14

Developing Secure Applications

ORACLE

Objectives

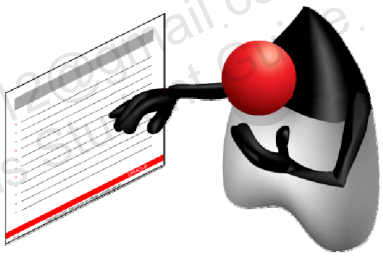
After completing this lesson, you should be able to:

- Describe aspects of security
- Describe fundamental software security concepts
- Avoid common injection and inclusion attacks
- Avoid common confidential data errors
- Limit class accessibility, extensibility, and mutability
- Define immutability
- Create classes that are immutable
- List security resources available on the Internet

ORACLE

Topics

- Security overview
- Secure software overview
- Attacks and best practices
- Security resources



Security Overview

The goal of computer security is to protect information from theft, corruption, or natural disaster while making it accessible for its intended purpose.

- Security is always a systemic solution, which includes:
  - Computer systems
  - Computer software
  - Human systems
- Aspects of security:
  - Confidentiality
  - Integrity
  - Availability

The goal of computer security is to protect information from theft, corruption, or natural disaster while making it accessible for its intended purpose. Security is often a balancing act. For example, the best way to make an application completely secure from the Internet would be to not connect to the Internet at all. However, your application would then have no availability to your users.

All computer systems are made up of many parts. You have operating systems, network systems, software, and finally, people systems. Each of these has an influence on overall security.

To further define security terms, we look at three aspects of security: confidentiality, integrity, and availability.

Confidentiality

Confidentiality is the concealment of information resources.

- Most systems require some degree of secrecy.
- Mechanisms that support confidentiality:
  - Encryption
  - Passwords
  - Access control
    - Limiting resource access to a few people
    - Administrator versus user access

Pretty much all systems today rely on some degree of secrecy. Anything from passwords to your system configuration system configuration needs protecting.

Encryption is the process of converting information into an unreadable code. With a key, information can be encrypted for safe storage, and then decrypted so the information can be read when needed.

Access control is the process of limiting access to important resources. For example, a system administrator is the only user on a system who can reset another user's password.

Integrity

Integrity means that your data is protected from unauthorized change.

- Authentication
  - A user must provide credentials.
  - Only authenticated users have access.
- Mechanisms that support integrity:
  - Prevention mechanisms
    - Blocking unauthorized users
    - Passwords
    - Certificates
  - Detection mechanisms
    - Access logs
    - Analyzing patterns

Integrity means that your data is protected from unauthorized change. This is normally accomplished through authentication. A user must provide some sort of credential (user name and password) to indicate they are who they say they are. This controls who and how changes are made to data.

However, authentication is not enough. Detection systems must be in place in case the authentication systems fails; for example, logging the IP address of a user to identify unusual changes in behavior.

Availability

Availability is the ability to use a system or resource when desired.

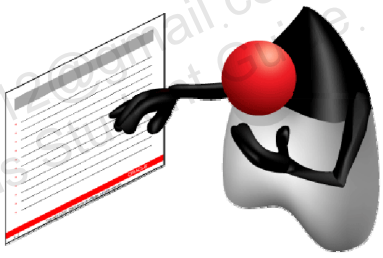
- A system must by accessible by its users.
  - An offline system cannot be used.
- Denial-of-service attacks:
  - Exploit system features to negatively affect performance
  - Deny users access to the system

A practical approach must be used when it comes to availability. If you were to design a truly secure system, you would put it in a bunker with armed guards and provide no network access. However, if network access is a requirement for your application, then there must be some compromise to make the system available to its users. Thus, the mere fact that a system is on a network makes it vulnerable to denial-of-service attacks.

Denial-of-service attacks block access to a system. They prevent users from accessing a system or using it at a normal level of performance. Systems should be flexible enough to be able to detect such attacks and respond to them.

Topics

- Security overview
- Secure software overview
- Attacks and best practices
- Security resources



Developing Secure Software

The focus of this lesson is on developing more secure software.

- Any system can be a target.
  - Theft
  - Sabotage
  - Controlling resources
- Avoiding common mistakes.
  - Be aware of common attacks.
  - Learn from others' experience.

This is an introduction, not comprehensive.

The goal of this lesson is to introduce some of the principles needed to develop more secure software. Any system that contains confidential information is likely to be a target of attackers. Whether the attacker is attempting theft, sabotage, or just wants another zombie for his/her bot net, your software should take into account possible exploitation attempts.

This lesson is an introduction to the most common security vulnerabilities found in software. It is not a substitute for a full security course or book on the subject. Links to recommended resources are provided at the end of the lesson.

Fundamental Security Concepts: Part I

- Prefer obviously no flaws, rather than no obvious flaws.
- Design APIs to avoid security concerns.
- Avoid duplication.
- Restrict privileges.

Prefer obviously no flaws: Design and write code that does not require clever logic to see that it is safe. Avoid overly complex code.

Secure APIs: It is much easier to design code to be secure from the ground up. Trying to retrofit existing code is difficult and error prone.

Example: Making a class final prevents a malicious subclass from adding finalizers, cloning, and overriding random methods.

Duplication: Code that is duplicated tends not to be treated the same way consistently when copied. In addition, this violates the Don't Repeat Yourself (DRY) principle from Agile programming.

Restrict privileges: If the code is operating with reduced privileges, then exploitation of any flaws is likely to be thwarted. For example, leaving applets and JWS jar files unsigned means the application runs in a sandbox and will not be able to execute any potentially dangerous code.

Fundamental Security Concepts: Part II

- Establish trust boundaries.
- Minimize security checks.
- Encapsulate.

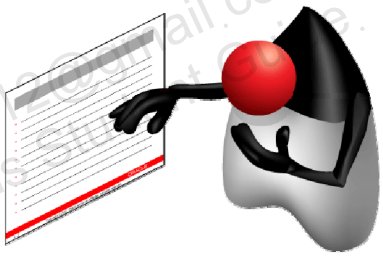
Trust boundaries: Establish boundaries between different parts of your application. For example, any data coming from a web browser to a web application should be sanitized before use.

Security checks: Perform security checks at a few defined points and return an object (a capability) that client code retains so that no further permission checks are required.

Encapsulate: Allocate behaviors and provide succinct interfaces. Fields of objects should be private and accessors avoided. The interface of a method, class, package, and module should form a coherent set of behaviors, and no more.

Topics

- Security overview
- Secure software overview
- Attacks and best practices
- Security resources



Types of Security Threats

There are lots of potential security threats to a program you write. Here are a few categories:

- Injection and inclusion, input validation
- Resource management
  - Buffer overflows
  - Denial of service
- Confidential information
- Accessibility and extensibility
- Mutability

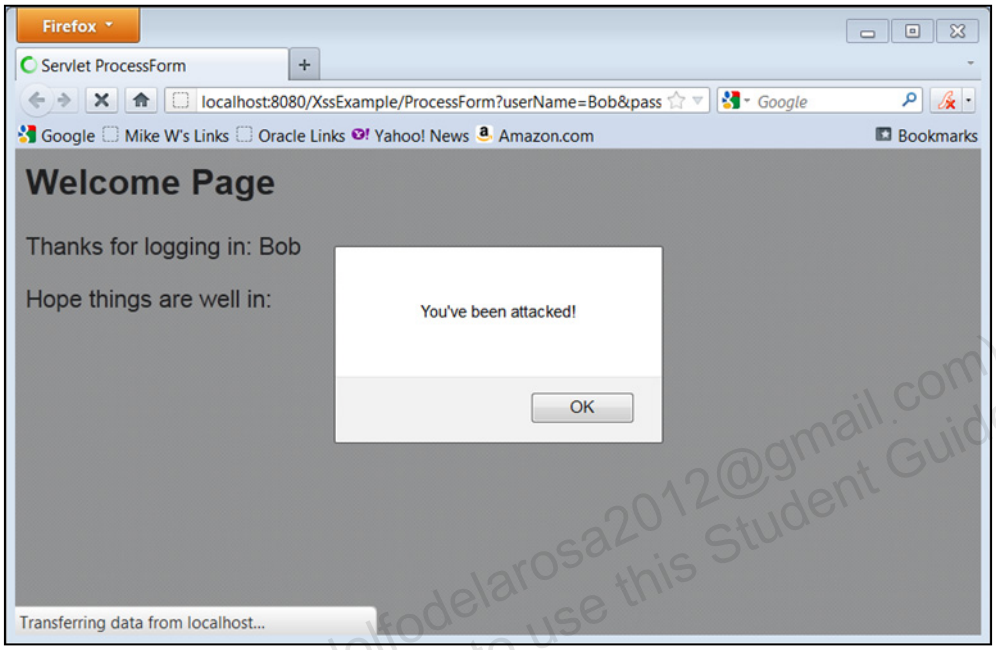
These are the various vulnerability categories as defined by the security personnel at Oracle. This lesson will try to highlight a few examples from each. However, the examples that follow are introductory and not exhaustive.

Injection and Inclusion

- An attack that causes a program to interpret data in an unexpected way and results in an unanticipated change of control
- Any data from an untrusted source must be validated.
  - Unfiltered data can lead to a number of attacks.
- Common forms of attack
  - Cross-site scripting (XSS)
  - SQL Injection
  - OS Command Injection
  - Uncontrolled format string

XSS Example

Inserting a script into form data



The picture shows an example of a servlet with a malicious script inserted into the input text.

Cross-Site Scripting

- Cross-site scripting (XSS) vulnerabilities occur when:
  1. Untrusted data enters a web application
  2. The web application dynamically generates a web page
  3. Exploit is sent to browsers
  4. A victim visits the generated web page
  5. The victim's web browser executes the malicious script
  6. This effectively violates the intention of the browser's same-origin policy
- A very common type of attack

Cross-site scripting (XSS) vulnerabilities occur when:

1. Untrusted data enters a web application, typically from a web request.
2. The web application dynamically generates a web page that contains this untrusted data.
3. During page generation, the application does not prevent the data from containing content that is executable by a web browser, such as JavaScript, HTML tags, HTML attributes, mouse events, Flash, or ActiveX.
4. Using a web browser, a victim visits the generated web page, which contains malicious script that was injected with the untrusted data.
5. Because the script comes from a web page that was sent by the web server, the victim's web browser executes the malicious script in the context of the web server's domain.
6. This effectively violates the intention of the web browser's same-origin policy, which states that scripts in one domain should not be able to access resources or run code in a different domain.

CERT Reference: http://cwe.mitre.org/data/definitions/79.html

SQL Injection Example

Given the following code fragment, could anything go wrong?

```
try {
   Connection con =
DriverManager.getConnection("jdbc:derby://localhost:1527/Simple
DBDemo", "demo", "demo");
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery("SELECT * FROM DEMO.Table1
        WHERE NAME='" + nameParam + "' AND AGE ='" +
        ageParam + "'");

        while (rs.next()) {
            String s = rs.getString("Name");
            float n = rs.getFloat("Age");
            System.out.println(s + "  " + n);
        }
    } catch (SQLException e) {
```


This is a typical piece of database code. Parameters are passed so that a search based on Name and Age can be done.

SQL Injection

An attack that relies on unfiltered data to modify SQL data results.

- The SQL statement is trying to do this:

```
SELECT * FROM DEMO.Table1 WHERE NAME=<nameParam>
AND AGE=<ageParam>
```

- However, what if an attacker submits:

```
someValue' OR 'a' = 'a'
```

- This make the statement always true. The result is:

```
SELECT * FROM DEMO.Table1
```


By modifying the input data, attackers can access confidential information or even modify data stored in the database. Therefore, any input data should be filtered before being used.

CERT reference: http://cwe.mitre.org/data/definitions/89.html

OS Command Injection Example

Given the following code fragment, what could go wrong?

```
try {
    Process p = Runtime.getRuntime()
        .exec("cmd /c dir C:\\Users\\" + userName);

    BufferedReader stdInput = new BufferedReader(
        new InputStreamReader(p.getInputStream()));
    BufferedReader stdError = new BufferedReader(
        new InputStreamReader(p.getErrorStream()));

    // read the output from the command
    System.out.println("Here is your home directory:");
        while ((curLine = stdInput.readLine()) != null) {
            System.out.println(curLine);
        }
```

ORACLE

This piece of code attempts to read the contents of a user's directory based on data passed in by a user. Could anything go wrong with this?

OS Command Injection

An attack the relies on unfiltered data to modify an operating system command

- The example looks harmless enough: Just get a directory listing.
  - `dir C:\Users\username`
- However, an attacker could submit the following:
  - `username;&&del *.*;`
- Which would result in the loss of a lot of data

This is just another example of how a seemingly harmless command could be turned into something very bad. Never trust unvalidated user input.

CERT Ref: http://cwe.mitre.org/data/definitions/78.html

Uncontrolled Format String Example

In this code fragment, the programmer is trying to print the results when two values did not match. What could happen if a format string is submitted instead of the month?

```
static Calendar c =
   new GregorianCalendar(1995, GregorianCalendar.MAY, 23);

 public static void main(String[] args) {
      String param = "%1$tY";
      // Example param is the credit card expiration date
      // If param contains %1$tm, %1$te or %1$tY as malicious
       args
      // Attacker and display value being compared.
      System.out.printf(param
             + " did not match! HINT: It was issued on %1$terd
             of some month\n", c);
 }
```


Even Java format strings can be exploited if code is not written properly and input is not checked.

Uncontrolled Format String

In the example, the idea was to compare two month values.

- Instead of submitting a month value, the attacker submits a format string:
  - `%1$tY`
- The attacker has now figured out the year the card expires from a format string.
- Java displays the output for both format strings on the line.

CERT Ref: https://www.securecoding.cert.org/confluence/display/java/IDS06-J.+Exclude+user+input+from+format+strings

Resource Management

- Buffer overflows
  - Copy an input buffer to an output buffer without checking the size.
  - Result: An attacker can execute arbitrary code outside the normal program.
- Java and buffer overflows
  - Java is immune to these sort of attacks because of its automatic memory management.
- Denial of service
  - Still possible in Java
  - Relate to inappropriate use of resources

Denial-of-Service Examples

Here are some examples of potential denial-of-service attacks:

- Zip bombs
  - Unzipping the file breaks down the system.
- Billion laughs attack
  - XML entity expansion causes the document to grow dramatically during expansion.
- XPath expressions can consume arbitrary amounts of processing time.
- Constructing object graphs

Zip bomb: A zip file that is relatively small but contains many nested zip files within it. Unzipping the files can tie up CPUs and eat up disk space. A relatively small file can contain petabytes worth of data. Limit the size and amount of processing that can be done on a resource like this.

Billion laughs attack: If you want to use the DOM API with an XML document, you must load the entire document into memory. An attack like this can eat up available memory and bring a system to its knees.

XPath: XPath is a language for querying and traversing XML documents. Some queries have a recursive nature that can return a lot more data than expected.

Object Graph: An object graph constructed by parsing a text or binary stream may have memory requirements many times that of the original data.

Confidential Exception Example

Is there anything wrong with this exception?

```
Cannot find main configuration file:
  C:\config\badfile.properties (The system cannot find the path
  specified)
  java.io.FileNotFoundException: C:\config\badfile.properties
    (The system cannot find the path specified)
    at java.io.FileInputStream.open(Native Method)
    at java.io.FileInputStream.<init>(FileInputStream.java:138)
    at java.io.FileInputStream.<init>(FileInputStream.java:97)
    at com.example.sec.FileException.readProps
                        (FileException.java:20)
    at com.example.sec.FileException.main
                        (FileException.java:12)
Java Result: -1
```


Providing an exception like this gives an attacker the exact location of your application's configuration file.

Confidential Log Example

Could this log information be improved?

```
Feb 08, 2012 10:55:14 AM com.example.sec.DetailedLogger
logMessages
SEVERE: ID 123-45-6778 for User John Adams does not
match.
Feb 08, 2012 10:55:14 AM com.example.sec.DetailedLogger
logMessages
SEVERE: User John Adams located at 123 Drury Lane,
Quincy, MA
Feb 08, 2012 10:55:14 AM com.example.sec.DetailedLogger
logMessages
SEVERE: Cannot find exp date for credit card 1111-1111-
1111-1111
```

Confidential Information

- Confidential data should:
  - Be readable only within a limited context
  - Not be exposed to tampering
  - Only provide users with the information that they need
  - Not be hard-coded into source
- Examples
  - Purge sensitive data from exceptions.
  - Do not log highly sensitive information.
  - Consider purging memory when done with sensitive information.

Keeping confidential information private seems like a common sense idea. However, if you do not think about security when creating your code, these sorts of leaks could easily happen.

Confidential data should not be included in exceptions or log files. In addition, you should not hard-code user names and passwords in source code. Instead, a properties file should be used to store this kind of information.

Accessibility and Extensibility

Limit the accessibility of classes.

- Classes and APIs
  - Any class that is part of a public API should be declared public.
  - Any other class should be declared package-private (no modifier in front of class).
- Class fields, methods, and APIs
  - Declare fields and methods public, protected, etc. as appropriate for the API.
  - All other methods and fields should be declared private.
- Packages in `.jar` files should be sealed.

ORACLE

Package-private at the class level means that a class can interact with classes only in its own package. Thus, you can not import a Package-Private class from another package.

A jar file can be sealed by adding:

```
Sealed:true
```

to the manifest file. The heading by itself seals the entire file. You can seal specific packages by using `Sealed` with `Name`:

```
Name: com/example/sec/
Sealed:true
```

By sealing a `.jar` file, only classes contained in that `.jar` may be loaded by the class loader. This prevents naming conflicts when other `.jar` files contain packages with the same name.

Minimize Mutable Classes

- An object is considered immutable if its state cannot change after it is constructed.
  - Think if it as a read-only feature for classes.
  - Once an object is created, it cannot be changed.
  - It is good to combine this with static factory initializers.
- Provides benefits for security and concurrency:
  - Hiding constructors allows more flexibility in instance creation and caching.
  - Making classes `final` prevents classes from being extended.
  - This prevents internal data from being changed outside the class.

Immutability Example

Transforming an Employee class into an immutable employee class.

- Employee01
- Employee02
- Employee03

Steps to Make a Class Immutable

- Set the class to final.
- Use a static initializer.
- Make copies of any mutable objects passed to an initializer.
- Set the fields to final.
- Return copies of any mutable objects.

Quiz

Most security vulnerabilities are a result of:

a. Not validating/filtering input
b. Poor encryption
c. Slow CPUs
d. Buffer overflows

Answer: a

Quiz

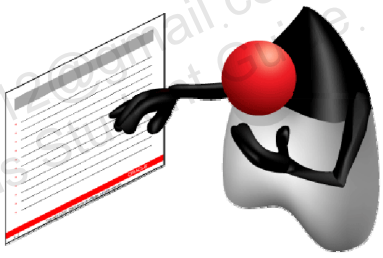
Which is true about a final field that is a mutable object?

a. Once an object is assigned, its values are immutable.
b. Once an object is assigned, its values are mutable.
c. Once an object is assigned, some of its values are mutable.
d. You cannot mark a mutable object field as final.

Answer: b

Topics

- Security overview
- Secure software overview
- Attacks and best practices
- Security resources



Security Resources on the Net

- CERT
  - Computer Emergency Response Team
  - Cert reports
    - https://www.securecoding.cert.org/confluence/display/java/The+CERT+Oracle+Secure+Coding+Standard+for+Java
- Secure Coding Guidelines for Java
  - http://www.oracle.com/technetwork/java/seccodeguide-139067.html#conclusion
- Books
  - Gary McGraw. *Software Security: Building Security In*. Boston: Addison-Wesley, 2006.
  - Joshua Bloch. *Effective Java Programming Language Guide*. 2nd ed. Addison-Wesley Professional, 2008.

The slide lists a few resources that you can use to learn more about security.

Summary

In this lesson, you should have learned how to:

- Describe the aspects of security
- Describe fundamental software security concepts
- Avoid common injection and inclusion attacks
- Avoid common confidential data errors
- Limit class accessibility, extensibility, and mutability
- Define immutability
- Create classes that are immutable
- List security resources available on the Internet

Practice 14: Overview

There are no practices for this lesson.