UniVoucher Security Review

October 9th, 2025

Contents

1 About SafetyBytes **3**

2 Disclaimer **3**

3 Introduction **3**

4 About UniVoucher **4**

5 Security Assessment Summary **4**

6 Risk Classification **4**

7 Executive Summary **6**

8 Findings **8**

8.1 Medium Findings 8

8.1.1 [M-01] Fee-on-Transfer Token Incompatibility 8

8.1.2 [M-02] Rebase Token Incompatibility . . . 12

8.1.3 [M-03] Partner Parameter Front-Running Vulnerability 15

8.2 Low Findings 17

8.2.1 [L-01] Missing Reentrancy Protection on Deposit Functions 17

8.3 Informational Findings 19

8.3.1 [I-01] Unused Code in _uintToString Function 19

1. About SafetyBytes

SafetyBytes is an independent Web3 security firm founded by two auditors, `yovchev_yoan` and `mladenov`, focused on smart contract security. We specialize in manual audits, aiming to uncover critical issues through careful review and a solid understanding of how each protocol works. We are on mission to secure every byte of your code.

Contact SafetyBytes for professional security audits at `@safetybytes` on X.

2. Disclaimer

A smart contract security review can only show the presence of vulnerabilities but not their absence. Audits are a time, resource, and expertise-bound effort where skilled technicians evaluate the codebase and their dependencies using various techniques to find as many flaws as possible and suggest security-related improvements. We as a company stand behind our brand and the level of service that is provided but also recommend subsequent security reviews, on-chain monitoring, and high whitehat incentivization.

3. Introduction

A time-boxed security review of the UniVoucher repository was done by SafetyBytes, with a focus on the security aspects of the application's smart contracts implementation.

4. About UniVoucher

UniVoucher is a decentralized gift card system that allows users to create crypto gift cards with ETH or ERC20 tokens that can be redeemed using cryptographic signatures from a card secret (private key). The protocol stores funds in address-based "slots" and generates unique card IDs, charging a 1 percent fee on creation and offering features like custom messages, partner referrals, and bulk operations. Creators can cancel their cards anytime to recover funds, or after 5 years the contract owner can reclaim abandoned cards.

5. Security Assessment Summary

review commit hash - `5e2ec2752829c74680a00f9af847708c74b25a2b`

Scope

The following smart contracts were in scope of the audit:

- `UniVoucher.sol`

6. Risk Classification

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

Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behaviour that's not so critical.

Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

Actions Required by Severity Level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.
- **Informational** - client may consider the recommendation.

7. Executive Summary

Protocol Summary

Protocol Name	UniVoucher
Repository	Private
Date	October 9 - October 10, 2025
Protocol Type	Payments

Findings Count

Severity	Amount
Medium	3
Low	1
Informational	1
Total Findings	5

Summary of Findings

ID	Title	Severity	Status
[M-01]	Fee-on-Transfer Token Incompatibility	Medium	Acknowledged
[M-02]	Rebase Token Incompatibility	Medium	Acknowledged
[M-03]	Partner Parameter Front-Running Vulnerability	Medium	Acknowledged
[L-01]	Missing Reentrancy Protection on Deposit Functions	Low	Acknowledged
[I-01]	Unused Code in uintToString Function	Informational	Acknowledged

8. Findings

8.1 Medium Findings

[M-01] Fee-on-Transfer Token Incompatibility

Severity: Medium
Likelihood: Medium

Description:

The **depositERC20** and **bulkDepositERC20** functions assume that the exact amount of tokens specified will be received by the contract. However, some ERC20 tokens deduct a fee during the transfer operation. This causes a mismatch between the amount stored in the contract state and the actual balance held by the contract. As stated, the protocol supports all kinds of ERC20 tokens.

When a user deposits a fee-on-transfer token, the contract follows this sequence:

1. Transfers `amount + protocolFee` tokens from the user
2. The token deducts its own transfer fee
3. Contract receives less than `amount + protocolFee`
4. Contract stores `amount` as the card value
5. Contract actually holds less than `amount`

When the card is later redeemed, the contract attempts to transfer the stored `amount` but does not have sufficient balance, causing the transaction to fail. The card becomes unredeemable and funds are stuck.

Example Scenario:

Deposit Phase:

1. User deposits 100 TokenX tokens (2% fee on transfer)
2. Contract expects: 101 tokens (100 + 1 protocol fee)
3. TokenX deducts: 2.02 tokens as its transfer fee
4. Contract receives: 98.98 tokens
5. Contract stores: 100 tokens

Redemption Phase:

1. Contract tries to send: 100 tokens
2. Contract has: 98.98 tokens
3. Result: Transaction fails, card cannot be redeemed

Code Location:

```
function depositERC20(address slotId, address tokenAddress,
   uint256 amount, ...) external {
    uint256 totalRequired = amount + fee;
    IERC20(tokenAddress).safeTransferFrom(msg.sender, address
       (this), totalRequired);
    // No check if full amount was actually received

    _createCard(slotId, tokenAddress, amount, fee, ...);
    // Stores 'amount' but contract may have less
}
```

Recommendations:

Option 1: Balance Check with Rejection (Recommended)

Check the contract balance before and after the transfer to verify the exact amount received. Reject fee-on-transfer tokens:

```
function depositERC20(address slotId, address tokenAddress,
   uint256 amount, ...) external {
    require(tokenAddress != address(0), "Use depositETH for
       native currency");
    require(amount > 0, "Cannot deposit 0 tokens");

    uint256 fee = calculateFee(amount);
    uint256 totalRequired = amount + fee;

    // Check balance before transfer
    uint256 balanceBefore = IERC20(tokenAddress).balanceOf(
       address(this));

    // Transfer tokens
    IERC20(tokenAddress).safeTransferFrom(msg.sender, address
       (this), totalRequired);

    // Check balance after transfer
    uint256 balanceAfter = IERC20(tokenAddress).balanceOf(
       address(this));
    uint256 actualReceived = balanceAfter - balanceBefore;

    // Verify full amount was received
    require(actualReceived == totalRequired, "Fee-on-transfer
        tokens not supported");

    accumulatedFees[tokenAddress] += fee;
    _createCard(slotId, tokenAddress, amount, fee, message,
       encryptedPrivateKey);
}
```

Apply the same fix to `bulkDepositERC20` *function.*

Option 2: Token Blacklist

Maintain a list of known fee-on-transfer tokens and block them:

```
mapping(address => bool) public blacklistedTokens;

function depositERC20(address tokenAddress, ...) external {
    require(!blacklistedTokens[tokenAddress], "Token not
       supported");
    // Rest of function
}

function setBlacklistedToken(address token, bool blacklisted)
    external onlyOwner {
    blacklistedTokens[token] = blacklisted;
}
```

This approach requires maintaining an updated list of problematic tokens and is less robust than Option 1.

[M-02] Rebase Token Incompatibility

Severity: Medium
Likelihood: Medium

Description:

The contract stores a fixed token amount for each card but does not account for tokens that automatically adjust holder balances over time (rebase tokens). Rebase tokens such as AMPL and stETH automatically increase or decrease all holder balances based on protocol events. This creates a mismatch between the stored amount and actual balance.

Negative Rebase Scenario:

1. User deposits 100 AMPL tokens
2. Contract stores `tokenAmount = 100`
3. AMPL executes a -10% rebase (supply contraction)
4. Contract balance automatically becomes 90 AMPL
5. Contract still stores `tokenAmount = 100`
6. User attempts redemption
7. Contract tries to transfer 100 AMPL but only has 90 AMPL
8. Transaction fails, card cannot be redeemed

Positive Rebase Scenario:

1. User deposits 100 AMPL tokens

2. Contract stores `tokenAmount = 100`
3. AMPL executes a +10% rebase (supply expansion)
4. Contract balance automatically becomes 110 AMPL
5. Contract still stores `tokenAmount = 100`
6. User redeems and receives 100 AMPL
7. Extra 10 AMPL remains stuck in contract
8. Funds are not tracked in `accumulatedFees` and cannot be withdrawn

Multiple Cards Amplify the Problem:
With negative rebases, multiple cards create a race condition where the last redeemer loses:

- Three cards of 100 AMPL each created (contract has 300 AMPL)
- -10% rebase occurs (contract now has 270 AMPL)
- First redemption: 100 AMPL sent (170 AMPL remaining)
- Second redemption: 100 AMPL sent (70 AMPL remaining)
- Third redemption: Tries to send 100 AMPL but only 70 available - FAILS

Code Location:

```
function depositERC20(...) {
    IERC20(tokenAddress).safeTransferFrom(msg.sender, address
       (this), totalRequired);
    _createCard(slotId, tokenAddress, amount, fee, ...);
    // Stores fixed amount, no tracking of balance changes
}

function redeemCard(...) {
    IERC20(slotCopy.tokenAddress).safeTransfer(to, slotCopy.
       tokenAmount);
    // Attempts to send stored amount, which may not match
       actual balance
}
```

Recommendations:

Maintain a list of known rebase tokens and prevent their deposit:

```
mapping(address => bool) public blacklistedTokens;

function depositERC20(address tokenAddress, ...) external {
    require(!blacklistedTokens[tokenAddress], "Token not
       supported");
    // Rest of function
}

function setBlacklistedToken(address token, bool blacklisted)
    external onlyOwner {
    blacklistedTokens[token] = blacklisted;
    emit TokenBlacklistUpdated(token, blacklisted);
}

event TokenBlacklistUpdated(address indexed token, bool
   blacklisted);
```

Implement this recommendation combined with clear documentation. This provides technical protection against known rebase tokens while being simple to implement and maintain. Update the blacklist as new rebase tokens emerge.

[M-03] Partner Parameter Front-Running Vulnerability

Severity: Medium
Likelihood: High

Description:

The **redeemCard** function does not include the **partner** parameter in the signature verification process. The signed message only contains the card ID and recipient address, but not the partner address.

When a user creates a signature to redeem a card without a partner, an attacker monitoring the mempool can extract this signature and front-run the transaction by submitting it with their own address as the partner parameter. The signature will still be valid because the contract only verifies that the signature matches the card ID and recipient address.

Attack Flow:

1. User signs a redemption message: `"Redeem card:[cardId]to:[recipientAddress]"`
2. User submits transaction with `partner = address(0)` (no partner, expects 100% of funds)
3. Attacker sees transaction in mempool
4. Attacker front-runs with higher gas, using the same signature but with `partner = attackerAddress`
5. Signature verification passes (partner not included in signed message)
6. Attacker receives 1% of card value
7. Recipient receives only 99% instead of expected 100

Impact:

- Users lose 1% of their card value to attackers
- Breaks the expected behavior of partner-less redemptions

- Creates persistent MEV opportunity for attackers

Code Location:

```
function redeemCard(string memory cardId, address payable to,
    bytes memory signature, address payable partner) {
    bytes32 messageHash = keccak256(abi.encodePacked("Redeem
       card:", cardId, "to:", to));
    // Partner parameter is NOT included in the message hash
}
```

Recommendations:

Include the `partner` parameter in the signed message to prevent front-running:

```
function redeemCard(string memory cardId, address payable to,
    bytes memory signature, address payable partner) {
    bytes32 messageHash = keccak256(abi.encodePacked(
        "Redeem card:",
        cardId,
        "to:",
        to,
        "partner:",  // Add this
        partner      // Add this
    ));

    bytes32 ethSignedMessageHash = keccak256(abi.encodePacked
       (
        "\x19Ethereum Signed Message:\n32",
        messageHash
    ));

    address signer = ethSignedMessageHash.recover(signature);
    require(signer == slotId, "Invalid signature");

    // Rest of function...
}
```

The frontend must also be updated to include the partner parameter when generating signatures. This ensures that each signature is only valid for the specific partner address (or `address(0)` for no partner) that the user explicitly approved.

8.2 Low Findings

[L-01] Missing Reentrancy Protection on Deposit Functions

Severity: Low
Impact: Low
Likelihood: Low

Description:
The deposit functions **depositETH**, **depositERC20**, **bulkDepositETH**, and **bulkDepositERC20** are missing the **nonReentrant** modifier.

Scenario 1: ETH Deposit Reentrancy
In **depositETH** and **bulkDepositETH**, excess ETH is refunded via an external call:

```
if (excess > 0) {
    (bool sent,) = payable(msg.sender).call{value: excess
       }("");
    require(sent, "Failed to return excess ETH");
}
```

A malicious contract can re-enter during this refund. However, state changes occur before the call, limiting exploit potential.

Scenario 2: ERC20 Token Reentrancy
In **depositERC20** and **bulkDepositERC20**, the external call happens BEFORE state changes:

```
IERC20(tokenAddress).safeTransferFrom(msg.sender, address(
   this), totalRequired);

// State updated AFTER external call
accumulatedFees[tokenAddress] += fee;
_createCard(...);
```

A malicious ERC20 token can re-enter **depositERC20** during **safeTransferFrom**, creating multiple cards while state is inconsistent. The attacker must still pay for each card created, preventing direct fund theft.

Recommendations:
Add `nonReentrant` modifier to:

- `depositETH`

- depositERC20
- bulkDepositETH
- bulkDepositERC20

8.3 Informational Findings

[I-01] Unused Code in _uintToString Function

Description:

The `_uintToString` function contains a check for zero that is never executed in the current implementation:

```
function _uintToString(uint256 _value) internal pure returns
   (string memory) {
    if (_value == 0) {
        return "0";  // This code is never reached
    }

    uint256 temp = _value;
    uint256 digits;
    // ... rest of function
}
```

This function is only called from `_generateCardId()`:

```
function _generateCardId() internal returns (string memory) {
    string memory cardId = string(abi.encodePacked(
        versionPrefix,
        chainPrefix,
        _uintToString(nextCardNumber)  // Called here
    ));
    nextCardNumber++;
    return cardId;
}
```

The `nextCardNumber` state variable is initialized to 1 and only incremented:

```
uint256 public nextCardNumber = 1;
```

Since `nextCardNumber` starts at 1 and is incremented after each card creation, it will never be 0. Therefore, the zero check in `_uintToString` is dead code that cannot be executed.

This does not affect contract security or functionality but represents unnecessary bytecode in the deployed contract.

Recommendations:

Remove the zero check since it will never be reached:

```
function _uintToString(uint256 _value) internal pure returns
   (string memory) {
    // Remove this:
    // if (_value == 0) {
    //     return "0";
    // }

    uint256 temp = _value;
    uint256 digits;

    while (temp != 0) {
        digits++;
        temp /= 10;
    }

    bytes memory buffer = new bytes(digits);
    while (_value != 0) {
        digits -= 1;
        buffer[digits] = bytes1(uint8(48 + uint256(_value %
           10)));
        _value /= 10;
    }

    return string(buffer);
}
```