HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: Paydece
Date: 02 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 Paydece
Approved By	Przemyslaw Swiatowiec | Lead Solidity SC Auditor at Hacken OÜ Kornel Światłowski | SC Auditor at Hacken OÜ Roman Tiutiun | SC Auditor at Hacken OÜ
Tags	Escrow
Platform	EVM
Language	Solidity
Methodology	Link
Website	https://www.paydece.io/
Changelog	05.10.2023 - Initial Review 02.11.2023 - 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 Paydece (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

PAYDECE - is an escrow protocol with the following contracts:

- *PaydeceEscrow.sol* — A smart contract decentralized escrow, that allows users to exchange goods or services using the ERC20 and native tokens they hold in a self-custodial wallet. Users connect to the Dapp with their wallet where they own their private keys (they have total control over their funds) and will block their funds in a smart contract that will hold the value until the transaction of the goods or service is completely done and validated by them.

Privileged roles

PaydeceEscrow is using the *Ownable* library from OpenZeppelin to restrict access for administrative functions. Contract owner have access for following functions:

- setFeeTaker() - configures the fee charged to the buyer (taker).
- setFeeMaker() - configure the fee charged to the seller (maker).
- setTimeProcess() - configure the time after which the cryptocurrency seller (maker) could cancel the transaction and be refunded the funds.
- releaseEscrowOwner() - release ERC20 tokens to the cryptocurrency buyer (taker) when a dispute arises.
- releaseEscrowOwnerNativeCoin() - release native tokens to the cryptocurrency buyer (taker) when a dispute arises.
- refundMaker() - refunding ERC20 tokens to the cryptocurrency seller (maker) when a dispute arises.
- refundMakerNativeCoin() - refunding native tokens to the cryptocurrency seller (maker) when a dispute arises.
- withdrawFees() - withdraw collected fees from different ERC20 tokens.
- withdrawFeesNativeCoin() - withdraw collected native token fees.
- addStablesAddresses() - add new tokens that are accepted by the contract.
- delStablesAddresses() - remove tokens accepted by the contract.
- CancelMakerOwner() - cancel the fund custody and refund it to the seller after available time passes.
- CancelTakerOwner() - cancel the fund custody and refund it to the seller before available time passes.

- setMarkAsPaidOwner() - confirm the fiat money transfer when issues arise and the buyer cannot perform it.

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 **9** out of **10**.

- Technical descriptions have some gaps.
    - No run instructions.

Code quality

The total Code Quality score is **7** out of **10**.

- Best practice violations. (I10, I14)
- Unused statuses. (I04)

Test coverage

Code coverage of the project is **90.71%** (branch coverage):

- Contracts are not tested thoroughly.
- Negative cases coverage missing.

Security score

As a result of the audit, the code contains **2** low severity issues. 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: **8.9**. The system users should acknowledge all the risks summed up in the risks section of the report.

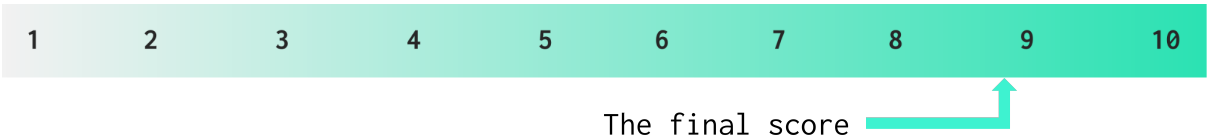

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
5 October 2023	5	1	4	1
02 November 2023	2	0	0	0

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

Risks

- The contract owner possesses significant control over the contract, which includes the ability to establish fees, disburse funds, issue refunds to buyers, and cancel escrow arrangements. This level of centralization could pose a risk in the event of the owner's account being compromised.
- The contract solely relies on the owner to define time locks for the escrow process. This arrangement could potentially result in funds becoming indefinitely locked within the contract if the owner neglects to define a time limit or if the owner's account is compromised. In the documentation, it is stated that the "timeProcess" variable should have a preset value of 45 minutes. However, it is important to note that, in reality, the owner has the flexibility to set any processing time they choose.
- The contract owner has the ability to whitelist any currency that may be employed in the escrow process. Users, particularly takers, must exercise caution and diligently confirm whether the correct currency has been specified in the escrow, including verifying the address associated with the escrow currency. This is crucial because if a malicious ERC20 token is whitelisted, takers could potentially suffer a loss of value.
- Escrowed funds can be released either by the maker or by the contract owner. Takers must place their trust in these entities, as in the current contract version, a process for raising disputes has not been implemented.

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

Findings

■■■■ Critical

C01. The maker fee is locked inside the contract when escrow is canceled or refunded

Impact	High
Likelihood	High

The Paydence charges fees from both the taker and the owner. Maker fees are collected at escrow creation, and taker fees are collected after funds are released.

It was identified that upon the creation of an escrow, fees are collected from the maker, However, these fees are not assigned to the *feesAvailable* mapping. Additionally, during the cancellation or refund processes (specifically, in the functions *CancelMaker(), CancelMakerOwner(), CancelTaker()*, and *CancelTakerOwner()*), only the escrow value is returned to the maker, without including the maker's fees. Consequently, this results in the fee maker amount becoming locked within the contract, inaccessible to the rightful owner.

```
function CancelMaker(uint256 _orderId) public nonReentrant
onlyMaker(_orderId){
    require( escrows[_orderId].status == EscrowStatus.CRYPTOS_IN_CUSTODY
, "El estado tiene que ser CRYPTOS_IN_CUSTODY" );

    uint256 _timeDiff = block.timestamp - escrows[_orderId].created;

    require(_timeDiff > timeProcess, "El tiempo todavia llego a su
termino" );

    escrows[_orderId].status = EscrowStatus.CANCEL_MAKER;
    //@audit fee is not returned or assign to be able to withdraw it
    escrows[_orderId].currency.safeTransfer(
        escrows[_orderId].maker,
        escrows[_orderId].value
    );
    emit EscrowCancelMaker(_orderId, escrows[_orderId]);
    }
```

Proof of Concept:

1. Owner sets *feeTaker* and *feeMaker* to non zero values.
2. New escrow is created for non-premium users. During an escrow creation *feeMaker* is collected from the maker - both escrow

value and *feeMaker* are transferred to the contract (funds escrow).
3. Maker, taker, or owner uses a dedicated function to cancel escrow.
4. Amount without maker fee is transferred to the maker address.
5. Fee amount (*feeMaker*) is permanently locked in the contract.

Path: ./contracts/PaydeceEscrow.sol: CancelMaker(), CancelMakerOwner(), CancelTaker(), CancelTakerOwner();

Recommendation: To prevent fee tokens lock, it's recommended either to update the fee balance during escrow creation or return the escrowed amount and corresponding fees to the maker during the cancellation process.

Found in: be0b8ce8b7a381dd89add981ba1dc32c6587ba0b

Status: Fixed (Revised commit: 93d6bf4)

Remediation: Process was introduced to return maker fee to the maker on refund and cancellation transactions.

■ ■ ■ High

H01. Missing functionality for cancellation of escrow holding native tokens

Impact	High
Likelihood	Medium

As documented, executing the *CancelMaker()* and *CancelMakerOwner()* methods is expected to result in a refund of cryptocurrencies to the seller. However, a critical issue has been identified. Attempting to cancel an escrow that holds native tokens (wei) does not work as intended - it leads to a transaction revert.

This issue arises because escrows denominated in wei store a zero address as the currency's address. When the cancel functions attempt to use this zero address as an ERC20 address and call transfer instructions on it, the transaction reverts.

In cases where the taker fails to use the *setMarkAsPaid()* function within the stipulated time, the native tokens held by the maker may become temporarily locked within the contract. Funds can be unlocked only by the owner using *refundMakerNativeCoin()* or *setMarkAsPaidOwner()*.

Proof of Concept:

1. Create an escrow involving native token.
2. Attempt to initiate the *CancelMaker()* function using the maker's account.

3. Observe the transaction revert.

Path: ./contracts/PaydeceEscrow.sol: CancelMaker(), CancelMakerOwner();

Recommendation: It is recommended to implement an alternative flow for cancelation escrows denominated in native tokens (wei).

Found in: be0b8ce8b7a381dd89add981ba1dc32c6587ba0b

Status: Fixed (Revised commit: 182aff2)

Remediation: Flow for cancelation escrows denominated in native tokens was introduced in *cancelMakerNative()* function.

H02. Users funds could be locked due to incorrect escrow status check

Impact	**High**
Likelihood	**Medium**

The PaydenceEscrow smart contract employs different statuses for escrows to manage their lifecycle. Initially, when created by the maker, an escrow is in the *CRYPTOS_IN_CUSTODY* status. Subsequently, the taker is required to mark the escrow as paid, which transitions its status to *FIATCOIN_TRANSFERED*. Importantly, only escrows with the *FIATCOIN_TRANSFERED* status are eligible for release.

There is a significant issue identified where user funds can become locked within the contract. This occurs because the *_releaseEscrowNativeCoin()* function checks for the *CRYPTOS_IN_CUSTODY* status rather than the *FIATCOIN_TRANSFERED* status. Consequently, if the taker marks the escrow as paid using *setMarkAsPaid()*, the funds become locked within the contract, and they cannot be released as *_releaseEscrowNativeCoin()* reverts due to the incorrect order status check.

```
require(
         escrows[_orderId].status == EscrowStatus.CRYPTOS_IN_CUSTODY,
            "USDT has not been deposited"
        );
```

Proof of Concept:

1. Create an escrow as the maker, which initially sets the status to *CRYPTOS_IN_CUSTODY*.
2. As the taker, initiate the *setMarkAsPaid()* function to mark the escrow as paid.
3. Attempt to release the escrow's funds by having the owner initiate the *_releaseEscrowNativeCoin()* function.

Path: ./contracts/PaydeceEscrow.sol: _releaseEscrowNativeCoin();

Recommendation: To mitigate the issue, it is recommended to verify order status during native tokens escrow release.

Found in: be0b8ce8b7a381dd89add981ba1dc32c6587ba0b

Status: Fixed (Revised commit: 182aff2)

Remediation: Check for correct order status was introduced in *_releaseEscrowNativeCoin()* function.

H03. Owner can drain users' funds from contract with refund mechanism

Impact	**High**
Likelihood	Medium

The contract owner has the authority to initiate refunds to escrow creators when disputes arise. However, there are significant issues with the current implementation:

1. **Lack of Dispute Resolution Mechanism**: The contract lacks a formal mechanism to raise and resolve disputes, including assigning a *REFUND* status to escrows in dispute.
2. **Missing Status Checks**: The *refundMaker()* and *refundMakerNativeCoin()* functions do not verify if an escrow is in dispute or the correct status for a refund, making it vulnerable to misuse.
3. **Escrow Status Not Updated**: After a refund, the escrow status remains unchanged, potentially leading to confusion regarding its state.

Proof of Concept:

1. As users create several escrows.
2. Create a new escrow as contract owner.
3. As the owner refunds the newly created escrow from point 2.
4. Escrow tokens are transferred to the owner, and statuses are not updated. Another refund of the same escrow is possible.
5. Point 3. and 4. can be done multiple times to drain all tokens from the contract.

```
function refundMaker(uint _orderId) external nonReentrant onlyOwner {
//require(escrows[_orderId].status == EscrowStatus.Refund,"Refund not
approved");
    uint256 _value = escrows[_orderId].value;
    address _maker = escrows[_orderId].maker;
    IERC20 _currency = escrows[_orderId].currency;
    _currency.safeTransfer(_maker, _value);
    emit EscrowDisputeResolved(_orderId);
}
```

Path: ./contracts/contract.sol: refundMaker(), refundMakerNativeCoin();

Recommendation: It is recommended to:

- verify order status in *refundMaker()* and *refundMakerNativeCoin()* function before proceeding with the refund process,
- update escrow (order) status after the refund,
- implement a procedure for raising disputes.

Found in: be0b8ce8b7a381dd89add981ba1dc32c6587ba0b

Status: Fixed (Revised commit: 93d6bf4)

Remediation: Correct order status check was introduced in refund functions. Only orders with statuses *FIATCOIN_TRANSFERED* and *CRYPTOS_IN_CUSTODY* could be refunded. Order status is correctly updated after refund operation.

Medium

M01. Lack of strict validation of msg.value can lead to funds lock

Impact	Medium
Likelihood	Medium

The *createEscrowNativeCoin()* function is intended to create an escrow with native tokens and receive them.

Due to the lack of precise validation of *msg.value*, there is a risk of locking funds inside the contract. If a user sends more native coins than required to cover ***_value + _amountFeeMaker***, the excess native tokens will become trapped within the contract without any means of withdrawal.

```
require((_value + _amountFeeMaker) <= msg.value, "Incorrect amount");
```

Path: ./contracts/PaydeceEscrow.sol: createEscrowNativeCoin();

Recommendation: It's recommended to verify that the user sent the exact amount of native tokens to escrow.

Found in: be0b8ce8b7a381dd89add981ba1dc32c6587ba0b

Status: Fixed (Revised commit: 182aff2)

Remediation: Conditional check for expected amount of wei (*msg.value*) was introduced in *createEscrowNativeCoin* function.

Low

L01. Zero-valued escrows

Impact	Low
Likelihood	Low

The functions *createEscrow()* and *createEscrowNativeCoin()* can execute a zero-valued transaction if zero is passed as a parameter. It is possible to create escrow with no amount, which could result in breaking the business process.

The following parameters are not checked for the zero value:

- PaydeceEscrow
  - *createEscrow()*
    - *uint256 _value*
  - *createEscrowNativeCoin()*
    - *uint256 _value*

Path: ./contracts/PaydeceEscrow.sol: createEscrow(), createEscrowNativeCoin()

Recommendation: It is recommended to implement conditional checks for the zero-valued transaction for *createEscrow(), createEscrowNativeCoin()* functions.

Found in: be0b8ce8b7a381dd89add981ba1dc32c6587ba0b

Status: Fixed (Revised commit: 182aff2)

Remediation: Conditional check for the zero-valued was introduced.

L02. Missing zero address check

Impact	Low
Likelihood	Low

There is possibility to set a zero address for the taker in creating the escrow process. Such flow could not be completed (only canceled), as ERC20 transfer function in escrow release checks if the recipient address is not 0.

The following parameters are not checked for the zero value:

- PaydeceEscrow
  - *createEscrow()*
    - *address payable _taker*

- *createEscrowNativeCoin()*
  - *address payable _taker*

Path: ./contracts/PaydeceEscrow.sol: createEscrow(), createEscrowNativeCoin()

Recommendation: It is recommended to implement zero address check for the taker address during escrow creation process.

Found in: be0b8ce8b7a381dd89add981ba1dc32c6587ba0b

Status: Fixed (Revised commit: 182aff2)

Remediation: Conditional check for the zero-address was introduced.

L03. Optimizing variable assignment in the constructor

Impact	Medium
Likelihood	Low

Within the *constructor()*, there is a redundant assignment of a value of **0** to both the *feeTaker* and *feeMaker* variables. This redundancy is unnecessary since, for variables of type uint256, the default value is inherently set to 0, making the explicit assignment superfluous.

Furthermore, an issue has been identified regarding the initialization of crucial contract variables. After deployment, these variables are not automatically initialized, requiring the owner to remember to set them manually in separate transactions. Failure to do so can result in a broken solution, with no fees and a lack of essential parameters such as *timeProcess*.

Path: ./contracts/PaydeceEscrow.sol: constructor()

Recommendation: To mitigate, it is recommended to set protocol parameters in the constructor

Found in: be0b8ce8b7a381dd89add981ba1dc32c6587ba0b

Status: Fixed (Revised commit: 182aff2)

Remediation: The *timeProcess* variable is set in the contract *constructor()*.

L04. Missing event for critical value updation

Impact	Low
Likelihood	Medium

The functions *setFeeTaker()*, *setFeeMaker()*, *setTimeProcess()*,

addStablesAddresses(), and *delStablesAddresses()* do 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/PaydeceEscrow.sol: setFeeTaker(), setFeeMaker(), setTimeProcess(), addStablesAddresses() and delStablesAddresses();

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

Found in: be0b8ce8b7a381dd89add981ba1dc32c6587ba0b

Status: Fixed (Revised commit: 182aff2)

Remediation: Events for *setFeeTaker()*, *setFeeMaker()*, *setTimeProcess()*, *addStablesAddresses()*, and *delStablesAddresses()* functions were introduced.

L05. Fee on transfer tokens can break contract logic

Impact	Low
Likelihood	Low

Contract is supposed to work with ERC20 tokens whitelisted by the contract owner. However, some ERC20s have a feature to collect a fee on transfer. On Ethereum, the USDT stablecoin has the fee feature implemented, which is disabled at the time of writing. But it may happen that they enable it, and this can mess up the contract accounting.

In the current implementation, fees on transfer tokens could break escrow flow. Consider the following scenario:

1. USDT authorities decided to activate the fee on the transfer feature and charge a 1% fee on token transfers.
2. Maker creates escrow for 100 USDT. But only 99 USDT were transferred to the escrow account (1 USDT fee on transfer was paid).
3. The taker paid for the escrow.
4. Funds should be released to the taker. However, the taker should receive 100 USDT, but the escrow account (contract) has only 99 USDT in balance. Release transaction reverts, and funds become locked in the contract.

Path: ./contracts/PaydeceEscrow.sol: *

Recommendation: It is recommended to identify the exact amount received by the receiver as a difference between the token balance before and after the transfer transaction is made.

Found in: be0b8ce8b7a381dd89add981ba1dc32c6587ba0b

Status: Reported

Remediation: The Paydence team has acknowledged the issue.

L06. Makers can omit fee payment

Impact	Medium
Likelihood	Low

There are significant concerns regarding the fee handling mechanism in the *createEscrow()* and *createEscrowNativeCoin()* functions.

The *_maker_premium* parameter, which specifies if a maker has a premium account, is being used to exempt makers from paying fees. However, this mechanism allows makers to omit fee payments even if they are not premium members. This means that makers can potentially avoid fees, regardless of their premium status.

Additionally, makers have the ability to manipulate the *_taker_premium* argument, which determines whether takers pay fees. This arrangement gives makers control over whether takers are required to pay fees, which may not align with the intended protocol design.

The decision of who should pay fees and who should be exempt should ideally be the prerogative of the protocol owners, rather than individual users. The current implementation grants excessive control to users (makers) over the fee structure, potentially leading to misuse or deviations from the intended protocol design.

Proof of Concept:

1. The owner configures the *feeTaker* and *feeMaker* parameters with non-zero values.
2. A non-premium maker creates a new escrow, set the *_maker_premium* parameter to **true** and the *_taker_premium* parameter to **false**.
3. A taker initiates the *setMarkAsPaid()* function.
4. The maker then triggers the *releaseEscrow()* function to finalize an order and update the associated fee.
5. Finally, the owner initiates the *withdrawFees()* function and observe that maker fee is not collected.

Path: ./contracts/PaydeceEscrow.sol: createEscrowNativeCoin();

Recommendation: It is recommended to redesign a smart contract, so protocol owners decide who is a premium member and who should pay fees.

Found in: be0b8ce8b7a381dd89add981ba1dc32c6587ba0b

Status: Reported

Remediation: The protocol owner, which is also a fee receiver accepts the risk of fee payment omission.

L07. 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 if a sender is a contract with a fallback function, the execution will fail.

Path: ./contracts/PaydeceEscrow.sol: cancelTakerNative();

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

Found in: 182aff27450ad5fc6d9f11e0bb931fec66143ac8

Status: Fixed (Revised commit: 93d6bf4)

Remediation: The *transfer()* and *send()* functions were replaced with *call().*

Informational

I01. Outdated Solidity version

Smart contract was compiled using **0.8.7,** whereas, at the time of creating this report, the newest Solidity versions are **0.8.19 / 0.8.20**.

Using an outdated compiler version can be problematic, especially if publicly disclosed bugs and issues affect the current compiler version. Using an old version for deployment prevents access to new Solidity security checks.

Path:
./contracts/Address.sol
./contracts/IERC20.sol
./contracts/Address.sol
./contracts/Ownable.sol
./contracts/PaydeceEscrow.sol
./contracts/ReentrancyGuard.sol
./contracts/SafeERC20.sol

Recommendation: It is recommended to deploy with any of the following Solidity versions: 0.8.19 or 0.8.20.

Found in: be0b8ce8b7a381dd89add981ba1dc32c6587ba0b

Status: Fixed (Revised commit: 182aff2)

Remediation: Pragma was locked to 0.8.19.

I02. Style guide violation

Contract readability and code quality are influenced significantly by adherence to established style guidelines. In Solidity programming, there exist certain norms for code arrangement and ordering. These guidelines help to maintain a consistent structure across different contracts, libraries, or interfaces, making it easier for developers and auditors to understand and interact with the code.

The suggested order of elements within each contract, library, or interface is as follows:

1. Type declarations
2. State variables
3. Events
4. Modifiers
5. Functions

Functions should be ordered and grouped by their visibility as follows:

1. Constructor
2. Receive function (if it exists)
3. Fallback function (if it exists)
4. External functions
5. Public functions
6. Internal functions
7. Private functions

Path: ./contracts/PaydeceEscrow.sol: *

Recommendation: Consistent adherence to the official Solidity style guide (https://docs.soliditylang.org/en/latest/style-guide.html) is recommended. This enhances readability and maintainability of the code, facilitating seamless interaction with the contracts.

Found in: be0b8ce8b7a381dd89add981ba1dc32c6587ba0b

Status: Fixed (Revised commit: 182aff2)

Remediation: Functions were ordered according to the official Solidity style guide recommendations.

I03. Functions that can be declared external

In order to make code easier to understand, public functions that are never called in the contract should be declared as external.

Path:./contracts/PaydeceEscrow.sol: getState(), addStablesAddresses(), delStablesAddresses(), CancelMaker(), CancelMakerOwner(), CancelTaker(), CancelTakerOwner(), setMarkAsPaid(), setMarkAsPaidOwner();

Recommendation: Use the external attribute for functions never called from the contract.

Found in: be0b8ce8b7a381dd89add981ba1dc32c6587ba0b

Status: Fixed (Revised commit: 182aff2)

Remediation: Functions were introduced as external.

I04. Unused escrowStatus enum values

The contract implements the Escrow statuses named *ACTIVE*, *DELETED*, *APPEALED*, and *RELEASE* but never uses them.

Unused functionality leads to increasing Gas needed for the deployment and decreases code quality.

Path: ./contracts/PaydeceEscrow.sol: EscrowStatus;

Recommendation: Verify if defined statuses are needed. Remove the redundant statuses from the contract codebase.

Found in: be0b8ce8b7a381dd89add981ba1dc32c6587ba0b

Status: Reported

Remediation: The status *ACTIVE* is redundantly defined but remains unused.

I05. Solidity style guide violation - naming conventions

According to the Solidity Style Guide Violation - Naming Conventions section, function names should follow mixedCase style.

CancelMaker(), *CancelMakerOwner()*, *CancelTaker()* and *CancelTakerOwner()* function names do not follow mixedCase style.

Path: ./contracts/PaydeceEscrow.sol: CancelMaker(), CancelMakerOwner(), CancelTaker(), CancelTakerOwner();

Recommendation: Follow Solidity Style Guide - Naming Conventions section. Change the mentioned function name to mixedCase style.

Found in: be0b8ce8b7a381dd89add981ba1dc32c6587ba0b

Status: Fixed (Revised commit: 158d4d5)

Remediation: Naming conventions were changed according to the Solidity Style Guide.

I06. State variables default visibility

Variable *whitelistedStablesAddresses* visibility is not specified. Specifying the state variable's visibility helps to catch incorrect assumptions about who can access the variable and increase code quality.

Path: ./contracts/PaydeceEscrow.sol: whitelistedStablesAddresses;

Recommendation: Specify variables as public, internal, or private. Explicitly define visibility for all state variables.

Found in: be0b8ce8b7a381dd89add981ba1dc32c6587ba0b

Status: Fixed (Revised commit: 182aff2)

Remediation: Visibility was added to the mapping.

I07. Variables can be downcasted to smaller size

Variables *feeTaker* and *feeMaker* can be downcasted from *uint256* to *uint16* type. Variables will hold only values in the range [0, 1000]. Changing their type from uint256 to uint16 will decrease the storage that the contract uses, which saves Gas.

What is more, variable *timeProcess* will hold time-related values and can be downcasted from *uint256* to *uint64* type to save memory slots.

Path: ./contracts/PaydeceEscrow.sol: whitelistedStablesAddresses;

Recommendation: Downcast mentioned variables to smaller uint type.

Found in: be0b8ce8b7a381dd89add981ba1dc32c6587ba0b

Status: Fixed (Revised commit: 182aff2)

Remediation: Variables were changed to smaller uint type.

I08. NatSpec and error messages are written in 2 languages

NatSpec comments and error messages are written in English and Spanish language.

This decreases code quality and contract readability.

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

Recommendation: Use either English or Spanish language in NatSpec comments and error messages for users. Add NatSpec comments to all of the contract functions.

Found in: be0b8ce8b7a381dd89add981ba1dc32c6587ba0b

Status: Fixed (Revised commit: 182aff2)

Remediation: The text was edited and unified into a single language.

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

I09. Unfinalized code

It is considered that the project should be consistent and contain no self-contradictions.

The implementation contains commented code:

- *onlyTakerOrOwner()* modifier;
- *uint256 private feesAvailable* variable;
- *require* check in *refundMaker()* and *refundMakerNativeCoin()*;
- commented line in *_releaseEscrow()*;

Commented code may decrease code readability.

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

Recommendation: It is recommended to remove the commented code or finalize its implementation.

Found in: be0b8ce8b7a381dd89add981ba1dc32c6587ba0b

Status: Fixed (Revised commit: 182aff2)

Remediation: Unfinalized code was removed from the contract.

I10. Code duplication

The contract contains code duplication, which refers to the presence of redundant or repeated code segments:

1. *CancelMaker()* and *CancelMakerOwner()* functions have the same code in their body. The only difference is the accessible modifier and event emitted. The body of these functions can be declared as a separate private function and reused in *CancelMaker()* and *CancelMakerOwner()*.
2. Fee calculation formula is defined in *createEscrow()*, *createEscrowNativeCoin()*, *_releaseEscrow()*, and *_releaseEscrowNativeCoin()*. Fee calculation can be a separate function that will be called every time a fee should be calculated.

Code duplication within the contract leads to increased deployment Gas costs and decreased code quality.

Path: ./contracts/PaydeceEscrow.sol: CancelMaker(), CancelMakerOwner(), CancelTaker(), CancelTakerOwner(), createEscrow(), createEscrowNativeCoin(), _releaseEscrow(), _releaseEscrowNativeCoin();

Recommendation: Refactor the duplicated code segments into reusable functions or employ appropriate design patterns to eliminate code duplication.

Found in: be0b8ce8b7a381dd89add981ba1dc32c6587ba0b

Status: Reported (Revised commit: 182aff2)

Remediation: One change was introduced to prevent code duplications - the second calculation formula has been incorporated into a dedicated function. However, other instances of code duplication were not refactored.

I11. Blacklisted addresses in USDC

The Paydence Escrow contract is going to use the USDC stablecoin, which includes a blacklist feature for both transfer sender and receiver, designed to prevent blacklisted users from utilizing the stablecoin.

If a blacklisted taker is defined in the escrow process, it becomes impossible to complete the escrow flow and collect maker and taker fees. The presence of a blacklisted user in the escrow process effectively blocks the completion of the transaction, including the gathering of fees.

Path: ./contracts/PaydeceEscrow.sol: *

Recommendation: It is recommended to implement a failover mechanism, which will transfer the taker's fund into the escrow account, in case of any issues with fund transfers, such as blacklisting of the user.

Found in: be0b8ce8b7a381dd89add981ba1dc32c6587ba0b

Status: Reported

Remediation: The Paydence team has acknowledged the issue.

I12. Redundant calculation

In `createEscrow()` and `createEscrowNativeCoin()` functions, fees for creating and escrow for the maker is calculated even when fees are not needed. Firstly, the fee is calculated, and later, the contract checks if the maker is premium (fee does not apply in this case) and assigns 0 value to fee variable (`_amountFeeMaker`).

In `_releaseEscrow()` and `_releaseEscrowNativeCoin()` functions, the fee for the maker and taker is calculated even when the fee is not needed. Firstly, the fee is calculated, and later, the contract checks if the maker or taker is premium (fee does not apply in this case) and assigns 0 value to the fee variable (`_amountFeeMaker` or `_amountFeeTaker`).

Path: ./contracts/PaydeceEscrow.sol: createEscrow(), createEscrowNativeCoin(), _releaseEscrow(), _releaseEscrowNativeCoin();

Recommendation: To save Gas, it is recommended to calculate fees only if this is needed (maker or taker is not premium).

Found in: be0b8ce8b7a381dd89add981ba1dc32c6587ba0b

Status: Fixed (Revised commit: 158d4d5)

Remediation: Redundant calculations were removed.

I13. Redundant check

Redundant conditional checks within Solidity smart contracts refer to situations where multiple conditional statements exist to validate the same condition, resulting in unnecessary complexity and potential confusion. These redundant checks often arise when two or more distinct sections of code attempt to verify a specific condition, although one of the checks already encompasses the other(s).

Inside the *createEscrow()* function there is a defined check if the *PaydeceEscrow* contract address is approved to transfer funds on behalf of *msg.sender* to *PaydeceEscrow* contract.

```
uint256 _allowance = _currency.allowance(msg.sender,
address(this));
require(
    _allowance >= (_value + _amountFeeMaker),
    "Taker approve to Escrow first"
);
```

The same check is also done in *_spendAllowance()* function in ERC20 contract (*transferFrom()* executes *_spendAllowance()*).

Path: ./contracts/PaydeceEscrow.sol: createEscrow();

Recommendation: To save Gas, it is recommended to remove redundant checks that duplicate the validation efforts.

Found in: be0b8ce8b7a381dd89add981ba1dc32c6587ba0b

Status: Fixed (Revised commit: 158d4d5)

Remediation: Redundant checks were removed.

I14. 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/PaydeceEscrow.sol.

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

Found in: be0b8ce8b7a381dd89add981ba1dc32c6587ba0b

Status: Reported

Remediation: The Paydence team has acknowledged the issue.

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.

Appendix 2. Scope

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

Initial review scope

Repository	https://github.com/PayDece/paydece-contracts/tree/v4
Commit	be0b8ce8b7a381dd89add981ba1dc32c6587ba0b
Whitepaper	Whitepaper
Requirements	Requirements
Technical Requirements	Technical Requirements
Contracts	File: contracts/Address.sol SHA3: f01ba4387f59155ade5ada4e88f4c5d58df6bf5f41a124f4baecca15b82e6850 File: contracts/Context.sol SHA3: 57c1449d21d6fa219cd2e1292b670e933e6ecebc9f3f678aaf3b2a903dd3fde6 File: contracts/IERC20.sol SHA3: 3bebc954a1035342cc9c62a9852b0f44f9af744573831ad5e710fa36b0a682fa File: contracts/Ownable.sol SHA3: 4fbd9f12e04a233e893823cf6385b203e39094b459b645b2f687449e6992781d File: contracts/PaydeceEscrow.sol SHA3: b64841ebdebbd8b0d070c746cfaf3d507654061c891180c74d172f0feaceea8b File: contracts/ReentrancyGuard.sol SHA3: 911b917bf808d585aacddcadf65ac33b10a1344599cd5fe316af70d07fed5d35 File: contracts/SafeERC20.sol SHA3: b374986cd70dda653f1d161373e7b968918062ebbc56129fc868c55ce99f7c21

Second review scope

Repository	https://github.com/PayDece/paydece-contracts/commits/v4.2
Commit	182aff27450ad5fc6d9f11e0bb931fec66143ac8
Whitepaper	Whitepaper
Requirements	Requirements
Technical Requirements	Technical Requirements
Contracts	File: contracts/Address.sol SHA3: c049c947fa9577a0bc107c6fb23e626015beda3791d4767278e0a8580f23ab97 File: contracts/Context.sol SHA3: 07895fbd70e6468d8c5b6366952fe06131f47fbc4c7dd21cca5507d441e5103c

	File: contracts/IERC20.sol SHA3: 3529aeeeed1740573201a3f4aa9b72a2b259526aa3dfca3625f28b0ebda84cf6 File: contracts/Ownable.sol SHA3: 9279860cab1aba92b38d6d28612c3efd146222e5a1a601c59d2c76ac39b90d2d File: contracts/PaydeceEscrow.sol SHA3: 8065e56002ad6a31e73f49f967913e071e857db3e48934ed4b38f801b45ba164 File: contracts/ReentrancyGuard.sol SHA3: 3807698e6fc8e90891bcd92e3130bd3fc2c98dcbfca93248190fa84e3d358bca File: contracts/SafeERC20.sol SHA3: 007da6383eb2180113349376f883f5666bf82e25a8d96ba1b558c4c2edf3425b