Web Security Essentials

Eli White

Vice President — One for All Events

@EliW

Web Security

Introduction

Course Information

This course has been created to discuss Web Security. Specifically as it relates to PHP Developers. Some of this course will be discussing PHP, some will be JavaScript, and some will touch on configuration & SQL.

This course assumes the developer has a working knowledge of the PHP, JavaScript, and SQL.

This course will not cover physical machine security, or network level intrusions. It is targeted at the developer.

Topics... lots of them

Securing Users
Filesystem & Database
Cookies
Session Hijacking
Passwords
Brute Force Attacks
2-factor authentication
Encryption/Hashing
Access Control

XSS (Cross Site Scripting)
CSRF (Cross Site Request Forgery)
Click-Jacking
SQL/Code/Command Injection
File Upload Vulnerabilities
Filter-Input Escape-Output
Preparing for Recovery
Online Resources
… and more …

Online vs Here

Just a note:

- Course designed originally as online for 10 hours
- We may skip a few things or cover some items briefly
- We will most likely work through examples live together, and skip things that were originally homework.

Please ask if there is anything you want covered more or less

System & User

Section 1

System Level Security

We mentioned that this course won't go over network or true 'system' level security, but there are a number of basic PHP/Web Server/Database things that we should cover.

Filesystem Security

Really comes down to one thing. Make sure that your webserver (and therefore PHP) does not run as ‘root’.

More so, make sure that the user it does run as, only has access to to your ‘web’ directory, and only read access, not write access, to the files. Only granting write access in specific needed cases.

NOTE:
This is commonly ignored advice, but offers great security in depth.

Database Security

Same advice as with filesystem. Make sure that the database user you are using for access only has the permissions that they need. (To specific databases)

Consider even making the normal user only have read access, and make separate connections with another user when you need to write.

PHP Configuration *(php.ini)*

```
display_errors = Off
display_startup_errors = Off
```

- Leave these off on your production server to not expose internal errors publicly, which will expose code & filesystem.

```
default_charset = "utf-8"
```

- Set your character set explicitly

```
session.use_cookies = 1
session.use_only_cookies = 1
session.cookie_httponly = 1
```

- We will discuss session security more, but set your system to use cookies only (and disallow JavaScript access). This greatly increases security of the sessions making them harder to hijack.

PHP 5.3 or Earlier *(php.ini)*

```
magic_quotes_gpc = Off
magic_quotes_runtime = Off
magic_quotes_sybase = Off
```

- Magic quotes were a great idea 10 years ago, now, not so.

```
register_globals = Off
```

- Registering the $_GET and $_POST into the global variable space can cause in-numerous untestable security flaws.

```
request_order = "GP"
```

- You should only use $_REQUEST when you truly have a need, and use $_GET and $_POST in typical practice. But for safety make sure that you remove 'C' ($_COOKIES) from it.

Additional PHP Thoughts

```
open_basedir = "/path/to/web/root"
```

- Forbids PHP from including files outside of this directory tree.

```
date.timezone = "UTC"
```

- Not directly a security flaw to have an unset/incorrect timezone, but might lead to incorrect date math, causing a vulnerability.

Removing .php extension via rewrites/framework

- Exposing which endpoints on your server are PHP is truly an extremely minor concern (as long as you stay up to date with security releases); however, it's ugly.

Cookie Security

The #1 rule for cookies, is to never store anything in a cookie that is able to be read & modified by users.

- IE: Never store: 'loggedin=true;user=bob'
- Hash/Encrypt/Obfuscate any data that you send to user
- Storing data in sessions, automatically handles this

Cookie Usage

```
bool setcookie ( string $name [, string $value [, int $expire = 0
                [, string $path [, string $domain [, bool $secure = false
                [, bool $httponly = false ]]]]]] )
```

Set your cookies only to the domain/path you need:

```
setcookie('uex', sha1($uid.'|'.$email), 3600, '/user', 'www.example.com');
```

Use secure-only cookies if on SSL:

```
setcookie('uex', sha1($uid.'|'.$email), 3600, '/user', 'www.example.com', true);
```

Set `$httponly` to disallow JavaScript access to PHP cookies:

```
setcookie('uex', sha1($uid.'|'.$email), 3600, '/user', 'www.example.com', true, true);
```

Exercise

Create functions that automatically handles setting & reading cookies in a more secure way for you:

- With a default timeout
- As httponly
- On HTTPS only, if detecting the request was HTTPS
- To your auto-detected subdomain only
- Bonus: Consider automatically obfuscating key names

http://php.net/setcookie

Session Security

Much of session security was already covered in PHP configuration. (No urlcookies, httponly sessions, etc)

Ensure session storage (filesystem, memcached, database) is secure. (PS. Don't use filesystem)

Consider changing your hash algorithm to something with less chances of collision:

```
session.hash_function = 1          ; sha1(), PHP 5.0+
session.hash_function = "sha512"   ; Any hash_algos(), PHP 5.3+
```

Password Hashing PHP 5.5

Also allows for upgrade paths for password security via the password_needs_rehash() function:

```
$options = [ 'cost' => 12 ];
if (password_verify($password, $hash)) {
    // Success - Log them in, but also check for rehash:
    if (password_needs_rehash($hash, PASSWORD_DEFAULT, $options)) {
        // The password was old, rehash it:
        $rehash = password_hash($password, PASSWORD_DEFAULT, $options);
        // Save this password back to the database now
    }
} else {
    // Failure, do not log them in.
}
```

User Permissions

Make sure that any user on your website is restricted to a minimum of the access that they need.

If you have a complicated system of permissions (IE, more than: Public, User, Admin), consider implementing Access Control Lists (ACL).

- Many frameworks provide this ability:
  - http://framework.zend.com/manual/2.2/en/modules/zend.permissions.acl.intro.html
  - http://symfony.com/doc/current/cookbook/security/acl.html

User Session Permissions

Always regenerate your session IDs when you have a change of permissions. (A user logs in, logs out, or re-authenticates as an admin):

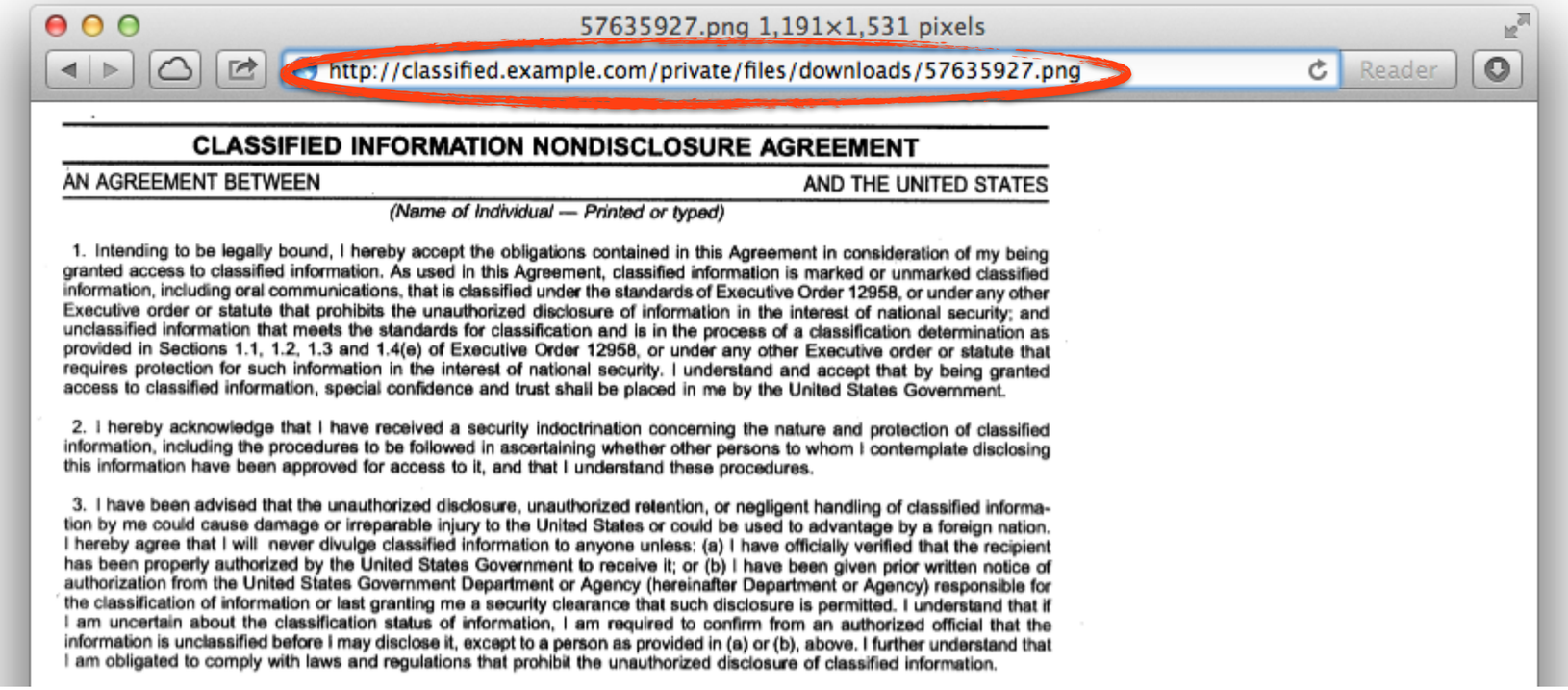
```
session_regenerate_id(true);
```

The ‘true’ parameter causes it to completely delete the old session.

Important to stop old sessions from hanging around.
(More on this later, in regards to Session Hijacking)

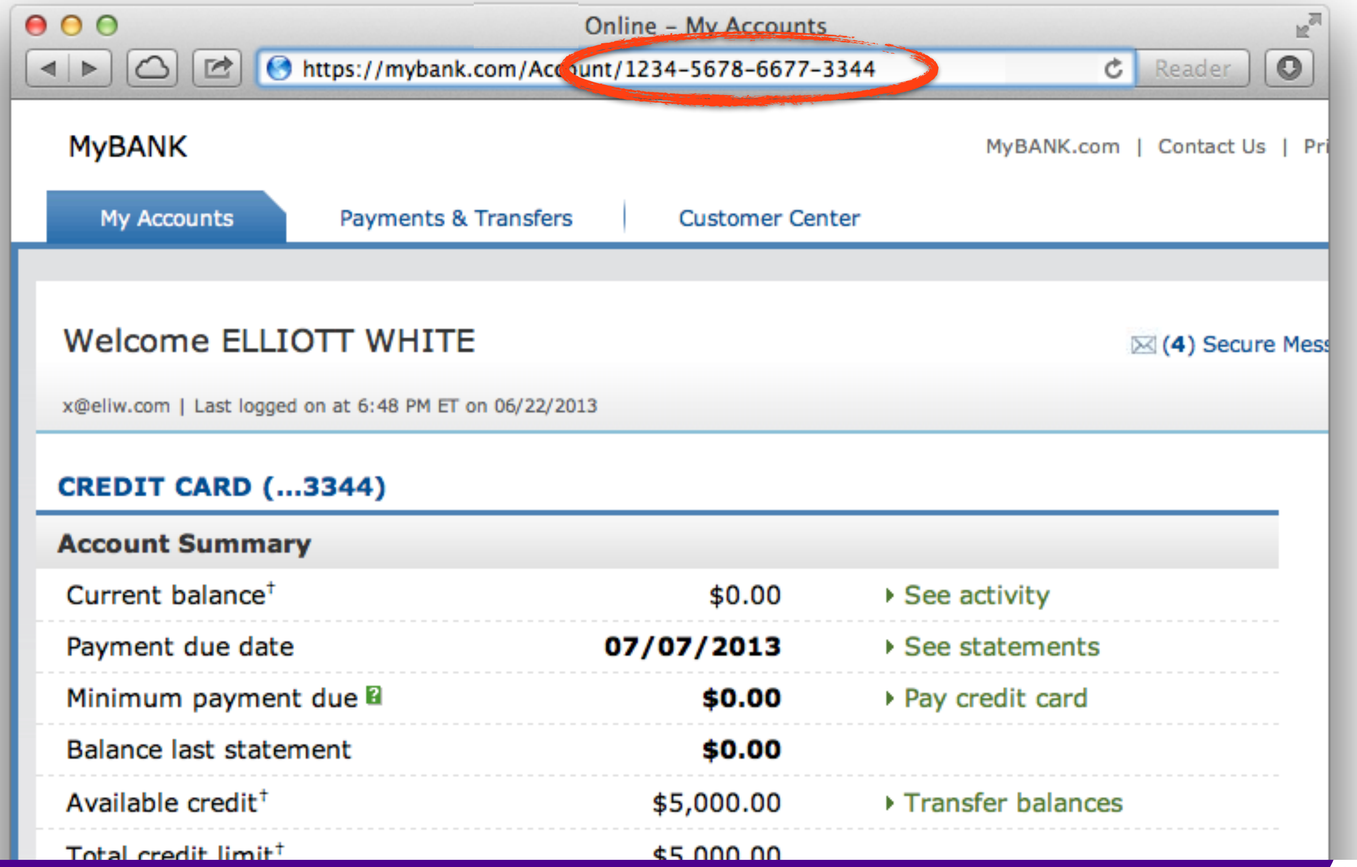
Avoid Simple Access Flaws

Direct URL access to a protected file



Avoid Simple Access Flaws

Ability to URL-hack to access unauthorized data.
Ensure user has rights to **that** data.



Best Password Practices

Rules for passwords:

- Do not restrict people from using letters, numbers, special characters, and most important, spaces.
- It's OK to have a minimum length (6?) but not max
- Up to you, if you want to require varied symbols. It does force the user into having a stronger password, but, it may encourage users to write their password down on a sticky note.

Password Hashing

Do not store plain text passwords, always 1-way hash.

Do not just use `md5()`, it is highly vulnerable to rainbow tables. Even `sha1()` is better, but...

The longer your hash takes to run, the longer (harder) it will be for someone else to try to crack it. It's OK if it takes a while to generate a password check.

Password Hashing PHP <5.5

Use a more secure algorithm, such as sha512:

```php
$str = "This is my secret data";
$hash = hash('sha512', $str);
```

Find a full list of supported algorithms via:

```php
var_dump(hash_algos());
```

Always generate & add a salt, to beat rainbow tables:

```php
$password = "MyVoiceIsMyPassport";

// Simple salt:
$salt = "PHP FOR LIFE";
$hash = hash('sha512', $salt . $password);

// More fancy & Unique
$salt = hash('sha1', uniqid(rand(), TRUE));
$hash = $salt . hash('sha512', $salt . $password);
```

Password Hashing PHP 5.5

PHP 5.5 has a built in `password_hash` function, that takes care of salting, has a configurable cost, and provides mechanisms for upgrading algorithms in the future:

```
string password_hash ( string $password , integer $algo [, array $options ] )
boolean password_verify ( string $password , string $hash )
```

Sample Usage:

```
$hash = password_hash('MyVoiceIsMyPassport', PASSWORD_DEFAULT);
$hash = password_hash('rootroot', PASSWORD_DEFAULT, ['cost' => 12]);
```

http://php.net/password

Secondary Measures

Many websites implement secondary measures...

Sometimes these are primarily to thwart phishing attempts. Such as showing a ‘known photo’ on login.

Other websites might ask for a second piece of information from the user’s profile, such as date of birth or 1st residence address, to verify that this wasn’t a stolen password.

These all have mixed effectiveness.

Brute Force Attacks

Brute force attacks involve either a computer, or a person, just attempting to gain access over and over again, either by guessing the password, or iteratively trying all of them.

Two best solutions to this, are CAPTCHA & Rate Limiting.

CAPTCHA

Don't build your own solution:
It won't be as comprehensive as a stock solution.
reCAPTCHA is free and easy to use.

http://recaptcha.org/
https://developers.google.com/recaptcha/docs/php

CAPTCHA Implementation

On the Form:

```
<?php require_once('recaptchalib.php'); ?>
<form method="POST" action="">
    <label>Username: <input name="user" /></label><br />
    <label>Password: <input name="pass" type="password"/></label><br />
    <?= recaptcha_get_html("YOUR-PUBLIC-KEY"); ?>
    <input type="submit" />
</form>
```

On the Server:

```
<?php
require_once('recaptchalib.php');
$check = recaptcha_check_answer(
    "YOUR-PRIVATE-KEY", $_SERVER["REMOTE_ADDR"],
    $_POST["recaptcha_challenge_field"], $_POST["recaptcha_response_field"]);

if (!$check->is_valid) {
    die("INVALID CAPTCHA");
} else {
    // Yay, it's a human!
}
```

Rate Limiting

Only allow so many failures, per IP address, over a predetermined period of time.

- Blocks scripts which won't wait
- Blocks repeated manual attempts

Be way of large amounts of people behind one IP

- Consider pros/cons of session based approach vs IP

Decide whether to block them visibly, or allow future attempts to silently fail.

Rate Limiting Example

IP-based Solution

```
$blocked = false;
$cachekey = 'attempts.'.$_SERVER['REMOTE_ADDR'];
$now = new DateTime();
$attempts = $memcached->get($cachekey) ?: [];
if (count($attempts) > 4) {
    $oldest = new DateTime($attempts[0]);
    if ($oldest->modify('+5 minute') > $now) {
        $blocked = true; // Block them
    }
}
if (!$blocked && $user->login()) {
    $memcached->delete($cachekey);
} else {
    array_unshift($attempts, $now->format(DateTime::ISO8601));
    $attempts = array_slice($attempts, 0, 5);
    $memcached->set($cachekey, $attempts);
}
```

Exercise

Build a purely session based Rate Limiter, instead of Memcached + IP checking.

- Store your data in the session.
- Base the ‘uniqueness’ on the session itself.

Remember Me

Most modern web applications want to provide a way for users to not have to log in every time they use it.

The solution are ‘Remember Me’ cookies.

Essentially create a unique key for this user, combining various aspects of their account, that is used as a secondary password.

Sample Implementation

```php
function tokenRememberMe($user) {
    $agent = $_SERVER['HTTP_USER_AGENT'];
    return implode('|', [$user->id, $user->username, $user->password, $agent]);
}

function hashRememberMe($user) {
    return password_hash(tokenRememberMe($user), PASSWORD_DEFAULT);
}

function setRememberMe($user) {
    $length = 7776000; /* 3 months */
    setcookie('Memory', $user->id . '|' . hashRememberMe($user), time()+$length);
}

function checkRememberMe() {
    if (empty($_SESSION['user_id']) && ($memory = $_COOKIE['Memory'])) {
        list($id, $hash) = explode('|', $memory);
        $success = FALSE;
        if ($id) {
            try {
                $user = new User($id); // Load the user:
                if (password_verify(tokenRememberMe($user), $hash)) {
                    $success = TRUE;
                }
            } catch (Exception $e) {}
        }
        if ($success) { /* Login User */ }
        else { setcookie('Memory', 0, time()-172800); }
    }
}
```

Extra Benefit

You can consider always setting a remember me cookie, even if the user doesn't choose it. But set it as a session cookie only.

Now if for some reason your underlying session disappears, the user is automatically logged back in.

Forgetful Users

Need a way for users to recover their accounts.

You can consider using questions (first pet, favorite band), but users often forget, and there have been public cases of people using these to break in.

Best practice is to create, save, and email (or SMS) the user a random one-use code, that they use to gain entry.

```
$code = hash('sha1', uniqid(rand(), TRUE));
```

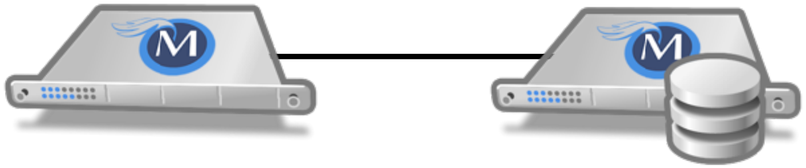
Exercise

Create a user class that mocks a real database backed version.

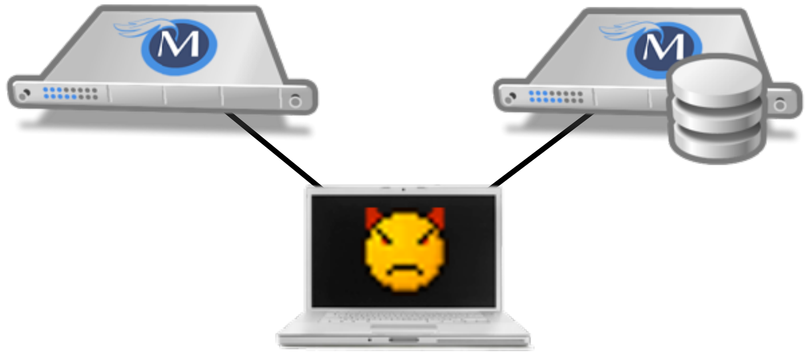
- Give it methods for saving & checking passwords.
- Give it methods for generating Remember Me cookies.
- Create a method for generating/storing/checking Forget Password Codes

Man in the Middle

Network level attack, where you think you are connecting to a known server:



But unknown to you, someone else has set up a machine between the server and your computer:



Solution: SSL

2-factor Authentication

Having something beyond username/password to ensure authentication is valid. Requires extra information that the user must have on them.

Originally involved keyfobs, or other physical devices that had to be plugged into the computer.

Today most commonly is done as sending an SMS to the user (Facebook), or via a token generator such as Google Authenticator for smartphones.

Implementing 2FA

SMS Method:

- Use Amazon SNS to send SMS upon login
- Use Twilio API to send SMS (or phone call)

Token Based 2FA:

- Use Google Authenticator Libraries:
  - http://code.google.com/p/ga4php/
  - https://www.idontplaydarts.com/2011/07/google-totp-two-factor-authentication-for-php/

Post-Class Exercise

Implement a 2-factor authentication scheme.

- Use either one of the SMS services, or the Google libraries, to generate some 2-factor authentication.

Filtering, XSS, and CSRF

Section 2

Filtering

Filtering Input

Standard Rule for security is **FIEO**:

- Filter Input, Escape Output
  - Filter the data that comes in, so that it's as expected
  - Escape the data going out, so that it's safe

Filtering input, is not 'direct' a security measure, but simplifies security later, and provides security in depth.

More layers means less chances of slipping through.

Sanitize vs Validate

Validate

- Check that the data is what was expected (an email address is an email address), and refuse if not.

Sanitize

- Attempt to convert the data into an expected value (convert any strings to integers)

S vs V: Pros & Cons

Validation

- Drawback is refusing data that could be figured out.
  - Declining ‘+44 020-7638-8811’ as a phone number because of dashes
  - Refusing ’42 towels’, when asking “How many?”, because non-int

Sanitization

- Drawback is accepting incorrect data
  - Converting ‘4.5’ to 4, when using intval()
  - Converting 'yes' to 0, when using intval()

There are appropriate times, for both options to be used.

Sanitizing: Expecting Input

There are a number of built-in type functions:

```php
$int = intval($input);
$flt = floatval($input);
$bol = boolval($input);
```

Be wary though, you may get unexpected results:

`boolval('no')` equals true;

Also some string functions that can help you:

```php
$str = trim($input);
$stp = strip_tags($input);
```

Validating via ctype_*

Validation can be done via regex, but `ctype_*` makes easy:

```
$test = ctype_alpha($input); // [A-Za-z] only
$test = ctype_alnum($input); // [A-Za-z0-9]
$test = ctype_digit($input); // [0-9]
$test = ctype_xdigit($input); // [0-9A-Fa-f]
```

Full list here: http://php.net/ctype

Also don't forget: handles all numeric formats, IE `+0123.45e6`

```
$test = is_numeric($input);
```

Sometimes you just gotta regex tho…

filter_var

First provided in PHP 5.2: http://php.net/filter

- Allows you to specify data, and a predefined filter.
- Provides both SANITIZE & VALIDATE style filters.

These functions operate on any variable or array of yours:

```
mixed filter_var (mixed $variable [, int $filter = FILTER_DEFAULT [, mixed $options ]])
mixed filter_var_array (array $data [, mixed $definition [, bool $add_empty = true ]])
```

These operate on the superglobal arrays, such as $_GET:

```
mixed filter_input (int $type, string $variable_name [, int $filter = FILTER_DEFAULT [, mixed $options ]])
mixed filter_input_array (int $type [, mixed $definition [, bool $add_empty = true ]])
```

Example

```php
<?php
/* Example POST:
$_POST = [
    'email'     => 'adent@example.com',
    'name'      => 'Arthur P. Dent',
    'meaning'   => '42',
    'favorites' => ['towel', 'lager', 'peanuts'],
    'website'   => 'http://hoopyfrood.com/',
    'username'  => 'arthurdent',
    'password'  => '******',
]; */

$args = [
    'email'     => FILTER_VALIDATE_EMAIL,
    'name'      => FILTER_SANITIZE_STRING,
    'meaning'   => ['filter'  => FILTER_VALIDATE_INT,
                    'options' => ['min_range' => 40, 'max_range' => 54] ],
    'favorites' => ['filter' => FILTER_SANITIZE_STRING,
                    'flags'  => FILTER_REQUIRE_ARRAY ],
    'website'   => FILTER_VALIDATE_URL,
    'username'  => ['filter' => FILTER_VALIDATE_REGEXP,
                    'options' => ['regexp' => '/^[A-Za-z0-9_]+$/'] ],
    'password'  => FILTER_UNSAFE_RAW,
];

$myinputs = filter_input_array(INPUT_POST, $args, TRUE);
```

Discussion: Versioned Filtering

Some people save both the raw input, and the filtered copy in the database, along with a version number.

Idea is that if you find a flaw in your filtering, you can create a new ‘version’, and re-apply it to the raw input.

Pros:

- Allows you the ability to protect yourself in the future more easily via re-filtering.

Cons:

- You are storing potential attacks in the database.
- You can always write post-fix scripts.

Exercise

Visit http://php.net/filter.filters and look through all the available filters:

- Experiment with the filters, see how they each work.
- Make a sample form to accept various items such as emails, urls, usernames, and random strings, experiment with the filters, filter options, and different inputs.
- **Bonus**: Consider the application of these into an automated framework, that takes all $_POST / $_GET input, applies filters, resaves the data in a $data response, and then calls unset() on the raw super-globals for safety.

XSS

XSS (Cross Site Scripting)

A user sending data that is executed as script

Many ways this attack can come in, but in all cases:
Everything from a user is suspect (forms, user-agent, headers, etc)
When fixing, escape to the situation (HTML, JS, XML, etc)

Don’t forget about rewritten URL strings!

Types of XSS

Reflected

- Directly echo'd back to the user.

Stored

- Saved (into the database) and later displayed to the user.

DOM

- Happens completely in JavaScript/Ajax.

XSS - Reflected XSS

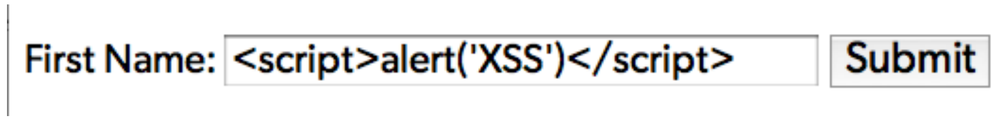
Reflected XSS

Directly echoing back content from the user

The Security Hole:

```
<p>Thank you for your submission: <?= $_POST['first_name'] ?></p>
```

The Attack:



XSS - Reflected XSS

Reflected XSS

Directly echoing back content from the user

The Solution (HTML):

```
$name = htmlentities($_POST['first_name'], ENT_QUOTES, 'UTF-8', FALSE);
```

The Solution (JS):

```
$name = str_replace(array("\r\n","\r","\n"),
            array("\n","\n","\\\n"),addslashes($_POST['first_name']));
```

The Solution (XML):

```
$name = iconv('UTF-8', 'UTF-8//IGNORE',
            preg_replace("#[\\x00-\\x1f]#msi", ' ',
            str_replace('&', '&amp;', $_POST['first_name'])));
```

Wait, why is this a problem?

The user can only hack themselves, right?

You are forgetting that users can be directed to your website (even with a POST), via clicking a link.

Also, users can be talked into anything, unfortunately.

Live Example

(Reflected XSS)

Exercise
Find the Bug

```
<html>
<head><title>Beta Website</title></head>
<body>
    <header>Account</header>
    <h1>Thanks for visiting our website</h1>
    <?php if (!count($_POST)): ?>
        <h2>Please let us know if you are interesting Beta Access:</h2>
        <form method="POST" action="">
            <label>Name: <input name="name" /></label>
            <label>Email: <input name="email" type="email" /></label>
            <label>Permission to Contact?
                <input name="contact" type="checkbox" value="yes" /></label>
            <input type="submit">
        </form>
    <?php else: ?>
        <h2>Thank you for your submission!</h2>
        <p>We have received the following submission and will contact
           you when our website is ready:</p>
        <ul>
            <li>Name: <?= $_POST['name'] ?></li>
            <li>Email: <?= $_POST['email'] ?></li>
            <li>Permission: <?= $_POST['contact'] ?: 'No' ?></li>
            <li>IP: <?= $_SERVER['REMOTE_ADDR'] ?></li>
            <li>Browser: <?= $_SERVER['HTTP_USER_AGENT'] ?></li>
        </ul>
    <?php endif; ?>
</body>
</html>
```

XSS - Stored XSS

Stored XSS
You store the data, then later display it

The Security Hole:

```php
<?php
$query = $pdo->prepare("UPDATE users SET first = ? WHERE id = 42");
$query->execute(array($_POST['first_name']));
?>

[...]

<?php
$result = $pdo->query("SELECT * FROM users WHERE id = 42");
$user = $result->fetchObject();
?>
<p>Welcome to <?= $user->first ?>'s Profile</p>
```

XSS - Stored XSS

Stored XSS
You store the data, then later display it

The Solution (HTML):

```
$name = htmlentities($_POST['first_name'], ENT_QUOTES, 'UTF-8', FALSE);
```

The Solution (JS):

```
$name = str_replace(array("\r\n","\r","\n"),
                    array("\n","\n","\\\n"),addslashes($_POST['first_name']));
```

The Solution (XML):

```
$name = iconv('UTF-8', 'UTF-8//IGNORE',
                       preg_replace("#[\\x00-\\x1f]#msi", ' ',
                       str_replace('&', '&amp;', $_POST['first_name'])));
```

The Same!

Live Example

(Stored XSS)

Exercise - Find the Bug

```
<html>
<head><title>My Website - <?= $page ?></title></head>
<body>
    <header class="<?= ($page == 'account') ?: 'active' ?>">
        Account (<?= $user->username ?>)
    </header>
    <h1>Welcome to <?= $user->fullname ?>'s Account</h1>
    <img src="<?= $user->headshot ?>" />
    <p><strong>BIO:</strong> <?= $user->bio ?></p>
    <p><strong>Latest Post:</strong> <?= $user->lastPost(); ?></p>
</body>
```

XSS - DOM XSS

DOM XSS

What happens in JavaScript, stays in JavaScript

The Security Hole:

```
<script>
$('#verify').submit(function() {
    var first = $(this).find("input[name=first]").val();
    $(body).append("<p>Thanks for the submission: " + first + "</p>");
    return false;
});
</script>
```

NOTE: JavaScript examples will use jQuery

XSS - DOM XSS

DOM XSS

What happens in JavaScript, stays in JavaScript

The Solution (Simple?):

```
<script>
function escapeHTML(str) {
    str = str + ""; var out = "";
    for (var i=0; i<str.length; i++) {
        if (str[i] === '<') { out += '&lt;'; }
        else if (str[i] === '>') { out += '&gt;'; }
        else if (str[i] === "'") { out += '&#39;'; }
        else if (str[i] === '"') { out += '&quot;'; }
        else { out += str[i]; }
    }
    return out;
}
</script>
```

Or just never directly echo in JS, always roundtrip to the server.

But you have to deal with attr vs HTML vs CSS etc
So use a library that handles encoding for you.

Live Example

(DOM XSS)

jQuery Encoder Usage

Since you need to escape output differently in JavaScript based upon whether it's being used as a tag name, CSS, attribute, class, etc.

One library that can help you is jQuery Encoder:

https://github.com/chrisisbeef/jquery-encoder/

Provided Methods:

```
encodeForCSS( String input, char[] immune )
encodeForHTML( String input )
encodeForHTMLAttribute( String input, char[] immune )
encodeForJavascript( String input, char[] immune )
encodeForURL( String input, char[] immune )
```

jQuery Encoder Example

```
<script>
$.post('/user/srv/account', { $('form').serialize() }, function(data) {
    $('#container').html(
        '<div style="background-color: ' +
            $.encoder.encodeForCSS(data.color) + '">' +
            $.encoder.encodeForHTML(data.html) +
        '</div>');
    });
</script>
```

Exercise

Find the Bug

```
<html>
<head><title>Eli's Account</title></head>
<body>
    <h1><strong>User</strong>: EliW</h1>
    <ul>
        <li>Name: <span class="editable">Eli White</span></li>
        <li>Company: <span class="editable">musketeers.me</span></li>
    </ul>
<script src="lib/jquery-2.0.3.min.js"></script>
<script>
$(document).ready(function() {
    $('ul').on('click', '.editable', function() {
        var $orig = $(this);
        var $input = $('<input />').val($orig.text());
        $input.on('blur keydown', function(e) {
            if (!e.keyCode || (e.keyCode == 13)) {
                var $input = $(this);
                var val = $input.val();
                $input.replaceWith('<span class="editable">'+val+'</span>');
            }
        });
        $orig.replaceWith($input);
    });
});
</script>
</body>
</html>
```

CSRF

CSRF (Cross Site Request Forgery)

A user having the ability to forge or force a request on behalf of another user.

Simplistically via IMG tag or POST forms

Complicated via JavaScript

CSRF (Cross Site Request Forgery)

A user having the ability to forge or force a request on behalf of another user.

The Attack:

```
<img width="1" height="“1" src="http://quackr.example.com/quackit?msg=CSRF+Attacks+Rock!" />
```

or

```
<form method="POST" action="http://remote.example.com/endpoint">
    <input type="hidden" name="msg" value="CSRF Attacks Rock!" />
    <input type="submit" name="Super Awesome Button" />
</form>
```

or

```
<script>
$.post({
    url: 'http://quackr.example.com/quackit',
    data: { msg: 'CSRF Attacks Rock!'}
});
</script>
```

CSRF (Cross Site Request Forgery)

Protect via CSRF token

The Solution (on form):

```php
<?php
function generateToken() {
    $token = empty($_SESSION['token']) ? false : $_SESSION['token'];
    $expires = empty($_SESSION['tExpires']) ? false : $_SESSION['tExpires'];
    if (!$token || ($expires < time())) {
        $token = sha1(uniqid(mt_rand(), true));
        $_SESSION['token'] = $token;
    }
    $_SESSION['tokenExpires'] = time() + 14400;
    return $token;
}
?>
<form method="POST" action="">
    <input name="msg" value="" />
    <input type="hidden" name="token" value="<?= generateToken() ?>" />
    <input type="submit" />
</form>
```

CSRF (Cross Site Request Forgery)

Protect via CSRF token

The Solution (on submission):

```php
<?php
$token = empty($_SESSION['token']) ? false : $_SESSION['token'];
$expires = empty($_SESSION['tExpires']) ? false : $_SESSION['tExpires'];
$check = empty($_POST['token']) ? false : $_POST['token'];

if ($token && ($token == $check) && ($expires > time())) {
    // SUCCESS - Process the form
} else {
    // FAILURE - Block this:
    header('HTTP/1.0 403 Forbidden');
    die;
}
?>
```

Live Example

(CSRF)

Post-Class Exercise

Create your own toolkit (a library or class), that handles CSRF token management for you. Give the ability to:

- Create a new token, based on a specific key name
- Generate the <input> or JavaScript variable for you
- Validate the token on return
- **BONUS**: Create a wrapper for your Ajax calls, that automatically adds in the CSRF token.

Additional Concerns

Section 3

Additional Attacks

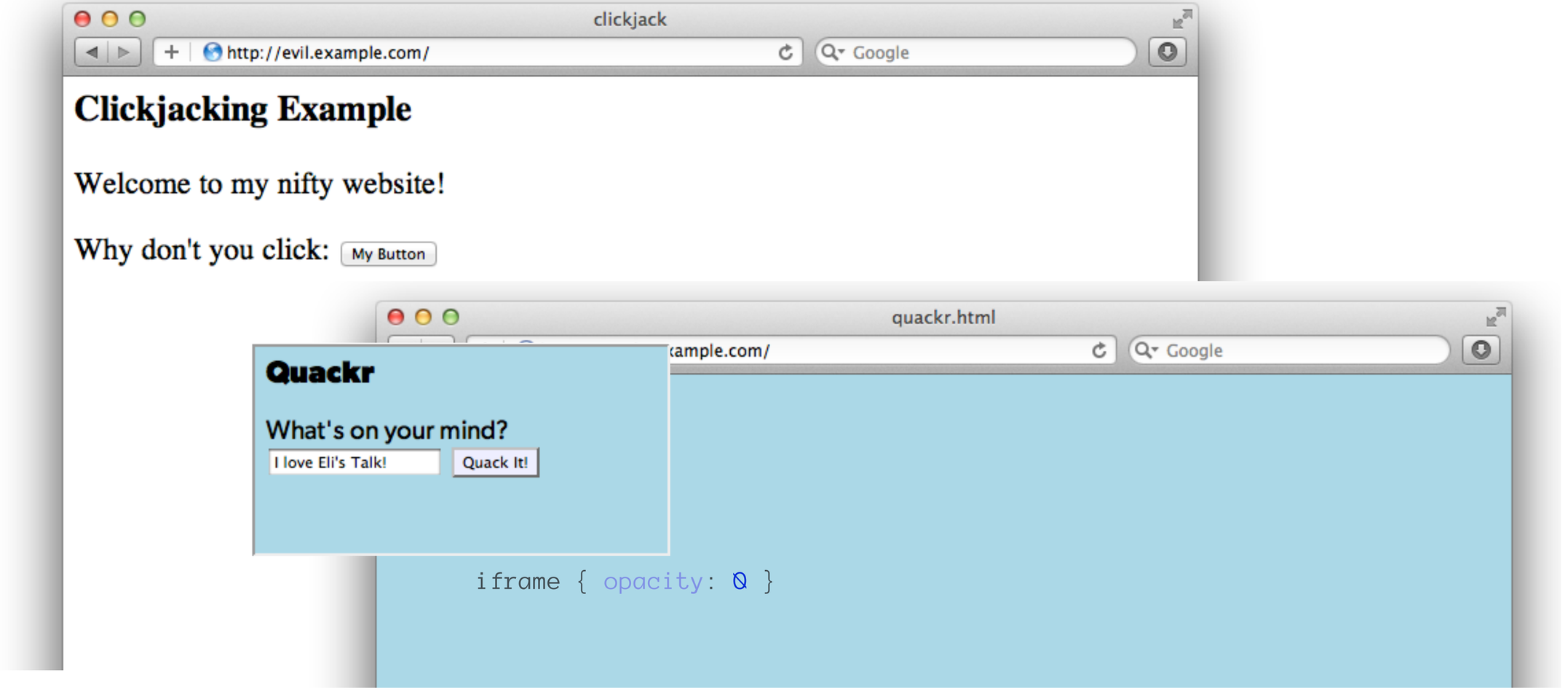
No less scary than before, just more focused

Clickjacking

Publicly hit the scene in 2008 when Twitter was hacked

Involves tricking the user into actually physically making a click on a remote website, without them realizing they did so. Thereby getting around any CSRF protection. This is hard to explain, so let's watch a demo:

Clickjacking Demonstration



Clickjacking - Solution 1

Use specific header, to disallow site framing:

The Solution:

```
header('X-Frame-Options: DENY');
```

or

```
header('X-Frame-Options: SAMEORIGIN');
```

Doesn't work in all browsers!

Became IETF standard RFC 7034 in October 2013

Clickjacking - Solution 2

Ensure you aren't displayed in iFrame

The Solution:

```
<html>
    <head>
        <style> body { display : none;} </style>
    </head>
    <body>
        <script>
            if (self == top) {
              var theBody = document.getElementsByTagName('body')[0];
              theBody.style.display = "block";
            } else {
              top.location = self.location;
            }
        </script>
    </body>
</html>
```

Session Hijacking

One user 'becoming' another by taking over their session via impersonation.

Avoid "Session Fixation"
Don't use URL cookies for your sessions.

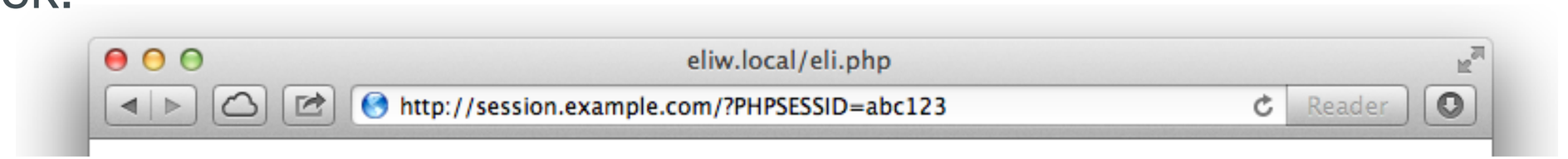
Always regenerate Session IDs on a change of access level.

Save an anti-hijack token to another cookie & session. Require it to be present & match. Salt on unique data (such as User Agent)

Session Fixation

A user being able to provide a known session ID to another user.

The Attack:



The Solution:

```
session.use_cookies = 1
session.use_only_cookies = 1
```

Don't use URL cookies for your sessions.

Session Fixation (Take 2)

Protect from more complicated fixation attacks, by regenerating sessions on change of access level.

The Solution:

```
session_start();
if ($user->login($_POST['user'], $_POST['pass'])) {
    session_regenerate_id(TRUE);
}
```

and

```
session_start()
$user->logout();
session_regenerate_id(TRUE);
```

Session Anti-Hijack Measures

Finally use anti-hijack measures to ensure user is legit

The Solution:

Not a few lines of code.
Store whatever unique you can about this user/browser combination and verify it hasn't changed between loads.

Note that IP changes or can be shared.
As happens with most other headers too.

Session Anti-Hijack Measures

```php
private function _sessionStart() {
    session_start();
    if (!empty($_SESSION)) { // Session not empty, verify:
        $token = $this->_hijackToken();
        $sh = empty($_SESSION['hijack']) ? NULL : $_SESSION['hijack'];
        $ch = empty($_COOKIE['data']) ? NULL : $_COOKIE['data'];
        if (!$sh || !$ch || ($sh != $ch) || ($sh != $token)) { // Hijacked!
            session_write_close();
            session_id(sha1(uniqid(rand(), TRUE)));
            session_start();
            setcookie('data', 0, -172800);
            header("Location: http://www.example.com/");
        }
    } else { // Empty/new session, create tokens
        $_SESSION['started'] = date_format(new DateTime(), DateTime::ISO8601);
        $_SESSION['hijack'] = $this->_hijackToken();
        setcookie('data', $_SESSION['hijack']);
    }
}

private function _hijackToken() {
    $token = empty($_SERVER['HTTP_USER_AGENT']) ? 'N/A' : $_SERVER['HTTP_USER_AGENT'];
    $token .= '| Hijacking is Bad mmmkay? |'; // Salt
    $token .= $_SESSION['started']; // Random unique thing to this session
    return sha1($token);
}
```

SQL Injection

A user having the ability to send data that is directly interpreted by your SQL engine.

The Security Hole:

```
$pdo->query("SELECT * FROM users
             WHERE name = '{$_POST['name']}' AND pass = '{$_POST['pass']}'");
```

The Attack:

```
$_GET['name'] = "' or 1=1; //";
```

SQL Injection

A user having the ability to send data that is directly interpreted by your SQL engine.

The Solution:

```php
$query = $pdo->prepare("SELECT * FROM users WHERE name = ? AND pass = ?");
$query->execute(array($_POST['name'], $_POST['pass']));
```

or

```php
$name = $pdo->quote($_POST['name']);
$pass = $pdo->quote($_POST['pass']);
$pdo->query("SELECT * FROM users WHERE name = {$name} AND pass = {$pass}");
```

Live Example

(SQL Injection)

Exercise - Fix the Hole

```
if (isset($_POST['type']) && isset($_POST['id'])):
    $type = $_POST['type'];
    $id = $_POST['id'];
    $results = db()->query(
        "SELECT *
         FROM {$type}
         WHERE id = {$_POST['id']}
         ");
    $data = $results->fetchObject();
}
```

Command Injection

The user being able to inject code into a command line.

The Security Hole:

```
$output = `convert {$_POST['option']}.jpg thumbnail.png`;
```

The Attack:

```
$_POST['option'] = "; rm -rf /;";
```

The Solution:

```
$option = escapeshellarg($_POST['option']);
```

Code Injection

The user being able to inject and execute raw PHP code.

The Security Hole:

```
eval("{$_POST['area']}::authenticate({$_SESSION['user']});");
```

The Attack:

```
$_POST['area'] = "rename('/etc/passwd', '/home/www/xyz.txt');";
```

The Solution:

Don't use eval()! **Eval == Evil**

Unchecked File Uploads

The user being able to upload executable files or scripts.
Or in some cases, being able to trick you into treating an existing file on your system, as if it was an upload.

The Security Hole:

```
rename($_FILES['up_img']['tmp_name'], ‘/images/‘.$_FILES['up_img']['name']);
```

The Solution:

```
if ($_FILES["up_img"]["error"] == UPLOAD_ERR_OK) {
    $tmp_name = $_FILES["up_img"]["tmp_name"];
    $name = $_FILES["up_img"]["name"];
    $extension = pathinfo($tmp_name, PATHINFO_EXTENSION);
    if (in_array($extension, ['gif', 'jpg', 'png']) {
        move_uploaded_file($tmp_name, "/my/images/{$name}");
    }
}
```

Exercise

Implement a file upload system that properly handles safely processing the file uploaded via POST:

- Documentation: http://php.net/file-upload
- Ensure your form uses enctype="multipart/form-data"
- Force the system to only accept certain files (ie Images)
- Allow the user to specify a ‘category’ & save the files into a separate directory per categeory.
- **BONUS**: Investigate the use of MAX_FILE_SIZE
- **BONUS**: Create a viewer to show/link to all uploads

Preparation

For when you get the 2am phone call…

Three Stages of Security

Prevention

Preparation

Panic

Discovery Methods

In descending order of usefulness:

- Whitehat contacts you
- Hacker announces it, and exploits it
- Hacker just announces it
- User complaints
- Physical Demarcation
- Logs/Stats monitoring

Immediate Response

Three options: Completely depends on the situation:

Let it Live
Break Functionality
Shutdown Website

Be prepared to make a quick decision!

On Breaking Functionality

It's become common practice, for large websites, to isolate each of their website features.

- They are enabled, via a configuration file.
  - Some go one step farther and allow a % of enable.

- Not only easily allows you to turn off a security hole, but allows you great flexibility in releasing & testing new code, or temporarily disabling features during high load.

Used by Facebook & Etsy for example.

Finding the Breach

So you now know that you have a security hole…

You need to look for the specific cause, from knowing all the aforementioned problems that we've discussed.

Use all the clues that you can find, based upon the visible traces left behind by the hacker.

If...the Hacker Posted It

This is a good situation (in a way), you supposedly have the exploit sitting live on a page, and you have access to that page:

- View the Page Source
- View the Generated Source
- Scan all included JavaScript and .js files
- Look for any iFrames

Error Logs

Make sure that you have a PHP Error Log enabled, and preferably sent into a separate file to look at easily.

Some attacks might cause PHP errors, allowing you insight into what the attacker was trying. Plus it's always good to log **all** errors, so that you can find bugs before your users do.

```
error_reporting -1
error_log "/var/logs/php.error.log"
```

Exceptions & Services

Services:

- Consider logging any connection failures to other services that you run (such as caching), or errors returned from external web services that you use. They are useful in general & might be a source of security info.

Exceptions:

- Consider implementing a generic exception handler to catch any un-handled ones, and log the result with as much information as you can for similar reasons.

Action Logs

An action log – is a separate log you keep, tracking actions taken by users with as much detail as possible, this might be logging in, adding a friend, editing their profile, or just anything that you feel is useful to log for your application:

- Track: Who, When, Action Taken, Remote IP, Refer, etc
- Gives you an absolute source, to watch a tricky action.

Keep a Failed SQL Log

You absolutely need to log any SQL query that fails.

- (The same can be said for NoSQL solutions as well)

- Most everything ends up in the database.
- Hackers will not succeed on their first attempt.
- This will created malformed SQL commands
- Scan for common XSS terms (script, onclick, etc)

Exercise

We've discussed a number of types of logging. To handle this, we will need to build a generic logging system:

- Make a function that handles writing messages to log files.
- Upon each call, allow specifying a 'type' of log message
  - Write each type into a separate log file.
- Allow specifying a 'severity level' to the log message.
  - Make what level is logged vs ignored configurable.
- Automatically add a timestamp to each entry.
- **BONUS**: Build this as a class with constants for severity.
- **BONUS**: Create an exception handler that auto-logs exceptions.
- **BONUS**: Create a database query wrapper that logs failures.
- **EXTENDED**: Refactor to use other storage (such as database)

HTTP Logs are your Friend

Make sure that you are keeping them!

You can browse the logs to find evidence of the hack.

- Limit your search to a known timeframe
- Look for odd or out of sequence events
- Scan for GET parameters that are improper
- Scan other fields, such as Refer and UserAgent

Scan your database for bad data

Look in the database for any bad data that somehow made it through your filtering, or could be the culprit when not escaped.

Query against any user-supplied fields, for common XSS terms:

- script, onclick, javascript:, onfocus, onload, etc

Obscure Obscurities

You've searched through all the basics places, you still can't find the security breach. Now what? Familiarize yourself with the more obscure things:

- Look at other user supplied data sources:
  - Refer, User Agent, Cookies, Custom URLs …
- Encoding issues:
  - UTF-8 (or not), Converting between encodings …
  - htmldecode() after escaping, double escaping …
  - Encoded JavaScript strings

Resources

http://owasp.org/

- A great resource, always has up-to-date security exploits
- Be wary of the ‘solutions’ though, user-edited wiki

http://phpsec.org/

- Good information, though unfortunately out of date.

http://phptherightway.com/ — http://phpbestpractices.org/ — http://phpdeveloper.org/

- Not security specific resources but are great general information sources on PHP and include security as well.

Questions?

For this presentation & more:
eliw.com

Twitter: @EliW

One for All Events:
www.oneforall.events

Thanks!

Please send us feedback on how this class went to:

training@phparch.com

If you are interested in extending your learning experience, we offer many other core PHP programming courses from beginner to expert.

We also produce books on PHP, our premier PHP magazine covering all topics related to Web Development, host online summits and run conferences.

Visit us at http://www.phparch.com/ for more information.