Errors and Exceptions

Thomas L. "Trey" Jones, CISSP, CEH

Introduction

- Security problems often begin with an attacker finding a way to violate a programmer's expectations
  - Do not lead directly to exploitable vulnerabilities the way buffer overflow.
  - Provide the conditions necessary for a later security failure.
- Topics
  - Handling errors with return codes
  - Managing exceptions
  - Preventing resource leaks
  - Logging and debugging

HANDLING ERRORS WITH RETURN CODES

Overview

- Fairly straightforward to use return value of a function to communicate success or failure, but
  - It makes it easy to ignore errors
  - Connecting error information with the code for handling the error makes programs harder to read.
  - There is no universal convention for communicating error information.

- C++ and JAVA provides exceptions rather than error codes.

Checking Return Values in C

- Programmer expects buf will contain a null-terminated string of length 9 or less.

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

- What if an I/O Error with fgets occurs?
- What if <EOF> is found instead of characters?

- buf may not have a null terminating character

Checking the Results

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

Must Know Your “tool”

- System functions (such as unlink(), ioctl(), and exec()) return -1 when they fail and 0 (NULL) when they succeed
- fgets() returns NULL when it fails and a pointer to the string it has read when it succeeds

Cleaning up Errors

```
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;
```

Structured Programming

```
char buf[10], cp_buf[10];
char* ret;

ret = fgets(buf, 10, stdin);
if (ret != buf) {
  report_error(errno);
  ... /* cleanup allocated resources */
}
else {
   strcpy(cp_buf, buf);
   ...
}
return;
```

Facilitating Programmer Change

```
int checked_chdir(const char* path) {
  int ret = chdir(path);
  if (ret != 0) {
    fatal_err("chdir failed for %s: %s", path,
              strerror(errno));
  }
  return ret;
}
```

Checking Return Values in Java

- Most errors and unusual events in Java result in an exception being thrown.
- Stream and reader classes do not consider it unusual or exceptional if less data available to read than the programmer requested
  - Add whatever data available to the return buffer
  - Set the return value to the number of bytes or characters read
  - No guarantee that the amount of data returned is equal to the amount of data requested.

```
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();
    }
}
```

Programmer assumes 1K!

```
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

Overview

- Exceptions solve many error handling problems.
- Programmer has to write code specifically to ignore it
- Exceptions allow for separation between:
  - code that follows an expected path and
  - code that handles abnormal circumstances.
- Exceptions come in two flavors: checked and unchecked.
  - A method that declares it throws a checked exception, all methods that call it must either handle the exception or declare that they throw it as well
  - Unchecked exceptions do not have to be declared or handled.
- All exceptions in C++ are unchecked

Catch Everything at the Top Level

- To shut down gracefully and avoid leaking a stack trace or other system information, programs should declare a safety-net exception handler that deals with any exceptions (checked or unchecked) that percolate to the top of the call stack
- DNS lookup failure throws an exception

```
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()));
}
```

Top-level Java methods

- 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);
      }
   }
}
```

The Vanishing Exception

- Both Microsoft C++ and Java support a try/finally syntax. The finally block is always executed after the try block, regardless of whether an exception is thrown.
- If the finally block contains a return statement, it will squash the exception.

Catch Only What You're Prepared to Consume

- Catching all exceptions at the top level is a good idea.
- Catching exceptions too broadly deep within a program can cause problems.
- Tomcat example
  - If any exception derived from java.lang.Exception occurs
    - NullPointerException,
    - IndexOutOfBoundsException, and
    - ClassCastException
  - The code silently falling back on an insecure source of random numbers: java.util.Random.
  - No error message is logged.

Tomcat 5.5.12 Session ID Flaws

```
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) ;
}
```

When Exception Handling Goes Too Far

- Static analysis tools look for code that catch exceptions
  - NullPointerException
  - OutOfMemoryError
  - StackOverflowError.
- Normally, these exceptions should NOT be caught.
- Poor NullPointerException Practices
  - The program contains a null pointer dereference. Catching the resulting exception was easier than fixing the underlying problem.
  - The program explicitly throws a NullPointerException to signal an error condition.
  - The code is part of a test harness that supplies unexpected input to the classes under test.
- The last is the only acceptable use.

Keep Checked Exceptions in Check

- An overabundance of checked exceptions can lead programmers in a number of bad directions.
- The first is to collapse a long list of exception types into the base type for all the exceptions.
- Instead of writing this

```
throws IOException, SQLException, IllegalAccessException
```

- it might seem preferable to write this:

```
throws Exception
```

- Defeats the purpose of meaningful checked exceptions

PREVENTING RESOURCE LEAKS

Overview

- Failing to release resources can affect performance
  - can be hard to track down
  - Surface sporadically under unusual circumstances or heavy load
- Resources include
  - heap-allocated memory,
  - file handles,
  - database connections
- Resource leaks might permit a denial-of-service attack or a quality problem (performance implications),
  - the solution is the same: Make your resource management systematic.

C and C++: Multiple Returns

```
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;
}
```

Better (according to Book)

```
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;
}
```

Best (guideline from "Writing Solid Code" 1993)

- Structured programming disallows “goto”

```
char* szGetBlock (int ifd) {
  char* cpBuf;
  unsigned int uiReadResult;

  cpBuf = (char*) malloc(BLOCK_SIZE)
  if (cpBuf) {
      uiReadResult = read(fd, cpBuf, BLOCK_SIZE);
    if (uiReadResult != BLOCK_SIZE) {
       free (cpBuf);
       cpBuf = NULL;
    }
  }
  return cpBuf;
}
```

Single Return Location

Error Handling C/C++

- C++ programs use exceptions, easier then C.
- C++ destructors always free memory when object goes out of scope.
- If you build your objects correctly, you never need to have an explicit call to close().
- Known by the unusual name Resource Acquisition Is Initialization (RAII).

File_handle “leaks” upon error

```
void decodeFile(char* fName) {
int return;  char buf[BUF_SZ]; FILE* f;

  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);
}
```

File_handle Class

```
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; }
    ...
};
```

Use File_handle class

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

   while (fgets(buf, BUF_SZ, f)) {
      if (!checkChecksum(buf)) {
         throw Decode_failure();
      } else {
         decodeBlock(buf);
      }
   }
}
```

Java Example: DB Query

```
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);
}
```

If an exception occurs while executing the SQL or processing the results, the Statement object will not be closed

The close() location

- In Java, always call close() in a finally block to guarantee that resources are released under all circumstances. Moving close() into a finally block has a number of complicating effects:
  - The resource object must now be declared outside the try block.
  - The resource object must be initialized to null (so that it will always be initialized, even if createStatement() throws an exception).
  - The finally block must check to see if the resource object is null.
  - The finally block must deal with the fact that, in many cases, close() can throw a checked exception.

Object Always Closed

```
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);
    }
  }
}
```

Alternative Method (helper function)

…

…

```
finally {
  safeClose(stmt);
}

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

LOGGING AND DEBUGGING

Overview

- Logging and debugging provide insight into understanding program execution
- Examine:
  - advantages of creating a constant logging behavior
  - segregating debugging aids from production code.

Centralize Logging

- A centralized framework makes it easier to do the following:
  - Provide one consistent and uniform view of the system reflected in the logs.
  - Facilitate changes, such as moving logging to another machine, switching from logging to a file to logging to a database, or updating validation or privacy measures.
- Avoid ad hoc logging through System.out and System.err,

Basic Logging Requirements

- Time-Stamp Log Entries
- Log Every Important Action
  - administration commands,
  - network communication,
  - authentication attempts,
  - an attempt to modify the ownership of an object
  - account creation,
  - password reset requests,
  - purchases,
  - sales,
  - paid downloads,
  - any other application event in which something of value changes hands

Do not log (leak) sensitive information!

Log Success and Failure Events

```
public int createUser(String admin, String usrName, String passwd) {
  logger.log(Level.INFO, admin + "initiated createUser()
             with name '"  + usrName + "'");
  int uid = -1;
  try {
    uid = provisionUid(usrName, passwd);
    return uid;
  }
  finally {
    if (uid != -1) {
      logger.log(Level.INFO, "finished createUser(), '"
                 + usrName + "' now has uid " + uid);
    } else {
      logger.log(Level.INFO, "createUser() failed for '"
                 + usrName + "'");
    }
  }
}
```

Protect the Logs

- Whether directly writing directly into log files or using sophisticated database:

Prevent attackers from gaining access to important details about the system or manipulating log entries in their own favor!

"Guide to Computer Security Log Management"

http://csrc.nist.gov/publications/nistpubs/800-92/SP800-92.pdf

Debug Aids

- Keep Debugging Aids and Back-Door Access Code out of Production
- Debugging code does not receive the same level of review and testing as the rest of the program and is rarely written with stability, performance, or security in mind.
- The same hooks that allow developers to debug allow attackers access to the code

Always remove debug code before deploying a production version of an application.

Back-door access code

- Back-door access code is a special case of debugging code.
- Back-door access code is designed to allow developers and test engineers to access an application
- Back-door access code is often necessary to test components of an application in isolation or before the application is deployed in its production environment.
- See Passport to Trouble Side Bar

- **"Netgear and Linksys hide router backdoor instead of closing it" – April 22, 2014**

Passport to Trouble (Microsoft Passport Vulnerability in 2003)

- All you have to do is hit the following in your browser:
  - https://register.passport.net/emailpwdreset.srf?lc=1033&em=victim@hotmail.com&id=&cb=&prefem=attacker@attacker.com&rst=1
- And you'll get an email on attacker@attacker.com asking you to click on a url something like this:
  - http://register.passport.net/EmailPage.srf?EmailID=CD4DC30B34D9ABC6&URLNum=0&lc=1033
- From that URL, you can reset the password and I don't think I need to say anything more about it.

Clean Out Backup Files

- Unused, temporary, and backup files never appear in production code
- Backup files offer attackers a way to travel back in time
- Backup files likely reflect antiquated code or settings,
  - Prime location for security vulnerabilities or other bugs
- Automated Web attack tools search for backup files by riffing on filenames that are exposed through the site.
- Use input validation techniques and create a whitelist that restricts the files

Web Application Archive

```
<war
       destfile="${web.war.file}"
       webxml="${config.dir}/webxml/web.xml">

        <fileset dir="${build.dir}">
          <include name="**/*.jsp"/>
          <include name="**/*.jar"/>
          <include name="**/*.html"/>
          <include name="**/*.css"/>
          <include name="**/*.js"/>
          <include name="**/*.xml"/>
          <include name="**/*.gif"/>
          <include name="**/*.jpg"/>
          <include name="**/*.png"/>
          <include name="**/*.ico"/>
        </fileset>
</war>
```

Do Not Tolerate Easter Eggs

- Easter eggs are hidden application features usually added for the amusement of programmers.
- Easter eggs are a problem from a security perspective.
  - First, they are rarely included in the security review process, so they are more likely to ship with vulnerabilities.
  - Second, it is difficult to assess the motivation behind a vulnerability in an Easter egg.
  - A zero-tolerance policy toward Easter eggs.

- https://www.youtube.com/watch?v=dtfBBNYdcPc

Conclusion

- Every serious attack on a software system begins with the violation of a programmer's assumptions.
- Communicating error information with return values leads to messy code – programmers are tempted to not implement.
- Exceptions are a superior way to communicate unexpected situations – can be consciously ignored.
- Java's checked exceptions are useful because they enable the Java compiler to find bugs at compile time.
- Insure your code releases any resources it uses.
- Use a logging framework as the basis for a consistent logging discipline.