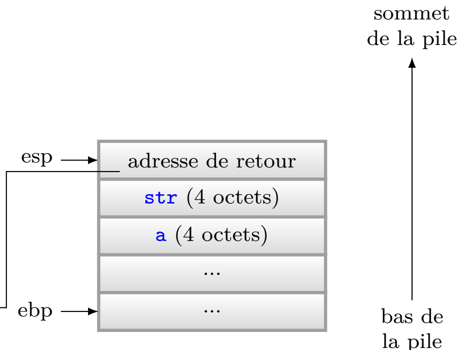
```
void f(int a, char* str) {
   char ch[8];
   int var;
   // Copie dangeureuse !
   strcpy(ch, str);
}
void main(int argc, char** argv) {
→  f(4, argv[1]);
}
```



Exemple de fonction vulnérable 2/4

Si la taille de `str` est inférieure à 8, pas de problème
écrasement de `ch` uniquement
Commande : `./program XXXX` 4

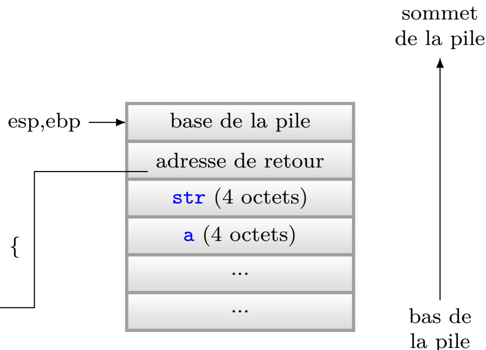
```
→ void f(int a, char* str) {
    char ch[8];
    int var;
    // Copie dangeureuse !
    strcpy(ch, str);
  }
  void main(int argc, char** argv) {
    f(4, argv[1]);
  }
```



Exemple de fonction vulnérable 2/4

Si la taille de `str` est inférieure à 8, pas de problème
écrasement de `ch` uniquement
Commande : `./program XXXX 4`

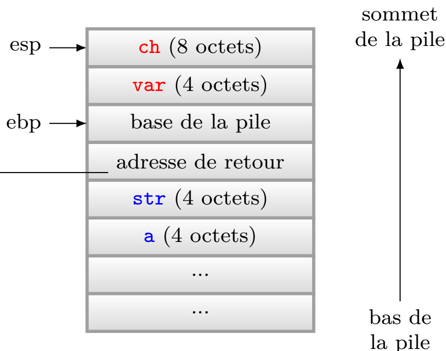
```
→ void f(int a, char* str) {
    char ch[8];
    int var;
    // Copie dangeureuse !
    strcpy(ch, str);
  }
  void main(int argc, char** argv) {
    f(4, argv[1]);
  }
```



Exemple de fonction vulnérable 2/4

Si la taille de `str` est inférieure à 8, pas de problème
écrasement de `ch` uniquement
Commande : `./program XXXX 4`

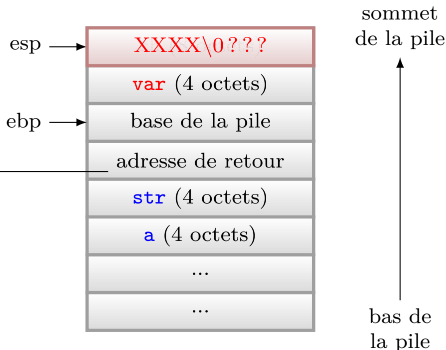
```
void f(int a, char* str) {
    char ch[8];
    int var;
    // Copie dangeureuse !
    strcpy(ch, str);
}
void main(int argc, char** argv) {
    f(4, argv[1]);
}
```



Exemple de fonction vulnérable 2/4

Si la taille de `str` est inférieure à 8, pas de problème
écrasement de `ch` uniquement
Commande : `./program XXXX 4`

```
void f(int a, char* str) {
   char ch[8];
   int var;
   // Copie dangeureuse !
   strcpy(ch, str);
}
void main(int argc, char** argv) {
   f(4, argv[1]);
}
```



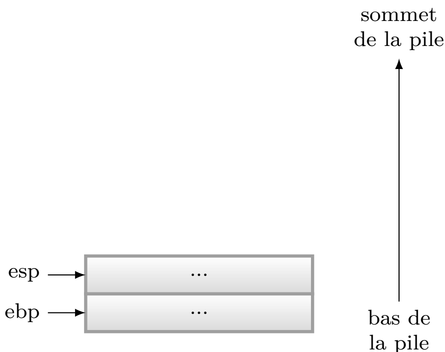
Exemple de fonction vulnérable 3/4

Si la taille de `str` est supérieure à 8 et inférieure à 12, problème écrasement de `ch` **et des octets de** `var`

Attention ! Si `var` *avait été utilisé pour un contrôle d'authentification, sa modification aurait permis à un attaquant de forcer cette authentification !*

Commande : `./program XXXXXXXXXXX` 4

```
void f(int a, char* str) {
    char ch[8];
    int var;
    // Copie dangeureuse !
    strcpy(ch, str);
}
void main(int argc, char** argv) {
    f(4, argv[1]);
}
```



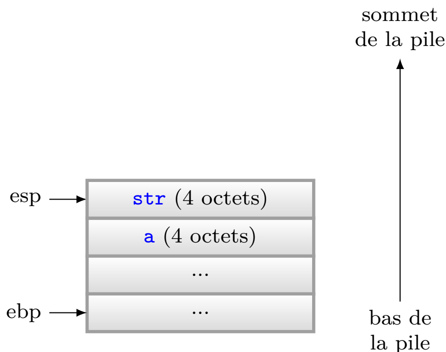
Exemple de fonction vulnérable 3/4

Si la taille de **`str`** est supérieure à 8 et inférieure à 12, problème
écrasement de `ch` **et des octets de** `var`

Attention ! Si `var` *avait été utilisé pour un contrôle d'authentification, sa modification aurait permis à un attaquant de forcer cette authentification !*

Commande : `./program XXXXXXXXXXX` 4

```
void f(int a, char* str) {
  char ch[8];
  int var;
  // Copie dangeureuse !
  strcpy(ch, str);
}
void main(int argc, char** argv) {
→ f(4, argv[1]);
}
```



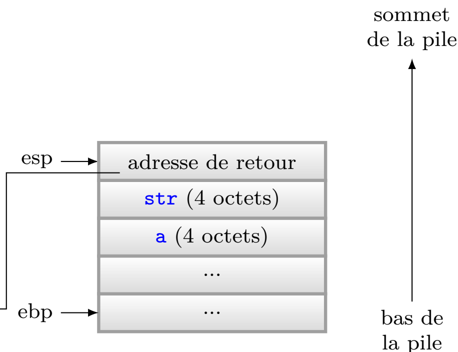
Exemple de fonction vulnérable 3/4

Si la taille de `str` est supérieure à 8 et inférieure à 12, problème écrasement de `ch` **et des octets de** `var`

Attention ! Si `var` *avait été utilisé pour un contrôle d'authentification, sa modification aurait permis à un attaquant de forcer cette authentification !*

Commande : `./program XXXXXXXXXXX` 4

```
→ void f(int a, char* str) {
    char ch[8];
    int var;
    // Copie dangeureuse !
    strcpy(ch, str);
  }
  void main(int argc, char** argv) {
    f(4, argv[1]);
  }
```

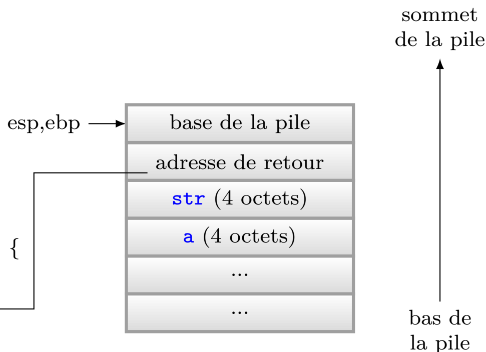


Exemple de fonction vulnérable 3/4

Si la taille de `str` est supérieure à 8 et inférieure à 12, problème écrasement de `ch` **et des octets de** `var`

Attention ! Si `var` *avait été utilisé pour un contrôle d'authentification, sa modification aurait permis à un attaquant de forcer cette authentification !*

Commande : `./program XXXXXXXXXXX` 4




```
void f(int a, char* str) {
    char ch[8];
    int var;
    // Copie dangeureuse !
    strcpy(ch, str);
}
void main(int argc, char** argv) {
    f(4, argv[1]);
}
```

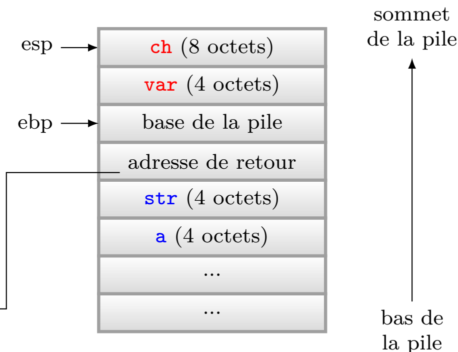
Exemple de fonction vulnérable 3/4

Si la taille de `str` est supérieure à 8 et inférieure à 12, problème écrasement de `ch` **et des octets de** `var`

Attention ! Si `var` *avait été utilisé pour un contrôle d'authentification, sa modification aurait permis à un attaquant de forcer cette authentification !*

Commande : `./program XXXXXXXXXXX` 4

```
void f(int a, char* str) {
    char ch[8];
    int var;
    // Copie dangeureuse !
    strcpy(ch, str);
}
void main(int argc, char** argv) {
    f(4, argv[1]);
}
```

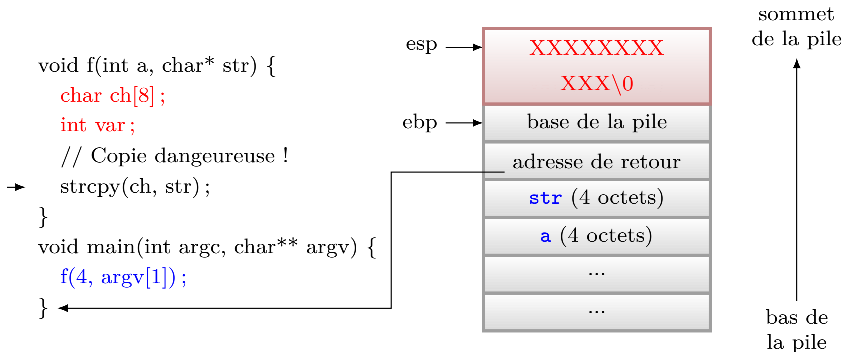


Exemple de fonction vulnérable 3/4

Si la taille de `str` est supérieure à 8 et inférieure à 12, problème écrasement de `ch` **et des octets de** `var`

Attention ! Si `var` *avait été utilisé pour un contrôle d'authentification, sa modification aurait permis à un attaquant de forcer cette authentification !*

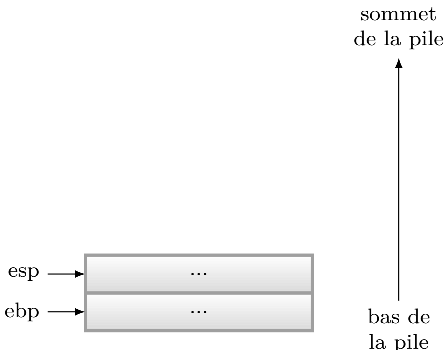
Commande : `./program XXXXXXXXXXX` 4



Exemple de fonction vulnérable 4/4

Si la taille de `str` est supérieure à 16, problème
écrasement de `ch`, de `var`, du pointeur de base de la pile et de l'adresse de retour !
Commande : `./program XXXXXXXXXXXXXXXXXXX 4`

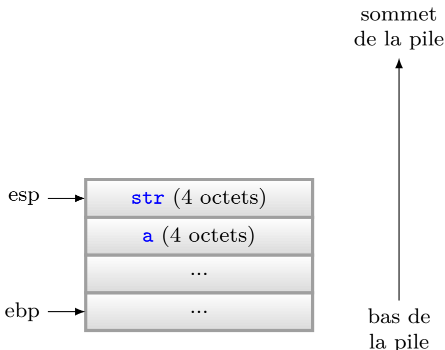
```
void f(int a, char* str) {
  char ch[8] ;
  int var ;
  // Copie dangeureuse !
  strcpy(ch, str) ;
}
void main(int argc, char** argv) {
  f(4, argv[1]) ;
}
```



Exemple de fonction vulnérable 4/4

Si la taille de `str` est supérieure à 16, problème
écrasement de `ch`, de `var`, du pointeur de base de la pile et de l'adresse de retour !
Commande : `./program XXXXXXXXXXXXXXXXXXXX` 4

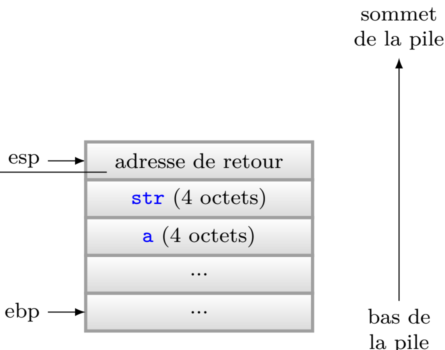
```
void f(int a, char* str) {
    char ch[8];
    int var;
    // Copie dangeureuse !
    strcpy(ch, str);
}
void main(int argc, char** argv) {
    f(4, argv[1]);
}
```



Exemple de fonction vulnérable 4/4

Si la taille de `str` est supérieure à 16, problème
écrasement de `ch`, de `var`, du pointeur de base de la pile et de l'adresse de retour !
Commande : `./program XXXXXXXXXXXXXXXXXXXX` 4

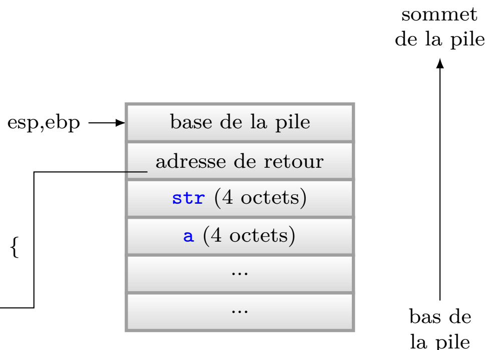
```
→ void f(int a, char* str) {
    char ch[8];
    int var;
    // Copie dangeureuse !
    strcpy(ch, str);
  }
  void main(int argc, char** argv) {
    f(4, argv[1]);
  }
```



Exemple de fonction vulnérable 4/4

Si la taille de `str` est supérieure à 16, problème
écrasement de `ch`, de `var`, du pointeur de base de la pile et de l'adresse de retour !
Commande : `./program XXXXXXXXXXXXXXXXXXXX 4`

```
void f(int a, char* str) {
  char ch[8];
  int var;
  // Copie dangeureuse !
  strcpy(ch, str);
}
void main(int argc, char** argv) {
  f(4, argv[1]);
}
```



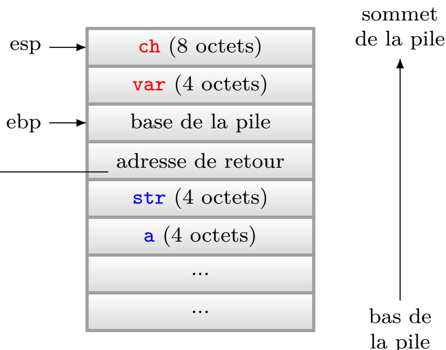
Exemple de fonction vulnérable 4/4

Si la taille de `str` est supérieure à 16, problème

écrasement de `ch`, de `var`, du pointeur de base de la pile et de l'adresse de retour !

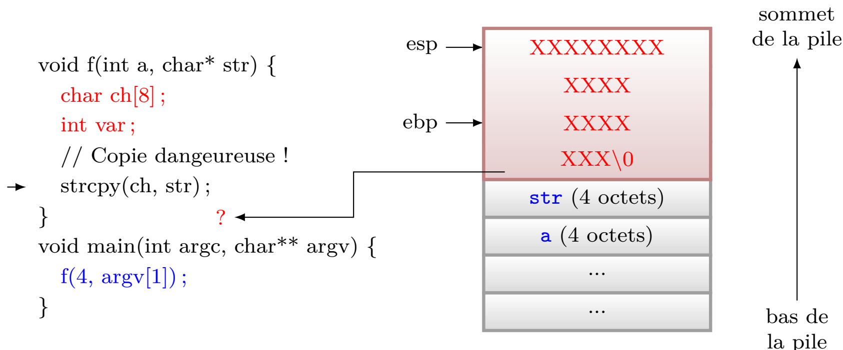
Commande : `./program XXXXXXXXXXXXXXXXXXXX` 4

```
→ void f(int a, char* str) {
    char ch[8];
    int var;
    // Copie dangeureuse !
    strcpy(ch, str);
  }
  void main(int argc, char** argv) {
    f(4, argv[1]);
  }
```



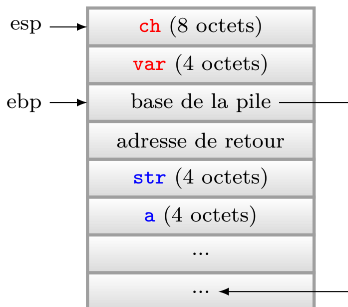
Exemple de fonction vulnérable 4/4

Si la taille de `str` est supérieure à 16, problème
écrasement de `ch`, de `var`, du pointeur de base de la pile et de l'adresse de retour !
Commande : `./program XXXXXXXXXXXXXXXXXXXX` 4



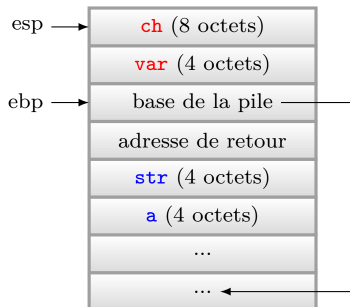
L'exploitation — Principe

- Principe : écraser l'adresse de retour de façon à la remplacer par l'adresse d'une zone mémoire que l'attaquant “maîtrise” (dans laquelle il peut injecter du code) ⇒ celle du début de la zone écrasée !
- Dans cette zone, on a en fait une copie de `str`, le paramètre fourni à la fonction `f` et qui provient de l'utilisateur (donc de l'attaquant)
- L'art consiste donc à fabriquer `str` de la façon suivante :
  - Le code arbitraire à exécuter au début du buffer
  - L'adresse de retour souhaitée à la fin du buffer



L'exploitation — Difficultés

- Il y a deux principales difficultés :
  - Identifier la position dans la pile de l'adresse de retour pour pouvoir l'écraser
  - Identifier l'adresse dans la pile de la zone mémoire que l'on veut écraser
- Si l'on possède le code source du programme que l'on veut exploiter, c'est beaucoup plus facile !
- Sinon, utiliser quelques techniques pour faciliter la recherche

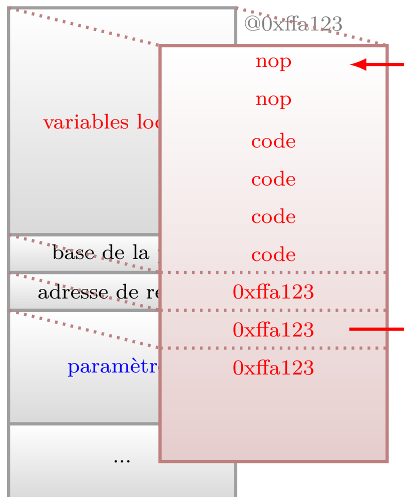


Que vaut cette adresse ?

Où est stockée l'adresse de retour ?

Quelques astuces

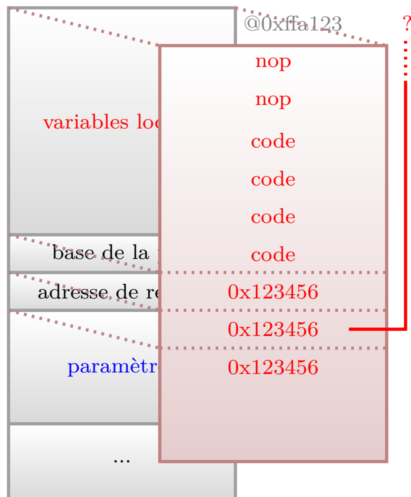
- Utiliser de nombreuses instructions `nop` au début de `str` autorise une marge d'erreur dans la recherche de l'adresse dans la pile du début de la zone que l'on veut écraser
- Ecrire de nombreuses fois, à la fin de `str`, l'adresse de retour estimée, permet d'augmenter les chances d'écraser l'adresse de retour
- La copie de `str` peut empiéter sur elle-même !



Shellcode adapté !

Quelques astuces

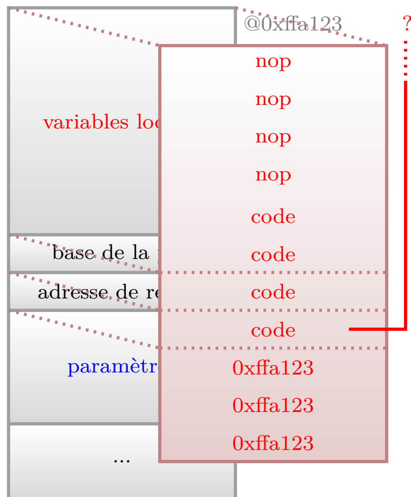
- Utiliser de nombreuses instructions `nop` au début de `str` autorise une marge d'erreur dans la recherche de l'adresse dans la pile du début de la zone que l'on veut écraser
- Ecrire de nombreuses fois, à la fin de `str`, l'adresse de retour estimée, permet d'augmenter les chances d'écraser l'adresse de retour
- La copie de `str` peut empiéter sur elle-même !



Shellcode inadapté !
Adresse de `str` *mal estimée*

Quelques astuces

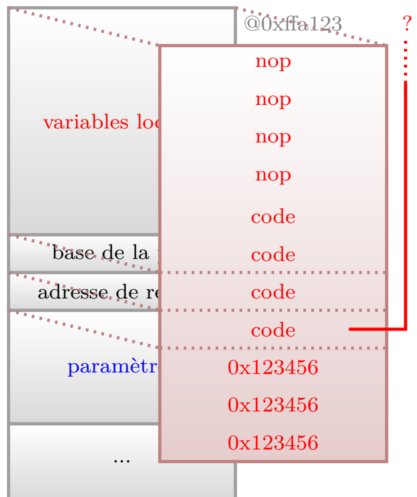
- Utiliser de nombreuses instructions `nop` au début de `str` autorise une marge d'erreur dans la recherche de l'adresse dans la pile du début de la zone que l'on veut écraser
- Ecrire de nombreuses fois, à la fin de `str`, l'adresse de retour estimée, permet d'augmenter les chances d'écraser l'adresse de retour
- La copie de `str` peut empiéter sur elle-même !



Shellcode inadapté !
Adresse de retour mal localisée

Quelques astuces

- Utiliser de nombreuses instructions `nop` au début de `str` autorise une marge d'erreur dans la recherche de l'adresse dans la pile du début de la zone que l'on veut écraser
- Ecrire de nombreuses fois, à la fin de `str`, l'adresse de retour estimée, permet d'augmenter les chances d'écraser l'adresse de retour
- La copie de `str` peut empiéter sur elle-même !



Shellcode inadapté !

Adresse de retour mal localisée et adresse de `str` *mal estimée*

Introduction

Rappels sur les appels de fonction

Les attaques de type “buffer overflow”

La technique

Les `shellcode`

Les défenses actuelles

Détection automatique

Résumé des problèmes à résoudre

- Pour réaliser le débordement, il faut connaître :
  - l'adresse de retour empilée lors de l'appel de `f`
  - l'adresse de `str` (l'adresse de retour va être écrasée par cette valeur)
- Ensuite, il faut fabriquer `argv[1]` de telle façon à ce qu'il contienne le shellcode (précédé de beaucoup de `nop`) suivi de l'adresse de `str` (que l'on écrit beaucoup de fois)

A la recherche de l'adresse de retour (1/5)

- Soit le programme `program.c` suivant :
- L'adresse de retour de la fonction `f` se situe quelque part dans la pile après les variables `buffer1` et `buffer2`
- Pas évident de savoir exactement où elle est en fonction du compilateur et des options

```
void f(int a, int b, int c) {
  char buffer1[4]="aaaa";
  char buffer2[8]="bbbbbbbb";
}
int main() {
  int x = 0;
  f(1, 2, 3);
  x = 1;
  return x;
}
```

- Solution : désassembler le `main` en utilisant, par exemple, les commandes `nm`, `objdump` ou le debugger `gdb`

A la recherche de l'adresse de retour (2/5)

- Utilisation de `nm` et `objdump` 5
- L'adresse de retour est juste après le `call`, soit `0x8048cf6`
- Cette adresse est donc forcément empilée lors de l'appel à la fonction `f`

```
# nm program | grep main
...
08048ccd T main
...
# objdump program --start-address 0x08048ccd --stop-address 0x08048d05 -D | more
...
08048ccd <main>:
 8048ccd:       55                          push   %ebp
 8048cce:       89 e5                       mov    %esp,%ebp
 8048cd0:       83 ec 10                    sub    $0x10,%esp
 8048cd3:       c7 45 fc 00 00 00 00        movl   $0x0,-0x4(%ebp)
 8048cda:       c7 44 24 08 03 00 00        movl   $0x3,0x8(%esp)
 8048ce1:       00
 8048ce2:       c7 44 24 04 02 00 00        movl   $0x2,0x4(%esp)
 8048ce9:       00
 8048cea:       c7 04 24 01 00 00 00        movl   $0x1,(%esp)
 8048cf1:       e8 ba ff ff ff              call   8048cb0 <f>
 8048cf6:       c7 45 fc 01 00 00 00        movl   $0x1,-0x4(%ebp)
 8048cfd:       8b 45 fc                    mov    -0x4(%ebp),%eax
 8048d00:       c9                          leave
 8048d01:       c3                          ret
...
```

A la recherche de l'adresse de retour (3/5)

- Autre possibilité avec `gdb` : l'adresse de retour est dans `saved eip`

```
# gdb -x sc.gdb ./program
(gdb) list 1
...
4         void f(int a, int b, int c) {
5           char buffer1[4] = "aaaa";
6           char buffer2[8] = "bbbbbbbb";
7         }
8
9         int main() {
10           int x = 0;
(gdb) b 7
Breakpoint 1 at 0x8048ccb: file program.c, line 7.
(gdb) run
Breakpoint 1, f (a=1, b=2, c=3) at program.c:7
7         }
(gdb) info frame
Stack level 0, frame at 0xbffff798:
 eip = 0x8048ccb in f (program.c:7); saved eip 0x8048cf6
 called by frame at 0xbffff7b0
 source language c.
 Arglist at 0xbffff790, args: a=1, b=2, c=3
 Locals at 0xbffff790, Previous frame's sp is 0xbffff798
 Saved registers:
  ebp at 0xbffff790, eip at 0xbffff794
```

A la recherche de l'adresse de retour (4/5)

- Il suffit de visualiser le contenu de la pile avec un gdb
- L'adresse de retour est donc en 0xbffff794, soit 8 octets après buffer1

```
# gdb a.out
(gdb) list 1
....
4    void f(int a, int b, int c) {
5      char buffer1[4] = "aaaa";
6      char buffer2[8] = "bbbbbbbb";
7    }
....
(gdb) b 7
Breakpoint 1 at 0x8048ccb: file program.c, line 7.
(gdb) run
...
Breakpoint 1, f (a=1, b=2, c=3) at program.c:7
7    }
(gdb) x /20x buffer1
0xbffff78c:     0x61616161      0xbffff7a8      0x08048cf6      0x00000001
...
(gdb) set {int} 0xbffff794 += 8
(gdb) p &buffer1
$6 = (char (*)[4]) 0xbffff794
(gdb) set *((int*) 0xbffff794) += 8
```

! Ces valeurs dépendent des versions du système, de gcc et des paramètres de compilation

A la recherche de l'adresse de retour (5/5)

- Comment faire sans le code source ?
  - Dans ce cas, il faut tenter en aveugle, sachant qu'en général, un programme habituel empile au maxium quelques centaines d'octets
  - Tenter donc de créer des chaînes de différentes tailles (itérativement de 100 à 1000 par exemple et écrire l'adresse de retour souhaitée autant de fois que possible à la fin de la chaîne)
- A présent, modification du retour avec `gdb`

A la recherche de la zone à écraser (1/2)

- Soit le programme vulnérable suivant :

```
void copie(char* ch) {
  char str[512];
  strcpy(str, ch);
}
int main(int argc, char** argv) {
  copie(argv[1]);
  return 0;
}
```

- La recherche de l'adresse de `str` peut s'avérer trés compliquée sur des noyaux récents
- Si on suppose que la pile est toujours stockée à la même adresse dans un processus en cours d'exécution, on peut trouver l'adresse de `str` par essais successifs :
  - On détermine l'adresse de base de la pile
  - On calcul l'adresse de `str` en soustrayant au moins la taille de str plus un offset que l'on fait varier en effectuant plusieurs tests.

A la recherche de la zone à écraser (2/2)

- Si on a la main sur la machine où on veut exploiter ce programme et si ASLR est desactivé (cf. slides suivants), on peut utiliser la fonction suivante qui nous permet de connaître l'adresse de base du pointeur de pile. Reste à tester avec différents offsets. 6

```
unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}
long l=get_sp();
```

Introduction

Rappels sur les appels de fonction

Les attaques de type “buffer overflow”

La technique

Les `shellcode`

Les défenses actuelles

Détection automatique

Propriétés des `shellcode` 1/4

- Taille réduite (quelques octets), pour éviter les erreurs de segmentation
- Absence de l'octet `0` qui est le marqueur de fin de chaîne de caractères
- Indépendant de la position en mémoire
- Indépendant de la vulnérabilité exploitée
- Dépendant de l'architecture matérielle (code assembleur)
- Certains `shellcode` chiffrent leur contenu (fonction `xor`)

Propriétés des shellcode 2/4

- La construction de ces programmes suit les étapes suivantes : 12
  - Création d'un programme en `C` réalisant ce que le `shellcode` va devoir faire
    Par exemple :
    ```
    void main(void) {
      printf("coucou\n");
    }
    ```
  - Les fonctions utilisées sont celles de la librarie `C`. Elles doivent être traduites en appels systèmes
    Commande : `strace ./program`
  - Modification du programme sans passer par la librairie `C`
  - Récupération du numéro de l'appel système
    Commande : `more /usr/include/asm-i386/unistd.h | grep write`
  - Paramètres des appels systèmes
    `eax` → numéro de l'appel système
    `ebx` → premier argument : 1
    `ecx` → deuxième argument : adresse de la chaîne à afficher
    `edx` → troisième argument : longueur de la chaîne
  - **Comment connaître l'adresse de la chaîne ?**

Propriétés des shellcode 3/4

- **Comment connaître l'adresse de la chaîne ?**
  - On profite du fait que les données contenues dans le `shellcode` se situent au même endroit que les instructions de ce `shellcode`
  - Lors du `call Y`, l'adresse de retour est empilée. Elle correspond à l'adresse de la chaîne. L'instruction `pop ecx` dépile cette adresse et la stocke dans `ecx`

Propriétés des shellcode 4/4

- shellcode chiffré avec la fonction xor

	jmp X
Y' :	xorl %eax,%eax pop esi

Z' :	cmp %al,(%esi)
	je @chaîne
	xorl $12,(%esi)
	add $1,%esi
	jmp Z

X' :	call Y
@chaîne :	*charge utile chiffrée*

Introduction

Rappels sur les appels de fonction

Les attaques de type “buffer overflow”

La technique

Les `shellcode`

Les défenses actuelles

Détection automatique

Différents mécanismes de défense

- Il est désormais de plus en plus difficile d'exploiter un buffer overflow dans la pile sur les noyaux récents
  si les protections sont bien activées !!
- Renforce le côté *challenge* pour les *hackers*
- Quelques mécanismes de protections :
  - Systèmes de détection d'intrusion
  - Randomization espace d'adressage
  - Pile non executable
  - Réordonnancement des variables
  - Réordonnancement des pointeurs
  - Canaries

Les systèmes de détection d’intusions

- *Intrusion Detection System – IDS* (*Snort*)
- Certaines vulnérabilités sont exploitées à distance
- La *payload* de l’attaquant est envoyée à travers le réseau
- Les *IDS* peuvent donc analyser les messages à la recherche de ces *payload*
- Certains systèmes peuvent même bloquer ces attaques
  Intrusion Prevention System – IPS
- Exemple de signature :

```
alert ip $EXTERNAL_NET any -> $HOME_NET any
  (
    msg:"SHELLCODE Linux shellcode";
    content:"|90 90 90 E8 C0 FF FF FF|/bin/sh";
    fast_pattern:only;
    reference:arachnids,343;
    classtype:shellcode-detect;
    sid:652;
    rev:11;
  )
```

La randomization de l'espace d'adresse (ASLR)

- L'adresse de la pile dans l'espace d'adressage d'un processus change à chaque exécution (*ASLR* : *Adress Space Layout Randomization*)
- Activation (en `root`) : `echo 1 > /proc/sys/kernel/randomize_va_space`
- Désactivation (en `root`) : `echo 0 > /proc/sys/kernel/randomize_va_space`
- Soit le programme suivant :

```
$ cat program.c
void copie(char* ch) {
  char str[512];
  printf("Adresse dans la pile de str %x\n", str);
  strcpy(str, ch);
}
void main(int argc, char** argv) {
  copie(argv[1]);
}
```

- Si *ASLR* est activé

```
# ./program
Adresse dans la pile de str bf9ee3d4
# ./program
Adresse dans la pile de str bfc47144
# ./program
Adresse dans la pile de str bff6b904
```

- Si ASLR n'est pas activé

```
# ./program
Adresse dans la pile de str bffff3a4
# ./program
Adresse dans la pile de str bffff3a4
# ./program
Adresse dans la pile de str bffff3a4
```

La pile non exécutable

- Avec le mécanisme de pagination des processeurs activé, les descripteurs de pages mémoire contiennent plusieurs informations sur les pages de la mémoire physique
- En particulier, un champs indique si le contenu de la page est exécutable ou non
- Lors de l'exécution d'un bout de code, le processeur vérifie que la page contenant le code est exécutable en analysant le descripteur correspondant
- Désormais, sur les noyaux Linux récents, la pile est non exécutable et il est donc impossible de réaliser un buffer overflow dans la pile simplement
- Option de compilation pour rendre la pile exécutable : `gcc -z execstack`

Réordonnancement des variables (1/4)

- Soit le programme suivant : 13

```
void check_password(char* pass) {
  int good = 2;
  char password[12];
  strcpy(password,pass);
  if (!strcmp(password, "passw0rd")) {
    good = 1;
  }
  if (good == 1) {
    printf("Password OK\n");
  } else {
    printf("Password KO\n");
  }
}

int main(int argc, char** argv) {
  if (argc != 2) {
    return(1);
  }
  check_password(argv[1]);
  return(0);
}
```

Réordonnancement des variables (2/4)

- good est après password, on peut donc l'écraser

```
$ gcc -g -fno-stack-protector test.c
$ gdb a.out
(gdb) b 6                (<- point d'arret a la ligne 6)
Breakpoint 1 at 0x80483be: file test.c, line 6.
(gdb) run AAAAAA
...
(gdb) n
...
gdb) x /20x password
0xbffff528:     0x41414141      0xb7ee0041      0xb7f90b19      0x00000002
0xbffff538:     0xbffff558      0x0804846c      0xbffff75a      0x08049650
0xbffff548:     0xbffff568      0xbffff570      0xb7ff1df0      0xbffff570
0xbffff558:     0xbffff5c8      0xb7e98450      0xb7ffece0      0x08048490
(gdb) p &good
$2 = (int *) 0xbffff534
```

Réordonnancement des variables (3/4)

- good a été écrasé (valeur 1)

```
$ gdb a.out
...
(gdb) b 6
...
(gdb) run `perl -e 'printf "A" x 12 . "\x01"'`
...
(gdb) n
...
(gdb) x /20x password
0xbffff528:     0x41414141      0x41414141      0x41414141      0x00000001
0xbffff538:     0xbffff558      0x0804846c      0xbffff752      0x08049650
0xbffff548:     0xbffff568      0xbffff570      0xb7ff1df0      0xbffff570
0xbffff558:     0xbffff5c8      0xb7e98450      0xb7ffece0      0x08048490
0xbffff568:     0xbffff5c8      0xb7e98450      0x00000002      0xbffff5f4
```

```
bash$ ./a.out `perl -e 'printf "A" x 12 . "\x01"'`
Password OK
```

Réordonnancement des variables (4/4)

- Avec utilisation de la protection de la pile, il n'est pas possible d'écraser good

```
bash$ gcc -g test.c
bash$ gdb a.out
(gdb) b 6
Breakpoint 1 at 0x8048452: file test.c, line 6.
(gdb) run AAAAAA
Starting program: a.out AAAAAA
Breakpoint 1, check_password (pass=0xbffff759 "AAAAAA") at test.c:6
6          strcpy(password,pass);
(gdb) n
7          if (!strcmp(password,"passw0rd"))
(gdb) x /20x password
0xbffff528:     0x41414141      0xb7004141      0xb7f90b19      0xfcf45000
0xbffff538:     0xbffff558      0x080484ee      0xbffff759      0x080496d0
0xbffff548:     0xbffff568      0xbffff570      0xb7ff1df0      0xbffff570
0xbffff558:     0xbffff5c8      0xb7e98450      0xb7ffece0      0x08048510
0xbffff568:     0xbffff5c8      0xb7e98450      0x00000002      0xbffff5f4
(gdb) p &good
$1 = (int *) 0xbffff524
```

Réordonnancement des pointeurs (1/4)

- Soit la fonction :

```
#include <string.h>

void function(char* one, char* two) {
  char buff[8];
  strcpy(buff,one);
}

int main(int argc, char** argv) {
  function(argv[1],"BBBBBBB");
  return 0;
}
```

Réordonnancement des pointeurs (2/4)

- Sans protection de la pile, buff est donc avant les variables locales et peut donc les écraser

```
bash$ gcc -fno-stack-protector -g test2.c
bash$ gdb a.out
(gdb) b 5
Breakpoint 1 at 0x804837a: file test2.c, line 5.
(gdb) run AAAAA
Starting program: a.out AAAAAA
Breakpoint 1, function (one=0xbffff759 "AAAAAA", two=0x8048490 "BBBBBBB")
    at test2.c:6
6         strcpy(buff,one);
(gdb) n
7       }
(gdb) x /20x buff
0xbffff530:     0x41414141      0x08004141      0xbffff558      0x080483b7
0xbffff540:     0xbffff759      0x08048490      0xbffff568      0x080483f9
0xbffff550:     0xb7ff1df0      0xbffff570      0xbffff5c8      0xb7e98450
0xbffff560:     0xb7ffece0      0x080483e0      0xbffff5c8      0xb7e98450
0xbffff570:     0x00000002      0xbffff5f4      0xbffff600      0xb7fe2b38
(gdb) p &one
$1 = (char **) 0xbffff540
(gdb) p &two
```

Réordonnancement des pointeurs (3/4)

- `two` est écrasé

```
bash$ gdb a.out
(gdb) b 5
Breakpoint 1 at 0x804837a: file test2.c, line 5.
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: a.out AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Breakpoint 1, function (one=0xbffff73a 'A' <repeats 37 times>,
    two=0x8048490 "BBBBBBB") at test2.c:6
6          strcpy(buff,one);
(gdb) n
7       }
(gdb) p &two
$1 = (char **) 0xbffff524
(gdb) x /20x buff
0xbffff510:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff520:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff530:     0x41414141      0xbfff0041      0xbffff5a8      0xb7e98450
0xbffff540:     0xb7ffece0      0x080483e0      0xbffff5a8      0xb7e98450
0xbffff550:     0x00000002      0xbffff5d4      0xbffff5e0      0xb7fe2b38
```

Réordonnancement des pointeurs (4/4)

- Avec protection de la pile, **buff** est donc après les variables locales et ne peut donc pas les écraser

```
bash$ gcc -g test2.c
bash$ gdb a.out
(gdb) b 5
Breakpoint 1 at 0x80483e1: file test2.c, line 5.
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: a.out AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Breakpoint 1, function (one=0xbffff730 'A' <repeats 47 times>,
    two=0x8048500 "BBBBBBB") at test2.c:6
6          strcpy(buff,one);
(gdb) n
7       }
(gdb) x /20x buff
0xbffff4fc:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff50c:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff51c:     0x41414141      0x41414141      0x41414141      0x00414141
0xbffff52c:     0xb7e98450      0xb7ffece0      0x08048450      0xbffff598
0xbffff53c:     0xb7e98450      0x00000002      0xbffff5c4      0xbffff5d0
(gdb) p &one
$1 = (char **) 0xbffff4f4
(gdb) p &two
$2 = (char **) 0xbffff4f0
```

Les canaries (1/6)

- *Canary* sous *Unix* et *Security Cookie* sous *Windows*
- Au début de la fonction appelée, une valeur est insérée entre les variables locales et la sauvegarde du registre `ebp`
- Un débordement d'une variable locale va écraser cette valeur avant d'écraser le pointeur de retour
- Avant le retour de la fonction, la valeur du *canary* est vérifiée
  Si il a été modifié, le programme est arrêté brutalement
- Le *canary* protège l'adresse de retour

Les canaries (2/6)

- Différents types de *canaries*
- *Null Canary*
  - *Canary* dont la valeur est `0x00000000`
- *Terminator Canary*
  - *Canary* dont la valeur contient l'octet `0x00`
  - Exemple de valeur : `0x000aff0d`
  - La copie de la chaîne contenant l'exploit, avec la fonction `strcpy` par exemple, sera stoppée au niveau de l'octet `0x00`
  - La suite du canary ne sera pas copiée (`aff0d`)
  - Même si l'attaquant prédit avec succès la valeur du *canary*, il ne pourra pas modifier l'adresse de retour
- *Random Canary*
  - *Canary* dont la valeur est basé sur le générateur aléatoire `urandom`
  - Les valeurs obtenues avec `urandom` dont difficilement prédictibles

Les canaries (3/6)

- Exemples de comportements avec le programme suivant : 8

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

int fun(char* arg) {
  int i = 12;
  char p[10] = "BBBBBBBBB";
  strcpy(p, arg);
  printf("Canary = 0x");
  for (i = 13; i > 9; i--) {
    printf("%02x", (unsigned char)*(p + i));
  }
  printf("\n");
}

int main(int argc, char** argv) {
  if (argc > 1) {
    fun(argv[1]);
  }
  return 0;
}
```

Les canaries (4/6)

- Le canary est en 0xbffff534 ; il est situé avant la sauvegarde de `ebp`, en `0xbffff538` et, dans cet exemple, sa valeur est `0xfcf45000`

```
bash$ gdb a.out
(gdb) b 7
Breakpoint 1 at 0x8048452: file canary.c, line 7.
(gdb) run AAAAAA
Starting program: a.out AAAAAA
Breakpoint 1, fun (arg=0xbffff759 "AAAAAA") at canary.c:7
7         char p[10]="BBBBBBBBB";
(gdb) n
9         strcpy(p,arg);
(gdb) x /20x &i
0xbffff524:    0x0000000c    0x4242f738    0x42424242    0x00424242
0xbffff534:    0xfcf45000    0xbffff558    0x0804850c    0xbffff759
0xbffff544:    0x080496e8    0xbffff568    0x08048549    0xbffff570
0xbffff554:    0xbffff570    0xbffff5c8    0xb7e98450    0xb7ffece0
0xbffff564:    0x08048530    0xbffff5c8    0xb7e98450    0x00000002
(gdb) info frame
Stack level 0, frame at 0xbffff540:
 ...
 Saved registers:
 ebp at 0xbffff538, eip at 0xbffff53c
```

Les canaries (5/6)

- Exécution sans écraser le *canary*

```
$ ./a.out AAAAAA
Canary = 0xfcf45000
```

- Exécution en écrasant le *canary*

```
$ ./a.out AAAAAAAAAAAAAAAA
Canary = 0x41414141
*** stack smashing detected ***: ./a.out terminated
======= Backtrace: =========
/lib/tls/i686/cmov/libc.so.6(__fortify_fail+0x48)[0xb7f6f138]
/lib/tls/i686/cmov/libc.so.6(__fortify_fail+0x0)[0xb7f6f0f0]
...
```

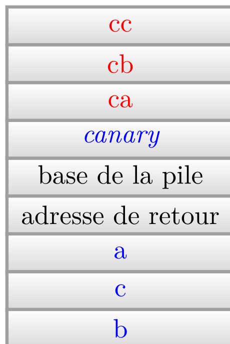
- Désactivation des *canary* : `gcc -fno-stack-protector`

Les canaries (6/6) – *canary à valeur fixe*

- Cas particulier des *Terminator Canary* et *Null Canary*
- Valeur fixe quelque soit l'exécution
- Soit le programme vulnérable suivant, avec l'état de la pile correspondant à l'invocation de la fonction `test`

```
void test(char* a, char* b, char* c) {
  char[4] ca;
  char[4] cb;
  char[4] cc;
  strcpy(ca, a);
  strcpy(cb, b);
  strcpy(cc, c);
}

void main(int argc, char** argv) {
  test(argv[1], argv[2], argv[3]);
}
```



- Exécution du programme :

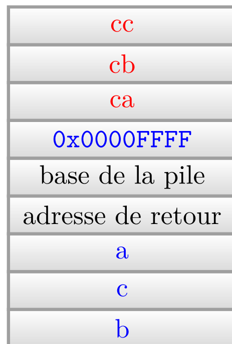
```
program `echo -e "AAAA\xFF\xFF11AAAAAAAA"` BBBBBBBBFFB CCCCCCCCCCCFF
```

Les canaries (6/6) – *canary à valeur fixe*

- Cas particulier des *Terminator Canary* et *Null Canary*
- Valeur fixe quelque soit l'exécution
- Soit le programme vulnérable suivant, avec l'état de la pile correspondant à l'invocation de la fonction `test`

```
void test(char* a, char* b, char* c) {
  char[4] ca;
  char[4] cb;
  char[4] cc;
  strcpy(ca, a);
  strcpy(cb, b);
  strcpy(cc, c);
}

void main(int argc, char** argv) {
  test(argv[1], argv[2], argv[3]);
}
```



- Exécution du programme :

```
program `echo -e "AAAA\xFF\xFF11AAAAAAAA"` BBBBBBBBBFFB CCCCCCCCCCCCFF
```

Les canaries (6/6) – *canary à valeur fixe*

- Cas particulier des *Terminator Canary* et *Null Canary*
- Valeur fixe quelque soit l'exécution
- Soit le programme vulnérable suivant, avec l'état de la pile correspondant à l'invocation de la fonction `test`

```
void test(char* a, char* b, char* c) {
  char[4] ca;
  char[4] cb;
  char[4] cc;
  strcpy(ca, a);
  strcpy(cb, b);
  strcpy(cc, c);
}

void main(int argc, char** argv) {
  test(argv[1], argv[2], argv[3]);
}
```

- Exécution du programme :

```
program `echo -e "AAAA\xFF\xFF11AAAAAAAA"` BBBBBBBBFFB CCCCCCCCCCCFF
```

Les canaries (6/6) – *canary à valeur fixe*

- Cas particulier des *Terminator Canary* et *Null Canary*
- Valeur fixe quelque soit l'exécution
- Soit le programme vulnérable suivant, avec l'état de la pile correspondant à l'invocation de la fonction `test`

```
void test(char* a, char* b, char* c) {
  char[4] ca;
  char[4] cb;
  char[4] cc;
  strcpy(ca, a);
  strcpy(cb, b);
  strcpy(cc, c);
}

void main(int argc, char** argv) {
  test(argv[1], argv[2], argv[3]);
}
```

- Exécution du programme :

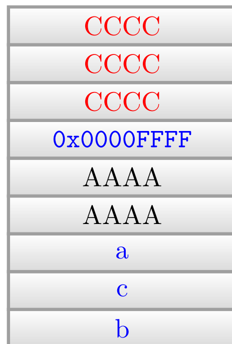
```
program `echo -e "AAAA\xFF\xFF11AAAAAAAA"` BBBBBBBBFFB CCCCCCCCCCCFF
```

Les canaries (6/6) – *canary à valeur fixe*

- Cas particulier des *Terminator Canary* et *Null Canary*
- Valeur fixe quelque soit l'exécution
- Soit le programme vulnérable suivant, avec l'état de la pile correspondant à l'invocation de la fonction `test`

```
void test(char* a, char* b, char* c) {
  char[4] ca;
  char[4] cb;
  char[4] cc;
  strcpy(ca, a);
  strcpy(cb, b);
  strcpy(cc, c);
}

void main(int argc, char** argv) {
  test(argv[1], argv[2], argv[3]);
}
```



- Exécution du programme :

```
program `echo -e "AAAA\xFF\xFF11AAAAAAAA"` BBBBBBBBBFFB CCCCCCCCCCCCFF
```

Introduction

Rappels sur les appels de fonction

Les attaques de type “buffer overflow”

La technique

Les `shellcode`

Les défenses actuelles

Détection automatique

La détection automatique des *buffer overflow*

- L'analyse automatique repose sur des outils permettant de vérifier si certains propriétés du programme sont invalidées – nous nous intéresserons aux propriétés de sécurité et en particulier à la détection des *buffer overflow*
- Les outils et approches présentés dans la suite sont considérés de notre point de vue (détection des *buffer overflow*)
- Deux approches : par analyse dynamique et par analyse statique
- Evaluation à partir des taux de faux positifs et de faux négatifs

Détection des *bof* par analyse dynamique

- Instrumentation du code pour marquer les tampons du programme
- Exécution effective du programme à tester, sur une véritable machine, avec des jeux de test variés
- Exemple – “compilateur” `stobo` [?]
  - Injection de code autour des déclarations de tableaux
  - Remplacement des fonctions de manipulation des tableaux
  - Vérification du comportement à l’exécution
  - \+ Taux de faux positifs faible – Pénalité à l’exécution

```
void f(char* a) {
  char b[100];
  char* p1;
  char* p2;
  strcpy(b, a);
  p1 = malloc(50);
  strcpy(p1, b);
  p2 = p1 + 10;
  strcpy(p2, b);
}
```

```
void f(char* a) {
  char b[100];
  __STOBO_first_stack_buf(b, sizeof(b));
  char* p1;
  char* p2;
  __STOBO_strcpy(b, a);
  p1 = __STOBO_const_mem_malloc(50);
  __STOBO_strcpy(p1, b);
  p2 = p1 + 10;
  __STOBO_strcpy(p2, b);
}
```

Détection des *bof* par analyse statique

- Les programmes ne sont pas exécutés mais ils sont analysés par des outils
- Différentes approches : *Data flow analysis*, Interprétation abstraite, Analyse symbolique, etc.
- Elles ne doivent pas fournir de résultats erronés ≠ elles peuvent fournir un résultat pessimiste
- Complexité des programmes ⇒ besoin de travailler sur une version simplifiée des programmes
- Certaines approches sont guidées par des annotations dans le code source

Exemple d'analyse statique

algorithme Gen/Kill

- Création du graphe de contrôle des flux à partir du code source de l'application

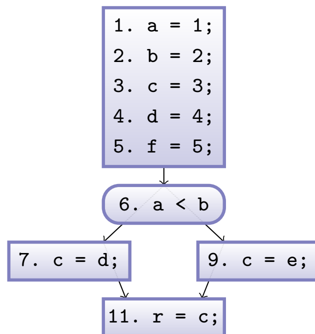
```
1.  a = 1;
2.  b = 2;
3.  c = 3;
4.  d = 4;
5.  f = 5;
6.  if (a < b) {
7.    c = d;
8.  } else {
9.    c = e;
10. }
11. r = c;
```

Exemple d'analyse statique

algorithme Gen/Kill

- Création du graphe de contrôle des flux à partir du code source de l'application

```
1.  a = 1;
2.  b = 2;
3.  c = 3;
4.  d = 4;
5.  f = 5;
6.  if (a < b) {
7.    c = d;
8.  } else {
9.    c = e;
10. }
11. r = c;
```

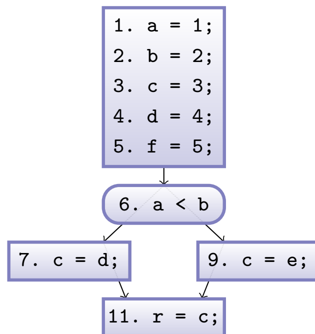


Exemple d'analyse statique

algorithme Gen/Kill

- Parcourt du graphe pour enrichir une table de faits – une variable utilisée en écriture (resp. lecture) est retirée de la colonne *Gen* (resp. *Kill*) et stockée dans la colonne *Kill* (resp. *Gen*)

l	$kill_l$	gen_l
1	$\{a\}$	$\{\}$
2	$\{b\}$	$\{\}$
3	$\{c\}$	$\{\}$
4	$\{d\}$	$\{\}$
5	$\{f\}$	$\{\}$
6	$\{\}$	$\{a, b\}$
7	$\{c\}$	$\{d\}$
9	$\{c\}$	$\{e\}$
11	$\{r\}$	$\{c\}$



Exemple d'analyse statique

algorithme Gen/Kill

- Mise sous forme d'équations, avec identification du transfert d'information (transfert *en arrière* utilisé dans cette exemple)

l	$kill_l$	gen_l
1	$\{a\}$	$\{\}$
2	$\{b\}$	$\{\}$
3	$\{c\}$	$\{\}$
4	$\{d\}$	$\{\}$
5	$\{f\}$	$\{\}$
6	$\{\}$	$\{a, b\}$
7	$\{c\}$	$\{d\}$
9	$\{c\}$	$\{e\}$
11	$\{r\}$	$\{c\}$

i_i	o_i
$i_1 = o_1 \backslash \{a\}$	$o_1 = i_2$
$i_2 = o_2 \backslash \{b\}$	$o_2 = i_3$
$i_3 = o_3 \backslash \{c\}$	$o_3 = i_4$
$i_4 = o_4 \backslash \{d\}$	$o_4 = i_5$
$i_5 = o_5 \backslash \{f\}$	$o_5 = i_6$
$i_6 = o_6 \cup \{a, b\}$	$o_6 = i_7 \cup i_9$
$i_7 = (o_7 \backslash \{c\}) \cup \{d\}$	$o_7 = i_{11}$
$i_9 = (o_9 \backslash \{c\}) \cup \{e\}$	$o_9 = i_{11}$
$i_{11} = (o_{11} \backslash \{r\}) \cup \{c\}$	$o_{11} = \{\}$

Exemple d'analyse statique

algorithme Gen/Kill

- Résolution par remplacements successifs
- La variable `f` n'est pas utilisée
- La variable `e` est utilisée en lecture avant d'être utilisée en écriture (elle appartient à i_1)

l	$kill_l$	gen_l
1	$\{a\}$	$\{\}$
2	$\{b\}$	$\{\}$
3	$\{c\}$	$\{\}$
4	$\{d\}$	$\{\}$
5	$\{f\}$	$\{\}$
6	$\{\}$	$\{a, b\}$
7	$\{c\}$	$\{d\}$
9	$\{c\}$	$\{e\}$
11	$\{r\}$	$\{c\}$

i_i	o_i
$i_1 = o_1 \backslash \{a\}$	$o_1 = i_2$
$i_2 = o_2 \backslash \{b\}$	$o_2 = i_3$
$i_3 = o_3 \backslash \{c\}$	$o_3 = i_4$
$i_4 = o_4 \backslash \{d\}$	$o_4 = i_5$
$i_5 = o_5 \backslash \{f\}$	$o_5 = i_6$
$i_6 = o_6 \cup \{a, b\}$	$o_6 = i_7 \cup i_9$
$i_7 = (o_7 \backslash \{c\}) \cup \{d\}$	$o_7 = i_{11}$
$i_9 = (o_9 \backslash \{c\}) \cup \{e\}$	$o_9 = i_{11}$
$i_{11} = \{c\}$	$o_{11} = \{\}$

Exemple d'analyse statique

algorithme Gen/Kill

- Résolution par remplacements successifs
- La variable `f` n'est pas utilisée
- La variable `e` est utilisée en lecture avant d'être utilisée en écriture (elle appartient à i_1)

l	$kill_l$	gen_l
1	$\{a\}$	$\{\}$
2	$\{b\}$	$\{\}$
3	$\{c\}$	$\{\}$
4	$\{d\}$	$\{\}$
5	$\{f\}$	$\{\}$
6	$\{\}$	$\{a, b\}$
7	$\{c\}$	$\{d\}$
9	$\{c\}$	$\{e\}$
11	$\{r\}$	$\{c\}$

i_i	o_i
$i_1 = o_1 \backslash \{a\}$	$o_1 = i_2$
$i_2 = o_2 \backslash \{b\}$	$o_2 = i_3$
$i_3 = o_3 \backslash \{c\}$	$o_3 = i_4$
$i_4 = o_4 \backslash \{d\}$	$o_4 = i_5$
$i_5 = o_5 \backslash \{f\}$	$o_5 = i_6$
$i_6 = o_6 \cup \{a, b\}$	$o_6 = i_7 \cup i_9$
$i_7 = (o_7 \backslash \{c\}) \cup \{d\}$	$o_7 = i_{11}$
$i_9 = (o_9 \backslash \{c\}) \cup \{e\}$	$o_9 = \{c\}$
$i_{11} = \{c\}$	$o_{11} = \{\}$

Exemple d'analyse statique

algorithme Gen/Kill

- Résolution par remplacements successifs
- La variable `f` n'est pas utilisée
- La variable `e` est utilisée en lecture avant d'être utilisée en écriture (elle appartient à i_1)

l	$kill_l$	gen_l
1	$\{a\}$	$\{\}$
2	$\{b\}$	$\{\}$
3	$\{c\}$	$\{\}$
4	$\{d\}$	$\{\}$
5	$\{f\}$	$\{\}$
6	$\{\}$	$\{a, b\}$
7	$\{c\}$	$\{d\}$
9	$\{c\}$	$\{e\}$
11	$\{r\}$	$\{c\}$

i_i	o_i
$i_1 = o_1 \backslash \{a\}$	$o_1 = i_2$
$i_2 = o_2 \backslash \{b\}$	$o_2 = i_3$
$i_3 = o_3 \backslash \{c\}$	$o_3 = i_4$
$i_4 = o_4 \backslash \{d\}$	$o_4 = i_5$
$i_5 = o_5 \backslash \{f\}$	$o_5 = i_6$
$i_6 = o_6 \cup \{a, b\}$	$o_6 = i_7 \cup i_9$
$i_7 = (o_7 \backslash \{c\}) \cup \{d\}$	$o_7 = i_{11}$
$i_9 = \{e\}$	$o_9 = \{c\}$
$i_{11} = \{c\}$	$o_{11} = \{\}$

Exemple d'analyse statique

algorithme Gen/Kill

- Résolution par remplacements successifs
- La variable `f` n'est pas utilisée
- La variable `e` est utilisée en lecture avant d'être utilisée en écriture (elle appartient à i_1)

l	$kill_l$	gen_l
1	$\{a\}$	$\{\}$
2	$\{b\}$	$\{\}$
3	$\{c\}$	$\{\}$
4	$\{d\}$	$\{\}$
5	$\{f\}$	$\{\}$
6	$\{\}$	$\{a, b\}$
7	$\{c\}$	$\{d\}$
9	$\{c\}$	$\{e\}$
11	$\{r\}$	$\{c\}$

i_i	o_i
$i_1 = o_1 \backslash \{a\}$	$o_1 = i_2$
$i_2 = o_2 \backslash \{b\}$	$o_2 = i_3$
$i_3 = o_3 \backslash \{c\}$	$o_3 = i_4$
$i_4 = o_4 \backslash \{d\}$	$o_4 = i_5$
$i_5 = o_5 \backslash \{f\}$	$o_5 = i_6$
$i_6 = o_6 \cup \{a, b\}$	$o_6 = i_7 \cup i_9$
$i_7 = (o_7 \backslash \{c\}) \cup \{d\}$	$o_7 = \{c\}$
$i_9 = \{e\}$	$o_9 = \{c\}$
$i_{11} = \{c\}$	$o_{11} = \{\}$

Exemple d'analyse statique

algorithme Gen/Kill

- Résolution par remplacements successifs
- La variable `f` n'est pas utilisée
- La variable `e` est utilisée en lecture avant d'être utilisée en écriture (elle appartient à i_1)

l	$kill_l$	gen_l
1	$\{a\}$	$\{\}$
2	$\{b\}$	$\{\}$
3	$\{c\}$	$\{\}$
4	$\{d\}$	$\{\}$
5	$\{f\}$	$\{\}$
6	$\{\}$	$\{a, b\}$
7	$\{c\}$	$\{d\}$
9	$\{c\}$	$\{e\}$
11	$\{r\}$	$\{c\}$

i_i	o_i
$i_1 = o_1 \setminus \{a\}$	$o_1 = i_2$
$i_2 = o_2 \setminus \{b\}$	$o_2 = i_3$
$i_3 = o_3 \setminus \{c\}$	$o_3 = i_4$
$i_4 = o_4 \setminus \{d\}$	$o_4 = i_5$
$i_5 = o_5 \setminus \{f\}$	$o_5 = i_6$
$i_6 = o_6 \cup \{a, b\}$	$o_6 = i_7 \cup i_9$
$i_7 = \{d\}$	$o_7 = \{c\}$
$i_9 = \{e\}$	$o_9 = \{c\}$
$i_{11} = \{c\}$	$o_{11} = \{\}$

Exemple d'analyse statique

algorithme Gen/Kill

- Résolution par remplacements successifs
- La variable `f` n'est pas utilisée
- La variable `e` est utilisée en lecture avant d'être utilisée en écriture (elle appartient à i_1)

l	$kill_l$	gen_l
1	$\{a\}$	$\{\}$
2	$\{b\}$	$\{\}$
3	$\{c\}$	$\{\}$
4	$\{d\}$	$\{\}$
5	$\{f\}$	$\{\}$
6	$\{\}$	$\{a, b\}$
7	$\{c\}$	$\{d\}$
9	$\{c\}$	$\{e\}$
11	$\{r\}$	$\{c\}$

i_i	o_i
$i_1 = o_1 \backslash \{a\}$	$o_1 = i_2$
$i_2 = o_2 \backslash \{b\}$	$o_2 = i_3$
$i_3 = o_3 \backslash \{c\}$	$o_3 = i_4$
$i_4 = o_4 \backslash \{d\}$	$o_4 = i_5$
$i_5 = o_5 \backslash \{f\}$	$o_5 = i_6$
$i_6 = o_6 \cup \{a, b\}$	$o_6 = \{d, e\}$
$i_7 = \{d\}$	$o_7 = \{c\}$
$i_9 = \{e\}$	$o_9 = \{c\}$
$i_{11} = \{c\}$	$o_{11} = \{\}$

Exemple d'analyse statique

algorithme Gen/Kill

- Résolution par remplacements successifs
- La variable `f` n'est pas utilisée
- La variable `e` est utilisée en lecture avant d'être utilisée en écriture (elle appartient à i_1)

l	$kill_l$	gen_l
1	$\{a\}$	$\{\}$
2	$\{b\}$	$\{\}$
3	$\{c\}$	$\{\}$
4	$\{d\}$	$\{\}$
5	$\{f\}$	$\{\}$
6	$\{\}$	$\{a, b\}$
7	$\{c\}$	$\{d\}$
9	$\{c\}$	$\{e\}$
11	$\{r\}$	$\{c\}$

i_i	o_i
$i_1 = o_1 \backslash \{a\}$	$o_1 = i_2$
$i_2 = o_2 \backslash \{b\}$	$o_2 = i_3$
$i_3 = o_3 \backslash \{c\}$	$o_3 = i_4$
$i_4 = o_4 \backslash \{d\}$	$o_4 = i_5$
$i_5 = o_5 \backslash \{f\}$	$o_5 = i_6$
$i_6 = \{a, b, d, e\}$	$o_6 = \{d, e\}$
$i_7 = \{d\}$	$o_7 = \{c\}$
$i_9 = \{e\}$	$o_9 = \{c\}$
$i_{11} = \{c\}$	$o_{11} = \{\}$

Exemple d'analyse statique

algorithme Gen/Kill

- Résolution par remplacements successifs
- La variable `f` n'est pas utilisée
- La variable `e` est utilisée en lecture avant d'être utilisée en écriture (elle appartient à i_1)

l	$kill_l$	gen_l
1	$\{a\}$	$\{\}$
2	$\{b\}$	$\{\}$
3	$\{c\}$	$\{\}$
4	$\{d\}$	$\{\}$
5	$\{f\}$	$\{\}$
6	$\{\}$	$\{a, b\}$
7	$\{c\}$	$\{d\}$
9	$\{c\}$	$\{e\}$
11	$\{r\}$	$\{c\}$

i_i	o_i
$i_1 = \{e\}$	$o_1 = \{a, e\}$
$i_2 = \{a, e\}$	$o_2 = \{a, b, e\}$
$i_3 = \{a, b, e\}$	$o_3 = \{a, b, e\}$
$i_4 = \{a, b, e\}$	$o_4 = \{a, b, d, e\}$
$i_5 = \{a, b, d, e\}$	$o_5 = \{a, b, d, e\}$
$i_6 = \{a, b, d, e\}$	$o_6 = \{d, e\}$
$i_7 = \{d\}$	$o_7 = \{c\}$
$i_9 = \{e\}$	$o_9 = \{c\}$
$i_{11} = \{c\}$	$o_{11} = \{\}$

Détection des *bof* par analyse statique

graphe de contrôle de flux – `uno`

- Utilise les graphes de contrôle de flux
- Aucune interprétation du code
- Utilisation d'heuristique pour détecter les débordements de tampons
- Exemple ... *uno*

Détection des *bof* par analyse statique

interprétation abstraite – `boon`

- Principe de l'outil `boon`
  - Prise en compte uniquement des chaînes de caractères
  - Représentation des chaînes sous forme de couples (*taille allouée*, *taille utilisée*)
  - Définition d'un langage de représentation de contraintes
  - Traduction des instructions avec ce langage et résolution des équations obtenues
- Structures manipulées
  - $\mathbb{Z}$ est l'ensemble des entiers et $\mathbb{Z}^\infty = \mathbb{Z} \cup \{-\infty, \infty\}$
  - Un intervalle est un ensemble $R \subseteq \mathbb{Z}^\infty$ de la forme $[a, b] = \{i \in \mathbb{Z}^\infty : a \leq i \leq b\}$
  - La fermeture d'un ensemble S est l'intervalle $R = [min(S), max(S)]$
  - Opérations sur les intervalles :
    $S + T = \{s + t : s \in S, t \in T\}$
    $S - T = \{s - t : s \in S, t \in T\}$
    $S * T = \{s * t : s \in S, t \in T\}$

Détection des *bof* par analyse statique

interprétation abstraite – `boon`

→ Règles de traduction des instructions (sous ensemble du langage `C`)

Instruction `C`	Contrainte
`i = i + j`	$i + j \subseteq i$
`char* s = "stylo"`	$6 \subseteq len(s), \quad 6 \subseteq alloc(s)$
`char* m = malloc(sizeof(char) * 20)`	$20 \subseteq alloc(m)$
`sprintf(b2, " est %s", c1)`	$len(c1) + 5 \subseteq len(b2)$
`sprintf(b2, "%s%s", c1, c2)`	$len(c1) + len(c2) - 1 \subseteq len(b2)$

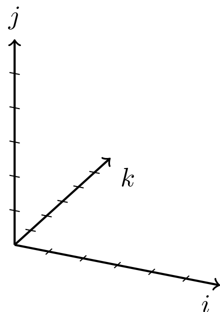
Propriété de sécurité considérée : $\forall s, \quad len(s) \subseteq alloc(s)$

Détection des *bof* par analyse statique

interprétation abstraite – `boon`

→ Résolution des contraintes

- **Intuition** : dans un programme avec n variables, on considère l'espace $\mathbb{Z}^n$ dont la $i^{\text{ème}}$ dimension correspond à la $i^{\text{ème}}$ variable. Une exécution de ce programme correspond à un chemin dans cet espace



- **Objectif** : identifier la plus petite boîte englobante correspondant à toutes les exécutions possibles (sans apprentissage !)
- **Méthode** : représentation des variables sous forme de nœuds et les contraintes sous forme d'arcs. Ce graphe est employé pour propager les changements sur les contraintes

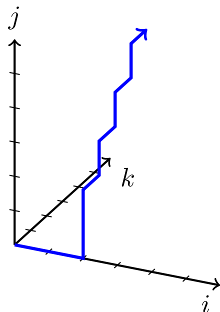
```
i = 2;
for (k = 0; k < 4; k++) {
  j = i + k;
}
```

Détection des *bof* par analyse statique

interprétation abstraite – boon

→ Résolution des contraintes

- **Intuition** : dans un programme avec n variables, on considère l'espace $\mathbb{Z}^n$ dont la $i^{\text{ème}}$ dimension correspond à la $i^{\text{ème}}$ variable. Une exécution de ce programme correspond à un chemin dans cet espace



- **Objectif** : identifier la plus petite boîte englobante correspondant à toutes les exécutions possibles (sans apprentissage !)
- **Méthode** : représentation des variables sous forme de nœuds et les contraintes sous forme d'arcs. Ce graphe est employé pour propager les changements sur les contraintes

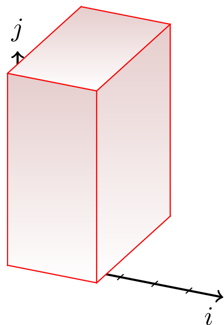
```
i = 2;
for (k = 0; k < 4; k++) {
  j = i + k;
}
```

Détection des *bof* par analyse statique

interprétation abstraite – `boon`

→ Résolution des contraintes

- **Intuition** : dans un programme avec n variables, on considère l'espace $\mathbb{Z}^n$ dont la $i^{\text{ème}}$ dimension correspond à la $i^{\text{ème}}$ variable. Une exécution de ce programme correspond à un chemin dans cet espace



- **Objectif** : identifier la plus petite boîte englobante correspondant à toutes les exécutions possibles (sans apprentissage !)
- **Méthode** : représentation des variables sous forme de nœuds et les contraintes sous forme d'arcs. Ce graphe est employé pour propager les changements sur les contraintes

```
i = 2;
for (k = 0; k < 4; k++) {
  j = i + k;
}
```

Détection des *bof* par analyse statique

interprétation abstraite – boon

→ Résolution des contraintes

```
void test(int i) {
  char* o1 = "stylo";
  char* o2 = "pinceau";
  char* c1 = "vert";
  char* c2 = "rouge";
  char* p = malloc(sizeof(char) * 20);
  char* b1 = malloc(sizeof(char) * 16);
  char* b2 = malloc(sizeof(char) * 12);
  if (i % 2 == 0) {
    sprintf(b1, "le %s", o1);
  } else {
    sprintf(b1, "le %s", o2);
  }
  if (i > 1) {
    sprintf(b2, " est %s", c1);
  } else {
    sprintf(b2, " est %s", c2);
  }
  sprintf(p, "%s%s", b1, b2);
  printf("phrase: %s\n", p);
}
```

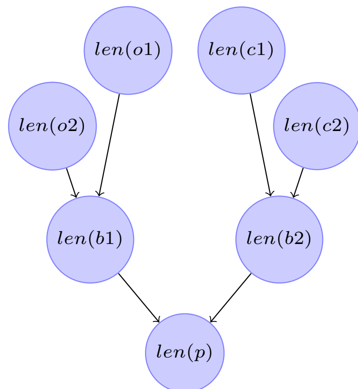
l	*Contraintes*
1	$6 \subseteq len(o1), \quad 6 \subseteq alloc(o1)$
2	$8 \subseteq len(o2), \quad 8 \subseteq alloc(o2)$
3	$5 \subseteq len(c1), \quad 5 \subseteq alloc(c1)$
4	$6 \subseteq len(c2), \quad 6 \subseteq alloc(c2)$
5	$20 \subseteq alloc(p)$
6	$16 \subseteq alloc(b1)$
7	$12 \subseteq alloc(b2)$
8	$3 + len(o1) \subseteq len(b1)$
9	$3 + len(o2) \subseteq len(b1)$
10	$5 + len(c1) \subseteq len(b2)$
11	$5 + len(c2) \subseteq len(b2)$
12	$len(b1) + len(b2) - 1 \subseteq len(p)$

Détection des *bof* par analyse statique

interprétation abstraite – boon

→ Résolution des contraintes indépendamment de l'ordre des instructions

l	*Contraintes*
1	$6 \subseteq len(o1), \quad 6 \subseteq alloc(o1)$
2	$8 \subseteq len(o2), \quad 8 \subseteq alloc(o2)$
3	$5 \subseteq len(c1), \quad 5 \subseteq alloc(c1)$
4	$6 \subseteq len(c2), \quad 6 \subseteq alloc(c2)$
5	$20 \subseteq alloc(p)$
6	$16 \subseteq alloc(b1)$
7	$12 \subseteq alloc(b2)$
8	$3 + len(o1) \subseteq len(b1)$
9	$3 + len(o2) \subseteq len(b1)$
10	$5 + len(c1) \subseteq len(b2)$
11	$5 + len(c2) \subseteq len(b2)$
12	$len(b1) + len(b2) - 1 \subseteq len(p)$

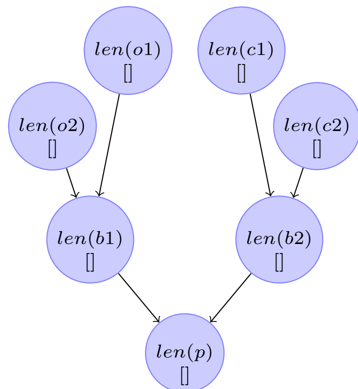


Détection des *bof* par analyse statique

interprétation abstraite – boon

→ Résolution des contraintes indépendamment de l'ordre des instructions

l	*Contraintes*
1	$6 \subseteq len(o1), \quad 6 \subseteq alloc(o1)$
2	$8 \subseteq len(o2), \quad 8 \subseteq alloc(o2)$
3	$5 \subseteq len(c1), \quad 5 \subseteq alloc(c1)$
4	$6 \subseteq len(c2), \quad 6 \subseteq alloc(c2)$
5	$20 \subseteq alloc(p)$
6	$16 \subseteq alloc(b1)$
7	$12 \subseteq alloc(b2)$
8	$3 + len(o1) \subseteq len(b1)$
9	$3 + len(o2) \subseteq len(b1)$
10	$5 + len(c1) \subseteq len(b2)$
11	$5 + len(c2) \subseteq len(b2)$
12	$len(b1) + len(b2) - 1 \subseteq len(p)$

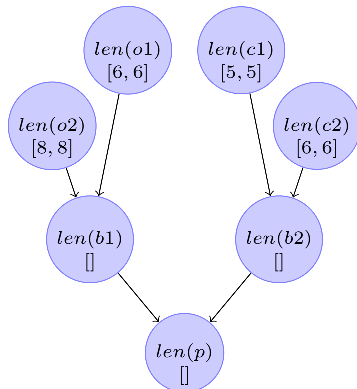


Détection des *bof* par analyse statique

interprétation abstraite – `boon`

→ Résolution des contraintes indépendamment de l'ordre des instructions

l	*Contraintes*
1	$6 \subseteq len(o1), \quad 6 \subseteq alloc(o1)$
2	$8 \subseteq len(o2), \quad 8 \subseteq alloc(o2)$
3	$5 \subseteq len(c1), \quad 5 \subseteq alloc(c1)$
4	$6 \subseteq len(c2), \quad 6 \subseteq alloc(c2)$
5	$20 \subseteq alloc(p)$
6	$16 \subseteq alloc(b1)$
7	$12 \subseteq alloc(b2)$
8	$3 + len(o1) \subseteq len(b1)$
9	$3 + len(o2) \subseteq len(b1)$
10	$5 + len(c1) \subseteq len(b2)$
11	$5 + len(c2) \subseteq len(b2)$
12	$len(b1) + len(b2) - 1 \subseteq len(p)$

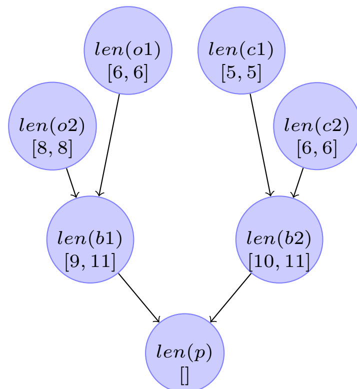


Détection des *bof* par analyse statique

interprétation abstraite – boon

→ Résolution des contraintes indépendamment de l'ordre des instructions

l	*Contraintes*
1	$6 \subseteq len(o1), \quad 6 \subseteq alloc(o1)$
2	$8 \subseteq len(o2), \quad 8 \subseteq alloc(o2)$
3	$5 \subseteq len(c1), \quad 5 \subseteq alloc(c1)$
4	$6 \subseteq len(c2), \quad 6 \subseteq alloc(c2)$
5	$20 \subseteq alloc(p)$
6	$16 \subseteq alloc(b1)$
7	$12 \subseteq alloc(b2)$
8	$3 + len(o1) \subseteq len(b1)$
9	$3 + len(o2) \subseteq len(b1)$
10	$5 + len(c1) \subseteq len(b2)$
11	$5 + len(c2) \subseteq len(b2)$
12	$len(b1) + len(b2) - 1 \subseteq len(p)$

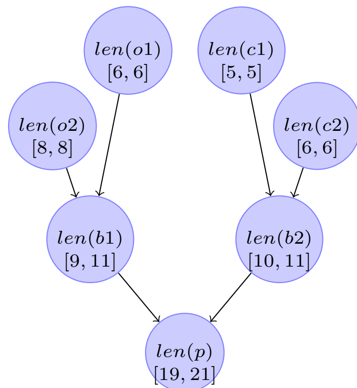


Détection des *bof* par analyse statique

interprétation abstraite – boon

→ Résolution des contraintes indépendamment de l'ordre des instructions

l	*Contraintes*
1	$6 \subseteq len(o1), \quad 6 \subseteq alloc(o1)$
2	$8 \subseteq len(o2), \quad 8 \subseteq alloc(o2)$
3	$5 \subseteq len(c1), \quad 5 \subseteq alloc(c1)$
4	$6 \subseteq len(c2), \quad 6 \subseteq alloc(c2)$
5	$20 \subseteq alloc(p)$
6	$16 \subseteq alloc(b1)$
7	$12 \subseteq alloc(b2)$
8	$3 + len(o1) \subseteq len(b1)$
9	$3 + len(o2) \subseteq len(b1)$
10	$5 + len(c1) \subseteq len(b2)$
11	$5 + len(c2) \subseteq len(b2)$
12	$len(b1) + len(b2) - 1 \subseteq len(p)$



$! \ max(len(p)) > alloc(p)$

Détection des *bof* par analyse statique

interprétation abstraite – `boon`

- Limites de l'approche
  - L'ordre des instructions n'est pas pris en compte

  → Beaucoup de faux positifs
  - Difficulté pour traiter les itérations : `i = 0; i = i + 1;` aboutit à $[0, \infty] \in i$

```
i = 0;
for (j = 0; j < 4; j++) {
  i = i + 1;
}
```

  - Le traitement de **`strcat`** entraîne systématiquement une alerte

    `strcat(a, b)` $\rightarrow len(a) + len(b) - 1 \subseteq len(s)$
  - Arithmétique des pointeurs non pris en compte

    `char* p; char* s = "abcd"; p = s + 4;`

Détection des *bof* par analyse statique

interprétation abstraite – ../..

- De nombreuses autres approches, basées sur l'interprétation abstraite
- Outil `astree` de `AbsInt` (`CNRS` et `ENS`) et `penjili` de `EADS-IW`
  - L'ordre des instructions **est** pris en compte
  - Bon taux de faux positifs et faux négatifs
  - Prise en compte de l'arithmétique des pointeurs
  - L'abstraction prend en compte la pile et le tas