Secure Coding Standards

Lotfi ben Othmane

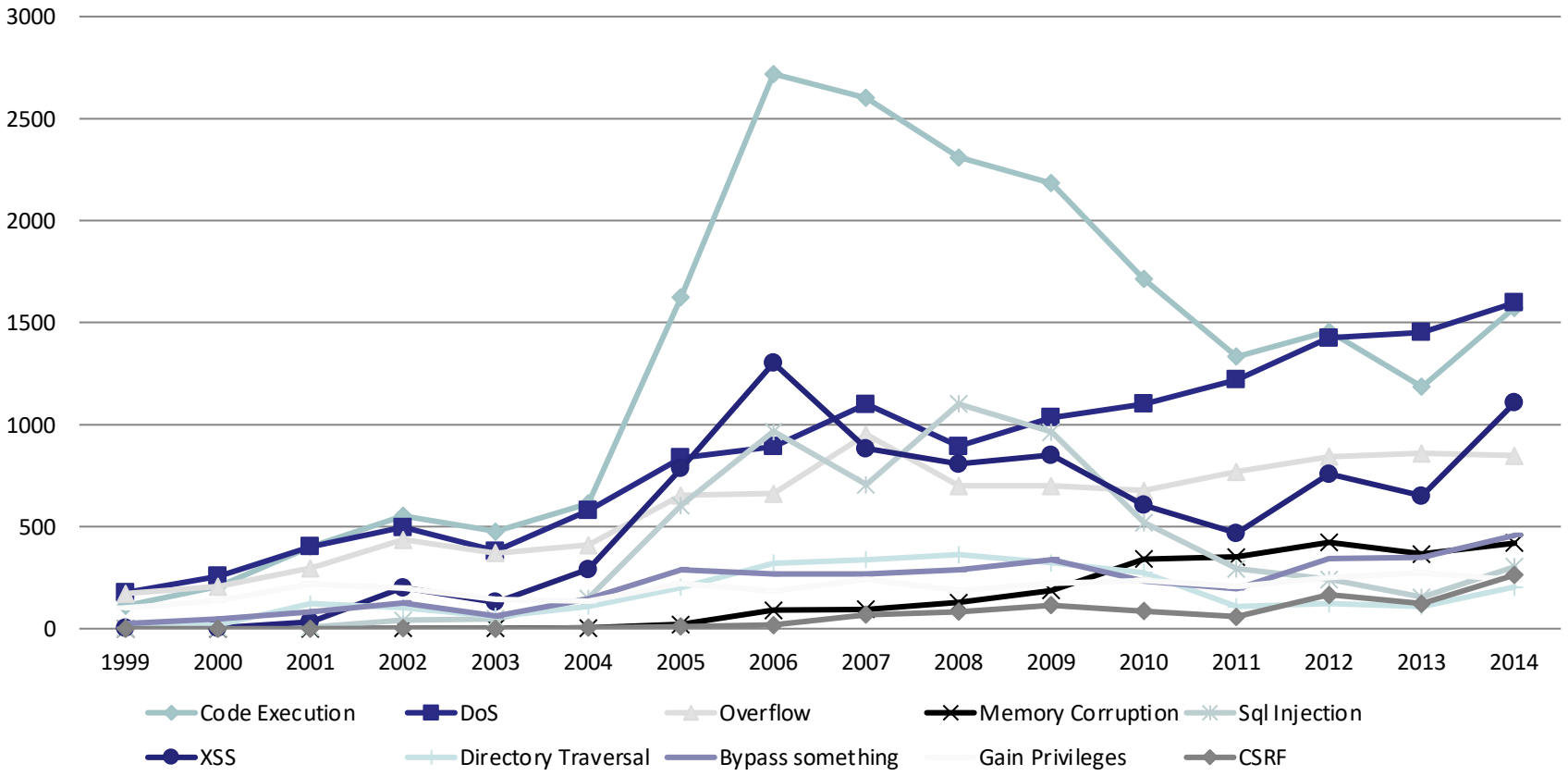
Most of the examples are taken from
https://www.securecoding.cert.org/

What a SDLC is Up To?

- Is SDLC sufficient to producing secure software?

- Does SDLC?
  - Ensure development of secure software?
  - lead to developing secure software?
  - support developing secure software?

Vulnerabilities Distribution



Source: www.cvedetails.com

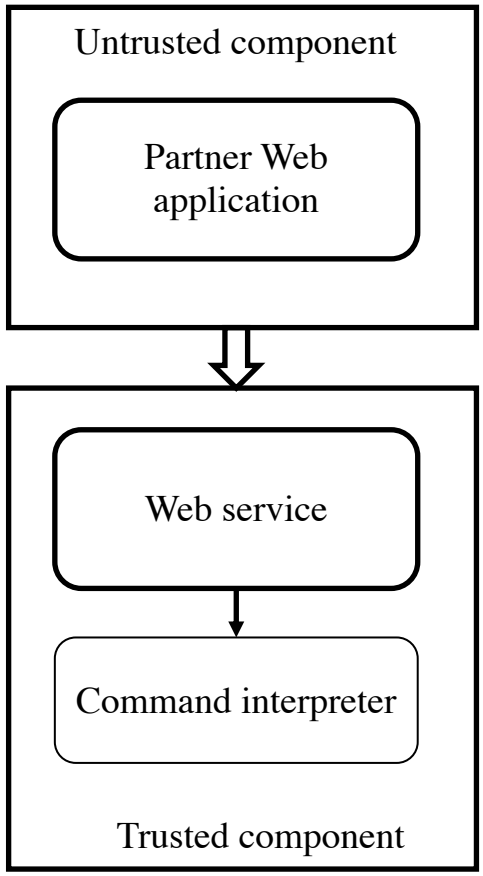
Types of Vulnerabilities

- Design-based vulnerabilities
  - E.g., Broken access control, data leakage due to logic flaw
- Language-based vulnerabilities
  - E.g., Buffer overflow, XSS

Misplaced Trust

- Different components may have different trust levels
- Data coming from untrusted component should be validated before used by a trusted component.

- Example of certificate authority of The Netherlands where the certificate signing system was hosted with a publicly accessible Web application (accessed using non-changed default password)



Secure Programming Overview

- How SQL injection attack works?
- How XSS attack works?
- How buffer overflow attack works?

- How to prevent them?

- What is the relation between programing languages and vulnerabilities?

Injection Attacks

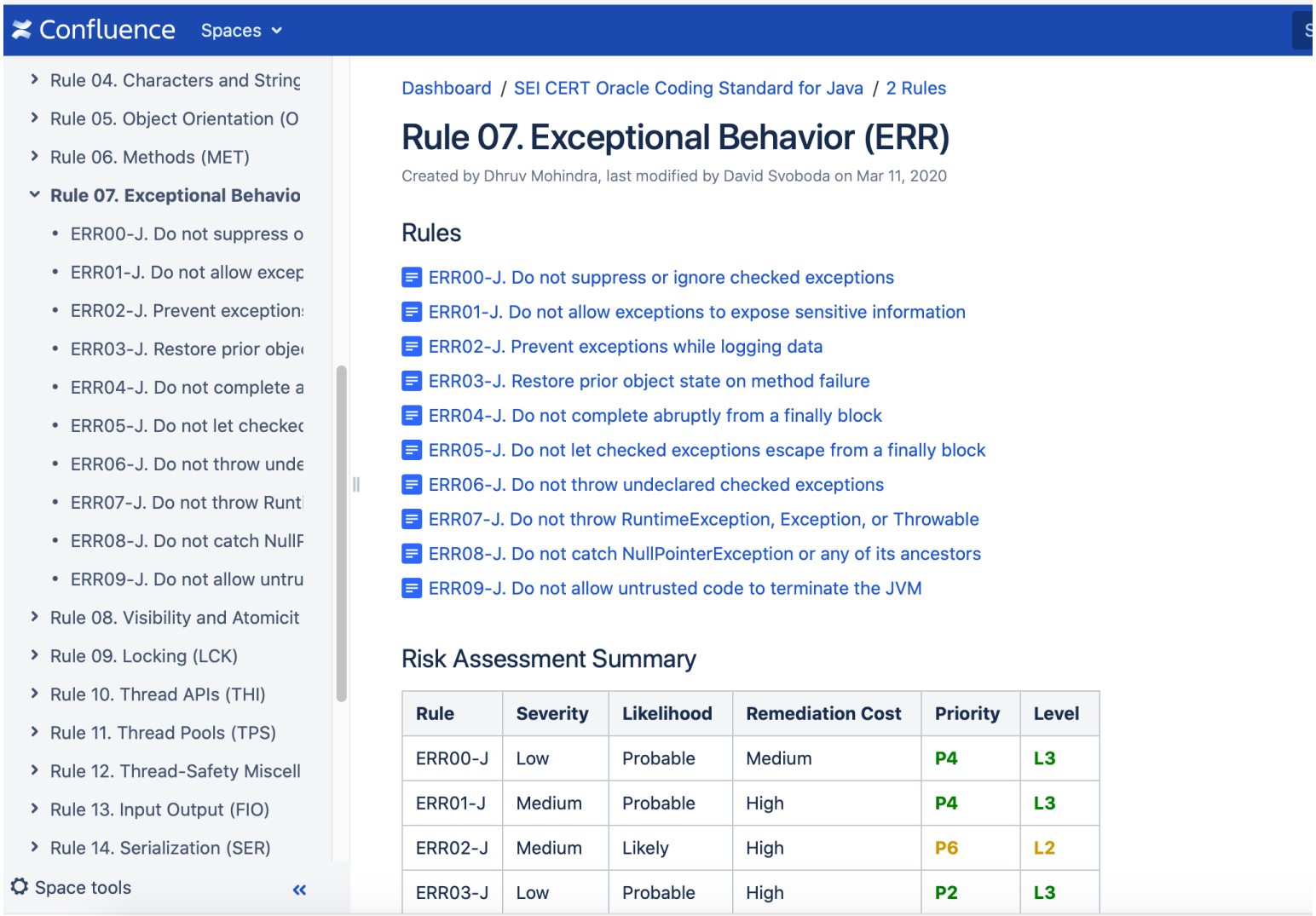
- Attack examples
  - Command injection
  - SQL injection
  - XSS
- Solution
  - ✓ Data validation – data are valid input
  - ✓ Sanitization – ensure conformance of input to a set requirements, by e.g., removing some characters
  - ✓ Canonicalization – reduce the data to minimum, e.g., file path

Secure Programming Overview

- Secure programming (aka defensive coding) refers to coding rules and practices that help avoiding known code vulnerabilities.
- There are several secure coding standards, such as CERT, OWASP, NIST.
- Organizations develop their own APIs to avoid known vulnerabilities. E.g., data validation.

- We give in this lecture some advices on how to avoid a set of well known attack vectors.

Secure Programming Overview



Injection Attacks-2

- SELECT * FROM db_user WHERE username='validuser' OR '1'='1' AND password=<PASSWORD>

- Non-Compliant

```
String sqlString = "SELECT * FROM db_user WHERE username = '"
+ username + "' AND password = '" + pwd + "'";
Statement stmt = connection.createStatement();
ResultSet rs = stmt.executeQuery(sqlString);
```

- Compliant

```
String sqlString = "select * from db_user where username=? and password=?";
PreparedStatement stmt = connection.prepareStatement(sqlString);
stmt.setString(1, username);
stmt.setString(2, pwd);
ResultSet rs = stmt.executeQuery();
```

Memory Mis-Management

```
# include <stdlib .h>
# include <stdio .h>
# include <string .h>

int bof( char *str) {
      char buffer [24];
      strcpy ( buffer , str );
      return 1;
}

int main (int argc , char ** argv ) {
      char str [517];
      FILE * badfile ;
      badfile = fopen (" badfile ", "r");
      fread (str , sizeof ( char ), 517 , badfile );
      bof (str );
      printf (" Returned properly \n");
      return 1;}
```

What is the vulnerability here?
How to avoid it?

Memory Mis-Management-2

Example of solutions

```
# include <stdlib .h>
# include <stdio .h>
# include <string .h>

int bof( char *str) {
        char buffer [24];
        buffer[23] = '\0';
        strncpy ( buffer , str, 23);
        return 1;
}

int main (int argc , char ** argv ) {
        char str [517];
        FILE * badfile ;
        badfile = fopen (" badfile ", "r");
        fread (str , sizeof ( char ), 517 , badfile );
        bof (str );
        printf (" Returned properly \n");
        return 1;}
```

Memory Mis-Management-3

Other rules

https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=437

- MEM30-C. Do not access freed memory
- MEM31-C. Free dynamically allocated memory when no longer needed
- MEM33-C. Allocate and copy structures containing a flexible array member dynamically
- MEM34-C. Only free memory allocated dynamically
- MEM35-C. Allocate sufficient memory for an object
- MEM36-C. Do not modify the alignment of objects by calling realloc()

Data Leakage

How can the data be leaked?

```
public class Credentials implements
Serializable {
  private string password;
  public string password) {
            this. password = password;
  }
}
```

```
public class usepassword extends Credentials {
  public static void main(String[] args) {
    FileOutputStream fout = null;
    try {
      Credentials p = new Credentials("SecretPassword");
      fout = new FileOutputStream("Credentials.ser");
      ObjectOutputStream oout = new
ObjectOutputStream(fout);
      oout.writeObject(p);
    } catch (Throwable t) {
      // Forward to handler
    } finally {
      if (fout != null) {
        try {
          fout.close();
        } catch (IOException x) {
....
  }
```

Data Leakage-2

```
public class Credentials implements
Serializable {
  private String password;

  public Credentials (String  password) {
   PerformSecurityManageCheck();
   this. password = password;
  }
}
```

```
public class usepassword extends credentials {
  public static void main(String[] args) {
    FileOutputStream fout = null;
    try {
      Credentials p = new
Credentials("SecretPassword");
      fout = new
FileOutputStream("Credentials.ser");
      ObjectOutputStream oout = new
ObjectOutputStream(fout);
      oout.writeObject(p);
    } catch (Throwable t) {
      // Forward to handler
    } finally {
      ....
  }
```

Does this solve the problem?

What about making the password field transient?

Denial of Service

- Causes include
  - Upload large files
  - Force infinite loops
  - Initiate many connections
  - Force deadlocks
  - Insert keys with the same hash code
  - Operate on mismanaged sharing of files
  - Force division by zero

Denial of Service

```
static final int BUFFER = 512;
// ...

public final void unzip(String filename) throws java.io.IOException{
  FileInputStream fis = new FileInputStream(filename);
  ZipInputStream zis = new ZipInputStream(new BufferedInputStream(fis));
  ZipEntry entry;
  try {
    while ((entry = zis.getNextEntry()) != null) {
      System.out.println("Extracting: " + entry);
      int count;
      byte data[] = new byte[BUFFER];
      // Write the files to the disk
      FileOutputStream fos = new FileOutputStream(entry.getName());
      BufferedOutputStream dest = new BufferedOutputStream(fos, BUFFER);
      while ((count = zis.read(data, 0, BUFFER)) != -1) {
        dest.write(data, 0, count);
      }
……..
  }
}
```

What is the problem?

Denial of Service

```
static final int BUFFER = 512;
static final int TOOBIG = 0x6400000; // 100MB
// ...

 public final void unzip(String filename) throws java.io.IOException{
 ……
 try {
  while ((entry = zis.getNextEntry()) != null) {
….
    byte data[] = new byte[BUFFER];
    // Write the files to the disk, but only if the file is not insanely big
    if (entry.getSize() > TOOBIG ) {
      throw new IllegalStateException("File to be unzipped is huge.");
    }
    if (entry.getSize() == -1) {
      throw new IllegalStateException("File to be unzipped might be huge.");
    }
    FileOutputStream fos = new FileOutputStream(entry.getName());
    BufferedOutputStream dest = new BufferedOutputStream(fos, BUFFER);
    while ((count = zis.read(data, 0, BUFFER)) != -1) {
     dest.write(data, 0, count);}
  …….
 } finally {
  zis.close();}}
```

Does this solve the problem?

https://www.securecoding.cert.org/

Exception Handling

How difficult it is to read sensitive files?

```
class ExceptionExample {
  public static void main(String[] args) throws FileNotFoundException {
    // Linux stores a user's home directory path in
    // the environment variable $HOME, Windows in %APPDATA%
    FileInputStream fis =
        new FileInputStream(System.getenv("APPDATA") + args[0]);
  }
}
```

Exception Handling-2

```
class ExceptionExample {
  public static void main(String[] args) {

    File file = null;
    try {
      file = new File(System.getenv("APPDATA") +
            args[0]).getCanonicalFile();
    } catch (IOException x) {
      System.out.println("Invalid file");
      return;
    }

    try {
      FileInputStream fis = new FileInputStream(file);
    } catch (FileNotFoundException x) {
      System.out.println("Invalid file");
      return;
    }}}
```

Does this solve the problem?

Exception Handling-3

```
public class Operation {
  public static void doOperation(String some_file) {
    // ... Code to check or set character encoding ...
    try {
      BufferedReader reader =
          new BufferedReader(new FileReader(some_file));
      try {
        // Do operations
      } finally {
        reader.close();
        // ... Other cleanup code ...
      }
    } catch (IOException x) {
      // Forward to handler
    }
  }
}
```

Exception Handling-3

```
public static void doOperation(String some_file) {
  // ... Code to check or set character encoding ...
  try ( // try-with-resources
    BufferedReader reader =
       new BufferedReader(new FileReader(some_file))) {
    // Do operations
  } catch (IOException ex) {
    System.err.println("thrown exception: " + ex.toString());
    Throwable[] suppressed = ex.getSuppressed();
    for (int i = 0; i < suppressed.length; i++) {
      System.err.println("suppressed exception: "
         + suppressed[i].toString());
    }
    // Forward to handler
  }}
```

Java SE 7 introduced a feature called *try-with-resources*

Cryptographic Weaknesses

- Causes include:
  - Use of weak cryptographic algorithms
    - E.g., DES encryption shall not be used anymore
  - Use of weak parameters
    - E.g., RSA encryption key of 1024 is weak
  - Wrong use of APIs
    - Wrong order of API calls
  - Bypass illegitimately errors and warnings

Cryptographic Weaknesses-2

- Weak

```
public static byte[] encrypt(String plainText, String encryptionKey) throws Exception {
    Cipher cipher = Cipher.getInstance(“DES/CBC/NoPadding", "SunJCE");
    SecretKeySpec key = new SecretKeySpec(encryptionKey.getBytes("UTF-8"),
    “DES");
    cipher.init(Cipher.ENCRYPT_MODE, key,new IvParameterSpec(IV.getBytes("UTF-
    8")));
    return cipher.doFinal(plainText.getBytes("UTF-8"));}
```

- Good

```
public static byte[] encrypt(String plainText, String encryptionKey) throws Exception {
    Cipher cipher = Cipher.getInstance(“AES/CBC/PKCS5Padding", "SunJCE");
    SecretKeySpec key = new SecretKeySpec(encryptionKey.getBytes("UTF-8"),
    "AES");
    cipher.init(Cipher.ENCRYPT_MODE, key,new IvParameterSpec(IV.getBytes("UTF-
    8")));
    return cipher.doFinal(plainText.getBytes("UTF-8"));}
```

Unauthenticated Method Calls

- Causes include
  - (illegitimate) Call of methods of frameworks/Libraries used by other co-hosted software
  - (illegitimate) Injection of method calls for libraries that accept dynamically generated code

Mis-configuration

- Causes include
  - Sensitive information, such as database passwords are kept in Web application configuration
  - Providing debugging information
  - Etc.

Conclusions

- Secure programming (aka defensive coding) refers to coding rules and practices that help avoiding known code vulnerabilities.
    - The rules and practices are, in general, technology-dependent
    - Developers shall apply these rules and practices
- Organizations develop APIs that help avoiding vulnerabilities such as malicious input
- Applying the recommended rules and principles does not imply safety from code-vulnerabilities
    - Last year an attacker succeeded in bypassing a rule for executing code embedded in JPEG images on WordPress

Thank you

Any Question?