

Web Application Security and Its Verification

Yih-Kuen Tsay

Dept. of Information Management
National Taiwan University

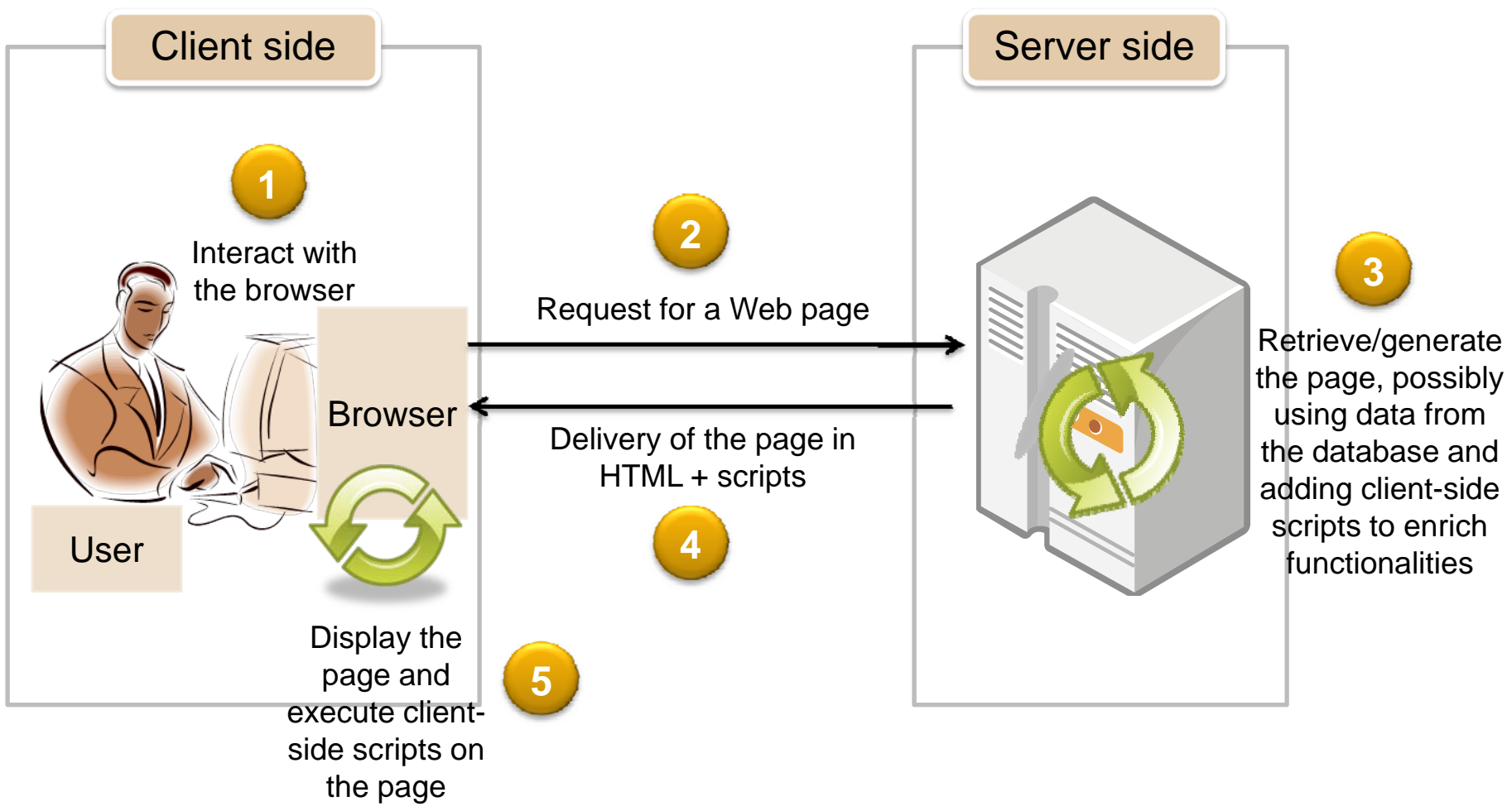
Caveats

- Concern only with security problems resulted from **program defects** (errors or bad practices)
- Will mostly assume using PHP, though there are many languages for programming the Web
- General interpretation of “Verification”
 - Testing and simulation
 - Formal verification
 - Static analysis
 - Model checking
 - Theorem proving
 - Manual code review

Outline

- Introduction
- Common Vulnerabilities and Defenses
- Objectives and Challenges
- Opportunities
- Conclusion
- References

How the Web Works



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 required by these activities, Web applications often involve user's **private and confidential data**.

Web Applications: Dynamic Contents

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

fixInput(); // invoke a user-defined function to sanitize 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>
<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>
```

Same-Origin Policy

- The policy permits scripts running on **pages originating from the same site** to access each other's DOM with no specific restrictions, but prevents access to DOM on different sites.
- The same-origin policy also applies to XMLHttpRequests unless the server provides an Access-Control-Allow-Origin (CORS) header.
- Notably, WebSockets are **not** subject to the same-origin policy.

Vulnerable Web Applications

- Web applications are supposed to be secure.
- Unfortunately, many of them do go wrong, having **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.

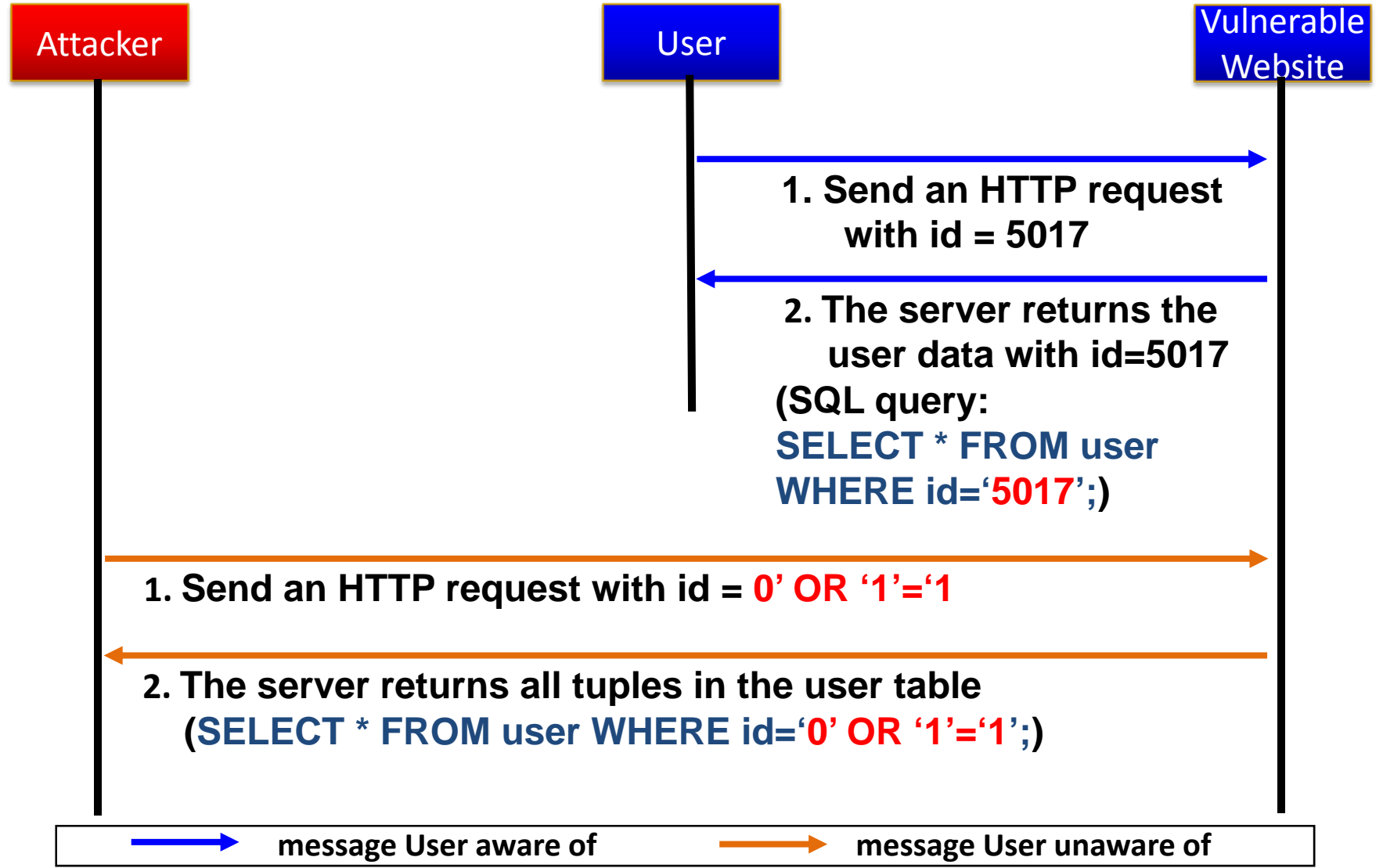
A Common Vulnerability: SQL Injection

- 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:

```
$sql = "SELECT * FROM users WHERE id = " . $_GET['id'] . """;
```
 - The attacker types in **a' OR 't' = 't** as the input for id.
 - The actual query executed:

```
SELECT * FROM users WHERE id = 'a' OR 't' = 't';
```
 - So, the attacker gets to see every row from the users table.

SQL Injection (cont.)



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.
- More than 70% of these are legitimate websites that have been hacked or compromised.

Source: Sophos security threat report 2011

Compromised Websites (cont.)

- 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
- Conclusion
- References

OWASP Top 10 Application Security Risks

- Injection
- Broken Authentication and Session Management
- Cross-Site Scripting (XSS)
- Insecure Direct Object Reference
- Security Misconfiguration
- Sensitive Data Exposure
- Missing Function Level Access Control
- Cross-Site Request Forgery (CSRF)
- Using Known Vulnerable Components
- Unvalidated Redirects and Forwards

What Changed from 2007 to 2010

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

What Changed from 2010 to 2013

OWASP Top 10 – 2010 (Previous)	OWASP Top 10 – 2013 (New)
A1 – Injection	A1 – Injection
A3 – Broken Authentication and Session Management	A2 – Broken Authentication and Session Management
A2 – Cross-Site Scripting (XSS)	A3 – Cross-Site Scripting (XSS)
A4 – Insecure Direct Object References	A4 – Insecure Direct Object References
A6 – Security Misconfiguration	A5 – Security Misconfiguration
A7 – Insecure Cryptographic Storage – Merged with A9 →	A6 – Sensitive Data Exposure
A8 – Failure to Restrict URL Access – Broadened into →	A7 – Missing Function Level Access Control
A5 – Cross-Site Request Forgery (CSRF)	A8 – Cross-Site Request Forgery (CSRF)
<buried in A6: Security Misconfiguration>	A9 – Using Known Vulnerable Components
A10 – Unvalidated Redirects and Forwards	A10 – Unvalidated Redirects and Forwards
A9 – Insufficient Transport Layer Protection	Merged with 2010-A7 into new 2013-A6

SQL Injection (cont.)

- Example (forget password):

Forgot Password

Email:

We will send your account information to your email address.

relevant code:

```
$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:

```
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
$str = "Is your name O'Brien?";
echo addslashes($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:

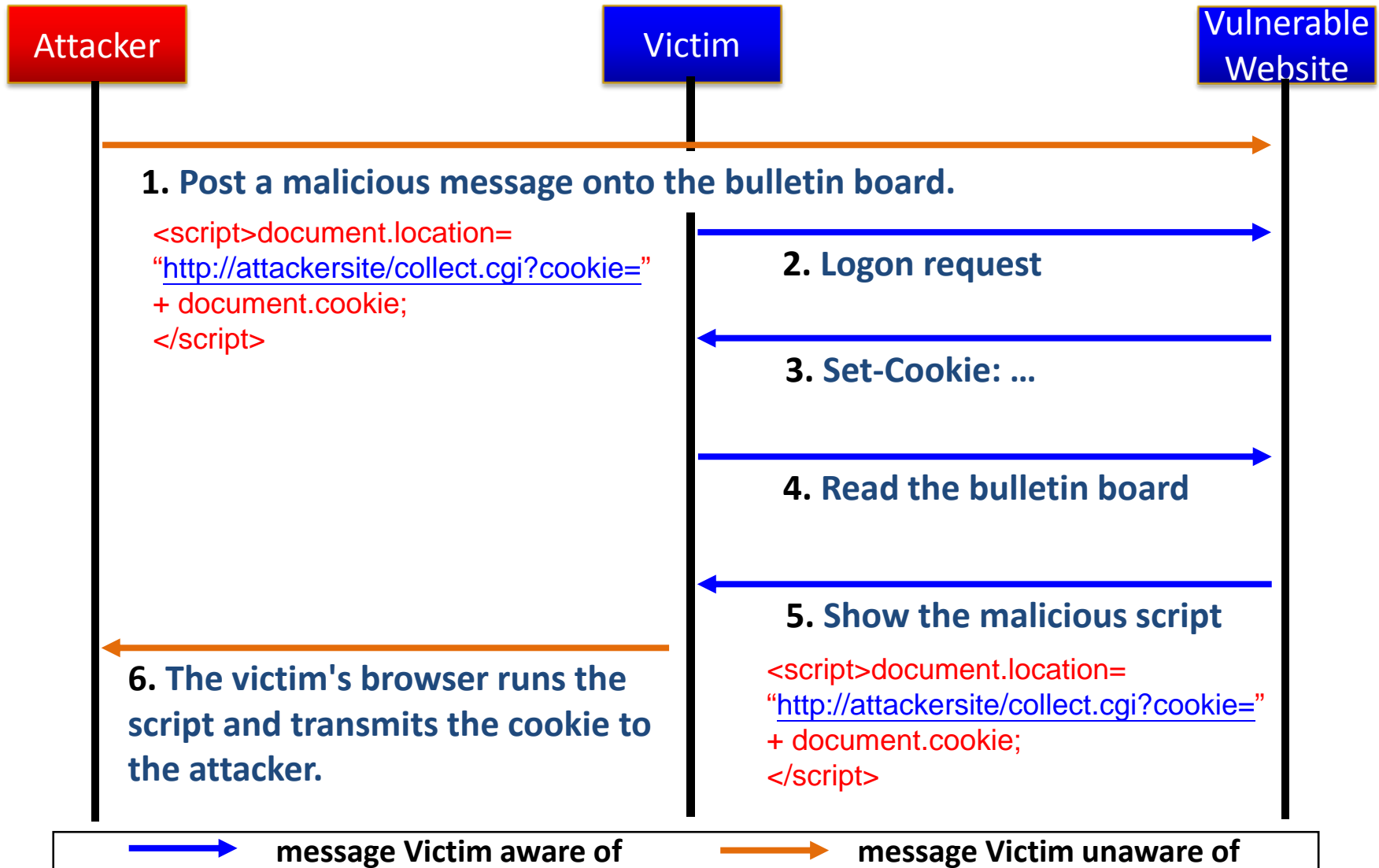
```
$db_connection = new mysqli("localhost", "user", "pass", "db");  
$statement = $db_connection->prepare("SELECT * FROM users WHERE id  
= ?");  
$statement->bind_param("i", $id);  
$statement->execute(); ...
```

- The ? is called a placeholder.
- To execute the above query, one needs to supply the actual value for ?.
- The first argument of `bind_param()` is the input's type: i for int, s for string, d for double

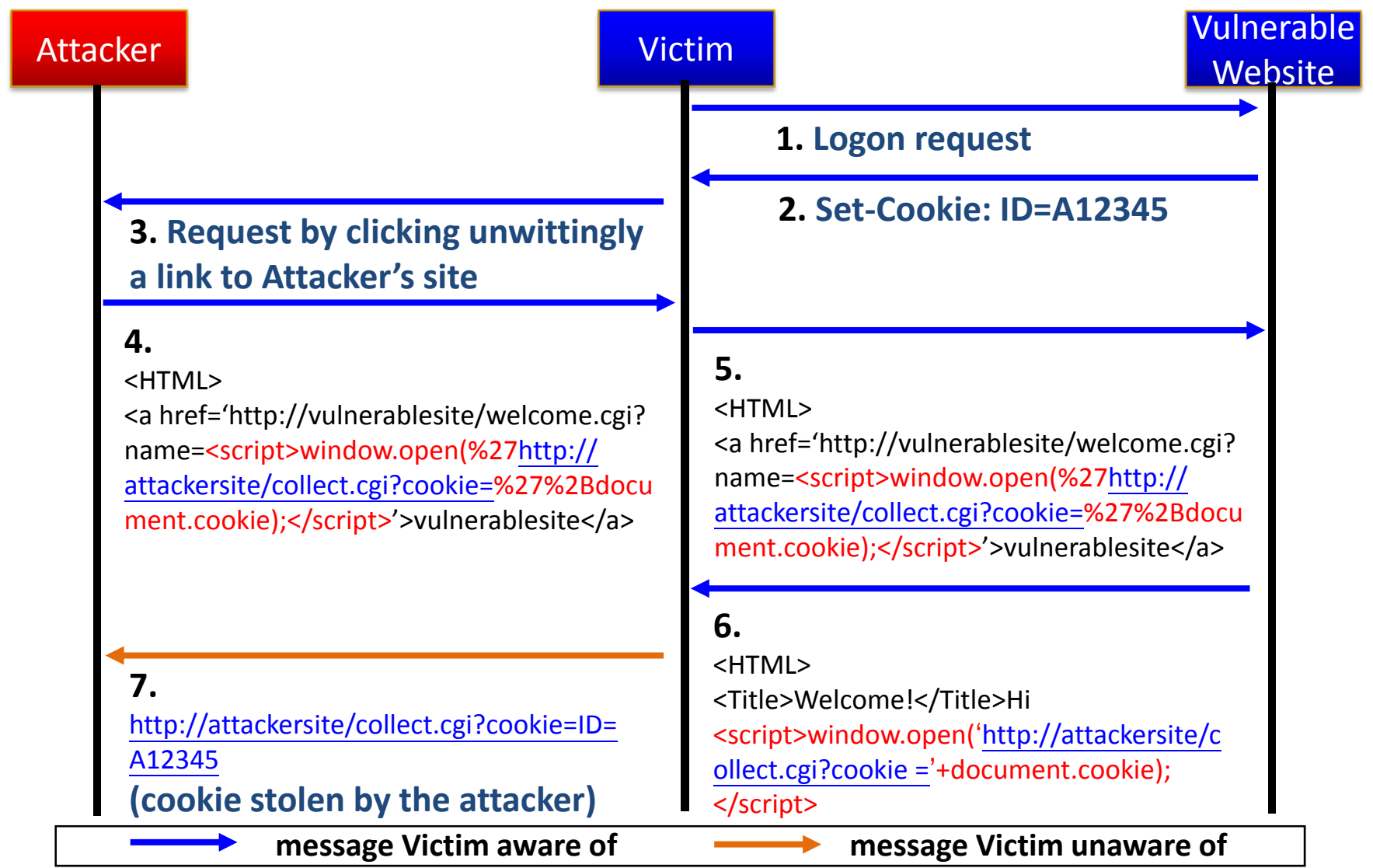
Cross-Site Scripting (XSS)

- 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



Reflected XSS



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()`
 - `htmlspecialchars()`

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 '&'
 - '"' (double quote) becomes '"'; when **ENT_NOQUOTES** is not set.
 - "'" (single quote) becomes '''; only when **ENT_QUOTES** is set.
 - '<' (less than) becomes '<'
 - '>' (greater than) becomes '>'

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

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

Outline

- Introduction
- Common Vulnerabilities and Defenses
- Objectives and Challenges
- Opportunities
- Conclusion
- References

Current Status

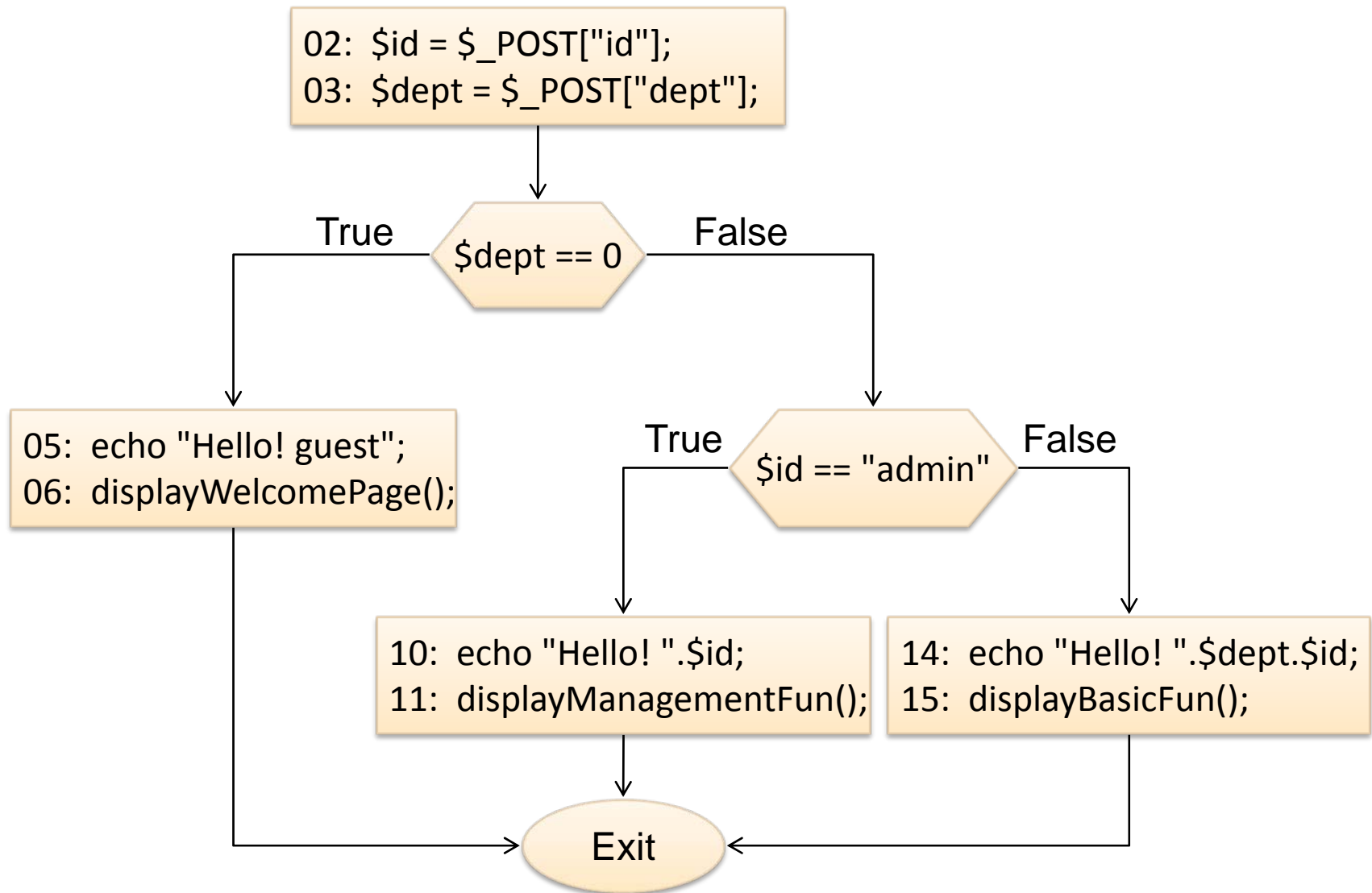
- Most known Web application security vulnerabilities can be fixed.
- There are code analysis tools that can help to detect such security vulnerabilities.
- So, what are the problems?

An Example

PHP code

```
01 <?php
02     $id = $_POST["id"];
03     $dept = $_POST["dept"];
04     if ($dept == 0) {           //guest
05         echo "Hello! guest";
06         displayWelcomePage();
07     }
08     else {                     // staff
09         if ($id == "admin") {
10             echo "Hello! ".$id;
11             displayManagementFun();
12         }
13         else {
14             echo "Hello! ".$dept.$id;
15             displayBasicFun();
16         }
17     }
18 ?>
```

Control Flow Graph



Dependency Graph (1/3)

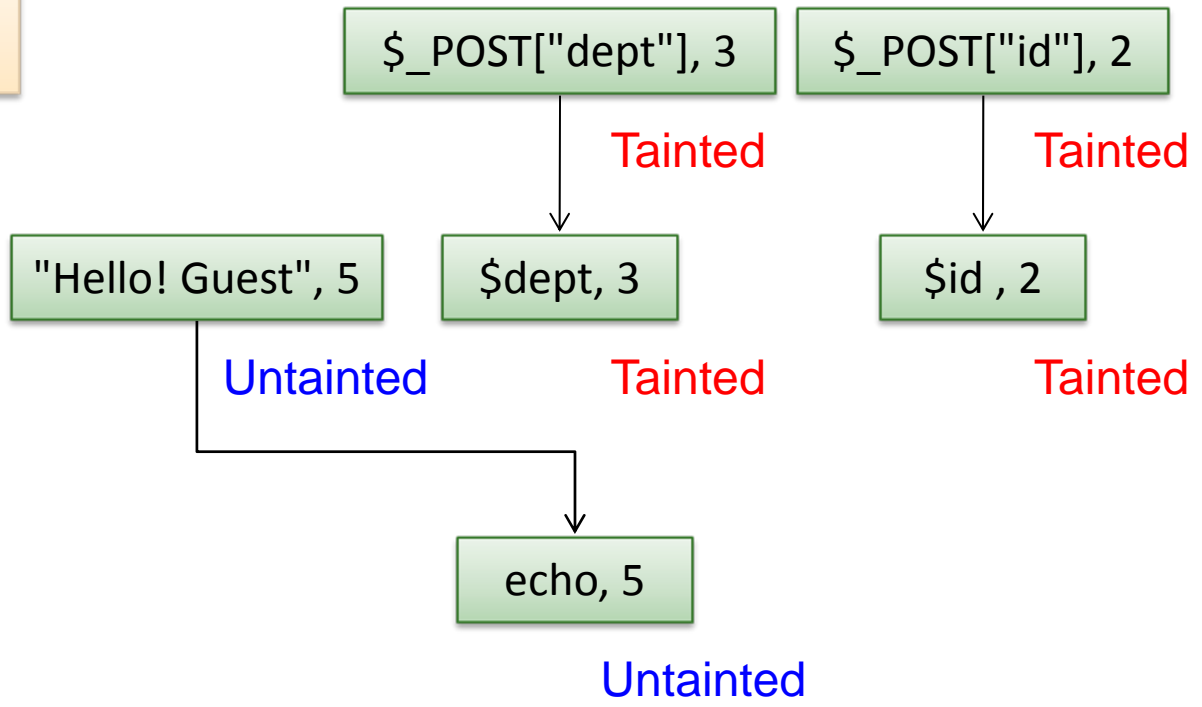
```
02: $id = $_POST["id"];
03: $dept = $_POST["dept"];
```

\$dept == 0

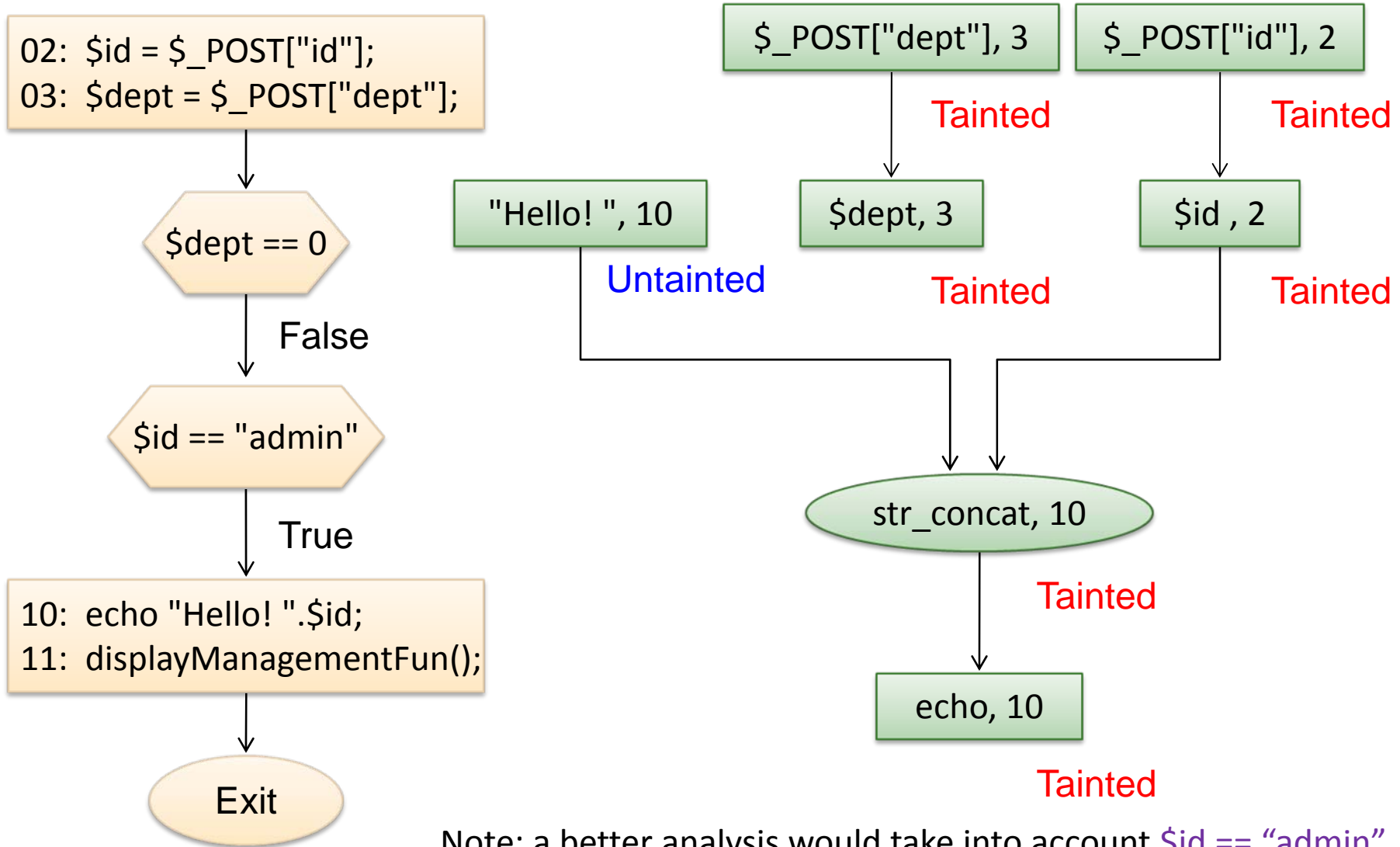
True

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

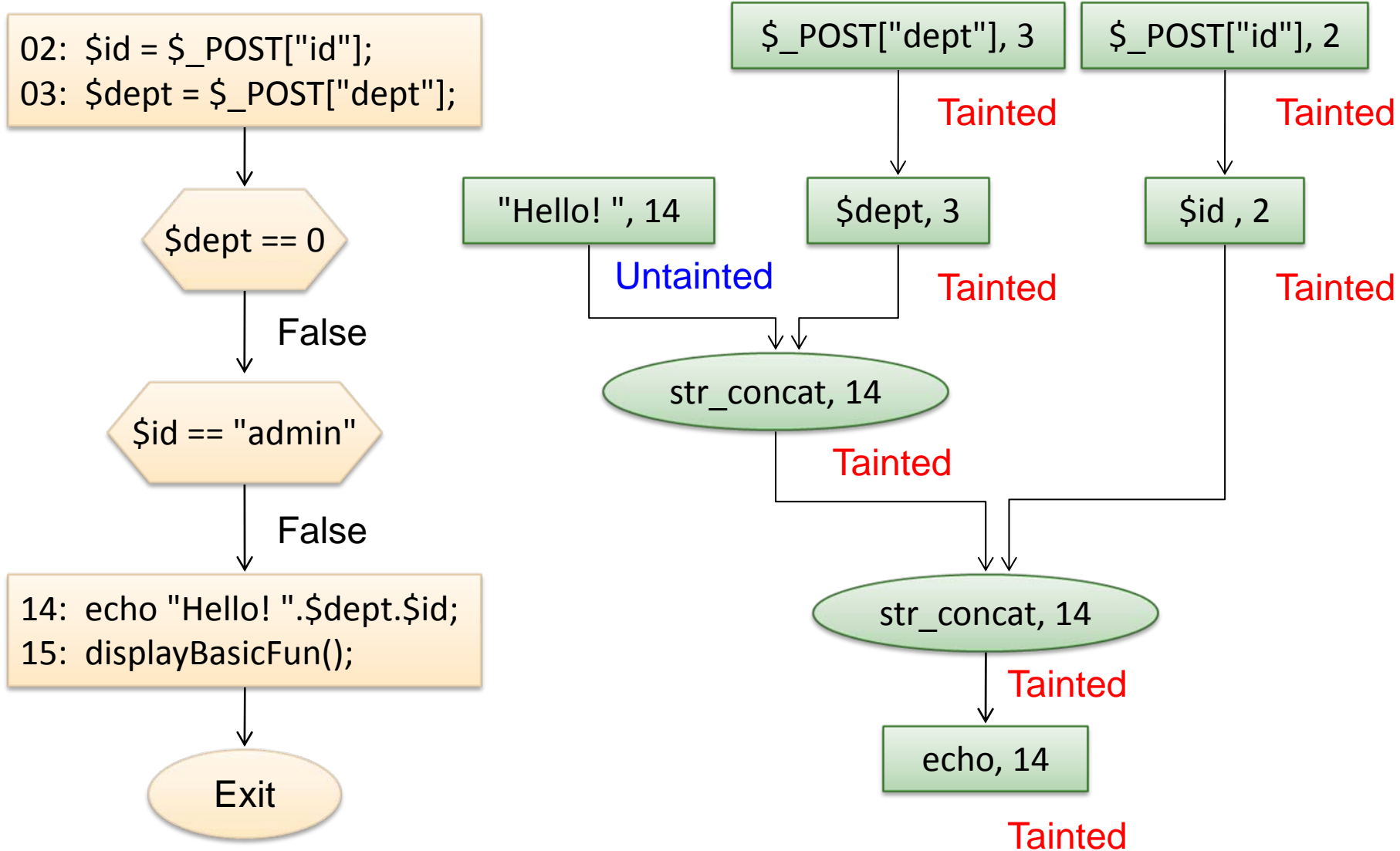
Exit



Dependency Graph (2/3)



Dependency Graph (3/3)

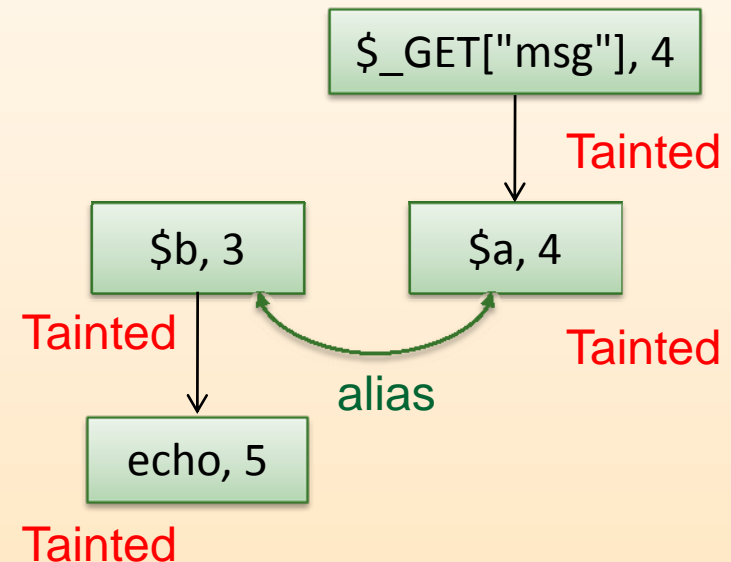


Alias

PHP code

```
01 <?php
02   $a = "message";
03   $b = &$a;
04   $a= $_GET["msg"];
05   echo $b;
06 ?>
```

Dependency Graph



Alias Information

must-alias{(a,b)}

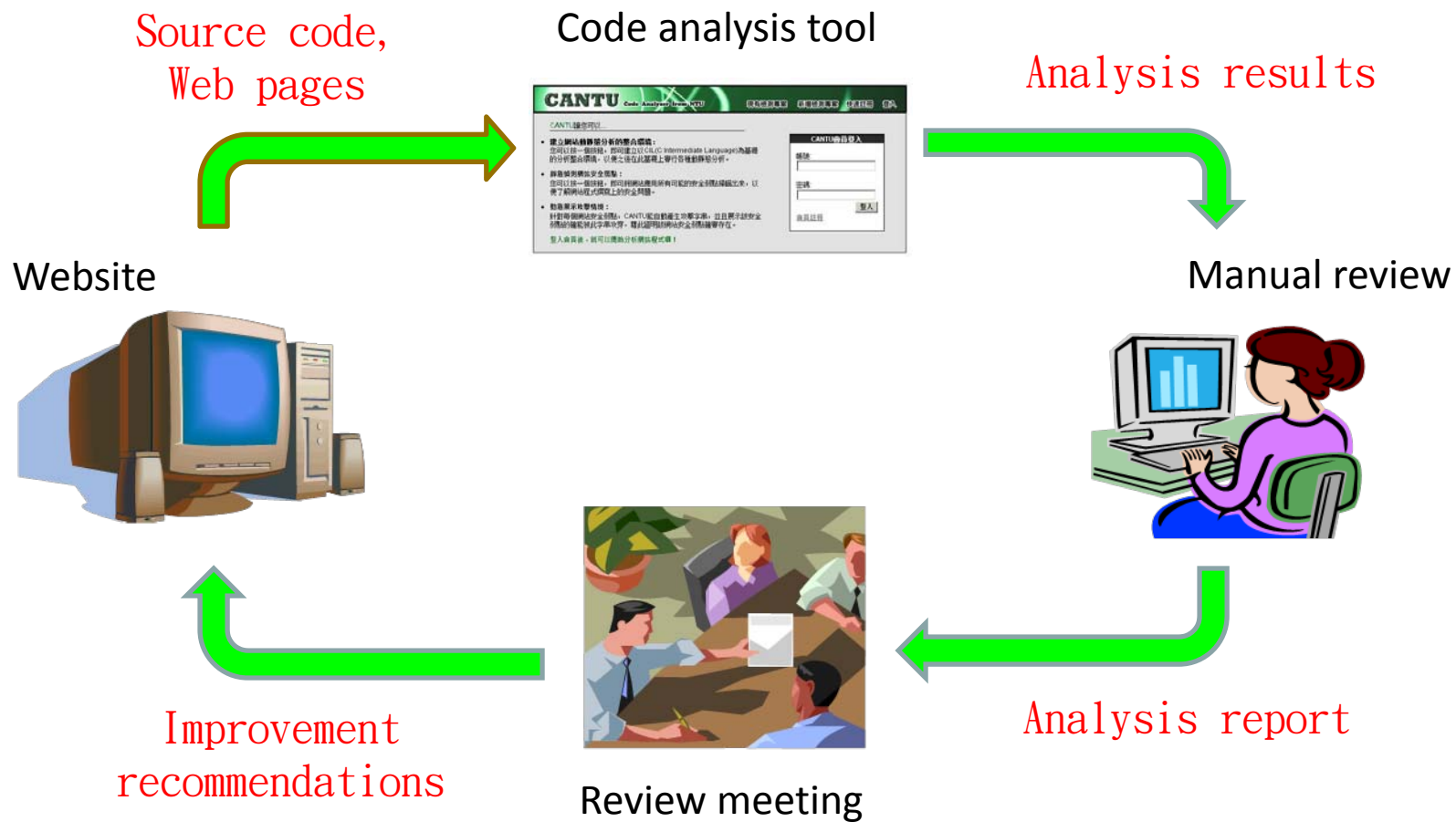
Detecting Vulnerabilities by Taint Analysis

- Build control and data flow graphs.
- 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

- Three 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 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



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 such as PHP:
 - 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

Challenges: Alias Analysis

- In PHP, aliases may be introduced by using the reference operator “&”.

PHP Code

```
<?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

PHP Code

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

- ❑ Tool A: false negative
- ❑ Tool B: false negative

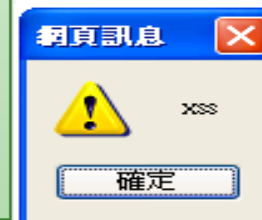
Note: Tool A and Tool B are two popular commercial code analysis tools.

Challenges: Alias Analysis (cont.)

- None of the existing tools (that we have tested) handles aliases between objects.

PHP Code

```
<?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>";
?>
```



Challenges: Strings and Numbers

```
1 if($_GET['mode'] == "add"){
2   if(!isset($_GET['msg']) || !isset($_GET['poster']))){
3     exit;
4   }
5   $my_msg = $_GET['msg'];
6   $my_poster = $_GET['poster'];
7   if (strlen($my_msg) > 100 && !ereg("script",$my_msg)){
8     echo "Thank you for posting the message $my_msg";
9   }
10 }
11 ...
```

- 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.

Challenges: A Theoretical Limitation

- 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
- Conclusion
- References

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.

Toward Formal Certification

- Current commercial code analysis tools are not precise enough and rely on competence of the programmer/reviewer.
- Ideally, every sensitive Web application should go through a thorough and **formal verification/certification** process.
- To be practical, one should probably focus on the **correctness of sanitization functions** (which are functions that validate user's input).
- There are quite a few issues that need further research.

Outline

- Introduction
- Common Vulnerabilities and Defenses
- Objectives and Challenges
- Opportunities
- Conclusion
- References

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.
- It should also create good opportunities for starting new businesses.

Outline

- Introduction
- Common Vulnerabilities and Defenses
- Objectives and Challenges
- Conclusion
- References

Selected References

- Huang *et al.*, “**Securing Web Application Code by Static Analysis and Runtime Protection**,” *WWW* 2004.
- Minamide, “**Static Approximation of Dynamically Generated Web Pages**,” *WWW* 2005.
- Xie and Aiken, “**Static Detection of Security Vulnerabilities in Scripting Languages**,” *USENIX Security Symposium* 2006.
- Su and Wassermann, “**The Essence of Command Injection Attacks in Web Applications**,” *POPL* 2006.
- Chess and West, *Secure Programming with Static Analysis*, Pearson Education, Inc. 2007.

Selected References (cont.)

- Lam *et al.*, “Securing Web Applications with Static and Dynamic Information Flow Tracking,” *PEPM* 2008.
- Yu *et al.*, “Verification of String Manipulation Programs Using Multi-Track Automata,” Tech Report, UCSB, 2009.
- Yu *et al.*, “Generating Vulnerability Signatures for String Manipulating Programs Using Automata-based Forward and Backward Symbolic Analyses,” *IEEE/ACM ICASE* 2009.
- Kiezun *et al.*, “Automatic Creation of SQL Injection and Cross-Site Scripting Attacks,” *ICSE* 2009.
- OWASP, <http://www.owasp.org/>.
- The CVE Site, <http://cve.mitre.org/>.