HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: Clearpool.finance
Date: 23 Aug, 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 Clearpool.finance
Approved By	Oleksii Zaiats | SC Audits Head at Hacken OÜ
Type	Staking (Lending Protocol Plugin)
Platform	EVM
Language	Solidity
Methodology	Link
Website	https://clearpool.finance
Changelog	21.07.2023 - Initial Review 10.08.2023 - Second Review 23.08.2023 - Third 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 Clearpool.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

The Term Protocol is an extension of the Clearpool Permissionless smart contracts. Its purpose is to allow users to lock their liquidity on *TermPool* over a fixed period of time for the promise (in the form of the *tpToken*) to receive additional APR.

There are several smart contracts in the audit scope:

- *TermUtils* – helper abstract contract.
- *TermPoolFactory* – contract for *TermPool* contracts deployment and management.
- *TermPool* – contract for locking liquidity for rewards, deploy *tpToken* contracts for each liquidity lock (*Term*).
- *TpToken* – simple mintable ERC20 token contract.

Roles

TermPoolFactory:

- Owner of *permissionlessFactory* is able to:
  - Change *TermPool* implementation used for deployment
  - Change *TpToken* implementation used for deployment by *TermPool*
  - Update *permissionlessFactory* address

TermPool:

- Borrower is able to:
  - Create a term to allow users locking liquidity
  - Top-up the contract with rewards to lenders
  - Cancel term if no one lent funds yet
- Owner of *permissionlessFactory* is able to:
  - Pause ability to lock liquidity in terms
  - Allow partial rewards top-up for the borrower

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 **6** out of **10**.

- Public functional requirements are not provided.
- Internal functional requirements are comprehensive.
- Technical description is not provided.
- Essential scripts are set up in the *package.json* file.

Code quality

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

- Development environment is set up.

Test coverage

Code coverage of the project is **100%** (branch coverage).

Security score

As a result of the audit, the code does not contain security issues. The security score is **10** out of **10**.

All found issues are displayed in the Findings section of the report.

Summary

According to the assessment, the Customer's smart contract has the following score: **9.6**.

The system users should acknowledge all the risks summed up in the Risks section of the report.

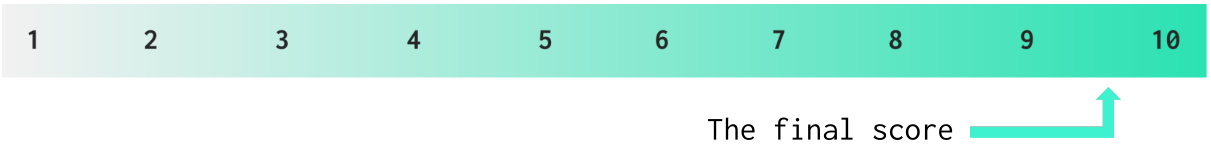

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
21 July 2023	3	2	0	0
10 August 2023	3	0	0	0
23 August 2023	0	0	0	0

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

Risks

- The smart contracts system is upgradeable. In case of a key leak, an attacker may receive access to user funds.
- The system highly depends on the *permissionlessFactory* state and implementation, which is out of the audit scope.
- The borrower may not provide any rewards for the lock activity.
- Rewards locked on the *TermPool* contract may not correspond to the promised reward rate. Users are unable to check if the *permissionlessFactory* owner allowed partial reward top-up for a specific *Term*.

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
Default Visibility	Functions and state variables visibility should be set explicitly. Visibility levels should be specified consciously.	Passed
Integer Overflow and Underflow	If unchecked math is used, all math operations should be safe from overflows and underflows.	Passed
Outdated Compiler Version	It is recommended to use a recent version of the Solidity compiler.	Passed
Floating Pragma	Contracts should be deployed with the same compiler version and flags that they have been tested thoroughly.	Passed
Unchecked Call Return Value	The return value of a message call should be checked.	Not Relevant
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
SELFDESTRUCT Instruction	The contract should not be self-destructible while it has funds belonging to users.	Not Relevant
Check-Effect-Interaction	Check-Effect-Interaction pattern should be followed if the code performs ANY external call.	Passed
Assert Violation	Properly functioning code should never reach a failing assert statement.	Passed
Deprecated Solidity Functions	Deprecated built-in functions should never be used.	Passed
Delegatecall to Untrusted Callee	Delegatecalls should only be allowed to trusted addresses.	Not Relevant
DoS (Denial of Service)	Execution of the code should never be blocked by a specific contract state unless required.	Passed
Race Conditions	Race Conditions and Transactions Order Dependency should not be possible.	Passed
Authorization through tx.origin	tx.origin should not be used for authorization.	Not Relevant

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

Block values as a proxy for time	Block numbers should not be used for time calculations.	Not Relevant
Signature Unique Id	Signed messages should always have a unique id. Chain identifiers should always be used. All parameters from the signature should be used in signer recovery. EIP-712 should be followed.	Not Relevant
Shadowing State Variable	State variables should not be shadowed.	Passed
Weak Sources of Randomness	Random values should never be generated from Chain Attributes or be predictable.	Not Relevant
Incorrect Inheritance Order	When inheriting multiple contracts, especially if they have identical functions, a developer should carefully specify inheritance in the correct order.	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
EIP Standards Violation	EIP standards should not be violated.	Passed
Assets Integrity	Funds are protected and cannot be withdrawn without proper permissions or be locked on the contract.	Passed
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. Contracts shouldn't rely on values that can be changed in the same transaction.	Passed
Token Supply Manipulation	Tokens should be minted only according to rules specified in a whitepaper or any other documentation provided by the Customer.	Passed
Gas Limit and Loops	Transaction execution costs should not depend dramatically on the amount of data stored on the contract. There should not be any cases when execution fails due to the block Gas limit.	Passed

Style Guide Violation	Style guides and best practices should be followed.	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.	Failed (Documentation)
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. Usage of contracts by multiple users should be tested.	Passed
Stable Imports	The code should not reference draft contracts, which may be changed in the future.	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

No high severity issues were found.

Medium

M01. Stuck Funds

Impact	Medium
Likelihood	Medium

There is no guarantee that the borrower will engage with the smart contract. In this case, the lender will not receive any expected reward and their funds will be stuck for a period of time, specified in *term.maturityDate*, possibly many months.

It is not possible for a borrower to deposit rewards for lenders before they deposit their liquidity. Once the lenders deposit their cpTokens, it is not possible to withdraw them until the end of term, even if there's no reward for them for participating in such a contract.

Path: ./contracts/TermPool.sol: lock(), topupReward()

Recommendation: Devise a way for ensuring mutual commitment. For example, allow the parties to withdraw their deposits to match the other party's deposit, according to the reward calculation.

Found in: d3bdc28

Status: Mitigated (The behavior is intended, there is no guarantee that rewards would be provided)

M02. Highly Permissive Role

Impact	Medium
Likelihood	Medium

Rewards meant for lenders can be withdrawn to borrowers by the protocol governor at any point. This will strip the liquidity providers of their reward.

Path: ./contracts/TermPool.sol: withdrawReward()

Recommendation: Restrict withdrawal access if any liquidity is provided. If there is a need to ensure there are no undistributed rewards left in the contract after all transactions, any leftovers

can be automatically returned to the borrower after all liquidity is withdrawn.

Found in: d3bdc28

Status: Fixed (Revised commit: 47f4682)

Low

L01. Unused Return Value

Impact	Low
Likelihood	Low

Return values of the functions are not being validated.

Handling the return values may be helpful to avoid reentrancies and erroneous function execution results.

Path: ./contracts/TermPool.sol: createTerm(), unlock(), cancelTerm()

Recommendation: Perform return value checks.

Found in: d3bdc28

Status: Fixed (Revised commit: 47f4682)

L02. Token Symbols Collision

Impact	Low
Likelihood	Medium

The algorithm trims 12 bits from a hash to use as part of the token symbol. Tokens may have the same symbols for different *termId* values.

For example, if the token symbol is *cpAPP-LINK*, and *termId* is 5 or 25, the resulting symbol is *tpAPE-326*.

Path: ./contracts/TermPool.sol: createTerm()

Recommendation: Consider using 2.5-4 bytes (5-8 characters) to make the collision probability quite small.

Found in: d3bdc28

Status: Fixed (Revised commit: d0cbc8e)

L03. Unbounded Loop

Impact	Medium
Likelihood	Low

The *listedPoolsCount* value is not limited. The function may fail due to Block Gas Limit being exceeded.

It is impossible to retrieve keys of *poolsByCpToken* mapping. Therefore, it is impossible to retrieve needed information in case many *TermPool* instances are deployed.

Path: ./contracts/TermPoolFactory.sol: getPools()

Recommendation: Limit the *listedPoolsCount* value, allow viewing keys of *poolsByCpToken* mapping.

Found in: d3bdc28

Status: Mitigated (on behalf of L04)

L04. Inefficient Gas Model

Impact	Medium
Likelihood	Low

The function returns an array of unlimited length. The transaction may be rejected due to consuming too much Gas.

Path: ./contracts/TermPoolFactory.sol: getUsedCpTokens()

Recommendation: Provide a function that allows retrieving elements of the array by index. The behavior could be reached by changing the *_usedCpTokens* variable visibility to *public*.

Found in: 47f4682

Status: Fixed (Revised commit: d0cbc8e)

L05. Token Dust Lock

Impact	Low
Likelihood	Medium

Some dust tokens may be locked on the contract.

This may be impactful in case the term token has a little number of decimals and a lot of users participate in the term.

There may be more rewards (up to *number of users deposited - 1*) than could be distributed.

Path: ./contracts/TermPool.sol

Each user may lose up to *1 cpToken* due to division before multiplication in case of partial rewards repayment allowed.

The *rewardOf* function performs division and then the result is multiplied in the *availableRewardOf* function. Example: let it be actual user reward is 1.5 and the reward availability rate is 90%, according to the current implementation, the user will receive 0 reward due to 1.5 being rounded to 1 and then multiplied by 0.9.

Path: ./contracts/TermPool.sol: availableRewardOf(), rewardOf()

Recommendation: Provide an ability to withdraw excessive reward after the term comes to the *Repayed* status.

Found in: 47f4682

Status: Mitigated (Tokens with 6+ decimals are lowly affected by the issue)

Informational

I01. State Variables Default Visibility

The variable visibilities are not specified. Specifying state variable visibility helps to catch incorrect assumptions about the scope of the variable accessibility.

Path: ./contracts/TermPool.sol: activeTermsIndex()

Recommendation: Specify variables as public (not available for *EnumerableSet*), internal, or private. Explicitly define visibility for all state variables.

Found in: d3bdc28

Status: Fixed (Revised commit: 47f4682)

I02. Inconsistent Operation

It is possible to get the remainder of the division using *%* operation. The *uint8(_i - (_i / 10) * 10))* code looks confusing.

Path: ./contracts/utils/TermUtils.sol: uint2bytes()

Recommendation: Use the modulo operation to get the remainder of the division.

Found in: d3bdc28

Status: Fixed (Revised commit: 47f4682)

I03. Redundant Statements

Avoid redundant statements in the code.

Redundant imports are presented.

Paths:

- ./contracts/TermPool.sol: IPermissionlessPoolFactory
- ./contracts/TermPoolFactory.sol: TermPool, IClassicPool

Redundant events are presented.

Path: ./contracts/interfaces/ITermPool.sol: FactoryAddressChanged, Borrowed

Redundant errors are presented.

Path: ./contracts/interfaces/ITermPool.sol: NotInMaturityWindow, TermSizeIsTooSmall, TermIsNotActive

Recommendation: Eliminate the redundancies.

Found in: d3bdc28

Status: Fixed (Revised commit: 47f4682)

I04. Check-Effect-Interaction Pattern Violations

Violation of the Check-Effect-Interaction pattern may cause possible reentrancy attacks. Provide all interactions with other contracts only after changing all state variables.

The *terms* variable is increased with data generated based on *terms.length* value. After the *terms.length* value is retrieved, an external call is performed, and the value may not be actual at the moment of the array increase.

Path: ./contracts/TermPool.sol: createTerm()

The *_usedCpTokens* variable is increased with no proof of the uniqueness of the pushed value due to an external call being performed and it is not checked that the function was reentered.

Path: ./contracts/TermPoolFactory.sol: createTermPool()

Recommendation: Follow the CEI pattern to eliminate theoretical reentrancies.

Found in: d3bdc28

Status: Mitigated (The functions are implemented under the *nonReentrant* modifier)

I05. Confusing Code

The code performs a set of various actions to get the *tpToken* symbol. However, the algorithm is not documented.

Path: ./contracts/TermPool.sol: createTerm()

Recommendation: Add corresponding documentation or declarative comments to the code.

Found in: d3bdc28

Status: Fixed (Revised commit: 47f4682)

I06. Override Overusage

The *override* keyword should be used to override functions declared as *virtual*. However, it is excessively used in most functions.

Paths:

- ./contracts/TpToken.sol: burn(), mint()
- ./contracts/TermPoolFactory.sol: createTermPool()

- ./contracts/TermPool.sol: cancelTerm(), withdrawReward(), topupReward(), unlock(), lock(), allowPartialRepayment()

Recommendation: Remove unnecessary *override* modifiers.

Found in: d3bdc28

Status: Fixed (Revised commit: 47f4682)

I07. Misleading Name

The *_annualRate* function has a misleading name as it calculates the actual reward amount for a provided period of time.

Path: ./contracts/TermPool.sol: _annualRate()

Recommendation: Provide names in a declarative way.

Found in: d3bdc28

Status: Fixed (Revised commit: 47f4682)

I08. Code Duplication

The *1e18* value is commonly used throughout the contract. However, the *MULTIPLIER* contract is declared.

Path: ./contracts/TermPool.sol

Recommendation: Replace the value with the constant identifier.

Found in: d3bdc28

Status: Fixed (Revised commit: 47f4682)

I09. Suboptimal Algorithm

The conversion from an integer to bytes is a bit inefficient. The number of instructions can be reduced.

Path: ./contracts/TermUtils.sol: uint2bytes()

Recommendation: The loop only needs to do this:

```
_uintAsString[--len] = bytes1(48 + uint8(_i % 10)); _i /= 10;
```

Found in: d3bdc28

Status: Fixed (Revised commit: 47f4682)

I10. Untrimmed Returned Array

It is possible to trim excessive space of memory array using the simple assembly pattern: *assembly { mstore(array, length) }*. It may save some Gas and prevent unexpectedly long outputs.

Path: ./contracts/TermPoolFactory.sol: getPools()

Recommendation: Use the pattern to trim excessive array space.

Found in: d3bdc28

Status: Fixed (Revised commit: 47f4682)

I11. Unused Field

The field *minSize* is never used.

Path: ./contracts/interfaces/ITermPool.sol: Term

Recommendation: Use or remove it.

Found in: d3bdc28

Status: Fixed (Revised commit: 47f4682)

I12. Redundant Check

The *counter < listedPoolsCount* check is useless as it is always *true* if *poolsByCpToken[_usedCpTokens[i]].isListed* and is not reached if the mentioned check is *false*.

Path: ./contracts/TermPoolFactory.sol: getPools()

Recommendation: Change the check order.

Found in: d3bdc28

Status: Fixed (Revised commit: 47f4682)

I13. Burn From Arbitrary Address

The *TermPool* is allowed to burn user funds on the *TpToken* contract without appropriate allowance.

Path: ./contracts/TpToken.sol: burn()

Recommendation: Use the *burnFrom* function of the *ERC20Burnable* extension.

Found in: d3bdc28

Status: Fixed (Revised commit: 47f4682)

I14. Unfinalized Code

The code contains *TODO* comments which imply the code is not finalized.

Paths:

- ./contracts/TermPool.sol
- ./contracts/TermPoolFactory.sol

Recommendation: Implement the code according to the plan or remove the comments.

Found in: d3bdc28

Status: Fixed (Revised commit: 47f4682)

I15. Redundant Statements

Avoid redundant statements in the code.

Redundant events are presented.

Path: ./contracts/interfaces/ITermPool.sol: RewardDrop

Recommendation: Eliminate the redundancies.

Found in: 47f4682

Status: Fixed (Revised commit: d0cbc8e)

I16. Grammar Errors

Some names are spelled wrong: `TermIdNotSetted`, `isTermIdSetted`, `TermStatus.Repayed`.

Recommendation: Use correct spelling.

Found in: 47f4682

Status: Fixed (Revised commit: d0cbc8e)

I17. Confusing Revert Message

The function may revert with `WrongTermState(expected: Created, expected: Created)` error.

Path: contracts/TermPool.sol: cancelTerm()

Recommendation: Avoid confusing revert messages.

Found in: 47f4682

Status: Fixed (Revised commit: d0cbc8e)

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	https://github.com/clearpool-finance/term-protocol
Commit	d3bdc283b23592b45f42c156e9fed98448bc9e88
Requirements	Public requirements are not provided
Contracts	File: ./contracts/TermPool.sol SHA3: a20bac53757243a4ca7be73232c5c4b5648f6e15511f44fa24d2015e5c9215d4 File: ./contracts/TermPoolFactory.sol SHA3: c34b4f27e07f547162c9a61bd53716839af541f30986bccaf301c9bf50e0460b File: ./contracts/TpToken.sol SHA3: 5cb05dec0a747590fd52ca45d167a344a42bec7c183d2b161fc4a6c2607887e8 File: ./contracts/interfaces/IClassicPool.sol SHA3: 9cdd13e7d0c9d7c2e31ce31c3bc49cee625a44aae016680378007d05a56ad5fb File: ./contracts/interfaces/IOwnable.sol SHA3: 43b07e927d69df0a8846563635b48a51570395f8d5745fcdcf545638d8d77af9 File: ./contracts/interfaces/IPermissionlessPoolFactory.sol SHA3: 26805f76eabd7488a9a32bc4d90b4fde70c52c77df00bd075f5d2c9376437cde File: ./contracts/interfaces/IPermissionlessPoolMaster.sol SHA3: 5ec9b2352e8f9a611dbed7208cca838febf7084b25f3c34c6a2152e8a99cc08e File: ./contracts/interfaces/ITermPool.sol SHA3: cafbdf0e1b1c3fdfaa038a009bd9ac05424d631d245adb7f2ea8cb2fffab20f0 File: ./contracts/interfaces/ITermPoolFactory.sol SHA3: 0c514e11fc4d163de32212ff3534a4e562e56c3c5b5ed8a13b0cc2ad8b11bcc5 File: ./contracts/interfaces/ITpToken.sol SHA3: 80c7d51787ceb707a72558511ad2e7e6e9d5bff27bbe7e925c5b2b7e255e6e24 File: ./contracts/utils/TermUtils.sol SHA3: 367317e063b9542d6f3d761bcaf06430b13c2afe65617ed606179e06875790ae

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

Second review scope

Repository	https://github.com/clearpool-finance/term-protocol
Commit	47f46822878932e1a92618be5d7f2d3049a50f87
Requirements	Public requirements are not provided
Contracts	File: contracts/TermPool.sol SHA3: f8b0f91c7ced329ba559e9fa2c43555e109ab3d7650d8b721801ef0aea4745c1 File: contracts/TermPoolFactory.sol SHA3: ddd479c67bfb5d1a1cd42a0aef8978b310da159d2e324e0ded5a50fe77476077 File: contracts/TpToken.sol SHA3: ada036535a315ec5e60edb47e1dd8aa8f9737efbddab718796004b1ed3fa0f83 File: contracts/interfaces/IClassicPool.sol SHA3: 9cdd13e7d0c9d7c2e31ce31c3bc49cee625a44aae016680378007d05a56ad5fb File: contracts/interfaces/IOwnable.sol SHA3: 43b07e927d69df0a8846563635b48a51570395f8d5745fcdcf545638d8d77af9 File: contracts/interfaces/IPermissionlessPoolFactory.sol SHA3: 26805f76eabd7488a9a32bc4d90b4fde70c52c77df00bd075f5d2c9376437cde File: contracts/interfaces/IPermissionlessPoolMaster.sol SHA3: 5ec9b2352e8f9a611dbed7208cca838febf7084b25f3c34c6a2152e8a99cc08e File: contracts/interfaces/ITermPool.sol SHA3: 6a9be87402e0fc513e350de2e961eaf1a54197eb19226ea39b564979f83a31b2 File: contracts/interfaces/ITermPoolFactory.sol SHA3: 7622cd4abe5325e2cd045e4fd469221130684fae89ee12b2713691079cd19c81 File: contracts/interfaces/ITpToken.sol SHA3: 5dd03ab78444f4e0b71bb9a576ca94f49c04ee7ea91ca2f8622689b52c0c16f9 File: contracts/utils/TermUtils.sol SHA3: 58f42c5fc6731c886869e8d8be96c3d2ae69591a8ed1694bd0eec8036be0d671

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

Third review scope

Repository	https://github.com/clearpool-finance/term-protocol
Commit	d0cbc8e4d0b6ee4e581701df85cd643ac8a5030d
Requirements	Public requirements are not provided
Contracts	File: contracts/TermPool.sol SHA3: 5fd17bbadbcbb137958053fb10eabb761509c9413c54dfc8bc28f7d3a4459a0f File: contracts/TermPoolFactory.sol SHA3: cfebe9af42cd92856bd35169188e314934261889a354096ddd230f301b97317c File: contracts/TpToken.sol SHA3: ada036535a315ec5e60edb47e1dd8aa8f9737efbddab718796004b1ed3fa0f83 File: contracts/interfaces/IClassicPool.sol SHA3: 9cdd13e7d0c9d7c2e31ce31c3bc49cee625a44aae016680378007d05a56ad5fb File: contracts/interfaces/IOwnable.sol SHA3: 43b07e927d69df0a8846563635b48a51570395f8d5745fcdcf545638d8d77af9 File: contracts/interfaces/IPermissionlessPoolFactory.sol SHA3: 26805f76eabd7488a9a32bc4d90b4fde70c52c77df00bd075f5d2c9376437cde File: contracts/interfaces/IPermissionlessPoolMaster.sol SHA3: 5ec9b2352e8f9a611dbed7208cca838febf7084b25f3c34c6a2152e8a99cc08e File: contracts/interfaces/ITermPool.sol SHA3: df2efd35be2e10765e2bb17d3eae78690b103a87a4c77e86ef0ef3a1cecd29f8 File: contracts/interfaces/ITermPoolFactory.sol SHA3: e573395df4eaab96ad3b8e1f59f18c891c1baffb4801eaa08f46e10705f91b90 File: contracts/interfaces/ITpToken.sol SHA3: 5dd03ab78444f4e0b71bb9a576ca94f49c04ee7ea91ca2f8622689b52c0c16f9 File: contracts/utils/TermUtils.sol SHA3: af79c92b5ba77d178e0147ed0143246a02403dc42d5960ce3af34e13b52dd433