# Lesser Known Security Problems in PHP Applications

Stefan Esser

*Zend Conference*
*September 2008*
*Santa Clara, CA*

SektionEins

# The Speaker

Stefan Esser

- 8 years of PHP Core Experience
- 10 years of Security Experience
- Suhosin and The Month of PHP Bugs
- Founder and Head of R&D at SektionEins GmbH

# Topics

- Lesser Known Security Problems
- Less Obvious Exploitation Paths
- Inter Application Exploitation
- Vulnerability Classes Discovered during Real Audits

# The Mantra...

- Filter Input, Escape Output
  - often misunderstood
  - vulnerabilities hidden in input filters
  - wrong escaping / encoding functions
  - not every vulnerability is caused by tainted data

# Input Filtering - Short reminder

- Filter **what you actually use** and **not what you believe** is the same

```
<?php
   // The TikiWiki approach to input filtering

   if (!is_numeric($_REQUEST['id'])) {
      die('Hack attack');     // <-- will discuss this later
   }
   ...
   $_REQUEST = array_merge($_COOKIE, $_GET, $_POST);
   // ^------ really bad idea: GPC != CGP
?>
```

# $_SERVER and URL Encoding

- PHP_SELF and REQUEST_URI often used
- assumed to be URL encoded, but
  - PHP_SELF is never encoded (typical XSS)
  - REQUEST_URI encoding depends on client

```
<?php
   if ($_SERVER[‘REQUEST_URI‘] == ‘common.php‘) {
      die(“do not call this file directly“);
   }
   // File can still be requested by common%2ephp
?>
```

# $_REQUEST and Cookies

- never forget $_REQUEST **also contains** cookie data
- cookies or cookie data **might be unexpected**
    - **injected** through XSS, HTTP Response Splitting or other cross domain browser bug
    - **TLD wide** cookies - *.co.uk / *.co.kr
    - **originating from another** application on same domain

# $_REQUEST and Cookie DOS

- An **injected cookie** might kill the application

```
<?php
   // one cookie to kill them all
   if (isset($_REQUEST[‘GLOBALS‘])) {
      die(‘GLOBALS overwrite attempt‘);
   }
?>
```

# $_REQUEST and Delayed CSRF

- An **injected cookie** manipulates/overrides the control flow of a request performed by the user
- Traditional CSRF protections **useless**

```
<?php
   // save only modified admin options
   foreach ($_REQUEST[‘options‘] as $key => $val) {
      if (isset($options[$key]) && $options[$key] != $val) {
         saveOption($key, $val);
      }
   }
   // Because options[includePath] could be an evil cookie
   // there is a Delayed CSRF vulnerability
   // that allows remote file inclusion
?>
```

# auto_globals_jit - Documentation

```
; When enabled, the SERVER and ENV variables are created when they're first
; used (Just In Time) instead of when the script starts. If these variables
; are not used within a script, having this directive on will result in a
; performance gain. The PHP directives register_globals, register_long_arrays,
; and register_argc_argv must be disabled for this directive to have any affect.
```

*infamous documentation in php.ini*

# auto_globals_jit - Open Questions

- Documentation is correct ?
  - Almost definitely maybe (probably)
    - Ok, no
- What about $_REQUEST ?
- Is JIT really just-in-time of first usage ?

# auto_globals_jit - Reality

- Documentation is wrong
  - There is no just-in-time creation on first usage
  - auto_globals are usually created **before** the **start** of the script **if the compiler detects** their usage
  - or when an **extension requests** their creation
- The compiler just detects direct usage
  - access by variable-variables is **NOT** detected

# auto_globals_jit - Security Problem

- prepended input filtering using variable-variables **FAILS**
- auto_globals **do not exist** when the filter executes

```
<?php
   $filterTargets = array(‘_REQUEST‘, ‘_SERVER‘, ‘_ENV‘, ...);
   foreach ($filterTargets as $target) {
      $$target = filterRecursive((array)$$target);
   }
?>
```

- when a PHP script accesses the auto_globals they are created and filled with the **not filtered** values

# Session Handling - Insecure Cookie Parameters

- **very very** common problem
- sites use **SSL** to protect against session identifier sniffing
- but **forgets to mark** session identifier cookie **as secure**
- attacker **injects HTTP requests** to get **plaintext cookie**

# Session Handling - Session Data Mixup (I)

- session data is stored in **/tmp by default**
- **can be changed** by configuration
- session data is **shared by all applications** that store it in the same location
- **bad** for shared hosts
- but can also lead to **inter application exploits**

# Session Handling - Session Data Mixup (II)

- Example 1 - Setup:
  - customer runs two applications on his own server
  - both applications contain multi-step forms
  - both applications store data of previous steps in a session
  - application 1 merges user input into the session and validates/filters after all steps are processed
  - application 2 merges only validated and filtered data into the session

# Session Handling - Session Data Mixup (III)

- Example 1 - Exploit:
  - enter malicious content (XSS, SQL Inj.) into application 1
  - copy session identifier of application 1 into session cookie of application 2
  - use application 2 which trust everything within the session

➡ XSS payload from session eventually exploits application 2

# Session Handling - Session Data Mixup (IV)

- Example 2 - Setup:
  - customer runs two applications on his own server
  - both applications serve a separate group of users
  - both applications are written by the same developers
  - both applications share a similar implementation

# Session Handling - Session Data Mixup (V)

- Example 2 - Exploit:
  - attacker is a legit user of application 1 (maybe even a moderator / admin)
  - attacker logs himself into application 1
  - and copies his session identifier into the session cookie of application 2
  - because the implementation of the User object is shared, application 2 finds a valid User object in its session
  - attacker is now logged into application 2

# Session Handling - Session Data Mixup (V)

- Best Practices
  - store session data in different locations
    - ➡ `ini_set(“session.save_path“, “/tmp/application_1/“);`
    - ➡ `user space session handler`
  - embed application marker into the session
    - ➡ `if ((string)$_SESSION[‘application‘] !== ‘application_1‘) die();`
  - encrypt session data with application specific keys

# Session Handling - Insecure Transactions (I)

- some PHP applications choose to override the internal session management with a user space session handler
  - usual implementation
    - open    - ignored
    - read    - SELECT * FROM tb_sessions WHERE sid=:sid
    - write   - INSERT/UPDATE tb_sessions SET data=:data WHERE sid=:sid
    - close   - ignore
    - destroy - ignore

# Session Handling - Insecure Transactions (II)

- Usual implementation ignores that reading, updating and storing the session data forms a transaction
- Most applications with user space session handlers are vulnerable to session race conditions

# Database Handling - Status Quo

- SQL Injection widely known
- SQL Transactions less known and used
- SQL Errors are seldomly handled
- Input filters let overlong input through

# Database Handling - MySQL's max_packet_size

- max_packet_size configures **maximum size** of a packet
- anything bigger will **not** be sent
- **overlong input** can result in queries not being sent
- allows e.g. **disabling** logging queries
  - referer header
  - user-agent header
  - session-identifiers, ...

# Database Handling - Truncated Data

- database columns have a **maximum width**
- by default MySQL will **truncate any data** that doesn‘t fit

  from ‘admin           x‘

  to     ‘admin           ‘

- by default string comparision will **ignore trailing spaces**

➡ **Security Problem** because there are 2 admin users now

# Database Handling - Best Practices

- Use **database transactions** for application transactions
- Handle errors, assume **everything could fail**
- Use MySQL's sql_mode **STRICT_ALL_TABLES**
- **Catch** overlong input in input filtering

# Multi-Byte Encodings - A security problem?

- PHP uses backslash escaping in many places

  ➡ ( \ => \\,  ‘ => \‘, “ => \“ )

- backslash escaping is a problem for multi-byte parsers if the encoding allows backslashes as 2nd, 3rd, ... byte

- UTF-8 not affected, but several asian encodings like GBK, EUC-KR, SJIS, ...

SELECT * FROM u WHERE login='X\' OR id=1/*' AND pwd='XXXXXXXXXX'

will be parsed as

SELECT * FROM u WHERE login='X\' OR id=1/*' AND pwd='XXXXXXXXXX'

# Multi-Byte Encodings - Still a problem

- SQL-Injection
  - mysql_real_escape_string() **not safe when** SET NAMES is used
- Shell-Command Injection
  - PHP <= 5.2.6 **doesn‘t escape** shell commands for MB-locales
- Eval/Preg-Replace/Create_Function Injection
  - PHP **doesn‘t escape correctly** for zend_multibyte mode
- PHP Cache/Config Injection
  - var_export() **doesn‘t escape correctly** for zend_multibyte mode

# Multi-Byte Encodings - Special Case UTF-7

- UTF-7 is a 7 bit wide encoding
- Characters used -+A-Za-z0-9
- not handled by any of PHP's escape functions
- browsers can be tricked to parse pages as UTF-7 when no charset is given

➡ XSS vulnerabilities (also common on banking sites)

# Random Numbers

- Random Number Generators
  - srand() / rand()
    - Wrapper around libc's rand() - **32 bit Seed**
  - mt_srand() / mt_rand()
    - Mersenne Twister - **32 bit Seed**
  - uniqid(?, true) / lcg_value()
    - Combined linear congruential generator - **weak 64 bit Seed**

# mt_srand() / srand() - weak seeding

- PHP seeds automatically since 4.2.0
- Disadvantages of manual seeding
  - random number generator state is easier to predict
  - seeding influences other applications
  - manual seeding usually weaker than PHP’s seeding

```
<?php
   // examples for very bad seedings
   mt_srand(time());
   mt_srand(microtime() * 100000);
   mt_srand(microtime() * 1000000);
   mt_srand(microtime() * 10000000); //<- Joomla Password Reset
?>
```

# mt_srand() / srand() - Automatic seeding

- Automatic seeding in PHP <= 5.2.5
  - time(0) * PID * 1000000 * php_combined_lcg()
- on 32bit systems
  - lower bits of time(0) and PID can be **controlled**
  - due to modular arithmethic **product is 0** every 2.1 years
- on 64bit systems
  - **precision loss** during double to int conversion
  - strength around **24 bits**

# mt_rand() / rand() - weak random numbers

- numbers depend only on **32 bit seed** and **running time**
- **not suited** for cryptographic secrets
- output of PRNG might **leak state**
- state is process-wide => PRNG is **shared resource**
- attacker can get **fresh seed** by crashing PHP

# mt_(s)rand / (s)rand - Shared Hosting

- CGI
  - PRNG **freshly seeded** for every request
  - running time **not necessary** for prediction
- mod_php / fastcgi
  - PRNG is **shared** for requests handled by **same process**
    - **e.g. Keep-Alive**
  - Sharing **across VHOSTS**
  - **mean customer** can seed PRNG to attack others

# mt_(s)rand / (s)rand - Cross Application Attacks

- **applications share** the same PRNG
- **leak** in one application **allows attacking** another
- **seeding** in one application **allows attacking** another
  - phpBB2 seeds random number generator and leaks state
  - allows predicting password reset feature in Wordpress

# mt_(s)rand / (s)rand - Best Practices

- **do not seed** the PRNGs
- **do not use** PHP's PRNGs for cryptographic secrets
- **do not directly output** random numbers
- **combine output** of different PRNGs
- use **/dev/(u)random** on unix systems

# PHP's ZipArchive

- 0-day Vulnerability in PHP
- exposed by applications using ZipArchive
- discovered during an audit of customer code
- reported 85 days ago to PHP's security response team
- unpacking a malicious ZIP can overwrite any file
    - Exploit: just name archived files like ../../../../../www/hack.php

# HTTP Header Response Splitting/Suppression

- Protection against HTTP Response Splitting
  - introduced with PHP 5.1.2
  - not sufficient for old Netscape Proxies
  - suppresses headers containing recognized attacks
    - allows suppressing HTTP headers
    - security problem when Content-Disposition: attachment is suppressed

# The End ?!?

There are more unusual, lesser known and dangerous vulnerabilities, but we are running out of time...

# Thank you for listening

QUESTIONS ???