

State of the Art Protection of Web Services

Alain Siegrist

`alain.siegrist@stud.hslu.ch`

Marc Siegrist

`marc.siegrist.01@stud.hslu.ch`

May 4, 2023

I Abstract

Web applications are some of the most attacked applications due to their widespread use. This paper describes common vulnerabilities and their corresponding security mitigations using a fictitious banking application as an example. Web application developers and security specialist need to consider a wide range of attack vectors to sufficiently protect applications from threats.

II Table of contents

1	Goals	3
2	State of the art	3
3	Ideas and concepts	3
3.1	Programming language	3
3.2	Database	3
3.3	Scenario	4
4	Methods	4
4.1	Literature review	4
4.2	Proof of concept	4
5	Implementation	5
5.1	Web application	5
5.2	Vulnerabilities	5
5.2.1	Passwords	5
5.2.2	Untrusted cookie usage	7
5.2.3	SQL Injection	10
5.2.4	Cross site request forgery	11
5.2.5	Data race condition	12
5.2.6	Logging and auditing	14
5.3	Other mitigations	15
5.3.1	Database authorization	15
5.3.2	Virtualization	16
5.3.3	Microservices	17
6	Evaluation and validation	18
6.1	Common security measures for a web application	18
6.2	Implementation of the chosen security measures	18
6.3	Scope	18
7	Outlook	19
8	Appendix	19
9	List of listings, figures, tables	19
10	Bibliography	20

1 Goals

This paper has two goals:

- Common security measures for a web application are described
- A simple web application which implements the chosen security measures is implemented

2 State of the art

Today many applications are developed on the web platform. For this reason attackers are increasingly focusing on exploiting and damaging such applications to generate a profit. To prevent these attacks, modern web applications implement a wide range of mitigations on multiple layers. These can be implemented in the application itself, the operating system it runs on or on the network between clients and the application. (Fredj, Cheikhrouhou, Krichen, Hamam, & Derhab, 2020)

To illustrate the different attacks a web app is exposed to, there exist web apps that were made to be vulnerable on purpose. An example of this is the Damn Vulnerable Web Application (Wood, 2023) or the OWASP Security Shepherd (OWASP Foundation, n.d.). These applications showcase different vulnerabilities on dedicated pages. This differs from the application in this paper, which aims to present an immersive scenario in an application, not a collection of exploitable example pages.

3 Ideas and concepts

There are many possible architectures, scenarios and technologies to choose for a vulnerable web application. The following sections detail what choices were made for this paper.

3.1 Programming language

The web application is written in the programming language Rust. The decision to use Rust has multiple reasons. One of the most important properties which distinguishes Rust from other native languages such as C, is the guaranteed memory safety. Security vulnerabilities which are caused by memory errors such as buffer overflows or use-after-frees can be effectively prevented. The type system in Rust can also be used to prevent security vulnerabilities. (Bugden & Alahmar, 2022)

3.2 Database

The web application uses a postgresql database (*PostgreSQL*, 2023) for persistence. postgresql was chosen because it is a standard relational database and used frequently in the industry. Another option that was considered is SQLite, but the limitations of in-memory

databases would not allow us to explore some of the possible mitigations in the area of database communication. (*SQLite*, n.d.)

3.3 Scenario

There are many possible scenarios one could develop a vulnerable web application for. To present a use case where security is of high importance, an online banking scenario was chosen. This allows the demonstration of many security issues and their mitigations in a realistic situation.

To focus on the security aspect, the banking application only has the minimum functionality to show possible vulnerabilities. It includes a login page, the ability to see your accounts and a way to create transactions. The application does not have any major styling or focus on UX design, to keep noise in the project to a minimum.

The initial code of the application will not implement security mechanisms, so a before and after comparison of weaknesses can be shown.

4 Methods

This paper investigates security vulnerabilities in web applications by conducting a literature review and developing a proof-of-concept application. The literature review will help identify the most relevant security vulnerabilities, while the proof-of-concept application will demonstrate their real-world implications and explore potential mitigations.

4.1 Literature review

A literature review is conducted to identify the most relevant security vulnerabilities affecting web applications. By examining existing research we can understand the nature of these vulnerabilities, their potential impact and the conditions under which they may arise. This review lays the groundwork for the development of a proof-of-concept application that includes these vulnerabilities to demonstrate their occurrence and consequences in a realistic context.

4.2 Proof of concept

Once the security vulnerabilities have been identified through the literature review they are deliberately incorporated into a proof-of-concept web application, either by adding them or by intentionally ignoring best practices. The literature is then consulted again to identify the appropriate mitigations for each vulnerability with further research conducted as needed to gain a deeper understanding. The mitigations are then implemented into the application and validated through manual testing against the running web application. This process allows for a practical examination of how these vulnerabilities manifest in real-world scenarios, as well as the effectiveness of the proposed mitigations.

5 Implementation

5.1 Web application

The web application is implemented in Rust using the axum web server library. (axum, 2023) The source code for the application can be found at https://github.com/Alainx277/too_big_to_exploit and in the submitted ZIP archive.

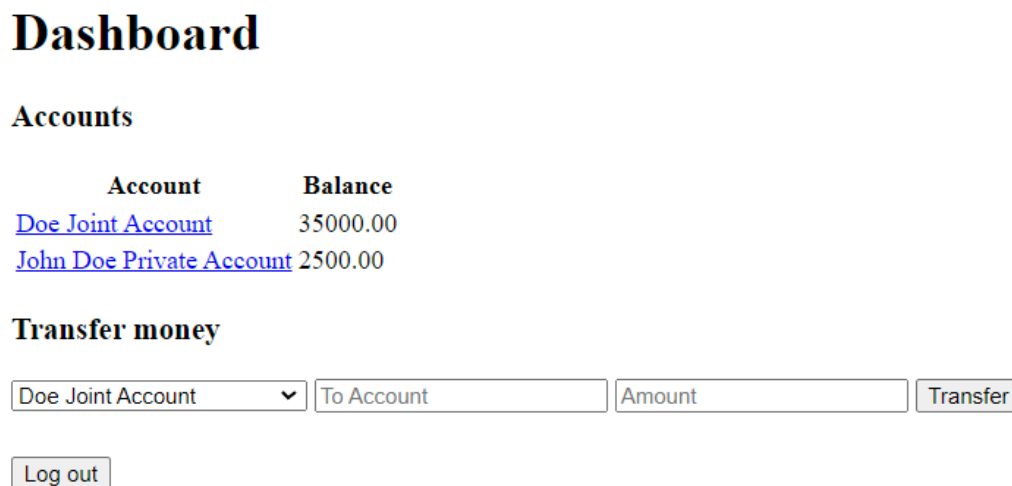
It features a login page, using a username and password, a dashboard page with account information and an option to transfer money between accounts.

Figure 1: Banking application login page



The login page features a large, bold, black serif title "Login" at the top. Below the title, there are three input fields arranged horizontally. The first field is labeled "Username" in a light gray font. The second field is labeled "Password" in a light gray font. To the right of the password field is a button labeled "Login" in a light gray font. All fields and the button have a thin gray border.

Figure 2: Banking application dashboard page



The dashboard page features a large, bold, black serif title "Dashboard" at the top. Below the title, there is a section titled "Accounts" in a bold, black serif font. Under "Accounts", there is a table with two columns: "Account" and "Balance". The table contains two rows of data. The first row shows "Doe Joint Account" with a balance of "35000.00". The second row shows "John Doe Private Account" with a balance of "2500.00". Below the table, there is a section titled "Transfer money" in a bold, black serif font. Under "Transfer money", there are four input fields arranged horizontally. The first field is a dropdown menu with "Doe Joint Account" selected. The second field is labeled "To Account" in a light gray font. The third field is labeled "Amount" in a light gray font. To the right of the amount field is a button labeled "Transfer" in a light gray font. Below the transfer fields, there is a button labeled "Log out" in a light gray font. All fields and buttons have a thin gray border.

Account	Balance
Doe Joint Account	35000.00
John Doe Private Account	2500.00

5.2 Vulnerabilities

5.2.1 Passwords

Many web applications include a password to authenticate users. The simplest way to handle a password is to treat it like any other value, and store it in your database:

Listing 1: User table definition

```
CREATE TABLE users (  
  id SERIAL PRIMARY KEY,  
  username VARCHAR NOT NULL,
```

```
password VARCHAR NOT NULL
);
```

This makes it trivial for an attacker to obtain all original passwords for accounts. Because people tend to reuse passwords on multiple sites, this allows attackers to perform credential stuffing attacks on other services after exfiltration. (Nathan, 2020)

To mitigate this attack, applications do not store the password directly, but transform it with a hashing function. The password can be validated by taking the user provided value, running it through the same hashing function, and comparing the result. If an attacker wants to restore the original value, they have no other choice than to hash all possible passwords and try to find a match (brute-force). This is because the hash used is a one-way-function.

There are some considerations in choosing the hashing algorithm. An attacker will likely execute the hash function many times, to try different passwords. In regular use by the web application, the hash function will only be executed when a user logs in, which happens far less often. An expensive hash function will inconvenience an attacker more than the web application, due to the difference in volume. Still, the server must be able to support logins without slowing down too much, so there is a tradeoff between server load and hash brute-force resistance. Modern algorithms like Argon2 require configurable amounts of both CPU and memory resources. (Biryukov, Dinu, & Khovratovich, 2023)

As a hash function is deterministic, two users with the same password will have the same hash value. This leads to attacks using a "rainbow table", where hashes for common passwords are precomputed. The attacker no longer has to execute the hash function for each user, but only once per password, significantly reducing the cost of a brute force attack. To prevent this a random value, commonly called a salt, is added before hashing. The salt is unique per account, making it impossible to reuse a precomputed hash for different accounts. (Rathod, Sonkar, & Chandavarkar, 2020)

The vulnerable web application replaced the plain-text password with a Argon2 hash:

Listing 2: Login check with hash

```
let row = db
    .query_opt (
        // We no longer check the password in the query
        "SELECT id, password_hash FROM users \
        WHERE username = $1 LIMIT 1",
        [&login.username],
    )
    .await
    .unwrap();
/* ... */
// Verify password against hash
let parsed_hash = PasswordHash::new(&password_hash).unwrap();
if Argon2::default()
```

```

    .verify_password(login.password.as_bytes(), &parsed_hash)
    .is_err()
{
    // Password is incorrect
    return /* ... */;
}
// Password is correct

```

Listing 3: Example argon2 hashes

```

-- ISpendAllMyMoney$$OnHats
$argon2id$v=19$m=19456,t=2,p=1
    $t6hGJwlwvqjhEfQAg7JDag$Q91/2kI7wqdFfHyf9NOBuZRQkOoIgOOZ26HbIWOR3XQ
-- HowDoIInvestInCrypto???
$argon2id$v=19$m=19456,t=2,p=1
    $zdlIdKhWICCNhk8fSQfz0Q$yCkd8rA9/dfxUibXw6RHY0e1gSZQVl0Y6/0pkpr3+t8
-- AllYourMoneyBelongsToMe:)
$argon2id$v=19$m=19456,t=2,p=1
    $8mT/HsK/0HxDkcxlCR2nxQ$MfU559U3UFERtYakxGozGVpPVQ44yWEfjfv9djz1io

```

5.2.2 Untrusted cookie usage

Cookies can be sent to clients to allow them to be identified in future requests. As the HTTP protocol is otherwise stateless, the usage of cookies allows the creation of user sessions. Cookies can be freely modified by a client between requests, and should be treated as untrusted input. (MDN, 2023b)

Listing 4: Vulnerable cookie implementation

```

// Setting the cookie
cookies.add(Cookie::new("user", user_id.to_string()));

// Retrieving the cookie in another request
let user_cookie = cookies.get("user") /* ... */;
let user_id: i32 = user_cookie.value().parse() /* ... */;

// Result: the attacker can freely control the user_id variable

```

Some web frameworks that implement session handling use an opaque token value that does not contain any actual information. These tokens should be generated by a cryptographic random number generator to prevent an attacker from predicting their values. When a client sends a request with a token, the server checks its data store for a match and finds the corresponding user data. (Mundada, Feamster, & Krishnamurthy, 2016)

Another option is to store actual information in a cookie. This allows the server to not store cookie data itself, but rely on the client providing it in the request. Without additional

measures this information must only be used for non-security relevant features, such as remembering a language selection.

It is possible to introduce a signature mechanism for the cookie, where it is signed with a key that only the server knows. On a new request, before the cookie value is used, the server verifies the signature, and will report an error if the cookie has been tampered with. It is important to note that a client can completely omit a cookie from a request, which a cookie signature cannot prevent.

If the client is not allowed to know the information stored in the cookie, it can be additionally encrypted. This prevents an attacker from extracting information from a cookie, which may be sensitive or aid in conducting further attacks. Many encryption algorithms are not tamper proof, which is why cookie encryption should always be combined with a signature.

Cookies used for session storage should have an expiration and revocation mechanism. Expiration prevents an attacker from stealing a cookie once and then reusing it continuously. Revocation allows a user to log out, making the cookie useless in the process. This can be useful in shared computer environments, such as in an university or library. Expiration can be implemented the same way for both cookie architectures. Attaching an expiry timestamp to the database or inside the cookie allows the server to check if the cookie is expired and reject it. For revocation the implementations differ. In the randomly generated token scenario, revocation is achieved by deleting the token from the servers data store. When using cookies that contain information, revocation can be implemented by attaching an id to the cookie and storing it in the server data store only for revoked cookies. This list is then checked against if a cookie has passed all other validations. (Ho, n.d.)

The banking application uses a traditional opaque token approach, storing it in the existing database:

Listing 5: SQL for new sessions table

```
CREATE TABLE sessions (  
  id SERIAL PRIMARY KEY,  
  token VARCHAR UNIQUE NOT NULL,  
  user_id SERIAL REFERENCES users(id) ON DELETE CASCADE NOT NULL,  
  valid_until timestamp NOT NULL DEFAULT NOW() + interval '1 day'  
);
```

Listing 6: Creating a session token

```
// Generate a session token (128 bits)  
// using a cryptographically secure generator  
let mut token_bytes = [0u8; 16];  
{  
  let mut rng = rand::thread_rng();  
  rng.fill_bytes(&mut token_bytes);  
}
```



```

let token =
    base64::prelude::BASE64_STANDARD.encode(token_bytes);

// Save to database
db.execute(
    "INSERT INTO sessions (token, user_id) VALUES ($1, $2)",
    [&token, &user_id],
)
.await
.unwrap();

// Set the session token as a cookie
cookies.add(Cookie::new("session", token.clone()));

```

Listing 7: Validating a session token

```

pub async fn validate_session(db: &Client, cookies: &Cookies)
    -> Option<i32> {
    let session_cookie = cookies.get("session"?);
    let token = session_cookie.value();
    // Try to find a valid session token in the database
    db.query_opt(
        "SELECT user_id FROM sessions \
        WHERE token = $1 AND valid_until > NOW()",
        [&token],
    )
    .await
    .unwrap()
    .map(|d| d.get(0))
}

```

Listing 8: Log out of session

```

pub async fn delete_session(db: &Client, cookies: &Cookies) {
    let Some(session_cookie) = cookies.get("session")
        else { return };
    let token = session_cookie.value();
    db.execute("DELETE FROM sessions WHERE token = $1", [&token])
        .await
        .unwrap();
}

```

5.2.3 SQL Injection

Web applications often utilize the Structured Query Language (SQL) to interface with their databases. An SQL injection vulnerability appears when user input is improperly handled or validated, allowing malicious actors to inject arbitrary SQL code into the application's queries. This can lead to unauthorized access, modification, or deletion of data.

The issue occurs in cases where web applications employ dynamic SQL queries, constructed by directly concatenating user inputs with SQL statements. An attacker can exploit this by injecting malicious SQL code into user input fields which can then be executed by the database server when the application processes the query. This happens because the database cannot differentiate between the trusted commands issued by the server, and the untrusted input an attacker provides. (Kindy & Pathan, 2011)

Listing 9: Vulnerable query

```
// SQL query is mixed with user input using string formatting
let query = format!(
    "SELECT id FROM users \
    WHERE username = '{} ' AND password = '{} ' LIMIT 1",
    login.username, login.password
);
let rows = db
    .query(&query, &[])
    .await
    .unwrap();
```

To mitigate SQL injection vulnerabilities, developers can employ various techniques. A common method is the use of parameterized queries or prepared statements. This approach separates the SQL code from the user input, ensuring that the input is treated as a parameter rather than as a part of the query itself. Consequently, it prevents the execution of any injected SQL code by treating it as text, without parsing any of the statements.

Listing 10: Parameterized query

```
// Query is a static string
let query = "SELECT id FROM users \
WHERE username = $1 AND password = $2 LIMIT 1";
let rows = db.query(
    query,
    // Parameters are passed using the designated argument
    [&login.username, &login.password],
)
    .await
    .unwrap();
```

Another mitigation is input validation and sanitization. By enforcing strict validation

rules and sanitizing user inputs, developers can ensure that only valid and safe data is passed to the application's SQL queries. This method is helpful for insertions that cannot be implemented using prepared statements, such as using a user provided table name in a query. (Kindy & Pathan, 2011)

5.2.4 Cross site request forgery

Requests between websites are generally restricted by browsers. The Cross-Origin-Resource-Sharing (CORS) mechanism prevents websites from making requests to other websites and reading the response by default. Before sending an HTTP request initiated by a site, for example via JavaScript code, the browser will make a preflight request to check if the receiving server allows the origin website to make the given request. If not, the request will be cancelled and not sent. (MDN, 2023a)

The problem appears when the HTTP methods GET and POST are used, which do not create preflight requests across different sites. If a POST endpoint exists which modifies data and no mitigations are in place, an attacker can force the victims browser to make a request to that endpoint with attacker-determined values, which the vulnerable server will accept if the client has an existing session. The attacker can do this by injecting code into an existing page with XSS or creating their own website and luring the victim to visit it. (OWASP Foundation, 2021)

Listing 11: Vulnerable form implementation

```
// Fund transfer endpoint definition
router.route(
  "/transfer",
  post(routes::transaction::transfer_funds)
)

// Implementation
#[derive(Deserialize)]
pub struct TransferParams {
  from: i32,
  to: i32,
  amount: Decimal,
}

pub async fn transfer_funds(
  /* ... */
  Form(transfer): Form<TransferParams>,
) -> /* ... */ {
  // Form data in "transfer" variable is used
  // without additional verification
```

```
}
```

A webpage that showcases the CSRF attack is located in the submitted ZIP archive.

One mitigation uses a randomly generated token (often called a CSRF token), which is added to the HTML of any form that causes a modification. When a user interacts with the page normally, the form submission will include this token and the server can then verify that the request was made legitimately. A attacker on another site cannot get a valid token, as CORS will not allow them to read the response containing the form with the token.

Another mitigation uses the SameSite attribute for the session cookie. If the attacker is prevented from sending user authentication data from another site, modifications will fail because the request is not authenticated. A SameSite attribute set to Strict will not send the cookie if the request is initiated from another site. Setting it to Lax is the same as Strict, except that normal navigation from another site, like links or redirects, will include the cookie. (OWASP Foundation, 2021)

Listing 12: Preventing CSRF with SameSite cookie

```
let mut cookie = Cookie::new("session", token.clone());
// Ensure the session cookie is not sent across origins
cookie.set_same_site(SameSite::Lax);
cookies.add(cookie);
```

5.2.5 Data race condition

Web applications frequently utilize concurrent access to databases to serve requests in parallel, which can lead to data race conditions when multiple threats access shared data simultaneously. Data race conditions occur when two or more threads modify, access, or update shared data concurrently, resulting in unpredictable and undesirable outcomes. These vulnerabilities can cause application instability and logically inconsistent data.

The problem arises when web applications fail to implement proper transaction isolation mechanisms, allowing multiple database connections to access shared resources without logical consistency. (Paleari, Marrone, Bruschi, & Monga, 2008)

Listing 13: Vulnerable transfer queries

```
// Add money to receiver
db
    .execute(
        "UPDATE accounts SET balance = balance + $1 WHERE accounts.id = $2",
        [&transfer.amount, &transfer.to],
    )
    .await
    .unwrap();
// If two requests enter this section at the same time,
// the amount will only be deducted once but
```

```

// added to the receiver twice
let new_balance = balance - transfer.amount;
db
  .execute(
    "UPDATE accounts SET balance = $1 WHERE accounts.id = $2",
    [&new_balance, &transfer.from],
  )
  .await
  .unwrap();

```

Listing 14: JavaScript code exploiting the data race

```

function spend(from, to, amount) {
  let formData = new URLSearchParams();
  formData.append('from', from);
  formData.append('to', to);
  formData.append('amount', amount);
  return fetch("/transfer", { body: formData, method: "post" });
}

// This sends two transfer requests at the same time
// to trigger the data race
spend(1, 3, 100); spend(1, 3, 100);

```

These race conditions can be mitigated by using database transactions and isolation levels. By wrapping operations within a transaction, developers can ensure that concurrent access to shared data is handled correctly, preventing data inconsistencies and maintaining data integrity.

Listing 15: Transfer queries with transaction

```

// Starts a transaction
// If anything goes wrong the changes will not be committed
let transaction = db.transaction().await.unwrap();

// Get the available balance
let result = transaction
  .query_opt(
    // Using FOR UPDATE will prevent
    // concurrent modification of the balance
    "SELECT accounts.balance FROM /* ... */ FOR UPDATE",
    /* ... */
  )
  .await
  .unwrap();

```

```

/* ... */

// If everything was successful changes are committed
transaction.commit().await.unwrap();

```

5.2.6 Logging and auditing

In the event of a security incident impacting an application, threat hunters should be able to reconstruct how the attack took place. A core part of the threat hunting process is analyzing log output of relevant systems, such as any potential firewalls, proxies, servers and the application itself. Distributed tracing frameworks like OpenTracing allow correlation between logs and events in different systems and services. This makes it easier for analysts to see related actions at a glance, without having to query for the logs separately. Logs should also be kept in a central repository to allow for automated or manual correlation on one system. (Wats, 2022)

A critical part of the logging strategy is to keep a record of the business logic inside the application. This allows an analysis of the impact on business data through modification or deletion. Logging access to data enables the same for monitoring data exfiltration.

To ensure non-repudiation all actions an authenticated user initiates should include the relevant account information, such as the user id. If a malicious employee decides to damage business critical data, they can be caught. Notifying employees that their actions on the application are monitored can also dissuade them from doing anything malicious at all. (Chuvakin & Peterson, 2010)

In the POC bank application at least logins and transactions should be logged:

Listing 16: Logging code

```

// In login endpoint
/* ... */
if rows.is_empty() {
    warn!(user = login.username, "Authentication failed");
    /* ... */
}
/* ... */
info!(user = login.username, "User authenticated");

// In transfer endpoint
info!(
    user = user_id,
    from = &transfer.from,
    to = &transfer.to,
    amount = transfer.amount.to_string(),
    "Executed transaction"
)

```

```
);
```

Listing 17: Example log output

```
WARN request{method=POST uri=/login}: routes::login:
  Authentication failed user="johndoe"
INFO request{method=POST uri=/login}: routes::login:
  User authenticated user="johndoe"
INFO request{method=POST uri=/transfer}: routes::transaction:
  Executed transaction user=1 from=1 to=3 amount="100"
```

When logging, care must be taken to not expose sensitive information through log files. If for example API keys or passwords are leaked, an attacker could acquire the log files through a different vulnerability and use them for data exfiltration or lateral movement. Such an attack can be seen with leaked backup keys in logs, leading to databases being stolen from backup storage.

5.3 Other mitigations

There are mitigations that do not address a specific vulnerability, but instead limit the impact of a successful attack, should one occur.

5.3.1 Database authorization

Databases are a valuable target for an attacker, as they allow modification of data to the attackers benefit and may contain sensitive data such as names and billing details. Implementing proper database authorization helps prevent unauthorized access, modification, or deletion of data, limiting the impact of a successful attack on the application.

Listing 18: Using the most privileged database user unnecessarily

```
tokio_postgres::connect(
    "host=localhost user=postgres /* ... */",
    /* ... */
);
```

The key principle in database authorization is the concept of least privilege. This principle ensures that users and processes are granted only the minimum necessary permissions to perform their tasks, restricting their access to sensitive data and reducing the risk of unauthorized actions. By following this principle, developers can limit the potential damage that can be caused by an attacker who manages to compromise an application that communicates with the database.

Implementing database authorization involves several steps. Developers must define user roles and permissions, assigning appropriate access levels for each role to ensure that they can only perform the specific tasks required. Access controls should be enforced at the database level to restrict access to tables, views, or stored procedures for authorized users

and processes. Regular auditing and reviewing of permissions is necessary to maintain relevance and eliminate unnecessary access. Monitoring and logging database activity helps in tracking user actions and detecting suspicious activities, allowing developers to identify potential security breaches and take appropriate action. (Bertino & Sandhu, 2005)

Listing 19: Creating database user

```
CREATE USER bank_app NOINHERIT;
-- Password must manually be set with:
-- ALTER USER bank_app WITH PASSWORD '<password>';
GRANT CONNECT ON DATABASE bank TO bank_app;
GRANT USAGE ON SCHEMA public TO bank_app;
GRANT SELECT ON TABLE users TO bank_app;
GRANT SELECT ON TABLE accounts TO bank_app;
GRANT UPDATE ON TABLE accounts TO bank_app;
GRANT SELECT ON TABLE account_owners TO bank_app;
GRANT UPDATE ON TABLE account_owners TO bank_app;
GRANT SELECT ON TABLE sessions TO bank_app;
GRANT INSERT ON TABLE sessions TO bank_app;
GRANT DELETE ON TABLE sessions TO bank_app;
GRANT USAGE ON SEQUENCE sessions_id_seq TO bank_app;
GRANT SELECT ON SEQUENCE sessions_id_seq TO bank_app;
```

Listing 20: Using the new database user

```
let connection_string = format!(
    // Using the low-permission user bank_app
    "host=localhost user=bank_app password={} dbname=bank",
    // Take the password from the environment
    // instead of hardcoding it
    env::var("DB_PASSWORD").unwrap()
);
```

5.3.2 Virtualization

In the event of a successful exploitation of a component in the web application the attacker will look to gain additional control using various lateral movement techniques. Virtualization plays a crucial role in limiting the impact of such security breaches by isolating different components and applications, reducing the risk of unauthorized access to other parts of the system.

Virtualization involves creating multiple virtual instances of computing resources such as operating systems, storage, and network devices, on a single physical host.

In the banking application, the database and web application should be in different virtual instances to minimize the opportunity for lateral movement.

By employing virtualization, developers can create isolated environments for each component of the web application. This separation ensures that even if an attacker manages to exploit a vulnerable component, their access will be limited to that specific virtual instance. The isolation provided by virtualization makes it more difficult for attackers to compromise other components or access sensitive data stored elsewhere in the system.

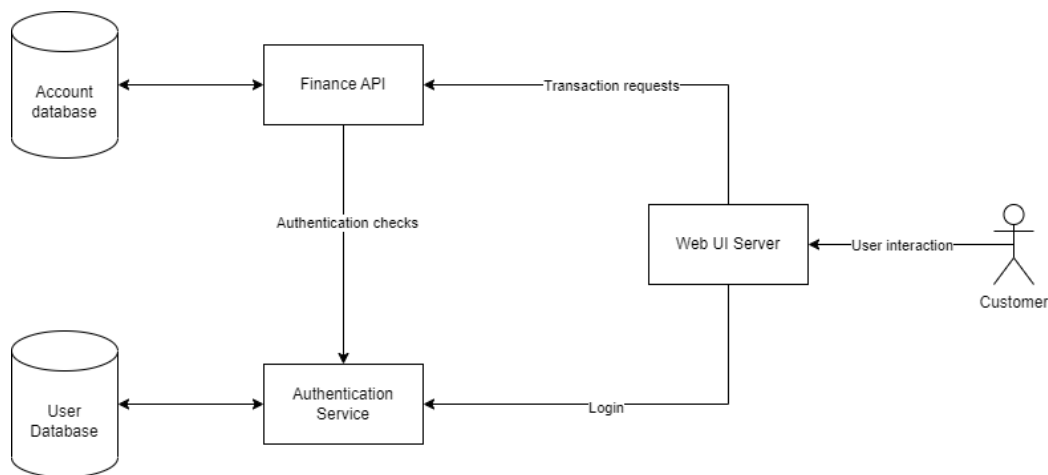
To further enhance security, developers can also implement additional controls within the virtual environment, such as network segmentation, strict access controls, and monitoring for suspicious activities. These measures can help detect and prevent lateral movement attempts by attackers, further limiting their ability to cause damage.

Another benefit of virtualization is the ability to quickly recover from a security breach. By maintaining snapshots of virtual instances, developers can rapidly restore the system to a known good state in the event of a compromise. This capability not only minimizes downtime after an attack but also prevents persistent threats from remaining in the system. (Pearce, Zeadally, & Hunt, 2013)

5.3.3 Microservices

Because components of an application are often tightly interwoven and cannot be easily split by technologies like virtualization, there are security advantages to splitting an application into more components in the design phase. This approach, known as microservices architecture, breaks down the application into smaller, loosely-coupled services that can be developed, deployed, and maintained independently.

Figure 3: Possible microservice architecture for banking application



By adopting a microservices architecture improved security can be achieved through the isolation of individual components. This isolation makes it more difficult for attackers to compromise the entire application, as gaining access to one microservice does not necessarily grant them access to others. Each microservice can have its own set of security controls, tailored to its specific requirements and risks, further increasing the difficulty of misuse.

The microservices approach also enables the principle of least privilege to be applied more effectively. By limiting the scope and responsibilities of each microservice, access controls and permissions can be better defined, reducing the attack surface and the potential impact of a security breach.

Microservices can also enhance the resilience of a web application because they can be independently deployed and scaled. If one service is compromised or experiences a failure, it is less likely to impact the overall functionality and availability of the application. (Chandramouli, 2019)

6 Evaluation and validation

In this section, we assess the success of the paper by examining the extent to which the goals set at the beginning of the paper have been achieved. We will review the two objectives: (1) identifying and describing common security measures for a web application and (2) implementing a simple web application that integrates the chosen security measures.

6.1 Common security measures for a web application

Through a comprehensive analysis of web application security, we have successfully described a range of common vulnerabilities and their corresponding security mitigations. By using the fictitious scenario banking application as an example, we have provided practical, real-world scenarios to demonstrate the importance of these security measures.

6.2 Implementation of the chosen security measures

In addition to describing common security measures, our research aimed to create a simple web application that implements the chosen security measures. By developing a functional prototype of the fictitious banking application, we have demonstrated and verified the integration of the identified security measures in a real-world context. This accomplishment provides a valuable proof-of-concept for the effectiveness of these security measures and serves as an example for future web application security research and development.

6.3 Scope

While this paper has addressed a variety of common web application security vulnerabilities and their corresponding mitigations, it is important to acknowledge that the field of web application security is vast and constantly evolving. As a result, there exists a wide range of other vulnerabilities and potential threats that have not been covered in this research.

7 Outlook

This paper has laid the groundwork for understanding and addressing some fundamental web application security vulnerabilities but there is still much to be explored in this field.

One area of potential exploration is denial of service (DoS) attacks, which pose a significant threat to public web applications, as they can render the applications inaccessible and consequently cause significant loss to a business. A comprehensive examination of various mitigation strategies for DoS attacks, tailored to the specific availability goals of the application in question, could be the focus of another paper. This research would contribute to a more robust understanding of how to protect web applications from such threats.

Another promising topic for future research is the design and implementation of security-focused microservice architectures. It could showcase the security challenges and advantages associated with developing applications with a larger scope and a distributed data model. By analyzing the additional threats that arise when data is distributed between systems, researchers can better understand the complexities of ensuring security in a microservices environment.

8 Appendix

The paper is submitted in a ZIP archive containing the following files:

Folder / File	Description
G10_Siegrist_Siegrist_Tempaper.pdf	The rendered paper
G10_Siegrist_Siegrist_Tempaper.tex	The latex source file of the paper
app/	The files for the insecure banking application
app_with_fixes/	The files for the insecure banking application with security mitigations implemented
images/	Images included in the paper

Table 1: Files in the ZIP archive

9 List of listings, figures, tables

List of listings

1	User table definition	5
2	Login check with hash	6
3	Example argon2 hashes	7
4	Vulnerable cookie implementation	7
5	SQL for new sessions table	8
6	Creating a session token	8
7	Validating a session token	9

8	Log out of session	9
9	Vulnerable query	10
10	Parameterized query	10
11	Vulnerable form implementation	11
12	Preventing CSRF with SameSite cookie	12
13	Vulnerable transfer queries	12
14	JavaScript code exploiting the data race	13
15	Transfer queries with transaction	13
16	Logging code	14
17	Example log output	15
18	Using the most privileged database user unnecessarily	15
19	Creating database user	16
20	Using the new database user	16

List of figures

1	Banking application login page	5
2	Banking application dashboard page	5
3	Possible microservice architecture for banking application	17

List of tables

1	Files in the ZIP archive	19
---	------------------------------------	----

10 Bibliography

- axum*. (2023, April). Retrieved 2023-05-04, from <https://crates.io/crates/axum>
- Bertino, E., & Sandhu, R. (2005, January). Database security - concepts, approaches, and challenges. *IEEE Transactions on Dependable and Secure Computing*, 2(1), 2–19. Retrieved 2023-05-04, from <http://ieeexplore.ieee.org/document/1416861/> doi: 10.1109/TDSC.2005.9
- Biryukov, A., Dinu, D., & Khovratovich, D. (2023, April). *Argon2*. Retrieved 2023-04-18, from <https://github.com/P-H-C/phc-winner-argon2/blob/f57e61e19229e23c4445b85494dbf7c07de721cb/argon2-specs.pdf> (original-date: 2015-10-04T08:25:27Z)
- Bugden, W., & Alahmar, A. (2022, June). *Rust: The Programming Language for Safety and Performance*. arXiv. Retrieved 2023-03-09, from <http://arxiv.org/abs/2206.05503> (arXiv:2206.05503 [cs]) doi: 10.48550/arXiv.2206.05503
- Chandramouli, R. (2019, August). *Security strategies for microservices-based application systems* (Tech. Rep. No. NIST SP 800-204). Gaithersburg, MD: National Institute of Standards and Technology. Retrieved 2023-05-04, from <https://nvlpubs>

- .nist.gov/nistpubs/SpecialPublications/NIST.SP.800-204.pdf doi: 10.6028/NIST.SP.800-204
- Chuvakin, A., & Peterson, G. (2010, July). How to Do Application Logging Right. *IEEE Security & Privacy Magazine*, 8(4), 82–85. Retrieved 2023-05-04, from <http://ieeexplore.ieee.org/document/5523872/> doi: 10.1109/MSP.2010.127
- Fredj, O., Cheikhrouhou, O., Krichen, M., Hamam, H., & Derhab, A. (2020). *An OWASP Top Ten Driven Survey on Web Application Protection Methods*. doi: 10.36227/techrxiv.13265180.v1
- Ho, C. K. (n.d.). Descriptive Research for JWT Implementation as Session Data.
- Kindy, D. A., & Pathan, A.-S. K. (2011, June). A survey on SQL injection: Vulnerabilities, attacks, and prevention techniques. In *2011 IEEE 15th International Symposium on Consumer Electronics (ISCE)* (pp. 468–471). Singapore, Singapore: IEEE. Retrieved 2023-05-04, from <http://ieeexplore.ieee.org/document/5973873/> doi: 10.1109/ISCE.2011.5973873
- MDN. (2023a, April). *Cross-Origin Resource Sharing (CORS)*. Retrieved 2023-05-04, from <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>
- MDN. (2023b, April). *Using HTTP cookies*. Retrieved 2023-05-04, from <https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies>
- Mundada, Y., Feamster, N., & Krishnamurthy, B. (2016, May). Half-Baked Cookies: Hardening Cookie-Based Authentication for the Modern Web. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security* (pp. 675–685). Xi'an China: ACM. Retrieved 2023-05-04, from <https://dl.acm.org/doi/10.1145/2897845.2897889> doi: 10.1145/2897845.2897889
- Nathan, M. (2020, January). Credential stuffing: new tools and stolen data drive continued attacks. *Computer Fraud & Security*, 2020(12), 18–19. Retrieved 2023-04-18, from <http://www.magonlinelibrary.com/doi/10.1016/S1361-3723%2820%2930130-5> doi: 10.1016/S1361-3723(20)30130-5
- OWASP Foundation. (n.d.). *OWASP Security Shepherd*. Retrieved 2023-05-04, from <https://owasp.org/www-project-security-shepherd/>
- OWASP Foundation. (2021). *Cross-Site Request Forgery Prevention*. Retrieved 2023-05-04, from <https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site-Request-Forgery-Prevention-Cheat-Sheet.html>
- Paleari, R., Marrone, D., Bruschi, D., & Monga, M. (2008). On Race Vulnerabilities in Web Applications. In D. Zamboni (Ed.), *Detection of Intrusions and Malware, and Vulnerability Assessment* (Vol. 5137, pp. 126–142). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved 2023-05-04, from http://link.springer.com/10.1007/978-3-540-70542-0_7 (ISSN: 0302-9743, 1611-3349 Series Title: Lecture Notes in Computer Science) doi: 10.1007/978-3-540-70542-0_7
- Pearce, M., Zeadally, S., & Hunt, R. (2013, February). Virtualization: Issues, security threats, and solutions. *ACM Computing Surveys*, 45(2), 1–39. Retrieved 2023-05-04, from <https://dl.acm.org/doi/10.1145/2431211.2431216> doi: 10.1145/

2431211.2431216

- PostgreSQL*. (2023, May). Retrieved 2023-05-04, from <https://www.postgresql.org/>
- Rathod, U., Sonkar, M., & Chandavarkar, B. R. (2020, July). An Experimental Evaluation on the Dependency between One-Way Hash Functions and Salt. In *2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT)* (pp. 1–7). Kharagpur, India: IEEE. Retrieved 2023-04-18, from <https://ieeexplore.ieee.org/document/9225503/> doi: 10.1109/ICCCNT49239.2020.9225503
- SQLite*. (n.d.). Retrieved 2023-05-04, from <https://sqlite.org/index.html>
- Wats, S. (2022, June). *Log Management: A Useful Introduction*. Retrieved 2023-05-04, from https://www.splunk.com/en_us/blog/learn/log-management.html
- Wood, R. (2023, May). *Damn Vulnerable Web Application*. Retrieved 2023-05-04, from <https://github.com/digininja/DVWA> (original-date: 2013-05-01T13:03:10Z)