

Web Application Security Threats and Counter Measures

Sang Shin

sang.shin@sun.com

www.javapassion.com

Java Technology Evangelist

Sun Microsystems, Inc.

**Today's Presentation is available from
<http://www.javapassion.com>**

**Demo is available from
[http://www.javapassion.com/handsonlabs/
4010_webappsecurity.zip](http://www.javapassion.com/handsonlabs/4010_webappsecurity.zip)**

Agenda

- How real is the Web application threat?
- 10 most common Web application threats and counter measures
- Security principles
- Tools

95% of Web Apps Have Vulnerabilities

- Cross-site scripting (80 percent)
- SQL injection (62 percent)
- Parameter tampering (60 percent)
- Cookie poisoning (37 percent)
- Database server (33 percent)
- Web server (23 percent)
- Buffer overflow (19 percent)

OWASP Top 10 Web Security Threats

1. Unvalidated input
2. Broken access control
3. Broken authentication
4. Cross-site scripting (XSS)
5. Buffer overflows
6. Injection flaws
7. Improper error handling
8. Insecure storage
9. Application denial-of-service
10. Insecure configuration management

#1: Unvalidated Input: Mother of All Web-tier Security Threats

#1: Unvalidated Input (Description)

- Attacker can easily tamper any part of the HTTP request before submitting
 - > URL
 - > Cookies
 - > Form fields
 - > Hidden fields
 - > Headers
- Common names for common input tampering attacks
 - > forced browsing, command insertion, cross site scripting, buffer overflows, format string attacks, SQL injection, cookie poisoning, and hidden field manipulation

#1: Unvalidated Input (Solutions)

- Do rigorous input data validation
 - > All parameters should be validated before use
- Do server-side validation
 - > Client side validation could be bypassed by the attacker easily
 - > Client side validation is to be used mainly for quick user responsiveness
- Do canonicalization of input data
 - > The process of simplifying the encoding

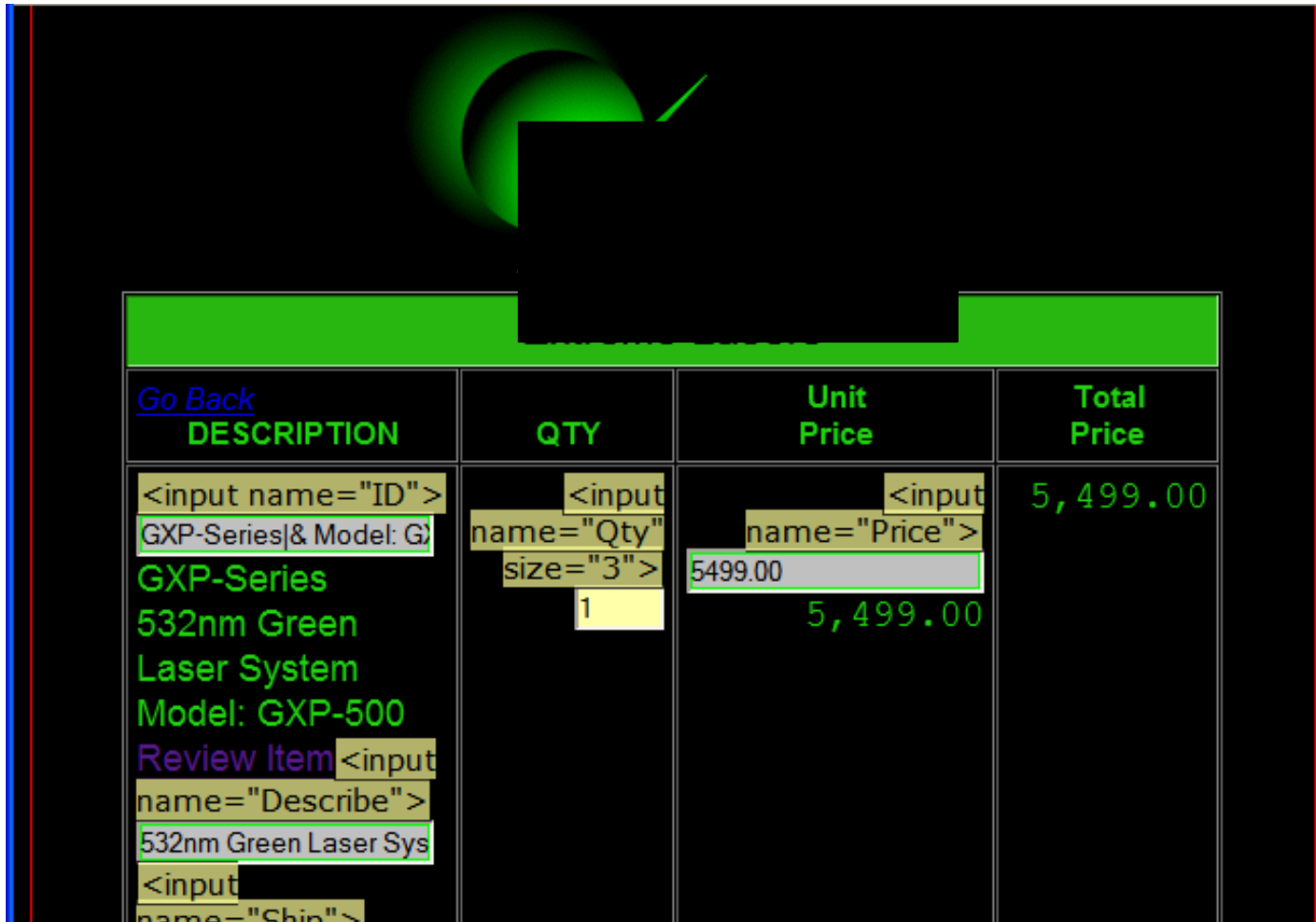
#1: Unvalidated Input (Solutions)

- Use centralized code for input validation
 - > Scattered code is hard to maintain
- Each parameter should be checked against a strict format that specifies exactly what input will be allowed
 - > This is called “positive” input validation
 - > “Negative” approaches that involve filtering out certain bad input or approaches that rely on signatures are not likely to be effective and may be difficult to maintain

#1: Unvalidated Input (Solutions)

- Validation Criteria
 - > Data type (string, integer, real, etc...)
 - > Allowed character set
 - > Minimum and maximum length
 - > Whether null is allowed
 - > Whether the parameter is required or not
 - > Whether duplicates are allowed
 - > Numeric range
 - > Specific legal values (enumeration)
 - > Specific patterns (regular expressions)

What's Wrong With This Picture?



Go Back		Unit Price	Total Price
DESCRIPTION	QTY		
<input type="text" value="GXP-Series & Model: G"/> GXP-Series 532nm Green Laser System Model: GXP-500 Review Item <input type="text" value="532nm Green Laser Sys"/> <input type="text" value=""/>	<input type="text" value="1"/>	<input type="text" value="5499.00"/> 5,499.00	5,499.00

#1: Unvalidated Input (Example)

```
public void doPost(HttpServletRequest req,...) {  
    String customerId =  
        req.getParameter("customerId");  
    String sku = req.getParameter("sku");  
    String stringPrice = req.getParameter("price");  
    Integer price = Integer.valueOf(stringPrice);  
    // Store in the database without input validation  
    // What happens if a hacker provides his own  
    // price as a value of "price" form field?  
    orderManager.submitOrder(sku, customerId, price);  
} // end doPost
```

#1: Unvalidated Input (Corrected)

```
public void doPost(HttpServletRequest req,...) {  
    // Get customer data  
    String customerId =  
        req.getParameter("customerId");  
    String sku = req.getParameter("sku");  
    // Get price from database  
    Integer price = skuManager.getPrice(sku);  
    // Store in the database  
    orderManager.submitOrder(sku, customerId, price);  
} // end doPost
```

#1: Unvalidated Input (Tools)

- OWASP's WebScarab
 - > By submitting unexpected values in HTTP requests and viewing the web application's responses, you can identify places where tainted parameters are used
- Stinger HTTP request validation engine (stinger.sourceforge.net)
 - > Developed by OWASP for J2EE environments

#2: Broken Access Control

#2: Broken Access Control (Examples)

- Insecure ID's
- Forced browsing pass access control checking
- Path traversal
- File permissions

Index of /config/firewall - Microsoft Internet Explorer provided by Ci...

File Edit View Favorites Tools Help

Back Forward Stop Refresh Home Search Favorites

Address [Redacted] Go

Index of /config/firewall

	Name	Last modified	Size	Descripti
	192.168.99.reverse	19-Dec-2004 17:35	1k	
	base.ldif	19-Dec-2004 19:26	1k	
	cisco.zone	19-Dec-2004 13:02	1k	
	dhcpd.conf	14-Dec-2004 14:56	2k	
	iptables-rules.forward	07-Mar-2005 07:04	1k	
	iptables-rules.snaf	07-Mar-2005 07:04	1k	
	ldap.conf	27-Jan-2005 19:07	1k	
	named.conf	12-Dec-2004 16:42	2k	
	nsswitch.conf	19-Dec-2004 19:08	2k	
	slapd.conf	07-Mar-2005 06:06	4k	
	smb.conf	11-Mar-2005 11:27	12k	
	system-auth	07-Mar-2005 06:09	1k	

Internet

#3: Broken Authentication & Session Management

#3: Broken Authentication & Session Management

- Includes all aspects of handling user authentication and managing active sessions
- Session hi-jacking
 - > If the session tokens (Cookies) are not properly protected, an attacker can hijack an active session and assume the identity of a user

#3: Broken Account/Session Management (Client Example—SSO)

```
public void doGet(HttpServletRequest req,...) {  
    // Get user name  
    String userId = req.getRemoteUser();  
    // Generate cookie with no encryption  
    Cookie ssoCookie =  
        new Cookie("userid",userId);  
    ssoCookie.setPath("/");  
    ssoCookie.setDomain("cisco.com");  
    response.addCookie(ssoCookie);  
    ...  
}
```

#3: Broken Account/Session Management (Server Example—SSO)

```
public void doGet(HttpServletRequest req,...) {  
    // Get user name  
    Cookie[] cookies = req.Cookies();  
    for (i=0; i < cookies.length; i++) {  
        Cookie cookie = cookies[i];  
        if (cookie.getName().equals("ssoCookie")) {  
            String userId = cookie.getValue();  
            HttpSession session = req.getSession();  
            session.setAttribute("userId",userId);  
        } // end if  
    } // end for  
} // end doGet
```

#3: Safe Account/Session Management (Client Solution—SSO)

```
public void doGet(HttpServletRequest req,...) {  
    // Get user name  
    String userId = req.getRemoteUser();  
    // Encrypt the User ID before passing it  
    // to the client as part of a cookie.  
    encryptedUserId = Encrypter.encrypt(userId);  
    Cookie ssoCookie =  
        new Cookie("userid", encrypteduserId);  
    ssoCookie.setPath("/");  
    ssoCookie.setDomain("cisco.com");  
    response.addCookie(ssoCookie);  
    ...  
}
```

#3: Safe Account/Session Management (Server Solution—SSO)

```
public void doGet(HttpServletRequest req,...) {  
    // Get user name  
    Cookie[] cookies = req.Cookies();  
    for (i=0; i < cookies.length; i++) {  
        Cookie cookie = cookies[i];  
        if (cookie.getName().equals("ssoCookie")) {  
            String encryptedUserId = cookie.getValue();  
            String userId = Encrypter.decrypt(encryptedUserId);  
            if (isValid(userId)) {  
                HttpSession session = req.getSession();  
                session.setAttribute("userId",userId);  
            } // end if isValid..  
        } // end if cookie = ssoCookie..  
    } // end for  
} // end doGet
```

#4 Cross Site Scripting (XSS)

#4: Cross Site Scripting (Description)

- An attacker can use cross site scripting technique to implement malicious script (into a server), which is then sent to unsuspecting users accessing the same server
 - > Example: Chat server
- The attacked user's (victim's) browser has no way to know that the script should not be trusted, and will execute the script
 - > Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by your browser and used with that site
 - > These scripts can even rewrite the content of the HTML page

#4: Cross Site Scripting (Description)

- XSS attacks usually come in the form of embedded JavaScript
 - > However, any embedded active content is a potential source of danger, including: ActiveX (OLE), VBscript, Shockwave, Flash and more

#4: Consequences of Cross Site Scripting (Examples)

- Disclosure of the user's session cookie – session high-jacking
- Disclosure of end user files
- Installation of Trojan horse programs
- Redirecting the user to some other page or site
- Modifying presentation of content

#4: Cross Site Scripting (How to Find them)

- Search for all places where input from an HTTP request could possibly make its way into the HTML output

#4: Cross Site Scripting (Counter Measures)

- Validate all inputs, especially those inputs that will later be used as parameters to OS commands, scripts, and database queries
- It is particularly important for content that will be permanently stored somewhere
- Users should not be able to create message content that could cause another user to load an undesirable page or undesirable content when the user's message is retrieved

#4: Cross Site Scripting (Counter Measures)

- Validate input against a rigorous positive specification of what is expected
 - > Validation of all headers, cookies, query strings, form fields, and hidden fields (i.e., all parameters) against a rigorous specification of what should be allowed
 - > ‘Negative’ or attack signature based policies are difficult to maintain and are likely to be incomplete
 - > White-listing: a-z, A-Z, 0-9, etc.
 - > Truncate input fields to reasonable length

#4: Cross Site Scripting (Counter Measures)

- Encode user supplied output
 - > Preventing inserted scripts from being transmitted to users in an executable form
- Applications can gain significant protection from javascript based attacks by converting the following characters in all generated output to the appropriate HTML entity encoding:
 - > from “<” to “<”
 - > from “>” to “>”
 - > from “(” to “(”
 - > from “)” to “)”
 - > from “#” to “#”
 - > from “&” to “&”

#4: Cross-Site Scripting (Flawed)

```
protected void doPost(HttpServletRequest req, HttpServletResponse res)
{
    String title = req.getParameter("TITLE");
    String message = req.getParameter("MESSAGE");
    try {
        connection = DatabaseUtilities.makeConnection(s);
        PreparedStatement statement =
            connection.prepareStatement("INSERT INTO messages VALUES (?,?)");
        // The "title" and "message" are saved into
        // the database. These "title" and "message"
        // might contain ill-intended JavaScript.
        statement.setString(1,title);
        statement.setString(2,message);
        statement.executeUpdate();
    } catch (Exception e) {
        ...
    } // end catch
} // end doPost
```

#4: Cross-Site Scripting (Solution)

```
private static String stripEvilChars(String evilInput) {
    Pattern evilChars = Pattern.compile("[^a-zA-Z0-9]");
    return evilChars.matcher(evilInput).replaceAll("");
}

protected void doPost(HttpServletRequest req, HttpServletResponse res) {
    // Do vigorous input validation
    String title = stripEvilChars(req.getParameter("TITLE"));
    String message = stripEvilChars(req.getParameter("MESSAGE"));
    try {
        connection = DatabaseUtilities.makeConnection(s);
        PreparedStatement statement =
            connection.prepareStatement
                ("INSERT INTO messages VALUES (?,?)");
        statement.setString(1,title);
        statement.setString(2,message);
        statement.executeUpdate();
    } catch (Exception e) {
        ...
    } // end catch
} // end doPost
```

Cross Site Scripting Demo

Demo Scenario (Stored XSS)

- The server is a chat server
- The chat server displays whatever message that is typed in by a particular user to all other users
- An attacker (user A) implements JavaScript as part of a message (message A)
- The chat server saves the message (into the database or whatever storage) without input validation
- When unsuspecting user (user B) reads the message A, the JavaScript will be executed

Demo Scenario (Reflected XSS)

- Whatever typed in by a user is reflected back to a browser
- A mal-intended JavaScript will be reflected back to a browser

#5 Buffer Overflow

#5: Buffer Overflow Errors (Description)

- Attackers use buffer overflows to corrupt the execution stack of a web application
 - > By sending carefully crafted input to a web application, an attacker can cause the web application to execute arbitrary code
- Buffer overflow flaws can be present in both the web server or application server products or the web application itself
- Not generally an issue with Java apps
 - > Java type checking

#6 Injection Flaws

#6: Injection Flaws (Description)

- Injection flaws allow attackers to relay malicious code through a web application to another system
 - > Calls to the operating system via system calls
 - > The use of external programs via shell commands
 - > Calls to backend databases via SQL (i.e., SQL injection)
- Any time a web application uses an interpreter of any type, there is a danger of an injection attack

#6: Injection Flaws (Description)

- Many web applications use operating system features and external programs to perform their functions
 - > `Runtime.exec()` to external programs (like `sendmail`)
- When a web application passes information from an HTTP request through as part of an external request, the attacker can inject special (meta) characters, malicious commands, or command modifiers into the information

#6: Injection Flaws (Example)

- SQL injection is a particularly widespread and dangerous form of injection
 - > To exploit a SQL injection flaw, the attacker must find a parameter that the web application passes through to a database
 - > By carefully embedding malicious SQL commands into the content of the parameter, the attacker can trick the web application into forwarding a malicious query to the database

#6: Injection Flaws (Examples)

- Path traversal
 - > “../” characters as part of a filename request
- Additional commands could be tacked on to the end of a parameter that is passed to a shell script to execute an additional shell command
 - > “; rm -r *”
- SQL queries could be modified by adding additional ‘constraints’ to a where clause
 - > “OR 1=1”

#6: Injection Flaws (How to find them)

- Search the source code for all calls to external resources
 - > e.g., system, exec, fork, Runtime.exec, SQL queries, or whatever the syntax is for making requests to interpreters in your environment

#6: Injection Flaws (Counter Measures)

- Avoid accessing external interpreters wherever possible
 - > Use library API's instead
- Structure many requests in a manner that ensures that all supplied parameters are treated as data, rather than potentially executable content
 - > For SQL, use PreparedStatement or Stored procedures
- Ensure that the web application runs with only the privileges it absolutely needs to perform its function

#6: SQL Injection (Counter Measures)

- When making calls to backend databases, carefully validate the data provided to ensure that it does not contain any malicious content
- Use PreparedStatement or Stored procedures

SQL Injection Demo

Demo Scenario

- A user access database through a web server to view his creditcard number by giving a userid
- A web server builds an SQL query to the database server using the user-entered userid without performing an input validation
- An attacker sends “.. OR 1=1” as part of userid
- The database server displays all users

#7: Improper Error Handling

#7: Improper Error Handling (Description)

- The most common problem is when detailed internal error messages such as stack traces, database dumps, and error codes are displayed to a potential hacker
 - > These messages reveal implementation details that should never be revealed
- Other errors can cause the system to crash or consume significant resources, effectively denying or reducing service to legitimate users
- Left-over during debugging process
- Inconsistent errors may reveal internal info.
 - > “File not found” vs. “Access denied”

#7: Improper Error Handling (Counter Measures)

- The errors must be handled according to a well thought out scheme that will
 - > provide a meaningful error message to the user
 - > provide diagnostic information to the site maintainers
 - > provide no useful information to an attacker
- All security mechanisms should deny access until specifically granted, not grant access until denied

#7: Improper Error Handling (Counter Measures)

- Good error handling mechanisms should be able to handle any feasible set of inputs, while enforcing proper security
- Error handling should not focus solely on input provided by the user, but should also include any errors that can be generated by internal components such as system calls, database queries, or any other internal functions

#7: Improper Error Handling (Counter Measures)

- A specific policy for how to handle errors should be documented, including
 - > The types of errors to be handled
 - > For each, what information is going to be reported back to the user
 - > What information is going to be logged
- All developers need to understand the policy and ensure that their code follows it
 - > An architect should play a role of coming up and enforcing a company-wide policy

#7: Improper Error Handling (Counter Measures)

- In the implementation, ensure that the site is built to gracefully handle all possible errors.
 - > When errors occur, the site should respond with a specifically designed result that is helpful to the user without revealing unnecessary internal details.
 - > Certain classes of errors should be logged to help detect implementation flaws in the site and/or hacking attempts.

#7: Improper Error Handling (Counter Measures)

- Very few sites have any intrusion detection capabilities in their web application, but it is certainly conceivable that a web application could track repeated failed attempts and generate alerts
 - > Note that the vast majority of web application attacks are never detected because so few sites have the capability to detect them. Therefore, the prevalence of web application security attacks is likely to be seriously underestimated

What's Wrong With This Picture?

Could not obtain post/user information.

DEBUG MODE

SQL Error : 1016 Can't open file: 'nuke_bbposts_text.MYD'. (errno: 145)

```
u.username, u.user_id, u.user_posts, u.user_from, u.user_website, u.user_email, u.user_icq, u.user_aim, u.user_y
.user_regdate, u.user_msnm, u.user_viewemail, u.user_rank, u.user_sig, u.user_sig_bbcodes_uid, u.user_avatar,
ser_avatar_type, u.user_allowavatar, u.user_allowsmile, p.*, pt.post_text, pt.post_subject, pt.bbcodes_uid FROM
bposts p, nuke_users u, nuke_bbposts_text pt WHERE p.topic_id = '1547' AND pt.post_id = p.post_id AND u.user_ic
p.poster_id ORDER BY p.post_time ASC LIMIT 0, 15
```

Line : 435

File : /usr/home/geeks/www/vonage/modules/Forums/viewtopic.php

#7: Improper Error Handling (Flaw)

```
protected void doPost(HttpServletRequest req, HttpServletResponse res)
{
    String query =
        "SELECT userid, name FROM user_data WHERE accountnum = '"
        + req.getParameter("ACCT_NUM") + "'";
    PrintWriter out = res.getWriter();
    // HTML stuff to out.println...
    try {
        connection = DatabaseUtilities.makeConnection(s);
        Statement statement = connection.createStatement();
        ResultSet results = statement.executeQuery(query);
        while (results.next()) {
            out.println("<TR><TD>" + rset.getString(1) + "</TD>");
            out.println("<TD>" + rset.getString(2) + "</TD>");
        } // end while
    } catch (Exception e) {
        e.printStackTrace(out);
    } // end catch
} // end doPost
```

#7: Improper Error Handling (Solution)

```
protected void doPost(HttpServletRequest req, HttpServletResponse res)
{
    String query =
        "SELECT userid, name FROM user_data WHERE accountnum = '"
        + req.getParameter("ACCT_NUM") + "'";
    PrintWriter out = res.getWriter();
    // HTML stuff to out.println...
    try {
        connection = DatabaseUtilities.makeConnection(s);
        Statement statement = connection.createStatement();
        ResultSet results = statement.executeQuery(query);
        while (results.next ()) {
            out.println("<TR><TD>" + rset.getString(1) + "</TD>");
            out.println("<TD>" + rset.getString(2) + "</TD>");
        } // end while
    } catch (Exception e) {
        Logger logger = Logger.getLogger();
        logger.log(Level.SEVERE, "Error retrieving account number", e);
        out.println("Sorry, but we are unable to retrieve this account");
    } // end catch
}
```

#9 Application Denial Of Service (DOS)

#9: Application DOS (Description)

- Types of resources
 - > Bandwidth, database connections, disk storage, CPU, memory, threads, or application specific resources
- Application level resources
 - > Heavy object allocation/reclamation
 - > Overuse of logging
 - > Unhandled exceptions
 - > Unresolved dependencies on other systems
 - > Web services
 - > Databases

#9: Application DOS (How to determine your vulnerability)

- Load testing tools, such as JMeter can generate web traffic so that you can test certain aspects of how your site performs under heavy load
 - > Certainly one important test is how many requests per second your application can field
 - > Testing from a single IP address is useful as it will give you an idea of how many requests an attacker will have to generate in order to damage your site
- To determine if any resources can be used to create a denial of service, you should analyze each one to see if there is a way to exhaust it

#9: Application DOS (Counter Measures)

- Limit the resources allocated to any user to a bare minimum
- For authenticated users
 - > Establish quotas so that you can limit the amount of load a particular user can put on your system
 - > Consider only handling one request per user at a time by synchronizing on the user's session
 - > Consider dropping any requests that you are currently processing for a user when another request from that user arrives

#9: Application DOS (Counter Measures)

- For un-authenticated users
 - > Avoid any unnecessary access to databases or other expensive resources
 - > Caching the content received by un-authenticated users instead of generating it or accessing databases to retrieve it
- Check your error handling scheme to ensure that an error cannot affect the overall operation of the application

Other Web Applications Security Threats

Other Web Application Security Threats

- Unnecessary and Malicious Code
- Broken Thread Safety and Concurrent Programming
- Unauthorized Information Gathering
- Accountability Problems and Weak Logging
- Data Corruption
- Broken Caching, Pooling, and Reuse

Broken Thread Safety Demo

Demo Scenario

- A servlet uses static variable called currentUser to set the username and then displays the value of it
- A servlet can be accessed by multiple clients
- A servlet is not written to be multi-thread safe
- The instance variable can be in race-condition
 - > Browser A sets the username to jeff
 - > Browser B sets the username to dave
 - > If these two browsers access the servlet almost at the same time, both browsers display one of the two names

Principles of Secure Programming

Principles of Secure Programming

1. Minimize attack surface area
2. Secure defaults
3. Principle of least privilege
4. Principle of defense in depth
5. Fail securely
6. External systems are insecure
7. Separation of duties
8. Do not trust security through obscurity
9. Simplicity
10. Fix security issues correctly

Minimize Attack Surface Area

- The aim for secure development is to reduce the overall risk by reducing the attack surface area
- Every feature that is added to an application adds a certain amount of risk to the overall application
 - > The value of adding a feature needs to be accessed from security risk standpoint

Secure Defaults

- There are many ways to deliver an “out of the box” experience for users. However, by default, the experience should be secure, and it should be up to the user to reduce their security – if they are allowed
- Example:
 - > By default, password aging and complexity should be enabled
 - > Users might be allowed to turn off these two features to simplify their use of the application

Principle of Least Privilege

- Accounts have the least amount of privilege required to perform their business processes.
 - > This encompasses user rights, resource permissions such as CPU limits, memory, network, and file system permissions
- Example
 - > If a middleware server only requires access to the network, read access to a database table, and the ability to write to a log, this describes all the permissions that should be granted

Principle of Defense In Depth

- Controls, when used in depth, can make severe vulnerabilities extraordinarily difficult to exploit and thus unlikely to occur.
 - > With secure coding, this may take the form of tier-based validation, centralized auditing controls, and requiring users to be logged on all pages

Fail Safely

- Applications regularly fail to process transactions for many reasons. How they fail can determine if an application is secure or not
- Example: In the code below, if `codeWhichMayFail()` fails, the attacker gets an admin privilege

```
isAdmin = true;
try {
    codeWhichMayFail();
    isAdmin = isUserInRole( "Administrator" );
}
catch (Exception ex) {
    log.write(ex.toString());
}
```

External Systems Are Insecure

- Implicit trust of externally run systems is not warranted
 - > All external systems should be treated in a similar fashion
- Example:
 - > A loyalty program provider provides data that is used by Internet Banking, providing the number of reward points and a small list of potential redemption items
 - > However, the data should be checked to ensure that it is safe to display to end users, and that the reward points are a positive number, and not improbably large

Separation of Duties

- A key fraud control is separation of duties
- Certain roles have different levels of trust than normal users
 - > In particular, Administrators are different to normal users. In general, administrators should not be users of the application
- Example
 - > An administrator should be able to turn the system on or off, set password policy but shouldn't be able to log on to the storefront as a super privileged user, such as being able to “buy” goods on behalf of other users.

Do Not Trust Security Through Obscurity

- Security through obscurity is a weak security control, and nearly always fails when it is the only control
 - > This is not to say that keeping secrets is a bad idea, it simply means that the security of key systems should not be reliant upon keeping details hidden
- Example
 - > The security of an application should not rely upon only on knowledge of the source code being kept secret
 - > The security of an application should rely upon many other factors, including reasonable password policies, defense in depth, business transaction limits, solid network architecture, and fraud and audit controls

Simplicity

- Attack surface area and simplicity go hand in hand. Certain software engineering fads prefer overly complex approaches to what would otherwise be relatively straightforward and simple code.
- Example
 - > Although it might be fashionable to have a slew of singleton entity beans running on a separate middleware server, it is more secure and faster to simply use global variables with an appropriate mutex mechanism to protect against race conditions.

Fix Security Issues Correctly

- Once a security issue has been identified, it is important to develop a test for it, and to understand the root cause of the issue
- Example
 - > A user has found that they can see another user's balance by adjusting their cookie. The fix seems to be relatively straightforward, but as the cookie handling code is shared amongst all applications, a change to just one application will trickle through to all other applications. The fix must therefore be tested on all affected applications.

Tools!

Tools

- WebScarab - a web application vulnerability assessment suite including proxy tools
- Validation Filters – (Stinger for J2EE, filters for PHP) generic security boundary filters that developers can use in their own applications
- CodeSpy – look for security issues using reflection in J2EE apps

Tools

- CodeSeeker - an commercial quality application level firewall and Intrusion Detection System that runs on Windows and Linux and supports IIS, Apache and iPlanet web servers,
- WebGoat - an interactive training and benchmarking tool that users can learn about web application security in a safe and legal environment
- WebSphinx – web crawler looking for security issues in web applications
- OWASP Portal - our own Java based portal code designed with security as a prime concern

Web Application Security Threats and Counter Measures

Sang Shin

sang.shin@sun.com

www.javapassion.com

Java Technology Evangelist

Sun Microsystems, Inc.