# CIS 418/518 – Secure Software Engineering
# Software Coding – Errors and Exceptions

Jagadeesh Nandigam

School of Computing
Grand Valley State University
nandigaj@gvsu.edu

## Outline

1. Security Implications of Errors and Exceptions
2. Handling Errors with Return Codes
3. Managing Exceptions
4. Resource Leaks and its Implications
5. Logging and Debugging



## Security Implications of Errors and Exceptions

- Programmers give less thought to error conditions and abnormal/exceptional situations than they do to the expected cases.
- Improper handling of error and exceptional conditions can
  - Cause serious performance problems
  - Have security implications that attackers can exploit.



## Handling Errors with Return Codes

- Using the return value of a function to communicate success or failure comes with a number of undesirable side effects or limitations:
  - It makes it easy to ignore errors – simply ignore a function's return value.
  - Mixing error handling logic with logic for handling expected cases makes programs harder to read.
  - There is no universal convention for communicating error information to the caller.
- C++, Java, and other languages included exceptions as a language feature to address the above limitations.

## Checking Return Values in C

Code that fails/ignores to check the return value of `fgets()`:

```
char buf[10], cp_buf[10];
fgets(buf, 10, stdin);
strcpy(cp_buf, buf);
```

Checking the return value of `fgets()` prevents a potential buffer overflow but makes the code messy:

```
char buf[10], cp_buf[10];
char* ret = fgets(buf, 10, stdin);
if (ret != buf) {
  report_error(errno);
  return;
}
strcpy(cp_buf, buf);
```



## Checking Return Values in C

Checking the return value of `fgets()` prevents a potential buffer overflow and a *forward-reaching goto statement* keeps the main body of the function clean:

```
char buf[10], cp_buf[10];
char* ret = fgets(buf, 10, stdin);
if (ret != buf) { goto ERR; }
strcpy(cp_buf, buf);
...
return;
ERR:
report_error(errno);
... /* cleanup allocated resources */
return;
```



## Checking Return Values in Java

This Java code fails to check the return value of `read()`, making it possible for private data to leak betweek users.

```
FileInputStream fis;
byte[] byteArray = new byte[1024];
for (Iterator i=users.iterator(); i.hasNext();) {
    String userName = (String) i.next();
    String pFileName = PFILE_ROOT + "/" + userName;
    FileInputStream fis = new FileInputStream(pFileName);
    try {
      fis.read(byteArray); // the file is always 1k bytes
      processPFile(userName, byteArray);
    } finally {
      fis.close();
    }
}
```



## Checking Return Values in Java

Checking the return value from `read()` fixes the bug but makes the code more complex.

```
FileInputStream fis;
byte[] byteArray = new byte[1024];
for (Iterator i=users.iterator(); i.hasNext();) {
  String userName = (String) i.next();
  String pFileName = PFILE_ROOT + "/" + userName;
  fis = new FileInputStream(pFileName);
  try {
    int bRead = 0;
    while (bRead < 1024) {
      int rd = fis.read(byteArray, bRead, 1024 - bRead);
      if (rd == -1) {
        throw new IOException("file is unusually small");
      }
      bRead += rd;
    }
  }
  finally {
    fis.close();
  }
  // could add check to see if file is too large here
  processPFile(userName, byteArray) ;
}
```

## Managing Exceptions

- Exception handling constructs in a language allow for separation between code that follows an expected path and code that handles abnormal circumstances.
- Exceptions come in two flavors: *checked* and *unchecked*.
- All exceptions in C++ are unchecked.
- Java supports both checked and unchecked exceptions.



## Managing Exceptions

- Checked Exceptions
  - Caller either handles the exception or declares that it throws the exception as well.
  - Forces the programmer to think about the exception and whether to handle or propagate the exception.
  - Compiler enforces the rules regarding checked exceptions making it hard for programmer to ignore them.
- Unchecked Exceptions
  - Do not have to be declared or handled. The compiler would not complain.
  - The danger is that programmers might be unaware that an exception can occur in a given context and might omit appropriate error handling.



## Safety-Net Exception Handler at the Top Level

- Applications should provide a top-level (catch-all) exception handler to prevent users from seeing a stack trace or other system-debugging information that could be useful in mounting an attack.
- In the following servlet doPost() method, if a DNS lookup failure occurs, the code throws an exception that leaks stack trace and other information to the user of the Web application.

```
protected void doPost (HttpServletRequest req,
                       HttpServletResponse res)
                throws IOException {
    String ip = req.getRemoteAddr();
    InetAddress addr = InetAddress.getByName(ip);
    out.println("hello "+Utils.processHost(addr.getHostName()));
}
```

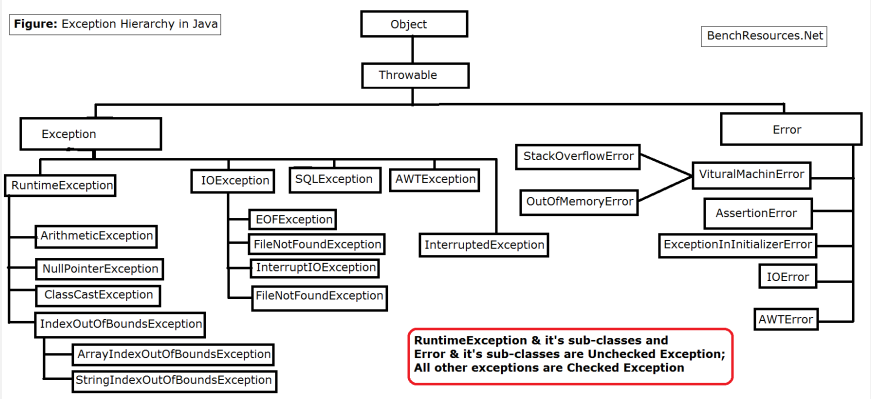


## Safety-Net Exception Handler at the Top Level

All remotely accessible top-level Java methods should catch `Throwable`.

```
protected void doPost (HttpServletRequest req,
                       HttpServletResponse res) {
      try {
          String ip = req.getRemoteAddr();
          InetAddress addr = InetAddress.getByName(ip);
          out.println("hello "+Utils.processHost(addr.getHostName()));
      }
      catch (UnknownHostException e) {
          logger.error("ip lookup failed", e);
      catch (Throwable t) {
          logger.error("caught Throwable at top level", t);
      }
  }
}
```

## Exception Class Hierarchy in Java






## Too Broad Exception Handler Deep Within a Program

Bad Idea: Catching exceptions too broadly deep within a program can be misleading and can cause problems.

```
protected synchronized Random getRandom() {
  if (this.random == null) {
      try {
        Class clazz = Class.forName(randomClass);
        this.random = (Random) clazz.newInstance();
        long seed = System.currentTimeMillis();
        char entropy[] = getEntropy().toCharArray();
        for (int i = 0; i < entropy.length; i++) {
            long update = ((byte) entropy[i]) << ((i % 8)*8);
            seed ^= update;
        }
        this.random.setSeed(seed);
      } catch (Exception e) {
          this.random = new java.util.Random();
      }
  }
  return (this.random) ;
}
```



## More on Checked Exceptions

- If the exceptions thrown by a method are not recoverable or should not generally be caught by the caller, consider throwing unchecked exceptions instead of checked exceptions.
- Wrap checked exceptions that cannot be handled or thrown in a `RuntimeException` to make sure they don't go unnoticed.

```
// empty catch block to ignore a checked exception - bad idea
try {
  doExchange();
} catch (RareException e) {
  // this can never hapen
}

// wrap checked exception instead
try {
  doExchange();
} catch (RareException e) {
  throw RuntimeException("This can never happen", e);
}
```



## Resource Leaks and its Implications

- Resource (heap-allocated memory, file handles, database connections, sockets, etc.) leaks can cause serious performance problems.
- Security implications of mismanaged resources are implicit and indirect.
- If a program mismanages resources, an attacker will have an easier time launching denial-of-service attacks.
- Resource leaks that occur as a result of error conditions and exceptions in a program are also hard to track down.
- Solution: Make resource management systematic using error handling *patterns and idioms* to prevent resource leaks.

## Preventing Resource Leaks in C

A memory leak occurs in the code below if the call to `read()` fails.

```
char* getBlock(int fd) {
  char* buf = (char*) malloc(BLOCK_SIZE);
  if (!buf) {
    return NULL;
  }
  if (read(fd, buf, BLOCK_SIZE) != BLOCK_SIZE) {
    return NULL;
  }
  return buf;
}
```



## Preventing Resource Leaks in C

Use *forward-reaching goto* statements so that functions have a single well-defined region for error-handling code and code that frees resources.

```
char* getBlock(int fd) {
  char* buf = (char*) malloc(BLOCK_SIZE);
  if (!buf) {
    goto ERR;
  }
  if (read(fd, buf, BLOCK_SIZE) != BLOCK_SIZE) {
    goto ERR;
  }
  return buf;

  ERR:
  if (buf) {
    free(buf);
  }
  return NULL;
}
```



## Preventing Resource Leaks in C++

The code below, which uses exceptions for error handling, leaks a file handle if an error occurs.

```
void decodeFile(char* fName)
{
    int return;
    char buf[BUF_SZ];
    FILE* f = fopen(fName, "r");

    if (!f) {
        printf("cannot open %s\n", fName);
        throw Open_error(errno);
    } else {
        while (fgets(buf, BUF_SZ, f)) {
            if (checkChecksum(buf) == -1) {
              throw Decode_failure();
            } else {
              decodeBlock(buf);
            }
        }
    }
    fclose(f);
}
```



## Preventing Resource Leaks in C++

In C++, a *destructor* method provides an excellent place to ensure that resources are always properly released. A destructor method always runs when an object goes out of scope.

```
class File_handle {
    FILE* f;
  public:
    File_handle(const char* name, const char* mode)
      {f = fopen(name,mode); if (f==0) throw Open_error(errno); }
    ~File_handle() { if (f) {fclose(f);} }
    operator FILE*() { return f; }
    ...
};

void decodeFile(const char* fName) {
    char buf[BUF_SZ];
    File_handle f(fName, "r");

    if (!f) {
        printf("cannot open %s\n", fName);
        throw Open_error(errno);
    } else {
        while (fgets(buf, BUF_SZ, f)) {
            if (!checkChecksum(buf)) {
              throw Decode_failure();
            } else {
              decodeBlock(buf);
            }
        }
    }
}
```

## Preventing Resource Leaks in Java

The code below fails to close the `Statement` object if an exception occurs.

```
try {
  Statement stmt = conn.createStatement();
  ResultSet rs = stmt.executeQuery(CXN_SQL);
  harvestResults(rs);
  stmt.close();
}
catch (SQLException e){
  log logger.log(Level.ERROR, "error executing sql query", e);
}
```



## Preventing Resource Leaks in Java

In Java, always call `close()` in a `finally` block to guarantee that resources are released under all circumstances.

```
Statement stmt=null;
try {
  stmt = conn.createStatement();
  ResultSet rs = stmt.executeQuery(CXN_SQL);
  harvestResults(rs);
}
catch (SQLException e){
  logger.log(Level.ERROR, "error executing sql query", e);
}
finally {
  if (stmt != null) {
    try {
      stmt.close();
    } catch (SQLException e) {
      log(e);
    }
  }
}
```



## Preventing Resource Leaks in Java

A reusable helper function might be handy when a given resource needs to be released from multiple places in a program.

```
Statement stmt=null;
try {
  stmt = conn.createStatement();
  ResultSet rs = stmt.executeQuery(CXN_SQL);
  harvestResults(rs);
}
catch (SQLException e){
  logger.log(Level.ERROR, "error executing sql query", e);
}
finally {
  safeClose(stmt);
}

public static void safeClose(Statement stmt) {
  if (stmt != null) {
    try {
      stmt.close();
    } catch (SQLException e) {
      log(e);
    }
  }
}
```



## Logging and Debugging

- Logging and debugging practices provide insight into program execution, particularly when it experiences errors or unexpected conditions.
- Poor logging practices and unsafe debugging code can leave an application vulnerable to attack.

## Good Logging Practices

- Use a centralized logging framework such as Log4j, Logback, JUL (java.util.logging).
- Consistently use the same logging mechanism throughout the code.
- Avoid ad hoc logging through System.out and System.err.
- Time-stamp log entries.
- Log every important action – administration commands, network communication, authentication attempts, attempt to modify the ownership of an object, account creation, password reset requests, etc.
- Do not log sensitive information such as credit card numbers or passwords.
- Protect the logs – prevent attackers from accessing or manipulating log entries; Web applications should store log files in a directory that is inaccessible to Web users.



## Safe Debugging Practices

- Ensure debugging code never appears in a production deployment. It is rarely written with security in mind!
- Keep back-door access code out of production.
  - Back-door access code is designed to allow developers and testers to access an application in ways that are not intended to be available to the end user.
  - If back-door access code is accidentally sent to production, it can give attackers ways to exploit the system.
- Make sure that unused, temporary, and backup files never appear in production. An application framework might serve a file named `index.jsp.bak` in raw source code form as a text file.
- Advocate a zero-tolerance policy towards Easter eggs.
  - Easter eggs are hidden application features usually added for the amusement of programmers.
  - They are a problem from a security perspective – possible vulnerabilities in the code and programmer's motivation behind them?



## References

- Brian Chess and Jacob West, "Secure Programming with Static Analysis", Chapter 8, Addison Wesley, 2007.