[SLPS] sécurité logicielle et développement sécurisé

Développement

Julien Sterckeman

EPITA

2013

Table des matières

Filtrer toutes les entrées

Solutions

- ✓ déterminer toutes les entrées possibles
- ✓ filtrer par liste blanche (pas par liste noire) :
  - chaîne : taille et caractères autorisés
  - nombre : min et max
  - autre : expression rationnelle
- ✓ refaire du coté serveur les vérifications clientes
- ✓ défense en profondeur : filtrer soi-même et activer les protections du langage (MAGIC_QUOTES)

Filtrer toutes les entrées

Exemple en PHP :

```
function ensure_digit($var)
{
  if (preg_match('/^[0-9]+$/', $var))
    return($var);
  error_log("PHP: ensure_digit($var) failed !");
  throw new Exception("variable");
}

try
{
  $i = ensure_digit($_GET["i"]);
}
catch () { ... }
```

Vérifier les données produites

Solutions

- ✓ vérifier l'encodage et l'échappement des données produites
- ✓ utiliser du typage fort (procédures stockées, etc.)
- ✓ vérifier que les données produites soient bien conformes au format attendu (communication à un autre système), et à l'encodage

Comment faire

Solutions

- ✓ filtrer toutes les entrées et les sorties
- ✓ utiliser des requêtes SQL préparées, qui possèdent un typage fort (séparation de la structure de la requête et des données)
- ✓ utiliser des fonctions d'échappement du langage pour les chaînes (`mysql_real_escape_string` en PHP par exemple)
- ✓ utiliser des comptes SQL avec le minimum de droits (uniquement SELECT sur les tables utiles pour la partie publique d'un site Web)
- ✓ journaliser les requêtes SQL

Note : problème équivalent avec les injections LDAP.

Problèmes multi-langages - Injections SQL

PHP

Pas bien

```php
$sql = "SELECT * FROM customers WHERE name = '" .
    $_POST['name'] . "'";
$query = mysql_query($sql) or die("Error!");
```

Bien

```php
$sql = "SELECT * FROM customers WHERE name = '" .
    mysql_real_escape_string($_POST['name']) . "'";
$query = mysql_query($sql) or die("Error!");
```

Attention : pour d'autres types que les chaînes de caractères, cette fonction ne sert à rien, il faut une expression rationnelle ou une requête SQL préparée !

Problèmes multi-langages - Injections SQL

Java

Pas bien

```java
PreparedStatement pstmt = con.prepareStatement(
  "SELECT * FROM customers WHERE name = " +
  request.getParameter("name"));
ResultSet rset = pstmt.executeQuery();
```

Bien

```java
PreparedStatement pstmt = con.prepareStatement(
  "SELECT * FROM customers WHERE name = ?");
pstmt.setString(1, request.getParameter("name"));
ResultSet rset = pstmt.executeQuery();
```

Principe de l'attaque

XSS = injection de code Javascript :

1. l'attaquant trouve un XSS et la chaîne pour l'exploiter
2. l'attaquant envoie un lien (mail, twitter, etc.) à la victime
3. la victime clique sur le lien malveillant (qui exploite le XSS)
4. le code JavaScript de l'attaquant s'exécute dans le navigateur de la victime

Risques

Exécution de code HTML/Javascript arbitraire sur le client, en se faisant passer pour le serveur Web :

- vol de session :
- pseudo défiguration :
- phishing :

Problèmes multi-langages - Cross-site scripting (XSS)

Types

- réflexif/réfléchi
  - génération à la volée du code HTML (paramètre d'une URL, d'un formulaire)
- permanent/stocké
  - stockage des données malveillantes sur le site (base de données, fichier, etc.)
  - très dangereux !

Problèmes multi-langages - Cross-site scripting (XSS)

Comment faire

Solutions

- ✓ filtrer toutes les entrées
- ✓ déterminer les données réécrites au navigateur et filtrer avant de les envoyer (ce qui dépend du contexte)
- ✓ utiliser des bibliothèques spécifiques : Microsoft Anti-XSS library, Apache Wicket, `htmlentities`
- ✓ toujours spécifier l'encodage des pages web
- ✓ déclarer les cookies en *HttpOnly*

Problèmes multi-langages - Cross-site scripting (XSS)

PHP

Pas bien

```
<?php echo "- $titre3[$i]"; ?>
```

Bien

```
<?php echo htmlentities("- $titre3[$i]"); ?>
```


Problèmes multi-langages - Cross-site scripting (XSS)

ASP.NET

Pas bien

```
Response.Write(Request.Form["name"]);
```

Bien

```
<%@ Page Language="C#" ValidateRequest="true" %>
...
Response.Write(HttpUtility.HtmlEncode(Request.Form["name"]));
```

Principe de l'attaque

CSRF = faire réaliser une action involontaire par l'utilisateur en étant connecté à un site :

1. l'attaquant trouve un CSRF et la chaîne pour l'exploiter
2. l'attaquant envoie un lien (mail, twitter, etc.) à la victime
3. la victime clique sur le lien malveillant (qui exploite le CSRF ou contient un formulaire automatique l'exploitant)
4. le site interprète la requête et l'exécute en pensant que c'est une action volontaire de la victime (si le cookie de session est valide)

Comment faire

Solutions

- ✓ générer un nombre unique pour chaque formulaire et le vérifier (attention à sa prédictabilité)
- ✓ réaliser une étape de confirmation pour les actions dangereuses (pas du coté client !), avec des CAPTCHA par exemple
- ✓ vérifier le *referer* HTTP

Problèmes multi-langages - Cross-site request forgery (CSRF)

Comment faire

Solutions

- ✓ utiliser une durée courte d'expiration de session
- ✓ ATTENTION : la plupart des méthodes de protection (nombre unique, vérification du referer, etc.) ne sont plus valables si le site est vulnérable aux XSS

Problèmes multi-langages - Injection de commandes système

Comment faire

Solutions

- ✓ filtrer les variables utilisées dans les fonctions dangereuses
- ✓ échapper convenablement les variables utilisées (`escapeshellcmd` en PHP par exemple)
- ✓ éviter les fonctions qui utilisent un shell (préférez `execl` en C)

Comment faire

Solutions

- ✓ en C, attention à `system`, `popen`, `execlp`, `execvp`, `ShellExecute`, `ShellExecuteEx`, `_wsystem`
- ✓ en PHP, attention à `system`, `popen`, `passthru`, `exec`, `shell_exec`, `proc_open`, `pcntl_exec`, backquote
- ✓ en Perl, attention à `system`, `exec`, backquote, `open`
- ✓ en Python, attention à `exec`, `os.system`, `os.popen`, `execfile`
- ✓ en Java, attention à `Runtime.exec`

PHP

Pas bien

```
exec($cmd." \"".$param_URI."\"",
     $results, $return);
```

Bien

```
exec($cmd." \"".escapeshellcmd($param_URI)."\"",
     $results, $return);
```

Problèmes multi-langages - Injection de commandes système

C

Pas bien

```c
char cmd[50];
strcpy(cmd, "/bin/cat");
strncat(cmd, argv[1], 40);
system(cmd);
```

Mieux (mais pas bien non plus)

```c
execl("/bin/cat", "cat", argv[1], NULL);
```


Problèmes multi-langages - Injection d'argument

Comment faire

Solutions

- ✓ filtrer les variables utilisées comme arguments pour des fonctions d'exécution (et pas que le premier caractère)
- ✓ utiliser -- avant les paramètres contrôlables par l'utilisateur

Comment faire

Solutions

- ✓ toujours filtrer les données avant d'utiliser des fonctions qui interprètent du code (`eval`, `include`, etc.)
- ✓ ne pas utiliser les fonctions d'interprétation de code dynamique (`eval`)

Comment faire

Solutions

- ✓ en PHP, attention à `eval`, `require*`, `include*`, `call_user_func`
- ✓ en Perl, attention à `eval`, `require`, `include`
- ✓ en Python, attention à `input`, `compile`,
- ✓ en Java, attention à `Class.forName`, `Class.newInstance`

Solutions

- ✓ vérifier la taille de toutes les chaînes !
- ✓ utiliser des alternatives sécurisées des fonctions dangereuses :
  - `gets` ⇒ `fgets` (l'octet nul est compris dans la taille)
  - `strcpy` ⇒ `strncpy` (sans octet nul final parfois)
  - `strcat` ⇒ `strncat` (l'octet nul n'est pas compris dans la taille)
  - `sprintf` ⇒ `snprintf` (l'octet nul est compris dans la taille)
  - famille `strl*` plus consistante
- ✓ vérifier tous les indexes
- ✓ compiler avec `/GS` ou `-fstack-protector`

Pas bien

```
char *get_pass(void)
{
  char *buf = malloc(16);
  if (!buf) return (NULL);
  printf("Enter password:\n");
  gets(buf);
  return (buf);
}
```

Bien

```
fgets(buf, 16, stdin);
```

Spécificités des langages - Débordement de tampon (buffer overflow)

Canaris

```
$ gcc test.c
$ ./a.out aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Erreur de segmentation
$ gcc -fstack-protector test.c
$ ./a.out aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
*** stack smashing detected ***: ./a.out terminated
======= Backtrace: =========
/lib/libc.so.6(__fortify_fail+0x37)[0x7fd29c474a07]
/lib/libc.so.6(__fortify_fail+0x0)[0x7fd29c4749d0]
./a.out[0x4005a5]
/lib/libc.so.6(__libc_start_main+0xb0)[0x7fd29c3afc00]
./a.out[0x400499]
======= Memory map: ========
[...]
```


Spécificités des langages - Chaînes de format (string format bug)

C/C++

Solutions

- ✓ ne pas laisser l'utilisateur contrôler une chaîne de format
- ✓ toujours utiliser `printf("%s", buf)`, même pour des chaînes constantes (évite le doute)

Spécificités des langages - Chaînes de format (string format bug)

C/C++

Pas bien

```
printf(buf);
```

Bien

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


Spécificités des langages - Int overflow

Comment faire

Solutions

- ✓ vérifier les valeurs des entiers avant de les utiliser
- ✓ vérifier toutes les tailles des types
- ✓ ne pas effectuer de comparaison entre des valeurs signées et non signées
- ✓ ne pas utiliser de variables signées quand ce n'est pas nécessaire

Pointeurs et objets

Solutions

- ✓ toujours initialiser les pointeurs à NULL
- ✓ toujours affecter la valeur NULL à un pointeur libéré
- ✓ toujours vérifier la nullitude d'un pointeur

Comment faire

Solutions

- ✓ éviter les variables globales (multithreads)
- ✓ utiliser `open` une seule fois et n'utiliser plus que le descripteur de fichier (fonctions `fchown`, `fstat`, etc.)
- ✓ vérifier le type de fichier avec `fstat` après ouverture, gérer les liens symboliques (surtout pour les SUID root)
- ✓ ne pas utiliser de répertoire en écriture pour tous
- ✓ créer des fichiers au nom aléatoire (utiliser `tmpfile` au lieu de `mkstemp` et surtout pas `mktemp`)

Comment faire

`mkstemp` maison :

```
char *filename;
int fd;
do {
  filename = tempnam (NULL, "foo");
  fd = open (filename,
             O_CREAT | O_EXCL | O_TRUNC | O_RDWR,
             0600);
  free (filename);
} while (fd == -1);
```


Définition

Problématique

Certains programmes réalisent des actions nécessitant des privilèges administrateur. Après avoir exécuté ces actions, le programme peut encore tout faire, il faut lui faire perdre les droits.

Le programme doit être lancé en administrateur : par celui-ci (ou le système sous cette identité) ou *setuid* sous Linux.

Spécificités des systèmes - Séparation de privilèges

Unix

Deux solutions :

- perdre totalement ses privilèges et changer d'utilisateur (après avoir exécuté les actions privilégiées) :
  - `setgroups`, `setegid`, `setgid`, `setregid` (selon Unix)
  - `seteuid`, `setuid`, `setreuid`
  - utiliser la fonction qui change de façon permanente l'utilisateur !
- se séparer en deux processus :
  - créer une socket pour communiquer (`socketpair`)
  - utiliser `fork` et changer d'utilisateur de façon permanente dans le fils
  - exécuter les actions dangereuses dans le fils (parsing, traitement, etc.), et faire exécuter par le père les actions nécessitant des privilèges

Spécificités des systèmes - Chroot

Unix

- restreindre le système de fichiers à une petite partie
- appel système `chroot` (nécessite d'être root) après avoir ouvert tous les fichiers nécessaires
- effectuer tout de suite `chdir("/");` pour le prendre en compte
- perte des privilèges ensuite pour éviter les techniques de *chroot breaking* (ne jamais garder l'identité root dans une `chroot`)

Spécificités des systèmes - Shatter attack

Comment faire

Solutions

- ✓ catcher les évènements dangereux (WM_TIMER, EM_SETWORDBREAKPROC, LVM_SORTITEMS, etc.) mais trop nombreux et inconnus
- ✓ ne pas créer de fenêtres pour les services privilégiés (processus utilisateur avec interface qui communique avec le programme privilégié, *via* un canal nommé)

Divers - Cryptographie

Comment faire

Solutions

- ✓ ne pas ré-implémenter d'algorithmes soi-même : utiliser des bibliothèques connues (openssl, celles des OS, etc.)
- ✓ proposer plusieurs algorithmes à l'utilisateur
- ✓ se renseigner sur l'utilisation des protocoles choisis
- ✓ se renseigner sur les bonnes pratiques (effacement de la mémoire, sources d'entropie, etc.)

Outils

C

Solutions

- ✓ compiler sous Visual Studio avec `/GS`, `/DYNAMICBASE`, `/NXCOMPAT`, `/SAFESEH` et Warning C4996
- ✓ compiler sous GCC avec `fstack-protector`, `Wl,pie`, `D_FORTIFY_SOURCE=2` et `Wformat-security`
- ✓ utiliser l'outil `RATS` (C, C++, Perl, PHP, Python), `splint` ou `cppcheck` (C++)

Outils

Perl

Solutions

- ✓ utiliser `use strict`
- ✓ utiliser l'option `-w`
- ✓ utiliser l'option `-T`
- ✓ utiliser `open` avec trois paramètres
- ✓ utiliser le module `Perl::Critic`

Outils
Python

Solutions

- ✓ ne pas utiliser le module `user` pour des scripts setuid
- ✓ utiliser l'outil `pychecker`

Outils
Ruby

Solutions

- ✓ utiliser la variable `$SAFE` ou l'option `-T`
- ✓ utiliser la commande `system` avec deux arguments

Solutions

- ✓ toujours désactiver `register_globals`
- ✓ toujours activer `magic_quotes_gpc`
- ✓ ne pas utiliser `$_REQUEST` mais `POST` ou `GET`
- ✓ écrire du code qui fonctionne avec le safe_mode
- ✓ ne pas reposer sur la sécurité du langage, mais tout vérifier soi-même (plus de safe_mode dans la version 6...)

Solutions

- ✓ ne pas placer de fichiers `.inc` contenant des informations sensibles dans l'arborescence (que des `.php` en vérifiant si le script est appelé directement ou par un `include`)
- ✓ utiliser les fonctions du langage (filtrage, gestion des sessions, etc.) et filtrer soi-même en plus

SANS Top 25 Most Dangerous Software Errors 2011 :

- *Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')*
- *Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')*
- *Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')*
- *Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')*
- *Execution with Unnecessary Privileges*
- *Cross-Site Request Forgery (CSRF)*
- *Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')*

SANS Top 25 Most Dangerous Software Errors 2011 :

- Inclusion of Functionality from Untrusted Control Sphere
- Use of Potentially Dangerous Function
- Incorrect Calculation of Buffer Size
- Uncontrolled Format String
- Integer Overflow or Wraparound