hashlock.

2024

Security Audit

5ire (Layer 1)

July 2024 info@hashlock.com.au Hashlock Pty Ltd

Table of Contents

Executive Summary	4
Project Context	4
Audit scope	7
Security Rating	8
Intended Application Behaviours	9
Code Quality	10
Audit Resources	10
Dependencies	10
Severity Definitions	11
Audit Findings	12
Conclusion	17
Our Methodology	18
Disclaimers	20
About Hashlock	21

CAUTION

THIS DOCUMENT IS A SECURITY AUDIT REPORT AND MAY CONTAIN CONFIDENTIAL INFORMATION. THIS INCLUDES IDENTIFIED VULNERABILITIES AND MALICIOUS CODE THAT COULD BE USED TO COMPROMISE THE PROJECT. THIS DOCUMENT SHOULD ONLY BE FOR INTERNAL USE UNTIL ISSUES ARE RESOLVED. ONCE VULNERABILITIES ARE REMEDIATED, THIS REPORT CAN BE MADE PUBLIC. THE CONTENT OF THIS REPORT IS OWNED BY HASHLOCK PTY LTD FOR THE USE OF THE CLIENT.

Executive Summary

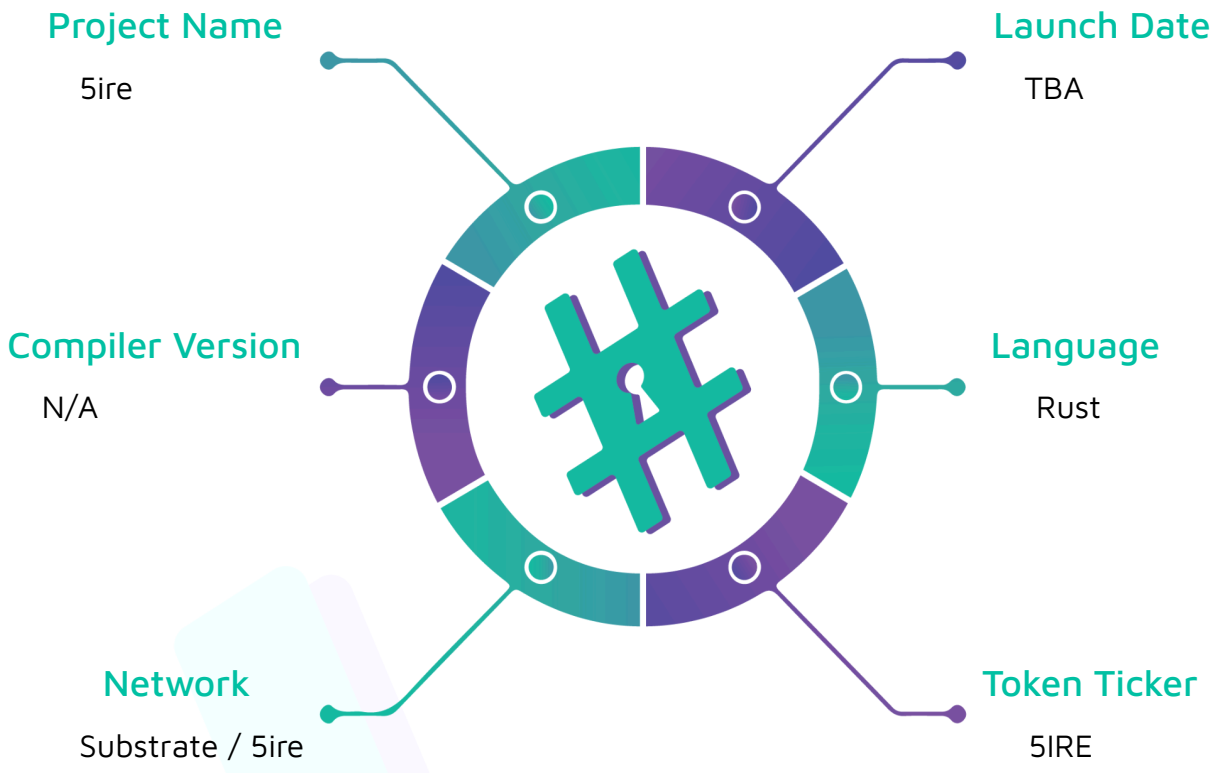
The 5ire team partnered with Hashlock to conduct a security audit of their Substrate-based blockchain. Hashlock manually and proactively reviewed the code to ensure the project's team and community that the deployed contracts were secure.

Project Context

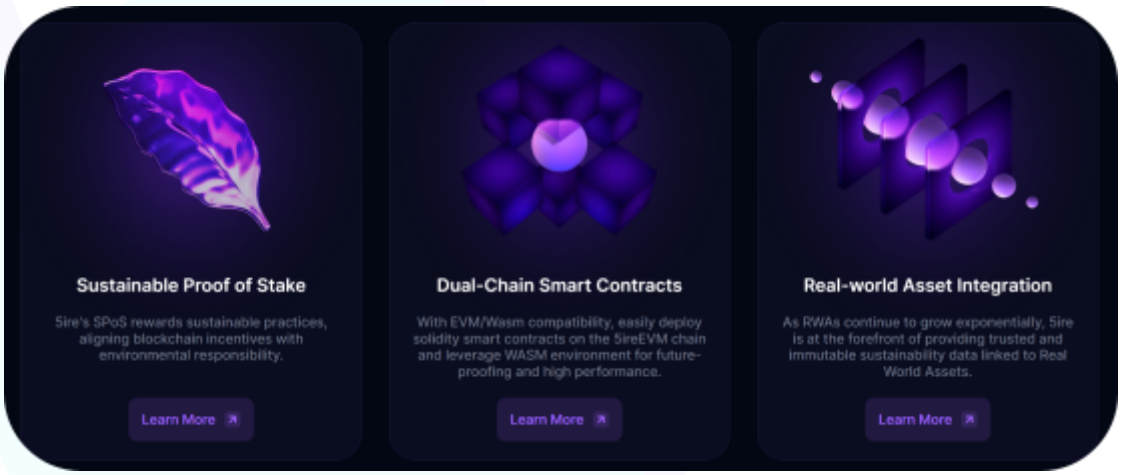
The 5ire project aims to create a blockchain ecosystem integrating sustainability with advanced technology to drive positive social impact. By emphasizing environmentally friendly practices and innovative solutions, 5ire seeks to address global challenges through decentralized applications and a transparent governance model, ultimately working towards a more sustainable and impactful future.

Project Name: 5ire
Compiler Version: ^0.8.42
Website: https://www.5ire.org/
Logo:

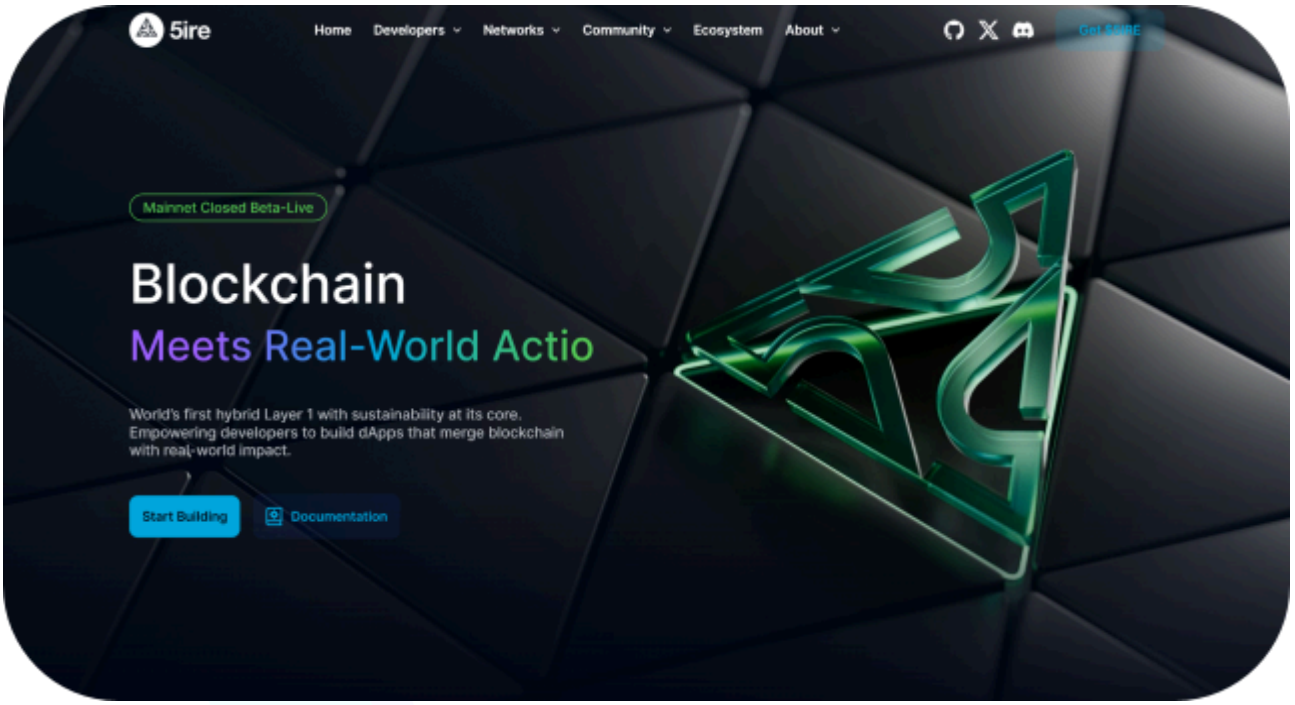
Visualised Context:

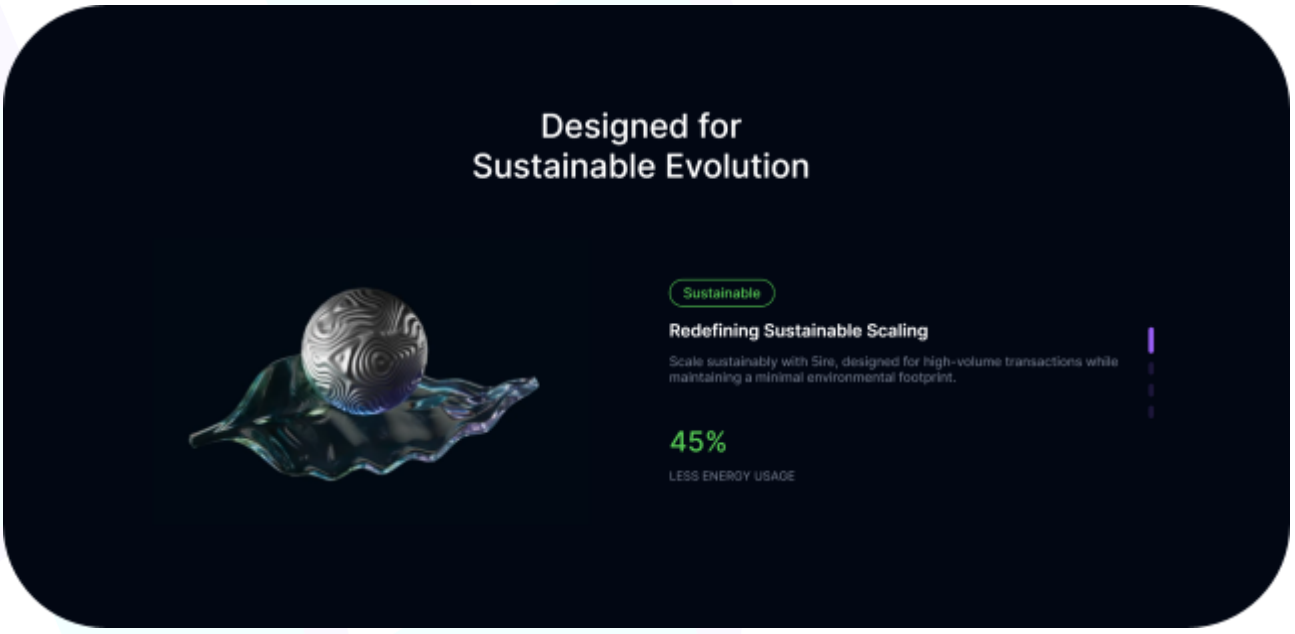


Iconography:



Project Visuals:





Audit scope

We at Hashlock audited the solidity code within the 5ire project, the scope of work included a comprehensive review of the smart contracts listed below. We tested the smart contracts to check for their security and efficiency. These tests were undertaken primarily through manual line-by-line analysis and were supported by software-assisted testing.

Description	**5ire Substrate Runtime**
Platform	**Substrate / Rust**
Audit Date	**July, 2024**
Scope 1	Unified Addresses
Commit Hash 1	c36c2b1c43b9c62b9f96694740038e5307ec1635
Scope 2	Revenue Sharing
Commit Hash 2	07bdedb215f1bf7782254ec6c94d62b8c5300c9f
Scope 3	Fixed Supply
Commit Hash 3	4f675214f9be33b50051625e779808ef8c523c0d#diff-cbe1855fd3bef9c6878a793c91771dd0013079a705b0ab1445273ea70dc3b25f
Scope 4	ESG Pallet
Commit Hash 4	ca08eb30bd6d00787298b6d8eba26d4b46118904

Security Rating

After Hashlock's Audit, we found the smart contracts to be **"Secure"**. The contracts all follow simple logic, with correct and detailed ordering. They use a series of interfaces, and the protocol uses a list of Open Zeppelin contracts.

The 'Hashlocked' rating is reserved for projects that ensure ongoing security via bug bounty programs or on-chain monitoring technology.

All issues uncovered during automated and manual analysis were meticulously reviewed and applicable vulnerabilities are presented in the Audit Findings section.

We initially identified some significant vulnerabilities that have since been addressed.

Hashlock found:

1 High-severity vulnerabilities

2 Medium-severity vulnerabilities

0 Low-severity vulnerabilities

2 Gas Optimisations

Caution: *Hashlock's audits do not guarantee a project's success or ethics, and are not liable or responsible for security. Always conduct independent research about any project before interacting.*

Intended Application Behaviours

Claimed Behaviour	Actual Behaviour
Unified Address Update - Unifies the address format	**Update achieves this functionality.**
Revenue Sharing Update - 50% of transaction fees are awarded to contract deployers for WASM and EVM contracts	**Update achieves this functionality.**
Fixed Supply Update - Caps the total supply of 5IRE by removing inflation	**Update achieves this functionality.**
ESG Pallet - Allows oracles to assigning scores	**Pallet achieves this functionality.**

Code Quality

This Audit scope involves the smart contracts of the 5ire project, as outlined in the Audit Scope section. All contracts, libraries, and interfaces mostly follow standard best practices to help avoid unnecessary complexity that increases the likelihood of exploitation, however, some refactoring was required.

The code is very well commented on and closely follows best practice nat-spec styling. All comments are correctly aligned with code functionality.

Audit Resources

We were given the 5ire project smart contract code in the form of GitHub access.

As mentioned above, code parts are well-commented. The logic is straightforward, and therefore it is easy to quickly comprehend the programming flow as well as the complex code logic. The comments help us understand the overall architecture of the protocol.

Dependencies

Per our observation, the libraries used in this smart contracts infrastructure are based on well-known industry-standard open-source projects.
Apart from libraries, its functions are used in external smart contract calls.

Severity Definitions

Significance	Description
High	High-severity vulnerabilities can result in loss of funds, asset loss, access denial, and other critical issues that will result in the direct loss of funds and control by the owners and community.
Medium	Medium-level difficulties should be solved before deployment, but won't result in loss of funds.
Low	Low-level vulnerabilities are areas that lack best practices that may cause small complications in the future.
Gas	Gas Optimisations, issues, and inefficiencies

Audit Findings

High

[H-01] Reward Pallet#get_rewards - Node Denial of Service condition due to zero-weight extrinsic

Description

The `reward` pallet defines a `get_rewards` extrinsic that is meant to add a validator to the list of rewards for the era. As per the design, a validator or someone associated with it would be calling this extrinsic. As the operations that it performs are time-dependent and are related to eras in the blockchain, it is necessary for transactions containing a call to `get_rewards` extrinsic to be added to the block as soon as possible. For this reason, it was marked with the weight value of `0` making it possible to add to any block.

However, the weight measurements are related to the amount of work node needs to perform while executing the code in order to complete it. It naturally corresponds to the amount of time needed to process the whole transaction and to the fee the caller needs to pay. A weight of `0` would indicate to the node that such an extrinsic does not perform any work, which is not true. Additionally, such an extrinsic is relatively cheap to execute as the caller would need to pay the length fee, but not the execution fee. As a consequence, a malicious user can flood the network with `get_rewards` calls forcing the nodes to process them. Such action would result in nodes becoming unresponsive, which in turn can lead to halting the whole blockchain.

Vulnerability Details

```
#[pallet::call_index(0)]
#[pallet::weight(Weight::zero())]
pub fn get_rewards(origin: OriginFor<T>, validator: T::AccountId)
-> DispatchResult { … }
```

Using zero for an extrinsic's weight is generally considered a vulnerability in Substrate-based blockchains. The weight system is designed to accurately represent the computational cost of executing extrinsics, which is crucial for maintaining the stability and security of the network. By setting the weight to zero, the true computational cost of the extrinsic is not accounted for.

Impact

The blockchain may become congested if too many zero-weight extrinsics are executed, possibly overwhelming node resources. Blocks may be delayed as weights are used to estimate how many transactions can be fit. Denial of Service of nodes trying to execute such an extrinsic.

Recommendation

It is recommended to calculate the appropriate weight for the extrinsic (for instance via benchmarking) and use it instead of zero-weight. Additionally, to make sure that the extrinsic is always included, `Mandatory dispatches` can be considered, which are inherent transactions that are submitted by the block author.

Status

Resolved

Medium

[M-01] Pallets - Multiple weight calculation issues

Description

It was observed that in many places within the codebase, the weights corresponding to the extrinsics are not calculated properly. Substrate weight is an indicator for nodes processing transactions on the complexity of the execution, which in turn translates to the estimated time needed to complete it along with the fees associated with it. Invalid weight calculation leads to inaccurate user billing and might lead to delayed block creation in cases when the weight is underestimated. In cases it is overestimated, it will

lead to higher than necessary weight fees and decreased network throughput as overestimated extrinsics take more space in the block.

The identified issues are as follows:
- The `upsert_esg_scores` function from the `ESG` pallet has a set weight with 1001 reads and 1000 writes to the database, while in fact this number might differ. Furthermore, the `upsert_esg_scores` function takes a `WeakBoundedVec` as a parameter. The length of this parameter is not taken into account during weight calculation.
- The `upsert_esg_scores`, `register_an_oracle` and `deregister_an_oracle` function from the `ESG` pallet all call `Self::is_an_oracle` function which loops over the Vectors containing Sudo Oracles and NonSudo Oracles. The execution of mentioned function will depend on the length of those Vectors, which is not taken into account

Impact

Potential Denial of Service scenarios, or inability to properly execute the extrinsics.

Recommendation

It is recommended to take into account lengths of a dynamic data structure, especially if they are provided by the caller. Additionally, the number of read and write operations should be calculated accurately, usually depending on the length of dynamic data structures.

Status

Acknowledged

[M-02] ESG Pallet - Oracle can create other Oracles

Description

The `ESG` pallet defines an Oracle role which is responsible for providing data related to the E, S and G scores for various organisations enrolled as validators. However, it was observed that if an Oracle is a so-called `sudo oracle` then it can register new oracles as though it was a root origin. As a consequence, a single malicious oracle can create

arbitrarily many new malicious oracles, which if created as `sudo oracles` can also create more.

Impact

Potential centralization issue and single-point-of-failure when some key leakage will happen.

Recommendation

It is recommended to allow only a privileged role to create new Oracles.

Status

Acknowledged

Gas

[G-01] Reward Pallet - Repetitive calls to one-shot function

Description

The code presents a claim_rewards function in the reward pallet that distributes rewards to validators and their nominators. An informational issue has been identified in the reward distribution logic for nominators.

Vulnerability Details

```
nominators.iter().for_each(|nominator| {
let _ = Self::distribute_validator_reward(validator.clone());
let _ = Self::distribute_nominator_reward(nominator.who.clone());
});
```

The distribute_validator_reward() function is called in each iteration of the nominator loop. However, this function is designed to distribute the reward to the validator and remove the reward from storage once it's distributed. Calling it multiple times is unnecessary and inefficient.

Impact

While this issue doesn't present a security vulnerability, it has the following impacts:

1. Inefficiency: The function unnecessarily attempts to distribute the validator's reward multiple times, which is wasteful in terms of computation.
2. Potential confusion: The code's intent becomes unclear, which could lead to maintenance issues or confusion for other developers.

Recommendation

To address this informational issue, move the function call outside of the loop.

Status

Resolved

[G-02] Reward Pallet - Unnecessary cast to f64

Description

The code casts the era_reward value, which is already of type `f64`, to `f64`.

Vulnerability Details

```
let era_reward = Self::calculate_era_reward();
let total_reward = era_reward as f64;
```

This cast is redundant as it's converting a value to the same type it already is.

Impact

Slight readability issue.

Recommendation

Remove the unnecessary cast.

Status

Resolved

Conclusion

After Hashlocks analysis, the 5ire project seems to have a sound and well-tested code base, now that our vulnerability findings have been resolved and acknowledged. Overall, most of the code is correctly ordered and follows industry best practices. The code is well commented on as well. To the best of our ability, Hashlock is not able to identify any further vulnerabilities.

Our Methodology

Hashlock strives to maintain a transparent working process and to make our audits a collaborative effort. The objective of our security audits is to improve the quality of systems and upcoming projects we review and to aim for sufficient remediation to help protect users and project leaders. Below is the methodology we use in our security audit process.

Manual Code Review:

In manually analysing all of the code, we seek to find any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our methodologies include manual code analysis, user interface interaction, and white box penetration testing. We consider the project's website, specifications, and whitepaper (if available) to attain a high-level understanding of what functionality the smart contract under review contains. We then communicate with the developers and founders to gain insight into their vision for the project. We install and deploy the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We undergo a robust, transparent process for analysing potential security vulnerabilities and seeing them through to successful remediation. When a potential issue is discovered, we immediately create an issue entry for it in this document, even though we still need to verify the feasibility and impact of the issue. This process is vast because we document our suspicions early even if they are later shown not to represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, and then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyse the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinised by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the contract details are made public.

Disclaimers

Hashlock's Disclaimer

Hashlock's team has analysed these smart contracts in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in the smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Hashlock is not responsible for the safety of any funds and is not in any way liable for the security of the project.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to attacks. Thus, the audit can't guarantee the explicit security of the audited smart contracts.

About Hashlock

Hashlock is an Australian-based company aiming to help facilitate the successful widespread adoption of distributed ledger technology. Our key services all have a focus on security, as well as projects that focus on streamlined adoption in the business sector.

Hashlock is excited to continue to grow its partnerships with developers and other web3-oriented companies to collaborate on secure innovation, helping businesses and decentralised entities alike.

Website: hashlock.com.au
Contact: info@hashlock.com.au


#Hashlock.