Zellic

Pocket

Web Application Security Assessment

November 17, 2023

Prepared for:

Avara Labs Ltd.

Prepared by:

SeungHyeon Kim and Yuhang Wu

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for Avara Labs Ltd. from September 25th, 2023 to October 4th, 2023. During this engagement, Zellic reviewed Pocket's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Is there any issue that could lead to the loss of user funds?
- If it is possible to bypass the authentication mechanism, can an attacker gain access to the user's wallet?
- Is there any issue that could lead to the loss of user data?
- Could an attacker gain access of the server?

1.2 Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Deployment issues
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

During this assessment, the absence of a specific project-execution environment prevented us from evaluating potential vulnerabilities related to third-party code hosting, domain names, emails, DNS, and other related environmental factors.

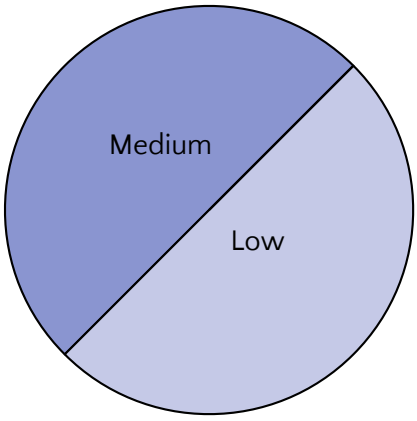
1.3 Results

During our assessment on the scoped Pocket modules, we discovered two findings. No critical issues were found. One finding was of medium impact and one was of low impact.

Additionally, Zellic recorded its notes and observations from the assessment for Avara Labs Ltd.'s benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	1
Low	1
Informational	0



2 Introduction

2.1 About Pocket

Pocket is a noncustodial wallet that is designed to lower the barrier of entry for regular users, speeding up the onboarding process into the crypto world. It is built with solid security in mind as well as an encryption system that makes it possible to recover the wallet in case the password is forgotten.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the modules.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general.

We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped modules itself. These observations — found in the Discussion (4) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3 Scope

The engagement involved a review of the following targets:

Pocket Modules

Repository	https://github.com/pocket-wallet/avara-abstracted-account
Version	avara-abstracted-account: `1b9da66d41adb9bde4499bfc5d83122769d7225e`
Program	*.ts, *.tsx, *.js, *.kt, *.c, *.swift
Type	Web application
Platform	Web, mobile

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of two person-weeks. The assessment was conducted over the course of two calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

SeungHyeon Kim, Engineer
seunghyeon@zellic.io

Yuhang Wu, Engineer
yuhang@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

September 24, 2023	Kick-off call
September 25, 2023	Start of primary review period
October 4, 2023	End of primary review period
October 4, 2023	Draft report delivered

3 Detailed Findings

3.1 Vulnerability in JWT role-based access for account recovery and log-in

- **Target**: apps/server/src/graphql/resolvers/recovery/index.tss
- **Category**: Coding Mistakes
- **Severity**: Medium
- **Likelihood**: Low
- **Impact**: Medium

Description

Upon analysis of Pocket's current design, it is evident that the system prioritizes security, especially in the `Recover` function meant for forgotten passwords. This necessitates the validation of either a device ID or offline ID even after the email verification code has been accepted, ensuring that a mere takeover of a user's email would not allow direct log-in or password modification.

However, a possible loophole was identified that might allow an attacker who has compromised an email, without the two-factor authentication enabled, to gain unauthorized access to the associated account. The identified steps are as follows:

Step 1

Get the identity of `AAAJwtAccessRole.validate` role. The `recoverySendValidationCode` function in apps/server/src/graphql/resolvers/recovery/index.ts can be exploited by an attacker, using just the email address, to achieve `AAAJwtAccessRole.validate` status.

```
@Mutation(() ⇒ JwtInfo)
  public async recoverySendValidationCode(
    @Arg('request') request: RecoverySendValidationCodeRequest,
  ): Promise<JwtInfo> {
    const identity = Identity.fromRawIdentity(request.username);
    const user = await getUserAccountByIdentity(identity);

    if (!user) {
      // we dont want people finding out if the user exists and then have
    a footprint
      // in trying to smash the password so throw generic error
      throw new ValidationError('No recovery keys found');
```

```
    }

    await sendValidationCode(identity, request.username,
    ValidationCodeSentFromStep.RECOVERY);

    return createJwtInfo(user.userAccountId, AAAJwtAccessRole.validate,
    null);
  }
```

Step 2

Send the log-in validation code. Once the attacker achieves the `AAAJwtAccessRole.validate` status, they can invoke the `sendValidationCode` function in apps/server/src/graphql/resolvers/login/index.ts to send the log-in validation code to the compromised email. Notably, this code and its type are stored in the database.

```
@Mutation(() => VoidScalar, { nullable: true })
  @UseMiddleware(authRequired(AAAJwtAccessRole.validate))
  public async sendValidationCode(
    @Arg('request') { username, validationCodeHash }:
    LoginSendValidationCodeRequest,
    @Ctx() context: AppContext,
  ): Promise<typeof VoidScalar> {
    // eslint-disable-next-line @typescript-eslint/no-non-null-assertion
    const user = await getUserAccountByUserId(context.user!.id);
    if (!user) {
      // do not a footprint
      return VoidScalar;
    }

    const lockParams = {
      identity: user.identity,
      ipAddressHash: context.ipAddressHash,
      trigger: AttemptTrigger.LOGIN_VERIFICATION_CODE,
    };

    const lockInfo = await getUserLockInfo(lockParams);
    if (lockInfo.isLocked) {
      return VoidScalar;
    }
```

```
    const wasAlreadySend = await isCodeSend(user.identity,
    ValidationCodeSentFromStep.LOGIN);

    if (wasAlreadySend) {
      // each new validation code request is treated like an unsuccessful
    attempt
      // given the user was not able to provide a correct validation code
      await reportUnsuccessfulAttemptToUserLock(lockParams);
    }

    await sendValidationCode(
      user.identity,
      username,
      ValidationCodeSentFromStep.LOGIN,
      validationCodeHash,
    );

    return VoidScalar;
  }
```

Step 3

Log in using the validation code. An attacker can then exploit the `loginValidateCode` function in apps/server/src/graphql/resolvers/login/index.ts with the acquired code from the email and the `AAAJwtAccessRole.validate` status. This allows them not only to log in but also to refresh to a new device, making it more challenging for the original user to recover the account.

```
@Mutation(() => ValidateCodeLoginResultUnion)
  @UseMiddleware(authRequired(AAAJwtAccessRole.validate))
  public async loginValidateCode(
    @Arg('request') request: LoginValidationCodeRequest,
    @Ctx() context: AppContext,
  ): Promise<typeof ValidateCodeLoginResultUnion> {
    // eslint-disable-next-line @typescript-eslint/no-non-null-assertion
    const user = await getUserAccountByUserId(context.user!.id);
    if (!user) {
      // we dont want people finding out if the user exists and then have
    a footprint
```

```
      // in trying to smash the password so throw generic error
      throw new ValidationError('Could not login at this time.');
    }

    const lockParams = {
      identity: user.identity,
      ipAddressHash: context.ipAddressHash,
      trigger: AttemptTrigger.LOGIN_VERIFICATION_CODE,
    };

    const lockInfo = await getUserLockInfo(lockParams);
    if (lockInfo.isLocked) {
      return handleUserLocked();
    }

    const codeValid = await validateCode(
      user.identity,
      request.validationCode,
      ValidationCodeSentFromStep.LOGIN,
    );

    if (!codeValid) {
      await reportUnsuccessfulAttemptToUserLock(lockParams);

      return buildValidationInvalidCode();
    }

    await reportSuccessfulAttemptToUserLock(lockParams);

    await redeemCode(user.identity, request.validationCode,
    ValidationCodeSentFromStep.LOGIN);

    const twoFactorEnabled
    = await isTwoFactorAuthEnabled(user.userAccountId);
    if (twoFactorEnabled) {
      return handleTwoFactorRequired(user.userAccountId, null);
    }

    return this.loginWithNewDeviceComplete(user.userAccountId);
  }
```

Impact

Attackers can bypass log-in restrictions and log in to the account using this method. Since all sensitive data is encrypted after logging in, the impact is not significant. However, attackers can block the user's normal log-in and have the opportunity to brute force the user's password locally, thereby taking over the user's wallet.

Recommendations

In addition to `AAAJwtAccessRole.validate` serving as an intermediate role for log-in, a new role should also be added for recovery.

Remediation

This issue has been acknowledged by Avara Labs Ltd., and a fix was implemented in commit 380d1fb3.

3.2 Missing rate limiting in `recoverySendValidationCode` function

- **Target**: apps/server/src/graphql/resolvers/recovery/index.ts
- **Category**: Coding Mistakes
- **Severity**: Low
- **Likelihood**: Medium
- **Impact**: Low

Description

The `recoverySendValidationCode` permits the dispatching of a validation code devoid of any inherent rate-limiting measure.

```
@Mutation(() => JwtInfo)
public async recoverySendValidationCode(
  @Arg('request') request: RecoverySendValidationCodeRequest,
): Promise<JwtInfo> {
  const identity = Identity.fromRawIdentity(request.username);
  const user = await getUserAccountByIdentity(identity);
  if (!user) {
    throw new ValidationError('No recovery keys found');
  }
  await sendValidationCode(identity, request.username,
    ValidationCodeSentFromStep.RECOVERY);
  return createJwtInfo(user.userAccountId, AAAJwtAccessRole.validate,
    null);
}
```

Impact

By leveraging this vulnerability, attackers can

- detect active users by repeatedly calling the function and discerning variations in response durations or behaviors.
- disturb user interactions by ceaselessly dispatching verification codes to their inboxes, possibly leading them to neglect or misplace pivotal emails or alerts.

Recommendations

Add a rate-limiting mechanism to the `recoverySendValidationCode` function.

```
const lockParams = {
      identity: user.identity,
      ipAddressHash: context.ipAddressHash,
      trigger: AttemptTrigger.RECOVERY_VERIFICATION_CODE,
    };

    const lockInfo = await getUserLockInfo(lockParams);
    if (lockInfo.isLocked) {
      return handleUserLocked();
    }
```

Remediation

This issue has been acknowledged by Avara Labs Ltd., and a fix was implemented in commit 380d1fb3.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1 Passwordless authentication

For the passwordless authentication method, the wallet supports the authentication via `Sign in with Apple`. Since the wallet is using Apple's `id_token`, the wallet needs strict rules for verifying the token.

We know misuse of the JWT library will lead to some vulnerabilities, like authentication bypass via changing the algorithm of JWT as described in the article "Critical Vulnerabilities in JSON Web Token Libraries". The token for `Sign in with Apple` is using the RS256 algorithm. So, the backend must restrict the ALG to RS256. If it does not restrict the algorithm, the attacker can change the ALG to HS256 simply and can make any valid JWT.

In the library code the Pocket team is using, it applies the algorithm from the given token. This means we can bypass the authentication as stated above.

While validating the vulnerability, it was discovered that one of the libraries being used, jwtwebtoken, is preventing the attack. However, the Pocket team decided to import the code from the library and modify it to ensure JWT safety.

4.2 Debugging information

While investigating the real server to gather information, we discovered that the server was operating in debug mode, raising potential security concerns. So, we would like to remind the Pocket team that if the actual project is not switched to production mode during deployment, some sensitive debug information might be leaked out (stack trace, library versions, etc.).

When the requests for `/api` fail during a `graphQL` query, it will reveal sensitive information, including the library version, application location, and structure of the application directory.

Additionally, we observed that `Access-Control-Allow-Origin` was set to `*`. This configuration could potentially introduce vulnerabilities, like untrusted pages using XMLHttpRequest and leaking the information of a user.

4.3 Hashed username on the DB

The Pocket team decided to store the hashed username on the DB for the backend's user validation. They explained it is because storing a username in plaintext can be a security issue given that once an attacker has access to the DB, they could try to brute force the encrypted seed. So, they store a hashed username called `identity` in the database. To create `identity`, they hash the username with `SERVER_SALT` using SHA-3.

There are several possible ways for this to lead to an attack scenario:

- The `SERVER_SALT` is too short and still possible to brute force.
- The attacker gets the `SERVER_SALT` somehow and then hashes it with the username.
- The `SERVER_SALT` is a fixed value, not a random.

So, hashing the username can be a great idea, but the Pocket team must be careful when choosing `SERVER_SALT`. A secure method is to use a random function to generate long `SERVER_SALT` randomly. This `SERVER_SALT` must be kept to make sure codes work with `identity`. So, we recommend the Pocket team make a local-only endpoint generating and returning `SERVER_SALT`.

5 Assessment Results

At the time of our assessment, the reviewed code was deployed to the Vercel.

During our assessment on the scoped Pocket modules, we discovered two findings. No critical issues were found. One finding was of medium impact and one was of low impact. Avara Labs Ltd. acknowledged all findings and implemented fixes.

5.1 Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.