Inflame

Technical Opinion

Regarding the Smart Contract developed

by Borderless's team

1. Introduction

Given the chance to examine the Borderless smart contract source code, we outline in this report our methodical approach to assess potential security issues in the smart contract implementation, draw attention to any potential semantic discrepancies between the smart contract code, and offer additional suggestions or recommendations for improvement.

1.1 The Process

Auditing approach and Methodologies applied.

In this audit, I consider the following crucial features of the code.

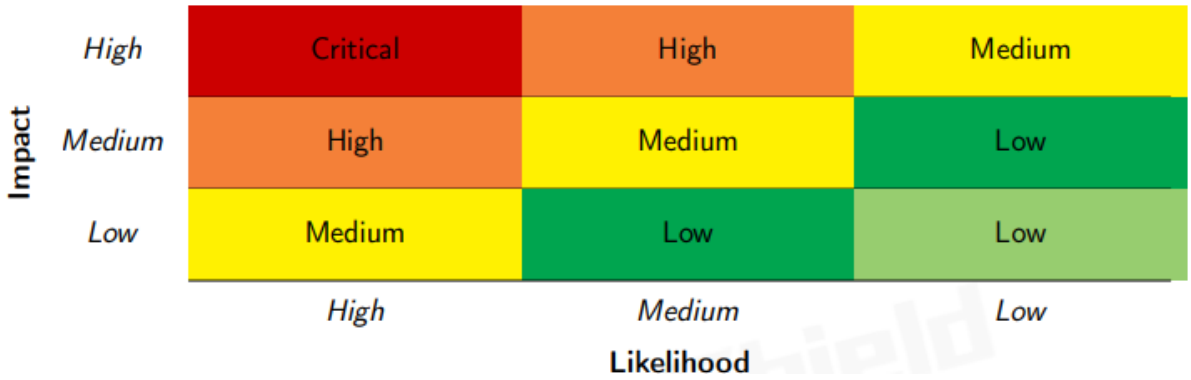
- Whether the implementation of protocol standards.
- Whether the code is secure.
- Whether the code meets the best coding practices.

The basic information of Borderless is as follows:

Item	Description
Issuer	Borderless
Website	https://borderless.money

Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Automated Security Analysis

The measuring system used is as follows:



1.2 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology:

• Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;

• Impact measures the technical loss and business damage of a successful attack;

• Severity demonstrates the overall criticality of the risk. Likelihood and impact are categorized into three ratings: H, M and L, i.e., high, medium and low respectively.

The audit has been performed according to the procedure manual and automated. First, manually analyzing the code for security vulnerabilities, assessing the overall project structure, complexity and quality, checking SWC Registry issues in the code, and checking whether all

the libraries used in the code of the latest version. After that, scanning the project's code with security tools.

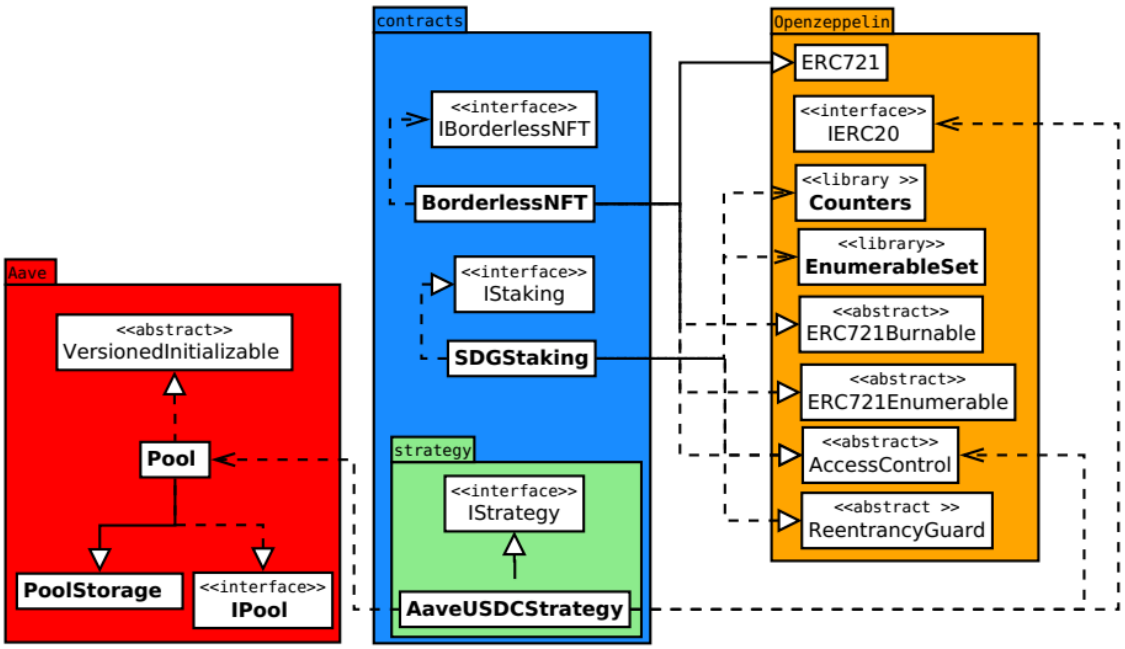
1.3 Disclaimer

Be aware that the results of this audit do not guarantee that all potential security flaws in the specified smart contract(s) have been discovered; in other words, the absence of any additional security flaws is not guaranteed. We always advise conducting multiple independent audits and a public bug bounty program because one audit-based assessment cannot be thought of as comprehensive when it comes to ensuring the security of smart contracts (s). Last but not least, you shouldn't consider the security audit as investment advice.

1.4 Our Analysis

Altogether the code is written and demonstrates effective use of abstraction, separation of concern, and modularity. Our findings demonstrate that the existing smart contract can be further improved. This document summarizes the findings of our audit.

Contract dependency diagram



contracts/SDGStaking.sol

1. Issue: A floating pragma is set.

Severity: Low

Type: SWC-103

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Relationships

CWE-664: Improper Control of a Resource Through its Lifetime

Description

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

Remediation

Lock the pragma version and also consider known bugs for the compiler version that is chosen.

Pragma statements can be allowed to float when a contract is intended for consumption by other developers, as in the case with contracts in a library or EthPM package. Otherwise, the developer would need to manually update the pragma in order to compile locally.

References

- Ethereum Smart Contract Best Practices - Lock pragmas to specific compiler version

2. Issue: Requirement violation
(L: 89 C: 6473; L: 89 C: 6473; L: 89 C: 6881; L: 89 C: 6881)
Severity: Low
Type: SWC-123

A requirement was violated in a nested call and the call was reverted as a result. Make sure valid inputs are provided to the nested call (for instance, via passed arguments).

Relationships

CWE-573: Improper Following of Specification by Caller

Description

The Solidity require() construct is meant to validate external inputs of a function. In most cases, such external inputs are provided by callers, but they may also be returned by callees. In the former case, we refer to them as precondition violations. Violations of a requirement can indicate one of two possible issues:

1. A bug exists in the contract that provided the external input.
2. The condition used to express the requirement is too strong.

Remediation

If the required logical condition is too strong, it should be weakened to allow all valid external inputs.

Otherwise, the bug must be in the contract that provided the external input and one should consider fixing its code by making sure no invalid inputs are provided.

References

- The use of revert(), assert(), and require() in Solidity, and the new REVERT opcode in the EVM

Contract Samples

requirement_simple.yaml

```
description: Simple requirement violation
issues:
- id: SWC-123
  count: 1
  locations:
  - bytecode_offsets:
      '0xba541cbb2ac6dda7664b6ebdc6372297425c2eeabd88e0af5ca6310f9cf7bbd2':
      - 263
      '0x2dbd524f734f1b06b19cbefbbb5e84bbc6203a04bc930c26141c40bb8ecf467b':
      - 145
    line_numbers:
      requirement_simple.sol: [6, 12]

```

requirement_simple_fixed.yaml

```
description: Simple requirement violation
issues:
- id: SWC-123
  count: 0
  locations: []

```

3. Issue: Use of block timestamp in application logic
(L:65 C:24)
Severity: Low
Type: SWC-116

```
63          _stakeIdToStakeInfo[stakeId] = StakeInfo({
64              amount: amount,
65              createdAt: block.timestamp,
66              stakePeriod: 10,
67              status: StakeStatus.UNDELEGATED,
68              strategies: new address[](0),
69              shares: new uint256[](0),
70              epoch: _epoch + 1
71          });
```

Relationships

CWE-829: Inclusion of Functionality from Untrusted Control Sphere

Description

Contracts often need access to time values to perform certain types of functionality. Values such as block.timestamp, and block.number can give you a sense of the current time or a time delta, however, they are not safe to use for most purposes.

In the case of block.timestamp, developers often attempt to use it to trigger time-dependent events. As Ethereum is decentralized, nodes can synchronize time only to some degree. Moreover, malicious miners can alter the timestamp of their blocks, especially if they can gain advantages by doing so. However, miners can't set a timestamp smaller than the previous one (otherwise the block will be rejected), nor can they set the timestamp too far ahead in the future. Taking all of the above into consideration, developers can't rely on the preciseness of the provided timestamp.

As for block.number, considering the block time on Ethereum is generally about 14 seconds, it's possible to predict the time delta between blocks. However, block times are not constant and are subject to change for a variety of reasons, e.g. fork reorganisations and the difficulty

bomb. Due to variable block times, block.number should also not be relied on for precise calculations of time.

Remediation

- Developers should write smart contracts with the notion that block values are not precise, and the use of them can lead to unexpected effects. Alternatively, they may make use of oracles.

References

- Safety: Timestamp dependence
- Ethereum Smart Contract Best Practices - Timestamp Dependence
- How do Ethereum mining nodes maintain a time consistent with the network?
- Solidity: Timestamp dependency, is it possible to do safely?
- Avoid using block.number as a timestamp

4. Issue: Explicitly mark visibility of state
(L 23,24,25,26,27,28,30,31,32,33**)**
Severity: Low
Type: SWC-100

Relationships

CWE-710: Improper Adherence to Coding Standards

Description

Functions that do not have a function visibility type specified are public by default. This can lead to a vulnerability if a developer forgot to set the visibility and a malicious user is able to make unauthorized or unintended state changes.

Remediation

Functions can be specified as being external, public, internal or private. It is recommended to make a conscious decision on which visibility type is appropriate for a function. This can dramatically reduce the attack surface of a contract system.

References

- Ethereum Smart Contract Best Practices - Explicitly mark visibility in functions and state variables
- SigmaPrime - Visibility

5. Issue: Functions Returns Type and No Return / Should return a struct
(L:221 C:4) / (L:111 C:26)
Severity: Low
Type: SWC-104

Relationships

CWE-252: Unchecked Return Value

Description

Function should return a type however nothing is returned. This can lead to unexpected behavior in the subsequent program logic If the call fails accidentally or an attacker forces the call to fail.

Remediation

Functions must return the type specified in its prototype. This is important because you can test possibilities that a function call will fail by checking the return value, in particular, If you use low-level call methods.

References

- Ethereum Smart Contract Best Practices - Handle errors in external calls

6. Issue: Gas Limit in Loops / Extra Gas in Loop
(L:184 C:8, C:8, L:322 C:8, L:490 C:8)
Severity: Low
Type: SWC-128

Relationships

CWE-400: Uncontrolled Resource Consumption

Description

State variable, .balance, or .length of non-memory array is used in the condition of for or while loop. In this case, every iteration of the loop consumes extra gas.

When smart contracts are deployed or functions inside them are called, the execution of these actions always requires a certain amount of gas, based on how much computation is needed to complete them. The Ethereum network specifies a block gas limit and the sum of all transactions included in a block can not exceed the threshold.

Programming patterns that are harmless in centralized applications can lead to Denial of Service conditions in smart contracts when the cost of executing a function exceeds the block gas limit. Modifying an array of unknown size, that increases in size over time, can lead to such a Denial of Service condition.

Remediation

Caution is advised when you expect to have large arrays that grow over time. Actions that require looping across the entire data structure should be avoided.

If you absolutely must loop over an array of unknown size, then you should plan for it to potentially take multiple blocks, and therefore require multiple transactions.

If a state variable, .balance, or .length is used several times,holding its value in a local variable is more gas efficient. If .length of calldata-array is placed into a local variable, the optimisation will be less significant.

References

- Ethereum Design Rationale
- Ethereum Yellow Paper
- Clear Large Array Without Blowing Gas Limit
- GovernMental jackpot payout DoS Gas

Issue	Description	Severity	Result
Constructor Mismatch	Whether the contract name and its constructor are not identical to each other.	Critical	Not Found
Ownership Takeover	Whether the set owner function is not protected.	Critical	Not Found
Redundant Fallback Function	Whether the contract has a redundant fallback function	Critical	Not Found
Overflows & Underflows	Whether the contract has general overflow or underflow vulnerabilities	Critical	Not Found
Reentrancy	Reentrancy is an issue when code can call back into your contract and change state, such as withdrawing Crypto.	Critical	Not Found
Unauthorized Self-Destruct	Whether the contract can be killed by any arbitrary address	Critical	Not Found

Revert DoS	Whether the contract is vulnerable to DoS attack because of unexpected revert.	Critical	Not Found
Costly Loop	Whether the contract has any costly loop which may lead to Out-Of-Gas exception.	Medium	Not Found
Use Of Untrusted Libraries	Whether the contract uses any suspicious libraries.	Medium	Not Found
Use Of Predictable Variables	Whether the contract contains any randomness variable, but its value can be predicated	Medium	Not Found
Transaction Ordering Dependence	Whether the final state of the contract depends on the order of the transactions.	Medium	Not Found

Conclusion

The smart contract presented is simple and the code is relatively small. Altogether the code is written and demonstrates effective use of abstraction, separation of concern, and modularity. But there are some issues/vulnerabilities to be tackled at low security levels, it is recommended to fix them before deploying the contract on the main network. Given the subjective nature of some assessments, it will be up to the Borderless team to decide whether any changes should be made.