Security Review Report
NM-0071 Jasmine

NETHERMIND

(May 05, 2023)

Contents

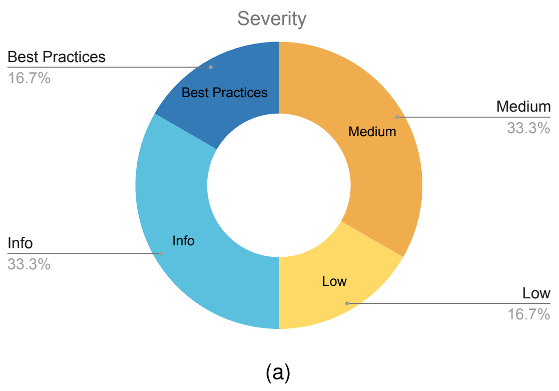
1 Executive Summary

This document presents the security review performed by Nethermind on the Jasmine Energy contracts. Jasmine Energy creates accessible and transparent climate asset markets. It allows Energy Attribute Certificates to be brought onto the Blockchain and markets to be created around them, providing benefits such as price discovery, a unified marketplace, and total price transparency.

The audited code consists of 600 lines of Solidity with 69% code coverage. Overall, the audited code is well-written and well-documented. The team provided general documentation for the protocol and its use cases. Together with the `NatSpec` provided in the code, we were able to understand the intention of the developers effectively. Multiple walkthroughs of the code and continuous communication with the team supported this documentation. **The audit was performed using** (a) manual analysis of the codebase, (b) automated analysis tools, and (c) simulation of the smart contracts.

Along this document, we report 6 points of attention, with two of them classified as `Medium`, one as `Low`, and three are `Best Practice` or `Informational` issues. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope of this audit. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology adopted for this audit. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the compilation, tests, coverage, and automated tests. Section 9 concludes the document.



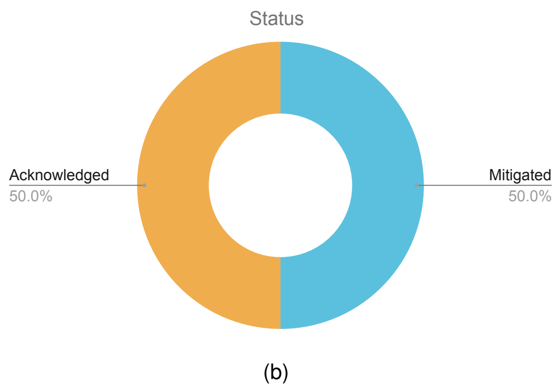


Fig. 1: Distribution of issues: Critical (0), **High** (0), **Medium** (2), **Low** (1), **Undetermined** (0), **Informational** (2), **Best Practices** (1).
Distribution of status: Fixed (0), **Acknowledged** (3), **Mitigated** (3), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Mar. 13, 2023
Response from Client	Apr. 20, 2023
Final Report	May. 5, 2023
Methods	Manual Review, Automated Analysis
Repository	Jasmine contracts
Commit Hash	5013da183570d83b6cf2480b34985c4e4a22303b
Documentation	README.md, Web docs
Documentation Assessment	High
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	scripts/Deploy.s.sol	65	10	15.4%	15	90
2	src/ERC1967UUPSProxy.sol	79	131	165.8%	9	219
3	src/JasmineMinter.sol	179	55	30.7%	24	258
4	src/JasmineEAT.sol	137	16	11.7%	21	174
5	src/JasmineOracle.sol	104	21	20.2%	18	143
6	src/interfaces/IERC1155Mintable.sol	15	1	6.7%	2	18
7	src/interfaces/IJasmineEATExtensions.sol	4	1	25.0%	1	6
8	src/interfaces/IJasmineOracle.sol	4	4	100.0%	1	9
9	src/interfaces/IERC1155Burnable.sol	13	1	7.7%	2	16
	Total	**600**	**240**	**40.0%**	**93**	**933**

3 Summary of Issues

	Finding	Severity	Update
1	Frozen tokens can trigger retirement	Medium	Mitigated
2	JasmineOracle information is not updated after burning tokens	Medium	Acknowledged
3	Series information can be updated at any time	Low	Mitigated
4	Check `initialOwner` address in the `initialize(...)` function	Info	Acknowledged
5	`setMinter(...)` input is not validated	Info	Mitigated
6	EATs can be frozen indefinitely	Best Practices	Acknowledged

4 System Overview

The Jasmine protocol consists of three upgradeable contracts: `JasmineEAT`, `JasmineMinter`, and `JasmineOracle`. Each one of them is deployed behind an `ERC1967 UUPS proxy`. For security reasons, the ownership of the contract is distinct from the minting access. In addition, each proxy contract has an owner with full control of the contract. The proxy contracts implement a 2-step process to transfer ownership, which also includes the right to set the addresses of the other entities in the protocol.

4.1 Proxy

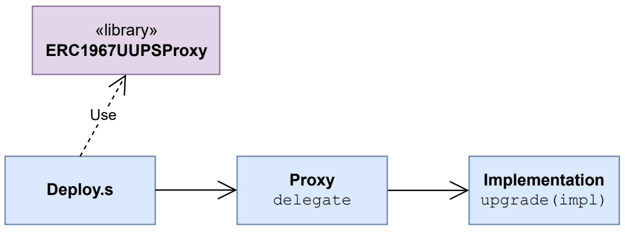


Fig. 2: Logical Flow Diagram of the Proxy

Fig.2 presents the logical flow of the proxy structure in Jasmine. The deploy script uses the create function from the `ERC1967UUPSProxy` library to deploy a `Proxy` contract pointing to an already existing implementation (`JasmineEAT`, `JasmineMinter`, or `JasmineOracle` contracts).

4.2 Implementation Contracts

Fig. 3 shows the logic contracts and their main association connectors (including generalization, aggregation, implementation, use, and uni-directional association).

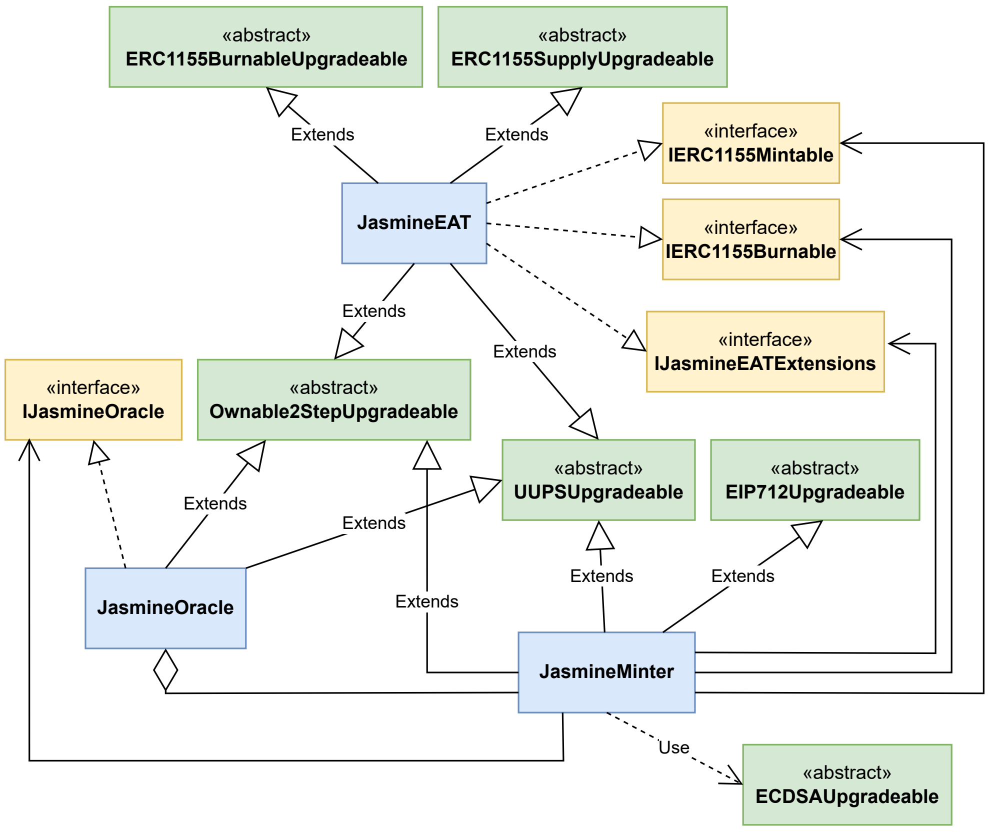


Fig. 3: Structural Diagram of the Contracts

4.2.1 `JasmineOracle`

The `JasmineOracle` extends the abstract contracts `Ownable2StepUpgradeable` and `UUPSUpgradeable`. A Jasmine Oracle is deployed behind the proxy `ERC1967UUPSProxy`. It provides readable metadata about each EAT series to other contracts with the following attributes: `UUID`, Registry, Fuel Type, Certificate type, Vintage, and Endorsement. This contract has the following public functions.

- `setMinter(...)`
- `initialize(...)`
- `updateSeries(...)`

In addition to these functions, the contract provides 8 (eight) `getters` which are not described here. For simplicity reasons, we do not list functions in the contracts `UUPSUpgradeable.sol`[1] and `Ownable2StepUpgradeable.sol`[2] since they are well-known open standard contracts.

4.2.2 `JasmineMinter`

As described in Fig. 3, `JasmineMinter` contract inherits `Ownable2StepUpgradeable`, `UUPSUpgradeable`, and `EIP712Upgradeable`. `JasmineMinter` implements all the logic for bridge-on, bridge-off, and redemption. To mint tokens on the `JasmineMinter` contract, a user must have a validly signed (Encoded `ECDSA` signature by a bridge over the `EIP712` minting authorization) and unexpired minting right. Once this requirement has been met, the user can call the appropriate minting function on the contract. The two available functions are `mint()` and `mintBatch()`.

The receiver address for the tokens is unconstrained and can be any valid address, it will receive the tokens after they have been minted. If the receiver is a contract, it must implement the `IERC1155Receiver` interface. If the contract does not implement the interface, the transaction will revert. However, the transaction can be safely retried (with the same signature) if the deadline hasn't been reached. This contract has the following public functions.

- `setBridge(...)`
- `initialize(...)`
- `mint(...)`
- `mintBatch(...)`
- `consumeNonce(...)`
- `burn(...)`
- `burnBatch(...)`

In addition to these functions, the contract provides 2 (two) `getters` which are not described here. Similarly to the previous subsection, we do not list functions in the contracts `UUPSUpgradeable`[1] and `Ownable2StepUpgradeable`[2].

4.2.3 `JasmineEAT`

As presented in Fig. 3, `JasmineEAT` implements three interfaces and inherits four abstract contracts (`UUPSUpgradeable` and `Ownable2StepUpgradeable`, `ERC1155BurnableUpgradeable`, `ERC1155SupplyUpgradeable`). `JasmineEAT` is an owner-upgradeable `ERC1155` multi-token with owner-designated centralized minting rights. Particularly, there is a minter attribute that is intended to be set to the address of the `JasmineMinter` proxy. The contract also implements the following features:

- a burn functionality which enables the redemption and bridge-off functionality;
- the freeze functionality to invalidate EAT series. Frozen EAT series still exists on chain, but they can not be minted, transferred, retired, or bridge-off. can be slashed to be destroyed on chain by the owner or the token holder can voluntarily burn them. The freeze functionality restricts only to the EAT series, not to token holders.

This contract has the following public functions.

- `setBridge(...)`
- `initialize(...)`
- `mint(...)`
- `mintBatch(...)`
- `consumeNonce(...)`
- `burn(...)`
- `burnBatch(...)`

In addition to these functions, the contract provides 2 (two) `getters` which are not described here. The list below does not include functions in contracts `UUPSUpgradeable.sol`[1] and `Ownable2StepUpgradeable.sol`[2].

[1] UUPSUpgradeable.sol
[2] Ownable2StepUpgradeable.sol

5 Risk Rating Methodology

The risk rating methodology used by Nethermind follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**. **Likelihood** is a measure of how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding other factors are also considered. These can include but are not limited to: Motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if the finding were to be exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue can cause significant damage such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to formally pass to the client;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [Medium] Frozen tokens can trigger retirement

File(s): `JasmineEAT.sol`

Description: Invalidating a minted token involves a two-step process. The first step is to freeze the token. While a frozen token still exists on-chain, it cannot be minted, transferred, redeemed, or bridged off. It is still possible for a user to burn a frozen token they own through the `burn(...)` function in the `JasmineEAT` contract. However, because this function triggers off-chain logic for redeeming/retiring the token, burning a frozen token from the `JasmineEAT` contract will redeem/retire it even if it is frozen.

Recommendation(s): Consider restricting the functionality for redeeming/retiring and bridging-off only to be accessible through the `JasmineMinter` contract. Also, limit the `burn(...)` and `burnBatch(...)` functions in the `JasmineEAT` contract only to burn frozen tokens.

Status: Mitigated

Update from the client: Jasmine Energy's backend systems are always "in the loop" for finalizing retirements that are initiated on-chain. Ultimately, both retirement and bridging-off of tokens are actions that are intended to initiate processes that interact with external, off-chain services. Initiating and executing these processes occurs entirely through Jasmine Energy's backend services directly communicating with the energy attribute registries and other institutions that act as the authoritative record keepers for energy attribute assets. As such, any on-chain retirement or bridging-off actions must, by design, pass through the Jasmine Energy backend to produce any downstream, off-chain, effects. In the backend services, the first step of either the retirement or bridging-off processes is to validate the transaction's inputs which include (among other factors) whether the token is frozen.

6.2 [Medium] JasmineOracle information is not updated after burning tokens

File(s): `JasmineMinter.sol`

Description: When a token is minted from the `JasmineMinter` contract, the `JasmineOracle.updateSeries(...)` function may be invoked to add metadata for that token. This metadata includes fuel type, certificate type, or endorsement. However, when the token is burned, the oracle does not remove the metadata related to the burned token. The oracle is the primary source of information about the `EAT` tokens, and other components of the `Jasmine` protocol or other apps built on top of it will use it to fetch the current state of the `EAT`. Outdated information fetched from the oracle could cause unexpected behavior in these other components.

Recommendation(s): Consider updating the Oracle by removing data from the `_metadata` mapping when all tokens in the series are burned.

Status: Acknowledged

Update from the client: Acknowledged, and primarily addressed through clarifying external facing documentation. The JasmineOracle contract serves to store and provide metadata about energy attribute tokens (EATs) that is used by other Jasmine contracts (and in the future third-party contracts) as inclusion criteria for deposit pools, and gate or modify other on-chain behaviors. This metadata is created on the Oracle at the time of token creation but is not removed when a token is burned (or retired). While this is atypical for similar types of oracle contracts, it is the intended behavior for this contract, primarily for auditability and traceability purposes. The process of 'retiring' an EAT serves to conclusively and irrevocably 'attribute' the quality of energy represented by the underlying energy attribute certificate to the beneficial person or organization specified by the retiring party. While other records for this attribution are produced (i.e. PDF attestations, etc.), retaining this information on-chain provides a permanently, publicly accessible record making audits and verification by external parties easier.

6.3 [Low] Series information can be updated at any time

File(s): `JasmineMinter.sol`

Description: Every time the `mint(...)` or `mintBatch(...)` functions are executed in the `JasmineMinter` contract, it is possible to update the information stored in the `JasminOracle` related to the series of the minted tokens. Whether to update this information or not is delegated to an off-chain component. However, the documentation states that this check is simple: `if the series has already had at least 1 token minted into it, omit the oracle message`. Incorrect behavior of this off-chain component could cause information in the oracle to not be synced. For example, two possible scenarios are: (a) A token for a new series is minted, but there is no `oracleData`, so this series does not exist in the oracle; (b) `oracleData` is sent for minting a token that is not the first of its series, modifying the current information in the oracle.

Recommendation(s): Consider adding a check in the `mint(...)` and `mintBatch(...)` functions to ensure that: (a) `updateSeries(...)` function is only called if the minted token is the first of its series; (b) If the minted token is the first of its series, `oracleData` must be non-empty.

Status: Mitigated

Update from the client: Mitigated, primarily through backend services restricting and validating token creation. While in theory, it is possible for the oracle contract to update the oracle data associated with a token series whenever a new token with the same 'series id' is minted, in practice this is not possible. The backend enforces that each new minting of a token from a certificate will generate a new unique token or series identifier; however, even without this restriction, the `updateSeries` call would still be idempotent since the backend would enforce the same metadata for all tokens in the same series. While not necessary, adding additional checks into the smart contract will be considered in future upgrades.

6.4 [Info] Check `initialOwner` address in the `initialize(...)` function

File(s): `JasmineEAT.sol`, `JasmineMinter.sol`, `JasmineOracle.sol`

Description: The `initialize(...)` function in the `JasmineEAT`, `JasmineMinter`, and `JasmintOracle` contracts set the initial state for these contracts, including the `owner` variable. The function signatures are as follows:

```
JasmineEAT.initialize(string memory initialURI, address initialMinter, address initialOwner)
JasmineMinter.initialize(string memory initialName, string memory initialVersion, address initialBridge, address
→ initialOwner)
JasmineOracle.initialize(address initialMinter, address initialOwner)
```

The `owner` role is essential for these contracts, as certain functions can only be called by the `owner`, such as `JasmineMinter.setBridge(...)`. If the `initialize(...)` function incorrectly sets the `owner` variable, the contracts must be redeployed.

Recommendation(s): Consider verifying that `initialOwner` is a valid address.

Status: Acknowledged

Update from the client: Acknowledged, but with no consequences to live deployments. This is a best practice for avoiding potential redeployments but has no effects whatsoever on deployed contracts.

6.5 [Info] `setMinter(...)` input is not validated

File(s): `JasmineEAT.sol`

Description: The `setMinter(...)` function sets the account that is able to call the `mint(...)` and `mintBatch(...)` functions. The code snippet below shows the function.

```
function setMinter(address newMinter) external onlyOwner {
    minter = newMinter;
    emit MinterChanged(newMinter);
}
```

The documentation states that, for security reasons, the minter and owner roles should be held by different accounts. However, as can be seen from the code, there is no restriction on setting a new minter.

Recommendation(s): Consider validating that the new minter account is not an invalid account, such as the current owner or the address zero.

Status: Mitigated

Update from the client: Mitigated at the process and organizational level. For security reasons, the `minter` account and the `owner` account are held separately; this shrinks the surface of vulnerabilities and more clearly delegates responsibility and permissions across accounts. This is enforced through organizational processes at Jasmine rather than at the smart contract level.

6.6 [Best Practice] EATs can be frozen indefinitely

File(s): `JasmineEAT.sol`

Description: There may be situations where an EAT series needs to be frozen. When an EAT series is frozen, the `JasmineEAT` contract owner can execute the following actions: i) `slash(. . . )` and `slashBatch(...)` to burn the EATs from this series or ii) `thaw(. . . )` to unfreeze them. If not all the tokens are slashed, and `thaw(...)` is never called, the remaining tokens could remain frozen indefinitely.

Recommendation(s): To prevent indefinite freezing of EATs, consider implementing a deadline feature that allows users to unfreeze their EATs themselves if the owner has not destroyed or unfrozen them within the deadline.

Status: Acknowledged

Update from the client: All EATs may be indefinitely frozen by the original JasmineMinter account; this feature is implemented because the originating registries of the underlying energy attribute certificates may invalidate certificates at any time for a number of reasons (e.g. faulty metering, operator error, or inaccurate record keeping). This invalidation is permanent and irreversible, and thus the freezing of tokens should not allow token owners to unfreeze after some period of time.

7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation plays a critical role in software development, as it enables effective communication between developers, testers, users, and other stakeholders involved in the software development process. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document that describes the design and technical details of the smart contract. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

To assist the audit process, the **Jasmine** team provided **Unit Tests** and two documents: **Jasmine Energy** and **Jasmine contracts**. Below is a summary of these two documents.

(a) Jasmine Energy - specifies the problem, the solution, and the product. In addition, the documentation provides several concepts and details the process of how Jasmine Energy works giving us a view of the big picture.

- `Glossary:` a list of terms with their respective descriptions some of which are used in the contracts;
- `Bridge:` a section presenting how the bridge works. Basically, it explains how the protocol turns Energy Attribute Certificates (EACs) into Energy Attribute Tokens (EATs) and retires an EAT to attribute holder energy consumption. Also, this section presents the flow of bridge-on and bridge-off;
- `Reference pools:` creates reference prices for energy attributes, how to deposit EATs to mint JLTs (Jasmine Liquid Tokens), and burn JLTs to withdraw EATs.

(b) Jasmine contracts: embraces detailed specifications for the contracts under auditing. For each contract, this documentation contains clarifications about the design decisions adopted, the application flow, and how features should behave for different scenarios.

These documents provided by the **Jasmine** team covered the terms used in the source code, explanations for architecture decisions, and core business logic and functions flow. By reading the whole documentation suite, we could get a proper comprehension of how the contracts should operate. The provided documentation is well-written and structured. It is a complete source of resources for developers and auditors. The codebase has sufficient inline comments and meaningful naming for functions and variables, helping the audit team to understand function flow and to detect issues.

7.1 The Long Parameter List smell

This code smell means a function that requires more than four parameters to be passed in. Often, this leads to increased complexity, and the **Single Responsibility Principle Violation** (`https://dl.acm.org/doi/book/10.5555/3175742`). In Jasmine's context, we notice that when we ran the command `forge coverage`, the output was a compiler error "`Stack too deep`", as described below. It is worth emphasizing that this is not an issue but an improvement for obtaining a code as documentation (`https://martinfowler.com/bliki/CodeAsDocumentation.html`) and evaluating the possibility of applying the **Single Responsibility Principle** in the code.

```
forge coverage
[] Compiling...
[] Compiling 51 files with 0.8.17
[] Solc 0.8.17 finished in 3.17s
Error:
Compiler run failed
CompilerError: Stack too deep. Try compiling with `--via-ir` (cli) or the equivalent `viaIR: true` (standard JSON)
↪ while enabling the optimizer. Otherwise, try removing local variables.
   --> test/JasmineMinter.t.sol:269:7:
    |
269 |       nonce,
    |       ^^^^^
```

We recommend applying refactoring to tackle the Long Parameter List bad smell detected in the function `mintBatch(...)` presented below. As we can see, the function contains 8 (eight) parameters. When we can obtain the required data for a parameter by requesting it from a single source, applying refactoring is the most suitable technique. **Recommendation:** Apply the **Preserve Whole Object technique** (https://martinfowler.com/books/refactoring.html), useful to replace a whole lot of data by a `struct`, for instance.

```
 function mintBatch(
    address receiver,
    uint256[] memory ids,
    uint256[] memory amounts,
    bytes memory transferData,
    bytes[] memory oracleDatas,
    uint256 deadline,
    bytes32 nonce,
    bytes memory sig
  ) external {
  ...
  }
```

8 Test Suite Evaluation

8.1 Contracts Compilation Output

```
$ forge compile
[] Compiling...
[] Compiling 51 files with 0.8.17
[] Solc 0.8.17 finished in 48.04s
Compiler run successful
```

8.2 Tests Output

```
$ forge test
[] Compiling...
No files changed, compilation skipped

Running 3 tests for test/JasmineOracle.t.sol:JasmineOracleTest
[PASS] testSetMinter() (gas: 27812)
[PASS] testUpdateSeries() (gas: 48550)
[PASS] testUpgrade() (gas: 1521212)
Test result: ok. 3 passed; 0 failed; finished in 114.85ms

Running 5 tests for test/JasmineEAT.t.sol:JasmineEATTest
[PASS] testFreeze() (gas: 114092)
[PASS] testMint() (gas: 193828)
[PASS] testSetMinter() (gas: 28257)
[PASS] testSlash() (gas: 119092)
[PASS] testUpgrade() (gas: 3505344)
Test result: ok. 5 passed; 0 failed; finished in 119.39ms

Running 8 tests for test/JasmineMinter.t.sol:JasmineMinterTest
[PASS] testBurn() (gas: 157121)
[PASS] testBurnBatch() (gas: 202653)
[PASS] testConsumeNonce() (gas: 69425)
[PASS] testDomainSeparator() (gas: 27229)
[PASS] testMint() (gas: 236753)
[PASS] testMintBatch() (gas: 360966)
[PASS] testSetBridge() (gas: 27415)
[PASS] testUpgrade() (gas: 2691114)
Test result: ok. 8 passed; 0 failed; finished in 119.31ms
```

8.3 Code Coverage

```
forge coverage
```

The relevant output is presented below.

```
| File                                                         | % Lines          | % Statements     | % Branches       | % Funcs          |
|--------------------------------------------------------------|------------------|------------------|------------------|------------------|
| src/ERC1967UUPSProxy.sol                                     | 0.00% (0/13)     | 0.00% (0/15)     | 0.00% (0/8)      | 0.00% (0/7)      |
| src/JasmineEAT.sol                                           | 59.38% (19/32)   | 57.89% (22/38)   | 64.29% (9/14)    | 84.62% (11/13)   |
| src/JasmineMinter.sol                                        | 85.11% (40/47)   | 87.72% (50/57)   | 88.46% (23/26)   | 90.00% (9/10)    |
| src/JasmineOracle.sol                                        | 85.19% (23/27)   | 87.10% (27/31)   | 50.00% (6/12)    | 91.67% (11/12)   |
| Total                                                        | 68.90% (82/119)  | 70.21% (99/141)  | 27.98% (38/60)   | 73.80% (31/42)   |
```

8.4 Slither

All the relevant issues raised by Slither have been incorporated into the issues described in this report.

8.5 Lack of testing for revert paths

The current testing suite does not include tests for all paths that should trigger a revert. It is important to check if the code reverts in predicted situations.

8.6 Upgradability testing

The tests do not include upgrading contracts with new data, such as version. The current testing suite only tests for new implementation with old data.

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

Disclaimer