

A decorative graphic consisting of thin, grey, stylized circuit lines with small circles at the ends, extending horizontally from the left and right sides of the central black box.

INFORMATION SECURITY INTRODUCTION

HAUTE-ÉCOLE LÉONARD DE VINCI

CHAPTER 3 – SECURE CODING – PART 2

A decorative graphic consisting of thin, grey, stylized circuit lines with small circles at the ends, extending horizontally from the left and right sides of the central text box.

3.4 SOFTWARE SECURITY TESTING



3.4 SOFTWARE SECURITY TESTING

No matter how well talented, the development team for an application is, there will be some form of flaws in the code.

In 2021, around 80% of the application tested by Vericode showed *at least* one security issue.

Source: <https://www.veracode.com/state-of-software-security-report>

3.4 ANALYSING & TESTING CODE

The source code can contain a variety of bugs and flaws, business logic, error handling, integration with other services and systems.

It is vital to be able to analyze the code.

The code analysis and testing is often performed via:



Static code
Analysis



Dynamic
code
Analysis



Fuzzing

3.4 STATIC CODE ANALYSIS

Static code analysis is a technic meant to help identify software defects or security policy violations and is carried out by examining the code without executing the program (hence the word *static*).

Static code analysis is an automated process! A human reviewing the code is named a *code review*.

3.4 DYNAMIC ANALYSIS

This type of analysis does not focus on what the binary *is*, but rather focus on what the binary *does*.

Dynamic analysis can detect dependencies that are not possible to detect in static analysis (dynamic dependencies). It helps analyse if part of code is obfuscated. It provide a timeline of behavior.

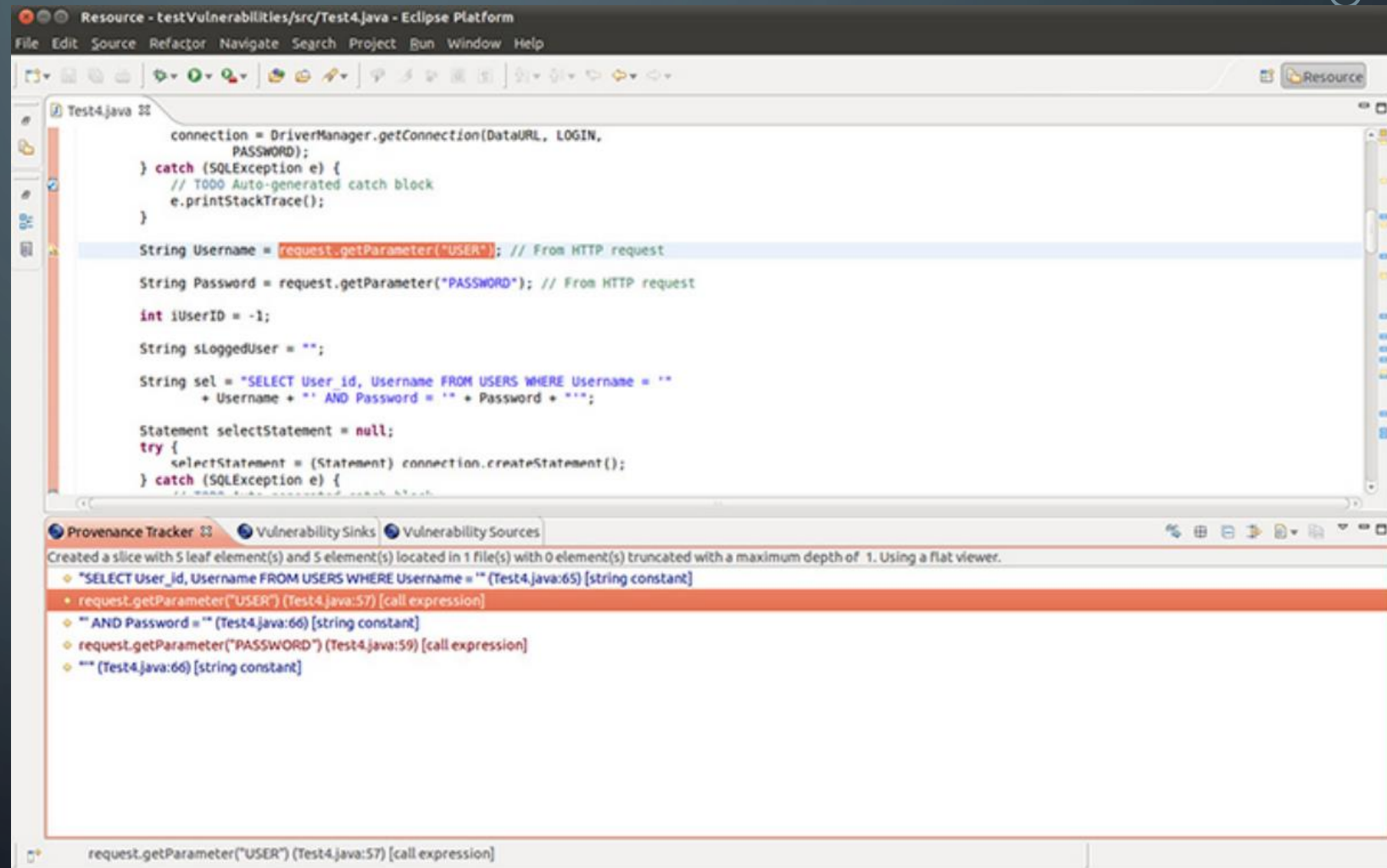
The biggest disadvantages is that dynamic analysis can only reveal what the binary has done, not what all the binary can do.

For example: Valgrind is a dynamic analysis tool for C language.

3.4 STATIC CODE ANALYSIS

Lapse+ is developed by OWASP to find vulnerabilities in Java applications.

The tool is highlighting usage of an input without sufficient validation to create an SQL query



The screenshot shows the Eclipse IDE with a Java file named `Test4.java`. The code contains a database connection and a SQL query. The query is: `"SELECT User_id, Username FROM USERS WHERE Username = " + Username + " AND Password = " + Password + ""`. The `request.getParameter("USER")` call is highlighted in red. Below the code editor, the `Provenance Tracker` window is open, showing a list of elements. The first element is `"SELECT User_id, Username FROM USERS WHERE Username = " (Test4.java:65) [string constant]`. The second element is `request.getParameter("USER") (Test4.java:57) [call expression]`. The third element is `"AND Password = " (Test4.java:66) [string constant]`. The fourth element is `request.getParameter("PASSWORD") (Test4.java:59) [call expression]`. The fifth element is `"" (Test4.java:66) [string constant]`. The status bar at the bottom shows `request.getParameter("USER") (Test4.java:57) [call expression]`.

```
Resource - testVulnerabilities/src/Test4.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help

Test4.java
connection = DriverManager.getConnection(DataURL, LOGIN,
    PASSWORD);
} catch (SQLException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}

String Username = request.getParameter("USER"); // From HTTP request

String Password = request.getParameter("PASSWORD"); // From HTTP request

int iUserID = -1;

String sloggedUser = "";

String sel = "SELECT User_id, Username FROM USERS WHERE Username = "
    + Username + " AND Password = " + Password + "";

Statement selectStatement = null;
try {
    selectStatement = (Statement) connection.createStatement();
} catch (SQLException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}

Provenance Tracker Vulnerability Sinks Vulnerability Sources
Created a slice with 5 leaf element(s) and 5 element(s) located in 1 file(s) with 0 element(s) truncated with a maximum depth of 1. Using a flat viewer.
* "SELECT User_id, Username FROM USERS WHERE Username = " (Test4.java:65) [string constant]
* request.getParameter("USER") (Test4.java:57) [call expression]
* "" AND Password = " (Test4.java:66) [string constant]
* request.getParameter("PASSWORD") (Test4.java:59) [call expression]
* "" (Test4.java:66) [string constant]

request.getParameter("USER") (Test4.java:57) [call expression]
```


3.4 FUZZING

Fuzzing is a technique used to discover flaws and vulnerabilities in software by sending large amounts of malformed, unexpected, or random data to the target program in order to trigger failures.

The purpose is to identify how the software behave to avoid having an attacker being in a position to manipulate these errors and flaws to inject their own code and compromise the system/software.

Fuzzing tools helps to identify buffer overflows, denial of service, injection weakness, crash or throw unexpected error.

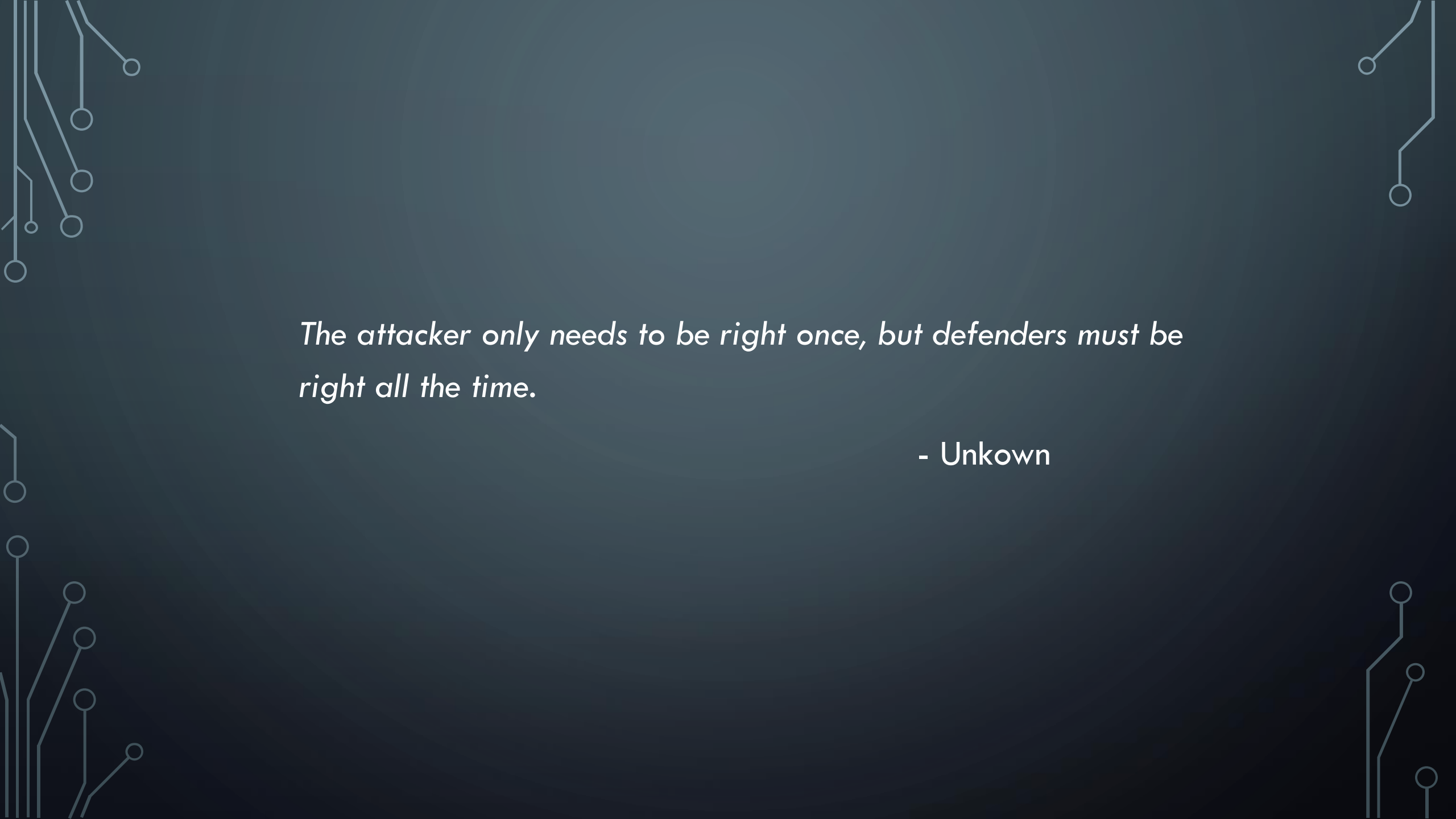
3.4 FUZZING

Multiple Fuzzing software exists :

- AFL & AFL++: American Fuzzy Lop
- Untidy: XML fuzzer used to test WebApp clients and server
- Peach Fuzzer: Fuzzer framework (<https://wiki.mozilla.org/Security/Fuzzing/Peach>)

A decorative graphic consisting of thin, grey, stylized circuit lines with small circles at the ends, extending horizontally from the left and right sides of the central black box.

3.5 APPLICATION SECURITY CONTROLS

The background is a dark blue gradient. In the corners, there are decorative white line art elements resembling circuit boards or neural networks, with lines and small circles.

The attacker only needs to be right once, but defenders must be right all the time.

- Unkown



100%

OF WHAT IS ON
INTERNET WILL BE
ATTACKED

3.5 APPLICATION SECURITY CONTROLS

Although vulnerabilities affecting applications are a significant source of concern for cybersecurity, the good news is that there are number of tools available to assist in development of a *defense-in-depth* approach to security.

A combination of secure coding practices and security infrastructure tools can create robust defenses against application exploits.

3.5 INPUT VALIDATION

Any application that allow user input must perform validation of that input to reduce the likelihood that it contains an attack.

Improper input handling practices can expose applications to injection attacks, XSS and other exploits.

To achieve this, developers uses:

Input whitelisting & Input blacklisting

3.5 INPUT WHITELISTING

The most effective form of input validation is *Input Whitelisting* in which the developer describes the exact type of input that is expected from the user and then verifies that the input matches the specification before passing the input to other resources.

Example: If an input form prompts a user to enter their age, input whitelisting could verify that the user supplied an integer value within the range [0-120] and reject any other values outside that range.

3.5 INPUT BLACKLISTING

It is often difficult to implement Input Whitelisting. For example a search field in a merchant website.

Developers can decide to implement *Input Blacklisting* to control user input. To disallow HTML tags or SQL commands or forbid any special character like the single quote.

This can create undesired side-effect, rendering impossible search for some product with a single quote.

3.5 INPUT VALIDATION

Perform input validation using libraries, not your own code!

Do your input validation SERVER SIDE.

3.5 DATABASE SECURITY

Secure applications depends on secure databases to provide the content and transaction processing necessary to support business operations.

Relational database form the core of most modern applications, and securing these databases goes beyond just protecting them against SQL injection attacks.

3.5 PARAMETERIZED QUERIES

Parameterized queries are designed to protect applications against injection attacks. In a parametrized query, the client does not directly send SQL code to the database server. Instead, the client sends arguments to the server, which then inserts those arguments into a precompiled query *template*.

This approach protects against injection attacks and also improve database performance!

3.5 PARAMETERIZED QUERIES

```
# BAD EXAMPLE. DON'T DO THIS!
def is_admin(username: str) -> bool:
    with connection.cursor() as cursor:
        cursor.execute("""
            SELECT
                admin
            FROM
                users
            WHERE
                username = '%s'
            """ % username)
        result = cursor.fetchone()

        if result is None:
            # User does not exist
            return False

        admin, = result
        return admin
```

```
def is_admin(username: str) -> bool:
    with connection.cursor() as cursor:
        cursor.execute("""
            SELECT
                admin
            FROM
                users
            WHERE
                username = %(username)s
            """, {
                'username': username
            })
        result = cursor.fetchone()

        if result is None:
            # User does not exist
            return False

        admin, = result
        return admin
```

3.5 OBFUSCATION AND CAMOUFLAGE

Maintaining sensitive personal information in database exposes an organization to risk in event that information is stolen by an attacker. Measures should be taken to avoid *data exposure*.



Minimization



Tokenization



Hashing

3.5 DATA MINIMIZATION

Data Minimization is the best defense. You should not collect sensitive information that you don't need and should dispose of any sensitive information that you do collect as soon as it is no longer needed for legitimate business purpose.

Data minimization reduces the risk because you can't lose control of information that you don't have in the first place!

3.5 TOKENISATION

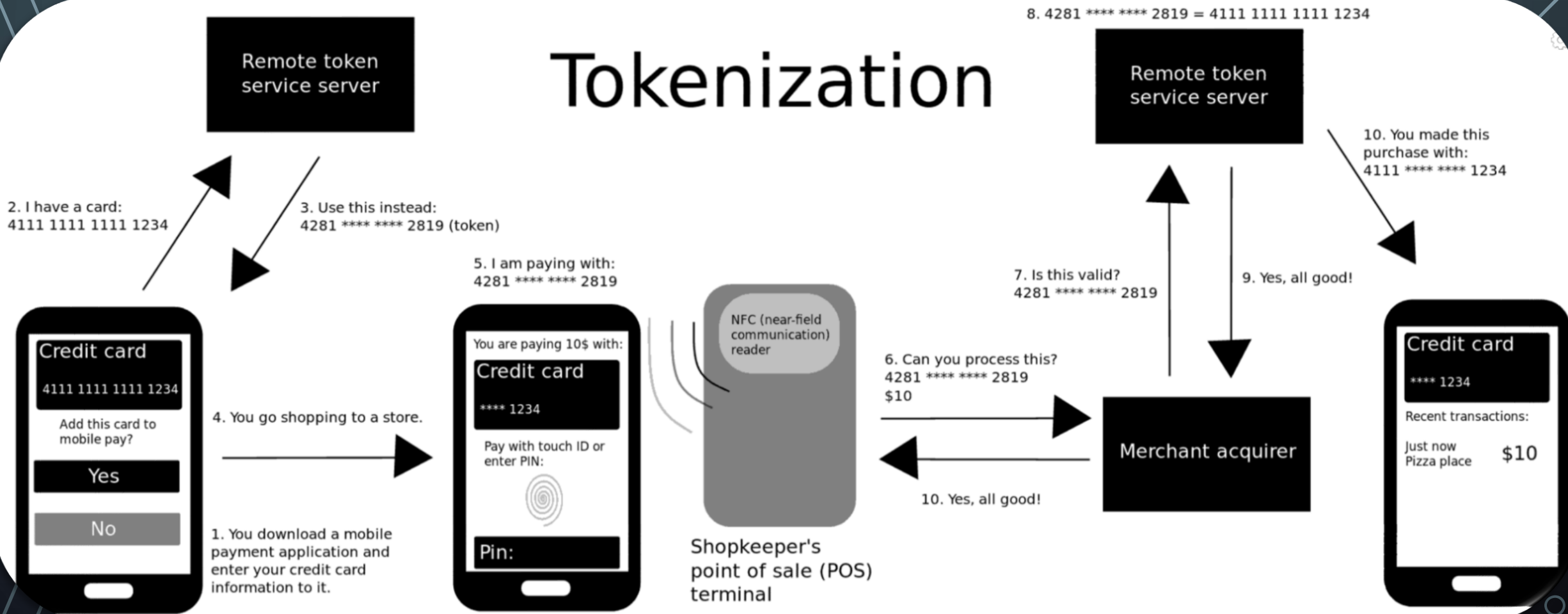
Tokenization replaces personal identifiers that might directly reveal an individual's identity with a unique identifier using a lookup table. For example, we might replace a widely known value, such as student ID, with a randomly generated 10-digit number.

We would then maintain a lookup table that allows us to convert those back to student IDs if we need to determine someone's identity.

Of course, if you use this approach, you need to keep the lookup table secure!

Tokenization is used by payments systems!

Tokenization



3.5 HASHING

Hashing uses a cryptographic hash function to replace sensitive identifiers with an irreversible alternative identifier.

Salting these values makes these hashed values resistant to a type of attack known as a rainbow table attack.*

**We will learn about hashing and salt in the chapter 4.*

3.5 ERROR HANDLING

Attackers thrive on exploiting errors in code.

Developers must understand this and write their code so that it is resilient to unexpected situations that an attacker might create to test the boundaries of code.

For example, it's insufficient to simply verify that the age of a person is an integer. Attackers might enter a 50,000-digit integer in that field in an attempt to perform an integer overflow attack.

Developers **MUST** ANTICIPATE unexpected situations and write error handling code that handles these situations in a secure fashion.

3.5 ERROR HANDLING

Perform proper logging information about errors in the application log files not in the error message displayed to the user (attacker).

Error!

Fuel\Core\Database_Exception [Error]: SQLSTATE[HY000]: General error: 2601 General SQL Server error: Check messages from the SQL Server [2601] (severity 14) [INSERT INTO products (descr,productgroup_id) VALUES ('Muffins','106')] with query: "INSERT INTO products (descr,productgroup_id) VALUES ('Muffins','106')"

COREPATH/classes/database/pdo/connection.php @ line 137

```
132         // This benchmark is worthless
133         Profiler::delete($benchmark);
134     }
135
136     // Convert the exception in a database exception
137     throw new \Database_Exception($e->getMessage().' with query: "'.$sql.'"');
138 }
139
140 if (isset($benchmark))
141 {
142     Profiler::stop($benchmark);
```

Backtrace

1. COREPATH/classes/database/query.php @ line 248

Backtrace

3.5 HARDCODED CREDENTIALS

Far too often, credentials (or private key !) are hardcoded or embedded within the code. It's a huge security concern.

Sometimes these credentials are used by developer as a backdoor to the application, bypassing the normal authentication process.

Another problem is when the code is pushed on a repository and becomes public by error...

3.5 HARDCODED CREDENTIALS

The screenshot shows a GitHub search interface for the query "password ftp". The left sidebar lists various repository categories: Repositories (0), Code (416K+), Commits (50K), Issues (0), Discussions (Beta, 0), Packages (1), Marketplace (0), Topics (0), Wikis (5K), and Users (0). The main content area displays "Showing 416,228 available code results" and a "Sort: Best match" dropdown. A repository titled "7h3rAm/writeups" is highlighted, showing a file path: "vulnhub.bsidesvancouver2018workshop/results/192.168.92.169/scans/tcp_21_ftp_hydra.txt". The code snippet within this file shows the output of a Hydra brute-force attack on an FTP server. The first line is a comment: "# Hydra v8.6 run at 2019-09-10 11:08:38 on 192.168.92.169 ftp (hydra -L /usr/share/seclists/Username/top-usernames-shortlist.txt -P /usr/share/seclists/Passwords/darkweb2017-top100.txt -e nsr -s 21 -o /root/toolbox/vulnhub/bsidesvancouver2018workshop/results/192.168.92.169/scans/tcp_21_ftp_hydra.txt)". The subsequent lines show successful login attempts with the username "ftp" and various passwords: "password: ftp", "password: ptf", and "password: 123456".

```
5 heroku config:add HUBOT_GITHUB_TOKEN="6d78678d408"
6 heroku config:add HUBOT_SLACK_BOTNAME="Test-Sl"
7 heroku config:add HUBOT_SLACK_TEAM="Dep1"
8 heroku config:add HUBOT_SLACK_TOKEN="xoxp-2446364950-2445644913-257783a"
```

3.5 APPLICATION SECURITY CONTROLS

Other security controls and development good practice exists:

- Web Application Firewalls (WAF)
- Code signing
- Memory management (Resource exhaustion, Buffer overflows)
- Protected API

