Security Problems in Web Applications except Injection Vulnerabilities

Stefan Esser <stefan.esser@sektioneins.de>
Ben Fuhrmannek <ben.fuhrmannek@sektioneins.de>
Who am I?

Ben Fuhrmann

- from Bonn/Germany
- Computer Scientist
- Professional Information Security since 2009
- @ SektionEins GmbH
- Focus on logical Flaws and esoteric Languages
Who we are?

Stefan Esser

- from Cologne/Germany
- Information Security since 1998
- joined the PHP Core Developers in 2001
- Suhosin / Hardened-PHP 2004
- Month of PHP Bugs 2007 / Month of PHP Security 2010
- Head of Research & Development at SektionEins GmbH
• vulnerabilities in web applications
• no injection vulnerabilities (XSS, SQL Injection, ...)
• vulnerabilities found during real audits
Part I

Plaintext vs. SSL and Certificate Verification
<?php
$data = file_get_contents("http://www.trusted.network/generate.php");
?>
External Data: Plaintext Protocol URLs

- many applications retrieve data from external sources
- this data is often trusted
- although it is requested through plaintext protocols
- man-in-the-middle attacks are possible

```php
<?php
$data = file_get_contents("http://www.trusted.network/generate.php");
?>```
<?php
$data = file_get_contents("https://www.trusted.network/generate.php");
?>
SSL Facts/History

- 1994: SSL v. 1.0 (Netscape/not public)
- 1995: SSL v. 2.0
- 1996: SSL v. 3.0
- 1999: TLS v. 1.0 (IETF)
SSL Facts/SSLv2 flaws

- 1 key für Auth. + Enc
- no handshake protection -> MITM
- MAC <- MD5
- TCP end = end of data
- MAC US export mode (40 bits)
SSL Facts/US Export Restrictions

- until 1992: U.S. Munitions List
- until 1996: commercial encryption
- until 1999: strong encryption (PGP)
SSL Facts/US Export Restrictions/Consequences

- Netscape: 2 versions: 40-bits (world) + 128-bits (US-only)
- PGPi Scanning Project (see http://www.pgpi.org/pgpi/project/scanning/)
- SSLv2 + Weak-Ciphers still supported
SSL facts/Weak Ciphers

$ openssl ciphers -v LOW:EXPORT

EDH-RSA-DES-CBC-SHA     SSLv3 Kx=DH       Au=RSA Enc=DES(56)   Mac=SHA1
EDH-DSS-DES-CBC-SHA     SSLv3 Kx=DH       Au=DSS Enc=DES(56)   Mac=SHA1
ADH-DES-CBC-SHA         SSLv3 Kx=DH       Au=None Enc=DES(56)   Mac=SHA1
DES-CBC-SHA             SSLv3 Kx=RSA      Au=RSA Enc=DES(56)   Mac=SHA1
DES-CBC-MD5             SSLv2 Kx=RSA      Au=RSA Enc=DES(56)   Mac=MD5
EXP-EDH-RSA-DES-CBC-SHA SSLv3 Kx=DH(512)  Au=RSA Enc=DES(40)   Mac=SHA1 export
EXP-EDH-DSS-DES-CBC-SHA SSLv3 Kx=DH(512)  Au=DSS Enc=DES(40)   Mac=SHA1 export
EXP-ADH-DES-CBC-SHA     SSLv3 Kx=DH(512)  Au=None Enc=DES(40)   Mac=SHA1 export
EXP-DES-CBC-SHA         SSLv3 Kx=RSA(512) Au=RSA Enc=DES(40)   Mac=SHA1 export
EXP-RC2-CBC-MD5          SSLv3 Kx=RSA(512) Au=RSA Enc=RC2(40)   Mac=MD5 export
EXP-RC2-CBC-MD5          SSLv2 Kx=RSA(512) Au=RSA Enc=RC2(40)   Mac=MD5 export
EXP-ADH-RC4-MD5          SSLv3 Kx=DH(512)  Au=None Enc=RC4(40)   Mac=MD5 export
EXP-RC4-MD5              SSLv3 Kx=RSA(512) Au=RSA Enc=RC4(40)   Mac=MD5 export
EXP-RC4-MD5              SSLv2 Kx=RSA(512) Au=RSA Enc=RC4(40)   Mac=MD5 export
SSL facts/Check for Weak Ciphers

openssl s_client -no_tls1 -no_ssl3 -cipher EXPORT \
   -connect bankingportal.sparkasse-koelnbonn.de:443

... 
New, SSLv2, Cipher is EXP-RC2-CBC-MD5 
Server public key is 2048 bit 
Secure Renegotiation IS NOT supported 
Compression: NONE 
Expansion: NONE 
SSL-Session:
   Protocol : SSLv2 
   Cipher   : EXP-RC2-CBC-MD5
...
SSLCipherSuite HIGH:MEDIUM:!SSLv2:!EXP:!aNULL:!eNULL
• stop using plaintext protocols
• so developers that heard about MITM just use https
• however PHP does not perform any checks by default
• without certificate verification SSL is still vulnerable to MITM attacks

```php
<?php
/* better but still insecure because no certificate verification */
$data = file_get_contents("https://www.trusted.network/generate.php");
?>
```
External Data: Using SSL securely...

- **PHP can verify SSL certificates** if instructed to do so
- requires a **configured stream context**
- can **verify different aspects**: arbitrary CA chain, self signed, CN name, ...

```php
<?php
    $ctx = stream_context_create();
    stream_context_set_option($ctx, "http", "max_redirects", 0);
    stream_context_set_option($ctx, "ssl", "verify_peer", true);
    stream_context_set_option($ctx, "ssl", "allow_self_signed", false);
    stream_context_set_option($ctx, "ssl", "CN_match", "www.trusted.network");
    stream_context_set_option($ctx, "ssl", "cafile", "/etc/perfectCA.pem");

    $data = file_get_contents("https://www.trusted.network/generate.php", false, $ctx);

?>
```
<?php

$ctx = stream_context_create();

stream_context_set_option($ctx, "http", "max_redirections", 0);
stream_context_set_option($ctx, "ssl", "verify_peer", true);
stream_context_set_option($ctx, "ssl", "allow_selfSigned", false);
stream_context_set_option($ctx, "ssl", "CN_match", "www.trusted.network");
stream_context_set_option($ctx, "ssl", "cafile", "/etc/perfectCA.pem");

stream_context_set_option($ctx, "ssl", "ciphers", "TLSv1:HIGH");

$data = file_get_contents("https://www.trusted.network/generate.php", false, $ctx);

?>
External Data: Paranoid SSL Cert Verification...

- **PHP can capture SSL certificates** for manual verification
- captured SSL certificate is **exported to user space**
- user space verification can e.g. **verify the subjectKeyIdentifier**

```php
<?php

$ctx = stream_context_create();

stream_context_set_option($ctx, "ssl", "capture_peer_cert", true);

$data = file_get_contents("https://www.sektioneins.de", false, $ctx);

$options = stream_context_get_options($ctx);

$cert = openssl_x509_parse($options['ssl']['peer_certificate']);
    die('illegal cert');
} else {
    die('certificate is okay');
}
```
Real-World Examples: SSL Connect-Back Auth.

1. Request
2. Connect
3. openssl verify
4. execute

Me
Update-Server
443
Certificate
Application
443
Reply
Part II

“Debugging” Features / Bugs
Debugging Features

- developers love debugging features
- however debugging features often allow crazy things
- and therefore should not be reachable by anyone except authorized debuggers
- should be removed on production systems
Secret Debugging Features

- found in 2009 during an audit
- developers had a super secret debugging switch
- gave access to some kind of admin console
- could be activated by adding "centauri=1" to the URL
- "There are no such things as wordlists"
Silverstripe’s SQL Functions (I)

- Silverstripe has lots of “debugging” features
- they use the Debug class to output debugging information
- Debug class actually validates that information is only sent to authorized debuggers

- However there is another problem, to you see it?

```php
public function query($sql, $errorLevel = E_USER_ERROR) {
    if(isset($_REQUEST['previewwrite']) &&
        in_array(strtolower(substr($sql, 0, strpos($sql, ' '))),
        array('insert', 'update', 'delete', 'replace'))
    ) {
        Debug::message("Will execute: $sql");
        return;
    }
}
```
Silverstripe’s SQL Functions (II)

- there will be no output because we are not a debugger
- but by simply setting a URL parameter we can disable the writes to the database
- very handy to stop login counters / logging / ...

```php
public function query($sql, $errorLevel = E_USER_ERROR) {
    if(isset($_REQUEST['previewwrite']) &&
        in_array(strtolower(substr($sql,0,strpos($sql,' '))),
               array('insert','update','delete','replace')))
    {
        Debug::message("Will execute: $sql");
        return;
    }
}
```
Silverstripe’s SQL Functions (III)

- but it is getting better
- the SQL query function continues with more debugging features

- Do you see the problem?

```php
$handle = mysql_query($sql, $this->dbConn);

if(isset($_REQUEST['showqueries'])){  
    $endtime = round(microtime(true) - $starttime,4);
    if (!isset($_REQUEST['ajax'])) Debug::message("\n$sql\n{\$endtime}ms\n", false);  
    else echo "\n$sql\n{\$endtime}ms\n";  
}  
```
Silverstripe’s SQL Functions (IV)

- URL parameter “showqueries” will output the SQL statement for debugging purposes
- Output is protected by Debug class - except for AJAX requests
- Tip: is very handy for secret token sent during password reset

```php
$handle = mysql_query($sql, $this->dbConn);

if(isset($_REQUEST['showqueries'])) {
    $endtime = round(microtime(true) - $starttime, 4);
    if (!isset($_REQUEST['ajax'])) Debug::message("\n\n$sql
\n{\n$endtime}ms\n", false);
    else echo "\n\n$sql
\n{\n$endtime}ms\n";
}
```
Part III

Trouble with Operators
Trouble with Operators

- every programming language has its own set of operators
- often similar but not identical
- e.g. PHP makes a difference between equal and identical
  - == / != equality operators
  - === / !== identical operators
- in security there is a huge difference between equal and identical
Equal vs. Identical (I)

- normal user input are strings
- therefore difference between == and === is believed to be small
- cannot be tricked most of the time

```php
if ($_POST['user'] == 'adminuser' && $_POST['pass'] == 'secretpass') {
    ...
}
```
Equal vs. Identical (II)

- new example
- still secure?

```php
$data = unserialize($_COOKIE['account']);
// or
$data = json_decode($_COOKIE['account']);

if ($data['user'] == 'adminuser' && $data['pass'] == 'secretpass') {
    ...
}
```
Equal vs. Identical (III)

- not secure
- result of unserialize() and json_decode() is typed
- equal operator can be tricked by booleans
  
  - a:2:{s:4:"user";b:1;s:4:"pass";b:1;}
  - {"user":true,"pass":true}

```php
$data = unserialize($_COOKIE['account']);
// or
$data = json_decode($_COOKIE['account']);

if ($data['user'] == 'adminuser' && $data['pass'] == 'secretpass') {
    ...
}
```
so is this code secure?

```php
$myuid = "100";

if ($_POST['uid'] == $myuid) {
    $db->query("UPDATE user SET pwhash=%s WHERE uid=%d",
                pwhash($_POST['newpass']), (int)$_POST['uid']);
} else {
    die("Trickster!");
}
```
• code is not secure
• can be tricked with the string “99.999999999999999999999999”
• password of previous user will be reset

```php
$myuid = "100";

if ($_POST['uid'] == $myuid) {
    $db->query("UPDATE user SET pwhash=%s WHERE uid=%d",
                pwhash($_POST['newpass']), (int)$_POST['uid']);
} else {
    die("Trickster!");
}
```
so is this code secure?

```php
if ($_POST['uid'] != 1) {
    $res = $db->query("SELECT * FROM user WHERE uid=%d", (int)$_POST['uid']);
    mail(...);
} else {
    die("Cannot reset password of admin");
}
```
Numbers and Equality (IV)

- not secure because of two reasons
- floating point numbers - "1.1"
- int cast truncates oversized numbers (32 bit system)

```php
if ($_POST['uid'] != 1) {
    $res = $db->query("SELECT * FROM user WHERE uid=%d", (int)$_POST['uid']);
    mail(...);
} else {
    die("Cannot reset password of admin");
}
```
Part IV

Script Interruptions
Spot the vulnerability...

- code like this was found in several different applications
- can you spot the security problem?

```php
<?php

/* remember previous user */
$prevUser = $_SESSION['user'];

/* temporary become admin */
$_SESSION['user'] = retrieveAdminUser();

/* prepare user for PW reset - required admin privs */
$token = prepareUserForPWReset($prevUser);

/* send out PW rest mail */
sendPasswordResetMail($prevUser['email'], $token);

/* restore previous user */
$_SESSION['user'] = $prevUser;
?
```
Script Interruption Vulnerability

- code is vulnerable to interruption while being temporary admin
- developer assumed that there is no interruption
- but interruption is possible
  - exceptions
  - FATAL errors
  - memory_limit interruption
- when interrupted the admin privileges are stored in the session
Script Interruption Defense

- avoid to do something that should not be interrupted
- handle errors correctly
- handle exceptions correctly
- disable memory_limit
Part V

unserialize() is not your friend
Analysing the Piwik Cookie (I)

- Piwik is a user tracking software similar to Google Analytics
- Tracking information is stored in a cookie that looks like this

piwik_visitor=8%3DMg%3D%3D%3A9%3DR29vZ2xI%3A10%3DYXN0YSBib25%3A4%3DYTE20ntp0jE7czozMjoiNTFjZTBhZDQyM2Y4Zjlk0DM0N2VmMTA5YzhkYTAt0AMDki02k6Mjtp0jEyODY0NjA4MjQ7aTozO2k6MTI4NjQ2MDgyNDtp0jQ7czo3%0A0iIxNjAx0DQ1Ijtp0jU7czo10i1MTcyNC17aToxMTtp0jA7fQ%3D%3D
Analysing the Piwik Cookie (II)

- Structure of cookie is
  
  \#id = base64 (value):

- So our cookie is

\[
\begin{align*}
8 &= \text{Mg} = : \\
9 &= \text{R29vZ2xl} : \\
10 &= \text{YXN0YSBib25u} : \\
4 &= \text{YTo20ntp0jE7czozMjoiNTFjZTBhZDQyM2Y4ZjlkODM0N2VmMTA5YzhkYTAxMDkiO2k6Mjtp0jEyODY0NjA4MjQ7aToz02k6MTI4NjQ2MDgyNDtp0jQ7czo3oiIxnjAxODQ1Ijtp0jU7czo10iI1MTcyNCI7aToxMTtp0jA7fQ=}
\end{align*}
\]
Base64 decoding reveals serialized PHP array

8 = 2
9 = Google
10 = asta bonn
4 = a:6:{i:1;s:32:"51ce0ad423f8f9d8347ef109c8da0109";i:2;i:1286460824;i:3;i:1286460824;i:4;s:7:"1601845";i:5;s:5:"51724";i:11;i:0;}

= 2
= Google
= asta bonn
4 = a:6:{i:1;s:32:"51ce0ad423f8f9d8347ef109c8da0109";i:2;i:1286460824;i:3;i:1286460824;i:4;s:7:"1601845";i:5;s:5:"51724";i:11;i:0;}

unserialize()

- allows to **deserialize** serialized PHP variables
- supports **most PHP data types** - including **PHP objects**
- when exposed to **user input can cause trouble**
- had **many internal vulnerabilities** in the past
unserialize() and PHP objects

- **deserializing objects** allows to control all **properties**
  - public
  - protected
  - private

- **but not the bytecode !!!**

- however **deserialized objects** get woken up **__wakeup()**

- and later **destroyed** via **__destruct()**

  ➡ **already existing code gets executed**
unserialize() and POP exploits

- if application has usable objects so called **POP exploits** are possible

- **POP = Property Oriented Programming**

- Idea behind POP
  - **start with** an object that has a `__wakeup()` or `__destroy()`
  - and then **hijack execution flow**
  - by carefully **filling properties** and **chaining existing methods**
  - **autoload** functionality helps a lot
Creating a POP Exploit for Piwik (I)

- Step 1 - Find classes useable for starting a POP chain
  - 8 classes from Zend Framework define __wakeup()
  - 11 classes from Zend Framework define __destruct()
  - 11 classes from Piwik’s core define __destruct()
class Zend_Log
{
    ...

    /**
     * @var array of Zend_Log_Writer_Abstract
     */
    protected $_writers = array();
    ...

    /**
     * Class destructor. Shutdown log writers
     */
    public function __destruct()
    {
        foreach($this->_writers as $writer) {
            $writer->shutdown();
        }
    }
}
Step 2 - Find classes that can continue the POP chain

- need to have a shutdown() method
- 6 classes from Zend Framework have a shutdown() method
- only ONE is interesting - Zend_Log_Writer_Mail
class Zend_Log_Writer_Mail extends Zend_Log_Writer_Abstract
{
    public function shutdown()
    {
        if (empty($this->_eventsToMail)) {
            return;
        }
        if ($this->_subjectPrependText !== null) {
            $numEntries = $this->_getFormattedNumEntriesPerPriority();
            $this->_mail->setSubject("{$this->_subjectPrependText} ({$numEntries})");
        }
        $this->_mail->setBodyText(implode('', $this->_eventsToMail));

        // If a Zend_Layout instance is being used, set its "events"
        // value to the lines formatted for use with the layout.
        if ($this->_layout) {
            // Set the required "messages" value for the layout. Here we
            // are assuming that the layout is for use with HTML.
            $this->_layout->events = implode('', $this->_layoutEventsToMail);

            // If an exception occurs during rendering, convert it to a notice
            // so we can avoid an exception thrown without a stack frame.
            try {
                $this->_mail->setBodyHtml($this->_layout->render());
            } catch (Exception $e) {
                trigger_error(...
                
            }
• **Step 3** - Find more classes that can **continue the POP chain**
  
  • one class needs to have `setBodyText()`/`setBodyHTML()` methods
    ➡ only **Zend_Mail** fits
  
  • another class needs to have a `render()` method
    • 14 classes of **Piwik’s core** match
    • 6 classes of the **HTML PEAR library** match
    • 21 classes of **Zend Framework** match
    • 1 **Piwik Plugin** matches
    ➡ ... after hard work ... **Piwik_View** is the most interesting one
class Piwik_View implements Piwik_iView {
    ...
    private $template = '';
    private $smarty = false;
    ...
    public function render() {
        try {
            $this->currentModule = Piwik::getModule();
            ...
            $this->loginModule = Piwik::getLoginPluginName();
        } catch (Exception $e) {
            // can fail, for example at installation (no plugin loaded yet)
        }

        $this->totalTimeGeneration = Zend_Registry::get('timer')->getTime();
        try {
            $this->totalNumberOfQueries = Piwik::getQueryCount();
        } catch (Exception $e) {
            $this->totalNumberOfQueries = 0;
        }

        @header('Content-Type: '. $this->contentType);
        ...

        return $this->smarty->fetch($this->template);
    }
}
Creating a POP Exploit for Piwik (III)

• **Step 4** - Find more classes that can **continue the POP chain**
  
  • need to have a `fetch()` method
  
  • 3 classes of **Piwik’s core** match (DB classes)
  
  • 1 class of the **Smarty library** matches
  
  • 8 classes of **Zend Framework** match (mostly DB classes)

  ➡️ ... after hard work ... **Piwik_Smarty** is the most interesting one
Piwik_Smarty (I)

class Smarty
{
    ...
    function fetch($resource_name, $cache_id = null, $compile_id = ..., $display = ...)
    {
        ...
        if ($display && !$this->caching && count($this->_plugins['outputfilter']) == 0) {
            if ($this->_is_compiled($resource_name, $_smarty_compile_path)
                || $this->_compile_resource($resource_name, $_smarty_compile_path))
                include($_smarty_compile_path);
        } else {
            ...
        }
    }

    function _is_compiled($resource_name, $compile_path)
    {
        ...
        // get file source and timestamp
        $_params = array('resource_name' => $resource_name, 'get_source'=>false);
        if (!$this->_fetch_resource_info($_params)) {
            return false;
        }
    }
};
class Smarty
{
    ...
    function _eval($code, $params=null)
    {
        return eval($code);
    }
    ...
    function _fetch_resource_info(&$params)
    {
        ...
        if ($this->_parse_resource_name($params)) {
            $resource_type = $params['resource_type'];
            $resource_name = $params['resource_name'];
            switch ($resource_type) {
                ...
                default:
                    // call resource functions to fetch the template source and timestamp
                    if ($params['get_source']) {
                        $_source_return = isset($this->_plugins['resource'][$resource_type]) &&
                            call_user_func_array($this->_plugins['resource'][$resource_type][0][0],
                                array($resource_name, &$params['source_content'], &$this));
                    } else {
                        ...
                    }
            }
        }
    }
}

_plugins['resource']['xpl'][0][0] = array($this, '_eval');
plugins['resource']['outputfiler'] = array();
Putting it all together...

Piwik_Smarty

_plugins = XPL Res Definition
debugging = false
debugging_ctrl = 123
caching = false
force_compile = true

Zend_Log

_writers

Zend_Log_Writer_Mail
_eventsToMail = array(1)
_subjectPrependText = null
_mail
_layout
_layoutEventsToMail = array(1)

Zend_Mail

Piwik_View

template = "xpl:phpinfo();die();"
smarty

Zend_Log_Writer_Mail

_eventsToMail = array(1)
_subjectPrependText = null
_mail
_layout
_layoutEventsToMail = array(1)
Part VI

Salted MD5/SHA1 vs. HMAC
Securing the Piwik Cookie with Hashes

- in PHP applications salted MD5 / SHA1 is used to stop tampering
- concatenating a secret and the data is supposed to be secure
- this is however a misbelief
- which one of these two is still an exploitable security hole?

```php
<?php
$secret = "T0Li!eR7&zFkT7";
$sHash = sha1($secret . $_POST['data']);
if ($sHash === $_POST['hash']) {
everythingIsFine();
}
?>

<?php
$secret = "T0Li!eR7&zFkT7";
$sHash = sha1($_POST['data'] . $secret);
if ($sHash === $_POST['hash']) {
everythingIsFine();
}
?>
```
Securing the Piwik Cookie with Hashes

- in PHP applications salted MD5 / SHA1 is used to stop tampering
- concatenating a secret and the data is supposed to be secure
- this is however a misbelief
- which one of these two is still an exploitable security hole?

```php
<?php
$secret = "T0Li!eR7&zEkt7";
$sHash = sha1($secret . $_POST['data']);
if ($sHash === $_POST['hash']) {
    everythingIsFine();
}
?>
```

```php
<?php
$secret = "T0Li!eR7&zEkt7";
$sHash = sha1($_POST['data'] . $secret);
if ($sHash === $_POST['hash']) {
    everythingIsFine();
}
?>
```

**security problem**

**safer but also not recommended**
MD5 / SHA1 / ...

- start with an IV
  - 0123456789ABCDEFFEDCBA9876543210 for MD5
  - 0123456789ABCDEFFEDCBA9876543210F0E1D2C3 for SHA-1
- work on blocks with fixed block size (64 byte for MD5/SHA1)
- repeat algorithm for each block of block size bytes
- preprocessing: pad the data to be multiple of block size
  - append bits 1 0 0 0 0 0 0 ... except last 64 bits
  - fill last 64 bits with input size
h1 = HASH($data)

h2 = HASH($data . $fake_padding . $payload)
can be calculated by using h1 as intermediate digest for block 1
Salted Hashes vs. HMAC

- salted hashes should not be used for authenticating data
- for authenticating messages one should use a MAC
- PHP implements HMAC in the hash extension
- HMAC is a salted hash of a secret plus another salted hash of the data

```php
<?php

$secret = "T0Li!eR7&zFkT7";

$sHMAC = hash_hmac("sha1", $_POST['data'], $secret);

if ($sHMAC === $_POST['hmac']) {
    everythingIsFine();
}
?>
```
Part VII

„Padding Oracle“
many developers just encrypt data to protect it
when a long encryption key is used this is considered secure
this is however sometimes a misbelief
in case of the CBC encryption mode a “padding oracle” can allow decryption and encryption without knowing the key

```php
<?php
$secretKey = "T0Li!eR7&zFkT7";

$cookie = openssl_decrypt($_COOKIE['piwik'], "aes-128-cbc", $secretKey);
?>
```
Securing the Piwik Cookie with Encryption

- many developers just encrypt data to protect it
- when a long encryption key is used this is considered secure
- this is however sometimes a misbelief
- in case of the CBC encryption mode a "padding oracle" can allow decryption and encryption without knowing the key

```php
<?php
$secretKey = "T0Li!eR7&zFkT7";
$cookie = openssl_decrypt($_COOKIE['piwik'], "aes-128-cbc", $secretKey);
?>
```
• block ciphers work on blocks of fixed size
• AES-128-CBC works on 16 byte blocks
• data has to be padded in order to encrypt it
• PKCS#5 pads with padding length bytes

Data To Crypt 05 05 05 05 05
What is a „Padding Oracle“?

- a “Padding Oracle” allows an attacker to determine if an encrypted block is correctly padded
- this might be a java application showing WrongPaddingException
- a timing attack
- or just a cookie that is not decrypted at all
- in case of Piwik this might be detected by the values returned in the new cookie

Illegally Padded Data

Decrypted Data 03 03 3A
CBC Decryption Diagram

- ciphertext is first decrypted with the key
- and then the previous ciphertext is XORed against it
- this setup allows to decrypt data with a “Padding Oracle”
Attacking CBC encryption (I)

captured encrypted data

```
- - - - - - - - - - - - - - - FE
```

decrypt

unknown decryption result

```
- - - - - - - - - - - - - - - ?
```

xor

attacker supplied data

```
- - - - - - - - - - - - - - - 56
```

fictive block with valid padding

```
- - - - - - - - - - - - - - - 01
```

we try all 255 possibilities

most probably 01 in case of a valid padding

encrypted byte

unknown

57 xor 01

most probably 01 in case of a valid padding
Attacking CBC encryption (II)

captured encrypted data

- - - - - - - - - - - - - - - 71 FE

encrypted byte

unknown decryption result

- - - - - - - - - - - - ? ?

unknown

attacker supplied data

- - - - - - - - - - - - - - - - 65 55

67 xor 02

fictive block with valid padding

- - - - - - - - - - - - - - - - 02 02

57 xor 02

we try all 255 possibilities

most probably 02,02 in case of a valid padding

xor

decrypt

captured encrypted data
Attacking CBC decryption (III)

captured encrypted data

\[
\begin{array}{cccccccccccccccccccccc}
\end{array}
\]

unknown decryption result

\[
\begin{array}{cccccccccccccccccccccc}
\end{array}
\]

attacker supplied data

\[
\begin{array}{cccccccccccccccccccccc}
\end{array}
\]

fictive block with valid padding

\[
\begin{array}{cccccccccccccccccccccc}
\end{array}
\]

decrypt

\[
\begin{array}{cccccccccccccccccccccc}
\end{array}
\]

xor

\[
\begin{array}{cccccccccccccccccccccc}
\end{array}
\]

\[
\begin{array}{cccccccccccccccccccccc}
\end{array}
\]

most probably 03, 03, 03 in case of a valid padding

we try all 255 possibilities

03 xor 03

03

10 xor 03

67 xor 03

57 xor 03
Wrapping it up

- “Padding Oracle” allows decryption of CBC encrypted data
- but encryption is also possible by doing it in a reverse fashion
- encryption leaves a prepended garbage block
- prefixing with captured ciphertexts to create valid data
Lesson learned...

- just encrypting data in CBC mode is insecure (with padding oracle)
- one should combine it with an additional HMAC
- HMAC validation before the decryption
- but best solution is still to NOT store valuable data in cookies

```php
<?php

$secretKey = "T0Li!eR7&zFkT7";

$sHMAC = hash_hmac("sha1", $_COOKIE['piwik'], $secretKey);

if ($sHMAC !== $_COOKIE['piwik_hmac']) {
    // cookie has been tampered with.
    $cookie = "";
} else {
    $cookie = openssl_decrypt($_COOKIE['piwik'], "aes-128-cbc", $secretKey);
}
?
```
QUESTIONS?

SektionEins

http://www.sektioneins.de