Zellic

Avara Abstracted Account

Web Application Security Assessment

February 22nd, 2023

Prepared for:

Avara Labs Ltd.

Prepared by:

Jasraj Bedi and Maik Robert

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 November 24th to December 9th, 2022. During this engagement, Zellic reviewed Avara Abstracted Account'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:

- Are there any security issues in the iFrame setup?
- Is sniffing inter-iFrame communication via postMessage and Broadcast channels possible?
- Is the cryptography setup used secure?

1.2 Non-goals and Limitations

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

- Issues stemming from code or infrastructure outside of the assessment scope

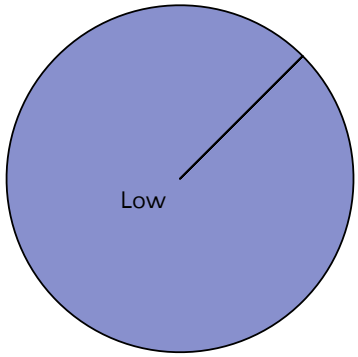
1.3 Results

During our assessment on the scoped Avara Abstracted Account contracts, we discovered one finding. No critical issues were found. The single finding was low in nature.

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	0
Low	1
Informational	0



2 Introduction

2.1 About Avara Abstracted Account

Avara Abstracted Account is a non-custodial 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 web-based applications arise from oversights during development. Missing an authentication check on a single API endpoint can circumvent the entire authentication model of the application. Failure to properly sanitize and encode user input can lead to vulnerabilities like SQL injection or cross-site scripting. We use automated tools to identify unsafe code patterns and perform a thorough manual review for vulnerable code patterns.

Business logic errors. Business logic is the heart of all applications. We manually review logic to ensure that the code implements the expected functionality specified in the platform's design documents. We also thoroughly examine the specifications and designs for inconsistencies, flaws, and vulnerabilities. This involves use cases that open the opportunity for abuse.

Complex integration risks. Web projects contain third-party dependencies and interact with APIs and code that are not under the developer's control. Auditors will review the project's external interactions and identify potentially dangerous behavior, such as implicitly trusting API responses or assuming third-party code functionality.

Code maturity. We review for possible improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also suggest possible improvements to code clarity, documentation, and usability.

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.

2.3 Scope

The engagement involved a review of the following targets:

Avara Abstracted Account Contracts

Repository	https://gitlab.com/aave-tech/avara-abstracted-account
Versions	`b55ceb36115a761dc29380fbf0fc3ad131753991`
Programs	*.ts, *.tsx, *.js, *.kt, *.c, *.swift
Type	Typescript, Javscript, Kotlin, C, Swift
Platform	Web, Mobile

2.4 Project Overview

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

Contact Information

The following project managers were associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Jasraj Bedi, Co-founder
jazzy@zellic.io

Maik Robert, Engineer
maik@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

November 22, 2022	Kick-off call
November 24, 2022	Start of primary review period
December 9, 2022	End of primary review period

3 Detailed Findings

3.1 Lack of rate limiting on validation codes

- **Target**: packages\server\src\services\validation-code.service.ts
- **Category**: Coding Mistakes
- **Severity**: Medium
- **Likelihood**: Low
- **Impact**: Low

Description

Many actions on the app require a validation code to be passed in before the action is carried out. This includes signup, sometimes on login and transactions/function calls not on the whitelist. The validation code lacks rate limiting or a way to track failed attempts; this could lead to an attacker bruteforcing these validation codes given the 15-minute expiry time, thereby bypassing the added security they provide.

```
export const validateCode = (
  username: Username,
  validationCode: ValidationCode,
  validationCodeSentFromStep: ValidationCodeSentFromStep
) => {
  return isValidCodeDb(username, validationCode,
    validationCodeSentFromStep);
};
```

```
const query = `
    SELECT *
      FROM validation_code_sent vcs
    WHERE vcs.email_address = $<username>
    AND vcs.validation_code = $<validationCode>
    AND vcs.triggered_from = $<validationCodeSentFromStep>
    AND vcs.invalid = false
    AND vcs.redeemed = false
    AND vcs.date_created_on + INTERVAL '15 minutes' > timezone('utc',
    now())
    LIMIT 1;
`;
```

```
export const isValidCodeDb = async (
  username: Username,
  validationCode: ValidationCode,
  validationCodeSentFromStep: ValidationCodeSentFromStep
): Promise<boolean> => {
  return exists(query, {
    username,
    validationCode,
    validationCodeSentFromStep,
  });
};
```

Impact

Bypassing the validation codes could lead to a wallet takeover if the attacker is already in possession of the username and password to the wallet.

Recommendations

Add a strict rate limit as well as track the amount of failed attempts for validation codes entered. If there have been too many failed attempts for a given code, take action like a temporary lockout. The same should be applied to 2FA codes.

Remediation

A comprehensive fix has been introduced in commit 5a1680d6. Strict rate limits have been introduced to prevent bruteforcing and have been applied to various flows throughout the code.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

4.1 iFrame setup

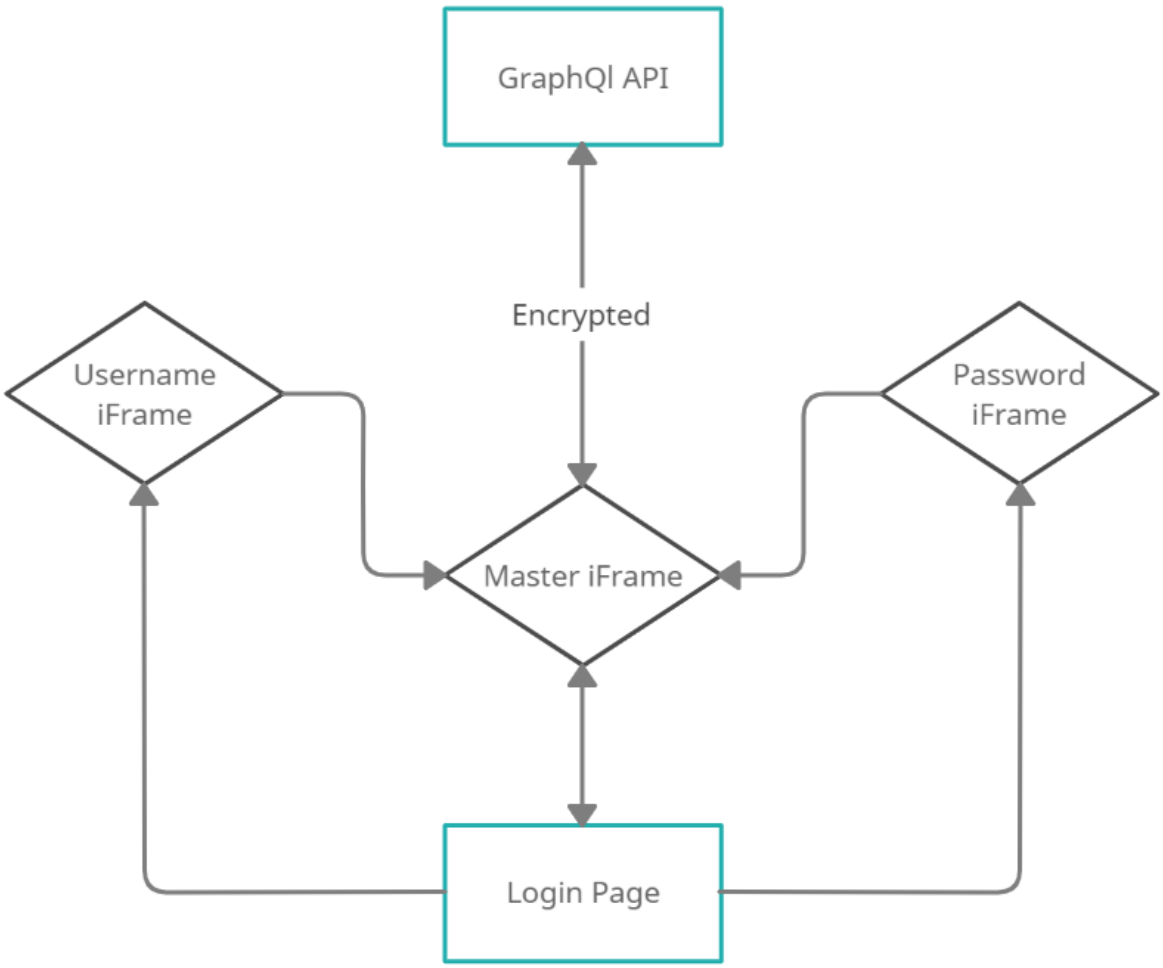


Figure 4.1: Example visualization of the iFrame setup.

To understand our security assumptions of the iFrame setup, we will look at the example of the login page, which is the first page the user interacts with. Upon browsing to the login page, the site spawns a master iFrame, which does the majority of the

communication, an iFrame that handles the user's password input, and an iFrame that handles the user's email/username input. The password and username iFrames pass the user's input to the master iFrame, which in turn sends an HTTP request to the GraphQl API and handles the response. The master iFrame communicates with the login page to update the display accordingly. The intention of this setup is memory separation of sensitive data as well as generally making it hard or impossible for an attacker to access the user's sensitive information or otherwise exploit vulnerabilities impacting the user's security. A few ways the security of this setup could be compromised are the following:

- cross-site scripting on the login page, which would most likely lead to a total compromise of the user's account
- malicious iFraming of the login page, leading to a clickjacking vulnerability
- direct attacks on the GraphQl API
- sniffing inter-iFrame communication

As is often the case in crypto applications, a cross-site scripting vulnerability can be devastating as it usually allows full control over the content displayed, actions taken as the victim/user, and in this case, abusing the trust of the iFrame setup by intercepting inter-iFrame communication. We could not identify any cross-site scripting vulnerabilities in the application. The team also took steps to mitigate possibilities of exploiting a cross-site scripting vulnerability by implementing a strong content security policy (CSP) as well as refraining from using dangerous sinks in React like "dangerouslySetInnerHtml".

iFraming the login page could trick a user into leaking sensitive information like a password to an attacker or taking actions the user was not meant to execute. Apart from the team's defense in-depth measures, like a strong CSP as well as planned security headers returned by the server, making iFraming the login page impossible, the setup itself is actually quite a good defense. As there are no cookies set to a domain or other identifiers in localStorage for example, an active "session" lives entirely in a single browser tab. This means the user has to log in again in another tab or an iFrame. This in itself already prevents an attacker from exploiting "authenticated" functions with a clickjacking attack.

Directly attacking the GraphQl is another option, for example with an SQL injection or bruteforcing. The impact of exploiting the server directly is severely limited by the core design of the server. Strong cryptography with recommended settings is used on the server side, making the data obtained by a potential breach mostly useless, as it should be impossible to crack user-relevant data. Zellic could not identify any SQL injections or dangerous usage of SQL statements using user-controlled input. One issue that was identified is that validation codes had no state tracking, meaning failed attempts were not tracked and a expiry time long enough to make bruteforcing them

potentially viable. This could have lead to a compromise of a wallet where an attacker had obtained a user's password by other means and was stopped by the validation code. The team will implement failed attempt tracking as well as a strict rate limit server side to make bruteforcing technically unfeasible. The same will be applied to 2FA codes.

At last, it was discussed if it would be possible to intercept the communication happening between iFrames, as this could leak sensitive data. We determined that apart from the mentioned cross-site scripting attack, it would require a flaw in the browser itself for this to be possible.

4.2 Encryption model

Despite the wallet communicating with a central server, it is non-custodial and zero knowledge. This means that the password is never sent to the server and the user has full control over the private keys and the currency residing in that wallet, without Avara Labs Ltd. having the ability to access the user's data. This allows the central storage of users' information, making it easy for end users to access their wallet and data from anywhere, while assuring that their data is still protected and safe from potential breaches or attacks. This setup also allows a user to recover their wallet via a previously saved offline recovery code, or a device key, which is generated on a known device the user has accessed the wallet from.

A few ways this could be attacked:

- Insecure generation of the master key
- Leakage of the user's password or master key
- Leakage of the user's offline recovery key or device key
- Insecure encryption used
- General logic flaws

The master key for the wallet is comprised of a user's email address and password. The application enforces strong password policies to prevent bruteforce attacks and safeguard the master key.

It is unlikely that a master key would be compromised unless there is a flaw in the browser, the computer is compromised, or there is physical access to the device, but this is beyond the control of the wallet. One known potential vulnerability that could make this possible is a cross-site scripting flaw, which could not be identified.

The device recovery keys are generated using 2048-bit RSA, which is currently impossible to be bruteforced and should be for the near future. The private key never leaves the user's device. The offline recovery keys use the same zero-knowledge ap-

proach, which makes it impossible for the team to maliciously decrypt your wallet. The generation uses sane standards, random 64-bytes, 100000 round PBKDF2, making a bruteforce attack unfeasable.

Zellic assessed that industry standard, and in some cases better than industry standard, encryption methods and settings are used. These provide sufficient protection against currently known feasible bruteforce and other attacks.

Other general logic flaws that could lead to the inability to decrypt data or the malicious decrypting of data could not be identified. Zellic tested the recovery of accounts using the device key and offline recovery key as well as the general encryption setup and found no way to attack or decrypt data in unintended ways.

The team also has excellent test coverage of the codebase, especially security relevant parts like the encryption flows.

The same applies to the iOS and Android native implementations written in Swift and Kotlin respectively. The mobile code follows the same design principles as the web version, but they are implemented natively for increased speed, leading to a better user experience. It has full passing test coverage of cryptographic functions, and Zellic could not identify issues, like the ones mentioned in the previous section for the web portion, relating to their implementation.

4.3 Transaction routing

A discussion arose regarding the appropriate method for transaction routing. One option is to have transactions routed through Avara Labs servers, while the current approach utilized by the wallet is to allow users to send transactions directly. A whitelist for methods and contracts has been implemented in the wallet that are deemed "safe"; every other contract interaction or method call that move funds require a validation code to be entered. This is to mitigate risks of bad actors and/or sites trying to send malicious transactions, for example a method call that moves all of the user's funds.

The approach of routing the calls through Avara servers could give the team greater control over identifying malicious behaviors and the ability to stop them before the end user is affected. However, there are a multitude of risks and additional workloads involved, some that may not be obvious at first. For one, it increases the attack surface of the wallet, which could introduce new and potentially critical bugs. It is another system that needs to be built, maintained, and audited, requiring additional engineering hours. There is also the potential for trust issues with end users if they are aware that their transactions are ultimately under the control of a third party, even if that party is trusted, leading to the reverse of the intended goal. Another potential hurdle could be a legal one. A legal team should be consulted before deciding

to implement the routing through Avara servers. The current implementation of the whitelist can be expanded upon to cover more cases with finer grained controls. The tradeoffs have to be weighed by the design team responsible for the implementation.

Zellic thinks that there is no correct answer to this design question. As it currently stands, expanding upon the whitelist approach, as well as adding finer grained controls and more cases, might be the safer option to prevent most regular users from falling victim to scams and malicious transactions.

4.4 General wallet security

The wallet employs a whitelist for methods and contracts that are allowed to be called without prompting for a validation code. Everything else, like methods that would move user funds, require a validation code to be entered. This should provide protection for users falling victim to scam attacks that drain a wallet's funds. In case a bad actor gets access to a wallet that is logged in, they would still need to enter a validation code to move funds, which they most likely do not have access to. A low-severity flaw regarding validation codes was identified and reported that could have lead to bypassing the validation code.

On signup when the seed phrase is generated, it is encrypted using the user's master key, which consists of the username, password, and a random 16-byte IV. The encrypted seed phrase and the IV are sent to the server for storage. This means that should the server be compromised, it is missing the master key to decrypt any data. In turn, if the master key of a user is leaked, an attacker would still need to proove possession of the username, in this case the email address, to receive the IV from the server that enables decrypting of the encrypted seed phrase. The setup ensures that an attacker would need both the user's master key and the contents of the server to successfully decrypt a victim's seed phrase.

It is evident that security was a key focus during the design process - not just in the server and wallet setup itself but also in trying to protect users from scams and malicious transactions. The use of zero-knowledge cryptography, the iFrame system adding additional security, and good fundamentals like high general test coverage and full test coverage of critical functions like the encryption is commendable.

5 Audit Results

At the time of our audit, the code was not deployed in production.

During our audit, we discovered one finding, which was low risk. Avara Labs Ltd. acknowledged all findings and implemented fixes.

5.1 Disclaimers

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 additional code added to the assessed project after the audit version of 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 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.

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.