Software Development Life Cycle & Vulnerability Signatures

Rebecca Ramnauth
PII CHPT 19: Finding Vulnerabilities in Source Code
CS 678 Data Security, Professor Chung
4 December 2017

Abstract. This paper discusses aspects of a system's development life cycle (SDLC) that contributes to potential source code vulnerabilities. An analysis of code review approaches lends to the discussion of code review as a useful complement to the more traditional black-box testing. This paper focuses particularly on how code review, relative to the SDLC, improves traditional methods of testing and reveals defects that are challenging to discover when dealing with an application from the outside.

Objectives. (1) Read and understand chapter 19 of *The Web Application Hacker's Handbook*; (2) write a comprehensive summary and analysis of the source code vulnerabilities discussed; (3) review previous chapters of *The Web Application Hacker's Handbook* to better understand how and from whence these source code vulnerabilities arise; (4) go beyond book-provided content to suggest an improved development process (i.e., SDLC) that produces more secure applications.

Contents.

Introduction

There is no one way to find vulnerabilities in application source code. In several circumstances, a source code audit may prove of assistance in attacking a target web application. The preference for code auditing arises from the inherent vulnerabilities of modern applications. For example, many applications tend to implement an open source philosophy, thus granting an attacker complete visibility to its underlying processes. Likewise, overlooked security concerns may permit attackers to discover file disclosure vulnerabilities, client-side code integration may cause undesignated privilege allocations, and other vulnerabilities ranging from session management to security misconfigurations are susceptible to exploitation if clear and comprehensive design and development protocols are not in place. These issues, I propose, originate in the development of the system.

The systems development life cycle (SDLC) describes the process of planning, creating, testing, and deploying an application system. Organizations that incorporate security in the SDLC benefit from applications that are *secure by design*. In contrast, organizations that fail to involve information security in the application lifecycle are eventually inhibited by costly or service-disruptive events.

Software Development Life Cycle

The SDLC is well-documented guideline by which a business functions and individuals responsible can optimally carry out the steps of application development. The standard SDLC model contains the following main functions[1]:

1. *Conceptual definition*, a basic description of the new program
2. *Functional requirements*, a list of specifications and requirements from the perspective of business operations
3. *Technical requirements*, a detailed description of the technical requirements and specifications in technical terms
4. *Design*, a formal design and documented details of the product/program
5. *Programming*, the actual development of the application in compliance with the requirements and specifications of function, technology, and design
6. *Testing*, the formal test phase of the application
7. *Implementation*, the application is installed in production

As illustrated, those involved in the development process require multidisciplinary information, including security information. With this, the following is a brief outline of the type of information that should be required during each phase of the SDLC.

1. *Conceptual definition* should require the organization's information, security principles, and strategies
2. *Functional requirements* should involve information security requirements
3. *Technical requirements* should involve information security requirements as well
4. *Design* should require an understanding of the enterprise security architecture and security product standards as outlined by both the organization and its regulatory agencies (e.g., ANSI, NERC, OSHA)

[1] The *maintenance* phase is omitted in this discussion because it is an iteration of the life cycle: where and when a change is needed, the phases listed are repeated. Therefore, the security activities are also re-evaluated or iterated.

5. *Testing* should demonstrate or verify the achievement of each security requirement
6. *Implementation* should implement procedures for authentication, access controls, encryption, backup, and other measures to ensure the confidentiality, integrity, and availability of resources.

With this security-intensive development process, organizations that involve information security will avoid having to experience costly and disruptive events. These events include[2] orphaned user accounts (due to the application's non-integration with the organization's identity management), defaced websites (a result of applications not built in accordance to security standards), and fraudulent transactions (typically due to inadequate audit trails). These issues are costly and time consuming. The effort required to build secure applications is possible and made time- and cost-efficient through code review approaches.

Approaches to Code Review

There are numerous approaches by which to carry out a code review whilst optimizing security-flaw discoveries in realistic budget and time windows. In this research, selected test approaches are summarized in purpose and design, and are compared with other approaches to leverage the strengths of each.

Black-Box Versus White-Box Testing

There is no debate that testing application security is essential to organizations, but the question of which approach is better persists. The answer is that there is no silver bullet; an enterprise must have multiple approaches to an application from different angles in order to have even an inkling of confidence that an application is secure. The two approaches of black-box and white-box testing will be compared to demonstrate that a hybrid model is best. The *black-box* approach to testing involves attacking an application from the outside and monitoring its inputs and outputs, with no knowledge of its inner workings. On the other hand, the *white-box* approach proves to be a highly effective way to discover vulnerabilities within an application. With access to an application's source code, an attacker[3] can quickly detect vulnerabilities that would otherwise be difficult and time-consuming to locate.

White-box testing, also known as structural testing, is commonly used early in the development process (e.g., phase 5 of the SDLC) because it can be applied effectively while application code/modules are under development. The function of white-box analysis can be compared to that of a spell-checker; developers can install white-box testing into their IDE and easily identify semantic errors before the code is compiled or published. Nonetheless, white-box testing is not a sufficient replacement for a security editor due to the fact application vulnerabilities are not purely semantic. For example, a developer may argue that there nothing inherently wrong in deploying a default-unmasked field for a user's credit account number; however, the design choice is in violation of the Payment Card Industry Data Security Standard (PCI DSS). In accordance to our discussion of the role SDLC plays in security assurance, coding errors are not the only source of application vulnerabilities; business rules and regulatory compliance requirements must be accounted for when assessing an application's security posture. A

[2] I speak according to my personal experience in corporate-level application developments

[3] "Attacker" is used in this paper to denote any individual or corporation that aims to discover vulnerabilities in source code; this includes "attackers" in consultancy contexts.

summary of the goals, advantages, and disadvantages of white-box testing with mention of its strengths and weakness as compared to black-box testing is provided as *Appendix A1*.

Black-box testing, or functional testing, examines application functionality without knowing its internal code structure. Therefore, in contrast to white-box testing, black-box is often implemented in the SDLC phases anticipating or during production. The testing method involves knowing the inputs and expected outputs, and evaluating the system in accordance with its handling of both valid and invalid inputs. Its goals, advantages, and disadvantages are listed in *Appendix A2*.

In many situations, black-box and white-box techniques can complement and enhance one another. For example, given a *prima facie* vulnerability, an easy and effective way to establish its existence is to test on the live application (a black-box approach). Then, having verified the vulnerability's existence and gauged the deviation of the application from its expected behavior, find the root cause by reviewing the appropriate source code segment (a white-box approach). As demonstrated by this example, the hybridized approach saves both time and effort, reaps combined benefits of both testing methods, and simplifies the size and complexity of the system's codebase and behavior.

Therefore, the key objective of effective code review is to identify security vulnerabilities whilst using best practices. A generalized threefold approach to auditing a web application is provided below.

1. Trace user-controllable data from its entry point(s) into the application and the review the code responsible for processing that data
2. Search the codebase for *signatures* that may indicate the presence of vulnerabilities. Then, review these instances to verify that such a vulnerability exists.
3. Perform a line-by-line review of apparently risky code to better understand the application's logic. Functional components that are commonly selected during this step include key security mechanisms such as authentication, session management, access control, and application-wide input validation.

Vulnerability Signatures

Many web-based application vulnerabilities have a generally consistent signature by which they may be easily identified. The examples presented in this section appear in several languages, though vulnerability signatures are typically language-neutral. The programming technique employed is emphasized more greatly than the syntax. Most of these examples are provided with minor edits from *The Web Application Hacker's Handbook.*

Cross-Site Scripting

Cross-Site Scripting (XSS) attacks are a form of injection, in which malicious scripts are injected into trusted websites. Development or design flaws that allow XSS attacks to succeed are widespread and occur in applications in which user input with its output are not validated or encoded. An attacker may use XSS to broadcast a malicious script to an unsuspecting user. The end user's browser cannot determine the legitimacy of the script, and will, thus, blindly execute the script. In transparent examples of XSS, parts of the HTML returned to the user by the web application are explicitly constructed from user-controllable data. In the following

demonstration, the target of an `HREF` link is built using user-inputted strings from the query string in the request.

```
String link = "<a href=" + HttpUtility.UrlDecode(Request.QueryString
["refURL"]) + "&SiteID=" + SiteId + "&Path=" + HttpUtility.UrlEncode
(Request.QueryString["Path"]) + "</a>";
objCell.InnerHtml = link;
```

In code vulnerabilities like these, the SDLC should incorporate XSS prevention rules. These rules, not thoroughly discussed in *The Web Application Hacker's Handbook*, are provided as *Appendix B*.

SQL Injection

SQL injection vulnerabilities often arise from hard-coded strings that are concatenated with user-controllable data to generate a SQL query, which is eventually executed in the database. The following is a query constructed using data from the request query string:

```
StringBuilder SqlQuery = newStringBuilder("SELECT name, accno FROM
TblCustomers WHERE " + SqlWhere);
if(Request.QueryString["CID"] != null &&
      Request.QueryString["PageId"] == "2") {
      SqlQuery.Append(" AND CustomerID = ");
      SqlQuery.Append(Request.QueryString["CID"].ToString());
}
…
```

Code review approaches identify this attack by searching the source for hard-coded substrings. Thus, searching includes pattern detections for SQL keywords (e.g., `SELECT`, `FROM`, `WHERE`) and quoted substrings. In summary, the application programmer should verify that these strings are not being concatenated with user-controllable data such as in the example above.

Path Traversal

Path traversal vulnerabilities provide a signature that often involve user-controllable inputs being passed to a filesystem API without input-validation or file-verification. For example, user data appended to a hard-coded directory path may enable an attacker to traverse the directory tree (using the `.../` sequence) and access files in other unauthorized directories:

```
public byte[] GetAttachment(HttpRequest Request) {
      FileStream fsAttachment = new FileStream(SpreadsheetPath +
      HttpUtility.UrlDecode(Request.QueryString["AttachName"]),
      FileMode.Open, FileAccess.Read, FileShare.Read);
      byte[] bAttachment = new byte[fsAttachment.Length];
      fsAttachment.Read(FileContent, 0,
      Convert.ToInt32(fsAttachment.Length,
      CultureInfo.CurrentCulture));
      fsAttachment.Close();
      return bAttachment;
}
```

An application functionality that enables the user to upload, download or access files should be carefully reviewed. The developer should, therefore, have an understanding of how filesystem APIs are being invoked in user-supplied data to ensure that an input cannot result in file-access in an unintended location.

Arbitrary Redirection

It is common for sites to have a mechanism for redirecting users to different pages, either internal or external. This redirection can be performed by the server (e.g., PHP, ASP, etc.) or client (e.g., JavaScript). The following example illustrates how user-supplied data from a query string can dynamically construct a URL by which the user is then redirected:

```
private void handleCancel() {
      httpResponse.Redirect(HttpUtility.UrlDecode(Request.QueryString[
      "refURL"]) + "&SiteCode=" +
      Request.QueryString["SiteCode"].ToString() +
      "&UserId=" + Request.QueryString["UserId"].ToString());
}
```

The problem with arbitrary redirection arises when the application performs a dynamic redirect using a user-controlled value; with this, a malicious attack can redirect to any URL indiscriminately. This, in turn, gives rise to phishing attacks which invades the integrity and legitimacy of the application.

OS Command Injection

Code that interacts or interfaces with external systems typically contain signatures that indicate code-injection flaws. The example provided in *The Web Application Hacker's Handbook* illustrates how the `message` and `address` parameters can be extracted from user-controllable form data and are passed into a call from the UNIX `system` API.

```
void send_mail(const char *message, const char *addr) {
      char sendMailCmd[4096];
      snprintf(sendMailCmd, 4096, "echo '%s' | sendmail %s", message,
      addr);
      system(sendMailCmd);
      return;
}
```

Backdoor Passwords

Backdoor passwords are important for testing and admission conveniences, predominantly in a review of the application's credential validation logic. However, in numerous situations, a malicious user may simply inspect the source code to determine the backdoor password.

```
private UserProfile validateUser(String username, String password) {
      UserProfile up = getUserProfile(username);
      if (checkCredentials(up, password) || "pass".equals(password))
            return up;
      return null;
}
```

Native Software Bugs

A *software bug* is a logical vulnerability that causes an application to crash or produce an invalid output. Most of the following bugs can be attributed to human errors in source code or design.

Buffer Overflow Vulnerabilities

Buffer overflow typically employs one of the unchecked APIs for buffer manipulation (e.g., `strcpy`, `strcat`, `memcpy`, and `sprint f`). The signature for this vulnerability is embedded in the use of APIs and a user-controllable source buffer. Precautionary steps should be taken to verify whether the code explicitly, as opposed to the implicit API command, ensures that the destination buffer is large enough to accommodate any possible data length being copied into it. Unsafe APIs have their own blatant signature. In the following example, a user-controllable string is copied into a fixed-size stack-based buffer without the program validating that the buffer is large enough to accommodate the string size.

```
BOOL CALLBACK CFiles::EnumNameProc(LPTSTR pszName) {
      char strFileName[MAX_PATH];
      strcpy(strFileName, pszName);
...
}
```

One of the many fixes for the above vulnerability is to explicitly truncate the user-controllable string in accordance to the buffer size; this is illustrated below.

```
BOOL CALLBACK CFiles::EnumNameProc(LPTSTR pszName) {
      char strFileName[MAX_PATH];
      strncpy(strFileName, pszName, strlen(pszName));
...
}
```

Integer Vulnerabilities

Integer vulnerabilities are extremely subtle, but several instances can easily be identified by signatures in an application's source code. For example, comparisons between signed and unsigned integers can lead to such vulnerabilities. The following is another solution to the buffer overflow concerns previously mentioned in which a signed integer is compared with an unsigned integer. If the user provides an input with a negative value, the comparison will succeed, and the unchecked `strcpy` will not produce the intended result.

```
BOOL CALLBACK CFiles::EnumNameProc(LPTSTR pszName, int len) {
      char strFileName[MAX_PATH];
      if (len < sizeof(strFileName))
            strcpy(strFileName, pszName);
...
}
```

Format String Vulnerabilities

Format string vulnerabilities are apparent in the uses of `printf` and `FormatMessage` functions where the format string parameter is not hard-coded but is user-controllable. The following examples this.

```
void logAuthenticationAttempt(char* username) {
      char tmp[64];
      snprintf(tmp, 64, "login attempt for: %s\n", username);
      tmp[63] = 0;
      fprintf(g_logFile, tmp);
}
```

Code Commenting

Several application vulnerabilities lie apparently in the source code comments. Developers, typically aware that an operation is unsafe by the various signatures discussed, record the vulnerabilities or reminders to fix the problem later. Furthermore, testing (more obviously so in black-box testing) also does not consider the role of comments for the belief that commented code or annotations do not influence how the program executes. The following is an example of an unsafe comment.

```
char buf[200];          // I hope this buffer is big enough
...
strcpy(buf, userinput);
```

The Web Application Hacker's Handbook documents the following commenting phrases that may indicate or increase an attacker's awareness of code vulnerabilities.

`bug`	`todo`	`inject`
`problem`	`fix`	`xss`
`bad`	`overflow`	`trust`
`hope`	`crash`	

The Java Platform

This section describes how user-supplied input is acquired, ways to safely interact with the user's session, potentially dangerous APIs, and security-relevant configuration options on the Java platform.

Identifying User-Supplied Data

API	*Description*

`getParameter`
`getParameterNames`
`getParameterValues`
	Parameters within the URL query string and the body of a `POST` request are map `String` names to `String` values

`getQueryString`	Return the query string; an alternative to `getParameter`

`getHeader`
`getHeaders`
`getHeaderNames`
	HTTP headers in the request map the `String` names to `String` values

`getRequestURI` `getRequestURL`	Return the URL and query string of the request
`getCookies`	Return an array of `Cookie` objects which contain the name, values, and other details of the cookies
`getRequestedSessionsId`	Returns the session ID value submitted within the request; an alternative to `getCookies` API
`getInputStream` `getReader`	Return representations of the raw request received from the client and can be used to access the information obtained by all other APIs.
`getMethod`	Returns the method of the HTTP request
`getProtocol`	Returns the protocol of the HTTP request
`getServerName`	Returns the value of the HTTP `Host` header
`getRemoteUser` `getUserPrincipal`	Returns the current user's authentication credentials; this may be a means of introducing malicious input into the processing of the application

Session Interaction

For the storing and retrieving of information within the current session, the following APIs allow for per-session mapping of string names to object values.

API	*Description*
`setAttribute` `putValue`	Store data within the current session

`getAttribute`
`getValue`
`getAttributeNames`
`getValueNames`
	Query data stored in the current session

File Access

The primary class for accessing files in Java is `java.io.File`. From the perspective of information security, this class allows calls to its constructor, which may be exploited to gain unauthorized access/privileges to parent directories. As previously described as a path traversal vulnerability, user-controllable data can be passed as the filename parameter without checking for `../`sequences. The following code demonstrates this issue by opening an unauthorized file directory on Windows.

```
String userinput = "..\\boot.ini";
File f = new File("C:\\temp", userinput);
```

Hence, classes, especially for the reading and writing of file contents like those listed below, should be audited thoroughly.

```
java.io.FileInputStream
java.io.FileOutputStream
java.io.FileReader
java.io.FileWriter
```

For example, a path traversal vulnerability using the `File` object:

```
String userinput = "..\\boot.ini";
FileInputStream fis = new FileInputStream("C:\\temp\\" + userinput);
```

Database Access

For the execution of arbitrary strings as a SQL query:

```
java.sql.Connection.createStatement
java.sql.Statement.execute
java.sql.Statement.executeQuery
```

The signature of user-controllable input as part of a query string, raises the concern for SQL injection. For example, this:

```
String username = "admin' or 1=1--";
String password = "foo";
Statement s = connection.createStatement();
s.executeQuery("SELECT * FROM users WHERE username = "' + username +
"' AND password = "' + password + "'");
```

will execute this unintended query:

```
SELECT *
FROM users
WHERE username = 'admin' or 1=1--'
AND password = 'foo'
```

Nonetheless, a “fix” to such a vulnerability is to use prepared statements, or precompiled SQL statements, and secure the value of its parameter placeholders. These APIs provide more robust and safe alternatives to the APIs demonstrated above.

```
java.sql.Connection.prepareStatement
java.sql.PreparedStatement.setString
java.sql.PreparedStatement.setInt
java.sql.PreparedStatement.setBoolean
java.sql.PreparedStatement.setObject
java.sql.PreparedStatement.execute
java.sql.PreparedStatement.executeQuery
…
```

These API, nevertheless, do not ensure complete prevention of SQL injection attacks and their use should be thoroughly validated.

Dynamic Code Execution

The Java language does not independently contain a mechanism for the dynamic evaluation of Java source code. If the application under review dynamically produces Java code, the developer must be able to determine whether any user-controllable data is being used in an insecure way.

OS Command Execution

The following allows for the execution of external operating system commands within the Java application:

```
java.lang.runtime.Runtime.getRuntime
java.lang.runtime.Runtime.exec
```

If the field is even partially user-controllable and, thus, the string parameter is passed to `exec`, the application is vulnerable to arbitrary command execution. In the following example, the Window `calculator` program runs according to the `userInput`:

```
//ENTIRELY USER-CONTROLLED
String userinput = “calculator”;
Runtime.getRuntime.exec(userinput);

//PARTIALLY USER-CONTROLLED
String userinput = “\\..\\system32\\calculator”;
Runtime.getRuntime().exec(“notepad” + userinput);
```

Though a partial implementation may seem to be relatively more secure against arbitrary command execution than an entire implementation, other vulnerabilities may arise. For example, if an application executes the program with a user-controllable parameter as the target URL, the attacker may be able to pass malicious command-line arguments to the process. For example, the attack may cause the program to download a document and save it to an arbitrary location in the filesystem.

URL Redirection

The following can be used to issue an HTTP redirect in Java

```
javax.servlet.http.HttpServletResponse.sendRedirect
javax.servlet.http.HttpServletResponse.setStatus
javax.servlet.http.HttpServletResponse.addHeader
```

If the value of the `sendRedirect` is user-controllable, the application is vulnerable to a phishing attack as well. Therefore, the developer or code auditor should review any uses of the `setStatus` and `addHeader` APIs.

Sockets

The `java.net.Socket` class permits inputs of target host and port details in its constructors. If the parameters passed are user-controllable, partially or entirely, the application may be vulnerable to a network connection to arbitrary hosts.

Java Environment Configurations

The `web.xml` file includes configuration setting for the Java platform and controls how an application built on the platform behaves. The application's authentication, authorization, and security schemas may be declared in the `web.xml`, outside the application code, and should be secured accordingly. In addition to `web.xml`, different application servers may utilize secondary deployment files (e.g., `weblogic.xml`) to contain other security-relevant configurations. The following are few of the many settings that should be thoroughly inspected during a code audit.

Setting	*Description*

`login-config`
`- form-login-page`
`- forms-based`
`- j_security_check`
`- j_username`
`- j_password`
`- Basic Auth`
`- Client-Cert`
	Authentication details can be generally configured within the `login-config` element; subcategories require more specific configurations. For example, in forms-based authentication, the specified form must have the action defined as `j_security_check`, submitting the parameters of `j_username` and `j_password`.

`security-constraint`	If the `login-config` element is defined, application resources can be restricted and protected using the `security-constraint` element.
`session-config`	The session timeout configuration
`error-page`	The application's generalized error handling
`init-param`	Initialization parameters configuration

Database Code Components

Web applications utilize databases for more than passive data storage. Modern database contains complex programming interfaces, enabling business logic (i.e., phase 2 of SDLC) to be implemented within the database tier itself. Developers and code auditors should evaluate the security of all logic implemented in the database code components spanning from stored procedures to user-defined functions. Programming errors in database code may result in any of the security vulnerabilities discussed in this paper. This section will highlight two main area of vulnerabilities: (1) database components containing SQL injection flaws, and (2) user input passed to apparently risky functions.

SQL Injection

As briefly mentioned, prepared statements (e.g., `java.sql.Connection.prepareStatement`) can be used as a safer alternative to dynamic SQL statements to prevent SQL injection attacks; nonetheless, prepared statements do not ensure complete protection and injection flaws may still

exist in user-controllable database queries. The following is an example of a stored procedure that is vulnerable to SQL injection:

```
CREATE PROCEDURE show_current_orders
      (@name varchar(400) = NULL)
AS
DECLARE @sql nvarchar(4000)
SELECT @sql = 'SELECT id_num, searchstring FROM searchorders WHERE ' +
              'searchstring = ''' + @name + ''';
EXEC (@sql)
GO
```

Regardless of whether the application securely passes the user-supplied `name` value into the stored procedure, the procedure itself concatenates this input directly into a dynamic query and is thereby still vulnerable. Hence, any `exec` command should be closely reviewed upon a code audit.

Calls to Risky Functions

Customized code components such as stored procedures can prove to be powerful in the application's purpose and operation. However, if user-supplied data is passed insecurely to a potentially dangerous function, various vulnerabilities may arise depending on the nature and intention of the function. The following example shows a stored procedure that is vulnerable to command injection in its parameters.

```
CREATE import_data (@loadfile varchar(25), @loaddir varchar(25) )
AS
BEGIN
SELECT @cmdstring = "$PATH/firstload " + @loadfile + " " + @loaddir
EXEC @ret = xp_cmdshell @cmdstring
...
...
End
```

The following is a list of functions that are considered risky because they may potentially use user-supplied data in an unsafe way:

- Default stored procedures that allow the execution of commands, registry access, etc.
- Functions that provide access to a filesystem
- User-defined functions that link to external libraries
- Functions that use network access

Tools for Code Browsing

This paper discusses several methodologies for and aspects of performing a code review, particularly in terms of auditing the source code and searching for signatures of vulnerabilities. The carrying out of code review is not a purely human or manual effort; it is preferable to use an intelligent tool to browse the codebase. In many languages, development studios (e.g., Visual Studio, NetBeans, or Eclipse) involve tools that understand code constructs, provide contextual information on APIs and expressions, and ultimately facilitates auditing navigation. The authors of *The Web Application Hacker's Handbook* suggest the Source Insight tool because it supports easy navigation of the application's source tree, a versatile search function, and a preview pane to describe contextual information about a selected expression.

Conclusion

There is no single and straightforward way to unveil the vulnerabilities in application source code. As demonstrated in this paper, a source code audit may prove of assistance. Overlooked security concerns may allow attackers to find file disclosure vulnerabilities, client-side code integration may cause undesignated privilege allocations, and other vulnerabilities ranging from session management to security misconfigurations are susceptible to exploitation if clear design and development protocols are not in place. This, I argue, originates in the programmer's approach to the system's lifecycle (SDLC). With a security-intensive development process, organizations that involve information security will avoid having to experience costly and disruptive events. The effort required to build secure applications is possible and made time- and cost-efficient through code review heuristics of vulnerability signatures.

References

Merkow, Mark S., and Lakshmikanth Raghavan. *Secure and Resilient Software Development*. CRC Press, 2010.

Shema, Mike. *Hacking Web Apps Detecting and Preventing Web Application Security Problems*. Elsevier/Syngress, 2012.

Stuttard, Dafydd, et al. “Source Code Vulnerabilities.” *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*, John Wiley & Sons, 2012, pp. 701–745.

Appendices

A1: The Black-box Approach

Goals.

- Verify expected outputs
- Test diverse range of inputs
- Ensure wide coverage of testing

Advantages.

- *Unbiased tests*; the designer and tester work independently and without conveyance of knowledge to each other
- *Flexibility in programming expertise*: the tester does not need to know how programming techniques to test the reliability and functionality of the application
- *Functional compliance*; black-box testing identifies contradiction and vagueness in the fulfillment of functional specifications (i.e., phase 2 of SDLC).
- *Point-of-view*; testing is performed from the point-of-view of a user, not the designer's
- *Prompt testing*; test cases can be promptly designed and executed after the completion of specifications

Disadvantages.

- *Redundancy*: testing cannot be partitioned by the module as in white-box testing. Thus, tests may be redundant if already performed by the application designer.
- *Development of test cases.* Test cases may be difficult to design without concise specifications (as outlined in phases 2 and 3 of the SDLC)
- *Range.* Black-box's attempt to test every possible input stream is theoretically impossible and time-consuming
- *Results are often overestimated*
- Unlike white-box testing, black-box cannot be used for testing segments of code.

A2: The White-Box Approach

Goals.

- Verify expected outputs
- Validate sections/modules of code
- Ensure wide coverage of testing
- Verify incomplete paths or statements

Advantages.

- *Thoroughness,* the philosophy of white-box testing to provide complete code coverage. As opposed to black-box testing, the thorough nature of white-box testing provides a clear structure (i.e., engineering-based, rule-defined, and concise) to testing.
- *Automation capabilities;* knowledge of the inner working of an application allow for unit testing which, in turn, simplify the programmatic need to run tests on small pieces or units of code.

- *Time*. White-box testing speeds up the testing process by eliminating the communication cost between developers and QA
- *Optimization*; unit testing emphasizes superfluous code by suggesting to developers areas that need to be condensed.
- *Introspection*. White-box testing force developers to consider the purpose of the individual section of code and its dependencies with other sections, allowing developers to reevaluate and improve designs (i.e., phase 4 of SDLC).

Disadvantages.

- *Expensive*; due to its thoroughness, white-box testing proves to be expensive in both time and cost to conduct. Compared to black-box testing, white-box requires knowledgeable and skilled programmers, thereby, increasing the business cost for development (i.e., phase 5 of SDLC).
- *Evolving code base* degrades the importance of automated test cases in the white-box approach. Redesigns prompted by testing may result in side-effects of useless and incomplete code.
- *Missed cases.* White-box testing validates features that are currently built; if a feature is partially implemented or a requirement not yet fulfilled, white-box testing cannot flag missed cases.

B: Basic Rules for XSS-Prevention

Rule 1: Default-deny unless specified; do not insert untrusted data into an HTML document unless it is specifically allowed. The need for this rule is that escaping and exception are vast and diverse that a default-deny environment can ensure the integrity of data in user-controllable fields.

```
<script> … TRUSTED DATA ONLY … </script>        //in a script
<!-- … TRUSTED DATA ONLY … -->                   //as a comment
<div … TRUSTED DATA ONLY … -->                   //in an attribute
<… TRUSTED DATA ONLY … href=”/test”/>            //in tag name
```

Rule 2: HTML escape before inserting untrusted data into an HTML element content. This rule is for situations in which an application develop would want to put untrusted data directly into the HTML body (e.g., `div`, `p`, `td`, etc. tags). In escaping characters, the developer is avoiding having to take precautionary measures before program execution. The six primary characters documented help to end an HTML entity. This rule should be extended to HTML attribute, JavaScript, and CSS escapes.

```
<!-- HTML CHARACTER ESCAPES -->
&amp;       // ‘ & ’    ampersand character
&lt;        // ‘ < ’    less than character
&gt;        // ‘ > ’    greater than character
&quot;      // ‘ ” ’    quotation – double quotes character
&#x27;      // ‘ ‘ ’    quotation – single quote character
&#x2F;      // ‘ / ’    back slash character

<!-- JAVASCRIPT ESCAPES [EXAMPLES] -->
<script> alert(‘… ESCAPED DATA HERE…’)</script> //inside quoted string
<script> x = ‘… ESCAPED DATA HERE…’</script>    //in quoted expression
```

```
<div onclick = "x='… ESCAPED DATA HERE …'"      //in quoted event

<!-- JAVASCRIPT EXCEPTION TO ESCAPING [EXAMPLE] -->
<script> window.setInterval('… ESCAPED DATA HERE IS STILL XSS-
VULNERABLE…'); </script>

<!—CSS ESCAPES [EXAMPLES] -->
<style> selector{property:…ESCAPE DATA HERE…;}</style>
<span style= "property:…ESCAPE DATA HERE…> lorem ipsum </span>
```

Rule 3: URL escape before inserting untrusted data into HTML URL parameter values.

```
<a href= "http://www.example.com?test=…ESCAPE DATA HERE…"> hyper <a/>
```

Rule 4: Sanitize HTML Markup with library; if an application handles markup (i.e., untrusted input that is intended to contain HTML), validation and encoding may prove to be difficult. Libraries can be used to parse and *sanitize* HTML text. An example sanitizing program is provided at github.com/rramnauth2220/HtmlSanitizer

Additional Rules. This list consists of only a few introductory rules by which a developer should abide by to ensure XSS prevention; this list does not touch upon topics such as using a response header or HTTPOnly cookie flagging.