Web Security Essentials

By: Eli White

CTO & Founding Partner:
musketeers.me

Managing Editor & Conference Chair:
php[architect] - phparch.com

eliw.com - @eliw

php[architect]

About Security

Do we really need to worry about this?

Security? Bah!

Whether big or small. Someone will try to hack you!

It only takes one person!

The Open Web Application Security Project

http://owasp.org/

The best online resource for learning about various attack vectors and solutions to them.

Use good judgement though, often wiki-user edited 'solutions'.

System Level Security

We won't go over this in depth, but a few notes …

Filesystem Security

Make sure your web server does not run as 'root'

The user it runs as should only have access to the 'web' directory

Commonly ignored, but offers great security-in-depth

Database Security

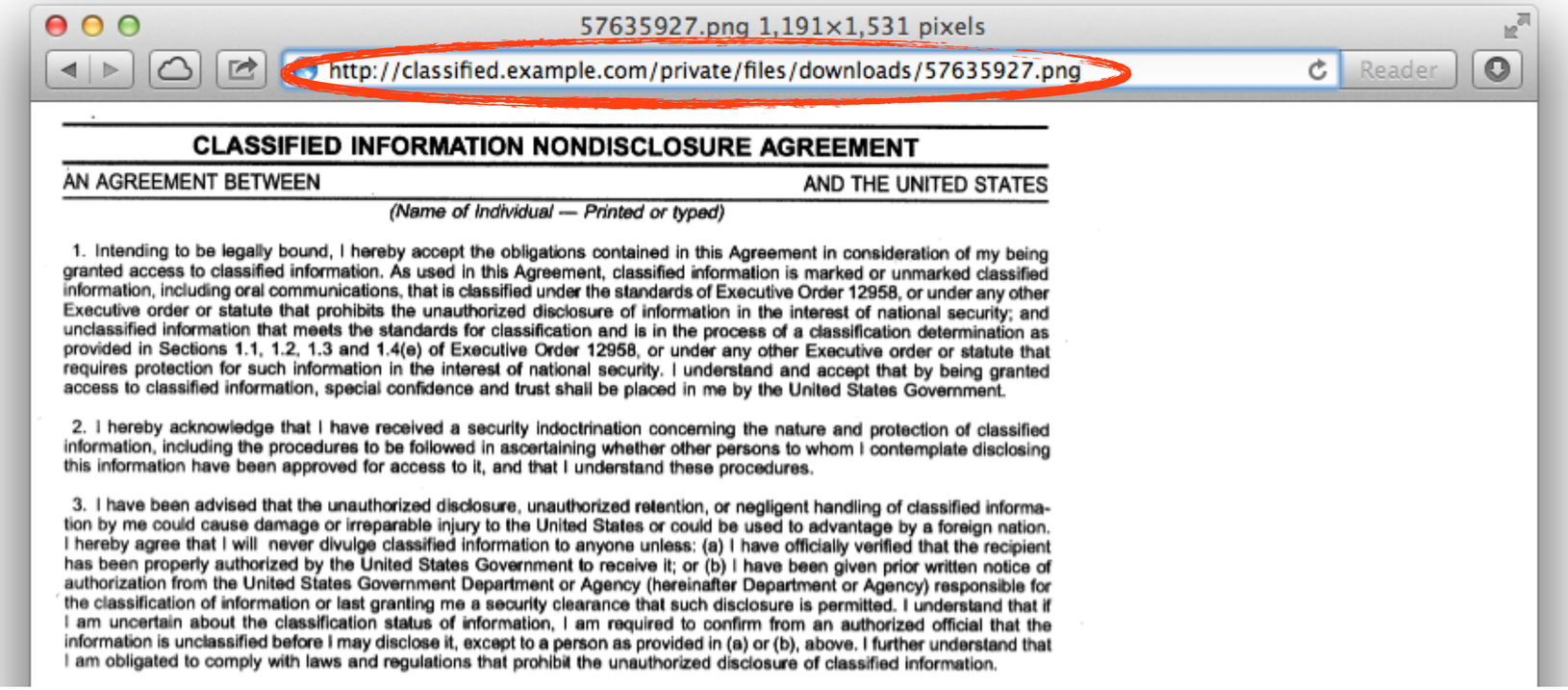
Same advice:
Make sure the database user only has permissions that it needs.

Consider:
Make the normal DB user only have read access.
Use separate connections with another user for writing.

Stupid Programmer Errors

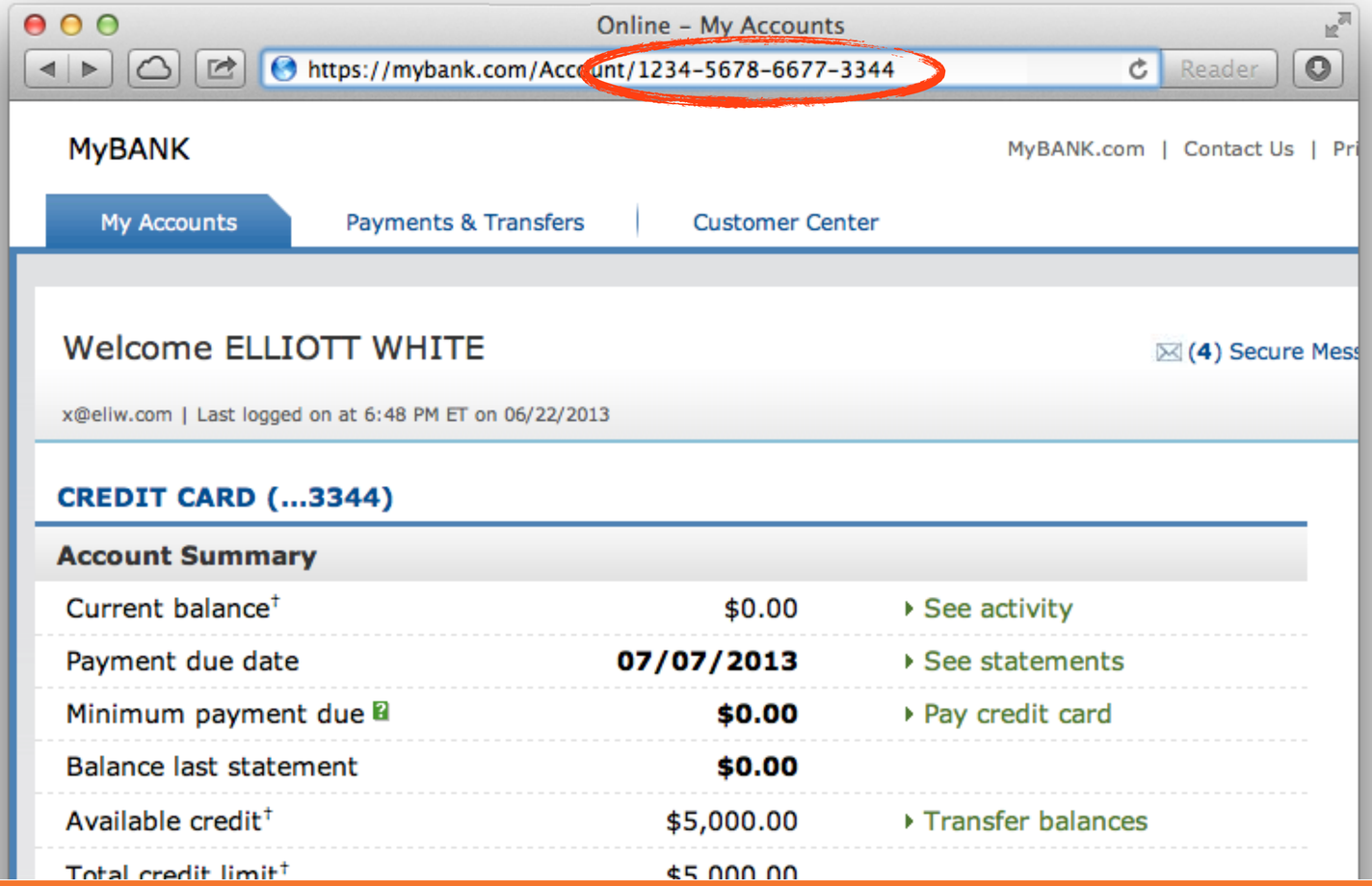
Let's clear the air on these …

Unchecked Permissions

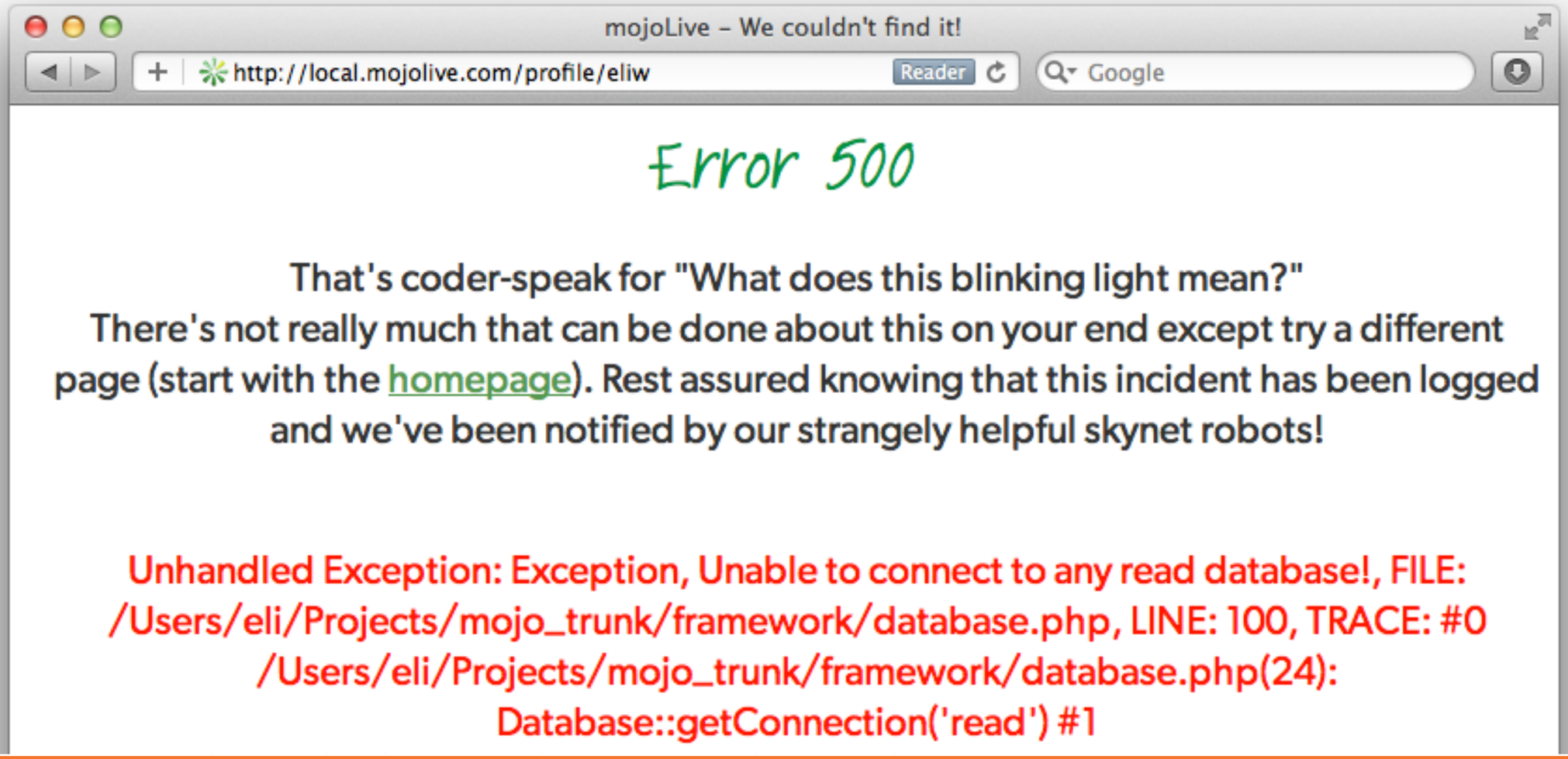


Unchecked Permissions

Ability to URL-hack to access unauthorized data.



Information leaks

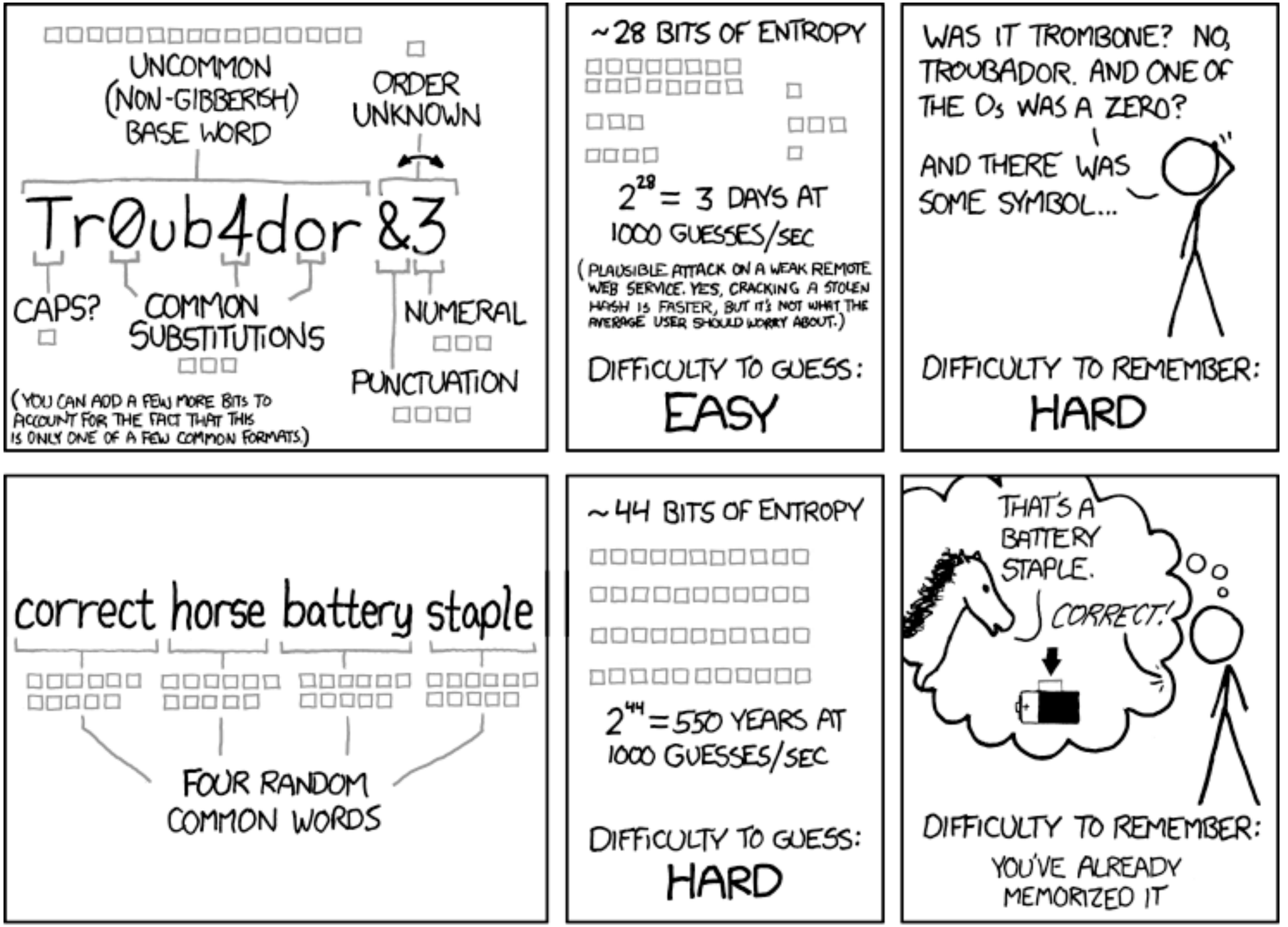


Password Protection

Best Password Practices

Rules for Passwords:

- Don't restrict people from using letters, numbers, special characters or spaces
- OK to have a minimum length but not max
- Requiring mixed symbols can help, but makes hard to remember


UNCOMMON (NON-GIBBERISH) BASE WORD
ORDER UNKNOWN
Tr0ub4dor&3
CAPS?
COMMON SUBSTITUTIONS
NUMERAL
PUNCTUATION
(YOU CAN ADD A FEW MORE BITS TO ACCOUNT FOR THE FACT THAT THIS IS ONLY ONE OF A FEW COMMON FORMATS.)
~28 BITS OF ENTROPY
2^28 = 3 DAYS AT 1000 GUESSES/SEC
(PLAUSIBLE ATTACK ON A WEAK REMOTE WEB SERVICE. YES, CRACKING A STOLEN HASH IS FASTER, BUT IT'S NOT WHAT THE AVERAGE USER SHOULD WORRY ABOUT.)
DIFFICULTY TO GUESS:
EASY
WAS IT TROMBONE? NO, TROUBADOR. AND ONE OF THE Os WAS A ZERO?
AND THERE WAS SOME SYMBOL...
DIFFICULTY TO REMEMBER:
HARD
correct horse battery staple
FOUR RANDOM COMMON WORDS
~44 BITS OF ENTROPY
2^44 = 550 YEARS AT 1000 GUESSES/SEC
DIFFICULTY TO GUESS:
HARD
THAT'S A BATTERY STAPLE.
CORRECT!
DIFFICULTY TO REMEMBER:
YOU'VE ALREADY MEMORIZED IT

Password Hashing

Example!

Do not store plain text passwords

Always 1-way hash

Do not just use MD5!
Highly vulnerable to rainbow tables

Don't even use SHA1
The longer your hashing takes to run,
the longer it takes for someone to crack it!

Password Hashing (Manually)

PHP <5.5

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

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

Find a full list of supported algorithms via:

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

Use a more secure algorithm, such as sha512:

```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

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.
}
```

Secondary Measures

Typically used to thwart phishing attempts

- Showing a known photo on login
- Asking for date of birth
- Asking for first place of residence
- etc…

All have mixed effectiveness

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/

FIEO

Filter Input Escape Output

FIEO

#1 Rule of Web Security!

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 to use

Filtering Input

Filtering is not directly a security measure, **but**:

- Simplifies security later
- Provides security in depth
- Makes for cleaner data

More layers of security mean less chance of exploit

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 '301-555-1234' 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 each option

Escaping Output

Escaping is actual protection

Making the output **safe** to be used

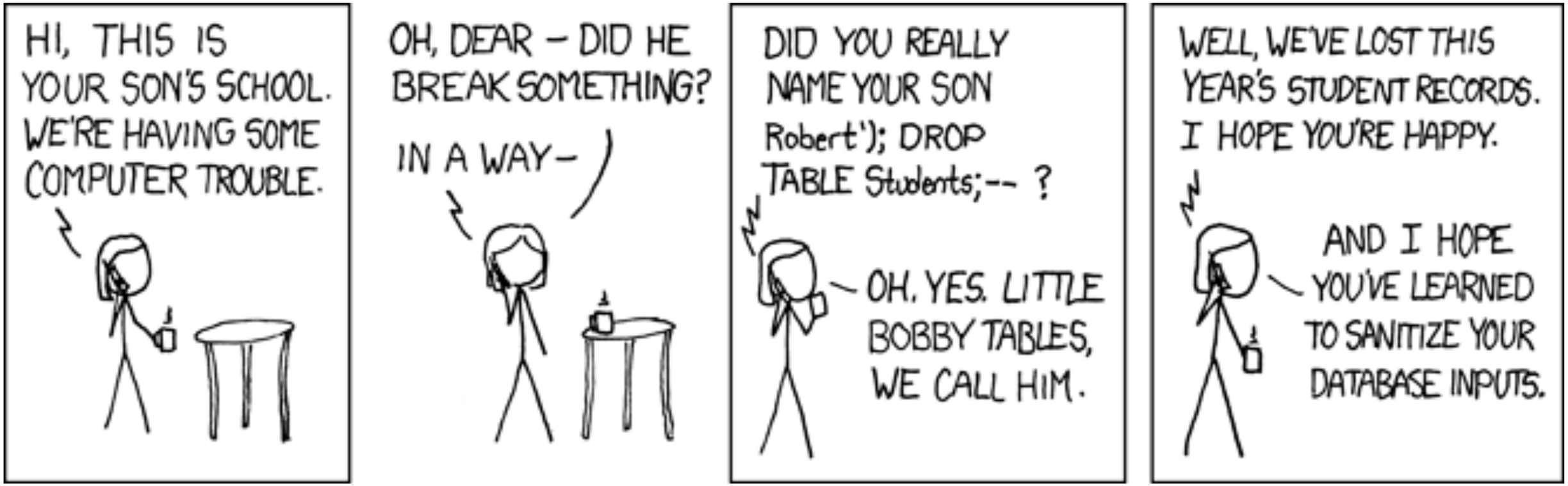
Must be done appropriate to context

Every type of output needs escaped **differently**

Various Attack Vectors

Now moving on to true ‘attacks’ …

SQL Injection



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

Example!

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

The Solution:

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

or:

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

Other Injection

Example!

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

Unchecked File Uploads:
The user being allowed to upload an executable file.

Code Injection:
User being able to directly inject code. (DON'T USE EVAL!)

Session Hijacking

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

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

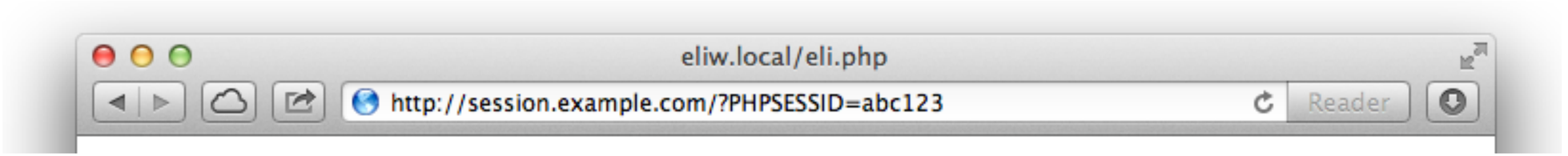
Always regenerate Session IDs on a change of access level.

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

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);
}
```

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)
FIEO (Filter Input, Escape Output)

Don't forget about rewritten URL strings!

XSS - Reflected XSS

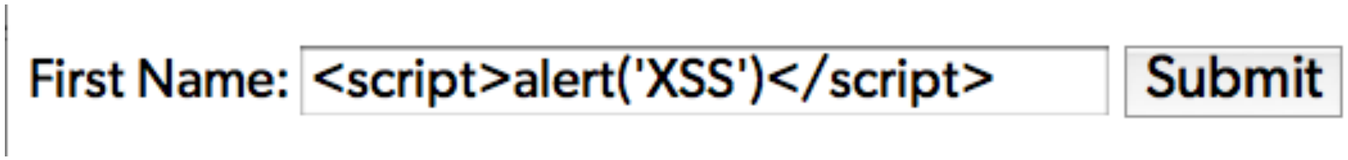
Reflected XSS

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

Example!

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?

1) Users can be directed to your website via links.

2) Also, users can be talked into anything…

XSS - Stored XSS

Stored XSS

You store the data, then later display it

The Security Hole:

```
<?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

Example!

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!

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>
```

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>
```

Example!

But you have to deal with attr vs HTML vs CSS etc
So use this: https://github.com/chrisisbeef/jquery-encoder/

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

```
<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
function generateToken() {
    $token = empty($_SESSION['token']) ? false : $_SESSION['token'];
    $expires = empty($_SESSION['tExpires']) ? false : $_SESSION['tExpires'];
    if (!$token || ($expires < time())) {
        $token = md5(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)

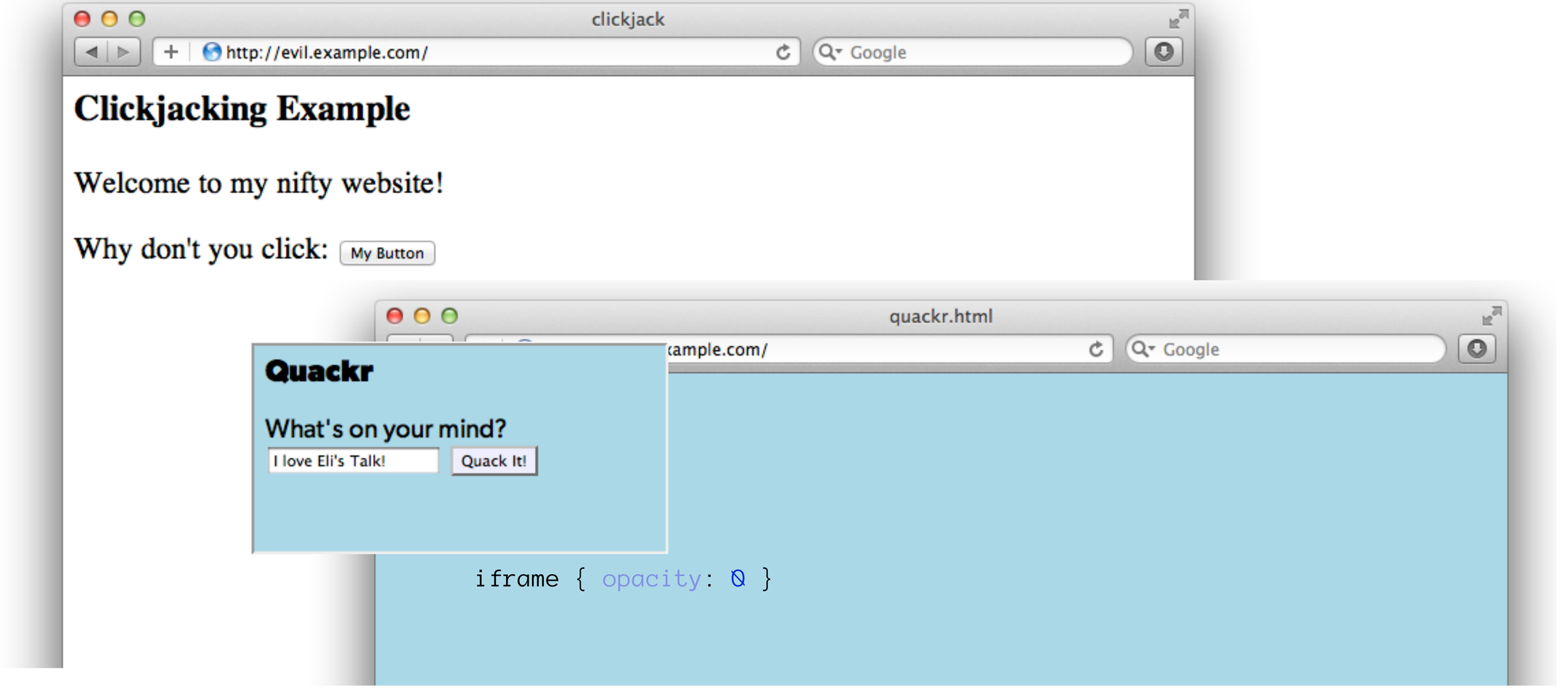
Protect via CSRF token

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;
}
?>
```

Clickjacking



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!

Clickjacking - Solution 2

The Solution:

Ensure you aren't displayed in iFrame

```
<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>
```

Brute Force Attacks (Password)

Really only two primary defenses:

CAPTCHA

IP rate limiting

Brute Force Attacks (CAPTCHA)

reCAPTCHA is free and easy to use

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>
```

Brute Force Attacks (CAPTCHA)

On the Server:

```php
<?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!
}
?>
```

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

Brute Force Attacks (Rate Limit)

The Solution:

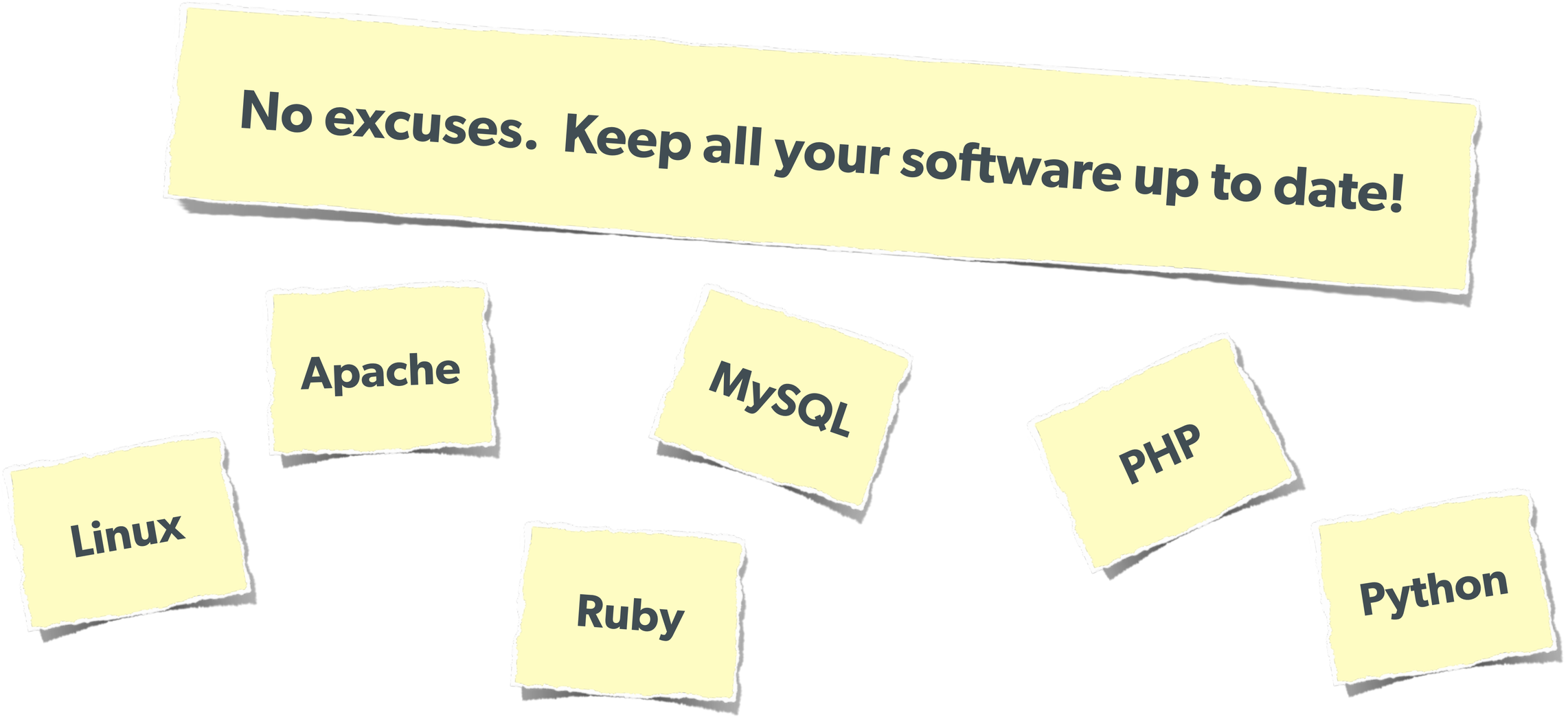
Only allow so many fails per IP

```
$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);
}
```

Server Level Security

Now moving on to true ‘attacks’ …

Keep Your Stack Patched



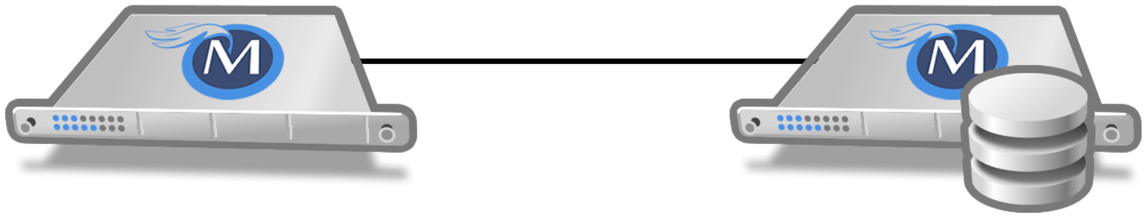
DDOS & Similar Attacks

Good luck!

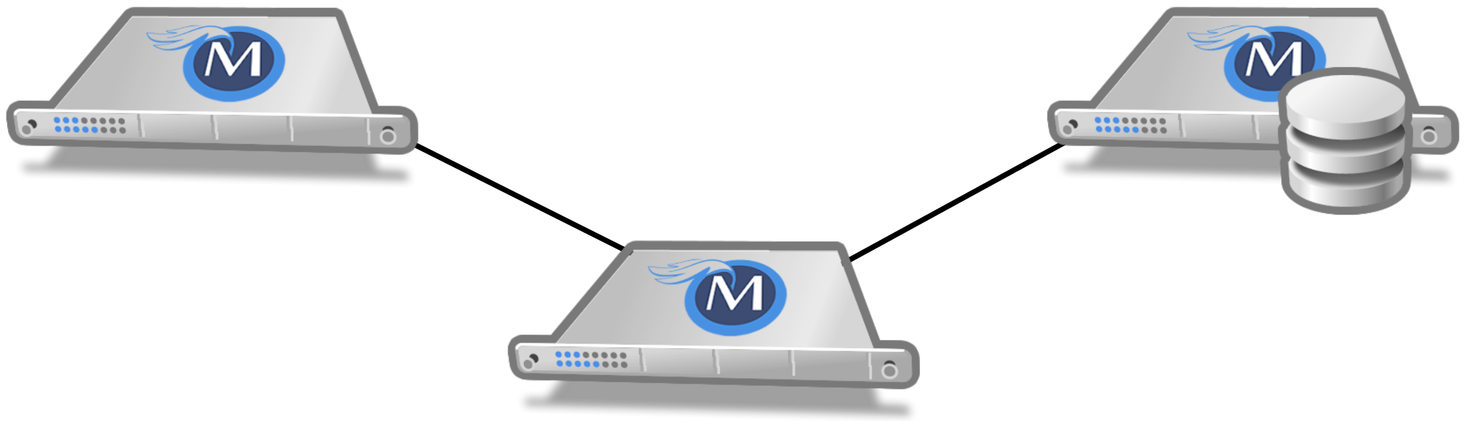
Rely on firewall features of your machines & hosting.

Hire a good ops team

Man in the Middle



Man in the Middle



The Solution: Use SSL

Tips for Recovery

Wait, you just got a 2am phone call?

Logging

You can't react, if you don't know what happened!

Log everything you can:

- Failed SQL queries
- Detected hijack attempts
- Code (PHP) errors
- Failed server connections

Plans of Action

Questions?

For this presentation & more:
http://eliw.com/

Twitter: @EliW

php[architect]: https://www.phparch.com/
musketeers: http://musketeers.me/

php[architect]