Zellic

Privy Shamir Secret Sharing

Smart Contract Security Assessment

August 15, 2023

Prepared for:

Asta Li

Privy

Prepared by:

Mohit Sharma

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 Privy from June 27th to June 29th, 2023. During this engagement, Zellic reviewed Privy Shamir Secret Sharing'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 issues in the finite field arithmetic implementation?
- Are there any unhandled edge cases in split and combine functionalities?
- Is randomness used correctly to generate shares?

1.2 Non-goals and Limitations

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

- Front-end components
- Infrastructure relating to the project

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

1.3 Results

During our assessment on the scoped Privy Shamir Secret Sharing modules, we discovered one finding, which was of low impact. Privy acknowledged the finding and implemented a fix.

Additionally, Zellic recorded its notes and observations from the assessment for Privy'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 Privy Shamir Secret Sharing

Privy Shamir Secret Sharing is a zero-dependency TypeScript implementation of Shamir's Secret Sharing algorithm.

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 contracts.

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 the contracts' 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 code base 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.

2.3 Scope

The engagement involved a review of the following targets:

Privy Shamir Secret Sharing Modules

Repository	https://github.com/privy-io/shamir-secret-sharing/
Version	shamir-secret-sharing: `cd8422d`
Programs	• index.ts • csprng.ts • csprng.node.ts
Type	TypeScript
Platform	Browser

2.4 Project Overview

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

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:

Mohit Sharma, Engineer
mohit@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

June 27, 2023	Kick-off call
June 27, 2023	Start of primary review period
June 29, 2023	End of primary review period
July 5, 2023	Draft report delivered
August 15, 2023	Final report delivered

3 Detailed Findings

3.1 Missing input bounds in arithmetic functions

- **Target**: index.ts
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Medium
- **Impact**: Low

Description

The Privy Shamir Secret Sharing library implements the scheme over the Rijndael field, so all field elements have a representation in the integers 0–255. The arithmetic functions `add`, `div`, and `mult` do not have explicit bounds on the inputs. The functions `div` and `mult` have log table lookups, and the inputs to `add` are unbounded.

Therefore `div` and `mult` return undefined if any of the inputs exceed 255. This can be chained with the missing bound check in the `add` function. Because of JavaScript arithmetic quirks, if one of the inputs to `add` is `undefined`, it is treated as 0.

```
console.log(add(0, mult(257, 1)) == 0);
```

This breaks the `interpolatePolynomial` and `evaluate` functions. Both of these functions return 0 for any x > 255.

```
console.log(interpolatePolynomial(new Uint8Array([257, 258,
    259]),new Uint8Array([4, 2, 3]), 257) == 0);
```

Impact

During the assessment, we did not discover an exploitation scenario from the high-level API of the library. However, we still recommend adding the bounds as the check is inexpensive and improves overall code maturity.

Recommendations

We recommend enforcing explicit bounds in the arithmetic functions as so:

```
function add(a: number, b: number): number {
    if(!Number.isInteger(a) || a < 0 || a > 255)
    throw new RangeError("Number is out of Uint8 range");
    if(!Number.isInteger(a) || a < 0 || a > 255)
    throw new RangeError("Number is out of Uint8 range");
    return a ^ b;
}
```

Remediation

This issue has been acknowledged by Privy, and a fix was implemented in commit c9f0ab86.

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 Test suite

The test suite contains adequate testing for the APIs of the library, `split` and `combine`. However, there is no testing for the internal functions. When building a solid codebase, comprehensive testing is essential. This includes testing for both positive and negative scenarios. Positive tests should verify that each function's side effect is as expected, while negative tests should cover every revert, preferably in every logical branch.

The test coverage for this project should be expanded to include all functionality, not just exported functions. It is important to test the invariants required for ensuring security and also verify mathematical properties as per the spec.

Good test coverage has multiple effects.

- It finds bugs and design flaws early (preaudit or prerelease).
- It gives insight into areas for optimization.
- It displays code maturity.
- It bolsters customer trust in your product.
- It improves understanding of how the code functions, integrates, and operates — for developers and auditors alike.
- It increases development velocity long-term.

The last point seems contradictory, given the time investment to create and maintain tests. To expand upon that, tests help developers trust their own changes. It is difficult to know if a code refactor — or even just a small one-line fix — breaks something if there are no tests. This is especially true for new developers or those returning to the code after a prolonged absence. Tests have your back here. They are an indicator that the existing functionality *most likely* was not broken by your change to the code.

5 Audit Results

At the time of our audit, the audited code was deployed.

During our assessment on the scoped Privy Shamir Secret Sharing modules, we discovered one finding, which was of low impact. Privy acknowledged the finding and implemented a fix.

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 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 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.

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.