Zellic

Prepared for
Alan Wang
Kevin Yu
Clique

Prepared by
SeungHyeon Kim
Sylvain Pelissier
Zellic

December 23, 2024

Bera Contracts

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 Clique from December 16th to December 20th, 2024. During this engagement, Zellic reviewed Bera Contracts'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 the paymaster implemented well?
- Is the implementation of communication between the paymaster and smart contracts designed safely?
- Could an attacker cause a denial of service (DoS) on the smart contracts?
- Could an attacker claim multiple times on the smart contracts?

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

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

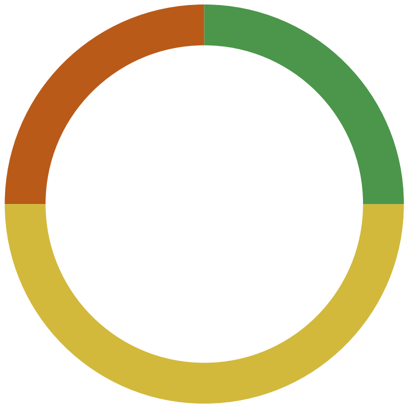
1.4. Results

During our assessment on the scoped Bera Contracts, we discovered four findings. No critical issues were found. One finding was of high impact, two were of medium impact, and one was of low impact.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Clique in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	1
🟨 Medium	2
🟩 Low	1
⬜ Informational	0



2. Introduction

2.1. About Bera Contracts

Clique contributed the following description of Bera Contracts:

> This is a protocol for managing NFT-based vesting streams and social verification rewards on Berachain through a paymaster implementation.

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 (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:

Bera Contracts

Type	Solidity
Platform	EVM-compatible

Target	bera-contracts
Repository	https://github.com/CliqueOfficial/bera-contracts ↗
Version	`4ee236909df1fa7dfda25986d17750159dfded01`

Programs	`ClaimBatchProcessor`
	`Distributor1`
	`StreamingNFT`
	`Transferable`
	`WrappedNFT`
	`paymaster/src/main.rs`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 7.5 person-days. The assessment was conducted by two consultants over the course of one calendar week.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io ↗

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

SeungHyeon Kim
Engineer
seunghyeon@zellic.io ↗

Sylvain Pelissier
Engineer
sylvain@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

December 16, 2024	Kick-off call
December 16, 2024	Start of primary review period
December 20, 2024	End of primary review period

3. Detailed Findings

3.1. Missing validation check on ERC-20 transfer

Target	Transferable		
Category	Coding Mistakes	Severity	Medium
Likelihood	Low	Impact	Medium

Description

The Transferable contract is used to handle native and ERC-20 tokens with a common contract. The native token is detected if the `token` address is zero. The `transfer` function checks if the token address is zero and uses either `call` or the ERC-20 `transfer` function to transfer tokens:

```
function transfer(address recipient, uint256 amount) internal {
        if (token == address(0)) {
            (bool success,) = recipient.call{value: amount}("");
            require(success, "Native token transfer failed");
        } else {
            IERC20(token).transfer(recipient, amount);
        }
    }
```

However, the success of the ERC-20 transfer operation is not verified. If the transfer fails, the `transfer` function will not revert, potentially leading to unexpected behavior or silent failures.

Impact

If a transfer fails during an airdrop claim, the airdrop will still be marked as claimed, but the corresponding amount will not be transferred, leading to a loss of funds. A similar issue occurs during stream creation and reward claiming, where failed transfers result in the amounts not being transferred as expected.

Recommendations

To mitigate this issue, the `safeTransfer` function from the SafeERC20 library can be used in place of `transfer`. Additionally, a test should be implemented to verify these scenarios, ensuring that the contract handles failed transfers correctly and behaves as expected under such conditions.

Remediation

This issue has been acknowledged by Clique, and a fix was implemented in commit 5dd44079 ↗. A check on the returned value was implemented, and the `transfer` function reverts in case the ERC-20 transfer fails.

3.2. Lack of comprehensive test suite

Target	All Scoped Contracts		
Category	Code Maturity	**Severity**	Medium
Likelihood	N/A	**Impact**	Medium

Description

When building a project with multiple moving parts and dependencies, 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 project currently lacks a test suite, with the sole exception being the StreamingNFTPool contract. However, for StreamingNFTPool, the tested use cases are minimal and do not test more complex scenarios. For example, the tests related to batch-stream creation implement only a case for a single token and not multiple, as supported by the function. The `claimBatchRewards` function is not tested at all. It is important to test the invariants required for ensuring security.

Impact

This code lacks a comprehensive test suite, increasing the likelihood of potential bugs.

Recommendations

We recommend building a rigorous test suite to ensure that the system operates securely and as intended.

Good test coverage has multiple effects.

- It finds bugs and design flaws early (preaudit or prerelease).
- It gives insight into areas for optimization (e.g., gas cost).
- 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.

Remediation

This issue has been acknowledged by Clique.

3.3. Centralization risk

Target	Distributor1 and StreamingNFT		
Category	Code Maturity	**Severity**	Critical
Likelihood	Low	**Impact**	High

Description

There are two types of privileged accounts for the Distributor1 contract: the owner and the signer. The signer address is only used to verify the signature before allowing airdrop claims.

For the Distributor1 contract, the owner can change the `claimRoot` value with the `setClaimRoot` function and render all the previous proof of inclusions invalid. Additionally, the owner has the ability to withdraw all funds from the contract using the `withdraw` function.

For the StreamingNFT contract, there is only one privileged address, the owner. The owner can change the value of the fees with the `setFee` function to a very high value, making the stream creation and reward-amount values almost null. Additionally, the owner has the ability to withdraw all funds from the contract using the `withdraw` function.

Impact

The above introduces centralization risks that users should be aware of, as it grants a single point of control over the system. If a malicious user gains access to the owner's private key, they can withdraw all funds.

Recommendations

We recommend that these centralization risks be clearly documented for users so that they are aware of the extent of the owner's control over the contracts. This can help users make informed decisions about their participation in the project. Additionally, clear communication about the circumstances in which the owner may exercise these powers can help build trust and transparency with users. Therefore, it is recommended to implement additional measures to mitigate these risks, such as implementing a multi-signature requirement for owner access.

Remediation

This issue has been acknowledged by Clique.

3.4. Fee transfers occur without owner consent and can be front-run

Target	StreamingNFT		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `createBatchStream` and `createStream` functions are used to create streams for the owner of NFTs. In case the NFTs are not owned by the caller of the functions, some fees are transferred to the address that originated the transaction:

```
function createStream(uint256 tokenId) external nonReentrant whenNotPaused {
    require(block.number >= vestingStartBlock, "Vesting has not started yet");
    require(claimedBlock[tokenId] == 0, "Stream already created");

    claimedBlock[tokenId] = vestingStartBlock;

    address onbehalfOf = credentialNFT.ownerOf(tokenId);
    uint256 instantAmount = (allocationPerNFT * instantUnlockPercentage)
    / 100;

    uint256 gasFee = 0;
    if (tx.origin != onbehalfOf) {
        gasFee = fee;
        require(gasFee != 0, "Gas fee not set");
        instantAmount -= gasFee;
        transfer(tx.origin, gasFee);
    }

    transfer(onbehalfOf, instantAmount);
    emit StreamCreated(tokenId, onbehalfOf, allocationPerNFT, gasFee);
}
```

If someone calls this function on behalf of the NFT owner before the owner does, they receive a fee (`gasFee`) deducted from the owner's allocation (`instantAmount`). This could reduce the owner's intended allocation without their consent.

Additionally, anyone monitoring the mempool could spot a transaction calling `createBatchStream` or `createStream` and propose an identical transaction with a higher gas fee to front-run it, capturing the fee for themselves.

Impact

The NFT owner's allocation is reduced without their consent, and the fee can be captured by generalized MEV bots.

Recommendations

Consider implementing a time window during which only the owner can call the functions. The owner must call the functions within a week (seven days or 86,400 blocks from `vestingStartBlock`); otherwise, the function becomes open to any caller.

Remediation

This issue has been acknowledged by Clique.

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. Enforce a more recent Solidity version

Some contracts specify Solidity version ^0.8.0 to be used, which allows compilation with any version of Solidity from 0.8.0 up to the latest release. If possible, enforcing the use of the latest Solidity version (at the time of writing, ^0.8.28) would prevent introducing bugs present in the previous versions and may add the latest optimizations to the contract, like the usage of custom errors instead of `require`.

4.2. The paymaster has a simple typo on the endpoint

There seems to be a typo in the endpoint configuration. The `/healthz` endpoint is expected to be hosted at `/health` based on the Dockerfile.

```
// Setup router
let app = Router::new().route("/healthz", get(health_check)).route(
    "/batch-claim",
    post(move |payload| handle_batch_claim(payload,
    batch_processor_contract.clone())),
);
```

```
# Add healthcheck
HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 \
    CMD curl -f http://localhost:3000/health || exit 1
```

Clique provided following response:

> `/healthz` is the right endpoint.

We confirmed that this issue is resolved with the commit `0439d696`.

5. System Design

This provides a description of the high-level components of the system and how they interact, including details like a function's externally controllable inputs and how an attacker could leverage each input to cause harm or which invariants or constraints of the system are critical and must always be upheld.

Not all components in the audit scope may have been modeled. The absence of a component in this section does not necessarily suggest that it is safe.

5.1. Paymaster

The paymaster is a Rust project that supports interaction between the smart contract ClaimBatchProcessor. This routes the endpoints `[GET] /healthz` and `[POST] /batch-claim`.

The `[GET] /healthz` endpoint is for checking the health of the paymaster.

The `[POST] /batch-claim` endpoint is for batching the claims. For this endpoint, it requires the `application/json` type data as the body, and the JSON must have the below values.

- `amount`
- `proof`
- `signature`
- `tokenIds`
- `userAddress`

These values will be passed to the function `claim` on the ClaimBatchProcessor contract. So, the flow will be like this.

```
User
=> POST `/batch-claim` to the Paymaster
=> `ClaimBatchProcessor`'s `claim()`
=> Distributor's `claim()`
=> StreamingNFT's `createBatchStream()`
```

6. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

6.1. Module: Distributor1.sol

Function: `claim(bytes32[] calldata _proof, bytes calldata _signature, uint256 _amount, address _onBehalfOf)`

The `claim` function is used to claim airdrops. The function receives and verifies a signature and a proof of inclusion of the claim.

Inputs

- `_proof`
    - **Validation**: Check that the amount and the account were previously included in the airdrop.
    - **Impact**: Prevents nonincluded airdrop being claimed.
- `_signature`
    - **Validation**: The address that signed the contract address — the amount and the account of the claim should match the `signer` address.
    - **Impact**: Prevents unauthorized claims.
- `_amount`
    - **Validation**: The amount has to be smaller than the contract balance.
    - **Impact**: The airdrop amount to transfer.
- `_onBehalfOf`
    - **Validation**: Check if the claim was already done.
    - **Impact**: Prevents a claim being executed twice.

Branches and code coverage (including function calls)

Intended branches

- A claim properly signed and included in the root of the Merkle tree is successfully distributed.
    - ☐ Test coverage

Negative behavior

- A claim already distributed is rejected.

- ☐ Test coverage
- A claim with an invalid signature is rejected.
  - ☐ Test coverage
- A claim with an invalid inclusion proof is rejected.
  - ☐ Test coverage
- A transfer failure during a claim properly reverts.
  - ☐ Test coverage

Function call analysis

- `claim -> _rootCheck(_proof, _amount, _onBehalfOf)`
  - **External/Internal?** Internal.
  - **Argument control?** All arguments.
  - **Impact**: Checks the proof of inclusion of the amount and the account.
- `_signatureCheck(_amount, _signature, _onBehalfOf);`
  - **External/Internal?** Internal.
  - **Argument control?** All arguments.
  - **Impact**: Checks the signature of the amount and the account.
- `transfer(_onBehalfOf, amount)`
  - **External/Internal?** Internal.
  - **Argument control?** All arguments.
  - **Impact**: Transfers the airdrop to the account.

6.2. Module: StreamingNFT.sol

Function: `claimBatchRewards(uint256[] calldata streamIds)`

The function is used to transfer the rewards for a batch of NFTs until the vesting period is finished.

Inputs

- `streamIds`
  - **Validation**: A stream must be created for each identifier to claim rewards.
  - **Impact**: It prevents claiming rewards for streams that have not started.

Branches and code coverage (including function calls)

Intended branches

- Rewards are claimed by the owner of the NFTs for streams already created. The vested awards are properly transferred to the beneficiary.
  - ☐ Test coverage

- Rewards are claimed by a third party for streams already created. The vested awards are properly transferred to the beneficiary.
  - ☐ Test coverage

Negative behavior

- Claiming rewards during the same block gives zero rewards.
  - ☐ Test coverage
- Claiming rewards for invalid NFTs reverts.
  - ☐ Test coverage
- Claiming rewards after the vesting period gives zero rewards.
  - ☐ Test coverage
- A transfer failure makes the transaction revert.
  - ☐ Test coverage

Function call analysis

- `_claimVestedRewards(streamIds[i], vestingEndBlock_)`
  - **External/Internal?** Internal.
  - **Argument control?** `streamIds`.
  - **Impact**: Computes and returns the value of the rewards depending on the current block number.

Function: `claimRewards(uint256 streamId)`

The function is used to transfer the rewards to the NFT owner until the vesting period is finished.

Inputs

- `streamId`
  - **Validation**: A stream must be created to claim rewards.
  - **Impact**: It prevents claiming rewards for a stream that has not started.

Branches and code coverage (including function calls)

Intended branches

- Rewards are claimed by the owner of the NFT for a stream already created. The vested awards are properly transferred to the beneficiary.
  - ☑ Test coverage
- Rewards are claimed by a third party for a stream already created. The vested awards are properly transferred to the beneficiary.
  - ☐ Test coverage

Negative behavior

- Claiming rewards during the same block gives zero rewards.
  - ☐ Test coverage
- Claiming rewards for an invalid NFT reverts.
  - ☐ Test coverage
- Claiming rewards after the vesting period gives zero rewards.
  - ☐ Test coverage
- A transfer failure makes the transaction revert.
  - ☐ Test coverage

Function call analysis

- `_claimVestedRewards(streamId, vestingEndBlock_)`
  - **External/Internal?** Internal.
  - **Argument control?** `streamId`.
  - **Impact**: Computes and returns the value of the rewards depending on the current block number.

Function: `createBatchStream(uint256[] calldata tokenIds, address onBehalfOfOwner)`

The function create a stream for each token in `tokenIds`. An aggregate allocation is transferred to the owner of the NFT, and a fee is transferred to the caller if they are not the NFT's owner.

Inputs

- `tokenIds`
  - **Validation**: The current block number is checked to be greater than `vestingStartBlock`. The `tokenId` is checked to be a valid `credentialNFT` by the `ownerOf` function.
  - **Impact**: It prevents the creation of a stream twice or the creation of a stream for a non-existing NFT.
- `onBehalfOfOwner`
  - **Validation**: Each stream is checked to be owned by the `onBehalfOfOwner` address and to have not being already created.
  - **Impact**: It prevents the creation of a stream for a token not owned by `onBehalfOfOwner`.

Branches and code coverage (including function calls)

Intended branches

- A stream is created by the owner of the NFT from a token array. The instant amount is properly transferred.
  - [x] Test coverage
- A stream is created by another user. The instant amount and the fees are properly transferred.
  - [x] Test coverage

Negative behavior

- A stream created for the second time reverts.
  - [ ] Test coverage
- A stream created before the vesting starts reverts.
  - [ ] Test coverage
- A stream created for an array containing an invalid identifier reverts.
  - [ ] Test coverage
- A transfer failure during a stream creation reverts.
  - [ ] Test coverage

Function call analysis

- `transfer(onbehalfOf, instantAmountAccum)`
  - **External/Internal?** Internal.
  - **Argument control?** No.
  - **Impact**: Transfers the fees.
- `transfer(tx.origin, payMasterFeeAccum)`
  - **External/Internal?** Internal.
  - **Argument control?** No.
  - **Impact**: Transfers the amount of the stream creation.

Function: `createStream(uint256 tokenId)`

The function creates a stream for the owner of a `credentialNFT`. An allocation is transferred to the owner of the NFT, and a fee is transferred if the caller of the function is not the NFT's owner.

Inputs

- `tokenId`
  - **Validation**: A stream was not already created for the current token identifier. The current block number is checked to be greater than `vestingStartBlock`. The `tokenId` is checked to be a valid `credentialNFT` by the `ownerOf` function.
  - **Impact**: It should prevent the creation of a stream twice or the creation of a stream for a nonexisting NFT.

Branches and code coverage (including function calls)

Intended branches

- A stream is created by the owner of the NFT. The instant amount is properly transferred.
  - ☑ Test coverage
- A stream is created by another user. The instant amount and the fees are properly transferred.
  - ☑ Test coverage

Negative behavior

- A stream created for the second time reverts.
  - ☑ Test coverage
- A stream created before the vesting starts reverts.
  - ☑ Test coverage
- A stream created for an invalid identifier reverts.
  - ☐ Test coverage
- A transfer failure during a stream creation reverts.
  - ☐ Test coverage

Function call analysis

- `transfer(tx.origin, gasFee)`
  - **External/Internal?** Internal.
  - **Argument control?** No.
  - **Impact**: Transfers the fees.
- `transfer(onbehalfOf, instantAmount)`
  - **External/Internal?** Internal.
  - **Argument control?** No.
  - **Impact**: Transfers the amount of the stream creation.

7. Assessment Results

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

During our assessment on the scoped Bera Contracts, we discovered four findings. No critical issues were found. One finding was of high impact, two were of medium impact, and one was of low impact.

7.1. 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.