

Plan de la section

7 Introduction

- Règles du jeux
- Vulnérabilités cryptographiques
- Vulnérabilités logiques
- Manipulation des chaînes de caractères

8 Attaque

9 OS Hardening

7 Introduction

- Règles du jeux
- Vulnérabilités cryptographiques
- Vulnérabilités logiques
- Manipulation des chaînes de caractères

8 Attaque

- Injections
- Corruption mémoire
- Programmation sécurisée

9 OS Hardening

- La part de l'administrateur
- Protections

Oublier tout ce que vous savez !

Postulats de base

- Le développeur a forcément laissé des bugs ;
 - Il ne peut pas avoir tout prévu
 - Il ne pense qu'au déroulement normal
- L'attaquant n'en a besoin que d'un seul !
 - Aucune affirmation n'est vraie, tout doit être démontré
 - Réflexion inverse
 - Tout est une question de temps.

Oublier tout ce que vous savez !

Postulats de base

- Le développeur a forcément laissé des bugs ;
 - Il ne peut pas avoir tout prévu
 - Il ne pense qu'au déroulement normal
- L'attaquant n'en a besoin que d'un seul !
 - Aucune affirmation n'est vraie, tout doit être démontré
 - Réflexion inverse
 - Tout est une question de temps.

Oublier tout ce que vous savez !

Postulats de base

- Le développeur a forcément laissé des bugs ;
 - Il ne peut pas avoir tout prévu
 - Il ne pense qu'au déroulement normal
- L'attaquant n'en a besoin que d'un seul !
 - Aucune affirmation n'est vraie, tout doit être démontré
 - Réflexion inverse
 - Tout est une question de temps.

Définitions

- Un **bug** est un comportement ou un résultat incorrect.
- Une **vulnérabilité** est un *bug* ayant un impact sur la sécurité du système d'information.
 - Son risque peut être minime ou critique ;
 - Il peut-être exploitable ou non.
- Un **exploit** est un programme utilisant une *vulnérabilité* pour enfreindre une règle de sécurité.
- Un **risque** est l'appréciation d'une *vulnérabilité* en fonction de sa probabilité d'exploitation et de la gravité de l'impact.

Quel est l'impact d'une vulnérabilité ?

- *Denial of Service* : le programme plante ;
- Exécution de code arbitraire ;
- Fuite d'information (fichier, privilège, clefs, tc.) ;
- Contournement d'authentification.

La gravité dépend de l'environnement.

Quel est l'impact d'une vulnérabilité ?

- *Denial of Service* : le programme plante ;
- Exécution de code arbitraire ;
- Fuite d'information (fichier, privilège, clefs, tc.) ;
- Contournement d'authentification.

La gravité dépend de l'environnement.

Algorithmique

Les algorithmes sont optimisées pour avoir la complexité la plus faible possible, parfois en se basant sur des hypothèses de fonctionnement normal.

Données issues de l'attaquant

En contrôlant les données en entrée, l'attaquant peut déclencher le pire cas possible.

⇒ DoS

Propriété

Une table de hashage est une structure de données permettant l'insertion et la lecture efficace d'un item.
⇒ Utilisation d'une fonction de hashage

L'efficacité d'une table de hashage se montre donc dans une distribution uniforme des items.

Par définition, il y aura toujours des collisions. Des attaques ont été montées contre :

- Bro : 78% de perte de paquet à 160 Kb/s
- Squid : ajout de 1.7 ms de latence par accès
- Perl : explosion exponentielle du temps de traitement

Propriété

Une table de hashage est une structure de données permettant l'insertion et la lecture efficace d'un item.
⇒ Utilisation d'une fonction de hashage

L'efficacité d'une table de hashage se montre donc dans une distribution uniforme des items.

Par définition, il y aura toujours des collisions. Des attaques ont été montées contre :

- Bro : 78% de perte de paquet à 160 Kb/s
- Squid : ajout de 1.7 ms de latence par accès
- Perl : explosion exponentielle du temps de traitement

La puissance des regexps est sous-estimée :

- Capacité de backtracking (utilisation des données matchées pour construire l'expression)
- Implémentation des moteurs déterministes ou non.
- Possibilité d'exécution de code

CVE-2009-3695 : Vulnérabilité par les données validées

```
1 - r ')@(?:[A-Z0-9]+(?:-*[A-Z0-9]+)*\.)+[A-Z]{2,6}$'  
2 + r ')@(?:[A-Z0-9](?:[A-Z0-9-]{0,61}[A-Z0-9])?\.\.)+[A-Z]{2,6}\.?$_'
```

Pour chaque caractère après @, on regarde si l'expression `(.*+ matche.`

Exploitation d'une attaque logique

7 Introduction

- Règles du jeux
- **Vulnérabilités cryptographiques**
- Vulnérabilités logiques
- Manipulation des chaînes de caractères

8 Attaque

- Injections
- Corruption mémoire
- Programmation sécurisée

9 OS Hardening

- La part de l'administrateur
- Protections

Une affaire d'experts

La crypto est le domaine le plus compliqué existant.

⇒ « *Le diable est dans le détail* » – Ben Laurie

- Si vous développez vous même votre système de cryptographie, c'est que vous êtes dans l'erreur.
- L'utilisation de systèmes crypto existants, libssl, n'est pas triviale non plus !

Une affaire d'experts

La crypto est le domaine le plus compliqué existant.

⇒ « *Le diable est dans le détail* » – Ben Laurie

- Si vous développez vous même votre système de cryptographie, c'est que vous êtes dans l'erreur.
 - Chiffrement ;
 - Signature ;
 - Intégrité.
- L'utilisation de systèmes crypto existants, libssl, n'est pas triviale non plus !

Une affaire d'experts

La crypto est le domaine le plus compliqué existant.

⇒ « *Le diable est dans le détail* » – Ben Laurie

- Si vous développez vous même votre système de cryptographie, c'est que vous êtes dans l'erreur.
- L'utilisation de systèmes crypto existants, libssl, n'est pas triviale non plus !
 - Mauvaise vérification de signature ;
 - Utilisation de mauvaise entropie ;
 - *Downgrading* d'algorithmes ;
 - Négligence des pré-requis (WEP).

Mauvais aléa

```
void gen_prime(Big_int* p, int size)
{
    int i;
    int rand_buf[1024];
    srand(time(NULL));
    for (i=0;i<size/32)
        rand_buf[i] = rand();

    get_next_prime(p, rand_buf);
    return;
}
```

- On a besoin d'une clef RSA 2048 bits
- Mais elle est générée en utilisant `gen_prime()`
- Le générateur d'aléa a une graine de 32 bits.
- La clef peut être devinée après 2^{32} essais au lieu de 2^{1024}

7 Introduction

- Règles du jeux
- Vulnérabilités cryptographiques
- Vulnérabilités logiques**
- Manipulation des chaînes de caractères

8 Attaque

- Injections
- Corruption mémoire
- Programmation sécurisée

9 OS Hardening

- La part de l'administrateur
- Protections

Hypothèses du développeur fausses.

- Taille de la ligne de commande limitée ;
- Vérification du certificat mais pas de la date d'expiration ;
- Confondre autorisation et authentification ;
- Être trop confiant sur le cercle de confiance.

time-of-check-to-time-of-use, TOCTTOU

C'est faire l'hypothèse qu'une ressource ne sera pas modifiée entre sa vérification et son utilisation.

(Mauvaises) Hypothèses :

- Une séquence d'instruction n'est pas atomique
- Une ligne de code source n'est pas atomique
- Le flot de programme peut être ralenti ou interrompu par un attaquant.

Vulnérable

```
1 st=os.stat('/tmp/toto')
2 if st.st_uid != os.getuid():
3     raise Exception('Erreur')
4 fd=open('/tmp/toto')
5 ...
```

Entre la ligne 1 et 4, le fichier a pu être remplacé.
⇒ Toujours travailler sur des objets « figés »

Version correcte

```
1 fd = open('/tmp/toto')
2 st = os.fstat(fd.fileno())
3 if st.st_uid != os.getuid():
4     fd.close()
5     raise Exception('Erreur')
```

Vulnérable

```
1 st=os.stat('/tmp/toto')
2 if st.st_uid != os.getuid():
3     raise Exception('Erreur')
4 fd=open('/tmp/toto')
5 ...
```

Entre la ligne 1 et 4, le fichier a pu être remplacé.
⇒ Toujours travailler sur des objets « figés »

Version correcte

```
1 fd = open('/tmp/toto')
2 st = os.fstat(fd.fileno())
3 if st.st_uid != os.getuid():
4     fd.close()
5     raise Exception('Erreur')
```

Pourquoi les scripts ne peuvent être setuid ?

Il est impossible pour le noyau de vérifier que le script initial sera bien celui qui sera exécuté par l'interpréteur.

```
1 userspace>execve("./myprogram.py", NULL, NULL);
2 ... kernel> f = open('myprogram.py')
3 ... kernel> if f is a script:
4 ... kernel>     execve('/usr/bin/python', ['myprogram.py'])
```

```
1 tmp = open('/tmp/myprog.%d' % os.getpid())
2 ...
```

Le PID est devinable donc un lien symbolique peut-être créé.

Un lien symbolique peut pointer :

- Nulle part
- Sur un fichier avec un uid/gid différent

Imaginons que le programme suivant tourne avec des priviléges élevés :

```
1 filename = '/home/' + username + '/.myconfigfile'
2 fd = open(filename)
3 ...
```

Si l'utilisateur crée un lien symbolique \$HOME/.myconfigfile pointant sur /etc/shadow. Il peut désormais lire et écrire dedans.

Un lien symbolique peut pointer :

- Nulle part
- Sur un fichier avec un uid/gid différent

Imaginons que le programme suivant tourne avec des privilèges élévés :

```
1 filename = '/home/' + username + '/.myconfigfile'
2 fd = open(filename)
3 ...
```

Si l'utilisateur créé un lien symbolique \$HOME/.myconfigfile pointant sur /etc/shadow. Il peut désormais lire et écrire dedans.

Page de manuel d'OpenBSD

« Most functions not in the above lists are **considered to be unsafe** with respect to signals. That is to say, the behaviour of such functions when called from a signal handler is **undefined**. »

Sendmail

```
1 void sighndl1r(int dummy) {
2     syslog(LOG_NOTICE, user_dependent_data);
3     free(global_ptr2);
4     free(global_ptr1);
5     exit(0);
6 }
```

7 Introduction

- Règles du jeux
- Vulnérabilités cryptographiques
- Vulnérabilités logiques
- Manipulation des chaînes de caractères

8 Attaque

- Injections
- Corruption mémoire
- Programmation sécurisée

9 OS Hardening

- La part de l'administrateur
- Protections

Définition

En langage C, une chaîne de caractères n'est qu'un ensemble de caractères terminé par NULL.

⇒ Il n'y a pas de notion de taille !

Cette (non-)propriété a de grave conséquences sur la sûreté des programmes. Et même sur les langages de haut-niveau...

```
1 char buf[512];
2 sprintf(buf, sizeof buf, "/var/app/%s.txt", username);
3 int fd = open(buf, O_RDONLY);
```

On prend l'hypothèse que `username` est contrôlé par l'utilisateur.

- Est-ce que `username` peut contenir le caractère NULL ?
- Et si `username` vaut `../../../../etc/passwd\x00` ?
- Et si `username` a une longueur de `512 - len("/var/app/")` octets ?

```
1 char buf[512];
2 sprintf(buf, sizeof buf, "/var/app/%s.txt", username);
3 int fd = open(buf, O_RDONLY);
```

On prend l'hypothèse que `username` est contrôlé par l'utilisateur.

- Est-ce que `username` peut contenir le caractère NULL ?
- Et si `username` vaut `../../../../etc/passwd\x00` ?
- Et si `username` a une longueur de `512 - len("/var/app/")` octets ?

```
1 char buf[512];
2 sprintf(buf, sizeof buf, "/var/app/%s.txt", username);
3 int fd = open(buf, O_RDONLY);
```

On prend l'hypothèse que `username` est contrôlé par l'utilisateur.

- Est-ce que `username` peut contenir le caractère NULL ?
- Et si `username` vaut `../../../../etc/passwd\x00` ?
- Et si `username` a une longueur de `512 - len("/var/app/")` octets ?

```
1 char buf[512];
2 snprintf(buf, sizeof buf, "/var/app/%s.txt", username);
3 int fd = open(buf, O_RDONLY);
```

On prend l'hypothèse que `username` est contrôlé par l'utilisateur.

- Est-ce que `username` peut contenir le caractère NULL ?
- Et si `username` vaut `../../../../etc/passwd\x00` ?
- Et si `username` a une longueur de `512 - len("/var/app/")` octets ?

Code supposé retirer tous ../ dans une chaîne de caractères :

```
1 i=0
2 while i+3 <= len(buf):
3     if buf[i] == '..' and buf[i+1] == '.' and buf[i+2] == '/':
4         buf = buf[:i] + buf[i+3:]
5     i += 1
```

Et si *buf* == "...//"

Code supposé retirer tous ../ dans une chaîne de caractères :

```
1 i=0
2 while i+3 <= len(buf):
3     if buf[i] == '..' and buf[i+1] == '.' and buf[i+2] == '/':
4         buf = buf[:i] + buf[i+3:]
5     i += 1
```

Et si *buf* == "...//"?

Code supposé retirer tous ../ dans une chaîne de caractères :

```
1 i=0
2 while i+3 <= len(buf):
3     if buf[i] == '..' and buf[i+1] == '.' and buf[i+2] == '/':
4         buf = buf[:i] + buf[i+3:]
5     i += 1
```

Et si *buf* == ".../..." ?

Dépassemement de tampon

Écriture en dehors de la zone allouée.

- Dans la pile (*stack*) ;
- Dans le tas (*heap*) ;
- N'importe où !

Plan de la section

7 Introduction

8 Attaque

- Injections
- Corruption mémoire
- Programmation sécurisée

9 OS Hardening

7 Introduction

- Règles du jeux
- Vulnérabilités cryptographiques
- Vulnérabilités logiques
- Manipulation des chaînes de caractères

8 Attaque

■ Injections

- Corruption mémoire
- Programmation sécurisée

9 OS Hardening

- La part de l'administrateur
- Protections

```
1  <?php
2      $color = 'blue';
3      if (!isset($_GET['COLOR'])) {
4          $color = $_GET['COLOR'];
5      require($color . '.php');
6  ?>
```

```
1 $myvar = 'somevalue';           1 foreach ($_GET as $key => $value) {  
2 $x = $_GET['arg'];           2   $$key = $value;  
3 eval('$myvar=' . $x . ';');  3 }
```

```
1 <?php
2     $user=$_GET[ 'username '];
3     $pass=$_GET[ 'password '];
4
5     $req = "select password from user"
6         . "where user=\"" . $user . "\"";
7     mysql_query($req , $ctx);
8 ?>
```

- \$user == "toto\" or 1=1 --"
- \$user == "toto\"; drop database user; --"

```
1 <?php
2     $user=$_GET[ 'username '];
3     $pass=$_GET[ 'password '];
4
5     $req = "select password from user"
6         . "where user=\"" . $user . "\"";
7     mysql_query($req , $ctx);
8 ?>
```

- \$user == "toto\" or 1=1 --"
- \$user == "toto\"; drop database user; --"

man 3 system

system() executes a command specified in command by calling
/bin/sh -c command

Extrait d'un firmware Linksys

```
1 system("ping $addr");  
      ⇒ addr == "1.2.3.4; rm -fr /"
```

man 3 system

system() executes a command specified in command by calling
/bin/sh -c command

Extrait d'un firmware Linksys

```
1 system("ping $addr");  
  
⇒ addr == "1.2.3.4; rm -fr /"
```

Deux méthodes de protection

- Liste blanche de tous les caractères autorisés
- Liste noire

Caractères dangereux

:	>	'	*	("	"	^	"\\\"	"\\r"
;	&	!	/)	[~	,	"\\n"	
<		-	?	.]	\	"		\$

Vraie solution

Utilisation directe de execve()

Deux méthodes de protection

- Liste blanche de tous les caractères autorisés
- Liste noire

Caractères dangereux

:	>	'	*	("	"	^	"\\\"	"\\r"
;	&	!	/)	[~	,	"\\n"	
<		-	?	.]	\	"		\$

Vraie solution

Utilisation directe de execve()

7 Introduction

- Règles du jeux
- Vulnérabilités cryptographiques
- Vulnérabilités logiques
- Manipulation des chaînes de caractères

8 Attaque

- Injections
- **Corruption mémoire**
- Programmation sécurisée

9 OS Hardening

- La part de l'administrateur
- Protections

Assembleur 101

- Registres sur x86 :
 - Généraux : eax, ebx, ecx, edx
 - Segment : cs, ds, es, ss
 - Pointeurs : eip, ebp, esp, esi, edi
 - Registres spéciaux (flottants, debug, MMX...)
- Registres intéressants :
 - eip pointe sur l'instruction courante ;
 - ebp pointe sur le haut de la pile ;
 - esp pointe sur le bas de la pile ;
- Instructions spéciales :
 - push : empile la valeur et décrémente esp ;
 - pop dépile la valeur et incrémente esp ;
 - sub y, esp : alloue de la mémoire sur la pile ;
 - add y, esp : libère de la mémoire.

Assembleur 101

- Registres sur x86 :
 - Généraux : eax, ebx, ecx, edx
 - Segment : cs, ds, es, ss
 - Pointeurs : eip, ebp, esp, esi, edi
 - Registres spéciaux (flottants, debug, MMX...)
- Registres intéressants :
 - eip pointe sur l'instruction courante ;
 - ebp pointe sur le haut de la pile ;
 - esp pointe sur le bas de la pile ;
- Instructions spéciales :
 - push : empile la valeur et décrémente esp ;
 - pop dépile la valeur et incrémente esp ;
 - sub y, esp : alloue de la mémoire sur la pile ;
 - add y, esp : libère de la mémoire.

Assembleur 101

- Registres sur x86 :
 - Généraux : eax, ebx, ecx, edx
 - Segment : cs, ds, es, ss
 - Pointeurs : eip, ebp, esp, esi, edi
 - Registres spéciaux (flottants, debug, MMX...)
- Registres intéressants :
 - eip pointe sur l'instruction courante ;
 - ebp pointe sur le haut de la pile ;
 - esp pointe sur le bas de la pile ;
- Instructions spéciales :
 - push : empile la valeur et décrémente esp ;
 - pop dépile la valeur et incrémente esp ;
 - sub y, esp : alloue de la mémoire sur la pile ;
 - add y, esp : libère de la mémoire.

Appel de fonction

```
1 call myfunc
2     prologue
3         code de la fonction
4     epilogue
5     ret
6     ...
```

Prologue

```
1 push ebp      ;; Creer une nouvelle stack frame
2 mov ebp, esp  ;; Installer la nouvelle stack frame
```

Épilogue

```
1 mov esp, ebp
2 pop ebp       ;; Restaure la stack frame
```

Buffer overflow

Un *overflow* est un débordement de tableau. Par exemple, une chaîne de caractère qui est copiée au delà de sa zone allouée.

Cela arrive quand :

- On utilise des fonctions non-sûre telles que `strcpy`, `sprintf` car elles n'ont pas la notion de taille maximale.
- Problème arithmétique : on calcule mal la longueur d'une chaîne de caractères.
- Problème d'algorithme qui boucle.

En fonction de la nature du buffer, on distingue plusieurs types de débordement :

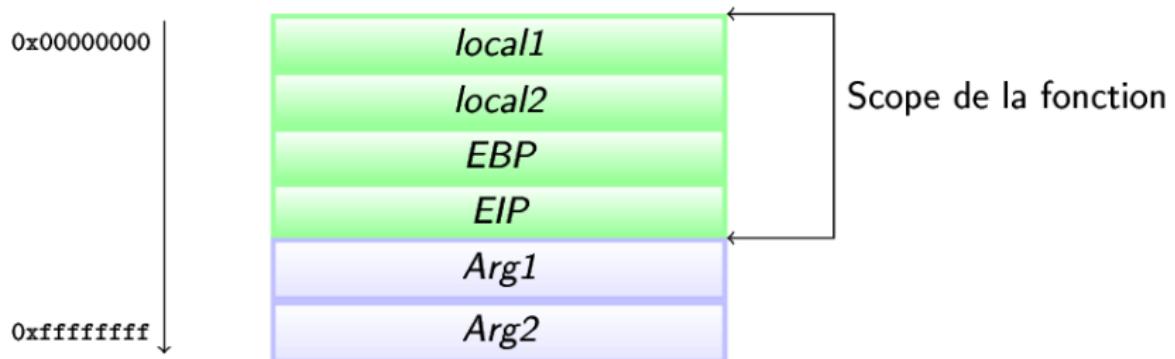
- Stack overflow : variables locales
- Heap overflow : variables dynamique (`malloc()`)
- Le reste.

```
1 char buf[128];
2 strcpy(buf, argv[1]);
```

Ou à la main :

```
1 char buf[128];
2 int len = atoi(argv[1]);
3 for (int i = 0 ; i < len ; i++)
4     buf[i] = 'A';
```

Stack overflow



Stack overflow



Le but du jeu est de remplacer l'adresse de retour (*saved eip*) pour pointer sur une zone contrôlée par l'attaquant.

Old school style

- Buffer en train de déborder ;
- Fausse *stack frame* et on appelle une fonction de bibliothèque ;
- `call esp.`

```
1  char *buf = malloc(12);
2  if (buf == NULL)
3      return 1;
4  free(buf);
```

Il y a un système complexe de gestion de la mémoire derrière ces deux fonctions :

- Wrapper autour de `brk()` et `mmap()` ;
- Pools de *chunks* prêt à l'usage ;
- Optimisation pour certaines tailles (pour éviter la fragmentation) ;
- Compromis rapidité/*overhead*.

Pour sa gestion, `dlmalloc()` utilise :

- Des variables globales ;
- Chaque *chunk* est constitué d'une structure de contrôle de la mémoire allouée.

Pour simplifier, on peut la considérer comme une liste doublement chaînée.

Pour sa gestion, `dlmalloc()` utilise :

- Des variables globales ;
- Chaque *chunk* est constitué d'une structure de contrôle de la mémoire allouée.
Pour simplifier, on peut la considérer comme une liste doublement chaînée.

unlink()

```
1 #define unlink( P, BK, FD ) {  
2     BK = P->bk;  
3     FD = P->fd;  
4     FD->bk = BK;  
5     BK->fd = FD;  
6 }
```

- Ligne 4 : Écrire à *(FD+12) le contenu de BK ;
 - Ligne 5 : BK est écrasé au milieu du *shellcode* ;
⇒ **Écriture de quatre octets n'importe où !**

Heap overflow : exploitation par frontlink

```
1 #define frontlink( A, P, S, IDX, BK, FD ) { \
2     if ( S < MAX_SMALLBIN_SIZE ) { \
3         ... \
4     } else { \
5         IDX = bin_index( S ); \
6         BK = bin_at( A, IDX ); \
7         FD = BK->fd; \
8         if ( FD == BK ) { \
9             mark_binblock(A, IDX); \
10        } else { \
11            while ( FD != BK && S < chunksize(FD) ) { \
12                FD = FD->fd; \
13            } \
14            BK = FD->bk; \
15        } \
16        P->bk = BK; \
17        P->fd = FD; \
18        FD->bk = BK->fd = P; \
19    } \
20 }
```

Double free()

Un double-free() est un programme qui libère deux fois le même pointeur.

- Comportement différent en fonction du status du *chunk*;
- Situation de désynchronisation ;
 - Structure différente entre un *chunk* libre et un en cours d'utilisation
- Contraignant à exploiter.

Erreur fréquence

- Utilisation de mauvaises bornes pour le parcours de tableau ;
- Écriture d'un '\0' trop loin ;
- Utilisation de strncat.

- Pas de contrôle sur la valeur écrasée ;
- Nécessite que le pointeur de pile sauvegardé ait une valeur "haute" ;
- Peut être inexploitable à cause d'optimisation du compilateur ;

Écrasement du pointeur de pile

```
1 mov esp, ebp
2 pop ebp
3 pop eip
```

Exemple :

- Avant : saved_ebp == 0x804d5f4
- Après : saved_ebp == 0x804d500

Si 0x804d500 pointe dans notre buffer, c'est gagné !

Rien d'infini

En C, il n'existe pas de type permettant d'avoir des nombres de tailles « *maléables* ».

⇒ Débordement de capacité

$$0xFFFFFFFF + 1 = 0 \quad (1)$$

$$0x40000020 * 4 = 0x80 \quad (2)$$

$$0 - 1 = 0xFFFFFFFF \quad (3)$$

```
1  unsigned int n;
2  struct req_hdr *hdr, **reqs;
3
4  n = get_number_of_requests(buf);
5  reqs = malloc(n*sizeof(*hdr)); /* integer overflow */
6
7  for (int i = 0; i < n ; i++) { /* on écrit dans le tableau */
8      reqs[i] = foo_bar(...);    /* de taille insuffisante */
9 }
```

```
1 int length = get_from_network(buf);
2 buf = malloc(BUFSIZE);
3 if (length < 0 || length+1 > BUFSIZE)
4     return -ERR;
5 else
6     read(fd, buf, length);
```

Et maintenant si `length = 0x7FFFFFFF` ?

- $\text{length} + 1 \Rightarrow 0x80000000 < 0$

```
1 int length = get_from_network(buf);
2 buf = malloc(BUFSIZE);
3 if (length < 0 || length+1 > BUFSIZE)
4     return -ERR;
5 else
6     read(fd, buf, length);
```

Et maintenant si `length = 0x7FFFFFFF` ?

- $\text{length} + 1 \Rightarrow 0x80000000 < 0$

```
1 int length = get_from_network(buf);
2 buf = malloc(BUFSIZE);
3 if (length < 0 || length+1 > BUFSIZE)
4     return -ERR;
5 else
6     read(fd, buf, length);
```

Et maintenant si `length = 0x7FFFFFFF` ?

- $\text{length} + 1 \Rightarrow 0x80000000 < 0$

Promotion des entiers

Si un type entier est plus petit qu'un int, alors la promotion le converti toujours en entier.

La promotion a lieu partout :

- Opérations (binaire, arithmétique, etc.) ;
- *switch-case*, if ;
- Appel de fonction.

Si deux opérandes sont de signes différents, et que le rang de conversion de l'unsigned est supérieur ou égal à celui du signed, alors le **signed est converti en unsigned**.

```
1 int foo = -1;
2
3 if (foo < sizeof(int)) {
4     /* toujours vrai? */
5 }
```

```
1 int foo = -1;
2
3 if (0xffffffff < sizeof(int)) {
4     /* toujours faux */
5 }
```

Si deux opérandes sont de signes différents, et que le rang de conversion de l'unsigned est supérieur ou égal à celui du signed, alors le **signed est converti en unsigned**.

```
1 int foo = -1;
2
3 if (foo < sizeof(int)) {
4     /* toujours vrai? */
5 }
```

```
1 int foo = -1;
2
3 if (0xffffffff < sizeof(int)) {
4     /* toujours faux */
5 }
```

Le pire outil du C est le *cast*, dans un code propre, on ne devrait pas en trouver.

Valeur	Hexadécimale	En signé	En non signé
0	0x00000000	0	0
12	0x0000000c	12	12
-1	0xFFFFFFFF	-1	4294967295

man 3 printf

```
1 int printf(const char *format, ...);
```

- format est la chaîne de caractères à afficher, avec des séquences spéciales (%c, %d, %s mais aussi %n) ;
- Nombre variable d'arguments.

```
1 char      buffer[512];
2 sprintf (buffer, sizeof (buffer), user);
3 buffer[sizeof (buffer) - 1] = '\0';
```

- `user` est contrôlé par l'attaquant ;
- On peut écrire et lire n'importe où.

7 Introduction

- Règles du jeux
- Vulnérabilités cryptographiques
- Vulnérabilités logiques
- Manipulation des chaînes de caractères

8 Attaque

- Injections
- Corruption mémoire
- **Programmation sécurisée**

9 OS Hardening

- La part de l'administrateur
- Protections

Roadmap

Identifier :

- 1 Les ressources ;
- 2 Les acteurs ;
- 3 Les relations entre les ressources et les acteurs.

Comme dans les réseaux, le cloisonnement est obligatoire.

Compromis

Un algorithme simple peut être préférable à une table de hashage même si les performances ne sont pas les meilleures.

⇒ Savoir relativiser les besoins est important.

Design de qmail

Combattez les idées reçues aussi ! Par exemple, qmail lance un programme uniquement pour le *parsing* d'une adresse mail. Et il fait partie des logiciels passant le mieux à l'échelle.

Compromis

Un algorithme simple peut être préférable à une table de hashage même si les performances ne sont pas les meilleures.

⇒ Savoir relativiser les besoins est important.

Design de qmail

Combattez les idées reçues aussi ! Par exemple, qmail lance un programme uniquement pour le *parsing* d'une adresse mail. Et il fait partie des logiciels passant le mieux à l'échelle.

Chaque module est indépendant des autres et ne fait confiance à personne d'autre, chaque flux entrant et sortant doit être considéré comme non sûr.

En considérant que le module :

- Tourne avec le moins de privilège(s) possible ;
- Cloisonné dans un espace confiné ;
- N'a accès qu'au strict minimum.

L'impact d'une vulnérabilité est diminué.

- Un processus écoutant sur le port 22, avec les droits root ;
- Authentification par un autre processus non privilégié ;
- Exécution d'un nouveau fils sous l'identité de l'utilisateur authentifié.

En cas de plantage...

En fonction du contexte, différentes actions peuvent être utilisées :

- *Fail open* : plus de protection, tout est autorisé
- *Fail closed* : on bloque tout.

Design conservateur

Un design « conservateur » est un design qui a été écrit en se demandant pourquoi les objets sont accessibles plutôt que pourquoi ils ne le sont pas.

⇒ Même principe qu'un firewall !

Exemple : l'authentification UNIX.

Quatre aspects à prendre en compte :

- Les entrées
- Les ressources externes
- Données internes
- Sortie du programme

Quatre aspects à prendre en compte :

- Les entrées
 - Validation des entrées
 - Intolérance aux erreurs
- Les ressources externes
- Données internes
- Sortie du programme

Quatre aspects à prendre en compte :

- Les entrées
- Les ressources externes
 - Effets de bord
 - Vérification des codes de retour
- Données internes
- Sortie du programme

Quatre aspects à prendre en compte :

- Les entrées
- Les ressources externes
- Données internes
 - Stockage des données : listes chaînées, tableaux, etc.
 - Effets de bord, hypothèses.
- Sortie du programme

Quatre aspects à prendre en compte :

- Les entrées
- Les ressources externes
- Données internes
- Sortie du programme
 - Ne pas trop en dire
 - Toujours contrôler ce qui est affiché.

On a vu que certaines fonctions ne permettent pas de traiter de données utilisateur de manière sûre :

- N'ont pas de bornes (`strcpy()`) ;
- Ont un comportement non sûr ((`strcat()`)) ;
- Ont des effets de bords non négligeables.

La première règle est de bannir ces fonctions.

La deuxième est l'emploi de bibliothèques spécialisées.

Tout repose sur l'auteur

Les vulnérabilités proviennent souvent :

- Manque de cohérence ;
- Manque de clarté ;
- Hypothèses invalides ;

Rien ne remplace un programmeur attentif

En théorie, les tests unitaires devraient être écrit avant l'écriture du code par une autre personne.
Ils permettent principalement de se forcer à réfléchir à tous les cas possibles.

Mode de développement

Un audit permanent est toujours indispensable.

Pendant le développement, l'intégration de modifications devrait être relues par un nombre minimum de personnes.

Cela assure :

- La documentation des modifications (commentaire, messages de logs) ;
- Une certaine cohérence dans le code ;
- Une relecture attentive.

Repose trop sur l'humain

Mode de développement

Un audit permanent est toujours indispensable.

Pendant le développement, l'intégration de modifications devrait être relues par un nombre minimum de personnes.

Cela assure :

- La documentation des modifications (commentaire, messages de logs) ;
- Une certaine cohérence dans le code ;
- Une relecture attentive.

Repose trop sur l'humain

vsftpd est un serveur FTP considéré comme l'un des plus sécurisé.

Audit rigoureux

- Chaque fichier a une note de *confiance*;
- À chaque audit du fichier, sa note est augmentée ;
- À chaque modifiction du fichier, sa note revient à 0.

L'un programme, l'autre supervise.

- L'observateur
 - A une vision globale du projet ;
 - Joue le rôle de Bernard Pivot ;
 - Lit la documentation pour vérifier les effets de bords ;
- Le programmeur
 - Code.

La confrontation des deux points de vue est intéressante.

Plan de la section

7 Introduction

8 Attaque

9 OS Hardening

- La part de l'administrateur
- Protections

7 Introduction

- Règles du jeux
- Vulnérabilités cryptographiques
- Vulnérabilités logiques
- Manipulation des chaînes de caractères

8 Attaque

- Injections
- Corruption mémoire
- Programmation sécurisée

9 OS Hardening

- **La part de l'administrateur**
- Protections

Définition

Un système d'exploitation a la charge d'abstraire le matériel aux applications en se basant sur des niveaux de privilège.

Il permet de faire tourner de multiples applications en même temps, de permettre son usage par plusieurs utilisateurs, etc.

La grande révolution des années 90

L'après MS-DOS a été magique : chaque processus tourne dans un espace distinct. Un processus qui plante ne peut plus impacter le reste du système.

Chaque processus tourne dans un espace mémoire complètement distinct et les seuls moyens de communiquer entre processus sont les IPC.

Inter-Process Communication

- Signaux ;
- Valeur de retour ;
- Socket UNIX ou réseau ;
- Mémoire partagée ;
- Sémaphore.

Les processus sont libres d'accéder n'importe où sur le système de fichier tant qu'ils ont les droits nécessaires.

Cela signifie qu'un processus `root` compromis peut récupérer et modifier tous les fichiers du système.

chroot()

Le *chrootage* permet de modifier la racine (/) d'un processus donné. Ainsi, il lui est impossible d'accéder en dehors de cet espace.

Ce n'est pas un outil de sécurité !

Sauf si vous utilisez un patch de sécurité comme *grsecurity*

chroot()

Le *chrootage* permet de modifier la racine (/) d'un processus donné. Ainsi, il lui est impossible d'accéder en dehors de cet espace.

Ce n'est pas un outil de sécurité !

Sauf si vous utilisez un patch de sécurité comme *grsecurity*

chroot()

Le *chrootage* permet de modifier la racine (/) d'un processus donné. Ainsi, il lui est impossible d'accéder en dehors de cet espace.

Ce n'est pas un outil de sécurité !

Sauf si vous utilisez un patch de sécurité comme *grsecurity*

Pour envoyer un signal, il faut soit :

- Être root ;
- Avoir le même ID (*effective* ou *real* ID) ;
- Être dans la même session.

Cas particulier : le *tracage* utilise la même règle.

⇒ Cloisonnement assez faible

Pour envoyer un signal, il faut soit :

- Être root ;
- Avoir le même ID (*effective* ou *real* ID) ;
- Être dans la même session.

Cas particulier : le *tracage* utilise la même règle.

⇒ Cloisonnement assez faible

Pour se protéger :

- Changer d'UID ;
- Se tracer soit-même.

Des patches noyau (*grsecurity*) permettent d'empêcher les processus à l'intérieur de chroot d'envoyer des signaux en dehors.

7 Introduction

- Règles du jeux
- Vulnérabilités cryptographiques
- Vulnérabilités logiques
- Manipulation des chaînes de caractères

8 Attaque

- Injections
- Corruption mémoire
- Programmation sécurisée

9 OS Hardening

- La part de l'administrateur
- Protections

Un processus est découpé en sections, chacune est définie par :

- Une adresse de début et de fin ;
- **Les permissions** (*read, write, execute*) ;
- Etc.

Pourquoi est-ce que les données devraient être exécutables ?

⇒ Malheureusement, ces permissions étaient virtuelles !

La granularité des permissions n'étaient pas suffisantes.

Un processus est découpé en sections, chacune est définie par :

- Une adresse de début et de fin ;
- **Les permissions** (*read, write, execute*) ;
- Etc.

Pourquoi est-ce que les données devraient être exécutables ?

⇒ Malheureusement, ces permissions étaient virtuelles !

La granularité des permissions n'étaient pas suffisantes.

Fonctionnalité du processeur

L'arrivée de l'architecture 64 bits par AMD a vu l'introduction d'un nouveau bit, le bit « *no execute* » qui marque chaque page.

Le jeu des attaquants s'est alors modifié, le but est maintenant de modifier des pages mémoires avec le bit d'exécution activé.

OpenBSD : Writing xor execute

Le noyau s'assure qu'une page ne peut pas avoir à la fois le bit d'écriture et le bit d'exécution.

Chat et la souris

À cause des protections précédentes, il n'est plus possible d'exécuter aussi facilement du code depuis le buffer en train d'être exploiter.
⇒ Car les pages ne sont pas exécutables.

Le jeu est maintenant d'exécuter des instructions « légitimes ». Réussir à sauter dans les fonctions de la libc par exemple.

Randomization des adresses

L'ASLR consiste à rendre aléatoire les adresses de section afin d'empêcher un exploit de sauter dans du code existant.

- La randomization est renouvelée à chaque exécution (ou pas) ;
- La quantité d'entropie n'est pas toujours parfaite ;
- Avoir un *information leak* devient un atout ;
- Un off-by-one peut devenir la méthode facile.

Stack overflow

Les variables sont réordonnancées pour éviter qu'un overflow puisse déborder dans des pointeurs de données utilisés plus tard.

Off-by-one

Pour des raisons d'optimisation des accès mémoire, les variables sont alignées en utilisant du *padding*.

Il y a désormais un peu de place entre deux variables si nécessaire.

Une valeur aléatoire est placée entre la première variable locale et le pointeur de pile sauvegardé.

Lorsqu'on sort d'une fonction, le canari est vérifié, s'il n'a pas été écrasé, on déroule l'épilogue.

GNU libc

Des vérifications supplémentaires sont désormais effectuées avant de supprimer un *chunk*.

Toutes les méthodes classiques d'exploitations sont caduques.

Dans Microsoft Vista, toutes les adresses sont **XORées** avec un nombre aléatoire.

Disclaimer

Chaque technique prise indépendamment est contournable, mais l'utilisation de toutes ces techniques permet d'assurer un niveau de sécurité suffisant.

⇒ Défense en profondeur

RBAC

Lors d'un appel système, le noyau vérifie que le processus en cours (qui tourne sous un rôle donné) a l'autorisation d'effectuer l'opération.

- Exemple : SELinux, AppArmor, GrSecurity, RSBAC.
- Complexité de configuration.

Quel est l'impact d'un bug dans le noyau ?

Réveillez votre voisin

C'est fini.