Zellic

Highlight

Smart Contract Security Assessment

May 22, 2022

Prepared for:

Kevin Matthews

Highlight, Inc.

Prepared by:

Ayaz Mammadov and Vlad Toie

Zellic Inc.

Contents

About Zellic 2

1 Introduction 3

1.1 About Highlight . . . 3

1.2 Methodology . . . 3

1.3 Scope . . . 4

1.4 Project Overview . . . 5

1.5 Project Timeline . . . 6

1.6 Disclaimer . . . 6

2 Executive Summary 7

3 Detailed Findings 8

3.1 Ownership transfer may be erroneous. . . . 8

3.2 Lack of input validation . . . 10

4 Discussion 12

4.1 MetaTXs . . . 12

4.2 0xSplits changes . . . 12

4.3 OpenZeppelin changes . . . 12

4.4 Gas optimizations . . . 13

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 email us at hello@zellic.io or contact us on Telegram at https://t.me/zellic_io.

1 Introduction

1.1 About Highlight

Highlight offers no-code tools and infrastructure to help creators and artists design and mint NFTs and build custom-branded membership communities in web3.

1.2 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 open-source 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. We analyze the scoped smart contract code using automated tools to quickly sieve out and catch these “shallow” bugs. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, etc. 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 manually review the contract logic to ensure that the code implements the expected functionality as specified in the platform’s design documents. We also thoroughly examine the specifications and designs themselves for inconsistencies, flaws, and vulnerabilities. This involves use-cases that open the opportunity for abuse, such as flawed tokenomics or share pricing, arbitrage opportunities, etc.

Complex integration risks. Several high-profile exploits have been the result of not any bug within the contract itself, but rather an unintended consequence of its interaction with the broader DeFi ecosystem. We perform a meticulous review of all of the contract’s possible external interactions, and summarize the associated risks; for example: flash loan attacks, oracle price manipulation, MEV/sandwich attacks, etc.

Code maturity. We review for possible improvements in the codebase in general. We look for violations of industry best practices and guidelines, or code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, etc.

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; we assign it on a case-by-case basis based on our professional judgment and experience. As one would expect, both the severity and likelihood of an issue affect its impact; for instance, a highly severe issue's impact may be attenuated by a very low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Similarly, Zellic organizes its reports such that the most important findings come first in the document, rather than 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 numerous soft factors, such as our clients' threat models, their business needs, project timelines, etc. We aim to provide useful and actionable advice to our partners that consider their long-term goals, rather than simply a list of security issues at present.

1.3 Scope

The engagement involved a review of the following targets:

Highlight Core Contracts

Repository	https://github.com/highlightxyz/hl-contracts
Versions	`8ce54987e940d359616d34112c24b0c783458705`
Contracts	• BasicCommunityV1 • CommunityReadManagerV1 • BeaconProxy • Community • CommunityAdmin • CommunityFactory • PermissionsRegistry • Clones • SplitMain • SplitWallet • TokenManager2 • TokenManagerUpgradeable2
Type	Solidity
Platform	Polygon

1.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants, for a total of 2 person-week. The assessment was conducted over the course of 1 calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jasraj Bedi, Co-Founder
jazzy@zellic.io

Stephen Tong, Co-Founder
stephen@zellic.io

The following consultants were engaged to conduct the assessment:

Ayaz Mammadov, Engineer
ayaz@zellic.io

Vlad Toie, Engineer
vlad@zellic.io

1.5 Project Timeline

The key dates of the engagement are detailed below.

May 13, 2022	Kick-off call
May 16, 2022	Start of primary review period
May 20, 2022	End of primary review period
June 13, 2022	Closing call

1.6 Disclaimer

This assessment does not provide any warranties on finding all possible issues within its scope; i.e., the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees on any additional code added to the assessed project after our assessment has concluded. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program. Finally, this assessment report should not be considered financial or investment advice.

2 Executive Summary

Zellic conducted an audit for Highlight, Inc. from May 16 to May 20th, 2022 on the scoped contracts and discovered 2 findings. Fortunately, no critical issues were found. We applaud Highlight, Inc. for their attention to detail and diligence in maintaining incredibly high code quality standards in the development of Highlight.

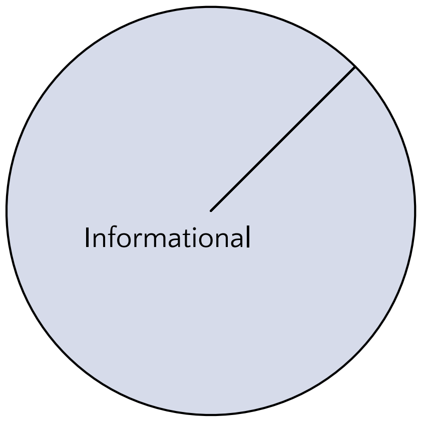
The sole finding was classified as informational. Additionally, Zellic recorded its notes and observations from the audit for Highlight, Inc.'s benefit at the end of the document.

Zellic thoroughly reviewed the Highlight codebase to find protocol-breaking bugs as defined by the documentation, or any technical issues outlined in the Methodology section of this document. Specifically, taking into account Highlight's threat model, we focused heavily on issues that would break core protocol functionalities, such as overwriting states of ownership, or maliciously interacting with the communities.

Our general overview of the code is that it was very well-organized and structured. The code coverage is high and tests are included for the majority of the functions. The documentation was adequate, although it could be improved. The code was easy to comprehend, and in most cases, intuitive.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	0
Informational	2



3 Detailed Findings

3.1 Ownership transfer may be erroneous.

- **Target**: BasicCommunityV1
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: N/A

Description

Currently, the `ownership` transfer in BasicCommunityV1 is done as:

```
function transferOwnership(address newOwner) public virtual onlyOwner
    whenNotPaused {
    require(newOwner ≠ address(0), “Invalid owner”);
    _transferOwnership(newOwner);
}

...

function _transferOwnership(address newOwner) internal virtual {
    address oldOwner = _owner;
    _owner = newOwner;
    emit OwnershipTransferred(oldOwner, newOwner);
}
```

To adhere to the highest standards of safe coding practices, it is necessary to do the ownership transfer of a contract in two steps. This prevents the current `owner` from mistakenly transferring ownership.

Impact

Although highly unlikely, a parameter error (such as a typo) could transfer the contract's ownership to an unusable address. The community would be directly affected by this.

Recommendations

We recommend implementing a two step transfer. This would allow the new `owner` to redeem the ownership via an `acceptOwnership` function.

```
function transferOwnership(address newOwner) public virtual onlyOwner
    whenNotPaused {
    require(newOwner ≠ address(0), “Invalid owner”);
    _newOwner = newOwner;
}

...

function acceptOwnership() public whenNotPaused {
    require(_newOwner == msg.sender);
    _owner = msg.sender;
    emit OwnershipTransferred(oldOwner, newOwner);
}
```

Remediation

The issue has been acknowledged by the Highlight team and a fix is planned for the future.

3.2 Lack of input validation

- **Target**: CommunityAdmin
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Informational
- **Impact**: Low

Description

Certain functions responsible for access control do not implement adequate sanitization, such as zero checks on addresses.

```
function swapDefaultAdmin(address newDefaultAdmin) external virtual
    override onlyRole(DEFAULT_ADMIN_ROLE) {
    _revokeRole(DEFAULT_ADMIN_ROLE, _msgSender());
    _grantRole(DEFAULT_ADMIN_ROLE, newDefaultAdmin);
}

/**
* @dev See {ICommunityAdmin-swapPlatform}
*/
function swapPlatform(address account) external virtual override
    onlyRole(PLATFORM_ROLE) {
    AccessControlUpgradeable._revokeRole(PLATFORM_ROLE, _msgSender());
    AccessControlUpgradeable._grantRole(PLATFORM_ROLE, account);
}
```

In the functions above the parameters `account` and `newDefaultAdmin` are not validated against the value `0x0`.

Impact

In case of invalid/empty parameter encoding, it could lead to a loss of control if roles are swapped to the 0x0 address.

Recommendations

We recommend implementing zero checks, such as the ones shown below to ensure no such incidents can occur.

```
function swapDefaultAdmin(address newDefaultAdmin) external virtual
    override onlyRole(DEFAULT_ADMIN_ROLE) {
    require(newDefaultAdmin ≠ address(0), “invalid admin”);
    _revokeRole(DEFAULT_ADMIN_ROLE, _msgSender());
    _grantRole(DEFAULT_ADMIN_ROLE, newDefaultAdmin);
}

/**
* @dev See {ICommunityAdmin-swapPlatform}
*/
function swapPlatform(address account) external virtual override
    onlyRole(PLATFORM_ROLE) {
    require(account ≠ address(0), “invalid account”);
    AccessControlUpgradeable._revokeRole(PLATFORM_ROLE, _msgSender());
    AccessControlUpgradeable._grantRole(PLATFORM_ROLE, account);
}
```

Remediation

The issue has been acknowledged by the Highlight team and a fix is planned for the future.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

We applaud Highlight for their well kept, organized code base. All functions include in-depth documentation explaining their parameters and return values. This happens even in more confusing areas such as the `0xSplits` changes, where functions have detailed explanations when more complex logic is encountered. Every interface is up to date and accurately representing the underlying contracts. The code base is structured in a composable manner with functionality being split into individual contracts representing contained features. The test coverage is also commendable, reaching 100% in multiple files.

4.1 MetaTXs

Highlight allows users to pay for their transactions with MetaTXs which can pose a problem if the forwarder has the same EOA of a priviledged account. This may happen where a contract may use `msg.sender` instead of `_msgSender`. After thorough discussions with the highlight team, we agreed that no such problems exist in the current codebase.

4.2 0xSplits changes

The several changes Highlight made to the `0xSplits` contracts added functionality to separate controllers into two different hierarchies, `Primary Controllers` and `Secondary Controllers`. We noticed that if the arrays for Primary Controllers/Secondary Controllers or Secondary Accounts get too large, the transactions' gas limit will be hit. However, this is very unlikely.

We also noticed that anyone can call functions that withdraw/route capital. These functions process the payouts for the creators with a small cut provided to the caller, effectively incentivizing the continuous flow of money. This is, of course, intended behavior, as discussed with the Highlight team.

4.3 OpenZeppelin changes

There are slight modifications added to several OpenZeppelin contracts, specifically those related to `Context`. Highlight did this to save gas, as well as to allow for the

MetaTX feature.

4.4 Gas optimizations

- Switch the function parameters from memory to calldata function wherever possible.
- In `SplitMain.sol` create a modifier for the following piece of code. As this bit is used more than once, it is possible to refactor it into a modifier or a function, for sake of simplicity.

```
if (msgSender != _splits[split].primaryController) {
    revert Unauthorized(msgSender);
}
```

- Cache length of arrays when accessed multiple times. This typically occurs in functions that have some type of `for` loop. It is essential to cache the length of arrays, such that the operation is only performed once and not on every loop iteration.
- In the `_registerTokenManager()` function found in `Community.sol`, remove the `contains()` require, since the `add()` function does the check by itself. Same applies for `unregisterTokenManager()` in the `BasicCommunityV1.sol`.

```
function _registerTokenManager(address tokenManager, address sender)
    internal {
    require(tokenManager ≠ address(this), "Invalid address");
    require(tokenManager.isContract(), "Not contract");
    require(tokenManager.supportsInterface(type(ITokenManager2).
    interfaceId), "Not token
    manager");

    // @audit-info not needed since the add() below does that check by
    itself.
    require(!_tokenManagers.contains(tokenManager), "Already
    registered");

    _tokenManagers.add(tokenManager);
```