04834580 Software Engineering (Honor Track) 2024-25

Error Handling

Sergey Mechtaev
mechtaev@pku.edu.cn
School of Computer Science, Peking University

Incorrect Error Handling May Cause Vulnerabilities

北京大学

```c
int check_if_ca(...) {
  ...
  result = ...;
  if (result < 0) {
    goto cleanup;
  }
  ...
  result = 0;
cleanup:
  return result;
}
int _gnutls_verify_certificates2(...) {
  ...
  if (check_if_ca(...) == 0) {
    result = 0;
    goto cleanup;
  }
  ...
  result = 1;
cleanup:
  return result;
```

GnuTLS error handling bug [1]:

- check_if_ca returns < 0 to indicate an error;
- _gnutls_verify_certificate2 does not handle negative values;
- Invalid certificate issuer is classified as valid.

Security vulnerability (CVE-2014-0092)

Types of Errors [2]

Definition (Recoverable error)

A recoverable error is usually the result of programmatic data validation, e.g. the program has examined the state of the world and deemed the situation unacceptable for progress. It is a predictable and, frequently, planned situation, despite being called an "error."

Definition (Bug)

A bug is a kind of error the programmer didn't expect, leading to an arbitrary damage to the program's state.

Not differentiating these categories frequently leads to unreliable code.

```
void businessLogic(File f) {
  try {
    processData(f);
  } catch ( Exception  e) {
    askUserDifferentFile(e);
  }
}

void processData(File f) throws IOException {
  Data d = parseFile(f);
  buggyAlgorithm(d);
}
```

Recoverable errors (e.g. non-existent files) and bugs (e.g. wrong implementation of algorithms) require different handling.

Error Handling Mechanisms

- Error Codes
- Defer
- Error Monad
- Exceptions

Error Codes


```c
int foo() {
    // <try something here>
    if (failed) {
        return 1;
    }
    return 0;
}
```

```c
int err = foo();
if (err) {
    // Error! Deal with it.
}
```

- \+ All functions that can fail are explicitly annotated.
- \+ All error handling is explicit.
- – Easy to forget to check errors.
- – Performance of success paths suffers.
- – Subpar usability.

Managing Resources with Error Code Handling

北京大學

```c
bool process_file(const char *filename) {
    FILE *file = NULL;
    char *buffer = NULL;
    bool success = false;

    file = fopen(filename, "r");
    if (file == NULL) {
        perror("Failed to open file");
        goto cleanup;
    }
    buffer = malloc(1024);
    if (buffer == NULL) {
        perror("Failed to allocate buffer");
        goto cleanup;
    }

    ...
    success = true;

cleanup:
    if (buffer)
        free(buffer);
    if (file)
        fclose(file);

    return success;
}
```

Managing Resources with `defer` in Zig

北京大学

Defer is used to execute a statement upon exiting the current block:

```
fn processFile(filename: []const u8) !bool {
    const file = try std.fs.cwd().openFile(filename, .{ .read = true });
    defer file.close(); // Ensure the file is always closed.

    const allocator = std.heap.c_allocator;
    const buffer = try allocator.alloc(u8, 1024); // Allocate a buffer.
    defer allocator.free(buffer); // Ensure the buffer is always freed.

    // Do some work with the file and buffer...
    try std.io.getStdOut().writer().print("Processing file: {s}\n", .{filename});

    return true; // Success
}
```

Monadic Error Handling


```
fn bar() -> Result<(), Error> {
    match foo() {
        Ok(value) => /* Use value ... */,
        Err(err) => return Err(err)
    }
}
```

```
fn bar() -> Result<(), Error> {
    let value = foo()? ;
    // Use value ...
}
```

+ All functions that can fail are explicitly annotated.
+ All error handling is explicit.
+ Doesn't let you forget to check errors.
– Performance of success paths suffers.
– Subpar usability when errors need to be propagated.

panic! is for situations that you deem as unrecoverable:

```
use std::net::IpAddr;

let home: IpAddr = "127.0.0.1"
    .parse()
    .expect("Hardcoded IP address should be valid");
```

Checking Result in situations where you can recover:

```
use std::net::IpAddr;

let home: IpAddr = "127.0.0.1"
    .parse()
    .unwrap_or_else(|_| {
        eprintln!("Failed to parse IP address, falling back to alternative.");
        "0.0.0.0".parse().expect("Alternative IP address should be valid")
    });
```

Checked Exceptions


```java
void foo() throws FooException,
                  BarException {
    ...
}
```

```java
// 1) Propagate exceptions as-is:
void bar() throws FooException, BarException {
    foo();
}

// 2) Catch and deal with them:
void bar() {
    try {
        foo();
    }
    catch (FooException e) {
        // Deal with the FooException
    }
    catch (BarException e) {
        // Deal with the BarException
    }
}
```

+ Simplify propagation of errors.
– Used for unrecoverable bugs, like null dereferences, divide-by-zero, etc.
– Performance suffers.

Anti-Pattern: Error Hiding

Complicates debugging, and degrades user experience:

```java
public String readNameFromFile(Path file) throws IOException {
  String name = "";
  Charset charset = Charset.forName("US-ASCII");
  if (file != null) {
    try (BufferedReader reader =
          Files.newBufferedReader(file, charset)) {
      name = reader.readLine();
    } catch (Exception e) {
      System.err.println("error");
    }
  }
  return name;
}
```

References I

[1] Suman Jana, Yuan Jochen Kang, Samuel Roth, and Baishakhi Ray.
Automatically detecting error handling bugs using error specifications.
In *25th USENIX Security Symposium (USENIX Security 16)*, pages 345–362, 2016.

[2] Joe Duffy.
The error model.
`https://joeduffyblog.com/2016/02/07/the-error-model/`, 2025.