Zellic

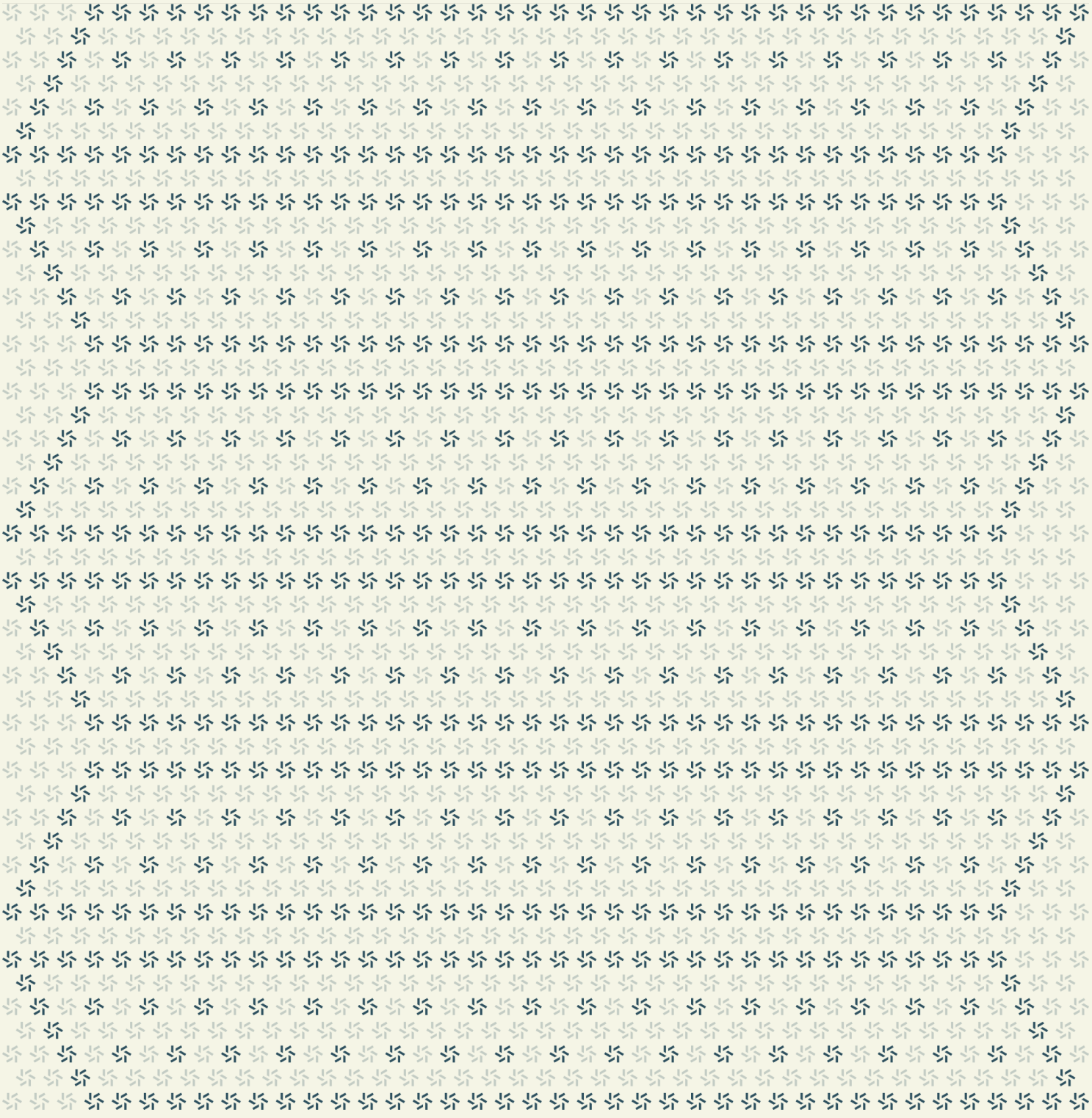
**Prepared for**
**Jorrit Palfner**
Light Protocol

**Prepared by**
**Frank Bachman**
**Jasraj Bedi**
**Junyi Wang**
Zellic

**September 6, 2024**

# Light Protocol

## Smart Contract Security Assessment



# Contents

## About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

# 1. Overview

## 1.1. Executive Summary

Zellic conducted a security assessment for Light Protocol from August 5th to September 6th, 2024. During this engagement, Zellic reviewed Light Protocol's code for security vulnerabilities, design issues, and general weaknesses in security posture.

## 1.2. 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 missing permission checks for `Signer`s?
- Do the programs follow the specification?
- Are there any problems with the phase calculations?
- Is there any way to cause a hash collision with the compressed accounts?

## 1.3. Non-goals and Limitations

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

- The security of the Poseidon hash function itself against various attacks
- Reliability and security of off-chain components
- Key custody

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

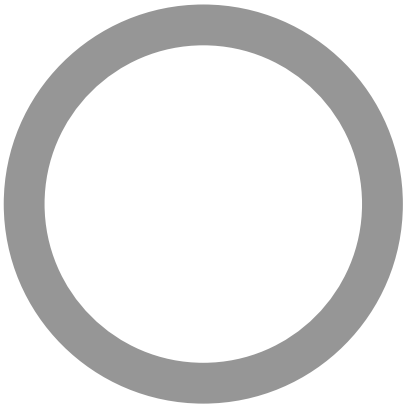
## 1.4. Results

During our assessment on the scoped Light Protocol programs, we discovered three findings, all of which were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Light Protocol's benefit in the Discussion section (4. ↗).

## Breakdown of Finding Impacts

| Impact Level | Count |
|---|---|
| Critical | 0 |
| High | 0 |
| Medium | 0 |
| Low | 0 |
| Informational | 3 |



# 2. Introduction

## 2.1. About Light Protocol

Light Protocol contributed the following description of Light Protocol:

> Light Protocol is a zero-knowledge infrastructure layer on Solana. Its ZK Compression primitive enables Solana developers to compress their on-chain state, reducing state costs while inheriting the security, performance, and composability of the Solana L1.

## 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 programs.

**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, upgradability 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 programs 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:

### Light Protocol Programs

| | |
|---|---|
| **Type** | Anchor |
| **Platform** | Solana |
| **Target** | light-protocol |
| **Repository** | https://github.com/Lightprotocol/light-protocol ↗ |
| **Version** | `2b4a6fadd7491f6a9f05479d0ebf09c96e4d8e8c` |
| **Programs** | `compressed-token`<br>`registry` |
| **Target** | light-protocol |
| **Repository** | https://github.com/Lightprotocol/light-protocol ↗ |
| **Version** | `a322268114d45621e7a23d5c1f0caa6925cebe02` |
| **Programs** | `system`<br>`account-compression` |

## 2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of six person-weeks. The assessment was conducted by three consultants over the course of four 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:

**Frank Bachman**
Engineer
frank@zellic.io

**Jasraj Bedi**
Co-Founder
jazzy@zellic.io

**Junyi Wang**
Engineer
junyi@zellic.io

## 2.5. Project Timeline

The key dates of the engagement are detailed below.

| | |
|---|---|
| **August 5, 2024** | Start of primary review period |
| **August 6, 2024** | Kick-off call |
| **September 6, 2024** | End of primary review period |

# 3. Detailed Findings

## 3.1. Dangerous memory-not-allocated check

| | | | |
|---|---|---|---|
| **Target** | process_mint | | |
| **Category** | Coding Mistakes | **Severity** | Informational |
| **Likelihood** | N/A | **Impact** | Informational |

### Description

There is the following vector allocation in the code.

```
let mut inputs = Vec::<u8>::with_capacity(inputs_len);
```

It is followed by the check below, which is meant to check that no memory was allocated by the `Vec`.

```
// # SAFETY: the inputs vector needs to be allocated before this point.
// This error should never be triggered.
if inputs.capacity() != inputs_len {
    msg!(
        "Used memory {} exceeds allocated {} memory",
        inputs.len(),
        inputs_len
    );
    return err!(crate::ErrorCode::HeapMemoryCheckFailed);
}
```

However, Rust's `Vec::with_capacity` can allocate extra memory according to the documentation, leading the check to fail. This quote ↗ is from the Rust documentation for `Vec::with_capacity`: "This method is allowed to allocate for more elements than `capacity`".

### Impact

Currently, the following code in `RawVecInner` is eventually used to allocate the `Vec`.

```
fn try_allocate_in(
    capacity: usize,
    init: AllocInit,
    alloc: A,
    elem_layout: Layout,
) -> Result<Self, TryReserveError> {
```

```
    // We avoid `unwrap_or_else` here because it bloats the amount of
    // LLVM IR generated.
    let layout = match layout_array(capacity, elem_layout) {
        Ok(layout) => layout,
        Err(_) => return Err(CapacityOverflow.into()),
    };

    // Don't allocate here because `Drop` will not deallocate when `capacity`
    is 0.
    if layout.size() == 0 {
        return Ok(Self::new_in(alloc, elem_layout.align()));
    }

    if let Err(err) = alloc_guard(layout.size()) {
        return Err(err);
    }

    let result = match init {
        AllocInit::Uninitialized => alloc.allocate(layout),
        #[cfg(not(no_global_oom_handling))]
        AllocInit::Zeroed => alloc.allocate_zeroed(layout),
    };
    let ptr = match result {
        Ok(ptr) => ptr,
        Err(_) => return Err(AllocError { layout, non_exhaustive: ()
    }.into()),
    };

    // Allocators currently return a `NonNull<[u8]>` whose length
    // matches the size requested. If that ever changes, the capacity
    // here should change to `ptr.len() / mem::size_of::<T>()`.
    Ok(Self { ptr: Unique::from(ptr.cast()), cap: unsafe { Cap(capacity) },
    alloc })
}
```

Note that the `capacity` allocated is exactly the `capacity` passed in, which coincidentally allows the allocation check above to work.

## Recommendations

Delete this check, or use a different method of detecting allocations.

### Remediation

This issue has been acknowledged by Light Protocol, and a fix was implemented in commit `043e22ad`.

## 3.2. The `HashSet` capacity can be unsafely set

| | | | |
|---|---|---|---|
| **Target** | hash-set | | |
| **Category** | Coding Mistakes | **Severity** | Informational |
| **Likelihood** | N/A | **Impact** | Informational |

### Description

The custom `HashSet` implementation contains a `Drop` implementation, which calls `alloc::dealloc(..)` when deallocating. The specific arguments, such as pointer and layout, are based on properties such as `HashSet::capacity` and `HashSet::buckets`. The call to `alloc::dealloc` needs to be matched with a similar call to `alloc::alloc` given the same `Layout` constraints. The issue is that `HashSet::capacity` is a public field which, after allocation, can be set manually.

### Impact

While this does not appear to currently affect the correctness of this protocol, it is a dangerous design pattern that can introduce undefined behavior very easily if done.

### Recommendations

Hide the `capacity` field behind a read-only function.

### Remediation

This issue has been acknowledged by Light Protocol, and a fix was implemented in commit `dd89e915` ↗.

### 3.3. The `HashSet` `from_bytes_zero_copy_mut` and similar allowance of unaligned read/write behavior

| | | | |
|---|---|---|---|
| **Target** | hash-set | | |
| **Category** | Coding Mistakes | **Severity** | Informational |
| **Likelihood** | N/A | **Impact** | Informational |

#### Description

The `HashSetZeroCopy` type implements a few `unsafe` methods such as `from_bytes_zero_copy_mut`. This method in particular creates a `HashSetZeroCopy` by deserializing the data passed in via `bytes: &'a mut [u8]`. When working with raw bytes like these, alignment is an important property. It is required that for a vast majority of read and write behavior, pointers must be aligned for the type; otherwise, undefined behavior can creep in. This function does not validate alignment of the data passed in.

That said, the `unsafe` function does mandate that the caller handle these properties ahead of time. It is our view that this method is still too dangerous, despite the warning, to use safely. And that, instead, it should internally check alignment before returning successfully. This is illustrated by `cargo miri`, identifying that the test case `test_hash_set_from_bytes_copy_6857_2400_3600` had unaligned pointer reads as a result of incorrect usage of this API.

#### Impact

The API is likely to be used incorrectly, which can easily result in undefined behavior.

#### Recommendations

Incorporate an alignment sanity check in the function that results in an error if the data is unaligned, making the function safer.

#### Remediation

This issue has been acknowledged by Light Protocol.

# 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. Risky heap allocator design

The `BumpAllocator` heap allocator in `heap/*` implements a risky design for bump allocation. By allowing control of the bump-allocator cursor via the "safe" method `free_heap`, it allows callers to find themselves in the position of allocating memory atop live references. This would result in undefined behavior. Consider, instead, leveraging a library such as bumpalo, which uses bump arenas as part of the design. The bumpalo library handles deallocation in two distinct ways: "all or nothing" and "last allocated".

If `bumpalo::Bump` is used as an arena, where all allocated values are the same type, and allocated via `Bump::alloc()`, then `Bump::reset()` can be used to completely "free" the entire bump allocator. This maintains safe access to references because allocation in this way requires an immutable reference to `Bump`, but a mutable reference to call `reset()`.

The second method for managing (de)allocation is via the `Allocator` API. This approach, which most closely resembles the use case used by this project, uses an implementation of the `Allocator` trait to enable the allocator to be used when allocating memory for arbitrary collections. Unlike the custom `BumpAllocator`, the `<Bump as Allocator>::deallocate(..)` method does support occasional deallocation to occur — but only in the special case where the request is for the last-known allocation.

## 4.2. Endianness in `HashSet`

The `HashSet` implementation in `hash-set/*` performs part of its deserialization logic by deserializing data with `::from_ne_bytes(..)`. While this may work, today, Borsh uses little-endian serialization regardless of platform preference. The suggestion to explicitly use `::from_le_bytes(..)` provides no net benefit in the immediate term but ensures it is future-proofed and more resilient to changes by Solana runtime and unique environments this code may run in (if, for example, a CI task were running on certain ARM environments).

This issue has been acknowledged by Light Protocol, and a fix was implemented in commit e499c4b6 ↗.

# 5. Assessment Results

At the time of our assessment, the reviewed code was not deployed to Solana mainnet.

During our assessment on the scoped Light Protocol programs, we discovered three findings, all of which were informational in nature.

## 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.