TuxCare
We Take Care of Linux

GUIDE

The Bugs Behind the Vulnerabilities

A look at code problems and their real-world impact on cybersecurity. Find out what to look for when you're developing, reviewing, or analyzing code.

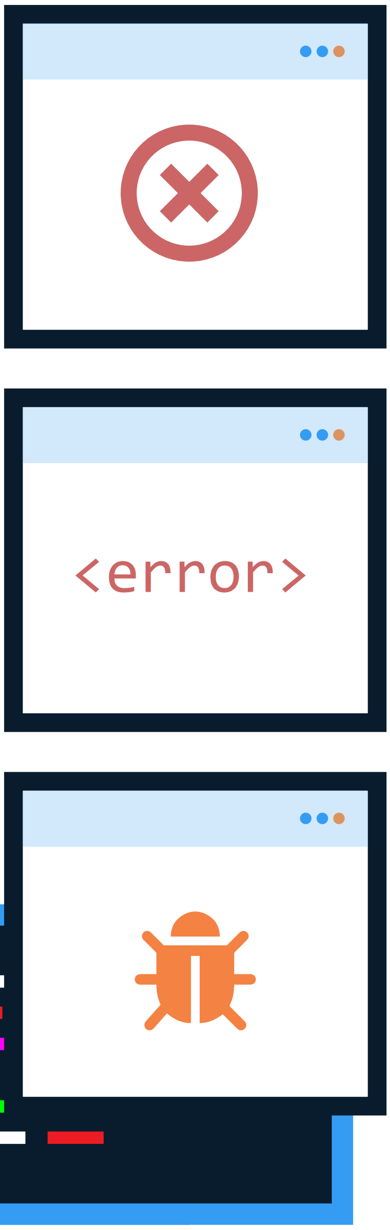
www.tuxcare.com

What this book is about

You've heard all the names before. You know about the most common ones – they're on the news every other day now – but maybe you're not familiar with all of them. You could either research each one as they come along, trying to figure out if the information you're looking for is accurate or up to date, or you can look it up in this book.

Here you'll find information about the top 25 most common software bugs of 2022, according to MITRE. Additionally, you'll get a look at some of the runner-ups that didn't make the top 25 but are still worthy of analysis.

This work started as a 5-part blog series on the TuxCare blog. All parts are compiled here for ease of access, and additional information is also found throughout the book. Most entries have been revised and include more detailed examples, explanations, or sources, should you want to dig deeper into any particular bug. Several new bugs have been added after the publication on the blog.



Who is it the target audience

If you're involved in the software development process, at any level, either directly as a developer, a reviewer, or a manager, this book provides you with the information you need to do your work more efficiently, by explaining the tell tale signs of common issues, and providing guidance on how to avoid and correct them when found.

After all, it's better to fix a potential problem before it can be exploited than try to pick up the pieces and deploy patches after launch.

If you're a cybersecurity professional, this book provides the background information that can help you understand the issues you work with every day. This book also enables you to create better reporting and presentation material for your stakeholders. After all, it's one thing to mention system A is vulnerable to this or that, and another to say that system A is vulnerable because in the code of the web server running on it is a potential buffer overflow that will lead to memory corruption – just to give an example.

The samples

The code samples used throughout the book are not tied to any specific language and include Java, C, C#, Python, and others. These bugs can, and do, affect code written in multiple languages. Relying on a single programming language for the samples might induce the reader into assuming that, when not coding in that particular language, he or she would inherently be safe from the bugs in this list. That is not the case. Even bugs that might, at face value, depend on language specific features (say, pointer arithmetic) which is only immediately accessible on some languages that permit it, might also affect others through indirect means – for example, C++ and C#, when using the “unsafe” keyword in the latter.

MITRE’s Top 25 Most Dangerous Software Weaknesses

As we embark on our exploration of software bugs and their real-world impact on cybersecurity, it's critical to understand the significance of the list that serves as our guide: MITRE's Top 25 Most Dangerous Software Weaknesses. This list is far from a random collection of bugs chosen arbitrarily by a group of industry insiders, or a random developer with a bit too much free time on his or her hands. Instead, it's a carefully curated compilation of vulnerabilities that have been repeatedly exploited in real-world cyber attacks.

The MITRE Corporation is a not-for-profit organization that operates research and development centers sponsored by the federal government. Its Top 25 list is part of the Common Weakness Enumeration (CWE), a community-developed list of software and hardware weakness types. The Top 25 list specifically focuses on the most widespread and critical weaknesses that can lead to serious vulnerabilities in software.

The process of creating the list is rooted in a scientific methodology and real-world data. The CWE team leverages data from security researchers, vulnerability databases, and the broader cybersecurity community to identify the most dangerous software errors. Each error is then scored using a formula that takes into account the weakness's frequency and average harm. This process ensures that the list is a true representation of the threats facing software systems today.

By focusing on the MITRE's Top 25 list, we're not only examining common code bugs, but we're also delving into the actual vulnerabilities that have affected individuals, organizations, and even governments. We're looking at the software errors that have led to data breaches, financial loss, and – in some cases – damage to an organization's reputation.

This real-world grounding is what sets the MITRE's Top 25 list apart. It's not a theoretical exercise; it's a practical guide to the software errors that have the most significant impact on our world. As you read this book, remember that the code samples, explanations, and analysis you find are not abstract concepts. They are real issues that have caused real problems, and they require our attention and understanding. By studying these software errors, we can learn how to prevent them, mitigate their effects, and create more secure software systems.

In the chapters that follow, we'll dive deep into each of the Top 25 software errors, exploring their causes, impacts, and solutions. We'll examine real-world examples and provide practical advice for avoiding these errors in your code. Whether you're a developer, a software reviewer, a manager, or a cybersecurity professional, this book will give you the tools you need to create safer software and defend against today's most dangerous software threats.

Let's begin our journey.

01 Out-of-bounds Write

Out-of-bounds write happens when a program writes data past the end or before the beginning of the intended buffer, which can result in data corruption, a crash, or code execution.

For example, consider the following C code snippet that writes to an array:

```c
#include <stdio.h>


void fill_array(int *array, int len) {

for (int i = 0; i <= len; i++) {
  array[i] = i * 10;
 }
}



int main() {
 int my_array[5];
 fill_array(my_array, 5);
 for (int i = 0; i < 5; i++) {
     printf("Element at index %d: %d\n", i, my_array[i]);
 }

return 0;
}
```

In this example, the function `fill_array` writes to the array `my_array` using the loop variable `i`. However, the loop condition is `i <= len`, which allows the loop to run one iteration beyond the intended bounds of the array, causing an out-of-bounds write.

To avoid this issue, ensure that you use proper boundary checks when writing to buffers. In this case, the loop condition should be `i < len`:

```
for (int i = 0; i < len; i++) {
    array[i] = i * 10;
}
```

With this modification, the code correctly writes to the array within its bounds, preventing an out-of-bounds write vulnerability.

02 Improper Neutralization of Input During Web Page Generation (or “Cross-Site Scripting”)

Cross-site scripting (XSS) vulnerabilities occur when an application does not neutralize or incorrectly neutralizes user-controllable input before it is placed in output used as a web page served to other users. In this context, “neutralization” means invalidating any piece of potentially executable instructions embedded in the input data. All input data should be treated as just data and not executed directly, either intentionally or unintentionally.

Consider the following PHP code snippet:

```
$username = $_GET['username'];
echo "Welcome, " . $username . "!";
```

In this example, user input is directly embedded into the web page without proper sanitization, allowing an attacker to inject malicious HTML or JavaScript code.

To prevent XSS, validate and sanitize user input, use secure output encoding techniques when displaying user input in web pages, and employ content security policies to reduce the impact of an XSS attack. In this case, we should sanitize the user input before displaying it:

```
$username = htmlspecialchars($_GET['username'], ENT_QUOTES, 'UTF-8');
echo "Welcome, " . $username . "!";
```

This type of bug is directly responsible for several different categories in this list, depending on context, and the underlying issue is always an incorrect inclusion of input as part of program flow without checking for malicious content in that input.

03 Improper Neutralization of Special Elements Used in an SQL Command (the dreaded "SQL Injection")

SQL injection occurs when an application constructs all or part of an SQL command using externally-influenced input from an upstream component without neutralizing or incorrectly neutralizing special elements that could modify the intended SQL command.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
    char query[1024];
    char *user_input = argv[1];

    snprintf(query, sizeof(query), "SELECT * FROM users WHERE username='%s'", user_input);

    printf("Generated query: %s\n", query);

    // Execute the query...

    return 0;
}
```

In this example, the user input is directly concatenated to the SQL query, allowing an attacker to inject malicious SQL code.

To prevent SQL injection, use parameterized queries, prepared statements, or stored procedures to separate user input from SQL statements. In this case, we should use prepared statements:

```c
#include <stdio.h>
#include <stdlib.h>
#include <mysql/mysql.h>

(...)

MYSQL_STMT *stmt = mysql_stmt_init(con);
    if (!stmt) {
        fprintf(stderr, "mysql_stmt_init() failed\n");
        exit(1);
    }

    const char *query = "SELECT * FROM users WHERE username=?";
    if (mysql_stmt_prepare(stmt, query, strlen(query)) != 0) {
        fprintf(stderr, "mysql_stmt_prepare() failed: %s\n", mysql_stmt_error(stmt));
        exit(1);
    }
```

By using prepared statements, the user input is treated as data and not as part of the SQL query, effectively preventing SQL injection.

04 Improper Input Validation

Improper input validation happens when a product receives input or data but does not validate or incorrectly validates the input, leading to altered control flow, arbitrary control of a resource, or arbitrary code execution.

```
private void buildList(int untrustedListSize) {
    if (0 > untrustedListSize) {
        die("Negative value supplied for list size, die evil hacker!");
    }

    Widget[] list = new Widget[untrustedListSize];
    list[0] = new Widget();
}
```

In this example, the code checks if the untrustedListSize is negative, but it does not check if it is zero. If a zero value is provided, the code will build an array of size 0 and then try to store a new widget in the first location, causing an exception to be thrown. To prevent this issue, use comprehensive input validation techniques, considering all potentially relevant properties of the input, and do not rely exclusively on looking for malicious or malformed inputs. In this case, we should check if the untrustedListSize is greater than 0 instead.

05 Out-of-Bounds Read

Out-of-bounds read occurs when a product reads data past the end or before the beginning of the intended buffer. This can allow attackers to read sensitive information from other memory locations or cause a crash.

For example, consider the following C code snippet that reads from an array using a user-controlled input:

```
#include <stdio.h>

int get_element(int *array, int len, int index) {
    int result;

    if (index < len) {
        result = array[index];
    } else {
        printf("Invalid index: %d\n", index);
        result = -1;
    }

    return result;
}
```

```
int main() {
    int my_array[5] = {10, 20, 30, 40, 50};
    int index = -1; // Simulating user-controlled input
    int value = get_element(my_array, 5, index);
    printf("Value at index %d: %d\n", index, value);

    return 0;
}
```

In this example, the function get_element checks if the provided index is within the maximum bound of the array, but it does not check the minimum bound. This allows negative values to be accepted as valid array indices, resulting in an out-of-bounds read. To mitigate this issue, add a check for the minimum bound:

```
if (index >= 0 && index < len) {
    result = array[index];
}
```

06 Improper Neutralization of Special Elements Used in an OS Command ("OS Command Injection")

OS command injection occurs when an application constructs an operating system command using externally influenced input without properly neutralizing special elements. This can allow attackers to execute arbitrary commands or code on the targeted system, potentially leading to unauthorized access or control over the system.

```
import os

user_input = input("Enter the name of the file to search: ")
command = f"find / -name {user_input}"

os.system(command)
```

To prevent OS command injection, developers should validate and sanitize user input, use parameterized APIs or functions that do not allow for command separators, and apply the principle of least privilege to minimize the potential impact of a successful attack.

07 Use After Free

"Use after free" is a memory corruption issue that arises when a program continues to use a pointer after it has been freed. This can lead to data corruption, crashes, or even arbitrary code execution, depending on the timing and instantiation of the flaw.

```c
// Example of a use after free error in C

char *ptr = (char *)malloc(SIZE);

if (err) {
    abrt = 1;
    free(ptr);
}

...

if (abrt) {
    logError("operation aborted before commit", ptr);
}
```

To avoid use after free issues, developers should follow secure coding practices, such as proper memory management, using smart pointers in languages like C++, and ensuring proper error handling to prevent dangling pointers.

08 Improper Limitation of a Pathname to a Restricted Directory (aka "Path Traversal")

Path traversal is a vulnerability that occurs when an application uses external input to construct file or directory paths without properly neutralizing special elements. This can allow attackers to access files or directories outside of the intended restricted location, potentially leading to unauthorized access, modification, or disclosure of sensitive information.

```php
// Example of a path traversal vulnerability in PHP

$filename = $_GET['file'];
$content = file_get_contents("/restricted_directory/$filename");

echo $content;
```

To prevent path traversal attacks, developers should validate user input, use secure functions or libraries for path manipulation, and apply proper access controls on the server-side to limit access to sensitive files and directories.

09 Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) is a type of attack that tricks users into executing unwanted actions on a web application in which they are currently authenticated. This occurs when a malicious website, email, or program causes a user's browser to perform an action on another site without the user's knowledge or consent.

```html
<!-- Example of CSRF attack in an HTML form -->

<form action="http://example.com/transfer_funds" method="POST">
  <input type="hidden" name="amount" value="1000" />
  <input type="hidden" name="destination_account" value="attackers_account" />
  <input type="submit" value="Click here for a free gift!" />
</form>
```

To mitigate CSRF attacks, developers should implement anti-CSRF tokens, use secure coding practices such as following the SameSite attribute in cookies, and ensure proper authentication and authorization checks for sensitive actions.

10 Unrestricted Upload of File with Dangerous Type

Unrestricted file upload occurs when an application allows users to upload files without proper validation, enabling them to upload files with dangerous content or types. Attackers can take advantage of this vulnerability to deploy malicious payloads or exploit other security weaknesses within the application or server.

```python
# Example of unrestricted file upload in Python

@app.route('/upload', methods=['POST'])

def upload_file():
    file = request.files['file']
    file.save(os.path.join('uploads', file.filename))
    return 'File uploaded successfully'
```

To prevent unrestricted file uploads, developers should implement robust file validation mechanisms, restrict file types to a known safe list, and perform server-side checks to ensure only safe files are accepted.

While most often found on web based applications, this type of issue can happen in non-web applications too. Just consider how many applications will let you upload files to a server for storage, validation, or other purposes.

11 NULL Pointer Dereference

This is one of the most common sources of vulnerabilities to be found in any given application written in a pointer-friendly language, like C. It consists of trying to use the value referenced by a pointer that has been invalidated previously or hasn't been initialized yet.

The usual result, barring architectural differences, is that the application will crash, as the operating system will detect the attempt to read a memory location outside the allowed memory space of that application.

The following C# sample illustrates the issue:

```
string name;
int k=0;
if(k==1)
{
  name=Console.ReadLine();
}
Console.Writeline(name.Length);
```

When execution reaches the last call, the "Length" method will be called on a null string, crashing the application.

It is interesting to note that many languages have introduced changes (nullable types, compiler flags, new warning types) specifically targeted at catching this type of error early in the development process – including C#, used in the previous example. Others forgo the explicit use of pointers altogether to guard against it, but other than cautious programming practices, there is no silver bullet that is 100% effective in finding and preventing NULL pointer dereferences.

Other bugs described in this list will in fact lead to behavior that triggers a NULL pointer dereferencing, like the one described in bug #12, that can lead to unexpected values being introduced in the flow of the application, possibly triggering this situation.

A relatively simple fix to this bug is to always prepend any suitable variable access by a check to validate it is not NULL. In typical procedural programming, this will often be enough, but concurrent/parallel programming can make that a less optimal solution. In the latter situation, locks, mutexes, and other synchronization algorithms should be employed around critical situations where the value of a variable can change and is shared among different threads or contexts.

12 NULL Pointer Dereference

Serialization (or marshaling) is the process of converting a piece of data stored in memory into a format that can be stored or otherwise sent (for example through a network connection) to a different system/application/component.

Deserialization (or unmarshaling) is the reverse process, where a stream of data gets converted into a memory representation of an application's internal object ("object" meaning variable or grouping of variables).

When an application blindly trusts that the data that it is receiving – either read from a file, accepted from a form submission on a website, or through a network connection – is inherently valid and then attempts to deserialize it into a given memory location means that the application is exposing itself to risk. Specially crafted data streams can trick an application into overwriting existing memory locations outside the intended location or not providing enough data to fill all the variables, potentially leading to uninitialized variables or even delivering data outside expected boundaries – which can all lead to unforeseen events.

As a development rule of thumb, no input should ever be trusted, with no exceptions. Even if the developer is writing the client and the server application themselves, it is always possible to subvert a connection, or attempt to tamper with the communication between them, or impersonate one side of the conversation, thus leading to a scenario where the data being accepted is no longer trustworthy, and should therefore not be deserialized.

The following Java code looks benign enough:

```java
try {
File file = new File("object.obj");
ObjectInputStream in = new ObjectInputStream(new FileInputStream(file));
javax.swing.JButton button = (javax.swing.JButton) in.readObject();
in.close();
}
```

source: https://cwe.mitre.org/data/definitions/502.html

But, upon closer inspection, the file "object.obj" is not validated at all, and could contain something other than a button definition, as the application expects.

13 Integer Overflow or Wraparound

This is one of the earliest forms of software bugs. In a given computer architecture (32/64 bit, ARM, mips, etc), any variable will occupy a given amount of memory. The larger the amount of memory, the larger the value that can be stored in that variable. However, when doing arithmetic operations, it is easy to miss doing a check to see if the result of such an operation will still fit in the memory size of the variable – like, for example, multiplying two integers together and storing the result might lead to a value so large that it no longer fits the maximum variable size. When this happens, the integer will "overflow."

Different computer architectures deal with this event in different ways: some will discard the extra value, resulting in storing the wrong value; others will write the value in the next available memory location, thus potentially overwriting something else already stored there; and others will wraparound the value – like the mileage counter in a car that returns to zero when it reaches the maximum value, and you continue to drive.

The following Python code will trigger the exception because the result from the number multiplication can no longer be stored in an integer:

```python
import sys

def calculate_product(a, b):
    product = a * b
    if product > sys.maxsize:
        raise OverflowError('Integer overflow: The product is too large to be represented 
as an int')
    return product

def main():
    try:
        a = 10**10
        b = 10**10
        product = calculate_product(a, b)
        print(f"The product of {a} and {b} is {product}.")
    except OverflowError as oe:
        print(oe)

if __name__ == "__main__":
    main()
```

Imagining the variable controls the number of iterations in a loop, it can lead to an infinite loop, or, depending on the actual language and architecture used, it can cause a crash, an unresponsive application, or an unresponsive service – and it could also exhaust available memory or overall system instability. It can also directly lead to buffer overflow scenarios (covered in a different bug on this list).

As many computer languages do not include bounds checks by default – and have to explicitly be enabled either through compiler flags or by using specific libraries – it is important to make an informed decision on the language to use for an application that is being planned.

If this is found to happen on an already developed application, then proper bounds checks need to be added to validate that the calculations will never fall outside available variable memory sizes, always remembering that, if an application is a cross-platform one, the different platforms/architectures will have different allowable sizes.

Tools like fuzzers usually trigger them and, as such, can be of great value to developers.

Finding situations in the flow of an application that are vulnerable to this type of bug is one of the most basic tasks that is done by an attacker.

14 Improper Authentication

Improper authentication describes a situation where, instead of validating a set of credentials, tokens, or other authentication mechanism, an application trusts a client that claims to have a given identity.

For example, in the following Node.js example, the validation is nothing but a check that a given user id exists:

```
const express = require('express');
const app = express();

app.use(express.json());

let userDB = [
    {id: 1, username: 'user1', password: 'pass1'},
    {id: 2, username: 'user2', password: 'pass2'}
];

app.get('/private_resource', (req, res) => {
    let userID = req.query.userID;
    let user = userDB.find(u => u.id === userID);

    if(user) {
        res.status(200).send('Access granted to private resource');
    } else {
        res.status(401).send('Unauthorized');
    }
});

app.listen(3000, () => {
    console.log('Server started on port 3000');
});
```

This is a flawed authentication mechanism because an attacker can simply guess or brute-force the user IDs to gain unauthorized access to the private resource. The server trusts the client's claim of identity (via the userID parameter) without any further authentication.

In a proper authentication system, the server should validate the client's credentials, such as a username and password, or a token that was issued upon successful authentication. It's important to note that authentication should be performed server-side, not client-side, to prevent attackers from bypassing it.

Unfortunately, this bug is very common when dealing with website connections that validate the presence of a cookie rather than validating the authentication to the site. A cookie can be forged to claim it has already been authenticated, and the website will trust it.

Another common scenario is when the authentication is validated client-side only – thus enabling an attacker to modify or interfere with the client software and gain access to server-side resources by claiming to already be authenticated.

15 Use of Hard-Coded Credentials

Issues classified under this vulnerability class originate from including credentials in a compiled package, or including them in the distribution media or storage device that reaches the end user. This means that, if discovered, those credentials would work on all the same devices. Rather than a coding bug per-se, this is more of an issue with packaging and releasing.

There are many ways this can happen – like the use of hard-coded test credentials during CI/CD that reach the deliverable at the end of the process (as opposed to using dedicated authentication tokens, for example), or including some form of backdoor to a production device that makes it to market.

Once found, this type of issue can be very difficult to solve. When it happens in a hardware device, the only way to remove a hard-coded credential is to update the firmware. How difficult that operation is will depend on the specific device.

But this can, and does, happen on software packages too – not just hardware. In this case, since the credential is baked into the actual software package, an updated version will usually be necessary to solve the problem.

It is also relatively easy to spot such types of credentials when reverse engineering software or hardware, so malicious actors really enjoy discovering them during their operations.

The real problem with hard-coded credentials is that they will overrule any access restrictions that you can set up in a given system. If a hard-coded admin credential exists, it doesn't matter if you have no other administrative accounts created – the hard-coded one will still be able to enter.

At an architectural level, hard-coded credentials also appear when configuring connections between systems, where a connection string or password are stored in a configuration file that is read to establish the connection. An attacker finding that file would be able to impersonate the connection.

Consider the following Java code:

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DatabaseConnection {

    public Connection getDBConnection() {
        Connection connection = null;
        try {
            String driverName = "com.mysql.jdbc.Driver";
            Class.forName(driverName);

            // Hard-coded credentials
            String serverName = "localhost";
            String schema = "myDatabase";
            String url = "jdbc:mysql://" + serverName + "/" + schema;
            String username = "admin";
            String password = "password123";
            connection = DriverManager.getConnection(url, username, password);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return connection;
    }
}
```

This Java code creates a connection to a MySQL database, but the database URL, username, and password are hard-coded into the program. These credentials are used to authenticate to the database server and access the myDatabase schema.

This is a bad practice because anyone who gains access to this source code will also have the database credentials. If this code were part of an application that is distributed to multiple users or systems, all of those instances would share the same database credentials.

You would want to avoid hard-coding credentials directly into your code. Instead, you could store them securely in environment variables or in a secure secrets manager, like the ones offered at the operating system level.

16 Missing Authorization

In #18 we will see how parts of an application could miss an authentication check. In this one, we look at how missing authorization can also lead to problems.

It is common for websites to protect sensitive sections behind authentication mechanisms, but each individual page has to enforce this check at the risk of direct access to a URL divulging information otherwise unreachable. This extends to downloads/uploads/queries and not just page accesses.

At the traditional application level, missing authorization happens when implicit trust is given to a user or external-calling application.

Like #17, the flaw usually stems from the architectural phase of development – not considering adequate security tactics to enforce proper access control mechanisms at all levels. Identifying and then correcting the code for a given application after this type of bug is identified usually carries a hefty workload price, as the code needs extensive refactoring to be properly secured after the fact.

At the operating system level, this type of issue occurs when a tool has no access control list associated with it, thus allowing any user to execute it.

Let’s look at this issue in more detail. What follows is a simple Python Flask sample:

```
from flask import Flask, redirect, url_for
from flask_login import login_required, current_user

app = Flask(__name__)

@app.route('/admin/dashboard')
```

```
@login_required
def admin_dashboard():
    # Missing authorization check here
    # Any authenticated user can reach this point
    return 'Welcome to the admin dashboard!'

if __name__ == "__main__":
    app.run()
```

While the check for being logged in exists (through the @login_required decorator), any logged in user will be able to access the admin dashboard.

This would add a layer of permissions' check:

```
@app.route('/admin/dashboard')
@login_required
def admin_dashboard():
    if current_user.is_admin:
        return 'Welcome to the admin dashboard!'
    else:
        return redirect(url_for('home'))
```

17 Improper Neutralization of Special Elements Used in a Command

Whenever any input is accepted by an application and then executed, if it is not properly checked, it could contain additional commands, parameters, or flags that modify its intended purpose.

If a system application is supposed to execute a command with a user-supplied parameter, a crafty user might include a command termination character followed by a different command that would then run with the same privilege level as the original program. These types of issues are closely related to "SQL injection" and "improper control of generated code" bugs, as they all come from implicitly trusting inputs.

The source of the problem usually lies at the architectural level, where security concerns are not considered adequately, thus introducing insecure behaviors in applications. As a rule of thumb, no input should ever be trusted as good – ideally even at the individual function level, but at the very least at the application level.

One very recent example of such a bug was found to exist, and be actively exploited by threat actors, inside the code of Barracuda's Email Security Gateway appliance (CVE-2023-2868).

A code section that dealt with analyzing email attachments contained the following perl code:

```
"qx()" from perl:qx{$tarexec -O -xf $tempdir/parts/$part '$f'};
```

"qx" is a function that executes a given command in the shell. In Perl, it is functionally equivalent to enclosing a command in backticks.

The bug in this situation is the use of "$f" without any type of check to ensure that it does not contain, as it happened in this situation, other shell executable commands. In this specific case, the value of $f would be the filename of a file inside a ".tar" email attachment. Crafting a filename that included a malicious payload when processed this way would, and did, lead to a full system compromise.

18 Missing Authentication for a Critical Function

Any functionality-changing feature should validate if the actor performing the action is entitled to do so or not, and prevent the action when the actor is not allowed to do so. When an application fails to validate this, then it falls into this category.

Applications will often perform tasks like adding, removing, modifying, or deleting data, users or connections. All it takes is one of those actions to be missing, such an authentication check for the whole application, for it to be compromised or unreliable. And there are several steps involved in each of those operations, so the validation should happen ideally at each step or, barring that, at the entry points for each process.

```java
public class User {
    private String username;
    private String password;
    // other properties...

    // constructor, getters and setters...
}

public class App {
    public static void main(String[] args) {
        // Creates a user object
        User user = new User();
        user.setUsername("admin");
        user.setPassword("password");
        // Checks if the user input matches the hardcoded values
        if (user.getUsername().equals("admin") && user.getPassword().equals("password")) {
            performCriticalAction();
        } else {
            System.out.println("Access denied. Invalid username or password.");
        }
    }

    public static void performCriticalAction() {
        // Here is where a critical action occurs. For example, altering sensitive data.
        System.out.println("Critical action performed.");
    }
}
```

There are multiple issues present in this example. The hard coded values are a problem, but also the fact that any user matching those credentials could execute the CriticalAction(). There should be a check to ensure that the given user was in fact authorized to perform the action, rather than simply using a (flawed) check for log in.

Another typical example is missing authentication checks in cloud-based storage (ie, “leaky S3 buckets”), where confidential data is accidentally stored in a publicly accessible way.

19 Improper Restriction of Operations within the Bounds of a Memory Buffer

Usually referred to as “buffer overflow” or “buffer overrun,” this anomaly attempts to read a position in memory outside of the pre-arranged space for a given variable, or write to such a location.

This can happen either by intentionally passing a memory location that falls outside of the buffer or by using the result of an operation as the position to read/write without validating if it is acceptable or not in the given context.

Take the following example:

```
int main (int argc, char **argv) {

char *items[] = {"A", "B", "C", "D"};
int index = AskUserForIndex();

printf("You selected %s\n", items[index-1]);
}
```

(Code sample adapted from the one present at https://cwe.mitre.org/data/definitions/119.html)

In this example, if the user enters “5” as the index, the program will happily comply and return the content of a memory location outside the array. Depending on the operating system and memory architecture, several things can happen: memory values outside the program space can be returned, the program can crash, another program can crash (if a write happens), the system state can be compromised, or nothing crashes and the attack can be repeated. It is possible to read large amounts of memory by repeating the process with different index values.

This class of bugs can create problems when reading, but also when writing values to a buffer – by writing more data than the buffer was created to accommodate. If the memory region right after the buffer happens to hold a flag that indicates if the user has privileges or not, a simple byte change could turn a regular user into an administrator.

While this type of code bug is usually taught in programming classes, as well as how to avoid it by placing checks before any memory access, it sometimes isn’t obvious where the problem happens in the first place. Static code analysis tools, peer review, and pair programming are all ways to try and catch these before they make it into production code.

Another related concept in programming is the “canary” (analogous to a canary in a coal mine), where a value is written right after the buffer and then checked for its presence after a value is written to the buffer. If the original value is no longer there, then the programmer can assume a buffer overflow happened and intentionally crash the application to prevent further damage or deal with it by alerting the user or some other mechanism. While useful, clever attackers can work around these by including the canary value as part of the new value in order to not trigger it.

20 Incorrect Default Permissions

This class of bugs comes from the way software packages are prepared – either at the operating system level or the application level – by including permissions for a subset of items that is too broad in scope. Usually, this refers to executable files that have permissions that enable unprivileged users to execute said files when they shouldn’t be able to or having those files run with such privileges that they can manipulate other components of the system in unintended ways.

It can happen the other way around as well: having a permission on an executable file that is too restrictive, and then having the operating system automatically find an alternative located in an attacker-controlled location (rather than the component included in the original package).

The main risk is that these are the default permissions, so as soon as the package is deployed it will immediately be at risk without requiring any other action. This shifts the risk vector from usage to deployment.

This is exemplified in the following “installation” script:

```
# A software package installation script might look like this
echo "Installing package..."

# Creating a new file
touch /usr/local/mysoftware/config.txt

# Setting incorrect, overly permissive default permissions
chmod -R 777 /usr/local/mysoftware/

echo "Installation complete."
```

The default permissions set for the application folder would trickle to the configuration file, which would then have read, write, and execute permissions for all users. If it contained any sensitive information, it would be available for all users to peruse.

21 Server-Side Request Forgery (SSRF)

Servers can be tricked into sending responses to queries to other destinations different from the original requesters' address.

As it will described later, in bug #24, it is possible to have an application access other content by including URIs in specially crafted XML files. If those files reside in a third-party location, then the request will appear to come from the server processing the XML file rather than the attacker's system. While not the only bug that falls under this category, it is a clear example of a server/application being tricked into starting a connection with a third-party system.

Any input that can contain a list of URIs that are not properly validated can be used as an SSRF mechanism.

For example, the M3U file format can include references to external files. If you had a video encoding web service that accepts an M3U file as the content list, it would be possible to include a reference to system files otherwise inaccessible and have those included as part of the output of the web service.

The following Python Flask sample illustrates the problem. It accepts a url as a parameter and makes a request to it without any validation. An attacker could potentially abuse this by retrieving content that is reachable by the server but not directly by the attacker (ie, for example, internal network restricted content).

```
from flask import Flask, request, redirect
import requests

app = Flask(__name__)

@app.route('/proxy')
def proxy():
    url = request.args.get('url')
    response = requests.get(url)
    return response.content

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=8000)
```

22 Concurrent Execution Using Shared Resource with Improper Synchronization ("Race Condition")

When multiple applications try to use the same resource, or even multiple threads inside the same application, some form of coordination has to take place to make sure that each one will have the access it needs for as long as it needs to have it.

If not, it can lead to situations where one application will start writing to a file at the same time as another application and in the end you either have a mix of the content that you were supposed to have in the file, only the content from one of the applications, or even just a blank file.

As this type of situation is very load dependent, it is also difficult to reproduce and identify adequately.

It can affect any type of resource, not just files. For example, it can impact in-memory data, networking, databases – so much so that many language-specific constructs have been added over time to address this situation with the ultimate goal of creating a type of access called "atomic", which is guaranteed to happen completely or not at all, thus making sure no partial reads/writes occur.

Something as simple as increasing a variable value (x++) is actually prone to this type of issue, as the actual operation is broken down into multiple machine level operations (reading the original value of x, incrementing it, assigning the new value back to x). If x was a variable shared by multiple threads inside an application, then it could lead to bugs if multiple threads tried to increment it at the same time.

The following sample showcases the problem. Please note that the result can depend on the actual system, load, architecture, or other parameters where it is tested:

```
using System;
using System.Threading;

namespace RaceCondition
{
    class Example
    {
        int result = 0;

        void Work1() { result = 1; }
        void Work2() { result = 2; }
        void Work3() { result = 3; }

        static void Main(string[] args)
        {
            Example ex = new Example();

            Thread worker1 = new Thread(ex.Work1);
            Thread worker2 = new Thread(ex.Work2);
            Thread worker3 = new Thread(ex.Work3);

            worker1.Start();
            worker2.Start();
            worker3.Start();

            Console.WriteLine(ex.result);
            Console.Read();
        }
    }
}
```

Language constructs like mutexes, semaphores, and other thread synchronization mechanisms were introduced specifically to address this type of problem. Based on the prevalence of vulnerabilities that comes from this class of bugs, the successful implementation of such constructs may be less than ideal.

23 Uncontrolled Resource Consumption

All resources available to an application are limited: processing power, storage, network bandwidth, memory capacity, database connections, etc.

If the application can be tricked into abusing one or more of those resources, then it can exhaust them and make the application, or in some cases the whole system, appear unresponsive or outright unusable.

It can appear in code as compute-intensive functions not being guarded against repeated calls (while previous calls haven't completed yet), or even functions that generate a disproportionate amount of data from very small inputs (for example, complex error messages from single character mistakes), which are both potential sources for this type of bug.

Another common manifestation of this bug is when an application does not dispose of resources it allocated, leading to out-of-memory errors or something similar.

This is exemplified by the following C# code:

```
using System;
using System.Collections.Generic;

namespace UncontrolledResourceConsumption
{
    class Program
    {
        static void Main(string[] args)
        {
            List<byte[]> memoryHog = new List<byte[]>();

            try
            {
                while (true)
                {
                    // Allocate 1 MB of memory at a time.
                    memoryHog.Add(new byte[1024 * 1024]);
                }
            }
            catch (OutOfMemoryException)
            {
                Console.WriteLine("Out of memory!");
            }
        }
    }
}
```

In this example, the application continuously creates new byte arrays of size 1 MB and adds them to a list. Because nothing is ever removed from the list, and because the garbage collector cannot free up the memory used by the byte arrays (since they are still referenced by the list), this program will eventually consume all available memory and throw an OutOfMemoryException.

This is a clear case of uncontrolled resource consumption, as the application continuously consumes memory without any mechanism to limit or control this consumption.

This is a particularly egregious type of bug that is often hard to diagnose before an application reaches the production stage – possibly due to improper testing – and one that can potentially leave an application in a crashed or, even worse, unexpected state.

Most static code analysis tools will have checks in place to detect this type of situation, but as code can be written in many different ways, it is always possible to have such convoluted code that these analysis tools will be unable to detect it.

24 Improper Restriction of XML External Entity Reference

XML files can contain references to external documents as part of their structure, but this can lead an application to load documents outside the intended scope. This can potentially result in path traversal problems, improper file accesses, or wrong data being taken as good by the application.

This applies to URIs (Unique Resource Identifiers) that point to other local files or even to files on the Internet by using file:// or http:// URIs.

In addition to allowing access to otherwise unreachable files, it can also be used to force the application to process inordinate amounts of data by feeding it unexpectedly large documents, slowing it down, or even making such applications unresponsive.

If the application does not guard against it, it becomes possible to exploit a system by crafting an XML file like:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "file:///etc/passwd"> ]>

<stockCheck>
<productId>&xxe;</productId>
</stockCheck>
```

And then observing the result.

[This example comes from https://portswigger.net/web-security/xxe.]

If the target server is able to access other internal systems that an attacker would normally not be able to access, then this bug could be leveraged as a stepping stone to obtain information from those systems.

25 Improper Control of Generation of Code ('Code Injection')

This is a common issue where a programmer does not provide sufficient defensive code in regards to user-fed inputs – that is, the software uses whatever input it receives without sanitizing it and then uses said input, as-is, to feed internal functionality.

At a very low level, specially crafted inputs can change the way the program operates in unforeseen ways, letting users have more control over the program, and in some cases the whole system, as a result.

This leads to situations where input is carefully constructed to introduce undesired side effects, like data corruption or unintended program flow changes.

A very common special case of this type of bug is “SQL injection,” which merits a category of its own in this book, but is nothing more than a special case of code injection.

As an example, the use of “eval()” in PHP is often a code “smell” that points to a possible code injection situation if the value passed is not properly validated beforehand:

```php
<?php
    // Ask the user to enter a number
    echo "Enter a number: ";
    $input = fgets(STDIN);

    // Evaluate the input
    $result = eval("return $input;");

    // Print the result
    echo "The result is: " . $result;
?>
```

In this example, the user input is not sanitized or validated before it is passed to eval(). This means a malicious user could enter PHP code that causes harmful side effects. For instance, a user could enter:

```
 "; system('ls -la /'); //"
```

as their input, which would attempt to list all files on the server. The ; symbol is used to end the previous command, the system() function is used to execute a shell command, ls -la / is a command that lists all files on a Unix-like system, and // is used to comment out the rest of the line.

This is a clear case of code injection, and it's why using eval() with user input is extremely dangerous and generally advised against. Never run the code sample above in a production environment. Setting up reachable code in such a way is simply an example of what not to do and should never be used otherwise.

The Runner-Ups and Honorable Mentions

These are some bugs that didn't make the Top 25, but are still either interesting in the way they are defined or behind some memorable (and costly) vulnerabilities.

01 Improper Certificate Validation

Improper certificate validation occurs when software that communicates over a network with encrypted connections does not properly verify the certificate offered by the target. This can allow attackers to eavesdrop on or modify a supposedly secure communication.

At a high level, this exposes the application to Man-In-The-Middle attacks, Server Impersonation, and DNS hijacking, since the certificate validation that ensures the server is who it is claiming to be fails to execute or executes improperly.

Consider the following Python code that uses the requests library to send an HTTP request:

```python
import requests

response = requests.get('https://example.com', verify=False)
```

In this example, the verify=False argument tells the requests library not to verify the SSL certificate of the server it is connecting to. This means that it will accept any certificate presented by the server, including a self-signed certificate or a certificate signed by an unknown authority. This makes it vulnerable to a "man-in-the-middle" (MITM) attack, where an attacker intercepts the secure communication.

To mitigate this issue, always verify SSL certificates when making secure network requests:

```python
import requests

response = requests.get('https://example.com')
```

In this revised code, the verify argument is left at its default value of True. This means that the requests library will verify the server's SSL certificate against a trusted certificate authority. If the certificate is not valid, it will raise an exception and refuse to establish the connection.

It is not uncommon to turn off certificate validation during the development process, so that the testing environment does not need to deploy certificates and becomes more flexible and easy to deploy, but this leads to then shipping the code and forgetting to restore the certificate checking process. Adding certificates to testing environments is the preferred method, and can be automated with traditional system management tools.

Please note, however, that ensuring secure communications involves more than just enabling certificate verification. It is also necessary to keep the list of trusted certificate authorities up to date, to check the certificate's validity period, and to verify that the certificate matches the server it was received from.

02 Uncontrolled Search Path Element

Uncontrolled search path element vulnerabilities occur when a product uses a fixed or controlled search path to locate resources, but one or more locations in that path can be manipulated by an unintended actor.

This issue frequently arises when a product uses a directory search path to locate executables or code libraries. If the path contains a directory that can be modified by an attacker, the product may inadvertently load and execute malicious code. This issue can also occur when software package management frameworks search public repositories before private ones, enabling an attacker to substitute a malicious package with the same name as a desired package.

This can be exemplified by library loading where only the library name is specified, rather than the whole path to the library. This is done usually when deploying applications to heterogeneous environments, where standardized locations are not available or are difficult to ascertain.

However, it can also apply to uncontrolled execution of code, as in the following C example:

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    system(argv[1]);
    return 0;
}
```

In this example, the program takes a command-line argument and passes it to the system function. This function uses the system's PATH environment variable to locate and execute the command. An attacker could manipulate the PATH variable or the command-line argument to cause the system function to execute a malicious command.

To mitigate this vulnerability, use fully qualified paths for all file operations. Additionally, ensure that the search order of directories prioritizes trusted locations over uncontrolled ones:

```c
#include <stdio.h>
#include <stdlib.h>
```

```
int main(int argc, char **argv) {
    char command[256];
    snprintf(command, sizeof command, "/usr/bin/%s", argv[1]);
    system(command);
    return 0;
}
```

In this modified code, the program appends the command-line argument to /usr/bin/ to form a fully qualified path, ensuring that the system function executes the command from the intended location. While this modification alone does not cover all possible scenarios, it should provide an idea of how to proceed in fixing existing problems.

03 Missing Release of Memory after Effective Lifetime (aka Memory Leak)

A memory leak occurs when a program allocates memory but fails to release it back to the operating system after it has finished using it. Over time, memory leaks can consume enough resources to slow down or crash the system, making them a critical issue in long-running programs or services.

Consider the following C code snippet:

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

void create_array() {
    int *temp_array = malloc(100 * sizeof(int));
    // Code that uses temp_array...
}

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

In this example, the function create_array allocates memory for an array of 100 integers using the malloc function. However, it does not free this memory before it returns. As a result, that chunk of memory becomes unreachable, leading to a memory leak.

To fix this issue, you need to ensure that you release any memory you've allocated once you've finished using it. You can do this using the free function:

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

void create_array() {
    int *temp_array = malloc(100 * sizeof(int));
    // Code that uses temp_array...
    free(temp_array);
}

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

In this updated code, the function create_array still allocates memory for the array. However, it now also releases that memory before it returns, preventing the memory leak.

Even with this simple example, tracking all the memory allocations and deallocations can get complicated in larger codebases. Using smart pointers in languages that support them or garbage-collected languages can help manage memory automatically and avoid these issues.

Note that these leaks can happen at the resource level as well (for example, storing images in memory and not releasing them after use), or even indirectly in libraries that your code imports. In addition to properly ensuring that your code does not contain such situations that can lead to leaks, it is also important to keep track of any arising issues or bug fixes in relevant libraries that indicate the presence of memory leaks in versions you are specifically relying upon.

04 Insufficiently Protected Credentials

Insufficiently protected credentials occur when a product transmits or stores authentication credentials in an insecure way, making it susceptible to unauthorized interception and retrieval. This can lead to breaches of sensitive data, unauthorized system access, and potential identity theft.

Consider the following Java code snippet that stores and transmits a password in plaintext:

```
public class User {
    private String username;
    private String password;

    public User(String username, String password) {
        this.username = username;
        this.password = password;
    }

    public void sendCredentials() {
        // An imaginary function that sends the username and password over the network
        Network.send(this.username, this.password);
    }
}
```

In this example, the password is stored and transmitted in plaintext. If an attacker can intercept the transmission or gain access to the stored password, they can obtain the user's credentials.

To mitigate this issue, you should store and transmit passwords in a securely hashed and salted form. Additionally, you should use secure, encrypted connections when transmitting credentials. Below is a safer version of the above code:

```java
import java.security.*;
import javax.crypto.*;
import javax.crypto.spec.*;
import java.util.Base64;

public class User {
    private String username;
    private String hashedPassword;

    public User(String username, String password) throws Exception {
        this.username = username;
        this.hashedPassword = hashPassword(password);
    }

    public void sendCredentials() {
        // An imaginary function that sends the username and hashed password over a secure,
encrypted connection
        Network.sendSecure(this.username, this.hashedPassword);
    }

    private String hashPassword(String password) throws NoSuchAlgorithmException {
        MessageDigest md = MessageDigest.getInstance("SHA-256");
        byte[] hash = md.digest(password.getBytes());
        return Base64.getEncoder().encodeToString(hash);
    }
}
```

In the updated code, the user's password is hashed using the SHA-256 algorithm before it's stored in the User object, and Network.sendSecure is an imaginary function that would use secure means (such as SSL/TLS encryption) to transmit the hashed password. This reduces the risk of the user's credentials being intercepted and used by an attacker. Further strengthening is always desired, like salting all the data with user or channel independent salt values, for example.

05 Cleartext Storage of Sensitive Information

Cleartext storage of sensitive information happens when a product stores data such as passwords, credit card numbers, health records, and other sensitive information in unencrypted, plain text format. This leaves it readable and accessible to anyone who can gain access to the storage location.

Consider the following example in Python that stores user credentials in a dictionary and writes it to a file in plaintext:

```python
user_credentials = {'username': 'user1', 'password': 'password123'}

with open('credentials.txt', 'w') as f:
    for key, value in user_credentials.items():
        f.write('%s:%s\n' % (key, value))
```

In this code, the user's username and password are stored in plaintext in a text file. If an attacker gains access to this file, they can easily read the user's credentials.

To avoid this issue, sensitive data should never be stored in cleartext. Instead, use secure methods for storing sensitive data, such as encryption or hashing. Also, it's advisable to use established security libraries or modules for such tasks, as they are likely to follow best practices for data protection.

Here is a safer version of the above code using the cryptography library to encrypt the data before storing it:

```
from cryptography.fernet import Fernet

# key should be kept secret
key = Fernet.generate_key()
cipher_suite = Fernet(key)

user_credentials = {'username': 'user1', 'password': 'password123'}

# Encrypting the user's credentials before writing them to a file
with open('credentials.txt', 'wb') as f:
    for key, value in user_credentials.items():
        encrypted_value = cipher_suite.encrypt(value.encode())
        f.write('%s:%s\n' % (key, encrypted_value))
```

In this code, the user's credentials are encrypted before being written to the file, protecting them even if an attacker gains access to the file.

It's also essential to protect the encryption key used in this process. If the key is compromised, an attacker could decrypt the sensitive information. This means that encrypting the data, even when stored in a database, but leaving the encryption key or secret used in the process, is functionally equivalent to not using any encryption at all.

Conclusion

As a takeaway, it is crucial for developers to be aware of the common code bugs that can lead to vulnerabilities in their applications. By understanding these issues and implementing secure coding practices, you can significantly reduce the risk of exploitation by attackers.

Here are some general secure coding principles to keep in mind:

- **Least Privilege**

  Ensure that your applications run with the minimum privileges necessary to complete their tasks. This limits the potential impact of a security breach by reducing the attacker's ability to escalate their privileges or gain unauthorized access to sensitive data or system resources.

- **Secure Defaults**

  Design your applications with security settings enabled by default. Users should have to explicitly disable security features rather than enable them. This encourages a more secure user experience and reduces the likelihood of insecure configurations.

- **In-Depth Defense**

  Implement multiple layers of security in your applications, so that if one layer is breached, others can still provide protection. This can include using input validation, output encoding, and access controls in combination to provide a comprehensive defense against various attacks.

- **Fail Securely**

  Design your applications to fail securely in the event of an error or unexpected input. For example, if an application encounters an error while processing a user's input, it should not reveal sensitive information or grant unintended access. Instead, it should handle the error gracefully and provide a useful error message to the user.

- **Keep it Simple**

  Complexity is the enemy of security. Strive for simplicity in your code and minimize the use of complex constructs that can be difficult to understand and maintain. This makes it easier to spot potential security issues and reduces the likelihood of introducing new vulnerabilities during development or maintenance.

- **Regularly Update Dependencies**

  Keep your application's dependencies up to date, as outdated libraries and frameworks can introduce security vulnerabilities. Regularly check for updates and patches, and incorporate them into your development and deployment processes.

By incorporating these secure coding principles and addressing the common code bugs mentioned in this series, you will be well on your way to developing more secure applications and protecting your users from potential security threats.