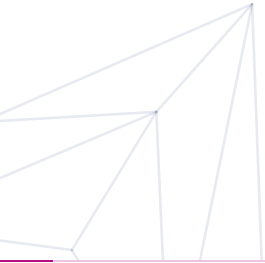
SEDE

D'après le cours de Marc Espie

Loïc Rouquette
EPITA - Laboratoire de Recherche de l'EPITA (LRE)
26 mars 2026

Table des matières



01

Introduction

« Il y a plus de conférences pour les attaquants que de conférences pour la sécurité. C'est ça le problème. »

– Theo de Raadt

Objectifs du cours

Objectifs de Base

- Ré-expliquer l'écosystème sous l'angle de la sécurité ;
- Vous donner le vocabulaire nécessaire pour réussir vos entretiens ;
- Dissiper les fauses idées sur le développement sécurisé.

Prérequis & Limites

- Vous avez principalement fait du `C` et de l'`Unix` jusqu'à présent.
- Cela a toujours de l'importance.
- Il y a aussi des problèmes qui ne sont pas spécifiques au C.

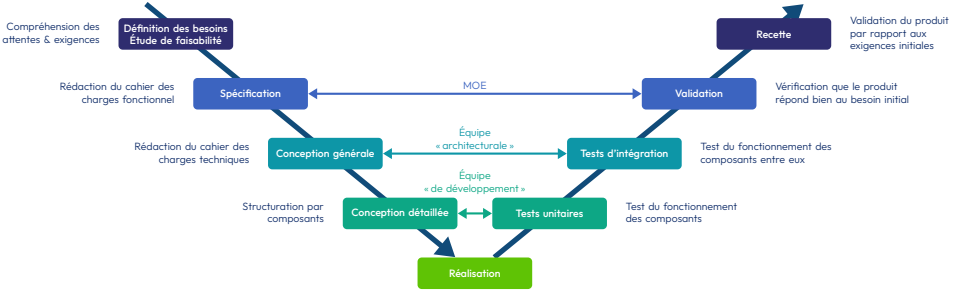
Prérequis & Limites

- Vous avez principalement fait du `C` et de l'`Unix` jusqu'à présent.
- Cela a toujours de l'importance.
- Il y a aussi des problèmes qui ne sont pas spécifiques au C.

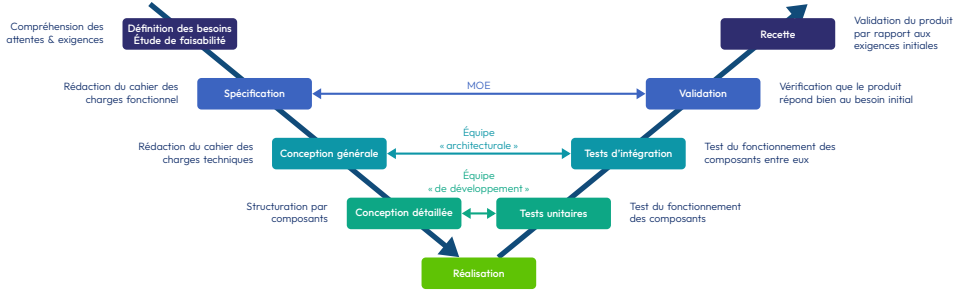
Les petites lignes

- La majorités des langages modernes sophistiqués possèdes des runtimes en C/C++.
- Unix possède un excellent modèle de sécurité.

Le Cycle de développement



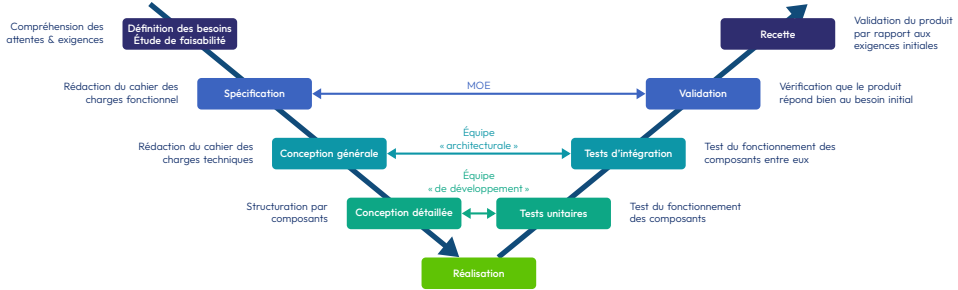
Le Cycle de développement



La cycle de dévoppement Les entreprises classiques écrivent les spécifications...

- Les développeurs les implémentent ;

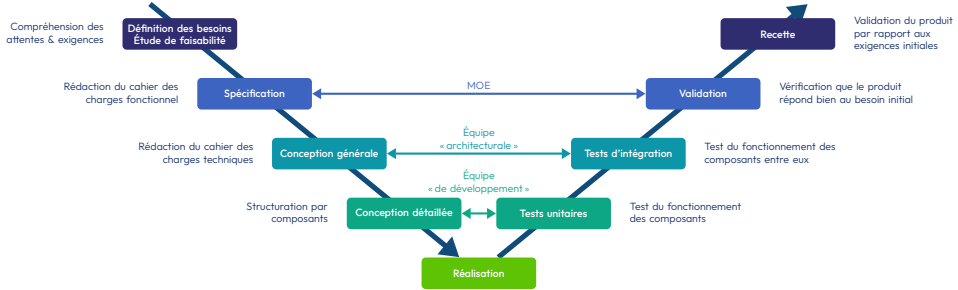
Le Cycle de développement



La cycle de dévoppement Les entreprises classiques écrivent les spécifications...

- Les développeurs les implémentent ;
- Les experts en base de donnée font les BDD ;

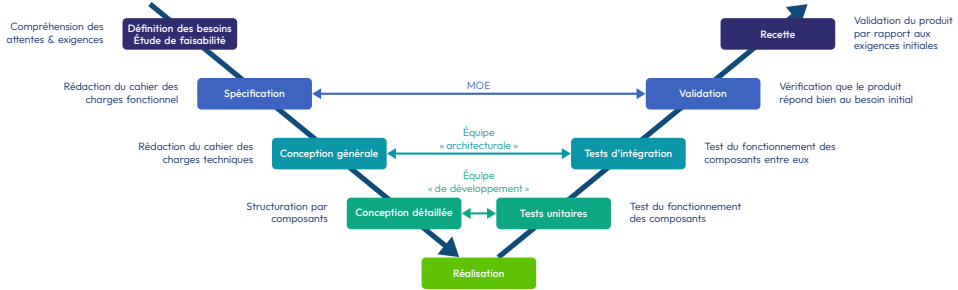
Le Cycle de développement



La cycle de dévoppement Les entreprises classiques écrivent les spécifications...

- Les développeurs les implémentent ;
- Les experts en base de donnée font les BDD ;
- Les ingénieurs systèmes font le déploiement ;

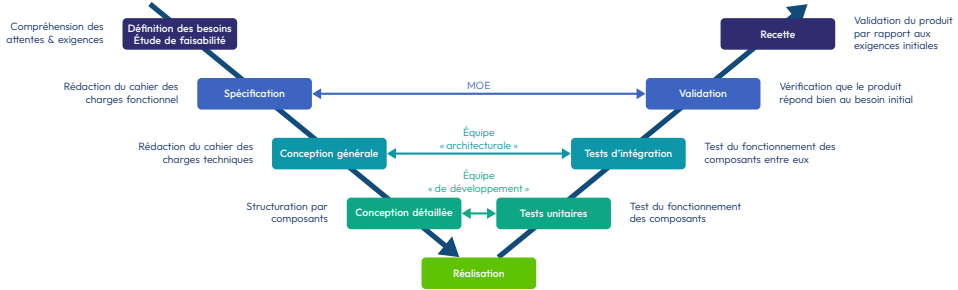
Le Cycle de développement



La cycle de dévoppement Les entreprises classiques écrivent les spécifications...

- Les développeurs les implémentent ;
- Les experts en base de donnée font les BDD ;
- Les ingénieurs systèmes font le déploiement ;
- Les testeurs font les recettes ;

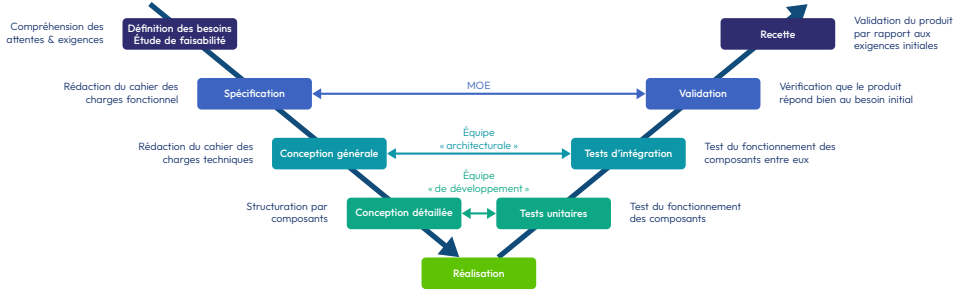
Le Cycle de développement



La cycle de dévoppement Les entreprises classiques écrivent les spécifications...

- Les développeurs les implémentent ;
- Les experts en base de donnée font les BDD ;
- Les ingénieurs systèmes font le déploiement ;
- Les testeurs font les recettes ;
- Les auditeurs font la sécurité.

Le Cycle de développement



La cycle de dévoppement Les entreprises classiques écrivent les spécifications...

- Les développeurs les implémentent ;
- Les experts en base de donnée font les BDD ;
- Les ingénieurs systèmes font le déploiement ;
- Les testeurs font les recettes ;
- Les auditeurs font la sécurité.

Cela ne fonctionne pas !

Pourquoi cela ne marche pas ?

« La spécialisation, c'est pour les insectes. »

– Robert A. Heinlein

Pourquoi cela ne marche pas ?

« La spécialisation, c'est pour les insectes. »

– Robert A. Heinlein

- Si les auditeurs trouvent un bug, c'est parfois parce que la conception même est mauvaise.
- Les auditeurs ne peuvent pas tout attraper.
- Les développeurs **doivent** donc connaître les bonnes pratiques.

Testeurs & Experts BDD

Testeurs vs Devs

- Il ne faut pas dresser les testeurs contre les développeurs ;
- Un bon testeur est inestimable ;
- Le but est de documenter et de corriger les bugs ensemble.

Testeurs & Experts BDD

Testeurs vs Devs

- Il ne faut pas dresser les testeurs contre les développeurs ;
- Un bon testeur est inestimable ;
- Le but est de documenter et de corriger les bugs ensemble.

Experts en base de données

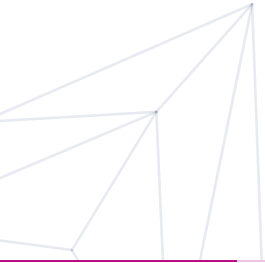
- Beaucoup d'« experts » en base de données ne connaissent même pas les injections SQL.

La réalité des Versions

- Vous finissez par ”sortir une version” (ex : V5.0).

La réalité des Versions

- Vous finissez par ”sortir une version” (ex : V5.0).
- Ce n’est pas toujours la fin. Êtes-vous le fournisseur ? Ce n’est pas le cas pour les distributions Unix.



La réalité des Versions

- Vous finissez par ”sortir une version” (ex : V5.0).
- Ce n’est pas toujours la fin. Êtes-vous le fournisseur ? Ce n’est pas le cas pour les distributions Unix.
- Branches et Support : Avant la sortie, vous créez une branche (5.0 beta). Après la sortie, vous continuez le développement, mais des ressources doivent rester sur la 5.0.

La réalité des Versions

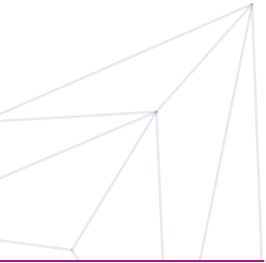
- Vous finissez par ”sortir une version” (ex : V5.0).
- Ce n’est pas toujours la fin. Êtes-vous le fournisseur ? Ce n’est pas le cas pour les distributions Unix.
- Branches et Support : Avant la sortie, vous créez une branche (5.0 beta). Après la sortie, vous continuez le développement, mais des ressources doivent rester sur la 5.0.
- Un bug survient ? Vous devez le corriger... publier la 5.1... le tester... et qu’en est-il de la branche 4 ? et la 3 ?

La réalité des Versions

- Vous finissez par ”sortir une version” (ex : V5.0).
- Ce n’est pas toujours la fin. Êtes-vous le fournisseur ? Ce n’est pas le cas pour les distributions Unix.
- Branches et Support : Avant la sortie, vous créez une branche (5.0 beta). Après la sortie, vous continuez le développement, mais des ressources doivent rester sur la 5.0.
- Un bug survient ? Vous devez le corriger... publier la 5.1... le tester... et qu’en est-il de la branche 4 ? et la 3 ?
- Notions clés : EOL (End of Life) et ESR (Extended Support Release).

Bugs de sécurité & Vulnérabilités

- Un simple bug n'est pas automatiquement une faille de sécurité.

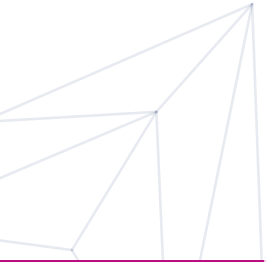


Bugs de sécurité & Vulnérabilités

- Un simple bug n'est pas automatiquement une faille de sécurité.
- La plupart des attaques sont basées sur une série de bugs.

Bugs de sécurité & Vulnérabilités

- Un simple bug n'est pas automatiquement une faille de sécurité.
- La plupart des attaques sont basées sur une série de bugs.
- Nous voulons une Défense en profondeur (Defense in depth).



Bugs de sécurité & Vulnérabilités

- Un simple bug n'est pas automatiquement une faille de sécurité.
- La plupart des attaques sont basées sur une série de bugs.
- Nous voulons une Défense en profondeur (Defense in depth).
- Corriger un seul de ces bugs peut stopper l'attaque !

Bugs de sécurité & Vulnérabilités

- Un simple bug n'est pas automatiquement une faille de sécurité.
- La plupart des attaques sont basées sur une série de bugs.
- Nous voulons une Défense en profondeur (Defense in depth).
- Corriger un seul de ces bugs peut stopper l'attaque !
- Une attaque est aussi appelée un exploit.

Bugs de sécurité & Vulnérabilités

- Un simple bug n'est pas automatiquement une faille de sécurité.
- La plupart des attaques sont basées sur une série de bugs.
- Nous voulons une Défense en profondeur (Defense in depth).
- Corriger un seul de ces bugs peut stopper l'attaque !
- Une attaque est aussi appelée un exploit.
- Les logiciels ont des vulnérabilités.

Qui l'a trouvé et quoi faire ?

Qui l'a trouvé ?

- Le développeur a trouvé le bug.
- Un utilisateur externe a trouvé le bug.
- Est-ce reconnu comme un problème de sécurité ?
- L'utilisateur externe est-il bienveillant ou non ?

Quoi faire ?

- Prouver que c'est un problème de sécurité.
- Être proactive à ce sujet.
- Le corriger **sans que les méchants ne le sachent**.

Exemples de CVE (Common Vulnerabilities and Exposures)

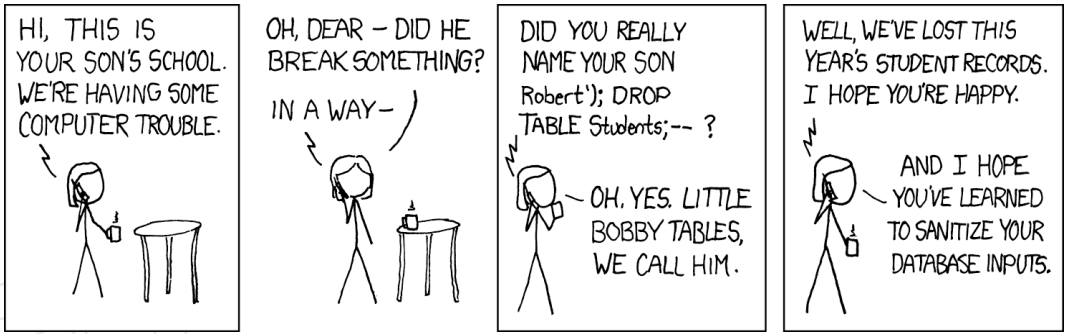
CVE-2019-19521 (OpenBSD)

Vulnérabilité de contournement d'authentification. Exploitable à distance dans smtpd, ldapd, etc.
Cependant, sshd n'est pas exploitable grâce à ses mécanismes de défense en profondeur.

CVE-2016-9843 (zlib)

La fonction crc32_big dans zlib permettait à des attaquants d'avoir un impact *non spécifié* via des calculs CRC en big-endian.

Injections SQL



Injections SQL

```php
<?php
// La mauvaise méthode en PHP
function ask_user_info($user) {
    $db->do("select * from users where user='" + $user + "'");
}
```

Injections SQL

```php
<?php
// La mauvaise méthode en PHP
function ask_user_info($user) {
    $db->do("select * from users where user='" + $user + "'");
}
```

Ce que l'attaquant fait : `user=robin'; drop all tables;--`

Injections SQL

```php
<?php
// La mauvaise méthode en PHP
function ask_user_info($user) {
    $db->do("select * from users where user='" + $user + "'");
}
```

Ce que l'attaquant fait : `user=robin'; drop all tables;--`
Résultat : `select * from users where user='robin'; drop all tables; --`

Injections SQL : Encore plus subtil

Ce que l'attaquant entre : `user=robin' or 1==1;--`

Injections SQL : Encore plus subtil

Ce que l'attaquant entre : `user=robin' or 1==1;--`

```
select * from users where user='robin' or 1==1;--'
```

Injections SQL : Encore plus subtil

Ce que l'attaquant entre : `user=robin' or 1==1;--`

```
select * from users where user='robin' or 1==1;--'
```

Pourquoi cela arrtive-t-il ?

- Parce que ce n'est pas enseigné dans tous les cours de bases de données.
- Parce que PHP (historiquement) ne supportait que la fonction **`do()`** (jusqu'à ce qu'ils aient un modèle objet).

La mauvaise Solution vs La bonne solution

La mauvaise solution

Sanitiser (mettre des guillemets) l'argument. Se dire que l'utilisateur est « safe » s'il n'y a pas de `'`.
Mais que fait-on de M. O'Brien ? Vous devez tout échapper. C'est une méthode vouée à l'échec.

La mauvaise Solution vs La bonne solution

La mauvaise solution

Sanitiser (mettre des guillemets) l'argument. Se dire que l'utilisateur est « safe » s'il n'y a pas de '.
Mais que fait-on de M. O'Brien ? Vous devez tout échapper. C'est une méthode vouée à l'échec.

Quoi faire ?

Utiliser des requêtes préparées (Prepared Statements) ou un véritable ORM (comme Symfony en PHP).

```php
<?php
// The better way
function ask_user_info($user) {
    $stmt = $db->prepare("select * from users where user=?");
    $stmt->bind_param("s", $user);
    $stmt->execute();
}
```

À propos de la validation (Quoting)

- Ce que vous ignorez vous TUERA !
- Ne faites **jamais** de vérification basée sur des modèles **négatifs** (Negative patterns).
  - Exemple : Une adresse email **N'EST PAS** quelque chose qui ne contient pas certains caractères...
- Elle **EST** quelque chose qui correspond uniquement à un modèle précis (Regex stricte).

Les vulnérabilites de Format String (`printf`)

```c
void print_msg(const char *msg)
{
    printf("There is a problem here;\n");
    printf(msg); // DANGER ICI
}
```

Cela pourrait lire des éléments sur la pile et vous montrer des choses que vous ne devriez pas voir (via `%x` ou `%s`). Mais c'est en fait **BIEN PIRE**.

Le danger mortel : (%n)

Extrait du manuel PRINTF(3)

```
n   The number of characters written so far is stored into the
    integer indicated by the int * (or variant) pointer argument.
    No argument is converted.
```

Le danger mortel : (%n)

Extrait du manuel PRINTF(3)

```
n   The number of characters written so far is stored into the
    integer indicated by the int * (or variant) pointer argument.
    No argument is converted.
```

Un attaquant peut utiliser %n dans la variable msg pour écrire dans la mémoire du programme !

Le danger mortel : (%n)

Extrait du manuel PRINTF(3)

```
n   The number of characters written so far is stored into the
    integer indicated by the int * (or variant) pointer argument.
    No argument is converted.
```

Un attaquant peut utiliser %n dans la variable msg pour écrire dans la mémoire du programme !

La correction évidente

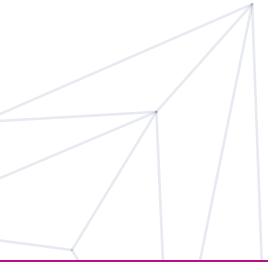
```
printf("%s", msg);
```

Le Syndrome LISP & Les Abstractions

- Chaque ”appel” a tendance à évoluer pour en faire plus que ce qui est bon pour lui.
- Méfiez-vous des violations des couches d’abstraction.

Les Shells Cachés

- Par exemple `system(3)` et `popen(3)`.
- Les deux exécutent basiquement `sh -c "string"`.
- Pour les utiliser correctement, vous devez échapper **TOUT**.



Les Shells Cachés

- Par exemple `system(3)` et `popen(3)`.
- Les deux exécutent basiquement `sh -c "string"`.
- Pour les utiliser correctement, vous devez échapper **TOUT**.
- Qu'est-ce que « tout » `? # {} ( ) " ' ` $` etc. C'est presque impossible à sécuriser parfaitement.

La bonne méthode : apprenez Unix

Au lieu de `system()`, utilisez `fork` et `exec` :

```
int r = fork();
if (r == -1) err(1, "fork");
if (r == 0) {
    execlp("mycmd", "cmd", "param", NULL);
    err(1, "exec");
}
int status;
r = wait(&status);
// check status
```

Les paramètres sont passés séparément, le shell n'interprète plus la chaîne ! (Ou utilisez `posix_spawn(3)`).

Scripts et variables mal protégées

Script 1

```
#!/bin/sh
file=$1
rm $file
```

L'appel `./s "my file"` (Échoue ou supprime `'my'` et `'file'`).

Script 2

```
#!/bin/sh
rm "$@"
```

L'appel `./s -rf /` (Catastrophe ! Les guillemets ne protègent pas des options.)

Scripts et variables mal protégées

Script 1

```
#!/bin/sh
file=$1
rm $file
```

L'appel `./s "my file"` (Échoue ou supprime `'my'` et `'file'`).

Script 2

```
#!/bin/sh
rm "$@"
```

L'appel `./s -rf /` (Catastrophe ! Les guillemets ne protègent pas des options.)

Solution : Utilisez `--` pour stopper l'analyse des options (ex : `rm -- "$@"`).

Fail closed (Échouer de manière sécurisée)

Utilisez toujours `set -e` dans vos scripts shell pour qu'ils s'arrêtent à la moindre erreur !

Fail closed (Échouer de manière sécurisée)

Utilisez toujours `set -e` dans vos scripts shell pour qu'ils s'arrêtent à la moindre erreur !

```
#!/bin/sh
set -e
error=false
if ! test -f Makefile ; then
    echo "No ports files ?"
    error=true
fi
...
$error && exit 1
# Suite du script sécurisée ...
```

Fail closed (Échouer de manière sécurisée)

Utilisez toujours `set -e` dans vos scripts shell pour qu'ils s'arrêtent à la moindre erreur !

```
#!/bin/sh
set -e
error=false
if ! test -f Makefile ; then
    echo "No ports files ?"
    error=true
fi
...
$error && exit 1
# Suite du script sécurisée ...
```

Pourquoi ?

Fail closed (Échouer de manière sécurisée)

```
#!/bin/bash
TARGET_DIR="/tmp/mon_projet_inexistant"
cd $TARGET_DIR
rm -rf * .
```

Fail closed (Échouer de manière sécurisée)

```
#!/bin/bash
TARGET_DIR="/tmp/mon_projet_inexistant"
cd $TARGET_DIR
rm -rf * .
```

Que fait l'exécution suivante ?

```
~ bash mon_script.sh
```

Fail closed (Échouer de manière sécurisée)

```
#!/bin/bash
TARGET_DIR="/tmp/mon_projet_inexistant"
cd $TARGET_DIR
rm -rf * .
```

Que fait l'exécution suivante ?

```
~ bash mon_script.sh
```

Que fait l'exécution suivante ?

```
~ bash mon_script.sh
```

Idées reçues

- « C'est trop compliqué, ça ne sera pas exploitable. »

Idées reçues

- **« C'est trop compliqué, ça ne sera pas exploitable. »**
  Faux ! L'URL overflow sur IIS (Internet Information Services) a prouvé le contraire en créant du shellcode dans des chaînes étendues Unicode (technique du « venetian blind »).

Idées reçues

- **« C'est trop compliqué, ça ne sera pas exploitable. »**
  Faux ! L'URL overflow sur IIS (Internet Information Services) a prouvé le contraire en créant du shellcode dans des chaînes étendues Unicode (technique du « venetian blind »).
- **« L'Open Source est plus/moins sécurisé que le code fermé. »**

Idées reçues

- **« C'est trop compliqué, ça ne sera pas exploitable. »**
  Faux ! L'URL overflow sur IIS (Internet Information Services) a prouvé le contraire en créant du shellcode dans des chaînes étendues Unicode (technique du « venetian blind »).
- **« L'Open Source est plus/moins sécurisé que le code fermé. »**
  Faux ! Beaucoup de gens savent faire de l'ingénierie inverse.

Idées reçues

- **« C'est trop compliqué, ça ne sera pas exploitable. »**
  Faux ! L'URL overflow sur IIS (Internet Information Services) a prouvé le contraire en créant du shellcode dans des chaînes étendues Unicode (technique du « venetian blind »).
- **« L'Open Source est plus/moins sécurisé que le code fermé. »**
  Faux ! Beaucoup de gens savent faire de l'ingénierie inverse.
- **« Je ne fais pas de bugs. »**

Idées reçues

- **« C'est trop compliqué, ça ne sera pas exploitable. »**
  Faux ! L'URL overflow sur IIS (Internet Information Services) a prouvé le contraire en créant du shellcode dans des chaînes étendues Unicode (technique du « venetian blind »).
- **« L'Open Source est plus/moins sécurisé que le code fermé. »**
  Faux ! Beaucoup de gens savent faire de l'ingénierie inverse.
- **« Je ne fais pas de bugs. »**
  Il suffit d'un seul bug. Tout est exploitable à terme.

Exemple : CVE-2017-7269 L'Illusion de l'Inexploitable

Conversion Unicode

Le serveur IIS transforme systématiquement l'URL entrante en y insérant un espace vide ou "zéro" (`\x00`) après chaque caractère de la requête.

Fragmentation

Les instructions malveillantes injectées en mémoire sont violemment scindées par ces zéros imposés, les rendant totalement illisibles pour le processeur.

Adressage Restreint

À cause de ces zéros intercalés, les adresses mémoire atteignables pour exécuter l'attaque sont extrêmement limitées, bloquant la redirection du système.

Le Venetian Blind

- **L'Analogie du Store** : Les données injectées par l'attaquant représentent les lattes pleines du store, tandis que les zéros (`\x00`) imposés par IIS représentent les espaces vides entre elles.
- **Création de Points** : L'attaquant utilise des micro-instructions inoffensives capables d'enjamber les espaces vides sans faire planter ou alerter le système en cours d'exécution.
- **Phase d'Assemblage :** Ce parcours d'obstacles minutieux mène la machine vers un petit outil de décodage caché, qui nettoie les zéros et reconstitue le code d'attaque final.

Buffer Overflow : L'exemple classique

```c
#include <stdio.h>

void f() {
    char s[70];
    gets(s);
}

int main() {
    f();
}
```

Buffer Overflow : L'exemple classique

```c
#include <stdio.h>

void f() {
    char s[70];
    gets(s);  // La fonction fatale
}

int main() {
    f();
}
```

Buffer Overflow : L'exemple classique

```c
#include <stdio.h>

void f() {
    char s[70];
    gets(s);  // La fonction fatale
}

int main() {
    f();
}
```

`gets()` ne vérifie pas la taille de l'entrée. Si l'utilisateur entre plus de 70 caractères, il écrase la mémoire adjacente.

Derrière le code : La Pile

```
f:
    pushq %rbp
    movq  %rsp, %rbp
    subq  $80, %rsp
    leaq  -80(%rbp), %rdi
    callq gets
    leave
    retq
```

- 80 octets sont alloués sur la pile (70 arrondis au multiple supérieur 16x5).
- L'attaquant remplit le buffet `s`.
- Il écrase l'ancien registre de base

Mitigrations contre les Buffer Overflows

- **Pile non exécutable (NX) :** Dépend de l'OS. Empêche le processeur d'exécuter du code situé dans la pile.
- **Randomisation (ASLR) :** Rend aléatoire l'espace d'adressage du programme, de la pile et du tas (heap) à chaque exécution.
- **Canaris (Canaries) :** Dépend du compilateur. Le prologue de la fonction insère une donnée aléatoire sur la pile. L'épilogue vérifie que cette donnée n'a pas changé avant d'exécuter ret.

Exemple de Canari en Assembleur

```
f:
 ...
# Prologue: Placement du canari
movq   __guard_local(%rip), %rax
movq   %rax, -8(%rbp)
 ...
callq gets@PLT
 ...
# Epilogue: Vérification du canari
movq   __guard_local(%rip), %rax
cmpq   -8(%rbp), %rax
jne    .LBB0_2  # Saute vers la gestion d'erreur (smash handler)
                # si corrompu
retq
```

Heap Overflow (Dépassement sur le tas)

Les Overflows ne sont pas toujours sur la pile :

```
char *make_filename(const char *dir, const char *file) {
    char *r = emalloc(strlen(dir) + strlen(file) + 1);
    strcpy(r, dir);
    strcat(r, "/");
    strcat(r, file);
    return r;
}
```

Heap Overflow (Dépassement sur le tas)

Les Overflows ne sont pas toujours sur la pile :

```
char *make_filename(const char *dir, const char *file) {
    char *r = emalloc(strlen(dir) + strlen(file) + 1);
    strcpy(r, dir);
    strcat(r, "/"); // Oublie pour le '/' dans le emalloc
    strcat(r, file);
    return r;
}
```

Heap Overflow (Dépassement sur le tas)

Les Overflows ne sont pas toujours sur la pile :

```
char *make_filename(const char *dir, const char *file) {
    char *r = emalloc(strlen(dir) + strlen(file) + 1);
    strcpy(r, dir);
    strcat(r, "/"); // Oublie pour le '/' dans le emalloc
    strcat(r, file);
    return r;
}
```

Conséquence : Vous écrasez le bloc mémoire suivant. Dans une liste chaînée, vous tuez le pointeur next. Dans un allocateur en puissance de deux, vous corrompez les métadonnées de l'allocateur.

Comment éviter cela ?

- Ne faites pas de bugs (facile à dire).
- Connaissez vos APIs.
- Préférez les idiomes sécurisés.
- Rendez le code simple. Un code simple doit avoir l'air simple. Rendez les choses explicites.
- Les APIs doivent calculer les tailles pour vous, retourner des erreurs propres et échouer de manière sécurisée (fail closed).

Un code simple et explicite

```c
char *make_filename(const char *dir, const char *file) {
    const char *slash = "/";
    // La taille est explicite, l'ordre correspond.
    char *r = emalloc(strlen(dir) + strlen(slash) + strlen(file) + 1);
    strcpy(r, dir);
    strcat(r, slash);
    strcat(r, file);
    return r;
}
```

Un code simple et explicite

```c
char *make_filename(const char *dir, const char *file) {
    const char *slash = "/";
    // La taille est explicite, l'ordre correspond.
    char *r = emalloc(strlen(dir) + strlen(slash) + strlen(file) + 1);
    strcpy(r, dir);
    strcat(r, slash);
    strcat(r, file);
    return r;
}
```

Le compilateur optimise très bien ce code, allant jusqu'à utiliser un `Tailcall` (saut direct) pour le dernier `strcat`, prouvant qu'un code clair n'est pas moins performant.

Les Pages de Guarde

Une autre technique de mitigation pour la mémoire (Tas) :

- Allouer les données de préférence à la fin d'une page mémoire.
- Laisser la page suivante vide (non mappée).
- Si un overflow se produit, il tentera d'écrire dans la page non mappée, provoquant un plantage immédiat (Segfault) au lieu d'une corruption silencieuse.

De meilleurs APIs

- N'utilisez PAS `strcpy`, `strcat`.

De meilleurs APIs

- **N'utilisez PAS** `strcpy`, `strcat`.
- **N'utilisez PAS** non plus `strncpy`, `strncat` (ils ne garantissent pas le caractère nul de fin).
- **Préférez** `strlcpy`, `strlcat`.

```
char pname[PATH_MAX];
if (strlcpy(pname, dir, sizeof(pname)) ≥ sizeof(pname))
    goto toolong;
if (strlcat(pname, file, sizeof(pname)) ≥ sizeof(pname))
    goto toolong;
```

La faille de Drepper

« Mais moi, je n'écris pas de code bogué. »

La faille de Drepper

« Mais moi, je n'écris pas de code bogué. »

C'est l'argument arrogant utilisé pour justifier la lente adoption de fonctions sécurisées comme **`strlcpy`** dans certains environnements (notamment par Ulrich Drepper pour la **`glibc`**).

La loi de Sturgeon

90% de tous les logiciels sont :

La loi de Sturgeon

90% de tous les logiciels sont :

- De la camelote (crap) ;

La loi de Sturgeon

90% de tous les logiciels sont :

- De la camelote (crap) ;
- Inutiles à optimiser ;

La loi de Sturgeon

90% de tous les logiciels sont :

- De la camelote (crap) ;
- Inutiles à optimiser ;
- Bogués ;

La loi de Sturgeon

90% de tous les logiciels sont :

- De la camelote (crap) ;
- Inutiles à optimiser ;
- Bogués ;
- Copiés-collés ;

La loi de Sturgeon

90% de tous les logiciels sont :

- De la camelote (crap) ;
- Inutiles à optimiser ;
- Bogués ;
- Copiés-collés ;
- Imparfaits.

La loi de Sturgeon

90% de tous les logiciels sont :

- De la camelote (crap) ;
- Inutiles à optimiser ;
- Bogués ;
- Copiés-collés ;
- Imparfaits.

Nous ne pouvons pas tout réparer.

La loi de Sturgeon

90% de tous les logiciels sont :

- De la camelote (crap) ;
- Inutiles à optimiser ;
- Bogués ;
- Copiés-collés ;
- Imparfaits.

Nous ne pouvons pas tout réparer.

Il ne faut pas **rien** réparer, mais réparer les corrections **faciles** à fort **impact**.

Encore des APIs

- **Préférez** `snprintf` à `sprintf`.
- **Utilisez** `asprintf` si vous le devez absolument (il alloue la mémoire dynamiquement pour vous).
- **Mettez des tailles partout :** vous voulez aider les auditeurs. Si la taille d'un tampon n'est pas évidente, faites-en un argument de votre API.

Les compilateurs vous aident

```c
char *make_filename(const char *file, const char *dir) {
    char buffer[MAXBUF];
    snprintf(buffer, sizeof buffer, "%s/%s", file, dir);
    return buffer;
}
```

Les compilateurs vous aident

```c
char *make_filename(const char *file, const char *dir) {
    char buffer[MAXBUF];
    snprintf(buffer, sizeof buffer, "%s/%s", file, dir);
    return buffer; // ERREUR CLASSIQUE
}
```

Les compilateurs vous aident

```
char *make_filename(const char *file, const char *dir) {
    char buffer[MAXBUF];
    snprintf(buffer, sizeof buffer, "%s/%s", file, dir);
    return buffer; // ERREUR CLASSIQUE
}
```

Le compilateur (si vous mettez `-Wall`) vous dira :

```
warning: address of stack memory associated with local variable 'buf-
fer' returned [-Wreturn-stack-address]
```

Le modèle de Sécurtié Unix

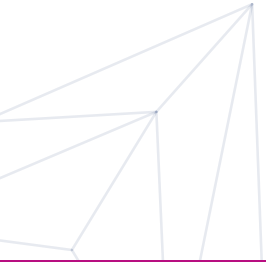
Quand vérifiez-vous que vous pouvez accéder à un fichier ?

- Au **open** ?

Le modèle de Sécurtié Unix

Quand vérifiez-vous que vous pouvez accéder à un fichier ?

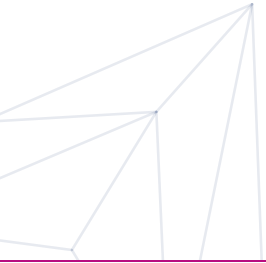
- Au **open** ?
  Les droits sont **vérifiés au moment** de l'appel à `open()`, en fonction des flags demandés (`O_RDONLY`, `O_WRONLY`, `O_RDWR`...).



Le modèle de Sécurtié Unix

Quand vérifiez-vous que vous pouvez accéder à un fichier ?

- Au `open` ?
  Les droits sont **vérifiés au moment** de l'appel à `open()`, en fonction des flags demandés (`O_RDONLY`, `O_WRONLY`, `O_RDWR`...).
- Au `read/write` ?



Le modèle de Sécurtié Unix

Quand vérifiez-vous que vous pouvez accéder à un fichier ?

- Au `open` ?
  Les droits sont **vérifiés au moment** de l'appel à `open()`, en fonction des flags demandés (`O_RDONLY`, `O_WRONLY`, `O_RDWR`...).
- Au `read/write` ?
  Les droits **ne sont plus vérifiés** à chaque appel. La vérification a déjà eu lieu lors du `open()`.
  Le noyau vérifie seulement que l'opération est **cohérente avec le mode d'ouverture** du `fd`.

Le modèle de Sécurtié Unix

Quand vérifiez-vous que vous pouvez accéder à un fichier ?

- Au `open` ?
  Les droits sont **vérifiés au moment** de l'appel à `open()`, en fonction des flags demandés (`O_RDONLY`, `O_WRONLY`, `O_RDWR`...).
- Au `read/write` ?
  Les droits **ne sont plus vérifiés** à chaque appel. La vérification a déjà eu lieu lors du `open()`.
  Le noyau vérifie seulement que l'opération est **cohérente avec le mode d'ouverture** du `fd`.
- Au `exec` ?

Le modèle de Sécurtié Unix

Quand vérifiez-vous que vous pouvez accéder à un fichier ?

- Au `open` ?
  Les droits sont **vérifiés au moment** de l'appel à `open()`, en fonction des flags demandés (`O_RDONLY`, `O_WRONLY`, `O_RDWR`...).
- Au `read/write` ?
  Les droits **ne sont plus vérifiés** à chaque appel. La vérification a déjà eu lieu lors du `open()`.
  Le noyau vérifie seulement que l'opération est **cohérente avec le mode d'ouverture** du `fd`.
- Au `exec` ?
  Les droits sont **vérifiés au moment** de l'appel à `exec()`.

Des Questions ?