Zellic

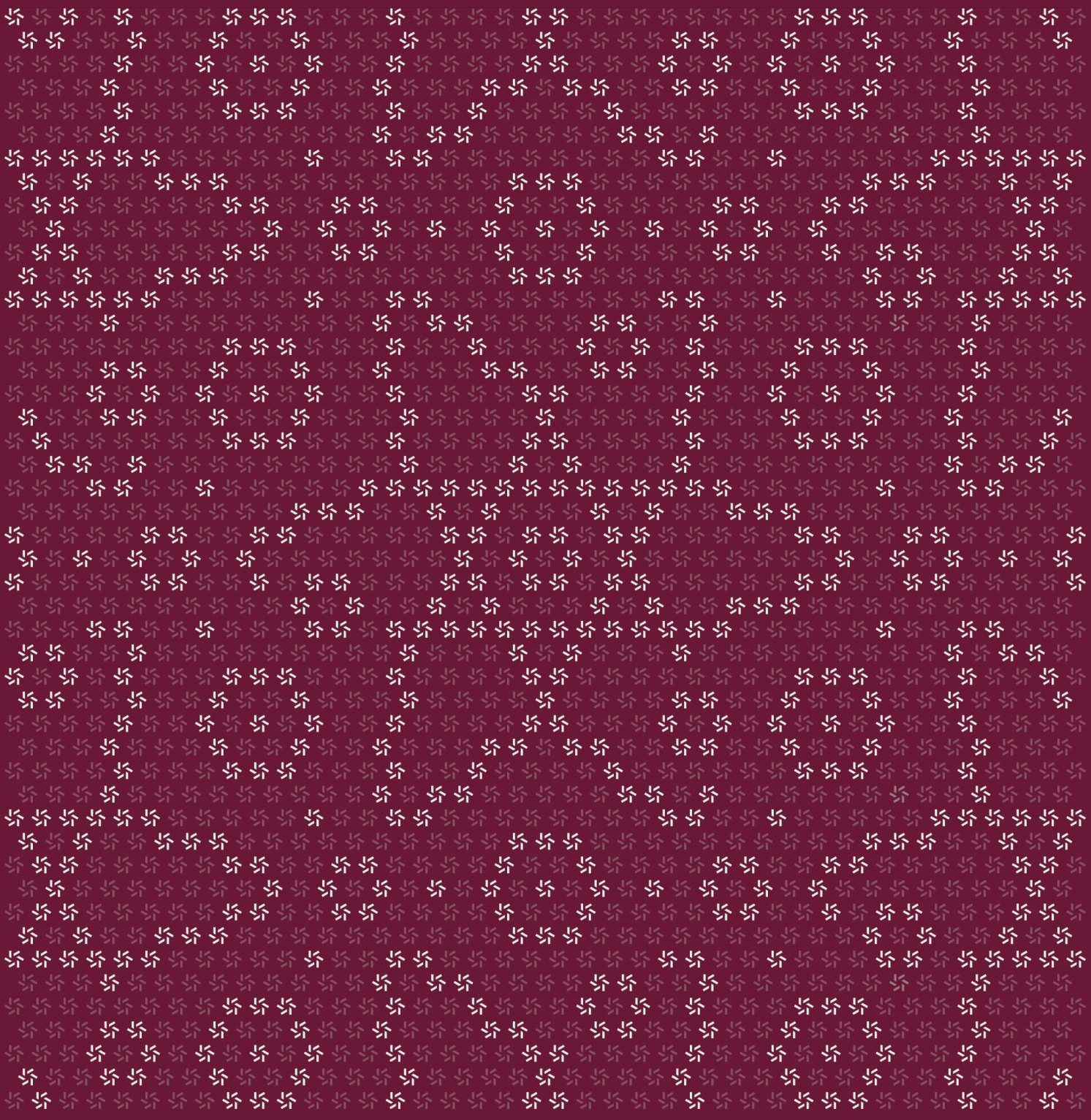
Prepared for
Tristan Litre
Nillion

Prepared by
Frank Bachman
Zellic

June 21, 2024

Nilchain

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 Nillion from June 18th to June 19th, 2024. During this engagement, Zellic reviewed Nilchain'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:

- Could an attacker store a resource as originating from an arbitrary user?
- Could an attacker delete or modify a stored resource?

1.3. Non-goals and Limitations

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

- Infrastructure relating to the project
- Key custody

1.4. Results

During our assessment on the scoped Nilchain contracts, there were no security vulnerabilities discovered.

Zellic recorded its notes and observations from the assessment for Nillion's benefit in the Discussion section (3. ↗).

Breakdown of Finding Impacts

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

2. Introduction

2.1. About Nilchain

Nillion contributed the following description of Nilchain:

> Nillion's Coordination Layer (Nilchain) keeps the global shared state of the whole network and facilitates payments, cryptoeconomics, and governance. The custom module is used specifically to pay for requests made to other side of the network, the Petnet.

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 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 contracts itself. These observations — found in the Discussion (3. ↗) 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:

Nilchain Contracts

Type	Go
Platform	Cosmos
Target	nilchain
Repository	https://github.com/NillionNetwork/nilchain ↗
Version	`38fb884dd2053cb979b0edc5cbcab0cc910edf60`
Programs	`x/meta`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 1 person-day. The assessment was conducted by one consultant over the course of 2 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:

Frank Bachman
Engineer
frank@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

June 18, 2024	Start of primary review period
June 19, 2024	End of primary review period

3. 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.

3.1. Meta Module

Nilchain is the Coordination layer for the Nillion network and has only one module, Meta. The Meta module is designed to hold metadata related to resources paid for in the PET network. The store for the Meta module maintains only a `Resources` Map in it's keeper. The Map implementation used is from the cosmos-sdk collections library, which is safe against any non-deterministic behaviour. The data stored in the store is not itself verified by the module, and the module has no concept as to what data is valid/invalid. It simply stores raw bytes as provided by the user.

For this the module implements the `MsgPayFor` message which is externally accessible by users.

`MsgPayFor`

The message has three parameters:

1. `from_address`: This account is required to be a signer and is used to pay for the resources.
2. `amount`: This is the amount of the resources being paid for.
3. `resource`: This is just an arbitrary byte array.

The functionality implemented is pretty straightforward. The amount sent by the user is transferred to the module and then burnt. The `resource` provided is stored in the module through `keeper.SaveResource`.

`SaveResource`

The `SaveResource` handler in the keeper is used by `MsgPayFor` to store a user-provided `resource`. The keeper consists of a Map, which stores all resources. To generate the key, `forgeResourceKey` is used. This simply generates a SHA-256 of the `resource`. A tuple of the account address and the hash is returned and used as the Map key.

Note that the Map key generated is always unique for a given resource and user address. It is therefore not possible to remove or overwrite any submitted resources from the map.

Arbitrary resource insertions through `InitGenesis`

The `InitGenesis` method is executed during `InitChain` when the application is first started. It uses `keeper.SaveResource` to pre-initialize the store with resources. Moreover, the `ExportGenesis` han-

dler can be used to export the application state. It uses `keeper.IterateResources` to enumerate and return all resources along with their addresses.

The `InitGenesis` handler allows insertion of resources into the store without requiring a user to sign the message. In other words, resources can be stored as originating from arbitrary accounts. In the usual scenario, this would just be data exported through `ExportGenesis`. This a requirement for upgrades through hard forks, and to recover the chain from any disasters. However since the state provided to `InitGenesis` is not guaranteed to have been exported from `ExportGenesis`, this poses certain centralization risk. We recommend that such risks are explicitly documented, allowing users to make informed decisions. It is important to note that this can only be done during chain initialisation and it's possible for users to easily verify such additions.

4. Assessment Results

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

During our assessment on the scoped Nilchain contracts, there were no security vulnerabilities discovered.

4.1. 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.