Zellic

Valts

Smart Contract Security Assessment

September 23, 2022

Prepared for:

Pat Yiu and Justin Moskowitz

Valts

Prepared by:

Vlad Toie and Aaron Jobé

Zellic Inc.

Contents

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 Executive Summary

Zellic conducted an audit for Valts from September 14th to September 16th, 2022.

Our general overview of the code is that it was 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.

We applaud Valts for their attention to detail and diligence in maintaining incredibly high code quality standards in the development of Valts.

Zellic thoroughly reviewed the Valts codebase to find protocol-breaking bugs as defined by the documentation and to find any technical issues outlined in the Methodology section (2.2) of this document.

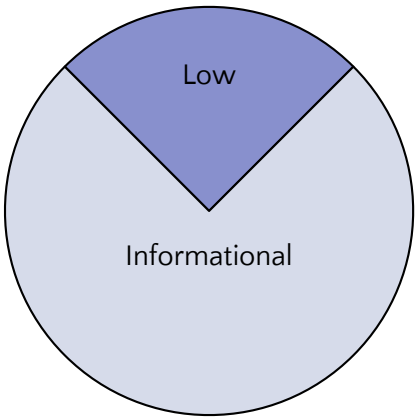
Specifically, taking into account Valts's threat model, we focused heavily on issues that would break core invariants such as the accounting of tokens.

During our assessment on the scoped Valts contracts, we discovered four findings. Fortunately, no critical issues were found. Of the four findings, one was of low severity, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the audit for Valts's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	1
Informational	3



2 Introduction

2.1 About Valts

Valts helps fans of the world's largest Twitch/YouTube livestream creators build meaningful connections with one another as they buy, sell, and engage to earn positive sum rewards. As fans connect more regularly, they develop stronger connections between one another and build an increasingly more vibrant community.

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 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, and so forth 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, and so forth.

Complex integration risks. Several high-profile 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. 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, and so forth.

Code maturity. We review for possible 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, upgradeability weaknesses, centralization risks, and so forth.

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 being 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 numerous soft factors, such as our clients' threat models, their business needs, their project timelines, and so forth. We aim to provide useful and actionable advice to our partners that consider their long-term goals rather than simply provide a list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

Valts Contracts

Repositories	• a. https://github.com/ValtsCo/valts-erc20 • b. https://github.com/ValtsCo/valts-erc1155
Versions	• a. `7c2b57239f2d41a869c0e3aa6dbadcf5689e146b` • b. `867c734077f0e546b40f46f88f17599f4002484f`
Programs	• a. valts • b. Valts1155
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of two person-days. The assessment was conducted over the course of two days.

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:

Vlad Toie, Engineer
vlad@zellic.io

Aaron Jobe, Engineer
aaronj@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

September 14, 2022	Start of primary review period
September 16, 2022	End of primary review period

3 Detailed Findings

3.1 Unnecessary use of the `receive()` function

- **Target**: valts.sol
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

The `receive()` function is typically used when the contract is supposed to receive ETH. In this case, the contract is expected **not** to receive any ETH, and for that reason, a revert is put in place so that it does not happen.

```
receive () external payable {
    revert();
}
```

Recommendations

We recommend removing the `receive()` function altogether, such that no ETH can be manually transferred by an EOA to the contract.

Remediation

The issue has been fixed in commit 2124d1a.

3.2 Inconsistent usage of modifiers

- **Target**: valts.sol, Valts1155.sol
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: N/A

Description

In `wipeApproval`, the `onlyRole(OWNER_ROLE)` can be enforced over the current `require` statement.

```
function wipeApproval(ApprovalType approvalType, address to,
    uint256 amount) external {
    require(hasRole(OWNER_ROLE, msg.sender), "Owner required");
    cleanupApproval(makeKey(approvalType, to, amount));
}
```

Recommendations

We recommend using the `onlyRole` modifier.

```
function wipeApproval(ApprovalType approvalType, address to,
    uint256 amount) external onlyRole(OWNER_ROLE) {
    cleanupApproval(makeKey(approvalType, to, amount));
}
```

Remediation

The issue has been fixed in commits 8bbb42b and 2124d1a.

3.3 Role checks are redundant

- **Target**: Valts1155.sol
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: N/A

Description

In `remminter` and `addminter`, there is a check on whether the account whose role is about to be changed actually has that role or not. The check does not need to be performed at the function level, since `_revokeRole` and `_grantRole` perform the necessary checks and reverts in the case those fail.

```
function remminter(address account) external onlyRole(OWNER_ROLE) {
    require(hasRole(MINTER_ROLE, account), “Not a minter”);
    _revokeRole(MINTER_ROLE, account);
}

function addminter(address account) external onlyRole(OWNER_ROLE) {
    require(!hasRole(MINTER_ROLE, account), “Already a minter”);
    _grantRole(MINTER_ROLE, account);
}
```

Recommendations

We recommend removing the `require` statements.

```
function remminter(address account) external onlyRole(OWNER_ROLE) {
    _revokeRole(MINTER_ROLE, account);
}

function addminter(address account) external onlyRole(OWNER_ROLE) {
    _grantRole(MINTER_ROLE, account);
}
```

Remediation

The issue has been fixed in commit 8bbb42b.

3.4 Unnecessary function statements

- **Target**: valts.sol, Valts1155.sol
- **Category**: Coding Mistakes
- **Severity**: Informational
- **Likelihood**: N/A
- **Impact**: N/A

Description

In the `cleanupApproval()` function, a `delete` statement is executed on the `_approvals`.

```
function cleanupApproval(bytes32 hashKey) private {

    for(uint8 i=0; i<_oracles.length; i++) {
        _approvals[hashKey].approvers[_oracles[i]] = false;
    }

    _approvals[hashKey].count = 0;
    _approvals[hashKey].hasowner = false;
    delete _approvals[hashKey];
}
```

There is no need to reset the variables manually, since the `delete` statement handles that on its own. Moreover, removing the unnecessary statements would lead to less gas being consumed on the function call.

Recommendations

We recommend removing the statements, adjusting the functions as so:

```
function cleanupApproval(bytes32 hashKey) private {

    for(uint8 i=0; i<_oracles.length; i++) {
        _approvals[hashKey].approvers[_oracles[i]] = false;
    }

    delete _approvals[hashKey];
}
```

Remediation

The issue has been fixed in commits 8bbb42b and 2124d1a.

4 Discussion

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

4.1 Formatting of code is inconsistent

In both valts.sol and Valts1155.sol, there are inconsistencies with naming, indentation, and line separation.

For example, there is a function in Valts1155.sol called `hasConsensus` and one called `addoracle`. The latter function could be called `addOracle` to match the style of the former. There are multiple variables, arguments, and events in both contracts that use different naming conventions.

Furthermore, the line spacing is inconsistent in both contracts. Auditors have observed one, two, three, four, five, and zero spaces of indentation throughout both files.

Finally, there seems to not be agreement on when a line separator should be used. One example is the `supportsInterface` and `mintBatch` functions in Valts1155.sol (both function declarations span multiple lines). The `supportsInterface` function has its first modifer on a different line while `mintBatch` has it on the first line of the function declaration. Additionally, the opening curly brace of `mintBatch` is on the same line as the function's return value while the opening curly brace of `supportsInterface` has its own line.

These issues do not affect the security of the code. However, they do hamper readability.

4.2 Adding oracles decreases the percentage of oracles required to reach consensus

In both tokens the `hasConsensus` function requires that two or three oracles approve of an action. These numbers are hardcoded and it should be noted that there is no limit on the number of oracles that can be added. Initially, 50% or 75% of the oracles need to approve of an action. However, as oracles are added, the percentage of oracles required for consensus goes down. This is not necessarily being treated as a centralization risk because adding an oracle does require consensus among the teller (in valts.sol) and three oracles.

5 Audit Results

At the time of our audit, the code was not deployed to mainnet evm.

During our audit, we discovered four findings. Of these, one was low risk and three were suggestions (informational).

5.1 Disclaimers

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 additional code added to the assessed project after the audit version of 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 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.

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.