Secure Programming Support in IDE

Bill Chu, Jing Xie

Department of Software and Information Systems
University of North Carolina at Charlotte

billchu, jxie2@uncc.edu
Overview

- Software vulnerabilities is a major contributor to information security problems
- Education and training is critical
- But even the most experienced developers make mistakes
- Tool support
  - Static and dynamic analysis tools are reactive
  - More effort is needed to proactively support programmers avoid making mistakes in the first place
  - Include developers in the “security loop”
Causes of developer errors

- Don Knuth’s case study on TEX

Knuth documented 867 errors over a period of 10 years. 368 errors were implementation errors, the rest are requirements/design errors. **Mistake of omission** is the largest contributor of implementation errors.

“Here I did not remember to do everything I had intended, when I actually got around to writing a particular part of the code. .. This seems to be one of my favorite mistakes: I often forget the most obvious things”

Knuth, D. The errors of TeX – *Software: Practice and Experience* 19(7) 1989
Causes of developer errors

- Disconnect between conceptual understanding of secure programming and its practice
- Our interviews of professional programmers indicates a clear pattern of programmers having a solid conceptual understanding of security but do not consistently apply them in practice
  - Reliance on framework and/or process
  - Input validation: functional/business vs. security

Our Approach

- Many common software vulnerabilities are caused by the **mistake of omission**, e.g.
  - Failure to perform input validation/output filtering
  - Failure to check security invariants before performing critical actions
    - CSRF
    - Broken Access Control
- **Interactively** identify common secure programming issues using reliable heuristics
- Enable developers to **select appropriate actions** while they are in the process of composing the program
  - Interactive code refactoring
  - Interactive code annotation
- **ASIDE (Application Security in IDE** plugin for Java and Eclipse) prototype
ASIDE Design Rationales

- Recognition instead of recalling, a key HCI design principle
- Take full advantage of developer’s application knowledge (e.g. business logic, application context)
- Support best secure software development practice
  - Using trusted library (e.g. OWASP ESAPI)
  - Statistics collection
- Policy driven (adapted to other development environment)
ASIDE Demo

- Based on Eclipse Java Development Tooling (JDT).
- Two major features:
  - Code refactoring (implemented)
  - Code annotation

Example: interactive code refactoring for input validation

- Identifying untrusted input requiring validation

```java
if (!session_pass.equals(password)) {
    RequestDispatcher dispatcher = request
        .getRequestDispatcher("/login");
    dispatcher.forward(request, response);
} else {
    String t_type, quantity, accNickname, stockname;
    t_type = request.getParameter("trans action");
    quantity = request.getParameter("quantity");
    accNickname = request.getParameter("acc nickname");
    stockname = request.getParameter("stockname");
    // Map-String, String> map = request.getParameterMap();
    // t_type = map.get("trans action");
    // quantity = map.get("quantity");
    // accNickname = map.get("acc nickname");
    // stockname = map.get("stockname");
    int q = 0;
    try {
        q = Integer.parseInt(quantity);
    } catch (Exception e) {
        makeTransactionForUser(out, user, t_type, q, accNickname,
                               stockname, "Invalid Quantity");
    }
    return;
```
The return value of `getParameter()` at line 84 is vulnerable to be manipulated by malicious users.

30 quick fixes available:
- `HTTPParameterValue`
- `HTTPPrimary`
- `HTTPHeaderValue`
- `HTTPURL`
- `HTTPQueryString`
- `HTTPSESSIONID`

ABSTRACT

The return value of `getParameter()` at line 84 is vulnerable to be manipulated by malicious users.

EXPLANATION

When untrusted input gets into the validation process, it may be used by an attacker to exploit vulnerabilities in the application. This can lead to data theft, DoS attacks, and other security issues.

REMEDIATION RECOMMENDATION

The best practice to avoid introducing the aforementioned vulnerabilities into your code is to validate all the values that are passed into your application. One option is to validate the input against established regular expressions.
Utilize reputable input validation library, e.g. OWASP ESAPI Validator.
## Code refactoring strategies for input validation

<table>
<thead>
<tr>
<th>Input validation strategy</th>
<th>Advantage(s)</th>
<th>Disadvantage(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Right before critical operations (e.g. inserting into database).</td>
<td>Developer knows for sure the type of input. (e.g. first name, password, credit card number, SSN, and etc.)</td>
<td>Redundant validation: a variable used in multiple places. Failure to validate: difficult to foresee all critical operations.</td>
</tr>
<tr>
<td>As soon as an untrusted input is read into a variable.</td>
<td>Has developer’s attention. Make sure all untrusted inputs are validated.</td>
<td>Can lead to false positive. Does not work well with dependency injection design pattern.</td>
</tr>
</tbody>
</table>
ASIDE implementation

- ASIDE can support either strategy
- We evaluated a version of ASIDE using the second strategy and discuss some of our evaluation results
Additional input validation features

- **Semantic validation**
  - E.g. once the input is identified as file path, further restrict to a particular file subtree
  - Bounds of integers

- For untrusted input of **composite type** (e.g. `getParameterMap()`):
  - Perform flow analysis
  - Request for validation as soon as an primitive type (e.g. `java.lang.String`) of data is extracted
Code Refactoring Evaluation

- Target Project: Apache Roller 3.0.0
  - 65K+ lines of code
  - Full featured blog server (1.8M+ hits on google for “powered by Apache Roller”)
- Comparison base: Fortify SCA based code review

Industry Best Practice Security Audit

- Performed by John Melton, a member of SSG at a large financial service company, core committer of OWASP AppSensor
- Fortify SCA reported 3,416 issues
- John manually audited each issue
- John determined 1,655 issues, as he would have done according to industry best practice
- Software of average quality according to John
- Would take 2.5 days based on standard workload estimate metrics
Details of Manual Audit

- Whether appropriate input validation/encoding has been performed
- Validate Fortify’s environmental assumptions (e.g. for log forging, whether logging mechanism has not been wrapped)
- Validate Fortify’s trust boundary assumptions
- Scrutinize input validation and encoding routines (e.g. black-list filtering)
- Filter out false positives in DOS warnings
## Secure Code Review Results

<table>
<thead>
<tr>
<th></th>
<th>Critical</th>
<th>High</th>
<th>Medium</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortify Vulnerability Categories</td>
<td>8</td>
<td>18</td>
<td>2</td>
<td>52</td>
</tr>
<tr>
<td>Raw Issues</td>
<td>164</td>
<td>653</td>
<td>13</td>
<td>2,597</td>
</tr>
<tr>
<td>Exploitable in Roller 3.0.0</td>
<td>37</td>
<td>397</td>
<td>0</td>
<td>1,221</td>
</tr>
</tbody>
</table>

922 of 1,655 findings are related to lack of proper validation/encoding
Validation /filter of untrusted data

<table>
<thead>
<tr>
<th>Severity</th>
<th>Category Name</th>
<th>Number of Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical</td>
<td>Cross-Site Scripting: Persistent</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Cross-Site Scripting: Reflected</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Path Manipulation</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>SQL Injection</td>
<td>11</td>
</tr>
<tr>
<td>High</td>
<td>Cross-Site Scripting: Persistent</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>Denial of Service</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Header Manipulation</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>Log Forging</td>
<td>252</td>
</tr>
<tr>
<td></td>
<td>Path Manipulation</td>
<td>6</td>
</tr>
<tr>
<td>Low</td>
<td>Cross-Site Scripting: Poor Validation</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Log Forging (debug)</td>
<td>531</td>
</tr>
<tr>
<td></td>
<td>SQL Injection</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Trust Boundary Violation</td>
<td>3</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>922</td>
</tr>
</tbody>
</table>

Table 2. Detail results from security auditing against Roller using Fortify SCA.
Validation/encoding of untrusted data

- 922 Fortify issues caused by 143 taint sources
  - Primitive data type (e.g. `java.lang.String`)
  - Composite data type (e.g. `java.util.Map`)
  - Variables require output encoding always result from untrusted data
- ASIDE identified 131 of 143 (92%) taint sources
- Taint source of composite data type is 41
- 12 issues not detected by ASIDE
  - JSP (not yet implemented in prototype)
  - Framework binding
    - Delayed binding (implementing the Dependency Injection design pattern)
False Positives of ASIDE

- ASIDE reported 118 more taint sources of primitive data types
  - Potentially exploitable (94), validate to practice defensive security
  - False positive (24)
Defensive Security

- A taint request URL is directly passed into an InvalidRequestException constructor.

```java
    } else {
        throw new InvalidRequestException("bad path info, "+
                                            request.getRequestURL());
    }
```
False positives

We regard 24 reported taint sources as real false positives, where inputs are used in ways that do not lead to any recognized security vulnerabilities.

```
    // are we doing a preview? or a post?
    String method = request.getParameter("method");
    boolean preview = (method != null && method.equals("preview")) ? true : false;
```

Figure 5. Untrusted input is used for logic test

```
    if(request.getParameter("excerpts") != null) {
        this.excerpts = Boolean.valueOf(request.getParameter("excerpts")).booleanValue();
    }
```

Figure 6. Untrusted input is parsed into harmless Boolean value
Summary of benefits of code refactoring

- Address Validation/Encoding issues at the time of development
  - Requires no specialized training
  - Capturing application context
  - Saving time to fix vulnerabilities that might be found later in security code audit
  - Saving efforts in fixing vulnerabilities (e.g. 143 taint sources vs. 922 issues)

- Save security code audit efforts
  - Significant reduction of workload (e.g. 922 out of 3,416)
Practical implications for using ASIDE

- Compliment to static analysis
  - e.g. Generating “cleansing rules” after validation/encoding to reduce number of issues raised

- A “light version of static analysis”
  - e.g. handling validation/encoding issues
Interactive Code Annotation

- Remind developers important program constructs for secure coding
  - Prevent vulnerable code from being written

- Annotate key application logic for
  - Source code review
  - Advanced analysis

- Different from traditional code annotation
  - Annotate security relationship between different parts of the system
  - Point and click
Interactive Code Annotation Example

- Database tables
  - `user(username,role,surname,givenName)`
  - `account(accountNumber,nickname,balance)`
  - `transaction(id,accountNumber,date,payee,amount)`
  - `account_user(accountNumber,username)`

- All tables are protected by SSG
ASIDE Detections and Annotations

1. Detect the need of authentication at line 63 in AccountsServlet.java

   - Annotate the required authentication process at line 47 in AccountsServlet.java

DETECTION DETAIL:

ASIDE detects method invocation `<b>AccountMapper.myAccounts(User user)</b>` at line 63 in AccountsServlet.java. The application is accessing sensitive database table ACCOUNT through SQL query:

```
SELECT account.accountNumber, account.nickname, account.balance
FROM account
INNER JOIN account_user
ON account_user.accountNumber = account.accountNumber
WHERE account_user.username = user.getUsername()
```

The ACCOUNT table requires authentication of the entity which tries to access it. In this case, the first parameter `<user>` of method invocation AccountMapper.myAccounts (User user) needs to be authenticated first.

ASIDE recommends to annotate the corresponding authentication process before proceeding.

ANNOTATION DETAIL:

The `<b>if</b>` control flow which is based on test condition of request.getSession().getAttribute("USER") == null starts at line 47 in AccountsServlet.java does the authentication.
Context for annotation

- **Where to raise question?**
  - Identifying database access functions (e.g. SQL statements), may be too low level
  - Access routines may be shared in different application threads
  - Identify “use case”/transaction level routines that lead to accessing protected data
  - E.g. a statement within a Servlet/Action for Java web applications

- **What is a valid annotation?**
  - A set of logic tests, or assertion (e.g. Spring Security)
  - On an execution path from web entry to data access point
Advanced analysis based on annotation

- Unchecked access path
  - There might be an execution path from web entry to data access point without access control check
Missing access control check

- Suppose there are two “use cases” that invoke the same access function
- They have different access control checks
Case Study

- Open source project
  - Apache Roller (Java): blog server software
  - Moodle (PHP): course management system (CMS)
- Statistics (bug track & security reports)

<table>
<thead>
<tr>
<th></th>
<th>Fixed issues with detailed information</th>
<th>Code Refactoring</th>
<th>Code Annotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Roller</td>
<td>6</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Moodle</td>
<td>14</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
3/7 cases are vulnerabilities caused by insufficient input validation.

All these cases can be handled by ASIDE as described above.
Broken Access Control

- `roller.weblogger.webservices.adminprotocol.BasicAuthenticator` is vulnerable to authentication bypass. If invalid headers are passed to it, an invalid user can gain access to protected resources.
Are web headers valid?

If No, Retrieve credentials

If Yes, Are credentials valid?

If No, Throw an Exception

If Yes, Get all users from DB
protected void doGet(HttpServletRequest req, HttpServletResponse res) {
    try {
        Handler handler = Handler.getHandler(req);
        String userName = handler.getUserName();
        EntrySet c = handler.processGet();
        res.setStatus(HttpServletResponse.SC_OK);
        res.setContentType("application/xml; charset=utf-8");
        String s = c.toString();
        Writer writer = res.getWriter();
        writer.write(s);
        writer.close();
    } catch (HandlerException he) {
        res.sendError(he.getStatus(), he.getMessage());
        he.printStackTrace(res.getWriter());
        logger.error(he);
    }
}
/**
 * This method should be called by extensions of this class within their
 * implementation of authenticate().
 */

protected void verifyUser(String userName, String password) throws HandlerException {
    User ud = getUserData(userName);
    String realPassword = ud.getPassword();

    boolean encrypted = Boolean.valueOf(WebloggerConfig.getProperty("passwds.encryption.enabled"));
    if (encrypted) {
        password = Utilities.encodePassword(password, WebloggerConfig.getProperty("passwds.encryption.algorith"));
    }

    if (!userName.trim().equals(ud.getUserName())) {
        throw new UnauthorizedException("ERROR: User is not authorized: " + userName);
    }

    if (!password.trim().equals(realPassword)) {
        throw new UnauthorizedException("ERROR: User is not authorized: " + userName);
    }

    if (!ud.hasRole("admin")) {
        throw new UnauthorizedException("ERROR: User must have the admin role to use the RAP endpoint: " + userName);
    }

    if (!ud.setEnabled().booleanValue()) {
        throw new UnauthorizedException("ERROR: User is disabled: " + userName);
    }
}
Cross-site Request Forgery

- 2/7 are CSRF vulnerabilities
- MSA-08-0013 & MSA-09-0008
- Moodle has developed a pattern to prevent CSRF
- But it was missed in at least these two cases by developers
Change an existing user’s profile

Client form submission

Update database content

```php
// If data submitted, then process and store.
if ($usernew = data_submitted()) {
    if (($USER->id <> $usernew->id) && !isadmin()) {
        print_error('onlyeditown');
    }

    $userold = get_record('user', 'id', $usernew->id);
    if (update_record("user", $usernew)) {
        if (function_exists("auth_user_update")) {
            // pass a true $userold here
            if (auth_user_update($userold, $usernew)) {
                // auth update failed, rollback for moodle
                update_record("user", $userold);
                error('Failed to update user data on external auth: ".$user->auth.
                    [ See the server logs for more details. ]');
            }
        }
    }
}
```
ASIDE solution

- **Heuristic:** Whenever a form submission/web request contains operation to update (add, delete, modify) database entries, the form submission needs to be checked for CSRF.

- Raise question at **Line 72**
Change an existing user’s profile

Client form submission

Update(delete) database content

```php
if (1 $post->parent) { // post is a discussion topic as well, so delete discussion
    if ($forum->type == 'single') {
        notice("Sorry, but you are not allowed to delete that discussion!",
            forum_go_back_to("discuss.php?d=$post->discussion");
    }
    forum_delete_discussion($discussion, false, $course, $cm, $forum);
    add_to_log($discussion->course, "forum", "delete discussion",
        "view.php?id=$cm->id", "$forum->id", $cm->id);
    redirect("view.php?f=$discussion->forum");
}
else if (forum_delete_post($post, has_capability('mod/forum:deleteanypost', $modcontext),
    $course, $cm, $forum)) {
    $replycount = forum_count_replies($post);
    if (!empty($confirm)) { // User has confirmed the delete
        if ($post->totalscore) {
            notice(get_string("couldnotdeleteratings", "forum"),
                "view.php?id=$post->id"));
        }
    }
```
## Applicability to secure coding errors

<table>
<thead>
<tr>
<th>Secure programming practices</th>
<th>CWE/SANS top 25 Dangerous Programming Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interactive code refactoring</td>
<td>XSS(1), SQL injection (2), Untrusted input (6), Path traversal (7), Dangerous file types (8), OS command injection (9), Improper control of file name (14), URL redirection (23).</td>
</tr>
<tr>
<td>Interactive code annotation</td>
<td>Buffer copy without checking size of input (3), CSRF (4), Improper access control (5), Buffer Access with incorrect length value (12), Missing authentication (19), Download code without integrity check (20), Incorrect permission assigned for critical resource (21), Race condition (25).</td>
</tr>
</tbody>
</table>
# Summary of benefits

<table>
<thead>
<tr>
<th>technique</th>
<th>Code refactoring</th>
<th>Code annotation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>audience</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Students</td>
<td>Shape awareness, reminder of secure coding best practices, aid in grading</td>
<td>Shape awareness, reminder of secure coding best practices, aid in grading</td>
</tr>
<tr>
<td>Professional developers</td>
<td>Reminder of secure coding best practices, take care of “grunt work”</td>
<td>Reminder of secure coding best practice, advanced analysis</td>
</tr>
<tr>
<td>Enterprise</td>
<td>Encourage secure coding, policies, practice and standards, collect SSDLC statistics</td>
<td>Collect SSDLC statistics, aid in code review</td>
</tr>
</tbody>
</table>
What do programmers think about Security?

- Conducted two user studies
  - 15 graduate students in web application development class
  - 10 professional Java developers

- Works well for students
  - All used ASIDE functions even though it is not required
  - Most of them felt it was very helpful

- Mixed reaction from developers
  - Developers are much more focused on functions, they are used to have warnings not being addressed
  - They need more contextual explanation before they accept code generated by ASIDE
  - Security savvy developers tend to reject the necessity of secure programming for code that does not impose immediate vulnerability threat
Conclusions

- Introduced two mechanisms to support secure programming in IDE (interactive code refactoring and annotation)
- ASIDE’s approach can be an effective addition to best practice SDLC
  - Preventing vulnerable code
  - Improve efficiency of static analysis
- ASIDE appears to be effective as an education tool in universities (NSF funded project to study the effect of ASIDE in CS1, CS2, and Web programming courses at three universities)
- Improvements are needed to make it usable by professional developers
Future work

- UI design, especially for Annotation
- Support Web frameworks (Struts I and II, Spring MVC, etc.)
- Make ASIDE appeal to professional developers
- Study the effect of ASIDE in university curriculum
Thank You

- Acknowledgement
  - National Science Foundation funding
  - Fortify education license

- Your input
  - https://www.owasp.org/index.php/OWASP_ASIDE_Project#tab=Main