Zellic

Prepared for
Ruslan Tsechoev
Chirp Wireless

Prepared by
Filippo Cremonese
Zellic

June 6, 2024

Chirp Network

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 Chirp Wireless from June 4th to June 6th, 2024. During this engagement, Zellic reviewed Chirp Network'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:

- Is $CHIRP Token safe against unauthorized actions?
- Is the configurable minting schedule correctly implemented?
- Is the configurable minted-tokens distribution correctly implemented?
- Is authorization for administrative actions correctly enforced?
- Are the contracts implemented following best practices that allow future upgradeability?

1.3. 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
- Key custody
- Configuration of the schedule of $CHIRP Token emission and distribution at the point of deployment or in the future

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 Chirp Network modules, we discovered one finding, which was informational in nature.

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

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	0
🟨 Medium	0
🟩 Low	0
⬜ Informational	1

2. Introduction

2.1. About Chirp Network

Chirp Wireless contributed the following description of Chirp Network:

> The smart contract defines the CHIRP coin on the SUI blockchain, used to reward users of ChirpWireless-developed devices.

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, 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 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:

Chirp Network Modules

Type	Move
Platform	Sui

Target: blockchain

Repository	https://gitlab.com/chirpwireless/blockchain/blhn-sui-cntrt-ctkn ↗
Version	`cb5b37c2711b4cd3a3251f86162bd44d8f259e92`

Programs	`chirp.move`
	`schedule.move`
	`treasury.move`
	`pool_dispatcher.move`

2.4. Project Overview

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

Filippo Cremonese
Engineer
fcremo@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

June 3, 2024	Kick-off call
June 4, 2024	Start of primary review period
June 6, 2024	End of primary review period

3. Detailed Findings

3.1. Maximum supply cap is not immutable

Target	treasury.move		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Informational

Description

While the design requirements of the contracts include the ability for Chirp to define and change the issuance schedule and distribution policy of the $CHIRP tokens, they also specify that the maximum token supply should be decided and immutable.

The current version of the code contains safeguards preventing a change to the schedule configuration from minting an amount greater than the maximum supply value. Specifically, in `treasury.mint::mint_entry`, the check `assert!(amount <= (max_supply - coin::total_supply(cap)), EMintLimitReached);` prevents the contract from minting more than the configured maximum supply.

The maximum supply value is configured at deployment time and stored in the `Treasury` object.

We note that it is possible to bypass the maximum supply cap by upgrading the contracts.

Impact

This issue is reported as informational as we consider the current implementation to reflect the intended design.

Recommendations

The Sui Coin module does not allow to set a limit to the amount of Coins that can be minted by a `SupplyCap`. Given the current limitations of the Sui Coin module, all possible solutions to strictly enforce a maximum supply require relatively invasive code changes.

One possible design is minting the entire supply at the time the Coin is created, and storing the Coins in a contract which distributes them over time according to the schedule.

Alternatively, the `SupplyCap` could be handed to a non upgradable contract that manages it according to a policy that cannot be changed after deployment.

4. Discussion

Remediation

Chirp Network acknowledged this potential issue and opted to not address it via code changes to the modules. They declared their intent to entrust the capability to upgrade modules to the community via a governance vote, to ensure stakeholders have the opportunity to decide on possible future maximum supply changes.

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. Design requirements evaluation

In addition to checking for possible security issues, we evaluated several functional requirements derived from a design document shared by Chirp Wireless.

Administrative functionality

The design document shared by Chirp Wireless explicitly required the following administrative features.

Configurable minting schedule

The contracts should implement a configurable token emission and distribution schedule. The configuration can be set before the contracts are deployed and also changed at a later date. Therefore, our evaluation is limited to the functional correctness of the implementation of the minting schedule and not to the schedule configuration (including time durations and asset amounts).

The contracts allow to make changes to the schedule via the `set_entry`, `insert_entry`, and `remove_entry` functions. These functions require to provide a `ScheduleAdminCap` to authorize the action.

The minting schedule is defined as a series of `ScheduleEntry` objects:

```
public struct ScheduleEntry<phantom T> has store, drop {
    /// Start time for this entry in milliseconds, indicating when the
    /// entry was activated. This field can be null if the entry has not
    /// been activated yet.
    start_time_ms: std::option::Option<u64>,
    /// Current count of completed minting epochs within this entry. Each epoch
    /// represents a single minting operation under the defined parameters.
    current_epoch: u64,
    /// Defines the operational parameters for minting including the period,
    /// distribution pools, and the amounts to be minted for each pool over
    /// the specified number of epochs.
    stage: Stage<T>,
}
```

Each `ScheduleEntry` describes one stage in the schedule and consists of a number of epochs. Every epoch in a given stage has the same duration and distributes a configurable amount of assets to one or more pools. The epoch configuration is described in the `Stage` structure:

```
public struct Stage<phantom T> has store, copy, drop {
    /// Time shift in milliseconds from the end of the previous stage,
    /// setting the delay before this stage activates.
    timeshift_ms: u64,
    /// Number of epochs this stage will persist, each epoch represents a
    /// discrete minting event.
    number_of_epochs: u64,
    /// Duration of each epoch within this stage in milliseconds.
    epoch_duration_ms: u64,
    /// Addresses of the pools where the minted tokens are distributed.
    pools: vector<String>,
    /// Specific amounts of tokens to mint per pool per epoch.
    amounts: vector<u64>,
}
```

The `timeshift_ms` field can be set to force a delay between the end of the last epoch of the previous stage and the start of the first epoch of the current stage.

Administrative roles' ownership flexibility and security

Administrative roles of the contracts must be transferable.

The `ScheduleAdminCap` capability and the package-upgrade capabilities obtained by the deployer are freely transferable.

Configurable liquidity pool addresses

The contracts must support changing the addresses of the liquidity pools.

This is implemented by the `set_address_pool` function, which can be used to update the address of any liquidity pool by referring to it via its name. The action is authorized by providing a `ScheduleAdminCap` capability.

Multi-sig compatibility

Administrative capabilities and liquidity pools must be compatible with multi-sig addresses.

Sui supports k-of-n signatures natively, and it is possible to transfer capabilities to these multi-sig addresses; on-chain modules are oblivious to this and do not distinguish between regular signers and multi-sig signers. For more information, refer to Sui documentation pages on multi-sig:

- https://docs.sui.io/guides/developer/cryptography/multisig
- https://docs.sui.io/concepts/cryptography/transaction-auth/multisig

Pools should be able to be governed by a contract

In the version of the code under review, pools are configured as fixed regular addresses. It is an explicit requirement that pools should be able to be governed by a smart contract.

This is possible via the Sui transfer-to-object feature. Since pools take ownership of `Coin<T>` objects shared using `transfer::public_transfer`, the address of a pool can be replaced with the address of any object. The module defining the pool object can call `receive` to obtain the transferred `Coin<T>` and implement any access-control logic and `Coin` redistribution policy as desired.

Upgradeability

The contracts should be future proof and support being upgraded.

Sui contracts are immutable in nature; once published, a given version of the code of a module is permanently stored and always available for any user. However, newer versions of a package can be published and used to access the previous state of the program. Chirp correctly implements the most common and recommended Sui upgradeability pattern, versioned shared objects. In this pattern, objects storing the shared state of the contract are augmented with a version field, which is checked by the contract code to ensure it matches the expected version. New contract upgrades change the version field, effectively disabling older versions of the code and forcing the adoption of the newest published version.

For more information about upgradeability, refer to https://docs.sui.io/concepts/sui-move-concepts/packages/upgrade#versioned-shared-objects.

Keeper rewards workflow

The design requirements specify that keeper rewards are to be minted according to a configurable schedule that regulates the rate of emission and the distribution of the minted $CHIRP tokens to the various pools.

Minting the tokens allocated to any given epoch is an unpermissioned action that can be initiated by anyone by calling the `mint` function. The amount of tokens allocated as keeper rewards by the schedule are to be sent to a pool address, which is controlled by Chirp Network. Chirp will distribute rewards to individual keepers according to off-chain calculations.

This is achieved via the `deposit` function, which stores a given amount of coins associated to the address of the respective recipient. Coins held by the depository can be claimed fully or in part, exclusively by their recipient, by invoking `claim`. Multiple deposits can occur before a recipient claims their assets.

The contracts correctly implement the flow described by the requirements.

5. Threat Model

This provides a description of the public functions implemented by the in-scope modules. As time permitted, we analyzed each function in the modules and created a written threat model. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

5.1. Module: chirp.move

The chirp module defines the $CHIRP Token as a standard Sui coin.

It also defines publicly accessible functions used to initiate all on-chain actions.

The module defines a shared `Vault` object, which maintains an `ObjectBag` (a named mapping) containing other objects defined by the other modules in the system. Existing objects can be replaced and the mapping can be extended with additional objects, allowing future changes. Currently, the vault object owns the following:

- A `Treasury`, which contains the capability required to mint new CHIRP, and also the schedule configuration
- A `PoolDispatcher`, which contains a `Bag` associating pool names with their effective addresses
- A depository, an `ObjectTable` containing CHIRP tokens given as rewards to keepers — the table key is the address of the keeper that can claim the rewards

Function: `mint`

This function can be called to mint $CHIRP according to the schedule. Anyone can call the function and cause tokens to be minted and sent to the pools, provided the schedule allows to mint a new epoch.

Function: `set_entry`

This function can be used to replace a schedule entry. Only future or current schedule entries can be altered, as the schedule for previous mints cannot be changed. The function can only be called by providing a `ScheduleAdminCap`.

Note that the check for which table entry is current is not based on the current time but rather on the latest active schedule entry. The active schedule entry is only changed when minting occurs, and therefore if no minting were to occur for a long time, the last schedule entry would remain active and could be changed. Since minting is an unpermissioned action, we consider this an intended design decision with no significant security impact.

Function: `insert_entry`

This function can be used to insert a new schedule entry. The new schedule entry can only be inserted in a future position. The function can only be called by providing a `ScheduleAdminCap`.

The consideration expressed for `set_entry` regarding how the currently active entry is computed applies for this function as well.

Function: `remove_entry`

This function can be used to remove a schedule entry. Only future schedule entries can be removed. The function can only be called by providing a `ScheduleAdminCap`.

The consideration expressed for `set_entry` regarding how the currently active entry is computed applies for this function as well.

Function: `set_address_pool`

This function can be used to set an entry in the address pool, changing which address is associated to a named pool. The function can only be called by providing a `ScheduleAdminCap`.

The function cannot be used to add new named pools but only to change the address of an existing one.

Function: `claim`

This function can be used by keepers to claim $CHIRP rewards from the depository. The caller can specify an amount of tokens to claim, and therefore it is free to claim their full balance or make just a partial claim.

The claimed $CHIRP tokens are directly transferred to the address of the caller; therefore, the function is not exposed to social engineering or confused deputy attacks.

Function: `deposit`

This unpermissioned function is intended to be used by Chirp to distribute rewards to keepers. The function accepts a list of recipients and $CHIRP coins and stores the coins in the depository associated with their designated recipient. The rewards can then be claimed using the `claim` function.

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Sui Mainnet.

During our assessment on the scoped Chirp Network modules, we discovered one finding, which was informational in nature.

6.1. 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.