Highest Priority Cyber Security Risks

See www.sans.org/top-cyber-security-risks to view entire report

This report uses current data from appliances and software in thousands of targeted organizations to provide a reliable portrait of the attacks being launched and the vulnerabilities they exploit.

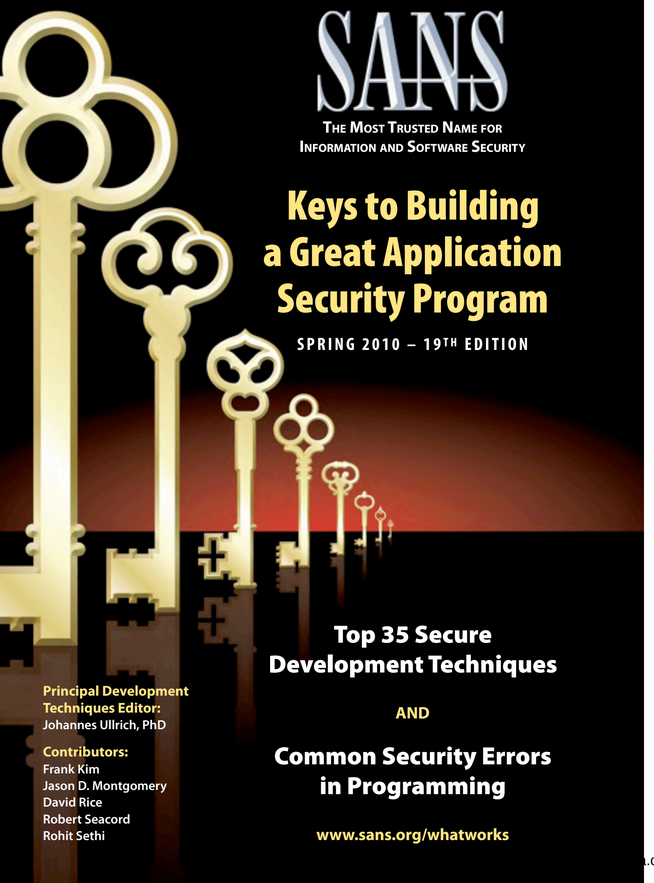
PRIORITY ONE: Client-side software that remains unpatched

Waves of targeted email attacks, often called spear phishing, are exploiting client-side vulnerabilities in commonly used programs such as Adobe PDF Reader, QuickTime, Adobe Flash, and Microsoft Office. This is currently the primary initial infection vector used to compromise computers that have Internet access. Those same client-side vulnerabilities are exploited by attackers who have infected visitors to insecure, but infected trusted web sites. Because the visitors feel safe downloading documents from the trusted sites, they are easily fooled into opening documents and music and video that exploit client-side vulnerabilities. Some exploits do not even require the user to open documents. Simply accessing an infected website is all that is needed to compromise the client software. The victims' infected computers are then used to propagate the infection and compromise other internal computers and sensitive servers incorrectly thought to be protected from unauthorized access by external entities. In many cases, the ultimate goal of the attacker is to steal data from the target organizations and also to install back doors through which the attackers can return for further exploitation. On average, major organizations take at least twice as long to patch client-side vulnerabilities as they take to patch critical operating system vulnerabilities.

PRIORITY TWO: Internet-facing Web sites that are vulnerable.

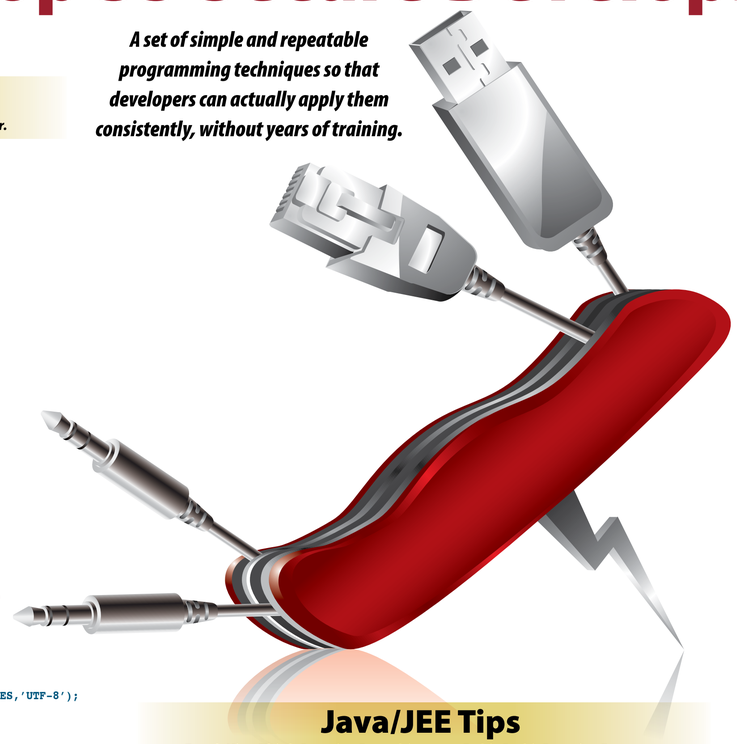
Attacks against Web applications constitute more than 60% of the total attack attempts observed on the Internet. These vulnerabilities are being exploited widely to convert trusted web sites into malicious Web sites serving content that contains client-side exploits. Web application vulnerabilities such as SQL injection and Cross-Site Scripting flaws in open-source as well as custom-built applications account for more than 80% of the vulnerabilities being discovered.

- Operating systems continue to have fewer vulnerabilities that can be remotely exploited and lead to massive Internet worms. Other than Conficker/Downadup, no new major worms for OSs were seen in the wild during the reporting period. Even so, the number of attacks against buffer overflow vulnerabilities in Windows tripled from May-June to July-August and constituted over 90% of attacks seen against the Windows operating system.
- World-wide there has been a significant increase over the past three years in the number of people discovering zero-day vulnerabilities, as measured by multiple independent teams discovering the same vulnerabilities at different times. Some vulnerabilities have remained unpatched for as long as two years. There is a corresponding shortage of highly skilled vulnerability researchers working for government and software vendors. So long as that shortage exists, the defenders will be at a significant disadvantage in protecting their systems against zero-day attacks. A large decline in the number of "PHP File Include" attacks appears to reflect improved processes used by application developers, system administrators, and other security professionals.



WHAT WORKS Presents Top 35 Secure Development Techniques

A set of simple and repeatable programming techniques so that developers can actually apply them consistently, without years of training.



PHP Tips

Johannes Ullrich, PhD, Chief technology officer of the Internet Storm Center.

1) Use prepared SQL statements.

BAD:

```
mysql_db_query("select id from users where username='$Username'")
```

BETTER:

```
$Stmt=$DB->prepare("select id from users where username=?");
$Stmt=$DB->bind_param("s",$Username);
$Stmt->execute();
```

2) Enable and configure Suhosin

See http://www.hardened-php.net/suhosin for details about Suhosin.

3) Extract data from super globals inside validation functions only

BAD:

```
$UserID=$_POST['userid'];
     if ( ! is_int($UserID) ) {
          $UserID=0;
     }
```

BETTER:

```
$UserID=get_userid('userid');
function get_userid($name) {
   $value=$_POST[$name];
   if ( is_int($value) ) {
         return $value;
   }
   return FALSE;
}
```

4) Replace "print" statements with a wrapper function escaping HTML tags like

BAD:

```
print $value;
```

BETTER:

```
safe_out($value);

function safe_out($value) {
    $value=htmlentities($value,ENT_QUOTES,'UTF-8');
    print $value;
}
```

5) Create a wrapper function to redirect users

BAD:

```
header("Location: $newlocation");
```

BETTER:

```
redir($newlocation,'enter reason for redirect');

function redir($newlocation,$reason) {
       locallogfunction($reason); /* to be defined */
       $lines=preg_split("/[\n\r]/",$newlocation);
       header("Location: ".$lines);
       exit();
}
```

6) Move include files outside of the document root

This is a configuration choice. A typical directory layout would look as follows:

```
/site/include  <- include path
     /html     <- DOCUMENT_ROOT
```

"Personally, I favor coding in unstructured languages like Perl and PHP for all the wrong reasons."

-Johannes Ullrich, PhD

C and C++ Tips

Robert Seacord, CERT Secure Coding Standards. https://www.securecoding.cert.org/

1) Validate input from all untrusted data sources.
2) Compile code using the highest warning level available for your compiler and eliminate warnings by modifying the code.
3) Create a software architecture and design your software to implement and enforce security policies.
4) Keep the design as simple and small as possible.
5) Base access decisions on permission rather than exclusion.
6) Adhere to the principle of least privilege.
7) Sanitize all data passed to complex subsystems such as command shells, relational databases, and off-the-shelf components.
8) Manage risk with multiple defensive strategies, so that if one layer of defense turns out to be inadequate, another layer of defense can prevent a security flaw from becoming an exploitable vulnerability and/or limit the consequences of a successful exploit.
9) Use effective quality assurance techniques.
10) Develop and/or apply a secure coding standard for your target development language and platform.

Java/JEE Tips

Frank Kim, Think Security Consulting • Rohit Sethi, Security Compass

1) Perform data validation with a security API such as OWASP ESAPI

See the following paper for some examples that use ESAPI for data validation:
http://www.sans.org/reading_room/application_security/protecting_web_apps.pdf

2) Use PreparedStatements with properly bound variables

BAD:

```
String query = "SELECT id FROM users WHERE userid = '" + userid + "'";
PreparedStatement stmt = con.prepareStatement(query);
ResultSet rs = stmt.executeQuery();
```

GOOD:

```
String query = "SELECT id FROM users WHERE userid = ?";
PreparedStatement stmt = con.prepareStatement(query);
query.setString(1, userid);
ResultSet rs = stmt.executeQuery();
```

3) Don't perform security-critical operations based on data from HttpServletRequest parameters

BAD:

```
String role = request.getParameter("role");
if (role != null && role.equals("admin") {
    // do admin stuff
}
```

4) Use a framework like Spring Security or ESAPI for authentication and authorization

See the following sites for additional information:

http://static.springsource.org/spring-security

http://www.owasp.org/index.php/ESAPI

5) Don't use instance variables in Servlets

BAD:

```
public class BadServlet extends HttpServlet {
    private String primaryKey;  // don't do this!
    ...
}
```

6) Use SecureRandom instead of Random

BAD:

```
Random random = new Random();
byte bytes[] = new byte[16];
random.nextBytes(bytes);
```

GOOD:

```
SecureRandom random = new SecureRandom();
byte bytes[] = new byte[16];
random.nextBytes(bytes);
```

.NET Tips

Jason D. Montgomery, Sr. Software Specialist/Security Specialist, Principal, ATGi (atgi.com)
David Rice, Director, The Monterey Group; Director, Policy Reform, U.S. Cyber Consequences Unit

1) For data validation, follow the Constrain, Reject/Replace, Assign *(to local variable)* paradigm.

2) Use a validation abstraction layer to make validating data easier and more consistent.

3) Validate data from any and all untrusted sources - including cookies, URL parameters, Form Fields, HTTP Headers, as well as inputs from external systems.

Code example combined for first three items above:

```
string sanitizedLastName = null;
if (ValidationUtility.TryValidateAndSanitizeLastName(txtLastName.Text, out
sanitizedLastName)) {
  // Success, use sanitizedLastName. Never use txtLastName.Text again (simplifies
  code review).
} else {
  // Failed, NEVER display txtLastName.Text back to user or use again in code
}

// Centralize Validation
public class ValidationUtility {
  public static bool TryValidateAndSanitizeLastName(string unsanitizedLastName,
                                                     out string sanitizedLastName) {
    // Fail Securely
    bool isValid = false;
    // Step 1: Constrain. Use whitelists, not blacklists.
    if (Regex.IsMatch(unsanitizedLastName, "^[a-z']+$", RegexOptions.IgnoreCase)) {
      // Step 2: Replace, substitue any potential bad characters with
      // something safe for storage. E.g., the tick ' char with the pipe | char
      unsanitizedLastName = unsanitizedLastName.Replace('\'', '|');
      isValid = true;
      // 3. Assign
      sanitizedLastName = unsanitizedLastName;
    } else {
      // Communicate intent to humans reading the code.
      isValid = false;
      sanitizedLastName = null;
    }
    return isValid;
  }
}
```

4) Use Microsoft's AntiXSS library to counter XSS attacks. Encode all untrusted output.

Available AntiXSS methods: `HtmlEncode()`, `HtmlAttributeEncode()`, `JavascriptEncode()`, `VisualBasicScriptEncode()`, `UrlEncode()`, `XmlEncode()`, `XmlAttributeEncode()`.

```
<div>Welcome, <%= AntiXss.HtmlEncode(Request.Form["FullName"]); %></div>
```

5) Use Anti-forgery Tokens in ASP.NET MVC 1.0 or include Session Key tokens in all Form POSTs to help protect against Cross-Site Request Forgery (CSRF) Attacks.

Example for ASP.NET*:

```
protected override void OnInit(EventArgs e) {
  if (User.Identity.IsAuthenticated) {
    this.ViewStateUserKey = User.Identity.Name;
  }
  base.OnInit(e);
}
```

** Make sure to set `<pages enableViewStateMac="true" />` in the `web.config` or in the `@Page` directive to guarantee the `ViewState` isn't modified by an attacker. Avoid HTTP GET Requests that use Query Parameters to perform work.*

6) Use parameterized SQL queries or LINQ to SQL when querying databases to protect against SQL Injection

```
SqlCommand cmd = new SqlCommand("SELECT UserName WHERE Username = @userName AND
PasswordHash = @passHash");
cmd.Parameters.Add("@userName", SqlDbType.VarChar, 20).Value = sanitizedUserName;
cmd.Parameters.Add("@password", SqlDbType.VarChar, 20).Value = passwordHash;
// ...etc.
```

7) Determine how you will make software security visible to development teams, for example Risk Density metrics

These metrics, while imperfect, can help provide a measure of the risk associated with code. Risk Density is calculated by dividing the number of high, medium, and low risk defects by the number of lines of code. Code reviews can provide the data points to capture this metric.

Risk Density = Risk Level / LoC

Examples:

10 Low-risk defects per 1000 lines of code

20 High-risk defects per 1000 lines of code

LoC - Lines of code (excludes comments, spaces, etc.)

Risk Level - High, Medium, or Low, determined by your organizations' standards and policies for code security.

8) Design windows and web applications that conform to the Principle of Least Privilege.

Some indicators that this principal is being violated by the software:

+ Granted Administrative permissions
+ Granted privileges in the computer's local Security Policy (e.g. Act as Part of the Operating System)

SANS Common Security Errors in Programming

Special thanks to the CWE Team at MITRE.

The SANS Common Security Errors in Programming map illustrates the software weaknesses that are responsible for the majority of the publicly known vulnerabilities reported in the Common Vulnerabilities and Exposures (CVE) and discovered in self-developed applications. It is based on the CWE (Common Weakness Enumeration) that provides a unified, measurable set of software weaknesses that will enable more effective discussion and action to find these weaknesses in source code and eliminate them. The CWE was developed by MITRE and sponsored by the Department of Homeland Security. The numbers between parentheses represent the CWE IDs for each weakness on the "2009 CWE/SANS Top 25 Most Dangerous Programming Errors" list. CWE IDs can be found at the MITRE CWE Website or accessed directly by putting the number (in place of ###) in the following URL:
http://cwe.mitre.org/data/definitions/###.html

SANS MITRE

2009's CWE/SANS Top 25 Most Dangerous Programming Errors

Handler Errors
- Deployment of Wrong Handler
- Missing Handler
- Dangerous Handler not Disabled During Sensitive Operations
- Unparsed Raw Web Content Delivery
- Incomplete Identification of Uploaded File Variables (PHP)
- Unrestricted File Upload

User Interface Errors
- UI Discrepancy for Security Feature
- Multiple Interpretations of UI Input
- UI Misrepresentation of Critical Information

Behavioral Problems
- Behavioral Change in New Version or Environment
- Expected Behavior Violation

Initialization and Cleanup Errors
- Insecure Default Variable Initialization
- External Initialization of Trusted Variables
- Non-exit on Failed Initialization
- Missing Initialization
- Incomplete Cleanup
- Improper Cleanup on Thrown Exception
- **Improper Initialization - (665)**

Data Handling

Numeric Errors
- Use of Incorrect Byte Ordering
- Unchecked Array Indexing
- Incorrect Conversion between Numeric Types
  - Unexpected Sign Extension
  - Signed to Unsigned Conversion Error
  - Unsigned to Signed Conversion Error
  - Numeric Truncation Error
- **Incorrect Calculation - (682)**
  - Incorrect Calculation of Buffer Size
  - Integer Overflow or Wraparound
  - Integer Underflow (Wrap or Wraparound)
  - Off-by-one Error
  - Divide By Zero

Representation Errors
- Cleansing, Canonicalization, and Comparison Errors
- Reliance on Data/Memory Layout

Information Management Errors
- Information Leak (Information Disclosure)
  - Information Leak Through Sent Data
  - Privacy Leak through Data Queries
  - Discrepancy Information Leaks
  - **Error Message Information Leak - (209)**
  - Cross-boundary Cleansing Information Leak
  - Intended Information Leak
  - Process Environment Information Leak
  - Information Leak Through Debug Information
  - Sensitive Information Uncleared Before Release
  - Information Leak of System Data
  - Information Leak Through Caching
  - Information Leak Through Environmental Variables
  - File and Directory Information Leaks
  - Information Leak Through Query Strings in GET Request
  - Information Leak Through Indexing of Private Data
- Information Loss or Omission
- Containment Errors (Container Errors)

Improper Access of Indexable Resource ('Range Error')

Type Errors

Improper Encoding or Escaping of Output - (116)

String Errors

Data Structure Issues

Improper Handling of Syntactically Invalid Structure

Modification of Assumed-Immutable Data (MAID)

Improper Input Validation - (20)
- Pathname Traversal and Equivalence Errors
- Process Control
- Missing XML Validation
- Failure to Sanitize Data into a Different Plane ('Injection')
  - Improper Sanitization of Special Elements used in a Command ('Command Injection')
    - **Improper Sanitization of Special Elements used in an OS Command ('OS Command Injection') - (78)**
  - **Failure to Preserve Web Page Structure ('Cross-site Scripting') - (79)**
  - **Improper Sanitization of Special Elements used in an SQL Command ('SQL Injection') - (89)**
  - Failure to Sanitize Data into LDAP Queries ('LDAP Injection')
  - XML Injection (aka Blind XPath Injection)
  - Failure to Sanitize CRLF Sequences ('CRLF Injection')
  - Uncontrolled Format String
  - Failure to Sanitize Special Elements into a Different Plane
  - Argument Injection or Modification
  - Improper Control of Resource Identifiers ('Resource Injection')
  - **Failure to Control Generation of Code ('Code Injection') - (94)**
  - Improper Sanitization of Special Elements
- Technology-Specific Input Validation Problems
- Misinterpretation of Input
- Unchecked Input for Loop Condition
- Null Byte Interaction Error (Poison Null Byte)
- Direct Use of Unsafe JNI
- Improper Output Sanitization for Logs
- **Failure to Constrain Operations within the Bounds of a Memory Buffer - (119)**
- Use of Externally-Controlled Input to Select Classes or Code ('Unsafe Reflection')
- ASP.NET Misconfiguration: Not Using Input Validation Framework
- URL Redirection to Untrusted Site ('Open Redirect')
- Variable Extraction Error
- Unvalidated Function Hook Arguments
- **External Control of File Name or Path - (73)**
- Improper Address Validation in IOCTL with METHOD_NEITHER I/O Control Code
- Use of Path Manipulation Function without Maximum-sized Buffer

Channel and Path Errors
- Channel Errors
- Failure to Protect Alternate Path
- Uncontrolled Search Path Element
- Unquoted Search Path or Element
- **Untrusted Search Path - (426)**

Error Handling
- Error Conditions, Return Values, Status Codes
- Failure to Use a Standardized Error Handling Mechanism
- Failure to Catch All Exceptions in Servlet
- Not Failing Securely ('Failing Open')
- Missing Custom Error Page

Pointer Issues
- Return of Pointer Value Outside of Expected Range
- Use of size of() on a Pointer Type
- Incorrect Pointer Scaling
- Use of Pointer Subtraction to Determine Size
- Assignment of a Fixed Address to a Pointer
- Attempt to Access Child of a Non-structure Pointer

Time and State

State Issues
- Incomplete Internal State Distinction
- State Synchronization Error
- Mutable Objects Passed by Reference
- Passing Mutable Objects to an Untrusted Method
- **External Control of Critical State Data - (642)**
- **Race Condition - (362)**

Session Fixation

Concurrency Issues

Temporary File Issues

Covert Timing Channel

Technology-Specific Time and State Issues

Symbolic Name not Mapping to Correct Object

Signal Errors

Unrestricted Externally Accessible Lock

Double-Checked Locking

Insufficient Session Expiration

Insufficient Synchronization

Use of a Non-reentrant Function in an Unsynchronized Context

Improper Control of a Resource Through its Lifetime

Exposure of Resource to Wrong Sphere

Incorrect Resource Transfer Between Spheres

Use of a Resource after Expiration or Release

External Influence of Sphere Definition

Uncontrolled Recursion

Redirect Without Exit

Failure to Fulfill API Contract ('API Abuse')
- Failure to Clear Heap Memory Before Release ('Heap Inspection')
- Call to Non-ubiquitous API
- Use of Inherently Dangerous Function
- Multiple Binds to the Same Port
- J2EE Bad Practices: Direct Management of Connections
- Incorrect Check of Function Return Value
- Often Misused: Arguments and Parameters
- Uncaught Exception
- **Execution with Unnecessary Privileges - (250)**
- Often Misused: String Management
- J2EE Bad Practices: Direct Use of Sockets
- Unchecked Return Value
- Failure to Change Working Directory in chroot Jail
- Reliance on DNS Lookups in a Security Decision
- Failure to Follow Specification
- Failure to Provide Specified Functionality

Web Problems
- Failure to Sanitize CRLF Sequences in HTTP Headers ('HTTP Response Splitting')
- Inconsistent Interpretation of HTTP Requests ('HTTP Request Smuggling')
- Improper Sanitization of HTTP Headers for Scripting Syntax
- Use of Non-Canonical URL Paths for Authorization Decisions

Indicator of Poor Code Quality
- NULL Pointer Dereference
- Incorrect Block Delimitation
- Omitted Break Statement in Switch
- Undefined Behavior for Input to API
- Use of Hard-coded, Security-relevant Constants
- Unsafe Function Call from a Signal Handler
- Suspicious Comment
- Return of Stack Variable Address
- Missing Default Case in Switch Statement
- Expression Issues
- Use of Obsolete Functions
- Use of Function with Inconsistent Implementations
- Unused Variable
- Dead Code
- Resource Management Errors
- **Improper Resource Shutdown or Release - (404)**
- Empty Synchronized Block
- Explicit Call to Finalize()
- Reachable Assertion
- Use of Potentially Dangerous Function

Security Features

Credentials Management
- **Hard-Coded Password - (259)**
- Unverified Password Change
- Missing Password Field Masking
- Weak Cryptography for Passwords
- Weak Password Requirements
- Not Using Password Aging
- Password Aging with Long Expiration
- Insufficiently Protected Credentials
- Weak Password Recovery Mechanism for Forgotten Password

Insufficient Verification of Data Authenticity
- Origin Validation Error
- Improper Verification of Cryptographic Signature
- Use of Less Trusted Source
- Acceptance of Extraneous Untrusted Data With Trusted Data
- Improperly Trusted Reverse DNS
- Insufficient Type Distinction
- **Cross-Site Request Forgery (CSRF) - (352)**
- Failure to Add Integrity Check Value
- Improper Validation of Integrity Check Value
- Trust of System Event Data
- Reliance on File Name or Extension of Externally-Supplied File
- Reliance on Obfuscation or Encryption of Security-Relevant Inputs without Integrity Checking

Privacy Violation

Reliance on Cookies without Validation and Integrity Checking in a Security Decision

Reliance on Cookies without Validation and Integrity Checking

Client-Side Enforcement of Server-Side Security - (602)

Improperly Implemented Security Check for Standard

Improper Authentication

User Interface Security Issues

Use of Insufficiently Random Values - (330)

Logging of Excessive Data

Certificate Issues

Security Decision

Cryptographic Issues
- Key Management Errors
- Missing Required Cryptographic Step
- Not Using a Random IV with CBC Mode
- Failure to Encrypt Sensitive Data
  - Cleartext Storage of Sensitive Information
  - **Cleartext Transmission of Sensitive Information - (319)**
  - Sensitive Cookie in HTTPS Session Without 'Secure' Attribute
- Reversible One-Way Hash
- Inadequate Encryption Strength
  - **Use of a Broken or Risky Cryptographic Algorithm - (327)**
- Use of RSA Algorithm without OAEP

Permissions, Privileges, and Access Controls
- Access Control (Authorization) Issues
  - **Improper Access Control (Authorization) - (285)**
  - Access Control Bypass Through User-Controlled Key
  - Use of Non-Canonical URL Paths for Authorization Decisions
- Permission Issues
  - Incorrect Default Permissions
  - Insecure Inherited Permissions
  - Insecure Preserved Inherited Permissions
  - Incorrect Execution-Assigned Permissions
  - Improper Handling of Insufficient Permissions or Privileges
  - Improper Preservation of Permissions
  - Exposed Unsafe ActiveX Method
  - **Incorrect Permission Assignment for Critical Resource - (732)**
  - Permission Race Condition During Resource Copy
- Privilege / Sandbox Issues
- Improper Ownership Management
- Incorrect User Management

Password in Configuration File

Insufficient Compartmentalization

Reliance on a Single Factor in a Security Decision

Insufficient Psychological Acceptability

Reliance on Security through Obscurity

Protection Mechanism Failure

Insufficient Logging

Insufficient Encapsulation

Mobile Code Issues
- Public cloneable() Method Without Final ('Object Hijack')
- Use of Inner Class Containing Sensitive Data
- Critical Public Variable Without Final Modifier
- **Download of Code Without Integrity Check - (494)**
- Array Declared Public, Final, and Static
- finalize() Method Declared Public

Leftover Debug Code

Use of Dynamic Class Loading

clone() Method Without super.clone()

Comparison of Classes by Name

Data Leak Between Sessions

Trust Boundary Violation

Reliance on Package-level Scope

J2EE Framework: Saving Unserializable Objects to Disk

Deserialization of Untrusted Data

Serializable Class Containing Sensitive Data

Information Leak through Class Cloning

Public Data Assigned to Private Array-Typed Field

Private Array-Typed Field Returned From A Public Method

Public Static Final Field References Mutable Object

Exposed Dangerous Method or Function

Critical Variable Declared Public

Access to Critical Private Variable via Public Method