L'Art de la Sécurité Informatique

une [trop] brève introduction

Nicolas Bernard

Université du Luxembourg
<n.bernard@lafraze.net>

Janvier 2005

Qu'est-ce que la sécurité ?

Buts :

- Maintenir un système informatique en état de fonctionner normalement ;
- Restreindre l'accès à certaines informations aux utilisateurs autorisés à les consulter.

Il peut y avoir des contradictions !

Principaux domaines de recherche

- Réseaux ;
- systèmes d'exploitation ;
- programmation ;
- cryptologie ;
- logique ;
- architecture ;
- etc.

Sommaire

1. Présentation Générale
2. Quelques grands principes de sécurité
3. Problèmes logiciels
4. Conclusion

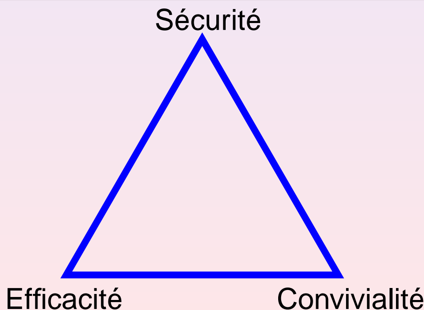
Sommaire

1. Présentation Générale
2. Quelques grands principes de sécurité
   - La sécurité est un compromis
   - Sécurité par l'obscurité
   - Systèmes de sécurité en “parallèle”
   - Systèmes de sécurité en “série”
   - Hétérogénéité : Darwin et la sécurité
   - Compartimenter
3. Problèmes logiciels
4. Conclusion

« La sécurité est un compromis. »
– Bruce Schneier

Principe

On ne peut pas tout avoir, il faut faire des choix, la sécurité n'est qu'un des choix possibles.



Principe

La **Sécurité par l'Obscurité** *ne marche pas !*

Conséquence (Loi de Kerckhoffs, 1883)

Un cryptosystème devrait être sûr même s'il est totalement public, à l'exception de la clef.

Conséquence

La sécurité d'un système ne doit pas dépendre du client.

Principe

La **Sécurité par l'Obscurité** *ne marche pas !*

Conséquence (Loi de Kerckhoffs, 1883)

Un cryptosystème devrait être sûr même s'il est totalement public, à l'exception de la clef.

Conséquence

La sécurité d'un système ne doit pas dépendre du client.

Principe

La **Sécurité par l'Obscurité** *ne marche pas !*

Conséquence (Loi de Kerckhoffs, 1883)

Un cryptosystème devrait être sûr même s'il est totalement public, à l'exception de la clef.

Conséquence

La sécurité d'un système ne doit pas dépendre du client.
Exemple: le système de protection des DVD-videos

« Une chaîne est aussi forte que son maillon le plus faible. »
– sagesse populaire

Principe

Si différents mécanismes de sécurité jouent le même rôle, le système est aussi sûr que le plus faible de ces mécanismes.

Exemple

Un ordinateur auquel on peut se connecter soit

- *via un mot de passe ;*
- *via un lecteur d'empreintes digitales.*

« Une chaîne est aussi forte que son maillon le plus faible. »
– sagesse populaire

Principe

Si différents mécanismes de sécurité jouent le même rôle, le système est aussi sûr que le plus faible de ces mécanismes.

Exemple

Un ordinateur auquel on peut se connecter soit

- *via un mot de passe ;*
- *via un lecteur d'empreintes digitales.*

« Remember this security mantra: defense in depth »
– Fyodor, on nmap-hackers, 2004-11-23

Principe (Défense en profondeur)

Utiliser plusieurs couches de sécurité permet à l'une d'échouer sans compromettre tout le système.

Exemple

Un réseau protégé par un firewall dont

- *chaque machine est sûre en elle-même ;*
- *le réseau interne est chiffré ;*
- *etc.*

« Remember this security mantra: defense in depth »
– Fyodor, on nmap-hackers, 2004-11-23

Principe (Défense en profondeur)

Utiliser plusieurs couches de sécurité permet à l'une d'échouer sans compromettre tout le système.

Exemple

Un réseau protégé par un firewall dont

- *chaque machine est sûre en elle-même ;*
- *le réseau interne est chiffré ;*
- *etc.*

Principe (diversité)

Des systèmes identiques sont vulnérables à des attaques à grande échelle contre leurs caractéristiques communes.

Exemple

Les vers et virus informatiques exploitent ces facteurs.

Principe (Séparation des pouvoirs)

Une entité (programme, utilisateur, . . .) devrait avoir accès à et uniquement à ce dont elle a besoin (informations, pouvoirs, ressources, . . .) pour accomplir sa tâche.

Exemple

Séparation exécutif / législatif / judiciaire.

Exemple

Droits d'accès sur un système d'exploitation.

Principe (Séparation des pouvoirs)

Une entité (programme, utilisateur, . . .) devrait avoir accès à et uniquement à ce dont elle a besoin (informations, pouvoirs, ressources, . . .) pour accomplir sa tâche.

Exemple

Séparation exécutif / législatif / judiciaire.

Exemple

Droits d'accès sur un système d'exploitation.

Principe (Séparation des pouvoirs)

Une entité (programme, utilisateur, . . .) devrait avoir accès à et uniquement à ce dont elle a besoin (informations, pouvoirs, ressources, . . .) pour accomplir sa tâche.

Exemple

Séparation exécutif / législatif / judiciaire.

Exemple

Droits d'accès sur un système d'exploitation.

Sommaire

1. Présentation Générale
2. Quelques grands principes de sécurité
3. Problèmes logiciels
   - Erreurs de conception
   - Erreurs d'implémentation : buffer-overflows
   - Erreurs d'implémentation : format strings
   - Erreurs d'implémentation : race conditions
   - Erreurs d'implémentation : injections de code
4. Conclusion

- Penser à la sécurité pendant la conception !
- La rajouter après sera au mieux difficile !

Exemple (e-mail)

- *Confidentialité : relativement facile à ajouter (OpenPGP, S/MIME)*
- *Intimité des relations : beaucoup plus difficile (mixmaster, etc)*
- *Protection contre le déni de service et les envois indésirés : impossible à ajouter ?*

- Penser à la sécurité pendant la conception !
- La rajouter après sera au mieux difficile !

Exemple (e-mail)

- *Confidentialité : relativement facile à ajouter (OpenPGP, S/MIME)*
- *Intimité des relations : beaucoup plus difficile (mixmaster, etc)*
- *Protection contre le déni de service et les envois indésirés : impossible à ajouter ?*

- Penser à la sécurité pendant la conception !
- La rajouter après sera au mieux difficile !

Exemple (e-mail)

- *Confidentialité : relativement facile à ajouter (OpenPGP, S/MIME)*
- *Intimité des relations : beaucoup plus difficile (mixmaster, etc)*
- *Protection contre le déni de service et les envois indésirés : impossible à ajouter ?*

- Penser à la sécurité pendant la conception !
- La rajouter après sera au mieux difficile !

Exemple (e-mail)

- *Confidentialité : relativement facile à ajouter (OpenPGP, S/MIME)*
- *Intimité des relations : beaucoup plus difficile (mixmaster, etc)*
- *Protection contre le déni de service et les envois indésirés : impossible à ajouter ?*

- Penser à la sécurité pendant la conception !
- La rajouter après sera au mieux difficile !

Exemple (e-mail)

- *Confidentialité : relativement facile à ajouter (OpenPGP, S/MIME)*
- *Intimité des relations : beaucoup plus difficile (mixmaster, etc)*
- *Protection contre le déni de service et les envois indésirés : impossible à ajouter ?*

Keep It Simple, Stupid !

« Chaque chose doit être rendue aussi simple que possible, mais pas trop simple »
– Albert Einstein

Principe (KISS : Keep It Simple, Stupid)

- *origine dans la "philosophie" Unix ;*
- *un programme fait une seule chose, mais il la fait bien.*

Limites :

- hypothèse implicite : il est facile de dire si une chose simple est sûre ;
- les interactions entres des choses simples peuvent devenir complexes !

Keep It Simple, Stupid !

« Chaque chose doit être rendue aussi simple que possible,
mais pas trop simple »
– Albert Einstein

Principe (KISS : Keep It Simple, Stupid)

- *origine dans la "philosophie" Unix ;*
- *un programme fait une seule chose, mais il la fait bien.*

Limites :

- hypothèse implicite : il est facile de dire si une chose simple est sûre ;
- les interactions entres des choses simples peuvent devenir complexes !

Le buffer-overflow

Famille des débordements de tampons (ou buffer-overflows) :

- sans doute celle ayant provoqué le plus de failles de sécurité ;
- ancienne :
  - risque perçu depuis les années 1960 !
  - utilisée par l'Internet Worm de 1988 ;
- bien connue depuis 1996 ;
- tend aujourd'hui à disparaître (?).

Le buffer-overflow

Famille des débordements de tampons (ou buffer-overflows) :

- sans doute celle ayant provoqué le plus de failles de sécurité ;
- ancienne :
  - risque perçu depuis les années 1960 !
  - utilisée par l'Internet Worm de 1988 ;
- bien connue depuis 1996 ;
- tend aujourd'hui à disparaître (?).

Programme chargé en mémoire (simplifié)

Le segment de "texte"

- Il contient les instructions ;
- Il a généralement les permissions "lecture" et "exécution".

Programme chargé en mémoire (simplifié)

Le segment de données

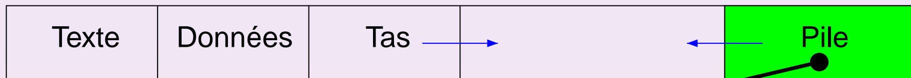
- Il contient les variables statiques ou globales ;
- note : l'exécutable sur le disque stocke généralement dans deux segments distincts, *data* et *bss* les données selon qu'elles sont initialisées ou non.

Programme chargé en mémoire (simplifié)

Le tas (*heap*)

- C'est là que se trouve la place allouée par malloc(3) ;
- Il grandit vers les adresses plus hautes.

Programme chargé en mémoire (simplifié)



La pile (*stack*)

- Contient les informations temporaires, les variables automatiques, les paramètres de fonctions, etc ;
- elle grandit vers le bas, vers le tas.

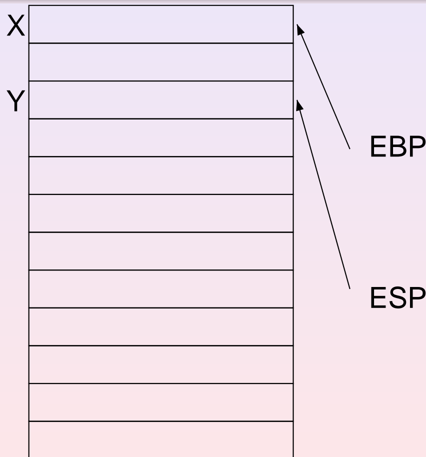
Que se passe-t'il lors d'un appel de fonction ?

```
int foobar(int a, int b)
{
    ⋮
    return c;
}

int main()
{
    ⋮
    a = foobar(32,52);
    ⋮
}
```

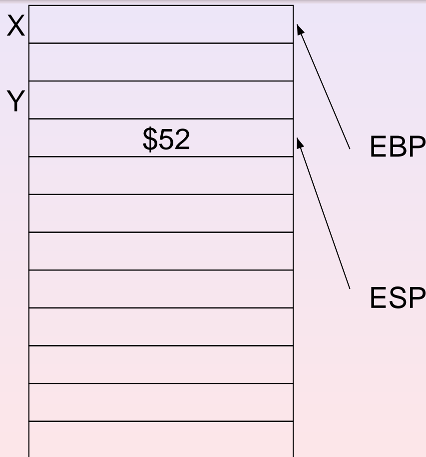
Que se passe-t'il lors d'un appel de fonction ?

```
foobar:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp
     ⋮
    leave
    ret
main:
     ⋮
    mov $52, (%esp)
    mov $32, (%esp)
    call foobar
     ⋮
```



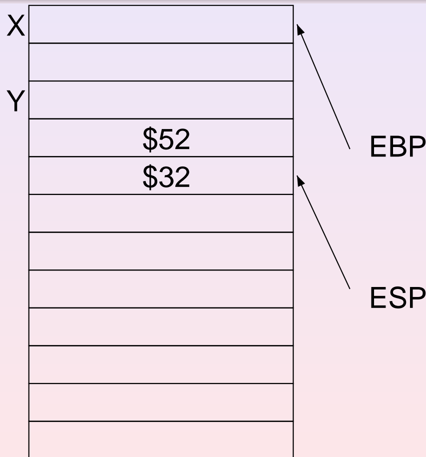
Que se passe-t'il lors d'un appel de fonction ?

```
foobar:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp
    ⋮
    leave
    ret
main:
    ⋮
    mov $52, (%esp)
    mov $32, (%esp)
    call foobar
    ⋮
```



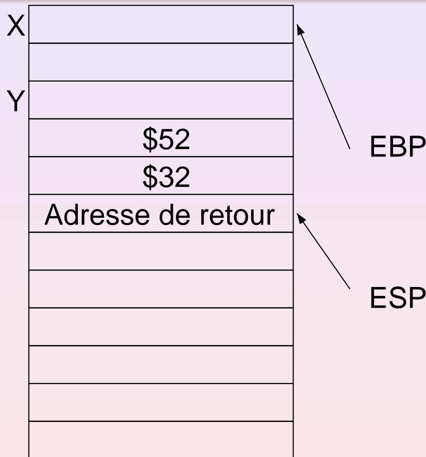
Que se passe-t'il lors d'un appel de fonction ?

```
foobar:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp
    ⋮
    leave
    ret
main:
    ⋮
    mov $52, (%esp)
    mov $32, (%esp)
    call foobar
    ⋮
```



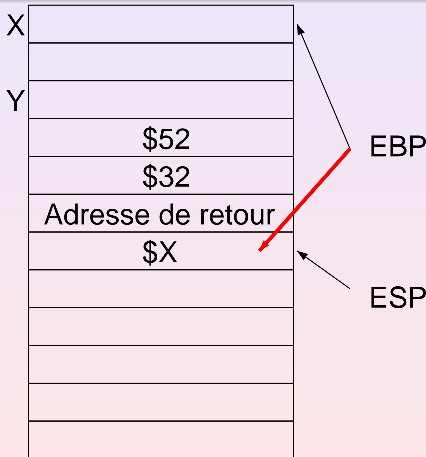
Que se passe-t'il lors d'un appel de fonction ?

```
foobar:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp
    ⋮
    leave
    ret
main:
    ⋮
    mov $52, (%esp)
    mov $32, (%esp)
    call foobar
    ⋮
```



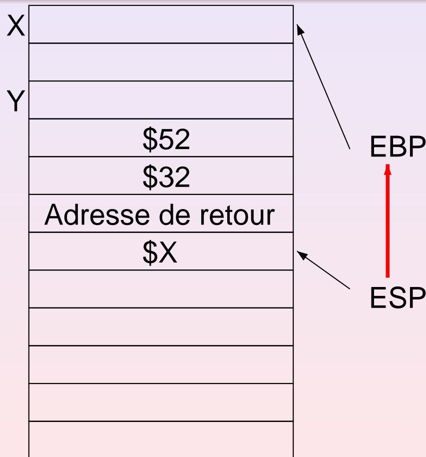
Que se passe-t'il lors d'un appel de fonction ?

```
foobar:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp
    ⋮
    leave
    ret
main:
    ⋮
    mov $52, (%esp)
    mov $32, (%esp)
    call foobar
    ⋮
```



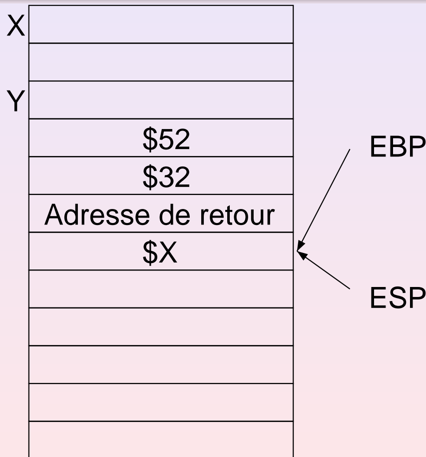
Que se passe-t'il lors d'un appel de fonction ?

```
foobar:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp
    ⋮
    leave
    ret
main:
    ⋮
    mov $52, (%esp)
    mov $32, (%esp)
    call foobar
    ⋮
```



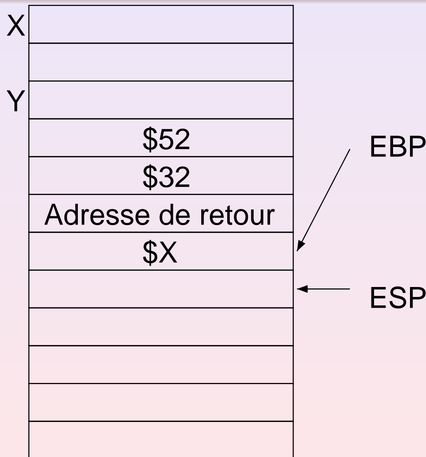
Que se passe-t'il lors d'un appel de fonction ?

```
foobar:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp
    ⋮
    leave
    ret
main:
    ⋮
    mov $52, (%esp)
    mov $32, (%esp)
    call foobar
    ⋮
```



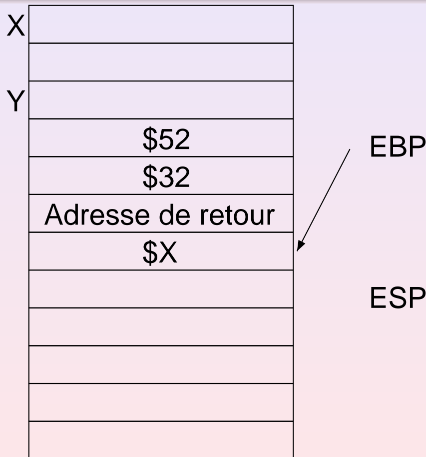
Que se passe-t'il lors d'un appel de fonction ?

```
foobar:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp
    ⋮
    leave
    ret
main:
    ⋮
    mov $52, (%esp)
    mov $32, (%esp)
    call foobar
    ⋮
```



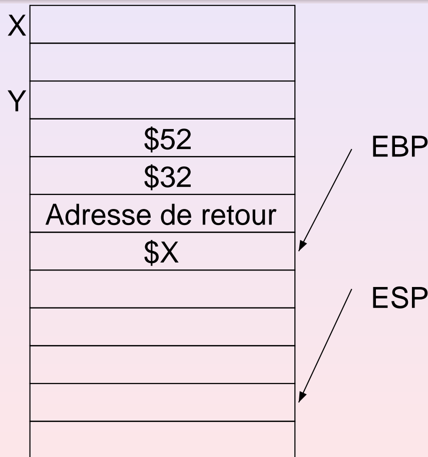
Que se passe-t'il lors d'un appel de fonction ?

```
foobar:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp
    ⋮
    leave
    ret
main:
    ⋮
    mov $52, (%esp)
    mov $32, (%esp)
    call foobar
    ⋮
```



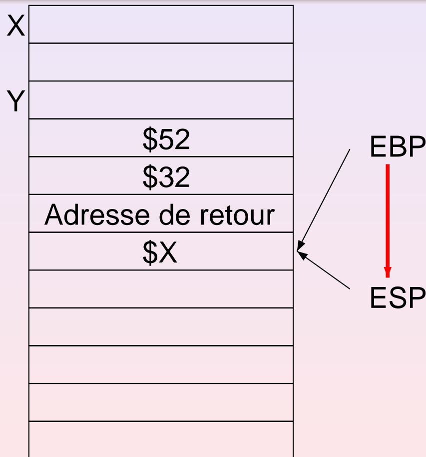
Que se passe-t'il lors d'un appel de fonction ?

```
foobar:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp
    ⋮
    leave
    ret
main:
    ⋮
    mov $52, (%esp)
    mov $32, (%esp)
    call foobar
    ⋮
```



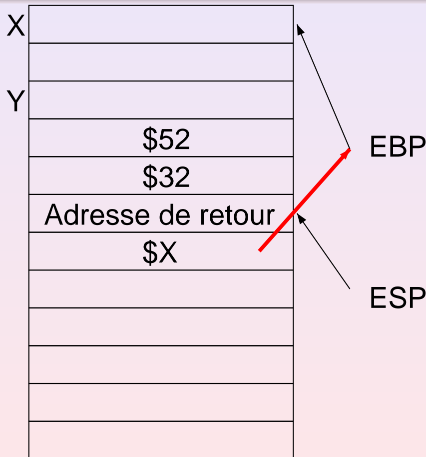
Que se passe-t'il lors d'un appel de fonction ?

```
foobar:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp
    ⋮
    leave
    ret
main:
    ⋮
    mov $52, (%esp)
    mov $32, (%esp)
    call foobar
    ⋮
```



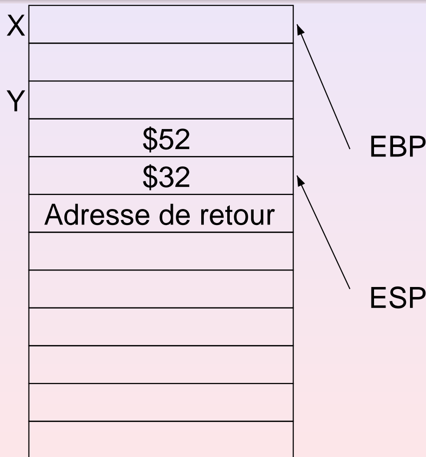
Que se passe-t'il lors d'un appel de fonction ?

```
foobar:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp
    ⋮
    leave
    ret
main:
    ⋮
    mov $52, (%esp)
    mov $32, (%esp)
    call foobar
    ⋮
```



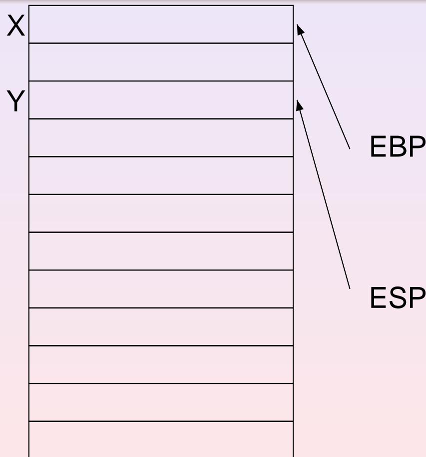
Que se passe-t'il lors d'un appel de fonction ?

```
foobar:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp
    ⋮
    leave
    ret
main:
    ⋮
    mov $52, (%esp)
    mov $32, (%esp)
    call foobar
    ⋮
```



Que se passe-t'il lors d'un appel de fonction ?

```
foobar:
    push %ebp
    mov %esp, %ebp
    sub $4, %esp
    ⋮
    leave
    ret
main:
    ⋮
    mov $52, (%esp)
    mov $32, (%esp)
    call foobar
    ⋮
```



Stack Smashing !

Principe :

- si une fonction contient un tableau automatique local, il est placé sur la pile ;

Stack Smashing !

Principe :

- si une fonction contient un tableau automatique local, il est placé sur la pile ;
- si la même fonction fait une copie dedans sans vérifier sa taille (ex : *strcpy(3)*), on écrase le reste de la pile ;

Stack Smashing !

Principe :

- si une fonction contient un tableau automatique local, il est placé sur la pile ;
- si la même fonction fait une copie dedans sans vérifier sa taille (ex : *strcpy(3)*), on écrase le reste de la pile ;

Stack Smashing !

Principe :

- si une fonction contient un tableau automatique local, il est placé sur la pile ;
- si la même fonction fait une copie dedans sans vérifier sa taille (ex : *strcpy(3)*), on écrase le reste de la pile ;

Stack Smashing !

Principe :

- si une fonction contient un tableau automatique local, il est placé sur la pile ;
- si la même fonction fait une copie dedans sans vérifier sa taille (ex : *strcpy(3)*), on écrase le reste de la pile ;

Stack Smashing !

Principe :

- si une fonction contient un tableau automatique local, il est placé sur la pile ;
- si la même fonction fait une copie dedans sans vérifier sa taille (ex : *strcpy(3)*), on écrase le reste de la pile ;

Stack Smashing !

Principe :

- si une fonction contient un tableau automatique local, il est placé sur la pile ;
- si la même fonction fait une copie dedans sans vérifier sa taille (ex : *strcpy(3)*), on écrase le reste de la pile ;

Stack Smashing !

Principe :

- si une fonction contient un tableau automatique local, il est placé sur la pile ;
- si la même fonction fait une copie dedans sans vérifier sa taille (ex : *strcpy(3)*), on écrase le reste de la pile ;

Stack Smashing !

Principe :

- si une fonction contient un tableau automatique local, il est placé sur la pile ;
- si la même fonction fait une copie dedans sans vérifier sa taille (ex : *strcpy(3)*), on écrase le reste de la pile ;

Stack Smashing !

Principe :

- si une fonction contient un tableau automatique local, il est placé sur la pile ;
- si la même fonction fait une copie dedans sans vérifier sa taille (ex : *strcpy(3)*), on écrase le reste de la pile ;

Stack Smashing !

Principe :

- si une fonction contient un tableau automatique local, il est placé sur la pile ;
- si la même fonction fait une copie dedans sans vérifier sa taille (ex : *strcpy(3)*), on écrase le reste de la pile ;

Stack Smashing !

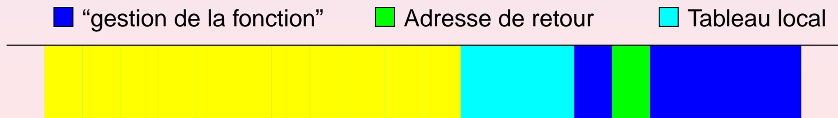
Principe :

- si une fonction contient un tableau automatique local, il est placé sur la pile ;
- si la même fonction fait une copie dedans sans vérifier sa taille (ex : *strcpy(3)*), on écrase le reste de la pile ;

Stack Smashing !

Principe :

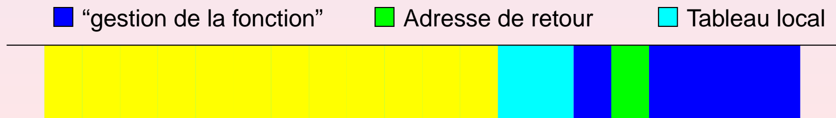
- si une fonction contient un tableau automatique local, il est placé sur la pile ;
- si la même fonction fait une copie dedans sans vérifier sa taille (ex : *strcpy(3)*), on écrase le reste de la pile ;



Stack Smashing !

Principe :

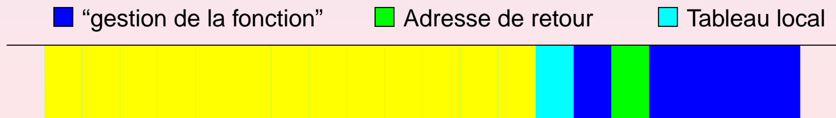
- si une fonction contient un tableau automatique local, il est placé sur la pile ;
- si la même fonction fait une copie dedans sans vérifier sa taille (ex : *strcpy(3)*), on écrase le reste de la pile ;



Stack Smashing !

Principe :

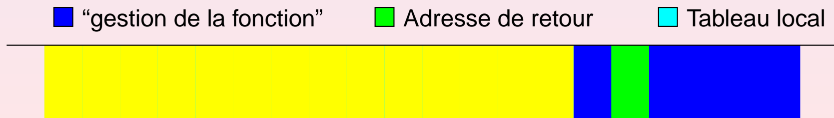
- si une fonction contient un tableau automatique local, il est placé sur la pile ;
- si la même fonction fait une copie dedans sans vérifier sa taille (ex : *strcpy(3)*), on écrase le reste de la pile ;



Stack Smashing !

Principe :

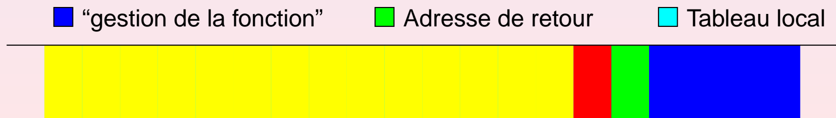
- si une fonction contient un tableau automatique local, il est placé sur la pile ;
- si la même fonction fait une copie dedans sans vérifier sa taille (ex : *strcpy(3)*), on écrase le reste de la pile ;



Stack Smashing !

Principe :

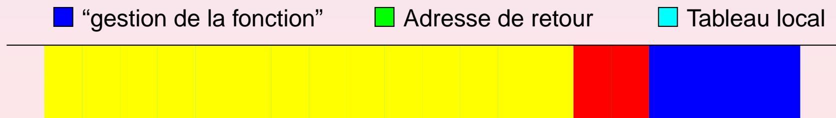
- si une fonction contient un tableau automatique local, il est placé sur la pile ;
- si la même fonction fait une copie dedans sans vérifier sa taille (ex : *strcpy(3)*), on écrase le reste de la pile ;



Stack Smashing !

Principe :

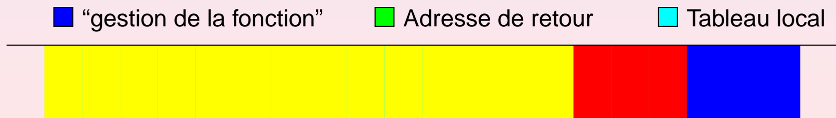
- si une fonction contient un tableau automatique local, il est placé sur la pile ;
- si la même fonction fait une copie dedans sans vérifier sa taille (ex : *strcpy(3)*), on écrase le reste de la pile ;



Stack Smashing !

Principe :

- si une fonction contient un tableau automatique local, il est placé sur la pile ;
- si la même fonction fait une copie dedans sans vérifier sa taille (ex : *strcpy(3)*), on écrase le reste de la pile ;

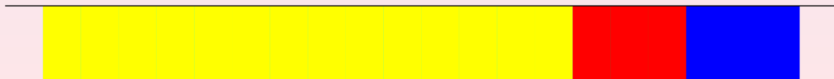


Stack Smashing !

Principe :

- si une fonction contient un tableau automatique local, il est placé sur la pile ;
- si la même fonction fait une copie dedans sans vérifier sa taille (ex : *strcpy(3)*), on écrase le reste de la pile ;
- on peut donc remplacer l’adresse de retour de la fonction par celle d’un morceau de code (*shellcode*) arbitraire.

“gestion de la fonction” Adresse de retour Tableau local



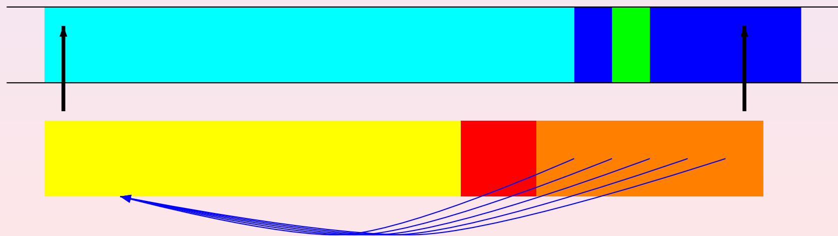
Stack Smashing !

Problèmes :

- placer le shellcode dans l'espace d'adressage du processus ;
- connaître l'adresse où il se trouve.

shellcode | pointeurs sur le buffer | *landing pad*

"gestion de la fonction" | Adresse de retour | Tableau local



Solution 1: placer un *canari* sur la pile

Avantages :

- fonctionne avec toutes les architectures ;
- ne dépend pas du système d'exploitation ;

Inconvénients :

- il est souvent possible de la contourner ;
- nécessite une recompilation des programmes ;
- baisse des performances (vérifications durant l'exécution) ;
- certains programmes exotiques ne marchent pas.

Méthode actuellement utilisée : gcc (patches Propolice, StackGuard, etc), MS Visual C, etc.

Solution 1: placer un *canari* sur la pile

Avantages :

- fonctionne avec toutes les architectures ;
- ne dépend pas du système d'exploitation ;

Inconvénients :

- il est souvent possible de la contourner ;
- nécessite une recompilation des programmes ;
- baisse des performances (vérifications durant l'exécution) ;
- certains programmes exotiques ne marchent pas.

Méthode actuellement utilisée : gcc (patches Propolice, StackGuard, etc), MS Visual C, etc.

Solution 1: placer un *canari* sur la pile

Avantages :

- fonctionne avec toutes les architectures ;
- ne dépend pas du système d'exploitation ;

Inconvénients :

- il est souvent possible de la contourner ;
- nécessite une recompilation des programmes ;
- baisse des performances (vérifications durant l'exécution) ;
- certains programmes exotiques ne marchent pas.

Méthode actuellement utilisée : gcc (patches Propolice, StackGuard, etc), MS Visual C, etc.

Contourner cette solution

Dans certains cas, il est possible de contourner le canari :

Exemple :

- un pointeur sur la pile ;
- deux strcpy (ou autres fonctions non sûres) ;
- le deuxième strcpy copie à l'adresse indiquée par p ;

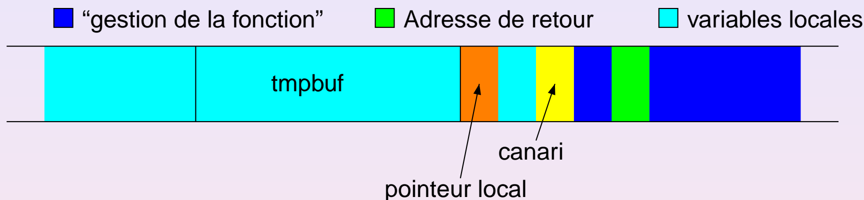
```
char* foobar()
{
    int i = 0;
    char* p = (char*) malloc(1024);
    char tmpbuf[1024];
    char toto = getchar(stdin);
    while (toto != '\n') {
        tmpbuf[i++] = toto;
        toto = getchar(stdin);
    }
    if (validate(tmpbuf))
        strcpy(p, tmpbuf);
    .
    .
    .
    return p;
}
```

Contourner cette solution

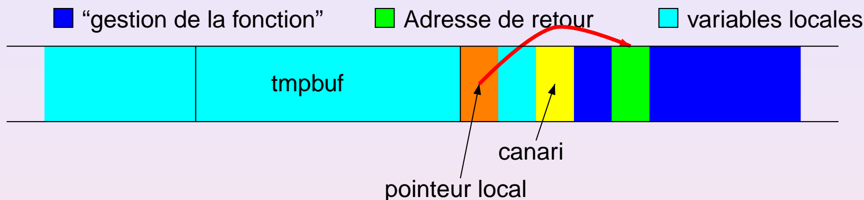
Dans certains cas, il est possible de contourner le canari :
Exemple :

- un pointeur sur la pile ;
- deux strcpy (ou autres fonctions non sûres) ;
- le deuxième strcpy copie à l'adresse indiquée par *p* ;

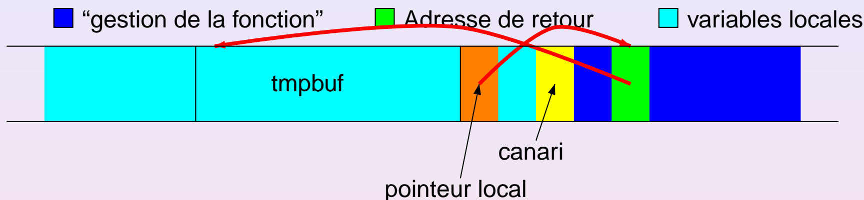
```
char* foobar()
{
    int i = 0;
    char* p = (char*) malloc(1024);
    char tmpbuf[1024];
    char toto = getchar(stdin);
    while (toto != '\n') {
        tmpbuf[i++] = toto;
        toto = getchar(stdin);
    }
    if (validate(tmpbuf))
        strcpy(p, tmpbuf);
    .
    .
    .
    return p;
}
```



- on utilise le premier strcpy pour faire pointer *p* sur la position de l'adresse de retour;
- on change l'adresse de retour sans toucher au canari avec le second strcpy.



- on utilise le premier strcpy pour faire pointer *p* sur la position de l'adresse de retour;
- on change l'adresse de retour sans toucher au canari avec le second strcpy.



- on utilise le premier strcpy pour faire pointer *p* sur la position de l'adresse de retour;
- on change l'adresse de retour sans toucher au canari avec le second strcpy.

Solution 2: rendre la pile non exécutable

Avantages :

- pratiquement pas de pertes de performances ;
- ne nécessite pas de recompilation ;

Inconvénients :

- nécessite un support du processeur ;
- nécessite un support du système ;
- certains programmes ne marchent pas.

Méthode en cours d'adoption : OpenBSD (intégré, W^X), Linux (patch PAX) sur AMD64, alpha, sparc, etc..

Solution 2: rendre la pile non exécutable

Avantages :

- pratiquement pas de pertes de performances ;
- ne nécessite pas de recompilation ;

Inconvénients :

- nécessite un support du processeur ;
- nécessite un support du système ;
- certains programmes ne marchent pas.

Méthode en cours d'adoption : OpenBSD (intégré, WˆX), Linux (patch PAX) sur AMD64, alpha, sparc, etc..

Solution 2: rendre la pile non exécutable

Avantages :

- pratiquement pas de pertes de performances ;
- ne nécessite pas de recompilation ;

Inconvénients :

- nécessite un support du processeur ;
- nécessite un support du système ;
- certains programmes ne marchent pas.

Méthode en cours d'adoption : OpenBSD (intégré, W^X), Linux (patch PAX) sur AMD64, alpha, sparc, etc..

"Et si on utilise un langage évolué ?"

Les langages "évolués" surveillent la mémoire pour le programmeur :

- Perl : modifie la taille des tableaux dynamiquement ;
- Caml, Java : exceptions en cas de dépassement.

Le risque est donc faible !

Attention !

Des bugs dans les compilateurs réintroduisent parfois de telles failles !

"Et si on utilise un langage évolué ?"

Les langages "évolués" surveillent la mémoire pour le programmeur :

- Perl : modifie la taille des tableaux dynamiquement ;
- Caml, Java : exceptions en cas de dépassement.

Le risque est donc faible !

Attention !

Des bugs dans les compilateurs réintroduisent parfois de telles failles !

format string vulnerabilities

- Problème identifié récemment : 1999 ;
- dans les chaînes de format de fonctions comme *printf*(3) ;
- se pose si l'attaquant peut contrôler la chaîne de format.

Rappel :

- %x permet d'afficher un type entier en hexadécimal ;
- ajouter un nombre comme dans %12x permet de spécifier le nombre de caractères imprimés ;
- dans *printf("foobar%n", &i)*, *%n* stocke le nombre de caractères imprimés (jusqu'ici) dans *i*.

format string vulnerabilities

- Problème identifié récemment : 1999 ;
- dans les chaînes de format de fonctions comme *printf*(3) ;
- se pose si l'attaquant peut contrôler la chaîne de format.

Rappel :

- *%x* permet d'afficher un type entier en hexadécimal ;
- ajouter un nombre comme dans %12*x* permet de spécifier le nombre de caractères imprimés ;
- dans *printf("foobar%n", &i)*, %*n* stocke le nombre de caractères imprimés (jusqu'ici) dans *i*.

format string vulnerabilities

- Problème identifié récemment : 1999 ;
- dans les chaînes de format de fonctions comme *printf*(3) ;
- se pose si l'attaquant peut contrôler la chaîne de format.

Rappel :

- *%x* permet d'afficher un type entier en hexadécimal ;
- ajouter un nombre comme dans %12*x* permet de spécifier le nombre de caractères imprimés ;
- dans *printf("foobar%n", &i)*, *%n* stocke le nombre de caractères imprimés (jusqu'ici) dans *i*.

format string vulnerabilities

- Problème identifié récemment : 1999 ;
- dans les chaînes de format de fonctions comme *printf*(3) ;
- se pose si l’attaquant peut contrôler la chaîne de format.

Rappel :

- *%x* permet d’afficher un type entier en hexadécimal ;
- ajouter un nombre comme dans %12*x* permet de spécifier le nombre de caractères imprimés ;
- dans *printf("foobar%n", &i)*, %*n* stocke le nombre de caractères imprimés (jusqu’ici) dans *i*.

Comment fonctionne *printf*(3) ?

printf

- parcourt la chaîne de gauche à droite ;
- à chaque '%', on regarde le caractère suivant et on interprète l'argument suivant sur la pile en conséquence.

Écrire en mémoire

Que fait *printf*(" *ABCD%08x%08x%n*"); ?

Principe :

- utiliser des %*x* pour remonter jusqu'à l'emplacement du début de la chaîne sur la pile ;
- mettre une adresse *ABCD* au début de la chaîne ;
- %*n* interprète alors cette adresse comme l'endroit où il doit écrire...

Écrire en mémoire

Que fait *printf*(" *ABCD*%08*x*%08*x*%*n*"); ?

Principe :

- utiliser des %*x* pour remonter jusqu'à l'emplacement du début de la chaîne sur la pile ;
- mettre une adresse *ABCD* au début de la chaîne ;
- %*n* interprète alors cette adresse comme l'endroit où il doit écrire...

Écrire en mémoire

Que fait *printf*(" *ABCD*%08*x*%08*x*%*n*"); ?

Principe :

- utiliser des %*x* pour remonter jusqu'à l'emplacement du début de la chaîne sur la pile ;
- mettre une adresse *ABCD* au début de la chaîne ;
- %*n* interprète alors cette adresse comme l'endroit où il doit écrire...

Écrire en mémoire

Que fait *printf*(" *ABCD*%08*x*%08*x*%*n*"); ?
Principe :

- utiliser des %*x* pour remonter jusqu'à l'emplacement du début de la chaîne sur la pile ;
- mettre une adresse *ABCD* au début de la chaîne ;
- %*n* interprète alors cette adresse comme l'endroit où il doit écrire...

Retour dans *libc*...

. . . ou "que faire quand on ne peut pas exécuter la pile ?"

L'idée est d'écrire dans la *Global Offset Table*.

dnprintf(buf, *n*, data);
file = fopen(buf, "r");

- remplacer le pointer sur *fopen*(3) par un pointeur sur *system*(3) ;
- mettre en début de chaîne la commande à exécuter, par exemple:
  cd /tmp; cp /bin/sh; chmod 4777 sh; exit;ABCD%08x.

Retour dans *libc*...

…ou “que faire quand on ne peut pas exécuter la pile ?”

L'idée est d'écrire dans la *Global Offset Table*.

dnprintf(buf, *n*, data);
file = fopen(buf, “r”):

- remplacer le pointer sur *fopen*(3) par un pointeur sur *system*(3) ;
- mettre en début de chaîne la commande à exécuter, par exemple:
  cd /tmp; cp /bin/sh; chmod 4777 sh; exit;ABCD%08x.

Retour dans *libc*...

. . . ou "que faire quand on ne peut pas exécuter la pile ?"

L'idée est d'écrire dans la *Global Offset Table*.

dnprintf(buf, *n*, data);
file = fopen(buf, "r"):

- remplacer le pointer sur *fopen*(3) par un pointeur sur *system*(3) ;
- mettre en début de chaîne la commande à exécuter, par exemple:
  cd /tmp; cp /bin/sh; chmod 4777 sh; exit;ABCD%08x.

Retour dans *libc*...

. . . ou “que faire quand on ne peut pas exécuter la pile ?”

L'idée est d'écrire dans la *Global Offset Table*.

dnprintf(buf, *n*, data);
file = fopen(buf, “r”):

- remplacer le pointer sur *fopen*(3) par un pointeur sur *system*(3) ;
- mettre en début de chaîne la commande à exécuter, par exemple:
  cd /tmp; cp /bin/sh; chmod 4777 sh; exit;ABCD%08x.

les *race conditions* par l'exemple

Contexte :

- utilisateur *lambda* ($\Rightarrow$ uid) ;
- programme setuid *root* ($\Rightarrow$ euid) ;
- le programme doit modifier un fichier de l'utilisateur.

les *race conditions* par l'exemple

Contexte :

- utilisateur *lambda* ($\Rightarrow$ uid) ;
- programme setuid *root* ($\Rightarrow$ euid) ;
- le programme doit modifier un fichier de l'utilisateur.

Dans le programme :

```
if (access(filename, W_OK) == 0) {
  if ((fd = open(filename, O_WRONLY)) == NULL) {
    perror(filename);
    return(0);
  }
  /* écrire dans le fichier */
```

les *race conditions* par l'exemple

Attaque :

- lambda stoppe le processus entre l'*access*(2) et le *open*(2) ;
- lambda remplace le fichier *filename* par un lien symbolique sur un autre fichier ;
- lambda relance le processus, qui ouvrira avec succès le fichier en écriture.

Solutions (?) :

les *race conditions* par l'exemple

Attaque :

- lambda stoppe le processus entre l'*access*(2) et le *open*(2) ;
- lambda remplace le fichier *filename* par un lien symbolique sur un autre fichier ;
- lambda relance le processus, qui ouvrira avec succès le fichier en écriture.

Solutions (?) :

les *race conditions* par l'exemple

Attaque :

- lambda stoppe le processus entre l'*access*(2) et le *open*(2) ;
- lambda remplace le fichier *filename* par un lien symbolique sur un autre fichier ;
- lambda relance le processus, qui ouvrira avec succès le fichier en écriture.

Solutions (?) :

les *race conditions* par l'exemple

Attaque :

- lambda stoppe le processus entre l'*access*(2) et le *open*(2) ;
- lambda remplace le fichier *filename* par un lien symbolique sur un autre fichier ;
- lambda relance le processus, qui ouvrira avec succès le fichier en écriture.

Solutions (?) :

les *race conditions* par l'exemple

Attaque :

- lambda stoppe le processus entre l'*access*(2) et le *open*(2) ;
- lambda remplace le fichier *filename* par un lien symbolique sur un autre fichier ;
- lambda relance le processus, qui ouvrira avec succès le fichier en écriture.

Solutions (?) :

- inverser l'ordre de access et de open.

les *race conditions* par l'exemple

Attaque :

- lambda stoppe le processus entre l'*access*(2) et le *open*(2) ;
- lambda remplace le fichier *filename* par un lien symbolique sur un autre fichier ;
- lambda relance le processus, qui ouvrira avec succès le fichier en écriture.

Solutions (?) :

- inverser l'ordre de access et de open. **NE CHANGE RIEN !**

les *race conditions* par l'exemple

Attaque :

- lambda stoppe le processus entre l'*access*(2) et le *open*(2) ;
- lambda remplace le fichier *filename* par un lien symbolique sur un autre fichier ;
- lambda relance le processus, qui ouvrira avec succès le fichier en écriture.

Solutions (?) :

- utiliser un *access* qui fonctionne sur les *file descriptors*.

les *race conditions* par l'exemple

Attaque :

- lambda stoppe le processus entre l'*access*(2) et le *open*(2) ;
- lambda remplace le fichier *filename* par un lien symbolique sur un autre fichier ;
- lambda relance le processus, qui ouvrira avec succès le fichier en écriture.

Solutions (?) :

- utiliser un *access* qui fonctionne sur les *file descriptors*.
  $\Longrightarrow$ Nécessite un changement dans le noyau...

les *race conditions* par l'exemple

Attaque :

- lambda stoppe le processus entre l'*access*(2) et le *open*(2) ;
- lambda remplace le fichier *filename* par un lien symbolique sur un autre fichier ;
- lambda relance le processus, qui ouvrira avec succès le fichier en écriture.

Solutions (?) :

- utiliser un *access* qui fonctionne sur les *file descriptors*.
  $\Longrightarrow$ Nécessite un changement dans le noyau...
- abandonner les privilèges root avant le *open*.

les *race conditions* par l'exemple

Attaque :

- lambda stoppe le processus entre l'*access*(2) et le *open*(2) ;
- lambda remplace le fichier *filename* par un lien symbolique sur un autre fichier ;
- lambda relance le processus, qui ouvrira avec succès le fichier en écriture.

Solutions (?) :

- utiliser un *access* qui fonctionne sur les *file descriptors*.
  $\Longrightarrow$ Nécessite un changement dans le noyau...
- abandonner les privilèges root avant le *open*.
  $\Longrightarrow$ Peut nécessiter des modifications de l'architecture du programme.

SQL Injections

- SQL est un langage pour interroger des bases de données ;
- Une application (ex : page web dynamique) peut l'utiliser pour accéder à une base de données.

Imaginons que la requête soit faite ainsi :

```
$res = request_sql("SELECT login FROM users WHERE login = $login AND password = $pass");
```

SQL Injections

- SQL est un langage pour interroger des bases de données ;
- Une application (ex : page web dynamique) peut l'utiliser pour accéder à une base de données.

Imaginons que la requête soit faite ainsi :

```
$res = request_sql("SELECT login FROM users WHERE login = $login AND password = $pass");
```

SQL Injections

- SQL est un langage pour interroger des bases de données ;
- Une application (ex : page web dynamique) peut l'utiliser pour accéder à une base de données.

Imaginons que la requête soit faite ainsi :

```
$res = request_sql("SELECT login FROM users WHERE login = $login AND password = $pass");
```

Un utilisateur malicieux entre comme mot de passe `toto OR 1=1` !

SQL Injections

- SQL est un langage pour interroger des bases de données ;
- Une application (ex : page web dynamique) peut l'utiliser pour accéder à une base de données.

Imaginons que la requête soit faite ainsi :

```
$res = request_sql("SELECT login FROM users WHERE login = $login AND password = $pass");
```

Un utilisateur malicieux entre comme mot de passe `toto OR 1=1` !

Moralité : **Ne jamais faire confiance aux données externes !**

Sommaire

1. Présentation Générale
2. Quelques grands principes de sécurité
3. Problèmes logiciels
4. Conclusion
   - Questions ?
   - Bibliographie

« Pense d'abord, utilise l'ordinateur ensuite. »
– Ian Stewart

« Réfléchir beaucoup l'emporte sur réfléchir peu, et plus encore sur ne pas réfléchir du tout. »
– Sun Zi

- **Pas de sécurité absolue** ;
- programmes sûrs : condition nécessaire ;
- mesures simples $\Longrightarrow$ attaques beaucoup plus difficiles ;

« Se faire battre est excusable, se faire surprendre impardonnable. »
– Napoléon

« Pense d'abord, utilise l'ordinateur ensuite. »
– Ian Stewart

« Réfléchir beaucoup l'emporte sur réfléchir peu, et plus encore sur ne pas réfléchir du tout. »
– Sun Zi

- **Pas de sécurité absolue** ;
- programmes sûrs : condition nécessaire ;
- mesures simples $\Longrightarrow$ attaques beaucoup plus difficiles ;

« Se faire battre est excusable, se faire surprendre impardonnable. »
– Napoléon

« Pense d'abord, utilise l'ordinateur ensuite. »
– Ian Stewart

« Réfléchir beaucoup l'emporte sur réfléchir peu, et plus encore sur ne pas réfléchir du tout. »
– Sun Zi

- **Pas de sécurité absolue** ;
- programmes sûrs : condition nécessaire ;
- mesures simples $\Longrightarrow$ attaques beaucoup plus difficiles ;

« Se faire battre est excusable, se faire surprendre impardonnable. »
– Napoléon

« Pense d'abord, utilise l'ordinateur ensuite. »
– Ian Stewart

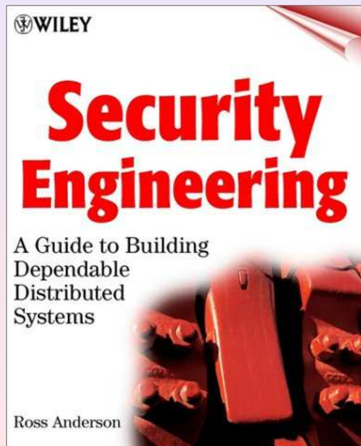
« Réfléchir beaucoup l'emporte sur réfléchir peu, et plus encore sur ne pas réfléchir du tout. »
– Sun Zi

- **Pas de sécurité absolue** ;
- programmes sûrs : condition nécessaire ;
- mesures simples $\Longrightarrow$ attaques beaucoup plus difficiles ;

« Se faire battre est excusable, se faire surprendre impardonnable. »
– Napoléon

Des questions ?

Quelques références bibliographiques. . .

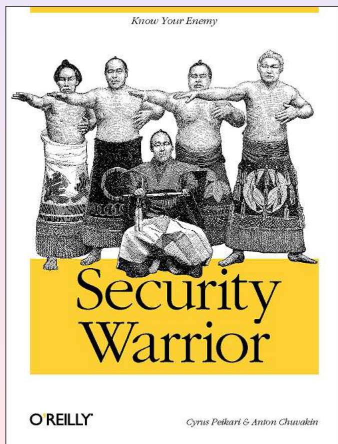


Ross ANDERSON.
Security Engineering.
Wiley Computer Publishing. John Wiley & Sons, Inc, 2001.

Quelques références bibliographiques. . .

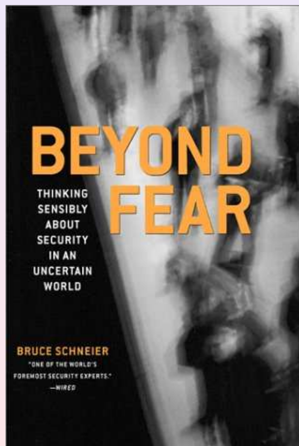
Aleph1.
Smashing the stack for fun and profit.
Phrack, 7(49), 1996.
L'article qui a exposé au grand jour le Buffer Overflow.

Quelques références bibliographiques. . .



Cyrus Peikari and Anton Chuvakin.
Security Warrior.
O'Reilly, 2004.
Couvre superficiellement un large panorama de méthodes d'attaques.

Quelques références bibliographiques. . .



Bruce SCHNEIER.
Beyond Fear, Thinking Sensibly about Security in an Uncertain World.
Copernicus Books, 2003.
L'expérience de la sécurité informatique appliquée au monde réel...

Quelques autres références.... . .

BULBA and KIL3R.
Bypassing stackguard and stackshield.
Phrack, 10(56), 2000.

Peter VAN DER LINDEN.
Expert C Programming – Deep C Secrets.
SunSoft Press / Prentice Hall, 1994.

David A. Wheeler.
Secure Programming for Linux and Unix HOWTO, 3.010 edition, Mars 2003.
http://www.dwheeler.com/secure-programs.