Chapter 6: Software Security:

Software security is an idea implemented to protect software against malicious attack and other hacker risks so that the software continues to function correctly under such potential risks. Security is necessary to provide integrity, authentication and availability.

Any compromise to integrity, authentication and availability makes a software unsecure. Software systems can be attacked to steal information, monitor content, introduce vulnerabilities and damage the behavior of software. Malware can cause DoS (denial of service) or crash the system itself.

Basic Attacks:

Buffer overflow, stack overflow, command injection and SQL injections are the most common attacks on the software.
Buffer and stack overflow attacks overwrite the contents of the heap or stack respectively by writing extra bytes.
Command injection can be achieved on the software code when system commands are used predominantly. New system commands are appended to existing commands by the malicious attack. Sometimes system command may stop services and cause DoS.
SQL injections use malicious SQL code to retrieve or modify important information from database servers. SQL injections can be used to bypass login credentials. Sometimes SQL injections fetch important information from a database or delete all important data from a database.
The only way to avoid such attacks is to practice good programming techniques. System-level security can be provided using better firewalls. Using intrusion detection and prevention can also aid in stopping attackers from easy access to the system.

Cross-site Scripting (XSS)

Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted web sites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it.

An attacker can use XSS to send a malicious script to an unsuspecting user. The end user'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 the browser and used with that site. These scripts can even rewrite the content of the HTML page

Cross-Site Scripting (XSS) attacks occur when:

1. Data enters a Web application through an untrusted source, most frequently a web request.
2. The data is included in dynamic content that is sent to a web user without being validated for malicious content.

The malicious content sent to the web browser often takes the form of a segment of JavaScript, but may also include HTML, Flash, or any other type of code that the browser may execute. The variety

of attacks based on XSS is almost limitless, but they commonly include transmitting private data, like cookies or other session information, to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site.

Stored and Reflected XSS Attacks

XSS attacks can generally be categorized into two categories: stored and reflected..

Stored XSS Attacks

Stored attacks are those where the injected script is permanently stored on the target servers, such as in a database, in a message forum, visitor log, comment field, etc. The victim then retrieves the malicious script from the server when it requests the stored information. Stored XSS is also sometimes referred to as Persistent or Type-I XSS.

Reflected XSS Attacks

Reflected attacks are those where the injected script is reflected off the web server, such as in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request. Reflected attacks are delivered to victims via another route, such as in an e-mail message, or on some other web site. When a user is tricked into clicking on a malicious link, submitting a specially crafted form, or even just browsing to a malicious site, the injected code travels to the vulnerable web site, which reflects the attack back to the user’s browser. The browser then executes the code because it came from a "trusted" server. Reflected XSS is also sometimes referred to as Non-Persistent or Type-II XSS.

Dos and Don'ts of client authentication:

Client authentication is a common requirement for modern Web sites as more and more personalized and access-controlled services move online. Unfortunately, many sites use authentication schemes that are extremely weak and vulnerable to attack. These problems are most often due to careless use of authenticators stored on the client. We observed this in an informal survey of authentication mechanisms used by various popular Web sites. Of the twenty-seven sites we investigated, we weakened the client authentication of two systems, gained unauthorized access on eight, and extracted the secret key used to mint authenticators from one.
This is perhaps surprising given the existing client authentication mechanisms within HTTP and SSL/TLS , two well-studied mechanisms for providing authentication secure against a range of adversaries. However, there are many reasons that these mechanisms are not suitable for use on the Web at large. Lack of a central infrastructure such as a public-key infrastructure or a uniform Kerberos contributes to the proliferation of weak schemes. We also found that many Web sites would design their own authentication mechanism to provide a better user experience. Unfortunately, designers and implementers often do not have a background in security and, as a result, do not have a good understanding of the tools at their disposal. Because of this lack of control over user interfaces and unavailability of a client authentication infrastructure, Web sites continue to reinvent weak home-brew client authentication schemes.
Clients want to ensure that only authorized people can access and modify personal information that they share with Web sites. Similarly, Web sites want to ensure that only authorized users have access to the services and content it provides. Client authentication addresses the needs of both parties.

Client authentication involves proving the identity of a *client* (or user) to a *server* on the Web. We

will use the term ``authentication'' to refer to this problem. Server authentication, the task of authenticating the server to the client, is also important but is not the focus this paper.

Practical limitations

Deployability:

- Technology must be widely deployed
- HTTP is stateless and sessionles
- Client must provide authentication token
- Useful but high overhead
- Javascript, Flash, Schockwave…

User Acceptability: Web sites must also consider user acceptability. Because sites want to attract many users, the client authentication must be as non-confrontational as possible. Users will be discouraged by schemes requiring work such as installing a plug-in or clicking away dialog boxes.

Performances: Stronger security protocols generally cost more in performance. Service providers naturally want to respond to as many requests as possible. Cryptographic solutions will usually degrade server performance. Authentication should not needlessly consume valuable server resources such as memory and clock cycles. With current technology, SSL becomes unattractive because of the computational cost of its initial handshaking.

Hints for Web client authentication

- Use Cryptography Appropriately
- Protect Passwords
- Handle Authenticators Carefully
- Use cryptography appropriately
- Appropriate amount of security

Keep It Simple, Stupid

- Do not be inventive

Designers should be security experts

- Do not rely on the secrecy of a protocol

Vulnerable to exposure

- Understand the properties of cryptographic tools

  Example: Crypt()

- Do not compose security schemes

Hard to foresee the effects

Protect Passwords

Limit exposure

Don’t send it back to the user (much less in the clear)

Authenticate using SSL vs. HTTP

Prohibit guessable passwords

No dictionary passwords

Reauthenticate before changing passwords

Avoid replay attack

Handle authenticators carefully

Make authenticators unforgeable

highschoolalumni.com

If using keys as session identifier: should be cryptographically random

Protect from tampering (MAC)

Protect authenticators that must be secret

Authenticator as cookie

Sent by SSL

Don't forget the flag! (SprintPCS)

Authenticator as part of URL

Avoid using persistent cookies

Persistent vs. ephemeral cookies

Cookie files on the web

Limit the lifetime of authenticators

Encrypt the timestamp

Secure binding limits the damage from stolen authenticators

Bind authenticators to specific network addresses

Increases the difficulty of a replay attack

SQL INJECTION:

SQL injection is a type of web application security vulnerability in which an attacker is able to submit a database SQL command that is executed by a web application, exposing the back-end database. A SQL injection attack can occur when a web application utilizes user-supplied data without proper validation or encoding as part of a command or query. The specially crafted user data tricks the application into executing unintended commands or changing data. SQL injection allows an attacker to create, read, update, alter or delete data stored in the back-end database. In its most common form, a SQL injection attack gives access to sensitive information such as social security numbers, credit card numbers or other financial data.

Key Concepts of a SQL Injection Attack

- SQL injection is a software vulnerability that occurs when data entered by users is sent to the SQL interpreter as a part of a SQL query.
- Attackers provide specially crafted input data to the SQL interpreter and trick the interpreter to execute unintended commands.
- Attackers utilize this vulnerability by providing specially crafted input data to the SQL interpreter in such a manner that the interpreter is not able to distinguish between the intended commands and the attacker's specially crafted data. The interpreter is tricked into executing unintended commands.
- A SQL injection attack exploits security vulnerabilities at the database layer. By exploiting the SQL injection flaw, attackers can create, read, modify or delete sensitive data.

How SQL Injection works

In order to run malicious SQL queries against a database server, an attacker must first find an input within the web application that is included inside of an SQL query.

In order for an SQL injection attack to take place, the vulnerable website needs to directly include user input within an SQL statement. An attacker can then insert a payload that will be included as part of the SQL query and run against the database server.

Lets look an example code

```
String SQLQuery ="SELECT Username, Password
 FROM users WHERE Username='" + Username +
   "' AND Password='" + Password +"'";

Statement stmt = connection.oreateStaement();
ResultSet rs = stmt.executeQuery(SQLQuery);
while (rs.next()) { … }
```

You will notice that user input is required to run this query. The interpreter will execute the command based on the inputs received for the username and password

```
String SQLQuery ="SELECT Username, Password
 FROM users WHERE Username='" + Username +
   "' AND Password='" + Password +"'";
```

fields.

If an attacker provides ‘or 0=0’ as the username and password, then the query will be constructed as:
String SQLQuery =”SELECT Username, Password FROM users WHERE Username=” or 0=0” AND Password=” or 0=0”;

```
String SQLQuery ="SELECT Username, Password
   FROM users WHERE Username=" or 0=0"
     "' AND Password=" or 0=0" "'";
```

Since under all circumstances, zero will be equal to zero, the query will return all records in the database. In this way, an unauthorized user will be able to view sensitive information.

Preventing SQL Injection

- You can prevent SQL injection if you adopt an input validation technique in which user input is authenticated against a set of defined rules for length, type and syntax and also against business rules.

- You should ensure that users with the permission to access the database have the least privileges. Additionally, do not use system administrator accounts like “sa” for web applications. Also, you should always make sure that a database user is created only for a specific application and this user is not able to access other applications. Another method for preventing SQL injection attacks is to remove all stored procedures that are not in use.
- Use strongly typed parameterized query APIs with placeholder substitution markers, even when calling stored procedures.
- Show care when using stored procedures since they are generally safe from injection. However, be careful as they can be injectable (such as via the use of exec() or concatenating arguments within the stored procedure).

State-Based attacks:

The concept of state, or the ability to remember information as a user travels from page to page within a site, is an important one for Web testers. The Web is stateless in the sense that it does not remember which page a user is viewing or the order in which pages may be viewed. A user is always free to click the Back button or to force a page to reload. Thus, developers of Web applications must take it upon themselves to code state information so they can enforce rules about page access and session management.

The first option is using **forms** and **CGI parameters**, which allow the transfer of small amounts of data and information to be passed from page to page, essentially allowing the developer to bind together pairs of pages. More sophisticated state requirements mean that data needs to be stored, either on the client or the server, and then made available to various pages that have to check these values when they are loaded.

Attack: Hidden Fields

- Something that is part of page, but not shown
  - Defined parameters & assigned values
  - Sent back to server to communicate state
  - Part of a form

Hidden Field Example

```
<html>
<head>
<title>My Page</title>
</head>
<body>
<form name="myform” action="http://www.foo.com/form.php” method="POST">
<div align="center">
<input type="text" size="25" value="Enter your name here!">
<input type="hidden" name="Language” value="English">
<br><br>
</div>
</form>
</body>
</html>
```

Hidden Field Attack

- Look for them (scan source)
- Change their values

Protection - Hidden Fields

- Use misleading names for hidden fields
- Use hashed/encrypted values
- Don’t use them!
- Use them for innocuous input only
- Treat as user input!
- Check their values (content filtering again)

CGI Parameters

- Part of the GET request with URL
- Name-Value pairs
- At end of URL (after ?)
- Pairs separated by &

http://www.foo.com/script.php?user=mike&passwd=guessWho

- Data is sent to server for processing

Parameter Attack

- View source
- Observe URL targets
- In status bar
- In tool tip
- Edit request URL by hand

http://www.foo.com/script.php?user=mike&passwd=bigDummy

- Try changing values
  - record=notMine
  - item=6789
- Adding parameters
  - debug=on
  - debug=true
  - debug=1

Protection - Paramters

- Treat as user input!
- Check values (content filtering again)

GET - POST, Who-What?

- Both send info to server
- GET
  - Per W3C, for idempotent form processing
  - No side-effect (e.g., database search)
- POST
  - For transactions that change state on server
  - Have a side-effect (e.g., database update/insertion)
  - For sending large requests
- Reload/refresh a GET vsa POST form – different behaviors

Cookie Poisoning:How to Off the Cookie Monster

- Cookies revisited
  - Local file to store data
  - Data about you, session, etc...
  - Plain text
- Change information in cookie file
  - Expiration date
  - Authentication data
  - Shopping cart

Protection - Cookies

- Don’t use cookies for any authentication data
- Encrypt critical data- not fool proof
- Treat cookies as user input
- Be careful what you trust!