Chapter 13
Exceptions

Revision 3.1, December 2025.

Objectives

- Introduce basic file system concepts and terminology in order to understand the problem.
- Understand the directory traversal attack (or path traversal attack) with examples.
- Learn how to mitigate this type of attack.
- Learn the difference between allowed and blocked list approaches to limiting access.

13.1 The Basics of Exceptions

Exceptions are a mechanism used to handle 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.

Figure 1 shows the basic parts of an exception handler in Java and C#.

Python and Ruby provide additional exception handling blocks with a few extra features that make it easy to effectively deal with exceptions properly. These features are shown in Figure 2[1]. Notice that the key words can vary from language to language, such as **`try`** in Java, C#, and Python, and **`begin`** in Ruby.

```
try {
  // Code where exception may arise
} catch (Exception e) {
  // Code for logging, cleanup, and error recovery
} finally {
  // Code executed whether an exception happened or not
}
```

Figure 1: Basic Parts of An Exception Handler in Java and C#

[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`.

Python	Ruby

`try:`
`  # Where exception may arise`
`except SomeError as e:`
`  # Handle specific exceptn type`
`except:`
`  # handling for all other types`
 	
`else:`
`  # When no exception occurs`
`finally:`
`  # Cleanup code unconditionally`
`  # executed whether exception`
`  # happens or not`
	`begin`
	`  # Where exception may arise`
	`rescue SomeError => e`
	`  # Handle specific exceptn type`
	`rescue`
	`  # handling for all other types`
	`  retry # retries from begin`
	`else`
	`  # When no exception occurs`
	`ensure`
	`  # Cleanup code unconditionally`
	`  # executed whether exception`
	`  # happens or not`

Figure 2: Exception Handler Features in Python and Ruby

In addition, both Python and Ruby allow you to list actions for multiple specific exception types with the **`except`** keyword in Python and **`rescue`** in Ruby. These languages allow you to specify a catch-all action that will be executed when any exception other than the specifically named ones occur. This feature is specified by the **`except`** or **`rescue`** keyword without an exception name listed.

Python and Ruby also allow you to specify an action to take after the **`try`**/**`begin`** code executes when no exception occurs. This action is specified by the **`else`** keyword.

And finally (pun intended), these languages allow you to specify an action that occurs after all other actions in the exception handler, whether or not an exception occurred. Python uses the **`finally`** keyword and Ruby uses **`ensure`**. This feature is useful to clean up and restore the system to a known state. For example, in this code, you might assure that files are closed or memory is freed.

13.2 Pitfalls and mitigations for exceptions

As with any powerful programming tool, the trick with exception handling is knowing when and how to use it and then effectively coding to handle all possible cases. You often 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 right will save many hours of dealing with improperly handled errors later – or worse, major problems happening unawares.

Note that testing of any error detection code is challenging. Error cases, by their very nature, occur less often than the non-error cases, so

Note that while exceptions are common in most modern programming languages, they are controversial because of the difficulty in handling exceptions correctly and completely. As a result, some recent programming languages, notably Rust and Go, have not included this language feature in favor of better and easier-to-use error checking and return values.

Now, we will describe some of 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 by 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 there are general best practices.

If the code has already made changes to the program state (variables) or makes memory allocations that the error will cause to be abandoned, be sure to revert the 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 – either through something like Ruby's **`retry`** primitive or by looping back and re-executing the code starting at the try block – be careful not to leave around redundant allocations or get caught in an infinite loop.

Also 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 it is best to be avoided.

Careful coverage of exceptions:

It is a challenge to make sure that you cover all the relevant exceptions for any given piece of code. If your current block of code does not handle the exception, it will be passed up the call chain until it reaches a method that handles the exception. Of course, if no method handles the exception, then it is likely to trigger a crash. Here are some points to consider.

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.

While this will keep the program from crashing, you might be unexpected behaviors later in the execution. And these unexpected behaviors sometimes can be exploited.

Catching and responding to specific exceptions is almost always preferred to having a generic exception handler. Code that anticipates specific problems (such as null pointer or I/O error) will more likely take correct remedial action.

If your program is a server that executes continuously, you should consider catching generic exceptions at a high level to avoid unexpected crashes. This is one exception to the rule about not catching generic exceptions. Ideally catch these in top level request handling code so that the error can be reported to the client and any changes can be reverted in the case of failure.

In **`try`** blocks that contain large amounts of code, the same specific exception may arise from multiple points in the code. One way to handle this situation is to set flags in the code record where you are and use the value of these flags in the **`catch`** blocks to determine how to recover from the exception. A better approach is to break up the **`try`** blocks into smaller pieces of code.

Information leakage:

It is 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. You can see more details about this risk in Chapter 26 on Address Space Layout Randomization (ASLR).

Given these potential pitfalls, here are a couple of best practices to securely code with exceptions.

Logging should never include sensitive information. It is best to avoid logging data values unless you are certain they are not sensitive. Section 13.4 shows an example of how to separately log sensitive information 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.

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.

```
01 boolean Login(String user, String pwd) {
02   boolean loggedIn = true;
03   String realPwd = GetPwdFromDB(user);
04   try {
05     if (!MessageDigest.getInstance
06         ("SHA-256").digest(pwd).equals(realPwd))
07     {
08       loggedIn = false;
09     }
10   } catch (Exception e) {
11      // This cannot happen, ignore
12   }
13   return loggedIn;
14 }
```

Figure 3: Password Checking Method with Empty **`catch`** Block

13.3 Exception handling examples

We will start by looking at a simple exception handler written in Java. (Note that equivalent example written in C# would look quite similar.) The code in Figure 3 is a typical login method based on code that we found in a real system during one of our assessment activities. Unfortunately, it is carelessly written. There are two problems with this method.

First, the boolean variable `loggedIn` is initialized to true. This is a bad practice as the default should never be that permission is granted.

Second, 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.

The `Login` method takes two string parameters, a user name (`user`) and the login password (`pwd`). Line 2 declares and sets the `loggedIn` flag to be true ahead of a test later to check if the password is correct. 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 line 5/6 proceeds to compute the cryptographic hash of the password

[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`

provided and compare that value to the password hash corresponding to the correct password. If the hashes do not match, line 8 sets the `loggedIn` flag false so that login will be denied. If no exception occurred, execution proceeds to line 13, returning the `loggedIn` flag directing if the password is accepted or login is denied.

To see the problem here, consider when an attacker manages to call `Login` with parameter `user` set to "`admin`", and `pwd` set to `null`. Line 5/6 now throws a null pointer exception and execution jumps to the **`catch`** block declared on line 10. As described in the comment on line 11, the **`catch`** block does nothing on the mistaken assumption this code path is never taken. Since the `loggedIn` flag was initially true, execution falls through to Line 13 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: set the `loggedIn` flag to false when any exception happens, as shown in the corrected version below on line 11. 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
06:         ("SHA-256").digest(pwd).equals(realPwd))
07:     {
08:       loggedIn = false;
09:     }
10:   } catch (Exception e) {
11:     loggedIn = false;
12    }
13:   return loggedIn;
14: }
```

Figure 4: Password Checking Method with Completed **`catch`** Block

13.4 WTMI (Way Too Much Information)

Error or debugging messages that publicly expose internal information are often useful to attackers finding and refining exploits, if not directly causing serious disclosures. Unfortunately, it is still commonplace to use a website and occasionally see a detailed internal error message such as a stack trace, 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.

```
01 boolean Login(String user, String password) {
02   try {
03     ValidatePassword(user, password);
04     return true;
05   } catch (Exception e) {
06     print("Login failed.\n");
07     print(e + "\n");
08      e.printStackTrace();
09      return false;
10   }
11 }

12 void ValidatePassword(String user, String password)
13         throws BadUser, BadPassword  {
14   realPassword = GetPwdFromDB(user);
15   if (realPassword == null)
16     throw BadUser("user=" + user);
17   if (!password.equals(realPassword))
18     throw BadPassword("user=" + user
19                 + " password=" + password
20                 + " expected=" + realPassword);
21 }
```

Figure 5: Example Java Code with a WTMI in the Form of a Stack Trace

We start in Figure 5 with a Java example of code that checks username and password for login. In the process of debugging the original code, the programmer added additional information to the second **`throw`** statement on line 18. And, somehow, this extra information was kept in the production version of the code (an all-too-common type of mistake). Take a careful look at the **`throw`** clause to see what is being included.

The best practice is to make sure that this information never appears to the user. One way to address this issue is to output a generic message to the user and output the detailed sensitive information to the system log, which should only be accessible to someone with administrator privileges or similar restrictions.

In Figure 6, we modify the `Login` method to follow the best practice. Any sensitive information goes into a log with restricted access `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.

```
01 boolean Login(String user, String password) {
02   try {
03     ValidatePassword(user, password);
04   } catch (Exception e) {
05     logID = LogError(e);
06     print("Bad login, username or pwd invalid.\n");
07     print("Contact support referencing problem ID "
08            + logID + " if the problem persists.\n");
09     return false;
10   }
11   return true;
12 }
```

Figure 6: `Login` Method in Figure 5 Modified to not Present Sensitive Information to User

Of course, this type of problem can occur in many languages. The same approach applies to any language that you use.

13.5 Summary

Exception handling done improperly or not at all can result in unpredictable behaviors or disclosing 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 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 secured store and use an associated ID to reference them in logs for authorized developers to access safely as needed.

13.6 Exercises

1. In your language of choice, write your own version of the exception handler in Figure 5 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 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 a report on your findings to the software project.