HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: GemDrop
Date: 13 Oct, 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 GemDrop
Approved By	Przemyslaw Swiatowiec | Lead Solidity SC Auditor at Hacken OÜ
Assistant Auditors	Kornel Światłowski | SC Auditor at Hacken OÜ Roman Tiutiun | SC Auditor at Hacken OÜ
Tags	EIP-712, DEX Premium Deposits
Platform	EVM
Language	Solidity
Methodology	Link
Website	https://gemdrop.xyz/
Changelog	10.10.2023 - Initial Review 13.10.2023 - Second Review

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

Table of contents

Introduction **4**
System Overview **4**
Executive Summary **5**
Risks **5**
Findings **7**
Critical 7
C01. Signature Replay Attack Can Led to Unlimited Deposits 7
High 8
Medium 8
Low 8
L01. Usage Of Built-in Transfer 8
L02. Missing ChainId Value Stored 8
Informational 9
I01. Floating Pragma 9
I02. Variables That Should Be Declared Constant 9
I03. Redundant Import Of IERC20 Interface 10
I04. Zero Valued Transactions 10
I05. Missing Event For Critical Value Updation 10
I06. Wrongly Generated Addresses In Fee Array Can Block Deposit Functionalities Or Lock Tokens 11
I07. Use Custom Errors Instead Of Error Strings To Save Gas 11
I08. Zero Address Backend Signer Can Disturb Deposit Flow 12
Disclaimers **14**
Appendix 1. Severity Definitions **15**
Risk Levels 15
Impact Levels 16
Likelihood Levels 16
Informational 16
Appendix 2. Scope **17**

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

Introduction

Hacken OÜ (Consultant) was contracted by GemDrop (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

GemDrop Depositor contract is designed to manage premium payments and emit notifications through the 'Deposit' event. The contract's primary objective is to facilitate payments for premium services, both in ERC-20 tokens and native tokens.

Privileged roles

The Depositor contract utilizes the Ownable library from OpenZeppelin to control access to critical functionalities. The contract owner has the following privileges:

- setBackendSigner() - set backend signer address that will be used to validate each signature.
- withdraw() - withdraw ERC20 tokens deposited in contract.
- withdrawNative() - withdraw native tokens deposited in contract.

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 requirements are detailed.
- Technical description is robust.

Code quality

The total Code Quality score is **9.5** out of **10**.

- Best practice violations. (L02)

Test coverage

Code coverage of the project is **89.13%** (branch coverage).

Security score

As a result of the audit, the code contains **1** low severity issue. The security score is **10** out of **10**.

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

Summary

According to the assessment, the Customer's smart contract has the following score: **9.9**. The system users should acknowledge all the risks summed up in the risks section of the report.

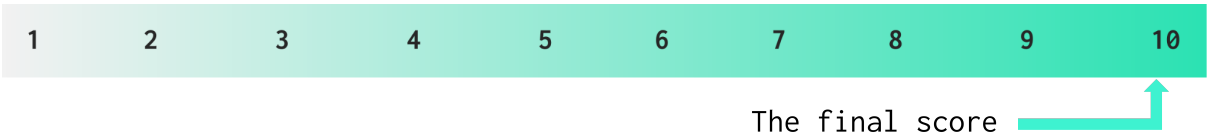

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
10 October 2023	2	0	0	1
13 October 2023	1	0	0	0

Risks

- Processes of signing messages and defining fees are handled by backend system, which is out of scope of this security assessment.

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

- Complete purchase flow depends on backend system, which is listening for deposit events and complete purchase. In case of backend system failure, user can not receive expected outcome even if deposit was successful.

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

Findings

■■■■ Critical

C01. Signature Replay Attack Can Led to Unlimited Deposits

Impact	High
Likelihood	High

The *deposit()* and *depositNative()* functions enable users to make payments in exchange for premium purchases. The backend system generates signatures to facilitate these payments, ensuring that users can execute valid transactions. Subsequently, upon the successful completion of a payment transaction, the smart contract emits a *Deposit* event for both the backend and the user, facilitating the delivery of the purchased items.

However, a significant security concern arises due to the absence of a *nonce* parameter in the signature and the failure to track it effectively. This oversight allows users to exploit the same signature multiple times within a specified deadline.

Steps to reproduce:

1. User gets a signature from the backend system.
2. User can deposit tokens with a dedicated function multiple times in the same block to omit the signature deadline.
3. Backend receives multiple Deposit events and gives user multiple goods.

This can lead to a situation where users can acquire premium goods without any restrictions, undermining the integrity of the entire ecosystem.

Path: ./contracts/Depositor.sol : deposit(), depositNative();

Recommendation: Contracts should track the Nonce to prevent signature replay attacks. By including this element in the signature verification process, contracts can ensure that each signature is only valid for a specific transaction and cannot be used again. If the backend generates a unique *paymentId* then this value can be used as a Nonce parameter instead.

Found in: 44cb86512058cdb4df5888589aa4f5a6afac6c3d

Status: Fixed (Revised commit: 4d99f19)

Remediation: The *paymentIdAllowed* modifier was introduced to verify that *paymentId* can be used only once, which acts as nonce parameter and prevent signature replay.

High

No high severity issues were found.

Medium

No medium severity issues were found.

Low

L01. Usage Of Built-in Transfer

Impact	Low
Likelihood	Low

The built-in *transfer()* and *send()* functions process a hard-coded amount of Gas. In case the receiver is a contract with receive or fallback function, the transfer may fail due to the "out of Gas" exception.

This can lead to denial of service situations for specific accounts.

Path: ./contracts/Depositor.sol: depositNative(), withdrawNative();

Recommendation: It is recommended to replace *transfer()* and *send()* functions with *call().*

Found in: 44cb86512058cdb4df5888589aa4f5a6afac6c3d

Status: Reported

Remediation: The *transfer()* was replaced with *call()* in *withdrawNative()*. However, in *depositNative()* fees are transferred using *transfer()*.

L02. Missing ChainId Value Stored

Impact	Medium
Likelihood	Low

The *Depositor* contract does not store the *chainId* value and does not verify whether the chainId has remained unchanged (*chainId* can change in case of fork).

Steps to reproduce:

1. Backend generates signature for user.
2. Chain for which signature is created is forked.
3. User can use signature on both forked chain.

This omission could potentially result in a scenario where a signature remains valid on both forked chains.

Path: ./contracts/Depositor.sol: _getDepositHash();

Recommendation: It is recommended to revise the current configuration to ensure that the *DOMAIN_HASH* is not defined as immutable. Instead, a mechanism should be implemented to verify the chainId each time the *DOMAIN_HASH* is utilized. If a change in chainId is detected, particularly in the event of a chain fork, it is advisable to generate a new *DOMAIN_HASH* to maintain the security and integrity of the system.

Found in: 44cb86512058cdb4df5888589aa4f5a6afac6c3d

Status: Fixed (Revised commit: 4d99f19)

Remediation: The *chainId* is verified on each deposit to match one defined during contract initialization (in the constructor). This is sufficient protection against signature replay in the case of chain fork.

Informational

I01. Floating Pragma

The project uses floating pragmas ^0.8.0.

This may result in the contracts being deployed using the wrong pragma version, which is different from the one they were tested with. For example, they might be deployed using an outdated pragma version, which may include bugs that affect the system negatively.

Path: ./contracts/Depositor.sol : *;

Recommendation: Consider locking the pragma version whenever possible and avoid using a floating pragma in the final deployment. Consider known bugs for the compiler version that is chosen.

Found in: 44cb86512058cdb4df5888589aa4f5a6afac6c3d

Status: Fixed (Revised commit: 4d99f19)

Remediation: Pragma was locked to 0.8.16.

I02. Variables That Should Be Declared Constant

State variables *DEPOSIT_FUNCTION_HASH* and *DEPOSIT_NATIVE_FUNCTION_HASH* do not change their value should be declared constant to save Gas.

Path: ./contracts/Depositor.sol: DEPOSIT_FUNCTION_HASH, DEPOSIT_NATIVE_FUNCTION_HASH;

Recommendation: Declare *DEPOSIT_FUNCTION_HASH* and *DEPOSIT_NATIVE_FUNCTION_HASH* variables as constants.

Found in: 44cb86512058cdb4df5888589aa4f5a6afac6c3d

Status: Fixed (Revised commit: 4d99f19)

Remediation: Parameters were declared as constants.

I03. Redundant Import Of IERC20 Interface

The *IERC20* interface import is redundant in the *Depositor* contract. *IERC20* interface is already imported within *SafeERC20* library.

This redundancy in import operations has the potential to result in unnecessary gas consumption during deployment and could potentially impact the overall code quality.

Path: ./contracts/Depositor.sol: *;

Recommendation: Remove redundant *IERC20* interface import.

Found in: 44cb86512058cdb4df5888589aa4f5a6afac6c3d

Status: Fixed (Revised commit: 4d99f19)

Remediation: *IERC20* interface import was removed.

I04. Zero Valued Transactions

The functions *deposit()* and *depositNative()* can execute a zero-valued transaction if *fee.amount* is 0.

This can lead to a transaction with zero value to be sent.

Path: ./contracts/Depositor.sol: deposit(), depositNative();

Recommendation: It is recommended to implement conditional checks for the zero-valued transaction.

Found in: 44cb86512058cdb4df5888589aa4f5a6afac6c3d

Status: Fixed (Revised commit: 4d99f19)

Remediation: Conditional checks for the zero-valued transaction were introduced.

I05. Missing Event For Critical Value Updation

The function *setBackendSigner()* does not emit events when important values change. This omission can lead to a significant drawback as users and external systems may be unable to subscribe to events to monitor and track important changes in the project (like modifying fees or whitelisting allowed currencies).

Path: ./contracts/Depositor.sol: setBackendSigner();

Recommendation: It is recommended to emit events on critical state changes.

Found in: 44cb86512058cdb4df5888589aa4f5a6afac6c3d

Status: Fixed (Revised commit: 4d99f19)

Remediation: Event emission for *setBackendSigner()* function was introduced.

I06. Wrongly Generated Addresses In Fee Array Can Block Deposit Functionalities Or Lock Tokens

The fee receiver addresses are generated by the backend system and are not subjected to validation within the smart contract. Incorrectly generated addresses can prevent users from depositing funds into the contract.

Examples of wrongly generated fee addresses for *deposit()*:

- One of the addresses is 0x0. This will result in a transaction revert because ERC20 transfers cannot be performed to the 0x0 address.
- One of the addresses is a Smart Contract that lacks the capability to withdraw transferred ERC20 tokens.

Examples of wrongly generated fee addresses for *depositNative()*:

- One of the addresses is 0x0. This will lock native tokens.
- One of the addresses is a Smart Contract that cannot receive native tokens.
- One of the addresses is a Smart Contract that lacks the capability to withdraw tokens.

Path: ./contracts/Depositor.sol: deposit(), depositNative();

Recommendation: It is recommended to ensure that addresses generated by the backend system do not result in a Denial of Service (DoS) on deposit functionalities or token locking. To resolve potential locking issues, 0x0 address check can be added to *depositNative()*.

Found in: 44cb86512058cdb4df5888589aa4f5a6afac6c3d

Status: Fixed (Revised commit: 4d99f19)

Remediation: Protection against zero address in fee receivers was introduced.

I07. Use Custom Errors Instead Of Error Strings To Save Gas

Custom errors were introduced in Solidity version 0.8.4, and they offer several advantages over traditional error-handling mechanisms:

1. Gas Efficiency: Custom errors can save approximately 50 Gas each time they are hit because they avoid the need to allocate and store revert strings. This efficiency can result in cost savings, especially when working with complex contracts and transactions.

2. Deployment Gas Savings: By not defining revert strings, deploying contracts becomes more gas-efficient. This can be particularly beneficial when deploying contracts to reduce deployment costs.
3. Versatility: Custom errors can be used both inside and outside of contracts, including interfaces and libraries. This flexibility allows for consistent error handling across different parts of the codebase, promoting code clarity and maintainability.

Path: ./contracts/Depositor.sol.

Recommendation: To save Gas, it is recommended to use custom errors instead of strings.

Found in: 44cb86512058cdb4df5888589aa4f5a6afac6c3d

Status: Fixed (Revised commit: 4d99f19)

Remediation: Custom errors are used instead of strings.

I08. Zero Address Backend Signer Can Disturb Deposit Flow

The backend signer plays a critical role in signing messages that grant users permission to deposit tokens. However, there are two potential issues:

1. **Initial Configuration**: After deploying the contract, the backend signer's address defaults to zero (0x0) since there is no address assignment in the constructor. It is crucial for the contract owner to remember to set the correct backend signer address to activate the intended business flow.
2. **setBackendSigner() Function**: It has been observed that the *setBackendSigner()* function can be used to inadvertently set the backend signer to zero address (0x0), which can disrupt the signing process.

In the event that the backend signer's address is set to zero, it renders the system incapable of signing and verifying messages, leading to users receiving an *Incorrect signature* error.

Path: ./contracts/Depositor.sol setBackendSigner();

Recommendation: It is recommended to assign backend signer address in the constructor and introduce zero address check in *setBackendSigner()* function.

Found in: 44cb86512058cdb4df5888589aa4f5a6afac6c3d

Status: Reported

Remediation: Protection against zero address assignment was introduced in the *setBackendSigner*. However, as *backendSigner* is still not assigned during contract initialization (in the constructor) - it is possible that contract could be in incorrect

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

state if owner forgets to set run *setBackendSigner* function after initialization.

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://gitlab.com/block-labs/gemdrop/smart-contracts/-/blob/develop/contracts/Depositor.sol
Commit	44cb86512058cdb4df5888589aa4f5a6afac6c3d
Whitepaper	Not provided
Requirements	NatSpec
Technical Requirements	NatSpec
Contracts	File: contracts/Depositor.sol SHA3: b9629c9b66a4c01e2a6eddf788129b564fd6568f98fef2228ade7578f4ff1137