Check Point
SOFTWARE TECHNOLOGIES LTD.

They Hate Us 'Cause They Ain't Us

How We Broke the Internet

Netanel Rubin

Secure Coding

- **Code development practice**
- Mitigates **basic** vulnerabilities
  - *XSS*
  - *SQL Injection*
- Made by security experts, for **non**-security experts
  - Mainly developers

- **In practice, commonly used as the only measure of security in the development cycle**

How Do I Secure Coding??

- There are “Secure Coding” courses
  - **A lot** of courses
- Developers *usually* choose the same course as their peers
  - Resulting in the same course being the only one taught in the same company
- Most developers *usually* pass only 1 or 2 of these courses in their **entire career**

Secure Coding Problems

- Most developers in a company pass the same course
  - Same course – **Same mistakes**
- Secure Coding focuses on input sanitization vulnerabilities
  - Neglecting false assumptions and logical vulnerabilities
- Secure Coding provides a **misguided** sense of security
  - Resulting in less CR, both internally and externally

Secure Coding Problems

- Secure coding gives you the feeling you are secure
- Without being secure

- But I guess I need to prove that…

How To Prove A Point 101

- How can I **prove** Secure Coding is a fallacy?
- **Using 3 things**
  - **0-days**
  - **0-days**
  - ~~**Top-Secret exploits**~~
  - **0-days**

Case Study 1 - MediaWiki

- The *most* popular Wiki platform
- Open Source – PHP
- Runs on Wikipedia.org
  - And 25,000 more sites

Type of Check	Implemented?
User Input Sanitization	✗
Dangerous Functions	
Language Quirks	
False Assumptions	

MediaWiki – The integer in the box

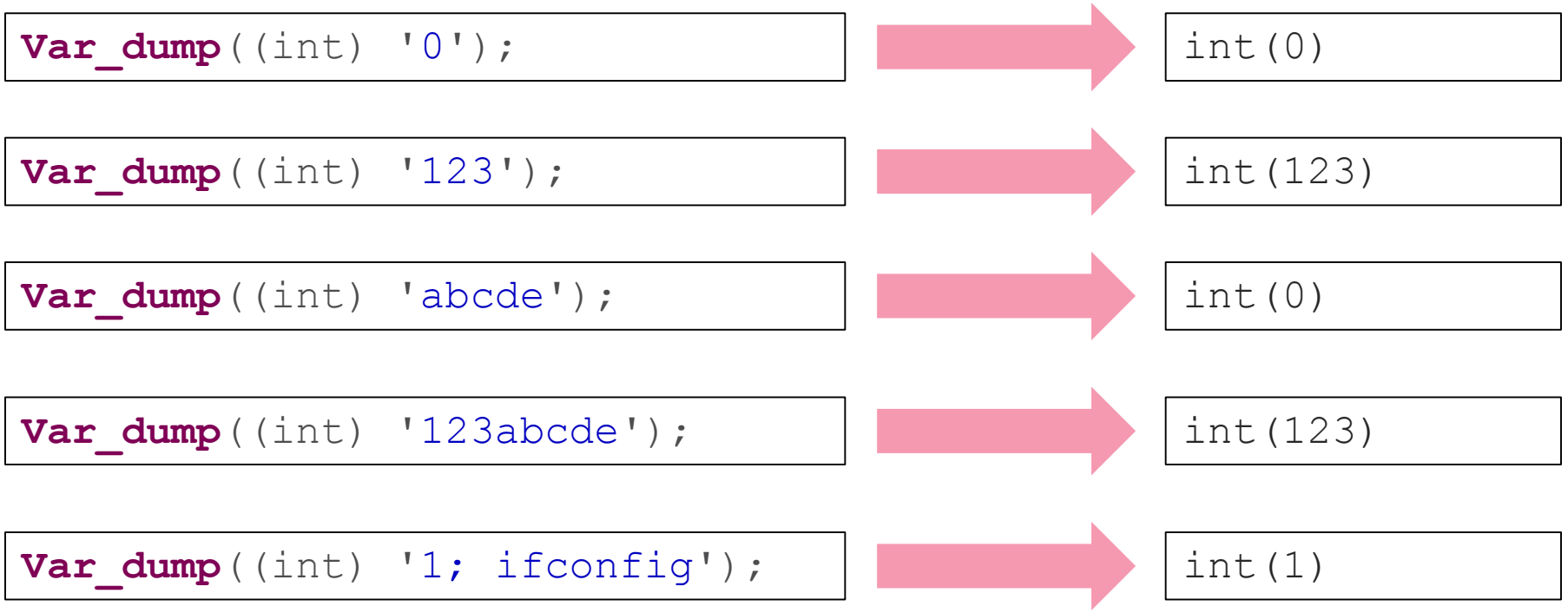
- MediaWiki relies on external binaries
  - For converting images
  - Analyzing documents
  - ... and more

```
// Make sure the page parameter is a valid number
if ($params['page'] > $image->pageCount())
        $params['page'] = $image->pageCount();
else if( $params['page'] < 1 )
        $params['page'] = 1;

exec('/usr/bin/convert' . // Execute the convert command
        ...
        $params['page']
        ...
);
```

MediaWiki – The integer in the box

- What happens when *'page'* is a string?



Case Study 1 - MediaWiki

- CVE-2014-1610
  - Unauthenticated RCE on MediaWiki

Case Study 2 - vBulletin

- The *most* popular Forum platform
- Commercial – PHP
- Runs on ubuntuforums.org
  - And 32,000 more sites

Type of Check	Implemented?
User Input Sanitization	✓
Dangerous Functions	✗
Language Quirks	
False Assumptions	

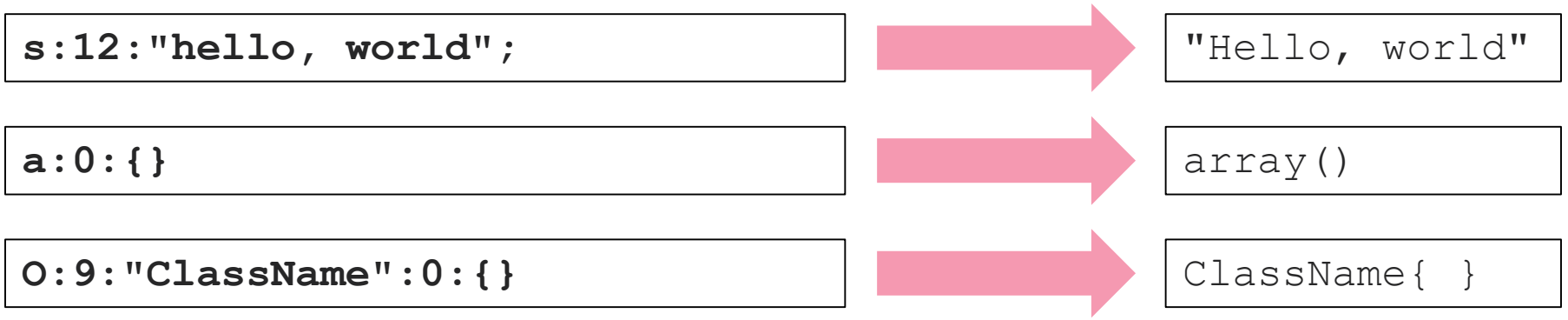
vBulletin – I Didn’t See-rialize That Coming

- vBulletin developers are aware of dangerous functions
  - eval
  - exec
  - popen

unserialize??

vBulletin – I Didn't See-rialize That Coming

- Unserialize Creates a PHP value from a stored representation



- When an object is unserialized, several "magic" methods are automatically called
  - __wakeup() – Right after the unserialize operation
  - __destroy() – When the object is destroyed
  - __toString() – When the object is converted to a string

vBulletin – I Didn't See-rialize That Coming

- Using these "magic" methods, we can expand our attack surface
- As a bonus, because we control the object, we control its properties as well

- **So down the rabbit hole we go!**

vBulletin – I Didn't See-rialize That Coming

- We create an "vB_vURL" object with the following "__destruct" method:

```
function __destruct()
{
 if (file_exists($this->tmpfile))
    {
     @unlink($this->tmpfile);
    }
}
```

vB_vURL::__destruct()

- Because the "tmpfile" property is used in an "unlink()" call
  - **It is considered as a string**

vBulletin – I Didn't See-rialize That Coming

- If "tmpfile" was an object, "__toString()" would have been called
- So we use a "vB_View" object as our "tmpfile" property
  - Which executes this code:

```
public function __toString()
{
        return $this->render();
}

vB_View::__toString()
```

vBulletin – I Didn't See-rialize That Coming

- From "render()" we jump through several functions
- Until we finally reach this "render()" code:

```
public function render() {
    …
    $templateCode=$templateCache->getTemplate($this->template);
    …
    @eval($templateCode);
}
```

Guess who controls $templateCode?

Case Study 2 - vBulletin

- CVE-2015-7808
  - Unauthenticated RCE on vBulletin

Case Study 3 - Bugzilla

- The *most* popular Bug Tracker
- Open Source – Perl
- Runs on bugzilla.mozilla.org
  - And 130 more major projects

Type of Check	Implemented?
User Input Sanitization	✓
Dangerous Functions	✓
Language Quirks	✗
False Assumptions	

Bugzilla – Exploiting, The Quirking Way

- Perl features an expression called “lists”
  - **Well known and documented**

```
@array = (1, 2, 'a', 'b', 'c');
%dict = (1 => 2, 'a' => 'b');
```

Bugzilla – Exploiting, The Quirking Way

- A list inside a dictionary can create new dictionary pairs
  - **Known for Perl Pros, partially documented**

```
@list = ('f', 'lol', 'wat');
$hash = {'a' => 'b',
         'c' => 'd',
         'e' => @list
};       'lol' => 'wat'
};
```

Bugzilla – Exploiting, The Quirking Way

- A list can be created from the user input
  - **Barely known and NOT documented**

```
index.cgi?foo=1&bar=a
index.cgi?foo=1&foo=2&bar=a&bar=b
```

Bugzilla – Exploiting, The Quirking Way

- Some privileges are given via an email regex
  - Example: *@mozilla.org can view confidential Firefox bugs
- when a new user registers, his email address is validated using a token sent to the mailbox

Bugzilla – Exploiting, The Quirking Way

- After the validation takes place, this code happens:

```
my $otheruser = Bugzilla::User->create({
        login_name => $login_name,
        realname   => $cgi->param('realname'),
        cryptpassword => $password});
```

$login_name => The email address validated (extracted from the DB)

$password => The user defined password (as a scalar)

$cgi->param('realname') => **Bingo!**

Bugzilla – Exploiting, The Quirking Way

```
a=confirm_new_account&t=[REGISTRATION _TOKEN]&passwd1=Password1!&passwd2=Password1!
&realname=Lolzor&realname=login_name&realname=admin@bugzilla.com
```

```
my $otheruser = Bugzilla::User->create({
        login_name => $login_name,
        realname   => 'Lolzor',
        login_name => 'admin@bugzilla.com'
        cryptpassword => $password});
```

- We control the email address
  - **We control the privileges group we join**

Case Study 3 - Bugzilla

- CVE-2014-1572
  - Authentication mechanism bypass on Bugzilla

- For more information about this vuln, watch my 31C3 talk "The Perl Jam"

Case Study 4 - Magento

- The *most* popular eCommerce platform
- Open Source – PHP
- Runs on (part of) ebay.com
  - And 270,000 more sites

Type of Check	Implemented?
User Input Sanitization	✓
Dangerous Functions	✓
Language Quirks	✓
False Assumptions	✗

Magento – Keep Going Forwarded

- Magento developers made sure regular users can't access the admin panel
  - They checked the user session
  - They checked its privileges
  - They checked that the user is not disabled

Magento – Keep Going Forwarded

- But how does Magento does all that?

```
$requestedActionName = $request->getActionName();
if ($user) { // A user exist
    $user->reload(); // Check validity of user
}
if (! $user || ! $user->getId()) { // No user
    if ($request->getPost('login')) { // login
        TRY_TO_LOGIN
    }
    if (! $request->getParam('forwarded')) {
        $request->setController('login');
    }
}
```

/index.php/admin/?forwarded=1

Magento – Keep Going Forwarded

- Magento developers used the "forwarded" parameter as an internal redirect mechanism
  - Used to allow components to create **their own authentication mechanisms**
- Unfortunately, they forgot this parameter can also be controlled by the user
  - Using HTTP Parameters
- **This effectively allows anyone to access the admin panel**

Case Study 4 - Magento

- CVE-2015-1397, CVE-2015-1398, CVE-2015-1399
  - Authentication mechanism bypass
  - SQLI
  - 2 LFIs
  - RFI

Case Study 5 - WordPress

- The *most* popular CMS/Blogging platform
- Open Source – PHP

- **WordPress is massively deployed**
- It handles **126M users** a month!

Type of Check	Implemented?
User Input Sanitization	✓
Dangerous Functions	✓
Language Quirks	✓
False Assumptions	✓

WordPress - How WordPress Works

- Any user can access the admin panel
  - But using a capabilities system, not every admin page

	Subscriber	Administrator
read_page	✓	✓
read_post	✓	✓
edit_posts	✗	✓
install_themes	✗	✓
edit_plugins	✗	✓

WordPress - Exploiting The Un-Exploitable

- We assume we are subscribers at the site
  - The lowest role possible
  - We can only read public posts and pages
    - *Can't even comment*

- We need more capabilities!

WordPress - Exploiting The Un-Exploitable

- How does WordPress check our capabilities?

```
if(current_user_can('edit_posts')) // Can we edit posts?
```

```
if(current_user_can('edit_post', 1)) // Can we edit post ID 1?
```

- Each role has specific permissions
- *'current_user_can()'* maps a requested capability into the appropriate role permission
  - And returns true/false based on our permissions

- **But how?**

WordPress - Exploiting The Un-Exploitable

- *'current_user_can()'* is a giant SWITCH statement
- Let's look on the *"edit_post"* capability check
  - *Responsible for checking if the user can edit a specific post*

```
case 'edit_post': // Edit Post/Page
case 'edit_page':
        $post = get_post( $args[0] ); // Get the post

        // If the post doesn't exist, no capabilities needed
        if ( empty( $post ) )
                break;
```

- If the post ID doesn't exist => no permissions needed!

WordPress - Exploiting The Un-Exploitable

- We can access code that checks capabilities for a post ID, but doesn't check it exists
- But we want to be able to edit a post that **does exist**!
- How can we do that?

WordPress - The Need For Speed

- Using the capabilities bug, we could access the post editing code

```
function edit_post( $post_data = null ) {
    if ( empty($post_data) )
        $post_data = &$_POST;

    $post_ID = (int) $post_data['post_ID']; // Get the post ID
    $post = get_post( $post_ID ); // Get the post
    …
    $success = wp_update_post( $post_data ); // Update the post
in the DB
}
```

WordPress - The Need For Speed

- But before the DB update occurs, a post ID validation check takes place

```
function wp_update_post($postarr = array(), $wp_error = false){
    // First, get all of the original fields.
    $post = get_post($postarr['ID'], ARRAY_A);

    if ( is_null( $post ) ) {
        if ( $wp_error )
            return new WP_Error('invalid_post', 'Invalid post');
        return 0;
    }
    ...
}
```

WordPress - The Need For Speed

- **We're stuck :(**

- We need an **INVALID** post ID for *'edit_post()'*
- But a **VALID** post ID for *'wp_update_post()'*

- Wait…
- **What if we could create the post between these function calls?**

WordPress - The Need For Speed

- WordPress doesn't allow subscribers to **create a post**

- In fact, when we try to do so it **blocks** our access by calling *'wp_dashboard_quick_press()'*:

```
switch($action) {
case 'post-quickdraft-save':
    if ( ! current_user_can( 'edit_posts' ) )
        $error_msg = "You don't have access to add new posts.";

    // If there's an error (no token, no capabilities)
    if ( $error_msg )
        return wp_dashboard_quick_press( $error_msg );
```

WordPress - The Need For Speed

- But what does *'wp_dashboard_quick_press()'* do?

- It creates a post.

```
function wp_dashboard_quick_press( $error_msg = false ) {
        ...
        $post = get_default_post_to_edit( 'post' , true);
        ...
}
```

WordPress - The Need For Speed

- Now we can create a post
  - **But how do we create it exactly at the right time?**

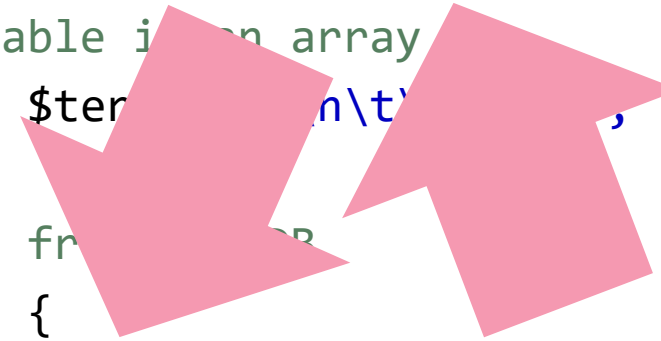
- **We will delay the script**
  - By executing a lot of DB queries

- **But again, how?**

WordPress - The Need For Speed

```
foreach ((array) $post_data['tax_input'] as $taxonomy => $terms) {
    // Make sure the terms variable i      n array
    $terms = explode(',', trim( $ter          n\t\          ,   ) );

    // Fetch the required terms fr          R
    foreach ( $terms as $term ) {
        $_term = get_terms( $term ) );
    }
}
```



- **We control the taxonomy array**
  - Each element is inserted into *'get_terms()'*
    - *'get_terms()' executes an **SQL SELECT query***
- We control the array => we control the number of elements
  - => **We control the number of SELECT queries**

WordPress - The Need For Speed

- Using the race condition, we were able to **edit a real post**

1. We send an "edit post" request with invalid post ID
   - containing our large taxonomy array

2. While the script executes, we send a "create post" request, which creates that post

3. When the taxonomy queries are done, the post already exists in the DB
   - **Allowing us to update it as we wish**

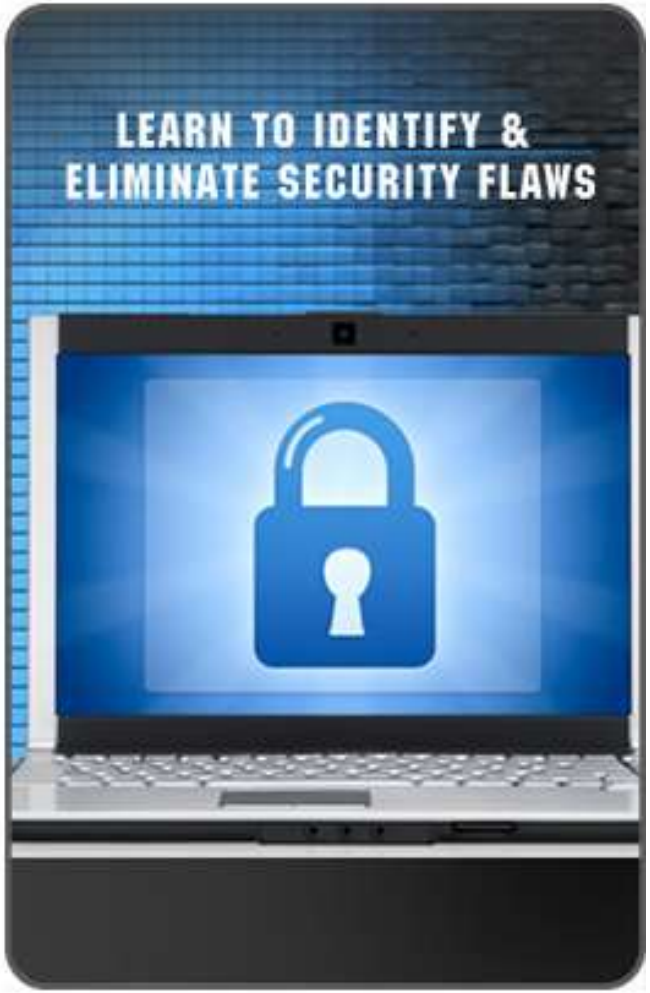
WordPress - PE'ing Like It's 1999

- Editing a post doesn't compromise anything
  - But this Privilege Escalation granted us **access to more code**
- More code => More attack surface
- More attack surface => **More vulnerabilities**

- Using that attack surface, we discovered a:
  - **Persistent XSS** on the front page of the site
  - **SQL Injection**, allowing us to compromise the DB

- **Basically, total WordPress PWNGE**

Case Study 5 - WordPress



- CVE-2015-5623
  - Privilege Escalation
- CVE-2015-2213
  - SQL Injection
- CVE-2015-5714
  - Shortcode XSS
- CVE-2015-5715
  - Post Publish Privilege Escalation

Why Even Secure Coding??

- Secure coding **does not** guarantee secure code
- It provides *another* layer of security

- Developers **ARE NOT** hackers
  - Because they **don't** have the time to
  - Because they **don't** have the budget to
  - Because they *(sometimes)* **don't** have the skillset to

So What Should I Do?

- **HIRE HACKERS**
- **TO DO THE HACKING**
  - Penetration Testing
  - Code Reviews
  - Consulting

- **DO NOT rely on Secure Coding alone!**

Thanks!