Introduction to Software Security
Chapter 3.4: Exceptions

Loren Kohnfelder
loren.kohnfelder@gmail.com

Elisa Heymann
elisa@cs.wisc.edu

Barton P. Miller
bart@cs.wisc.edu

Revision 2.0, January 2022.

Objectives

- Understand in depth what exceptions are and how they work.
- Enumerate the kinds of flaws that arise with the use of exceptions that can lead to vulnerabilities.
- Describe mitigation strategies to avoid or reduce harm with problematic exceptions usage.

Exceptions basics

Exceptions are a nonlocal control flow mechanism typically used to propagate error conditions, in stack-based languages such as Java, C#, C++, Python, Ruby, and many more. The exception **try** block specifies the scope of the code where an exception may arise and the **catch** block(s) handle exceptions that may occur, typically doing clean up and logging what happened.

Here is an example illustrates the basic parts of an exception handler in Java and C#:

```
try {
   // code where exceptions may arise
} catch (Exception e) {
  // perform logging, cleanup, and error recovery
} finally {
   // clean-up code unconditionally executed whether an exception happened or not
}
```

Python and Ruby provide additional exception handling blocks with a few extra features that make it easy to effectively deal with exceptions properly. Here is what the Python and Ruby syntax look like:[1]

Python	**Ruby**
try: # code where exceptions may arise **except** SomeError **as** e: # specific exception type handling **except**: # handling for all other exceptions **else**: # handling when no exception occurred **finally**: # clean-up code unconditionally executed whether an exception happened or not	**begin** # code where exceptions may arise **rescue** SomeError => e # specific exception type handling **rescue** # handling for all other exceptions **retry** # retries execution from **begin** **else** # handling when no exception occurred **ensure** # clean-up code unconditionally executed whether an exception happened or not

Pitfalls and mitigations for exceptions

As with any powerful programming tool, the trick is knowing when and how to use it and then effectively coding to handle all possible cases. Sometimes it seems like you spend more time dealing with error conditions than with the code you set out to write in the first place, but cutting corners dealing with exceptions is rarely a good idea. Time spent getting this stuff right will save many hours of dealing with improperly handled errors later — or worse, major problems happening unawares. Let's consider the common categories of mistakes with exceptions that can lead to security vulnerabilities.

Doing nothing about exceptions: If your code never handles exceptions then one error can lead to a crash and take down an entire system. An attacker can exploit this triggering an exception, achieving a Denial of Service.

Catching exceptions without doing enough recovery: Rarely is it good practice to catch but then ignore an exception. What needs doing depends on the code but here is some general guidance:

- If code has already made changes or allocations that the error will cause to be abandoned, be sure to revert changes and deallocate resources.
- Generally **try** blocks should constitute operations that should complete as a whole, and when an exception prevents this from happening, it is best to completely restore state as if nothing happened at all.
- If you attempt to retry, be careful not to leave around redundant allocations or get caught in an infinite loop.

[1] Other popular languages use different keywords and have some additional features. There is a nice summary in Wikipedia, `https://en.wikipedia.org/wiki/Exception_handling_syntax`.

- Be careful of anything inside a **catch** block that could trigger an exception itself. While these exceptions can be handled, the logic often gets confusing and may be best avoided.

Information leakage: It is very common for systems under development to be full of debugging output to help with diagnosis, but unless this is thoroughly removed or disabled in production systems it can lead to serious unintended disclosure of information. Publicly visible logging or error messages may expose internal state, possibly including sensitive information. Many languages make it easy for exception handlers to log stack traces which is great for debugging but can reveal internal information.

Given all these potential pitfalls, here are some best practices to securely code with exceptions.

- Add proper exception handling. Callers of methods that throw exceptions will typically need to handle all possible exceptions unless you are certain that the condition cannot possibly arise.
- Throw exceptions to alert calling code of problems rather than ad hoc measures that some callers may fail to test for. For example, a function that returns an object might return null to indicate failure: any calling code that does not check properly will attempt to access the null pointer resulting in an ugly and confusing crash.
- Catching and responding to specific exceptions is almost always preferred to generalized abstract exception handling of what could be anything. Code that anticipates specific problems (e.g. null pointer, or I/O error) will more likely take correct remedial action.
- In **try** blocks that contain large amounts of code, the same specific exception may arise from multiple points in the code. In order to remediate appropriately it may be necessary to use flags or other mechanisms to make it possible to detect where the exception arose.
- Good test coverage of all exception code paths is the best way to avoid surprises in production.
- Exceptions happen for good reasons: catching exceptions and doing nothing is almost never the right thing to do. Silently consuming an exception shields all callers higher up the stack from knowledge of the problem happening.
- Either recover from the error gracefully or rethrow the exception to be handled further up the stack. Unless you are certain that recovery is successful, rethrow to be on the safe side.
- Long-lived services should consider catching general exceptions at a high level to avoid unexpected crashes. (This is one exception to the rule about not catching generalized exceptions.) Ideally catch these in top level request handling code so the error can be reported to the client and any transactional changes can be reverted cleanly in the case of failure.
- Logging should never include sensitive information: it's best to avoid logging data values unless you are certain they are not sensitive. See an example later on of how to separately log sensitive information when necessary without disclosing it in logs or error responses. It can be difficult to judge sensitivity so when in doubt err on the side of caution. General purpose code is a particular challenge because potentially any kind of data might flow through it depending on the application.
- Avoid logging stack traces or configuration data which often contain sensitive data as well as technical details such as internal class and method name that are potentially helpful to attackers.

Exception handling examples

A Java example will clarify the basic points about exception handling, and how to do it correctly. (Note that the equivalent example in C# would be quite similar.)

```
01: boolean Login(String user, String pwd) {
02:     boolean loggedIn = true;
03:     String realPwd = GetPwdFromDb(user);
04:     try {
05:         if (!MessageDigest.getInstance("SHA-256").digest(pwd).equals(realPwd))
06:         {
07:             loggedIn = false;
08:         }
09:     } catch (Exception e) {
10:         // This cannot happen, ignore
11:     }
12:     return loggedIn;
13: }
```

This code is a traditional login method, with the user name and the user-supplied password as arguments. The boolean variable loggedIn is initialized to true. This is a bad practice as, by default, permissions should be never be granted, but this piece of code is based on code that we found in a real system during one of our assessment activities.

In this code, we see that the exception catch block is empty. Again, this is a bad practice and likely indicates that the programmer could not see a case where an exception could be raised in the try block. Since the programmer could not see an obvious case where an exception could be raised, they probably did not have any good ideas as to how to handle an exception here.

Line 1 declares the Login method passing two string parameters: a user name (user) and the login password (pwd) for a login attempt. Line 2 sets the loggedIn flag to be true ahead of a test later to check if the password is correct or not. Line 3 looks up the precomputed hash of the correct password from a database (those details are not relevant for this example)[2] for the given user. Line 4 opens the try block and on line 5 proceeds to compute the cryptographic hash of the password provided and compare that to the expected password hash corresponding to the correct password. If the hashes do not match, line 7 sets the loggedIn flag false so that login will be denied. If no exception occurred, execution proceeds to line 12, returning the loggedIn flag directing if the password is accepted or to deny login.

To see the problem here, consider when an attacker manages to call Login with parameters user="admin", and pwd=null. Line 5 now throws a null pointer exception and execution jumps to the catch block declared on line 9. Per the comment on line 10, the catch block does nothing on the

[2] Passwords should never be stored in their plain text form. Instead they should be stored in a hashed form (not encrypted form). A nice technical discussion of this topic can be found at `https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet`

mistaken assumption this code path is never taken. Since the loggedIn flag was initially true, execution falls through to Line 12 where the loggedIn flag value of true is returned, meaning that login is permitted. By causing the exception the attacker succeeded logging in without having to provide a valid password.

The fix is easy: clear the loggedIn flag to false when any exception happens, as shown in the corrected version below on line 10. Now the exception case should never allow login, and the attack is foiled.

```
01: boolean Login(String user, String pwd){
02:     boolean loggedIn = true;
03:     String realPwd = GetPwdFromDb(user);
04:     try {
05:         if (!MessageDigest.getInstance("SHA-256").digest(pwd).equals(realPwd))
06:         {
07:             loggedIn = false;
08:         }
09:     } catch (Exception e) {
10:         loggedIn = false;
11:     }
12:     return loggedIn;
13: }
```

WTMI (Way Too Much Information)

Error messages or debugging aids that publicly expose internal information are often useful to attackers finding and refining exploits, if not directly serious disclosures. Unfortunately it is still commonplace to use a website and occasionally see detailed internal error messages such as stack traces, error codes, and data dumps, displayed in response to some error condition. Production applications should never publicly expose stack traces or internal state via error messages or logging, and internal logs must be access controlled.

Here is a Java example of code that checks username/password authentication attempts that for our purposes here the author had trouble getting to work. In order to see what was going on, exception messages were modified to convey actual data values — but of course, username/password data is highly sensitive in production use.

```
boolean Login(… user, … pwd)  {
   try {
      ValidatePwd(user, pwd);
      return true;
   } catch (Exception e) {
      print("Login failed.\n");
      print(e + "\n");
      e.printStackTrace();
      return false;
   }
}

void ValidatePwd(… user, … pwd)
      throws BadUser, BadPwd  {
   realPwd = GetPwdFromDb(user);
   if (realPwd == null)
      throw BadUser("user=" + user);
   if (!pwd.equals(realPwd))
     throw BadPwd("user=" + user
           + " pwd=" + pwd
           + " expected=" + realPwd);
}
```

Never logging any sensitive data is always best, but for debugging purposes it is not always practical to always do so. One technique that addresses this issue is to set up a second logging repository locked down to access only with administrator privileges or similar restrictions.

```
boolean Login {
   try {
      ValidatePwd(user, pwd);
    } catch (Exception e) {
       logId = LogError(e);  // write exception and return log ID.
       print("Login failed, username or password is invalid.\n");
       print("Contact support referencing problem id " + logId
             + " if the problem persists.");
       return false;
   }
   return true;
}

// (ValidatePwd is unchanged)
```

With this approach, any sensitive information goes into a restricted store that `LogError()` returns an ID (handle) for reference. Only an authorized developer with the permission of the administrator will be able to get the details corresponding to the ID so it is very unlikely to fall into the hands of an attacker now. Note that the reference ID should ideally be unpredictable and arbitrary handles. Sequential numbers as ID might, for example, leak to attackers the frequency of these errors occurring: not a major leak, but knowing even that might help the attacker learn a little about how the system is functioning.

Of course, this type of problem can occur in many languages. Here is an example in Ruby with the same mistakes as the problematic code Java above.

```
def Login(user, password)
        ValidatePwd(user, password);
        return true
rescue Exception => e
        puts “Login failed”
        puts e.message
        puts e.backtrace.inspect
        return false
end

def ValidatePwd(user, password)
        if wrong password
                raise “Bad passwd for user #{user} expected #{password}”
        end
end
```

Summary

Exception handling done improperly or not at all can result in incorrect state or disclosing additional information that can be useful to attackers. Understanding the possible exceptions and handling those cases properly are well worth the effort. This effort helps to avoid vulnerabilities, improves the overall quality of the code, and makes debugging and testing that much easier in the long run.

Mitigation essentially consists of using exceptions properly and catching and responding correctly when they do occur. Writing test cases to exercise all exception catching code is a great way to ensure that it works correctly. Any empty `catch` block should be suspect: after careful review, if you are convinced that an exception cannot occur, consider adding code logging a fatal error and terminating the process. We recommend this approach because if the termination code ever executes, your analysis was faulty and termination may be preferable to an unanticipated code path running.

Avoid logging possibly sensitive information in reporting exceptions (or at any other time). When you must log details that might be sensitive, log them to a well secured store and use an associated opaque ID to reference them in logs for authorized developers to access safely as needed.

Exercises

1. Write your own version of the exception handler that we presented with an empty catch section.
    a. Test this code to see if you can make it misbehave as we describe.

    b. Add the fix that we suggested and test the code again.

2. Write a general library that handles logging of sensitive information using an opaque ID as a reference to a protected log as described in the WTMI section.
3. Advanced: Choose an open source project written in a language with exception handling and review the code for secure handling of exceptions. (Hint: it's easy to search for the relevant keywords, like "try", "catch" or "except".)
    a. Check that exceptions are handled safely, and ideally tested, in all cases.
    b. Review exception handling logging code to see if it potentially leaks information.
    c. For any problems identified, write and test fixes and ideally submit to the project.