SOLIDIFIED

Audit Report for Ivy Cash - May 27, 2021

DRAFT - NOT FOR PUBLICATION

Summary

Audit Report prepared by Solidified covering the Ivy Cash / Hawk smart contracts.

Process and Delivery

Three (3) independent Solidified experts performed an unbiased and isolated audit of the code. The debrief on 27 May 2021.

Audited Files

The source code has been supplied in the form of a GitHub repository:

https://github.com/Waly-Cash/hawk/tree/feature/audit

Commit number: `be554e3695fc098001037ca3e72bdc1ad0d65c60`

The scope of the audit was limited to the following files:

```
contracts
├── EscrowAdmin
│   ├── EscrowAdmin.sol
│   ├── FeeAdmin.sol
│   └── TreasuryAdmin.sol
├── EscrowManager
│   ├── Base.sol
│   ├── DepositManager.sol
│   ├── EscrowManager.sol
│   ├── FeeManager.sol
│   ├── Getters.sol
│   ├── Interfaces.sol
│   ├── ModifiersAndEvents.sol
│   └── Setters.sol
├── Paymaster.sol
├── RewardToken.sol
└── Treasury.sol
```

Intended Behavior

The smart contracts implement an Escrow solution supporting a number of ERC20 tokens. A reward system is also included.

Code Complexity and Test Coverage

Smart contract audits are an important step to improve the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of a smart contract system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**.

Note, that high complexity or lower test coverage does equate to a higher risk. Certain bugs are more easily detected in unit testing than a security audit and vice versa. It is, therefore, more likely that undetected issues remain if the test coverage is low or non-existent.

Criteria	Status	Comment
Code complexity	Medium	-
Code readability and clarity	Medium	-
Level of Documentation	Medium	-
Test Coverage	High	-

Issues Found

Solidified found that the Ivy Cash contracts contain 1 critical issue, 4 major issues, 7 minor issues, in addition to 3 informational notes.
1 Warning aimed at end users has been noted.

We recommend all issues are amended, while the notes are up to the team’s discretion, as they refer to best practices.

Issue #	Description	Severity	Status
1	Treasure contract has unsustainable economic model can be exploited by contract owner	Warning	-
2	PayMaster.sol: Unfinished implementation	Critical	Pending
3	TreasuryAdmin.sol: Unbounded Array of tokens might cause block gas limit issues	Major	Pending
4	ERC-20 return values ignored throughout the codebase	Major	Pending
5	Treasury.sol: Team reward is always zero	Major	Pending
6	RewardToken.sol: Incorrect initialReserveMinted calculation in constructor when decimals <> 18	Major	Pending
7	Escrow solution is not suitable for ERC-777 tokens and vulnerable to malicious ERC-20 implementation due to reentrancy	Minor	Pending
8	Feemanager.sol: Missing fee calculation	Minor	Pending
9	TreasuryAdmin.sol: setUSDPrice() allows adding price of unaccepted tokens	Minor	Pending
10	Base.sol: _addERC20() allows adding tokens with zero balance	Minor	Pending
11	Base.sol: _addERC721() does not check for max tokens limit	Minor	Pending

12	Base.sol: _addERC721() allows tokens with duplicate id	Minor	Pending
13	Base.sol: Create escrow from escrow methods allow creating new escrows of unaccepted tokens	Minor	Pending
14	Left-over commented-out code and debug statements	Note	-
15	Treasury.sol: The contract can be arbitraged in case TreasuryToken's price moves and its price is not updated in TreasuryAdmin.sol.	Note	-
16	TreasuryAdmin.sol: possible gas/logic optimization	Note	-

Warnings

1. Treasure contract has unsustainable economic model can be exploited by contract owner

The Treasury reward system relies on continuous inflow of “stakes”, since rewards of earlier staker always rely on added funds.
This is further aggravated by the fact that the `TreasuryAdmin.sol` owner has extensive privileges. For instance, the owner of `TreasuryAdmin.sol` can add new Treasury Tokens and set any price for them. Thus, the owner could mint any amount of Reward Tokens and redeem them for other Treasury Tokens.
Similarly, the owner of `TreasuryAdmin.sol` can remove a Treasury Token from the approved tokens list. The users redeeming Reward Tokens will not receive their share of that removed TreasuryToken.

Recommendation
Consider revising the staking model, reducing admin rights and using an oracle for USD pricing.

Critical Issues

2. PayMaster.sol: Unfinished implementation

The implementation of this contract seems incomplete and does not provide the expected functionality. In particular, the function `swapTokensForEth()` does not return any ETH, it simply receives tokens in the current implementation.

Recommendation
Provide a complete implementation or remove unfinished contracts (if not required).

Major Issues

3. TreasuryAdmin.sol: Unbounded Array of tokens might cause block gas limit issues

The set of accepted tokens are stored in an array that can grow without bounds. The function `redeem()` in `Treasury.sol` iterates over this array. If this array grows too large, the function may hit the block gas limit, always reverting. This would lead to stuck funds.

Recommendation
Consider placing an upper bound on the number of tokens, similar to the one employed in the `EscrowManager`.

4. ERC-20 return values ignored throughout the codebase

Throughout the codebase, ERC-20 return values are ignored. This assumes that ERC-20 tokens always return on failure (for example when a transfer is not possible due to lack of approval). However, many implementations do not revert on failure but do instead return a boolean value to indicate success or failure.

Recommendation
Check the ERC-20 return value. OpenZeppelin's SafeERC20 library provides a convenient wrapper for this functionality:
https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/utils/SafeERC20.sol

5. Treasury.sol: Team reward is always zero

The team reward in function `directBuy()` will always be zero, since the local `teamReward` variable is never set before the transfer. This is due to the incorrect return value assignment in the following code segment:

```
uint teamReward;
uint callerReward;
(callerReward, callerReward) = escrowAdmin.calculateTeamReward(rewardMinted);
```

Recommendation
Correct the reward assignment.

6. RewardToken.sol: Incorrect initialReserveMinted calculation in constructor when decimals <> 18

The constructor calculates `initialReserveMinted` using the (default) decimals value of 18. The provided `_decimals` value is not taken into account yet.

Recommendation
Consider calling function `_setupDecimals(_decimals)` before calculating `initialReserveMinted`.

Minor Issues

7. Escrow solution is not suitable for ERC-777 tokens and vulnerable to malicious ERC-20 implementation due to reentrancy

The escrow system has many state changes after external calls to ERC-20 tokens. This makes the contracts vulnerable to potential reentrancy attacks. In the case of ERC-777 tokens, this could be achieved by hooks that allow users to inject code. In the case of ERC-20 tokens, the vulnerability might be exploited by a malicious token implementation.

Functions affected by this include:

```
EscrowManager.claimEscrow()
EscrowManager._withdrawAllTokens()
FeeManager._tranferFees()
EscrowManager.confirmEscrow()
EscrowManager.createEscrow()
EscrowManager.createEscrowFromEscrowERC20()
EscrowManager.createEscrowFromEscrowERC721()
DepositManager.depositERC20()
DepositManager.depositERC721()
```

`DepositManager.sendERC20()`
`DepositManager.sendERC721()`
`EscrowManager.withdrawERC20()`
`EscrowManager.withdrawERC721()`

NOTE: This issue is classified as minor, since its impact is mitigated by the fact that only treasury-supported tokens added by a privileged account can be used.

Recommendation
Consider using a reentrancy guard for the functions listed above.

8. Feemanager.sol: Missing fee calculation

When the `_amountInTreasuryUnits` is greater than the `maxTreasuryUnitsRequired + escrowAdmin.gasFeeInTreasuryUnits()`, the gas fee is not considered.

Recommendation
Consider adding the gas fee in all cases.

9. TreasuryAdmin.sol: setUSDPrice() allows adding price of unaccepted tokens

The variable `acceptedTokensPrice` expects to store the USD price of accepted tokens. Whereas the method `setUSDPrice` in the TreasuryAdmin contract allows adding the price of any token.

Recommendation
Consider validating if a token exists in the accepted tokens list before allowing the method to add or update the price. It is furthermore recommended to clear the price of the token when the token is removed from the accepted tokens list using the method `removeTreasuryToken`.

10. Base.sol: _addERC20() allows adding tokens with zero balance

The method `_addERC20` allows adding any token with zero balance. This also enables escrows to be created with zero balance tokens.

Furthermore, the method allows only upto `maxTokens - 1` instead of `maxTokens.`

Recommendation
It is recommended to check the token amount to be greater than zero while adding the token.

11. Base.sol: _addERC721() does not check for max tokens limit

The method `_addERC721` does not consider the max tokens limit while adding new ERC721 tokens. This allows adding as many tokens as the user wants irrespective of the limit specified.

Recommendation
Consider checking for max token limit while adding ERC721 tokens.

12. Base.sol: _addERC721() allows tokens with duplicate id

The method `_addERC721` in the Base contract allows adding duplicate token ids which results in incorrect total tokens.

Recommendation
Consider checking if the token id already exists to avoid incorrect total tokens.

13. Base.sol: Create escrow from escrow methods allow creating new escrows of unaccepted tokens

The methods `createEscrowFromEscrowERC20` and `createEscrowFromEscrowERC721` allows creating new escrows of unaccepted tokens if the token is removed from accepted tokens list after creating the parent escrow.

Recommendation
Consider checking for accepted tokens before allowing the user to create escrow from escrow.

Informational Notes

14. Left-over commented-out code and debug statements

The codebase contains a number of code segments that have been commented out. There are also several `console.log()` calls used for debugging.

Recommendation
Consider removing commented-out code and debug statements before release.

15. Treasury.sol: The contract can be arbitraged in case TreasuryToken's price moves and its price is not updated in TreasuryAdmin.sol.

Once the price of a single TreasuryToken falls, it will be profitable to get Reward Tokens by supplying that Treasury Token, and redeem the RewardTokens for other TreasuryTokens. While this might be profitable for the project team since it will generate fees, the arbitrage will be done at the expense of other Reward Token holders.

Recommendation
Consider using a single oracle.

16. TreasuryAdmin.sol: possible gas/logic optimization

There is potential gas optimization in the function `addTreasuryToken()`. `acceptedTokensPos[_token] = acceptedTokens.length` could be assigned before `acceptedTokens.push(_token)`, removing the need for add/sub operations in function `removeTreasuryToken()`.

Recommendation

Apply the optimization.

Disclaimer

Solidified audit is not a security warranty, investment advice, or an endorsement of Ivy Cash or its products. This audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

The individual audit reports are anonymized and combined during a debrief process, in order to provide an unbiased delivery and protect the auditors of Solidified platform from legal and financial liability.

Solidified Technologies Inc.