HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: Gable Finance
Date: 01 Nov, 2023

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Document

Name	Smart Contract Code Review and Security Analysis Report for Gable Finance
Approved By	Luciano Ciattaglia | Director of Services at Hacken OÜ
Auditor	Jakub Heba
Auditor	Vladyslav Khomenko
Tags	Flashloans, Staking
Platform	Radix DLT
Language	Rust/Scrypto
Methodology	Link
Website	https://gable.finance/
Changelog	10.10.2023 - Initial Review 01.11.2023 - Second Review

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Table of contents

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Introduction

Hacken OÜ (Consultant) was contracted by Gable Finance (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report presents the findings of the security assessment of the Customer's smart contracts.

System Overview

Gable is a protocol creating the liquidity market on Radix DLT. Users can borrow funds without collateral in the form of flash loans, as well as earn staking rewards and interest earnings from supplying their tokens to the protocol.

Users that are staking XRD on the *Gable* validator receive liquid staking units (LSU tokens) that can be deposited in the *flashloan* pool to earn some interest.

Borrowers that take loans repay it with some interest that is then split 50-50 between the users that deposited LSU and the smart contract owner.

Privileged roles

- The owner of the contract could perform multiple administrative changes, like changing interest rates, updating suppliers key value store, deposit and withdraw liquidity to the pool directly, depositing and withdrawing validator node ownership token, as well as perform unstaking and claiming operations.
- The admin role can perform most of the owner’s actions, except from depositing and withdrawing liquidity and validator node ownership token.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Executive Summary

The score measurement details can be found in the corresponding section of the scoring methodology.

Documentation quality

The total Documentation Quality score is **10** out of **10**.

- Functional and technical requirements were provided.
- Technical description and diagrams were provided.
- The code implements complex calculations logic with small amounts of descriptions and requirements.

Code quality

The total Code Quality score is **10** out of **10**.

- The development environment is configured.
- The code is readable and easy to digest.
- Most of the methods are described with appropriate comments.
- Test cases are well described with requirements.

Test coverage

Code coverage of the project could not be directly calculated with common tools due to the likely lack of support for Scrypto. Nevertheless, taking into account the functional coverage and the number of tests available in the code repository - the tests cover approximately **2/3** of the codebase.

- Deployment and basic user interactions are covered with tests.
- Negative cases coverage is missed.
- Interaction with validator and methods associated with these operation are not covered

Security score

As a result of the audit, the code contains no issues. The security score is **10** out of **10**.

All found issues are displayed in the “Findings” section.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Summary

According to the assessment, the Customer's smart contract has the following score: **8.7**.

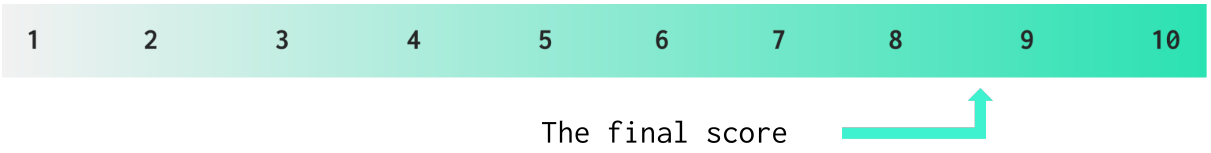

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
09 October 2023	5	2	1	0
1 November 2023	0	0	0	0

Risks

- The smart contract could be upgraded and its functionality may be changed.
- Centralization and the owner's ability to withdraw whole liquidity from the pool might be dangerous, if his wallet/badge will be compromised.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Checked Items

We have audited the Customers' smart contracts for commonly known and specific vulnerabilities. Here are some items considered:

Item	Description	Status	Related issues
Integer Overflow and Underflow	If unchecked math is used, all math operations should be safe from overflows and underflows.	Passed	
Unchecked Errors	If a function returns a Result, it should not be ignored.	Passed	
Access Control & Authorization	Ownership takeover should not be possible. All crucial functions should be protected. Users could not affect data that belongs to other users.	Passed	
Assert Violation	Properly functioning code should never reach a failing assert statement.	Passed	
Deprecated Rust Functions	Deprecated built-in functions should never be used.	Passed	
DoS (Denial of Service)	Execution of the code should never be blocked by a specific contract state unless required.	Passed	
Block values as a proxy for time	Block numbers should not be used for time calculations.	Not Relevant	
Signature Reuse	Signed messages that represent an approval of an action should not be reusable.	Not Relevant	
Weak Sources of Randomness	Random values should never be generated from Chain Attributes or be predictable.	Not Relevant	
Race Conditions	Race Conditions and Transactions Order Dependency should not be possible.	Passed	
Calls Only to Trusted Addresses	All external calls should be performed only to trusted addresses.	Passed	
Presence of Unused Variables	The code should not contain unused variables if this is not justified by design.	Passed	
Assets Integrity	Funds are protected and cannot be withdrawn without proper permissions or be locked on the contract.	Passed	

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

User Balances Manipulation	Contract owners or any other third party should not be able to access funds belonging to users.	Passed	
Data Consistency	Smart contract data should be consistent all over the data flow.	Passed	
Flashloan Attack	When working with exchange rates, they should be received from a trusted source and not be vulnerable to short-term rate changes that can be achieved by using flash loans. Oracles should be used.	Passed	
Token Supply Manipulation	Tokens can be minted only according to rules specified in a whitepaper or any other documentation provided by the Customer.	Passed	
Gas and Loops	Transaction execution costs should not depend dramatically on the amount of data stored on the contract.	Passed	
Compiler Warnings	The code should not force the compiler to throw warnings.	Passed	
Requirements Compliance	The code should be compliant with the requirements provided by the Customer.	Passed	
Environment Consistency	The project should contain a configured development environment with a comprehensive description of how to compile, build and deploy the code.	Passed	
Secure Oracles Usage	The code should have the ability to pause specific data feeds that it relies on. This should be done to protect a contract from compromised oracles.	Not Relevant	
Tests Coverage	The code should be covered with unit tests. Test coverage should be sufficient, with both negative and positive cases covered. The usage of contracts by multiple users should be tested.	Failed	
Stable Imports	The code should not reference draft contracts, that may be changed in the future.	Passed	
Unsafe Rust code	The Rust type system does not check the memory safety of unsafe Rust code. Thus, if a smart contract contains any unsafe Rust code, it may still suffer from memory corruptions such as buffer overflows, use after frees, uninitialized memory, etc.	Passed	

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Findings

■■■■ Critical

No critical severity issues were found.

■■■ High

H01. Owner could withdraw more than he has deposited as owner_liquidity

Impact	High
Likelihood	Medium

The owner can provide funds to the flashloan pool in the form of a deposit, so that there is some liquidity in the protocol - as well as withdraw them. The functions *owner_deposit_xrd* and *owner_withdraw_xrd* are used for this purpose.

An incorrect validation was found in the *owner_withdraw_xrd* function regarding verification whether the amount paid is equal to or less than the liquidity in the pool. Unline, there should be a check comparing the mentioned amount with the amount of liquidity deposited by the owner.

```
pub fn owner_withdraw_xrd(&mut self, amount: Decimal) -> Bucket {
    // Ensure amount is positive
    assert!(
        amount > Decimal::ZERO,
        "Please withdraw an amount larger than 0"
    );

    // Ensure amount is less or equal to liquidity provided by owner
    assert!(
        amount <= self.liquidity_pool_vault.amount(),
        "Please withdraw an amount smaller than or equal to {}",
        self.owner_liquidity
    );
```

Consequently, if, via the *claim_xrd* function, there are more funds in the pool than the owner initially deposited, then he is able to withdraw all of them, even though they are not his property. This is possible because amounts and liquidity are based on values of the *Decimal* type, which can be negative. Then the contract will not return panic if *owner_liquidity* drops below zero, for example to -1000.

Proof of Concept:

Test shows that after the validator sends funds to the liquidity pool, the owner is able to withdraw 400 tokens more than he himself deposited.

```
#[test]
fn owner_withdraws_more_than_deposited() -> Result<(), RuntimeError> {
    //
    let (mut env, mut flashloanpool) = setup_flashloan_pool()?;
    let xrd_bucket: Bucket = env.with_auth_module_disabled(|env| {
        // owner deposits 100 XRD
        let rtn = ResourceManager(XRD).mint_fungible(100.into(), env);
        let _ = flashloanpool.owner_deposit_xrd(rtn.unwrap(), env);
        // validator sends 1000 XRD as staking rewards
        let rtn2 = ResourceManager(XRD).mint_fungible(1000.into(), env);
        let _ = flashloanpool.deposit_batch(rtn2.unwrap(), env);
        // owner withdraws 500 XRD, 400 more than he deposited
        let bucket = flashloanpool.owner_withdraw_xrd(dec!("500"), env);
        bucket
    })?;

    let flashloanpool_state =
env.read_component_state::<FlashloanpoolState, _>(flashloanpool)?;
    let xrd_amount = flashloanpool_state.liquidity_pool_vault.amount(&mut
env)?;
    let owner_amount = flashloanpool_state.owner_liquidity;
    // Tokens left in the pool
    assert_eq!(xrd_amount, dec!("600"));
    // Tokens owned by the owner - as far as he withdraws more than he
deposited, value is lower than zero
    assert_eq!(owner_amount, dec!("-400"));
    Ok(())
}
```

Path: ./flashloan-pool/src/lib.rs : owner_withdraw_xrd()

Recommendation: *owner_withdraw_xrd* function should be adjusted so that assert is checking that the amount requested is equal to or less than the liquidity owned by the owner.

Found in: afeb034

Status: Fixed (Revised commit: a3fe638)

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Medium

M01. Missing upper bound on interest rate change

Impact	Medium
Likelihood	Medium

It was noticed that one of the methods available only to the contract *OWNER* or wallet with the *ADMIN* role is to change the interest rate in the protocol. While there is validation that this value is not negative, so there is no upper maximum value it can take. As a consequence, the user may pay a "fee" several times higher than the loan amount he took out.

This is especially dangerous in the current configuration, in which *OWNER* is a single wallet and is exposed to compromise.

Path: ./flashloan-pool/src/lib.rs : update_interest_rate()

Recommendation: A maximum cap on the *interest_rate* variable should be implemented so that even the *OWNER* of the contract cannot set it to an illogically high value that affects critical protocol functionality.

Found in: afeb034

Status: Fixed (Revised commit: cedf40a)

M02. Missing validations in multiple calculations could lead to unexpected state

Impact	Medium
Likelihood	Medium

Most of the numeric variables in the contract are of *Decimal* type. Unlike *Uint*, *Decimal* allows you to store and manipulate negative numbers. If the business logic does not take into account a number less than zero in a given context, appropriate validation should take place so that if the "zero" threshold is exceeded, an error is returned.

This situation can be observed in the *update_aggregate_im* function, where during the *interest_new* calculation it is not verified whether *owner_liquidity* is not equal to *total_liquidity*, so if the values of *rewards_new*, *rewards_aggregated* or *interest_aggregated* are greater than zero, then *interest_new* will be negative.

Additionally, in the *update_supplier_kvs* function it was noticed that there is no validation whether *box_lsu* is different from zero, which

may cause the contract to panic when calculating the *supplier_relative_lsu_stake* variable.

Path: ./flashloan-pool/src/lib.rs : update_interest_rate()

Recommendation: We suggest adding additional validations in all places where *Decimal* values should not exceed the zero threshold. Additionally, the suggested solution is to verify all divisor values in the context of their being different from zero.

Found in: afeb034

Status: Fixed (Revised commit: e5c8e29)

Low

L01. Macros used for debugging should not be used in production code

Impact	Low
Likelihood	Medium

The current contract code has been found to contain many instances of debugging macros, such as *debug!()* and *info!()*. While they are very helpful during the code development and testing phase, their use in production code is considered bad practice. Additionally, each operation that stores some data in memory causes the virtual machine to perform some work - and thus increases the cost of gas needed to perform this transaction.

Path: ./flashloan-pool/src/lib.rs : most of the contract's functions

Recommendation: If this type of methods are used to transfer certain state to off-chain components, for example in the form of logs - it seems more appropriate to use event emitting. Otherwise, you should consider removing unnecessary code fragments from the codebase.

Found in: afeb034

Status: Fixed (Revised commit: 6cfbaf4)

L02. Owner is able to unlock and update royalties for function calls

Impact	Low
Likelihood	Low

The owner has set up some royalties for public methods and even specified that the amount is locked. However they reserved a right to unlock and update the amount they charge from users for calling these methods (protocol enforces a hard cap).

Since the royalties are marked as locked, the assumption is that they should be unchanged.

```
.enable_component_royalties(component_royalties! {
    roles {
        royalty_setter => OWNER;
        royalty_setter_updater => OWNER;
        royalty_locker => OWNER;
        royalty_locker_updater => OWNER;
        royalty_claimer => OWNER;
        royalty_claimer_updater => OWNER;
    },
    init {
         get_flashloan => Xrd(1.into()), locked;
         ...
```

Path: ./flashloan-pool/src/lib.rs : instantiate_flashloan_pool()

Recommendation: Restrict ability to unlock royalties and/or mark the royalties as *updatable*. You can also specify the amount to be an approximate USD equivalent with *Usd(amount.into())*.

Found in: afeb034

Status: Fixed (Revised commit: aee1aca)

L03. Wrong limit for the size of box

The *box_size* is set to *250* in the beginning. It is possible to lower it, but not possible to change it back to *250*. This change multiplies as the *box_size * box_size* is the maximum possible number of users of the contract.

Path: ./flashloan-pool/src/lib.rs : update_box_size()

Recommendation: Check the value inclusively.

Found in: afeb034

Status: Fixed (Revised commit: a748fc7)

L04. Floating Language Version

It is preferable for a production project, especially a smart contract, to have the programming language version pinned explicitly. This results in a stable build output, and guards against unexpected toolchain differences or bugs present in older versions, which could be used to build the project.

The language version could be pinned in automation/CI scripts, as well as proclaimed in README or other kinds of developer documentation. However, in the Rust ecosystem, it can be achieved more ergonomically via a *rust-toolchain.toml* descriptor (see https://rust-lang.github.io/rustup/overrides.html#the-toolchain-file)

Paths: ./rust-toolchain.toml

Recommendation: Pin the language version at the project level.

Found in: afeb034

Status: Fixed (Revised commit: e8da2c1)

L05. Test functions should be removed

Impact	Low
Likelihood	Low

One of the methods available for public call is *deposit_batch*. It is used to manually add funds to stacking rewards, however, both in the documentation and in the contract code it is specified as “Temporary” due to the need to simulate the validator during the testing phase.

While it does not pose a direct threat to the contract, such features should not be available in the production version of the code.

Path: ./flashloan-pool/src/lib.rs : deposit_batch()

Recommendation: You should make sure that the final version of the protocol will be free from test functions and these simulating certain operations. In code marked as "release version", the *deposit_batch* function and its associated method should be removed.

Found in: afeb034

Status: Fixed (Revised commit: 49452f7 and 53f392b)

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Informational

I01. Suggestion for searching a vacant box

In the *deposit_lsu* function, code is looking for an index of a box (*box_nr*) for saving deposit info.

```
for (key, values) in &self.supplier_aggregate_im {
    // Check if the Vec is not empty and satisfies your condition
    if let Some(first_value) = values.first() {
        if *first_value < self.box_size.into() {

            // update box number
            box_nr = *key;

            // update existing supplier's info before adding a new
supplier
            self.update_supplier_kvs(box_nr);

            // Set the flag to true to indicate that the condition has
been satisfied
            condition_satisfied = true;

            break;
        }
    }
}
```

After that, condition is checked, and based on a result, some actions are performed:

```
if condition_satisfied {
    // Scenario 1: Add new supplier to the existing key value store and
index map
    ...
} else {
    // Scenario 2: In case that all boxes are full or no box exists, a
new box has to be inserted
    ...
}
```

The code can be simplified to improve Code Quality.

Path: ./flashloan-pool/src/lib.rs : deposit_lsu()

Recommendation: Consider using optional box index, and matching on the option. Sample implementation is lister below:

```
let mut vacant_box = None;
for (box_nr, values) in &self.supplier_aggregate_im {
    if values.first().is_some_and(|suppliers_in_box| *suppliers_in_box <
self.box_size.into()) {
        vacant_box = Some(*box_nr);
        break;
    }
}

match vacant_box {
    Some(box_nr) => ...,
    None => ...,
}
```

Found in: afeb034

Status: Fixed (Revised commit: 35f25c6)

I02. Unformatted Code

cargo fmt yields changes in 2 files total. Formatting the code is recommended for good Code Quality.

Path:

- ./flashloan-pool/src/events.rs
- ./flashloan-pool/src/lib.rs

Recommendation: Consider formatting the code using *rustfmt* or an equivalent.

Found in: afeb034

Status: Fixed (Revised commit: 16dd2e3)

I03. The contract code is a single monolith file

In this commit, the *lib.rs*, main source file has 1009 lines. Splitting it into separate files/modules would increase its readability, hence its quality. Big chunks of logic can be extracted into separate functions, even if they are called once; just to make it easier to digest and reason about.

Path: ./flashloan-pool/src/lib.rs

Recommendation: Consider splitting large functions into smaller and/or moving the actual code of the contract's methods into a separate file.

Found in: afeb034

Status: Fixed (Revised commit: 0b13351)

I04. Suggestions for idiomatic code style

`cargo clippy` is a popular tool for catching common mistakes and improving the code. It reports some possibly useful code changes.

Path: ./flashloan-pool/*

Recommendation: Consider following its suggestions and/or using `cargo clippy --fix` to apply some of them automatically.

Found in: afeb034

Status: Fixed (Revised commit: 82f9ac1)

I05. Former name is mentioned

The README.md file as well as Functional Requirements mentions the old protocol name.

Path: ./flashloan-pool/README.md

Recommendation: Consider replacing it with the relevant name.

Found in: afeb034

Status: Fixed (Revised commit: 794794e)

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in 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).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

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 recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

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 vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Appendix 1. Severity Definitions

When auditing smart contracts Hacken is using a risk-based approach that considers the potential impact of any vulnerabilities and the likelihood of them being exploited. The matrix of impact and likelihood is a commonly used tool in risk management to help assess and prioritize risks.

The impact of a vulnerability refers to the potential harm that could result if it were to be exploited. For smart contracts, this could include the loss of funds or assets, unauthorized access or control, or reputational damage.

The likelihood of a vulnerability being exploited is determined by considering the likelihood of an attack occurring, the level of skill or resources required to exploit the vulnerability, and the presence of any mitigating controls that could reduce the likelihood of exploitation.

Risk Level	High Impact	Medium Impact	Low Impact
High Likelihood	**Critical**	**High**	**Medium**
Medium Likelihood	**High**	**Medium**	**Low**
Low Likelihood	**Medium**	**Low**	**Low**

Risk Levels

Critical: Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.

High: High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.

Medium: Medium vulnerabilities are usually limited to state manipulations and in most cases cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.

Low: Major deviations from best practices or major Gas inefficiency. These issues won't have a significant impact on code execution, don’t affect security score but can affect code quality score.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Impact Levels

High Impact: Risks that have a high impact are associated with financial losses, reputational damage, or major alterations to contract state. High impact issues typically involve invalid calculations, denial of service, token supply manipulation, and data consistency, but are not limited to those categories.

Medium Impact: Risks that have a medium impact could result in financial losses, reputational damage, or minor contract state manipulation. These risks can also be associated with undocumented behavior or violations of requirements.

Low Impact: Risks that have a low impact cannot lead to financial losses or state manipulation. These risks are typically related to unscalable functionality, contradictions, inconsistent data, or major violations of best practices.

Likelihood Levels

High Likelihood: Risks that have a high likelihood are those that are expected to occur frequently or are very likely to occur. These risks could be the result of known vulnerabilities or weaknesses in the contract, or could be the result of external factors such as attacks or exploits targeting similar contracts.

Medium Likelihood: Risks that have a medium likelihood are those that are possible but not as likely to occur as those in the high likelihood category. These risks could be the result of less severe vulnerabilities or weaknesses in the contract, or could be the result of less targeted attacks or exploits.

Low Likelihood: Risks that have a low likelihood are those that are unlikely to occur, but still possible. These risks could be the result of very specific or complex vulnerabilities or weaknesses in the contract, or could be the result of highly targeted attacks or exploits.

Informational

Informational issues are mostly connected to violations of best practices, typos in code, violations of code style, and dead or redundant code.

Informational issues are not affecting the score, but addressing them will be beneficial for the project.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Initial review scope

Repository	The scope of the audit is a part of the repository: https://github.com/gable-finance/gable/tree/main/src/backend/scrypto/flashloan-pool/
Commit	afeb0343533798020630fcf45432abce7580b7e8
Whitepaper	Link
Requirements	Link
Technical Requirements	Link
Contracts	File: ./flashloan-pool/src/events.rs SHA3: 60281431dfd06c6392f01fedccd6b01a189101edaef3401cd4a6910b File: ./flashloan-pool/src/lib.rs SHA3: 75250e5c25471a463a9f82e54fcec1633e9e9fcd04379233aa5f36d8

Second review scope

Repository	The scope of the audit is a part of the repository: https://github.com/gable-finance/gable/tree/main/src/backend/scrypto/flashloan-pool/
Commit	82f9ac166b194a7b437deb35dabf3c8cccd6f327
Whitepaper	Link
Requirements	Link
Technical Requirements	Link
Contracts	File: ./flashloan-pool/src/events.rs SHA3: 30ab06eb53d6444a1ff3d1430834230549ac06b4ec3083cf21b993774e092e25 File: ./flashloan-pool/src/lib.rs SHA3: 09618aad988af20b259fec5faa1bbae678cf54517f108319d2c0c99e2fce7e02 File: ./flashloan-pool/src/nft_data.rs SHA3: 94d831216fb51c3129417545d8055b49f458664a04be748f9182a160c88d5c58 File: ./flashloan-pool/src/tokens.rs SHA3: c05b032f8bd908354545e17c1b53f751a0c45637f0efc10575d22fe9daf1985f