ZEALYNX SECURITY

Web3 Security & Smart Contract Development

Paymatic

Security Assessment

May 8th, 2025

Prepared by Zealynx Security
contact@zealynx.io
Zealynx.io

Sergio	@Seecoalba
Bloqarl	@TheBlockChainer

Contents

1. About Zealynx

Zealynx Security, founded in January 2024 by Bloqarl and Sergio, specializes in smart contract audits, development, and security testing using Solidity and Rust. Our services include comprehensive smart contract audits, smart contract fuzzing, and formal verification test suites to enhance security reviews and ensure robust contract functionality. We are trusted by clients such as Cyfrin, Monadex, Lido, Inverter, Ribbon Protocol, and Paragon.

Our team believes in transparency and actively contributes to the community by creating educational content. This includes challenges for Foundry and Rust, security tips for Solidity developers, and fuzzing materials like Foundry and Echidna/Medusa. We also publish articles on topics such as preparing for an audit and battle testing smart contracts on our website Zealynx.io and our blogs.

Zealynx has achieved public recognition, including winning 1st prize in the Uniswap Hook Hackathon and a Top 5 position in the Beanstalk Audit public contest, rewarded with $8k. Our ongoing commitment is to provide value through our expertise and innovative security solutions for smart contracts.

2. Disclaimer

A security review of a smart contract can't guarantee there are no vulnerabilities. It's a limited effort in terms of time, resources, and expertise to find as many vulnerabilities as possible. We can not assure 100% security post-review or guarantee the discovery of any issues with your smart contracts.

3. Overview

3.1 Project Summary

A time-boxed independent security assessment of the Paymatic was done by Zealynx Security, with a focus on the security aspects of the application's implementation.

We performed the security assessment based on the agreed scope during 3 days, following our approach and methodology. Based on the scope and our performed activities, our security assessment revealed 2 Medium, 2 Low and 2 Informational security issues.

3.2 About Paymatic

Website : paymatic.xyz
Docs: crypto-paymatic

Paymatic is a trust-minimized escrow protocol for ERC-20 tokens that secures token transfers through delayed settlement and sender-controlled execution. It allows a sender to lock tokens in a smart contract, specifying the recipient and a timelock duration, which can be customized at the time of payment creation. After the timelock expires, the recipient can withdraw the funds—unless the sender cancels the payment beforehand.

The protocol addresses risks such as address poisoning and misdelivery by enforcing a verification window before settlement. All state changes emit on-chain events, supporting reliable tracking and off-chain integration.

3.2 About Codebase

The code under review is composed of single smart contract written in Solidity language and includes 189 nSLOC - normalized source lines of code.

4. Audit Methodology

Approach

During our security assessments, we uphold a rigorous approach to maintain high-quality standards. Our methodology encompasses thorough **Fuzz** and **Invariant Testing**, **Formal Verification**, and **meticulous manual security review.**

Throughout the smart contract audit process, we prioritize the following aspects to uphold excellence:

1. **Code Quality**: We diligently evaluate the overall quality of the code, aiming to identify any potential vulnerabilities or weaknesses.
2. **Best Practices**: Our assessments emphasize adherence to established best practices, ensuring that the smart contract follows industry-accepted guidelines and standards.
3. **Documentation and Comments**: We meticulously review code documentation and comments to ensure they accurately reflect the underlying logic and expected behavior of the contract.

5. Severity Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

6. Executive Summary

Over a 3-day engagement, the Zealynx Security team, including both senior auditors and interns, conducted a focused audit of the PaymaticPayments.sol contract developed by the Paymatic team.

The audit aimed to identify security risks, design flaws, and implementation issues that could compromise the safety or reliability of the protocol's payment flow.

A total of 6 issues were identified, categorized as follows:

• 2 Medium severity
• 2 Low severity
• 2 Informational

The key findings include:

• Fee miscalculations caused by assumptions about uniform token decimal precision, potentially leading to disproportionate charges across different ERC-20 tokens.

• Incompatibility with fee-on-transfer tokens, which may result in incorrect accounting and the potential loss of user funds due to failed settlements or locked balances.

• Lack of guard clauses for zero-value fee transfers, which can cause transaction reverts on certain token implementations.

Overall, the contract demonstrates solid structure, but the audit highlighted the importance of accounting for ERC-20 variations and edge-case conditions to ensure reliability across different token types.

Summary of Findings :

Vulnerability	Severity
[M-01] Decimal Precision Issues in Fee Calculation	Medium
[M-02] Fee-on-Transfer tokens lead to locked funds	Medium
[L-01] Zero-Value Fee Transfers Can Revert	Low
[L-02] The Created state alone does not indicate whether a payment is unlocked	Low
[I-01] Misleading Documentation on `Pausable` Feature	Informational
[I-02] Unbounded `feeValue` May Undermine User Trust	Informational

7.1 Medium Severity Findings

[M-1] Decimal Precision Issues in Fee Calculation

Description

The protocol's fee calculation mechanism fails to account for varying decimal places across different ERC20 tokens, resulting in inconsistent fee implementation.
The core issue exists in the _processFee function:

```
function _createERC20Payment(...) internal {
    (uint256 feeAmount, uint256 paymentAmount) = _processFee(amount);
    // feeAmount could be 0 for very small payments
    SafeERC20.safeTransfer(token, feeRecipient, feeAmount);
}
```

This function calculates fees using a fixed divisor (100,000) without normalizing for token decimal precision, causing the effective fee rate to vary dramatically between tokens with different decimal standards.

Consider two common tokens with different decimal representations:

USDC (6 decimals): 1000 USDC = 1000_000000
WETH (18 decimals): 1000 WETH = 1000_000000000000000000

When processing a fee of 0.3% (where feeValue = 300):

```
// USDC (6 decimals)
feeAmount = 1000_000000 * 300 / 100000 = 3000 (0.003 USDC)

// WETH (18 decimals)
feeAmount = 1000_000000000000000000 * 300 / 100000 = 3000_000000000000000 (3 WETH)
```

This calculation shows that the effective fee rate is 0.0003% for USDC but 0.3% for WETH when using the same nominal amount (1000 tokens).

Impact

- Tokens with fewer decimals pay much lower fees than intended or even none at all.
- Tokens with more decimals pay correct fees
- Protocol earns significantly less fees from low-decimal tokens
- Unfair fee distribution among users of different tokens
- Protocol economics are affected as users might prefer low-decimal tokens

Mitigation

1. Scale amounts to 18 decimals before fee calculation:

```
function _processFee(uint256 amount, address tokenAddress) internal returns (uint256 feeAmount, uint256 paymentAmount) {
+   uint8 decimals = ERC20(tokenAddress).decimals();
+   uint256 scaledAmount = amount * 10**(18 - decimals);
-   feeAmount = amount * feeValue / 100000;
+   uint256 scaledFee = scaledAmount * feeValue / 100000;
+   feeAmount = scaledFee / 10**(18 - decimals);
    paymentAmount = amount - feeAmount;
}
```

2. Store different fee values per token decimals:

```
+ mapping(uint8 => uint256) public decimalToFeeValue;

function setFeeValue(uint256 newFeeValue, uint8 decimals) external onlyOwner {
+   decimalToFeeValue[decimals] = newFeeValue;
    emit FeeValueChanged(newFeeValue, decimals);
}

function _processFee(uint256 amount, address tokenAddress) internal returns (uint256 feeAmount, uint256 paymentAmount) {
+   uint8 decimals = ERC20(tokenAddress).decimals();
+   feeAmount = amount * decimalToFeeValue[decimals] / 100000;
    paymentAmount = amount - feeAmount;
}
```

3. Enforce minimum fee amounts based on decimals:

```
function _processFee(uint256 amount, address tokenAddress) internal returns (uint256 feeAmount, uint256 paymentAmount) {
+   uint8 decimals = ERC20(tokenAddress).decimals();
    feeAmount = amount * feeValue / 100000;
+   uint256 minFee = 10**decimals / 1000; // Minimum 0.001 tokens
+   if (feeAmount < minFee) {
+       feeAmount = minFee;
+   }
    paymentAmount = amount - feeAmount;
}
```

Each approach has its tradeoffs:

1. Most accurate but higher gas costs
2. Flexible but requires more maintenance
3. Simplest but might overcharge small amounts

[M-2] Fee-on-Transfer Tokens Lead to Locked Funds

Description

The contract assumes that the amount of tokens transferred is equal to the amount specified in the transaction. However, fee-on-transfer tokens (like SafeMoon) deduct a fee from each transfer, resulting in the contract receiving less tokens than expected.

```
function _createERC20Payment(
    address to,
    address tokenAddress,
    uint256 amount,
    uint256 timelockPeriod
) internal {
    // ... checks ...

    (uint256 feeAmount, uint256 paymentAmount) = _processFee(amount);

    // Records full amount before transfer
    payments[idCounter].amount = paymentAmount;

    // Actual received amount may be less due to transfer fee
    SafeERC20.safeTransferFrom(token, msg.sender, address(this), amount);

    // Fee calculation based on pre-fee amount
    SafeERC20.safeTransfer(token, feeRecipient, feeAmount);
}
```

Impact

- Contract receives less tokens than recorded in payment.amount
- Settlement will fail due to insufficient balance
- Protocol fees are calculated on pre-fee amount, leading to incorrect fee collection
- Payments become unserviceable
- Users lose funds due to failed transactions

Mitigation

Check actual received amount:

```
function _createERC20Payment(...) internal {
    uint256 balanceBefore = token.balanceOf(address(this));
    SafeERC20.safeTransferFrom(token, msg.sender, address(this), amount);
+   uint256 actualReceived = token.balanceOf(address(this)) - balanceBefore;
+   require(actualReceived ⩾ amount - maxSlippage, "Fee-on-transfer not supported");
}
```

7.4 Low Severity Findings

[L-01] Zero-Value Fee Transfers Can Revert

Description

Some ERC20 tokens (like LEND) revert on zero-value transfers. The contract doesn't check if the fee amount is zero before attempting the transfer:

```
function _createERC20Payment(...) internal {
    (uint256 feeAmount, uint256 paymentAmount) = _processFee(amount);
    // feeAmount could be 0 for very small payments
    SafeERC20.safeTransfer(token, feeRecipient, feeAmount);
}
```

Impact

- Payments with very small amounts where fee rounds to zero will revert
- Users cannot create small payments
- Wastes gas on failed transactions
- Poor user experience

Mitigation

The recommended approach is to add zero-check in the function before transfer:

```
function _createERC20Payment(...) internal {
    (uint256 feeAmount, uint256 paymentAmount) = _processFee(amount);
+   if (feeAmount > 0) {
        SafeERC20.safeTransfer(token, feeRecipient, feeAmount);
+   }
}
```

[L-02] The Created State Alone Does Not Indicate Whether a Payment is Unlocked

Description

Payments are created in the Created state with an unlockTime determining when they can be settled. However, the Created state alone does not indicate whether a payment is unlocked (i.e., unlockTime <= block.timestamp).

This ambiguity complicates querying payment status, as users must check unlockTime separately.
The settlePayment function enforces this via two modifiers:
- paymentCreatedStatus (ensures Created state)
- paymentUnlocked (verifies unlockTime).

Impact

- More complex user interfaces - UI needs to display both the payment status and unlock timing information
- Filtering/sorting challenges - Difficult to implement efficient filtering of payments that are "ready to settle" versus those still locked

Proof of Concept

```
function testUnlockedButNotSettledState() public {
        uint256 timelockPeriod = 1 days;

        vm.startPrank(user1);
        token.approve(address(payments), paymentAmount);
        payments.createERC20PaymentWithTimeLock(user2, address(token), paymentAmount, timelockPeriod);
        vm.stopPrank();

        PaymaticPayments.Payment memory payment = payments.getPaymentDetails(1);
        recordStateChange(1, payment.status);

        assertEq(uint256(payment.status), uint256(PaymaticPayments.PaymentStatus.Created));

        vm.startPrank(user2);
        vm.expectRevert(abi.encodeWithSelector(PaymaticPayments.PaymentTimelocked.selector));
        payments.settlePayment(1);
        vm.stopPrank();

        vm.warp(block.timestamp + timelockPeriod + 1);

        payment = payments.getPaymentDetails(1);
        assertEq(uint256(payment.status), uint256(PaymaticPayments.PaymentStatus.Created));

        assertTrue(payment.unlockTime < block.timestamp);
```

```
        vm.startPrank(user2);
        payments.settlePayment(1);
        vm.stopPrank();

        payment = payments.getPaymentDetails(1);
        recordStateChange(1, payment.status);

        // Verify state changed to Settled
        assertEq(uint256(payment.status), uint256(PaymaticPayments.PaymentStatus.Settled));

        // Verify state history
        assertTrue(validateStateHistory(1));

        // Verify transition was directly from Created to Settled,
        // without an intermediate state indicating it was unlocked
        assertEq(paymentStateHistory[1].length, 2);
        assertEq(uint256(paymentStateHistory[1][0]), uint256(PaymaticPayments.PaymentStatus.Created));
        assertEq(uint256(paymentStateHistory[1][1]), uint256(PaymaticPayments.PaymentStatus.Settled));
    }
```

Mitigation

Add a view function to check if a payment is unlocked, improving usability without altering the contract's state structure.

```
    function getPaymentLockStatus(uint256 id) external view returns (bool) {
        if(payments[id].status == PaymentStatus.Created && payments[id].unlockTime > block.timestamp){
            return true;
        }
        return false;
    }
```

7.3 Informational Findings

[I-01] Misleading Documentation on Pausable Feature

Description

The documentation claims that:

```
Pausable Owner can pause/unpause all state-changing actions.
```

However, in the actual implementation, the `whenNotPaused` modifier is applied only to the createERC20Payment() and createERC20PaymentWithTimelock() functions.

Other state-changing functions such as cancelPayment() and settlePayment() are not protected by this modifier, creating a discrepancy between the documented and actual behavior of the contract.

Users and integrators may assume that all state-changing functions are disabled when the contract is paused, potentially leading to unexpected behaviors or incorrect assumptions about the system's security posture during emergency pauses.

Suggestion

Either update the documentation to accurately reflect which functions are protected by the Pausable mechanism, or apply the `whenNotPaused` modifier consistently to all state-changing functions such as cancelPayment() and settlePayment()

[I-02] Unbounded feeValue May Undermine User Trust

Description

The setFeeValue (uint256 newFeeValue) function allows the contract owner to set any fee amount without restriction. There is no upper limit enforced in the contract logic.

While technically not a vulnerability, the lack of a maximum fee limit may raise concerns among users and integrators. Without clear boundaries, users have no assurance that fees will remain fair over time.

This uncertainty can erode trust in the protocol, especially in permissioned systems where owner actions directly affect user experience.

Suggestion

To build confidence and ensure predictability, consider introducing a sensible maximum cap on fees (e.g., 5%).
Alternatively, explicitly document that fees are fully discretionary and controlled by the owner, so users can make informed decisions before interacting with the protocol.