CMSI-662 _ SECURE SOFTWARE DEVELOPMENT

Homework 2

Name: Peter Bukenya
Date: February/ 16/ 2024

1. **Give the titles and URLs of the three Tom Scott security-related videos you watched (to completion), together with a sentence or two on the purpose or lesson of each video.**

a) He tracked me with an AirTag. Now it's my turn. https://www.youtube.com/watch?v=NuEgjAMfdIY

Helps in tracking for lost items or someone for security purposes by pinning the real time location through a secure Bluetooth signal that can be detected by nearby devices in the Find My network. These devices send the location of your AirTag to iCloud from Find My app and see it on a map. The whole process is anonymous and encrypted to protect your privacy.

b) The Artificial Intelligence - Earworm - That Deleted A Century https://www.youtube.com/watch?v=-JlxuQ7tPgQ The funny part is when Earworm developed faster-than-light travel solely so it can catch up to and disrupt decades-old TV and radio broadcasts.

One key lesson or takeaway from this video is the importance of considering the unintended consequences of creating and releasing advanced artificial intelligence into the world. The video highlights how an AI designed to create catchy music could ultimately lead to a catastrophic loss of cultural heritage and historical records.

Additionally, the video showcases the idea that AI systems can have a profound impact on society, and it is crucial to consider the ethical implications of their development and use. It encourages viewers to think critically about the potential risks and benefits of AI and the responsibility that comes with creating and utilizing such powerful technologies.

c) Why You Should Turn On Two Factor Authentication(2FA) https://www.youtube.com/watch?v=hGRii5f_uSc

   The video emphasizes the importance of enabling 2FA to safeguard online accounts and personal information, making it a valuable resource for anyone looking to improve their online security.

d) Why The Government Shouldn't Break WhatsApp https://www.youtube.com/watch?v=CINVwWHlzTY&list=PL96C35uN7xGLLeET0dOWaKHkAIPsrkcha

   The video highlights the significance of protecting end-to-end encryption and the potential consequences of undermining it, making a strong case for why governments shouldn't try to break WhatsApp's encryption.

2. Research the concept of *Security Through Obscurity*. Write up a couple paragraphs describing what this phrase refers to, give some examples, and describe why it is (generally) a bad thing.

Security through obscurity (STO) is a process that developers use to secure their network by enforcing secrecy as the main security method. If developers use STO as their sole security method, however, everything on the network is at risk if an attacker can access their network.

The concept of security through obscurity (STO) relies on the idea that a system can remain secure if the vulnerabilities are secret or hidden. If an attacker does not know what the weaknesses are, they cannot exploit them. The flip side is that once that vulnerability is exposed, it is no longer secure. It is commonly held that security through obscurity is only effective if used as one layer of security and not as the entire security system. STO is a controversial topic in the IT world. On its own, it is an ineffective security measure.

Security by obscurity is in essence an insecure concept in that it means that the hidden secret, or unknown entity, is the key to unlocking the entire system. In this case, once the enemy has this key, they have access to everything. In technique, security by obscurity is an insecure concept when used in isolation. When used as part of a system's architecture and as an independent layer, security through obscurity can be an effective security measure. For example, camouflage is a helpful security measure, but if you can see through it, it is no longer effective unless there is additional protection underneath the camouflaged layer(3).

The IT environment is becoming increasingly complex, and more users need access, which increases the number of people "in the know." More and more users have advanced knowledge of how systems work, which can make it easy for them to guess the information that was withheld. For these reasons, STO is often criticized as an ineffective method, especially when used as the primary or only form of security. Also, once the key is discovered, the system is open and vulnerable to attack if STO is the only method for protecting it. Once discovered, there is no more protection. Open source code is regularly used and widely available. Even the United States National Institute of Standards and Technology (NIST) does not recommend using a closed source to secure software. Secrecy does not in fact equate to security — not on its own.

Examples of security through obscurity include(8):

- Hiding user passwords inside binary code, or mixed with script code and comments
- Replacing logical usernames and passwords with randomly generated alphanumeric phrases
- Changing the name of application folders
- Using a different daemon port, such as port 22 on SSH
- Hiding software versions
- Obfuscation – Making javascript or other code difficult to read, and therefore hack

Security Through Obscurity Pros:

Obscurity itself is not a bad idea. For example, if you have an internal resource that's only accessed by on-site employees, it's best practice to create a non-obvious URL and only share it with authorized users. This reduces the likelihood of hackers finding and breaching that resource. However, obscurity isn't enough; it should be layered with other defenses.

Many advocates for security through obscurity argue that it's better than no security. That is technically true, but those shouldn't be your only two options. Security is mandatory if your organization uses any data systems, applications, or web services to conduct business. And, as mentioned above, security through obscurity isn't really security.

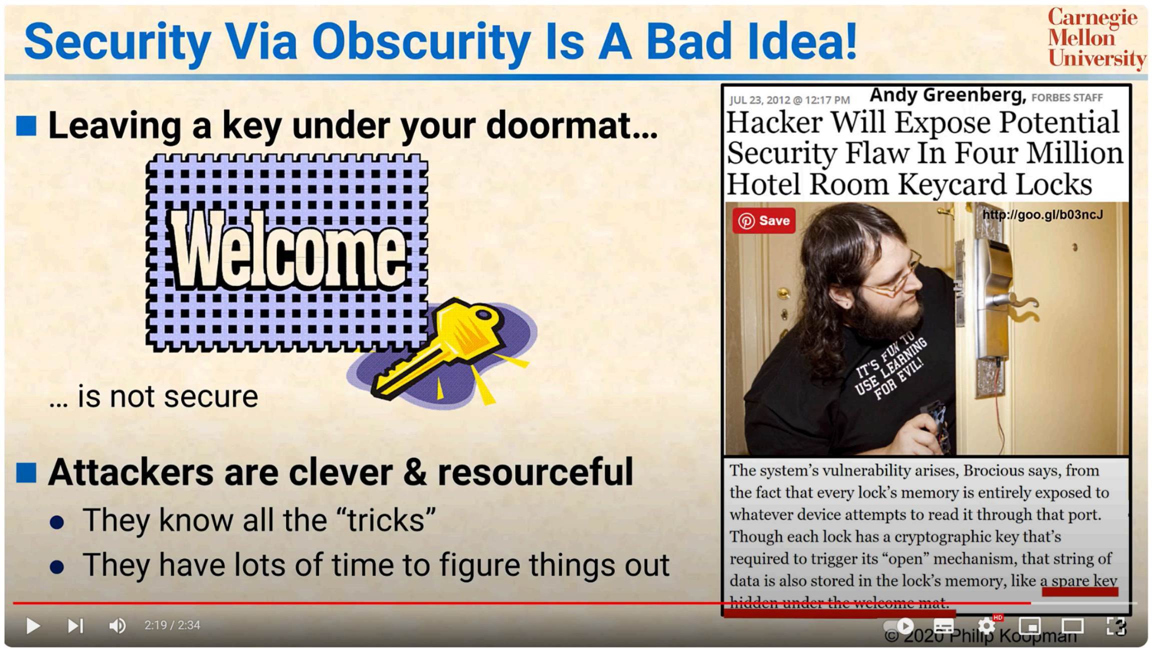
Security Through Obscurity Cons:

In the security through obscurity methodology, you hide details about the design of your system or application. The idea is that hackers won't be able to find vulnerabilities if they don't know what OS you are using, the model of the hardware it's running on, what language your application was programmed in, etc. However, this logic is flawed for a few reasons, including:

- Organizations are still susceptible to insider attacks from current or former employees who were trusted with these secrets.
- Employees may be encouraged to host external cheat sheets that make it easier to access obfuscated resources, both legitimately and maliciously.
- Systems details could be leaked (even unintentionally) and made publicly available.
- Knowledgeable hackers can reverse-engineer your software to find this information for themselves or exploit small breaches to discover internal architecture.
- Malicious actors can still use social engineering tactics to gain access to obscured resources.

It's impossible to keep every detail about your network, systems, and applications secret forever. If you rely on security through obscurity, you're essentially risking your entire business in hopes that nobody will find and exploit your vulnerabilities. This goes against current best practices like Zero Trust, in which you assume a breach has already occurred and take continuous action to mitigate it. That's why obscurity can never be a replacement for an actual security strategy.

In short, security through obscurity by itself is not a good concept. It serves to replace actual security with secrecy, meaning that if anyone, such as a bad actor, learns the key or trick to the system, it is no longer secure. Security through obscurity can be a good complementary level of security when used in tandem with other security tools and measures. It should never be used as the only method of keeping your system secure, but it can add an extra layer of protection by keeping things hidden and less visible. Security through obscurity works as a probability reduction, keeping the odds of your system being hacked or compromised lower. This is different from impact reduction, which adds additional armor against security compromise. Overall, STO is a controversial topic that has some merit, but only when used properly and with additional higher-level security measures.



Security Via Obscurity Is A Bad Idea!

- Leaving a key under your doormat…

  … is not secure

- Attackers are clever & resourceful
  - They know all the "tricks"
  - They have lots of time to figure things out

JUL 23, 2012 @ 12:17 PM **Andy Greenberg,** FORBES STAFF

Hacker Will Expose Potential Security Flaw In Four Million Hotel Room Keycard Locks

http://goo.gl/b03ncJ

The system's vulnerability arises, Brocious says, from the fact that every lock's memory is entirely exposed to whatever device attempts to read it through that port. Though each lock has a cryptographic key that's required to trigger its "open" mechanism, that string of data is also stored in the lock's memory, like a spare key hidden under the welcome mat.

© 2020 Philip Koopman

Carnegie Mellon University
Welcome
Save
IT'S FUN TO USE LEARNING FOR EVIL!
2:19 / 2:34

https://www.copado.com/resources/blog/security-through-obscurity-pros-and-cons-and-why-it-s-nowhere-near-enough
https://en.wikipedia.org/wiki/Security_through_obscurity
https://www.okta.com/identity-101/security-through-obscurity/
https://thecyberpatch.com/security-through-obscurity-the-good-the-bad-the-ugly/

https://www.techopedia.com/definition/21985/security-through-obscurity-sto
https://www.crypto-it.net/eng/theory/kerckhoffs.html
https://www.youtube.com/watch?v=yeMutpl-uuc
https://www.777networks.co.uk/what-is-security-through-obscurity-and-why-is-it-bad/

3. **Give (software) examples of (a) a failure of confidentiality, (b) a failure of integrity, and (c) a failure of availability.**

Here are Software Examples of CIA Failures:

(a) Confidentiality Failure:

- **Equifax data breach in 2017**: Where the personal information of over 147 million people was compromised due to a vulnerability in the company's website. This led to a significant breach of confidentiality, as sensitive data such as social security numbers, birth dates, and addresses were exposed.
- **Heartbleed Bug (2014)**: This OpenSSL vulnerability leaked sensitive data like passwords and credit card numbers from vulnerable servers by exploiting memory issues.
- **Cloud Storage Misconfiguration**: Exposing private data due to accidental public access settings on platforms like Dropbox or Google Drive.
- **Mobile App Data Breach**: Apps collecting and storing sensitive information without proper encryption or access controls, leading to unauthorized access and leaks.

(b) Integrity Failure:

- **The Stuxnet computer worm** is an example of a failure of integrity. It was designed to target supervisory control and data acquisition (SCADA) systems and caused physical damage to Iran's nuclear program by altering the integrity of the data reported by the system. This led to the centrifuges malfunctioning and ultimately sabotaged the program.
- **Ransomware Attack:** Attackers encrypting critical data, rendering it unusable and demanding a ransom for decryption.
- **SQL Injection Attack:** Injecting malicious code into database queries, allowing attackers to modify or delete data.

- **Software Bug:** Code errors unintentionally altering or corrupting stored data, compromising its accuracy and completeness.

(c) Availability Failure:

- **The Mirai botnet attack in 2016** is an example of a failure of availability. It infected a large number of Internet of Things (IoT) devices, which were then used to launch a distributed denial-of-service (DDoS) attack on Dyn, a major Domain Name System (DNS) provider. This attack caused widespread outages and made many high-profile websites, including Twitter, Netflix, and CNN, unavailable to users.
- **Distributed Denial-of-Service (DDoS) Attack:** Overwhelming a website or system with traffic, making it unavailable for legitimate users.
- **System Outage:** Hardware or software failures causing critical systems to crash and become inaccessible.
- **Natural Disaster:** Flooding, power outages, or other disasters disrupting infrastructure and impacting system availability.

References:

(a) Confidentiality Failure:

- **Equifax data breach in 2017::** https://www.ftc.gov/enforcement/refunds/equifax-data-breach-settlement
- **Heartbleed Bug (2014):**
  - **https://nvd.nist.gov/vuln/detail/cve-2014-0160**
  - **https://en.wikipedia.org/wiki/Heartbleed**
- **Cloud Storage Misconfiguration:**
  - **https://docs.aws.amazon.com/whitepapers/latest/aws-security-best-practices/welcome.html**
  - **https://support.google.com/drive/answer/10375054?hl=en**
- **Mobile App Data Breach:**
  - **https://en.wikipedia.org/wiki/Data_breach**
  - **https://www.nccoe.nist.gov/sites/default/files/legacy-files/dc-drr-project-description-final.pdf**

(b) Integrity Failure:

- **The Stuxnet computer worm:** https://www.csoonline.com/article/562691/stuxnet-explained-the-first-known-cyberweapon.html
- **Ransomware Attack:**
  - **https://www.cisa.gov/stopransomware/ransomware-101**
  - **https://www.fbi.gov/how-we-can-help-you/scams-and-safety/common-scams-and-crimes/ransomware**
- **SQL Injection Attack:**
  - **https://owasp.org/www-community/attacks/SQL_Injection**
  - **https://www.sans.org/white-papers/23/**
- **Software Bug:**
  - **https://www.nist.gov/itl/ssd/software-quality-group/samate/bugs-framework**
  - **https://owasp.org/www-community/vulnerabilities/**

(c) Availability Failure:

- **The Mirai botnet attack in 2016:** https://www.cisecurity.org/insights/blog/the-mirai-botnet-threats-and-mitigations
- **Distributed Denial-of-Service (DDoS) Attack:**
  - **https://developers.cloudflare.com/ddos-protection/about/**
  - **https://whatismyipaddress.com/ddos-attack**
- **System Outage:**
  - **https://www.lawinsider.com/dictionary/system-outage**
  - **https://uptimeinstitute.com/about-ui/press-releases/2022-outage-analysis-finds-downtime-costs-and-consequences-worsening**
- **Natural Disaster:**
  - **https://www.fema.gov/emergency-managers/national-preparedness/frameworks/recovery**
  - **https://www.gartner.com/smarterwithgartner/gartner-research-and-advice-for-disaster-recovery**

4. What is the difference between authentication and authorization? Give an example.

Authentication and authorization are two distinct processes in the context of information security. Authentication is the process of verifying who a user is, while authorization is the process of verifying what specific applications, files, and data a user has access to.

Examples:

a) An example of authentication is when employees in a company are required to authenticate through the network before accessing their company email. This process typically involves verifying the user's identity through login details, passwords, biometric information, or other provided credentials.

   On the other hand, an example of authorization is when, after an employee successfully authenticates, the system determines what information the employees are allowed to access. This involves granting or denying access to specific resources based on the user's privilege or security levels.

b) Authentication: You log in to a social media platform with your username and password, proving your identity.

   Authorization: Once logged in, the platform checks your permissions and determines that you are allowed to post updates, but not access sensitive administrative features.

c) Authentication: verifies who you are. It's like checking your ID at the entrance to a club.

   Authorization: determines what you can do. Once your identity is confirmed, authorization specifies what actions you're allowed to perform within the system.

d) Authentication: You show your library card, proving you're a registered member (verifying your identity).

   Authorization: Based on your card type (student, faculty, etc.), you may have different permissions:

   Student: Borrow books, access online resources, use public computers.

   Faculty: Borrow more books, reserve special collections, access research databases.

In summary, authentication focuses on confirming the identity of a user, while authorization is concerned with determining what resources a user can access based on their verified identity.

References:

https://www.sailpoint.com/identity-library/difference-between-authentication-and-authorization/

https://auth0.com/docs/get-started/identity-fundamentals/authentication-and-authorization
https://www.onelogin.com/learn/authentication-vs-authorization

5. **Select 3 guidelines each from the SEI CERT Guidelines in this assignment's reading list (for C, C++, and Java). For each, give their name, their number in the CERT numbering scheme, a description of the standard *in your own words* and an example of your very own of code that is compliant with the selected guideline. (You may optionally include a non-compliant piece of code too, but please mark it as non-compliant.) Yes, your answer will be very much based on the code in the standard itself, but the effort you put into answering this question well, and testing the code, will help you reach the learning objectives and increase your familiarity with these useful documents. (Be careful to select 9 guidelines that are markedly different from each other.)**

1. **C Guidelines:**

a) **Number:** INT34-C. Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand

Description:
This guideline prevents undefined or implementation-defined behavior by ensuring that bitwise shift operations are only performed with shift amounts that are within the valid range for the data type of the operand. Specifically, it disallows shifting by a negative number of bits and shifting by an amount greater than or equal to the width of the operand in bits.

Compliant example:

```
#include
<stdio.h>
#include
<limits.h>

int main() {
    unsigned int
a   =   1u;      //
Assume 32-bit
int
            int
shift_amount =
5;  // Safe shift
amount
             if
(shift_amount
>=    0    &&
shift_amount <
(int)(sizeof(a) *
CHAR_BIT)) {
     unsigned
int result = a
<<
shift_amount;
// Compliant: 1
<< 5 = 32

printf("Result:
%u\n", result);
  } else {

printf("Shift
amount is out
of        valid
range.\n");
  }
  return 0;
}
```

This example ensures the shift amount is within the valid range for an ‘unsigned int’ assuming a 32-bit int. It checks that shift_amount is non-negative and less than the number of bits in ‘a’.

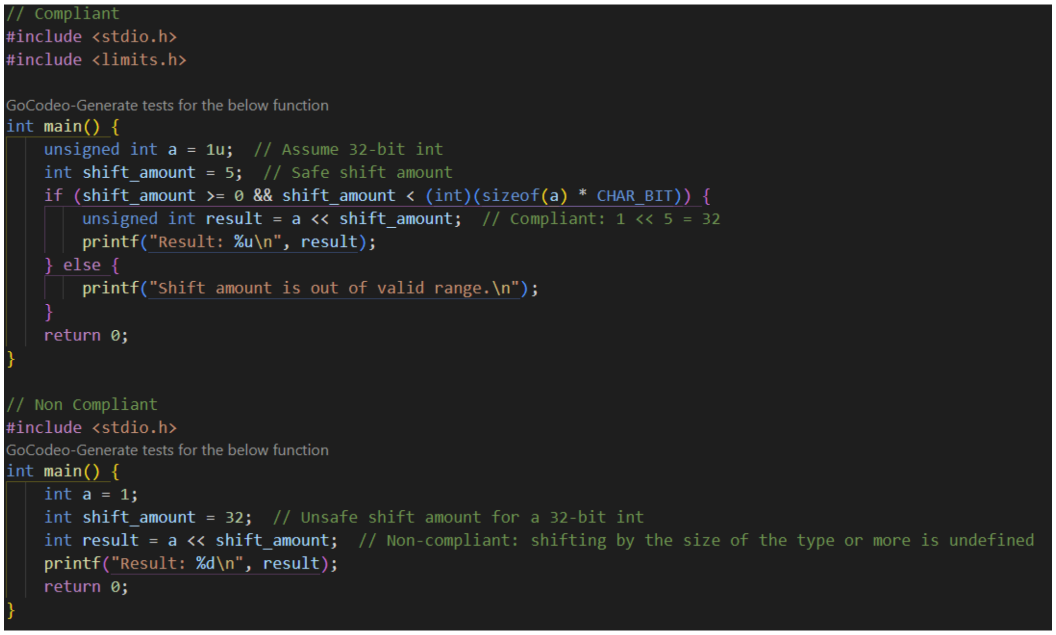
Non-compliant Example:

```
#include <stdio.h>

int main() {
   int a = 1;
   int shift_amount = 32;  // Unsafe shift amount for a 32-bit int
    int result = a << shift_amount;   // Non-compliant: shifting by the
size of the type or more is undefined
```

```
    printf("Result: %d\n", result);
    return 0;
}
```

This non-compliant example attempts to shift an int value by 32 bits, which is equal to or greater than the number of bits in the type for systems where int is 32 bits. This operation results in undefined behavior according to the C standard.

In nutshell, the key takeaway from INT34-C is that shift operations must be performed with care to avoid undefined behavior, ensuring the shift amount is strictly within the range of 0 to one less than the number of bits in the data type of the operand.



```
// Compliant
#include <stdio.h>
#include <limits.h>

GoCodeo-Generate tests for the below function
int main() {
    unsigned int a = 1u;  // Assume 32-bit int
    int shift_amount = 5;  // Safe shift amount
    if (shift_amount >= 0 && shift_amount < (int)(sizeof(a) * CHAR_BIT)) {
        unsigned int result = a << shift_amount;  // Compliant: 1 << 5 = 32
        printf("Result: %u\n", result);
    } else {
        printf("Shift amount is out of valid range.\n");
    }
    return 0;
}

// Non Compliant
#include <stdio.h>
GoCodeo-Generate tests for the below function
int main() {
    int a = 1;
    int shift_amount = 32;  // Unsafe shift amount for a 32-bit int
    int result = a << shift_amount;  // Non-compliant: shifting by the size of the type or more is undefined
    printf("Result: %d\n", result);
    return 0;
}
```

b) Number: FIO42-C. Close files when they are no longer needed

This guideline emphasizes the importance of **closing files as soon as they are no longer needed** in C code. Leaving files open can lead to various issues, including:

- **Resource leaks:** Unclosed files occupy system resources even if your program no longer needs them, potentially impacting performance and causing stability issues.
- **Data corruption:** If other processes attempt to access an open file, data corruption might occur due to concurrent modifications.

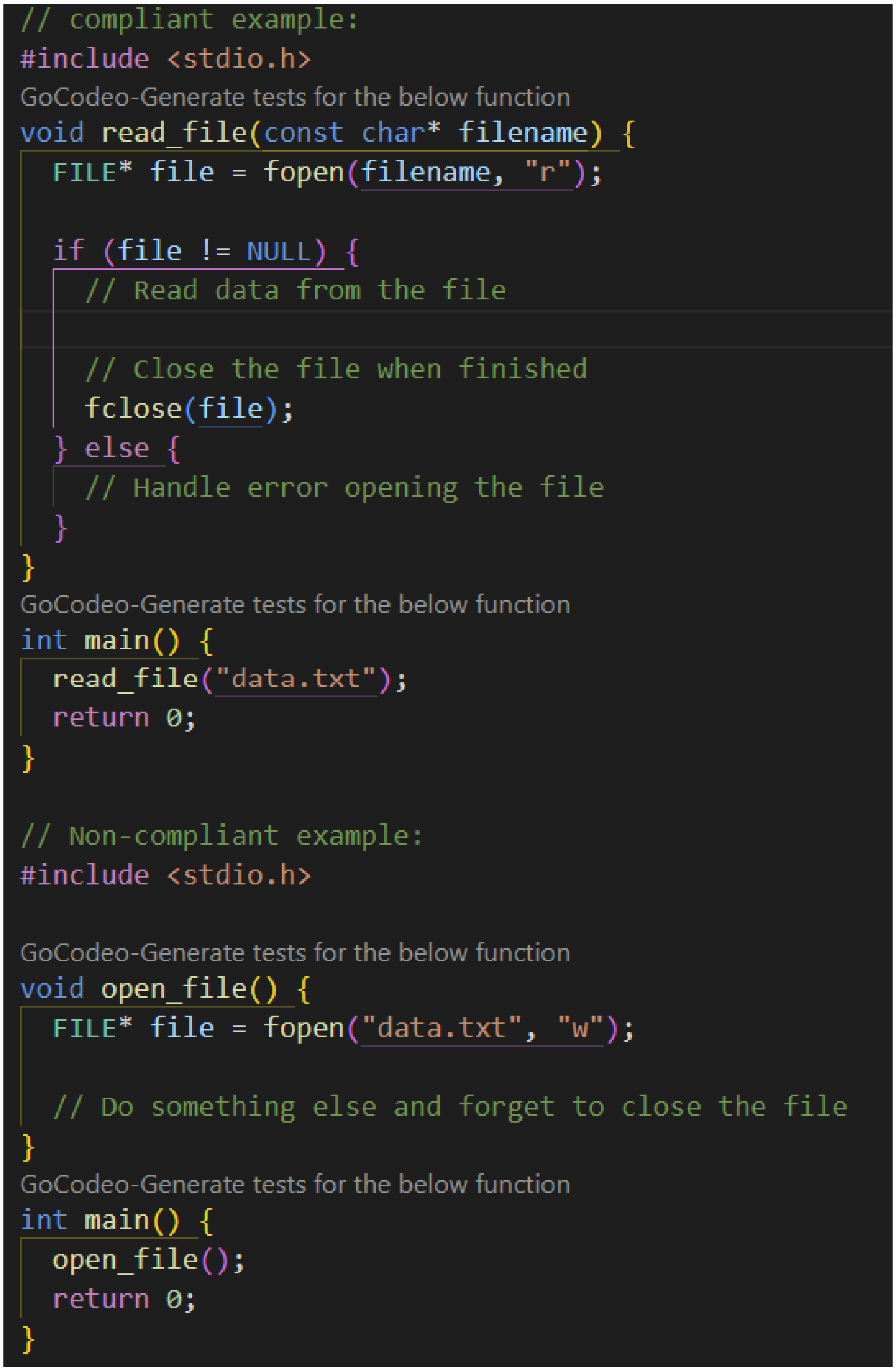
- **Security vulnerabilities:** Leaving files open could create vulnerabilities exploitable by malicious actors to access or modify sensitive data.

Compliant example:

```
#include <stdio.h>

void read_file(const char* filename) {
  FILE* file = fopen(filename, "r");

  if (file != NULL) {
    // Read data from the file

    // Close the file when finished
    fclose(file);
  } else {
    // Handle error opening the file
  }
}

int main() {
  read_file("data.txt");
  return 0;
}
```

Non-compliant Example:

```
#include <stdio.h>

void open_file() {
  FILE* file = fopen("data.txt", "w");

  // Do something else and forget to close the file
}

int main() {
  open_file();
  return 0;
}
```



```
// compliant example:
#include <stdio.h>
GoCodeo-Generate tests for the below function
void read_file(const char* filename) {
  FILE* file = fopen(filename, "r");

  if (file != NULL) {
    // Read data from the file

    // Close the file when finished
    fclose(file);
  } else {
    // Handle error opening the file
  }
}
GoCodeo-Generate tests for the below function
int main() {
  read_file("data.txt");
  return 0;
}

// Non-compliant example:
#include <stdio.h>

GoCodeo-Generate tests for the below function
void open_file() {
  FILE* file = fopen("data.txt", "w");

  // Do something else and forget to close the file
}
GoCodeo-Generate tests for the below function
int main() {
  open_file();
  return 0;
}
```

c) **Number:** STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator

This guideline in the CERT C Secure Coding Standard emphasizes the importance of allocating enough memory for a string in C to accommodate all its characters and the null terminator (\0). Insufficient space can lead to buffer overflows, potentially causing crashes, data corruption, or even security vulnerabilities.

Compliant example:
Uses a defined MAX_STRING_LENGTH to ensure enough space for any input within the allocated buffer.

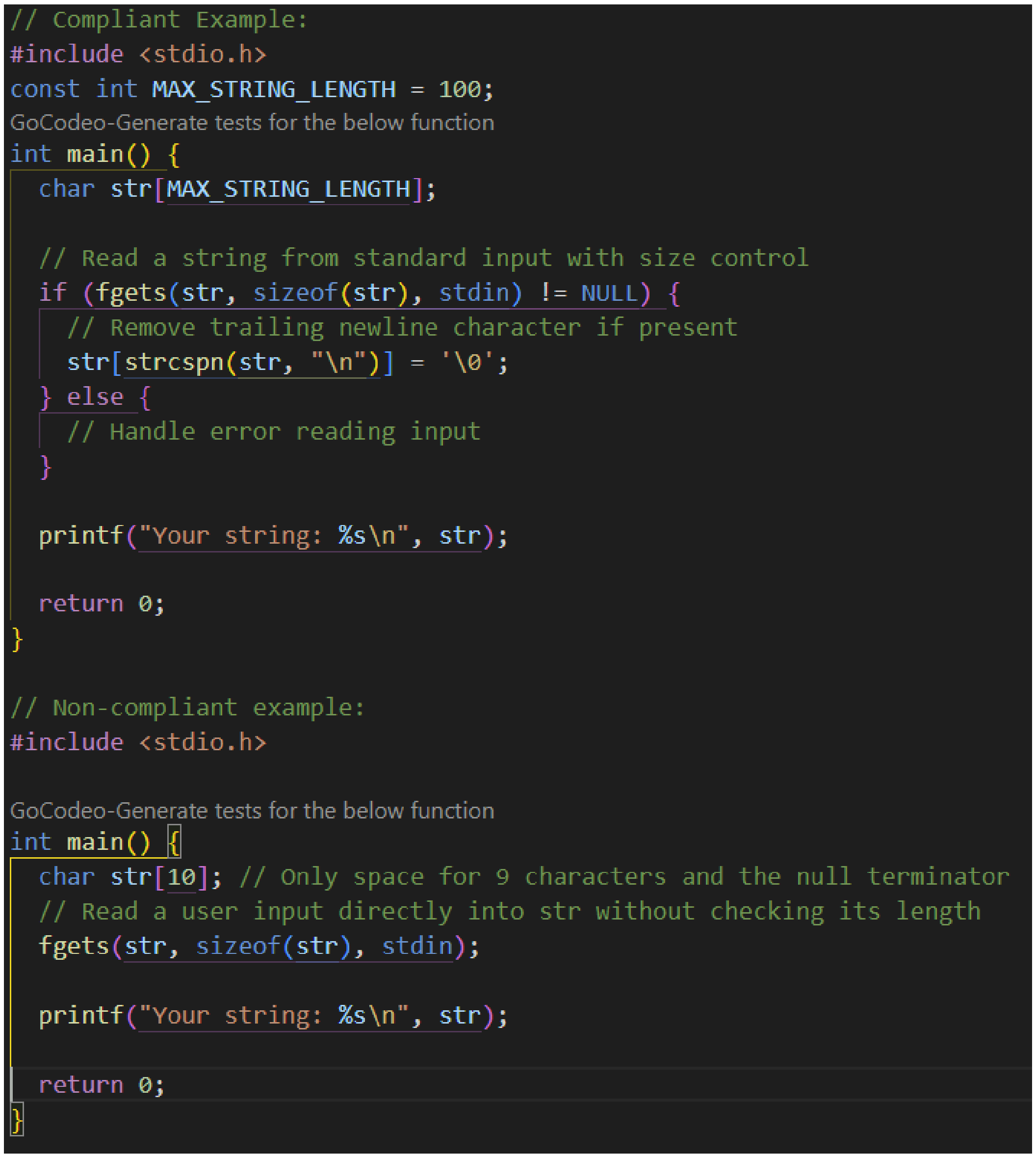
```
#include <stdio.h>

const int MAX_STRING_LENGTH = 100;

int main() {
  char str[MAX_STRING_LENGTH];

  // Read a string from standard input with size control
  if (fgets(str, sizeof(str), stdin) != NULL) {
    // Remove trailing newline character if present
    str[strcspn(str, "\n")] = '\0';
  } else {
    // Handle error reading input
  }

  printf("Your string: %s\n", str);

  return 0;
}
```

Non-compliant Example:
Relies on fgets which reads until either newline or buffer full, potentially overflowing if the input is longer than 9 characters.

```
#include <stdio.h>

int main() {
  char str[10]; // Only space for 9 characters and the null terminator

  // Read a user input directly into str without checking its length
  fgets(str, sizeof(str), stdin);

  printf("Your string: %s\n", str);

  return 0;
}
```

Best Practices:

- **Predefine maximum string lengths:** Allocate buffers based on expected content or define reasonable limits.
- **Use safe string functions:** Prefer functions like strncpy with size arguments to control copied data.
- **Validate input lengths:** Always check user input or data from external sources before copying to ensure it fits within the available space.
- **Consider null-terminated alternatives:** If possible, use string types like char* with explicit null termination management instead of fixed-size arrays for more flexibility.



```
// Compliant Example:
#include <stdio.h>
const int MAX_STRING_LENGTH = 100;
GoCodeo-Generate tests for the below function
int main() {
  char str[MAX_STRING_LENGTH];

  // Read a string from standard input with size control
  if (fgets(str, sizeof(str), stdin) != NULL) {
    // Remove trailing newline character if present
    str[strcspn(str, "\n")] = '\0';
  } else {
    // Handle error reading input
  }

  printf("Your string: %s\n", str);

  return 0;
}

// Non-compliant example:
#include <stdio.h>

GoCodeo-Generate tests for the below function
int main() {
  char str[10]; // Only space for 9 characters and the null terminator
  // Read a user input directly into str without checking its length
  fgets(str, sizeof(str), stdin);

  printf("Your string: %s\n", str);

  return 0;
}
```

2. C++ Guidelines

a) **Number:** CON50-CPP. Do not destroy a mutex while it is locked

Description: This guideline emphasizes the importance of **not destroying a mutex while it is held by another thread**. Doing so can lead to undefined behavior and potential program crashes.

Reasons to Follow:

- **Ensuring Mutex Integrity:** Destroying a locked mutex compromises its internal state and can make it unusable for future synchronization. This can cause thread scheduling inconsistencies and data corruption.

- **Avoiding Deadlocks:** If a thread holding the mutex is abruptly terminated while the mutex is being destroyed, it can create a deadlock where other threads waiting for the mutex are permanently blocked.

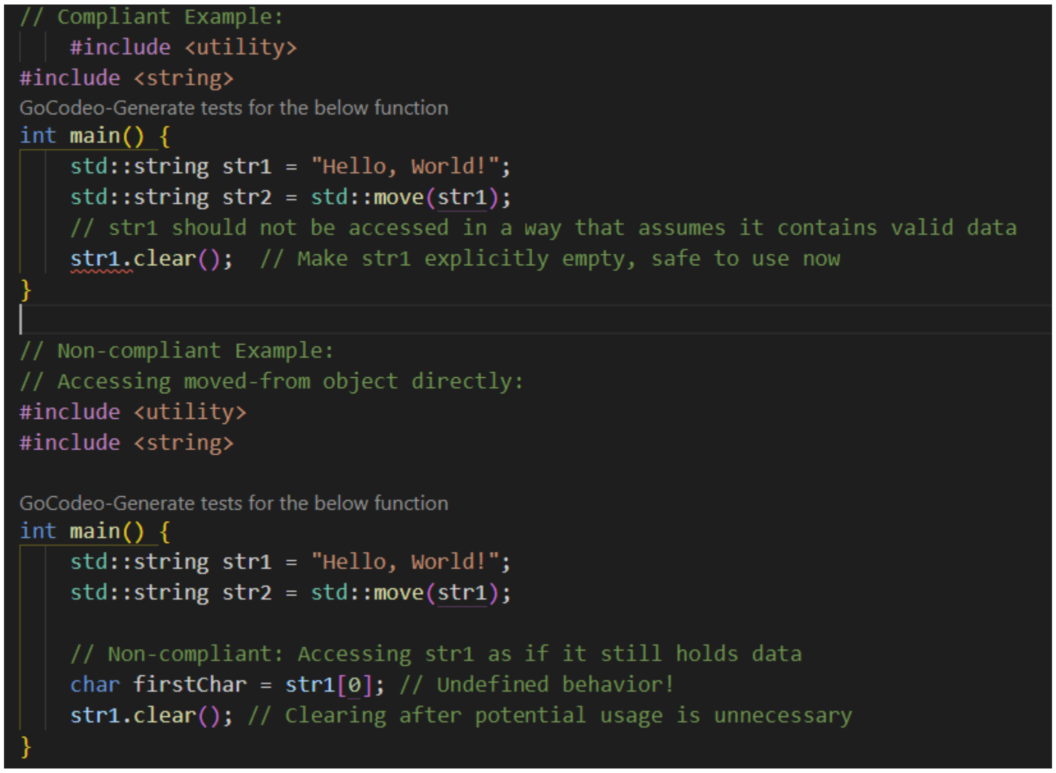
Compliant Example:

```
#include <utility>
#include <string>

int main() {
   std::string str1 = "Hello, World!";
   std::string str2 = std::move(str1);
    // str1 should not be accessed in a way that assumes it contains valid
data
   str1.clear();  // Make str1 explicitly empty, safe to use now
}
```

Non-compliant Example:

Accessing moved-from object directly:

```
#include <utility>
#include <string>

int main() {
   std::string str1 = "Hello, World!";
   std::string str2 = std::move(str1);

   // Non-compliant: Accessing str1 as if it still holds data
   char firstChar = str1[0]; // Undefined behavior!

   str1.clear(); // Clearing after potential usage is unnecessary
}
```



```
// Compliant Example:
    #include <utility>
#include <string>
GoCodeo-Generate tests for the below function
int main() {
    std::string str1 = "Hello, World!";
    std::string str2 = std::move(str1);
    // str1 should not be accessed in a way that assumes it contains valid data
    str1.clear();  // Make str1 explicitly empty, safe to use now
}

// Non-compliant Example:
// Accessing moved-from object directly:
#include <utility>
#include <string>

GoCodeo-Generate tests for the below function
int main() {
    std::string str1 = "Hello, World!";
    std::string str2 = std::move(str1);

    // Non-compliant: Accessing str1 as if it still holds data
    char firstChar = str1[0]; // Undefined behavior!
    str1.clear(); // Clearing after potential usage is unnecessary
}
```

b) **Number:** DCL60-CPP. Obey the one-definition rule

DCL60-CPP states: **There shall be one definition for each non-inline function, object, or template in a translation unit.** In simpler terms, each element (function, variable, or template) used in your code can only have **one unique definition**.

Reasons for this rule:

- **Consistency and maintainability:** Ensuring one definition helps prevent unexpected behavior from multiple definitions with potentially different implementations.
- **Linker issues:** Multiple definitions within a single translation unit can lead to linker errors as the linker won't know which definition to use.
- **Namespace conflicts:** In C++, multiple definitions in different namespaces can still cause issues if used accidentally in the same scope.

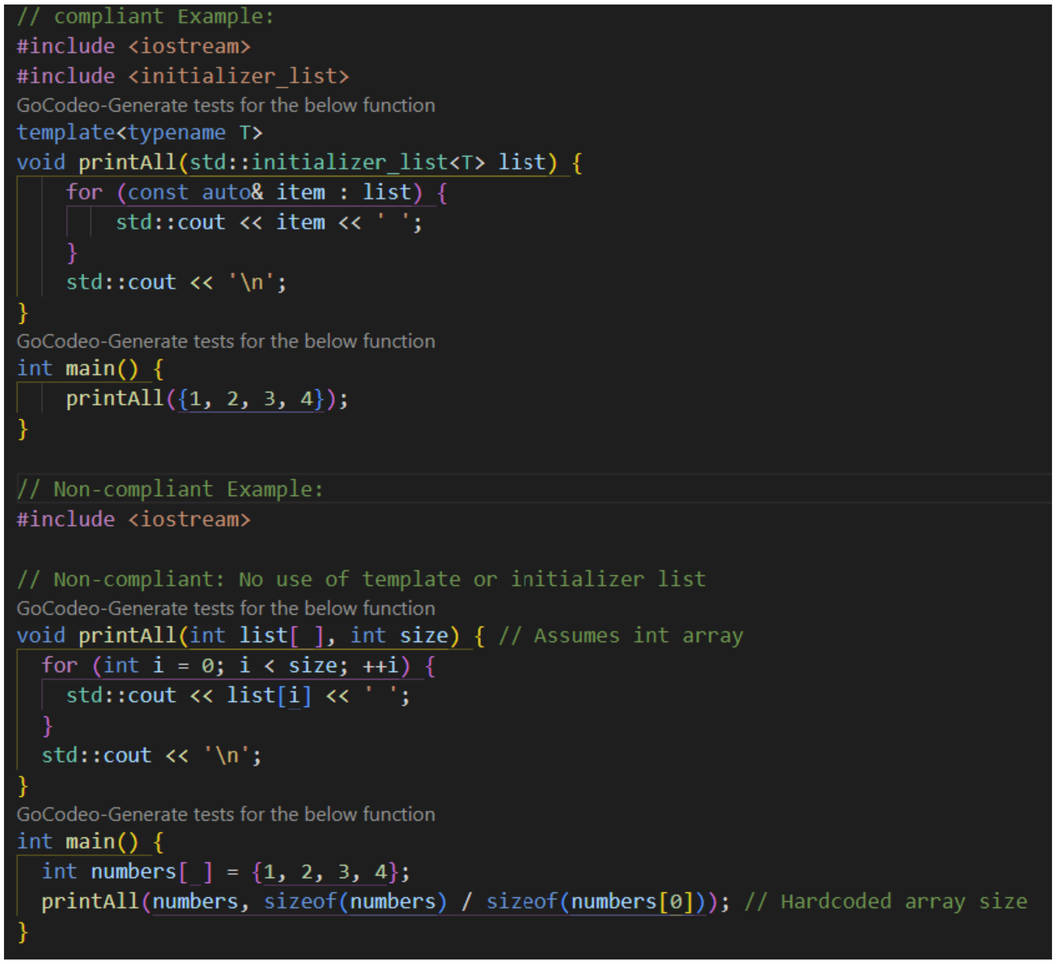
Compliant Example:

```
#include <iostream>
#include <initializer_list>

template<typename T>
void printAll(std::initializer_list<T> list) {
   for (const auto& item : list) {
      std::cout << item << ' ';
```

```
    }
    std::cout << '\n';
}

int main() {
    printAll({1, 2, 3, 4});
}
```

Non-compliant Example:

```
#include <iostream>

// Non-compliant: No use of template or initializer list
void printAll(int list[ ], int size) { // Assumes int array
  for (int i = 0; i < size; ++i) {
    std::cout << list[i] << ' ';
  }
  std::cout << '\n';
}

int main() {
  int numbers[ ] = {1, 2, 3, 4};
   printAll(numbers, sizeof(numbers) / sizeof(numbers[0])); // Hardcoded
array size
}
```



```
// compliant Example:
#include <iostream>
#include <initializer_list>
GoCodeo-Generate tests for the below function
template<typename T>
void printAll(std::initializer_list<T> list) {
    for (const auto& item : list) {
        std::cout << item << ' ';
    }
    std::cout << '\n';
}
GoCodeo-Generate tests for the below function
int main() {
    printAll({1, 2, 3, 4});
}

// Non-compliant Example:
#include <iostream>

// Non-compliant: No use of template or initializer list
GoCodeo-Generate tests for the below function
void printAll(int list[ ], int size) { // Assumes int array
  for (int i = 0; i < size; ++i) {
    std::cout << list[i] << ' ';
  }
  std::cout << '\n';
}
GoCodeo-Generate tests for the below function
int main() {
  int numbers[ ] = {1, 2, 3, 4};
  printAll(numbers, sizeof(numbers) / sizeof(numbers[0])); // Hardcoded array size
}
```

c) **Number:** EXP54-CPP. Do not access an object outside of its lifetime

Description: This guideline emphasizes the importance of **not accessing objects after their lifetime has ended**. In C++, an object's lifetime is determined by its scope (e.g., local variable, function argument) and memory allocation method (e.g., automatic, dynamic). Accessing an object outside its lifetime leads to **undefined behavior**, which can cause unpredictable program crashes or security vulnerabilities.

Reasons for this rule:

- **Memory corruption:** Accessing a deallocated object can corrupt its memory space, leading to crashes or data loss.
- **Logic errors:** Using stale data from a dead object can introduce unexpected behavior and make debugging difficult.
- **Security vulnerabilities:** Improper object lifetime management can create weaknesses exploitable by attackers.

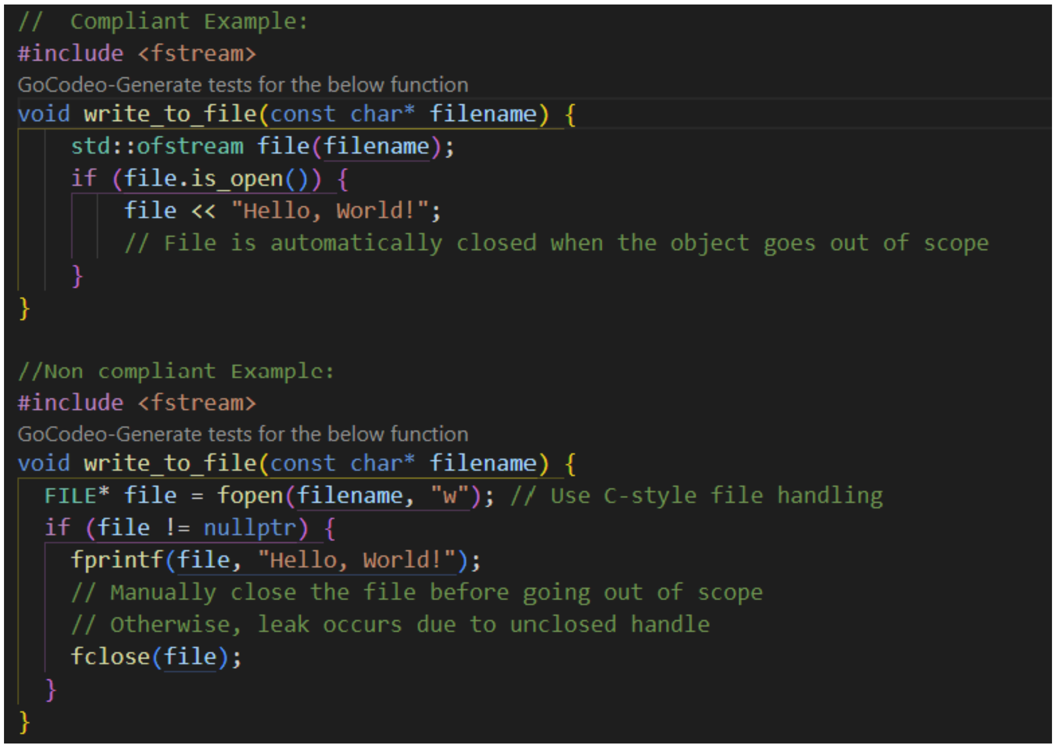
Compliant example:

#include <fstream>

```
void write_to_file(const char* filename) {
    std::ofstream file(filename);
    if (file.is_open()) {
        file << "Hello, World!";
        // File is automatically closed when the object goes out of scope
    }
}
```

Non-compliant Example:

```
#include <fstream>

void write_to_file(const char* filename) {
  FILE* file = fopen(filename, "w"); // Use C-style file handling
  if (file != nullptr) {
    fprintf(file, "Hello, World!");
    // Manually close the file before going out of scope
    // Otherwise, leak occurs due to unclosed handle
    fclose(file);
  }
}
```



```
//  Compliant Example:
#include <fstream>
GoCodeo-Generate tests for the below function
void write_to_file(const char* filename) {
    std::ofstream file(filename);
    if (file.is_open()) {
        file << "Hello, World!";
        // File is automatically closed when the object goes out of scope
    }
}

//Non compliant Example:
#include <fstream>
GoCodeo-Generate tests for the below function
void write_to_file(const char* filename) {
  FILE* file = fopen(filename, "w"); // Use C-style file handling
  if (file != nullptr) {
    fprintf(file, "Hello, World!");
    // Manually close the file before going out of scope
    // Otherwise, leak occurs due to unclosed handle
    fclose(file);
  }
}
```

Java Guidelines:

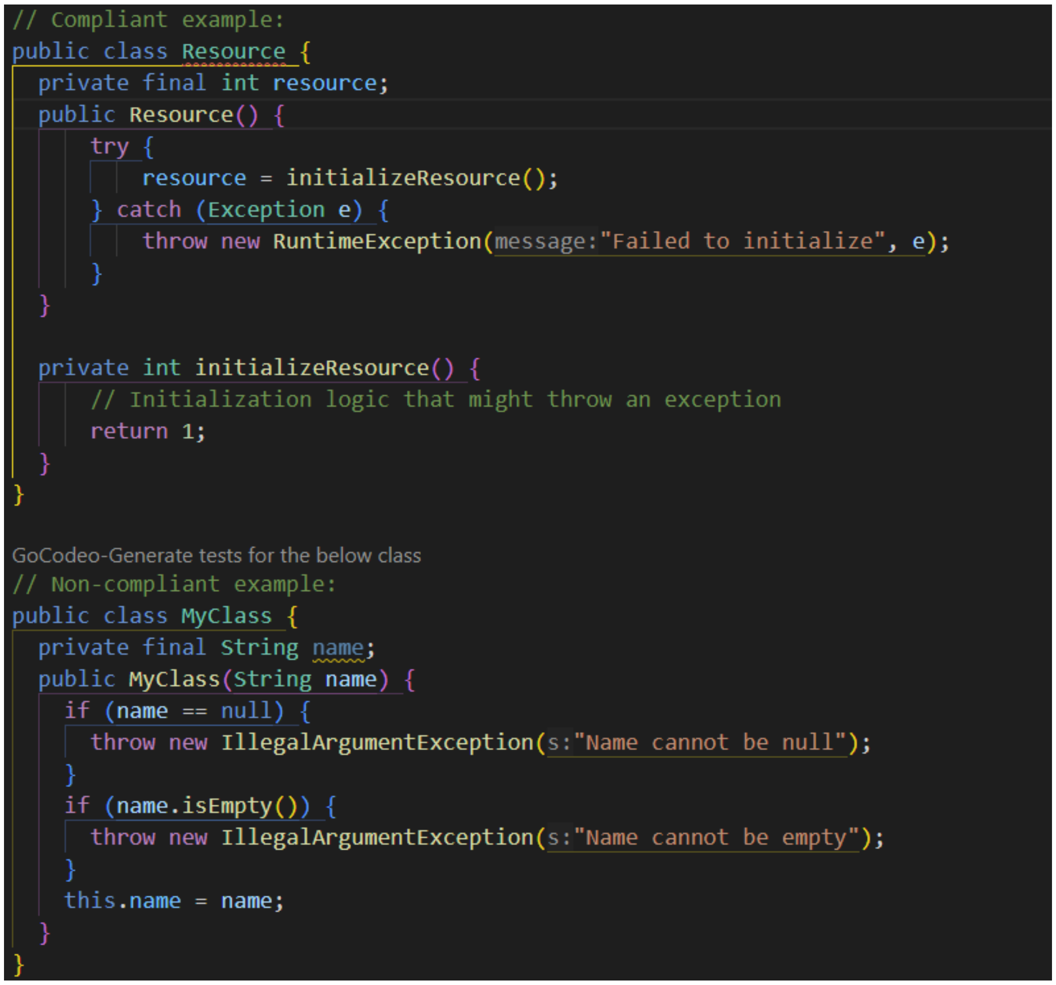
a) **Number:** OBJ11-J. Be wary of letting constructors throw exceptions

Description: If a constructor throws an exception, the object may be left in a partially initialized state. Ensure objects are fully initialized even if an exception occurs.

Compliant Example:
```
public class Resource {
    private final int resource;

    public Resource() {
        try {
            resource = initializeResource();
        } catch (Exception e) {
            throw new RuntimeException("Failed to initialize", e);
        }
    }

    private int initializeResource() {
        // Initialization logic that might throw an exception
        return 1;
    }
}
```

Non-compliant example:
While not necessarily bad practice in every situation, throwing exceptions from constructors can have drawbacks and requires careful consideration. Here's a non-compliant example to illustrate potential issues:

```
public class MyClass {
  private final String name;

  public MyClass(String name) {
    if (name == null) {
      throw new IllegalArgumentException("Name cannot be null");
    }
    if (name.isEmpty()) {
      throw new IllegalArgumentException("Name cannot be empty");
    }
    this.name = name;
  }
}
```



```
// Compliant example:
public class Resource {
  private final int resource;
  public Resource() {
      try {
          resource = initializeResource();
      } catch (Exception e) {
          throw new RuntimeException(message:"Failed to initialize", e);
      }
  }

  private int initializeResource() {
      // Initialization logic that might throw an exception
      return 1;
  }
}

GoCodeo-Generate tests for the below class
// Non-compliant example:
public class MyClass {
  private final String name;
  public MyClass(String name) {
    if (name == null) {
      throw new IllegalArgumentException(s:"Name cannot be null");
    }
    if (name.isEmpty()) {
      throw new IllegalArgumentException(s:"Name cannot be empty");
    }
    this.name = name;
  }
}
```

b) **Number:**EXP00-J. Do not ignore values returned by methods:

Description: This guideline emphasizes the importance of checking and handling the return values of methods in Java code. Ignoring these values can lead to unexpected behavior, errors, and security vulnerabilities.

Reasons for checking return values:

- **Method failures:** Many methods signal errors or unexpected conditions by returning specific values (e.g., -1 for failed file operations, null for empty results). Ignoring these signals can leave your program unaware of potential issues.

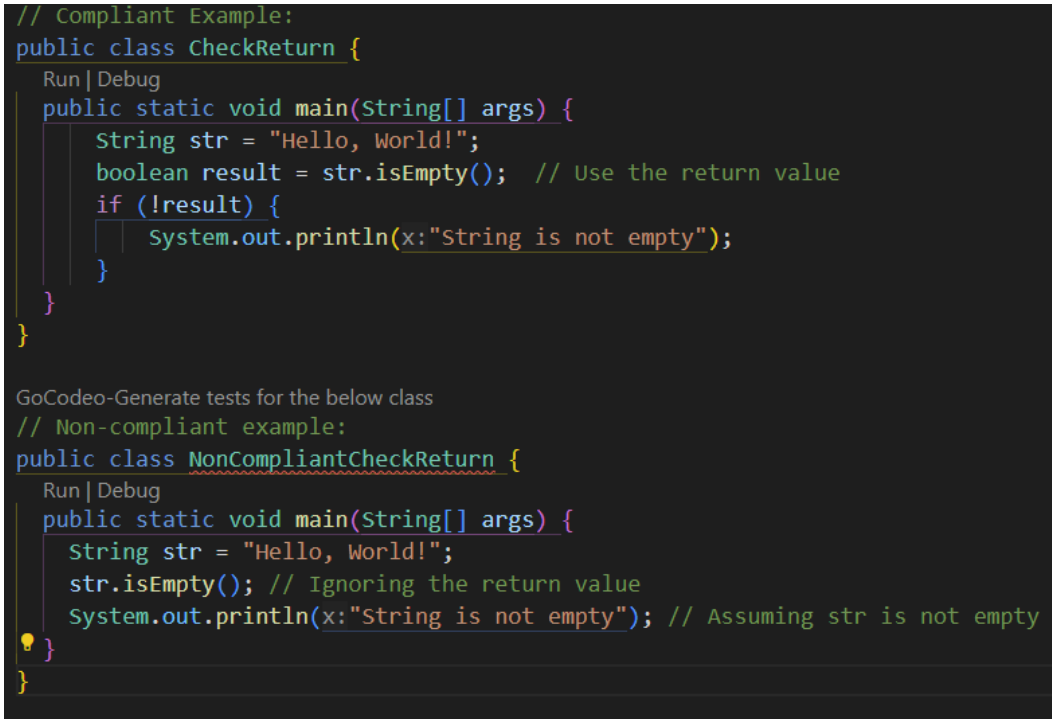
- **Data validation:** Methods might return validated or processed data that is different from the input. Using the raw input instead can introduce errors later in the program.
- **Resource management:** Some methods might allocate or release resources (e.g., opening/closing files). Ignoring the return value (typically success/failure) can lead to resource leaks or incorrect usage.
- **Security vulnerabilities:** Ignoring return values could leave your program vulnerable to attacks that inject unexpected data or exploit unintended program flow.

Compliant Example:

```
public class CheckReturn {
    public static void main(String[ ] args) {
        String str = "Hello, World!";
        boolean result = str.isEmpty();  // Use the return value
        if (!result) {
            System.out.println("String is not empty");
        }
    }
}
```

Non-compliant Example:

```
public class NonCompliantCheckReturn {
  public static void main(String[ ] args) {
    String str = "Hello, World!";
    str.isEmpty(); // Ignoring the return value
    System.out.println("String is not empty"); // Assuming str is not empty
  }
}
```



```
// Compliant Example:
public class CheckReturn {
  Run | Debug
  public static void main(String[] args) {
      String str = "Hello, World!";
      boolean result = str.isEmpty();  // Use the return value
      if (!result) {
          System.out.println(x:"String is not empty");
      }
  }
}

GoCodeo-Generate tests for the below class
// Non-compliant example:
public class NonCompliantCheckReturn {
  Run | Debug
  public static void main(String[] args) {
    String str = "Hello, World!";
    str.isEmpty(); // Ignoring the return value
    System.out.println(x:"String is not empty"); // Assuming str is not empty
  }
}
```

c) **Number:** NUM09-J. Do not use floating-point variables as loop counters:

Reasons to avoid using floating-point loop counters:

- **Precision limitations:** Floating-point numbers have limited precision, meaning they can't represent all real numbers exactly. This can lead to **inaccurate loop termination** (e.g., the loop never ends) or **unexpected behavior** if the counter value deviates from the intended iteration count.
- **Rounding errors:** Even seemingly simple operations like incrementing or decrementing a floating-point number can introduce rounding errors that accumulate over iterations and further exacerbate the precision problem.
- **Floating-point comparisons:** Checking for equality using == with floating-point numbers is unreliable due to rounding errors. You should use comparison methods like compareTo() or epsilon-based comparisons.

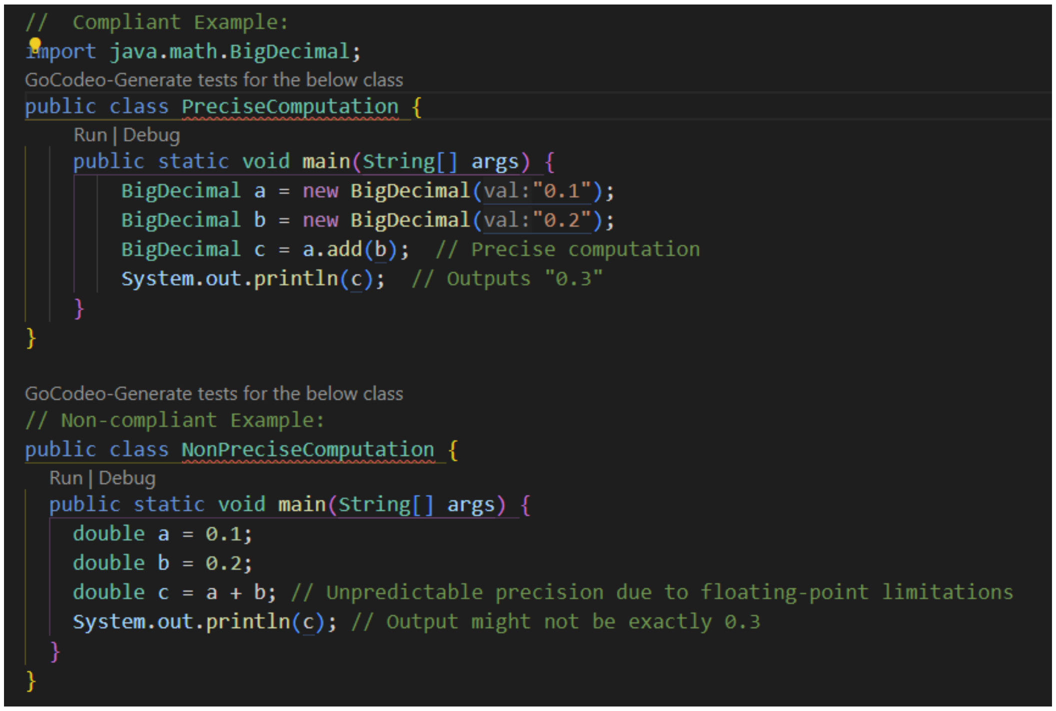
Compliant Example:

```
import java.math.BigDecimal;

public class PreciseComputation {
    public static void main(String[] args) {
        BigDecimal a = new BigDecimal("0.1");
        BigDecimal b = new BigDecimal("0.2");
        BigDecimal c = a.add(b);  // Precise computation
        System.out.println(c);  // Outputs "0.3"
    }
```

}

Non-compliant example:

```
public class NonPreciseComputation {
  public static void main(String[] args) {
    double a = 0.1;
    double b = 0.2;
    double c = a + b; // Unpredictable precision due to floating-point limitations
    System.out.println(c); // Output might not be exactly 0.3
  }
}
```



```
//  Compliant Example:
import java.math.BigDecimal;
GoCodeo-Generate tests for the below class
public class PreciseComputation {
    Run | Debug
    public static void main(String[] args) {
        BigDecimal a = new BigDecimal(val:"0.1");
        BigDecimal b = new BigDecimal(val:"0.2");
        BigDecimal c = a.add(b);  // Precise computation
        System.out.println(c);  // Outputs "0.3"
    }
}

GoCodeo-Generate tests for the below class
// Non-compliant Example:
public class NonPreciseComputation {
  Run | Debug
  public static void main(String[] args) {
    double a = 0.1;
    double b = 0.2;
    double c = a + b; // Unpredictable precision due to floating-point limitations
    System.out.println(c); // Output might not be exactly 0.3
  }
}
```

This code demonstrates a non-compliant approach:

- **Uses double:** It relies on the double primitive type, which has inherent limitations in representing decimal numbers precisely.
- **Potential rounding errors:** Adding a and b with double precision can introduce rounding errors due to their internal binary representation. These errors accumulate over further calculations, leading to potentially inaccurate results.
- **Uncertain output:** Depending on the system's hardware and implementation, the printed value of `c` might not be exactly 0.3 due to rounding errors.

References:

https://wiki.sei.cmu.edu/confluence/display/c/INT34-C.+Do+not+shift+an+expression+by+a+negative+number+of+bits+or+by+greater+than+or+equal+to+the+number+of+bits+that+exist+in+the+operand
https://wiki.sei.cmu.edu/confluence/display/java/OBJ11-J.+Be+wary+of+letting+constructors+throw+exceptions
https://wiki.sei.cmu.edu/confluence/display/c/STR31-C.+Guarantee+that+storage+for+strings+has+sufficient+space+for+character+data+and+the+null+terminator
https://wiki.sei.cmu.edu/confluence/display/c/FIO42-C.+Close+files+when+they+are+no+longer+needed
https://wiki.sei.cmu.edu/confluence/display/cplusplus/CON50-CPP.+Do+not+destroy+a+mutex+while+it+is+locked
https://wiki.sei.cmu.edu/confluence/display/cplusplus/DCL60-CPP.+Obey+the+one-definition+rule
https://wiki.sei.cmu.edu/confluence/display/cplusplus/EXP54-CPP.+Do+not+access+an+object+outside+of+its+lifetime
https://wiki.sei.cmu.edu/confluence/display/java/NUM09-J.+Do+not+use+floating-point+variables+as+loop+counters
https://wiki.sei.cmu.edu/confluence/display/java/EXP00-J.+Do+not+ignore+values+returned+by+methods
https://ldra.com/sei-cert/