SOLIDIFIED

Audit Report for Immutable on May 27th, 2020.

Summary

Audit Report prepared by Solidified for Immutable covering the platform's randomness, payment and escrow smart contracts (and their associated components).

Process and Delivery

Three (3) independent Solidified experts performed an unbiased and isolated audit of the code below. The debrief took place on May 27th, 2020, and the final results are presented here.

Audited Files

Folders `escrow`, `pay` and `randomness` form the repository and commit listed in the next section.

Notes

The audit was initially performed on commit `bc3c474e239633e694026180ffb65b52462a20fb` of repository https://github.com/immutable/platform-contracts/tree/develop/contracts/platform/contracts, using Solidity compiler version `0.5.11`.

Intended Behavior

Randomness provides a random source using a commit reveal scheme and future block hashes.
Payment processor is used to process ETH and USD payments within the platform, and lastly Escrow implements functions to keep the assets for fixed periods of time (while waiting for payment confirmations in legacy credit card payment systems, for example).

SOLIDIFIED

Audit Report for Immutable on May 27th, 2020.

Issues Found

Critical

1. Escrow.sol: Escrow contract allows for stealing funds(1)

It's possible for an attacker to steal most of the assets present in the `Escrow.sol` contract. This happens because the function `callbackEscrow` allows anyone to perform arbitrary calls on behalf of the contract. This could be used to call any token address that is currently on the vault and ask for sending funds to the attacker. It's also possible to make Escrow call `approve` and then funds could be transferred later.

Recommendation:
Consider limiting the scope of arbitrary calls, or even restricting the allowed tokens on the escrow. Another possible solution is forwarding funds to a kind of vault contract and keep it separate from the escrow logic, and while it's not a bullet proof solution, it might downsize the potential damage.

Amended [02-06-2020]
The issue was fixed and is no longer present in commit `f4d6848f97b04cf61ed01250ef0272ddc0014fdd`.

2. Escrow.sol: Escrow contract allows for stealing funds(2)

One could call the escrow function with any address set as `_from`. This enables frontrunning, i.e:

A user approves the escrow contract on the token contract.
The attacker then frontruns the vault creation calling escrow, with the same parameters the user is passing `(assets, _from == userAddress)`.

On another note, some token implementations do not require an `approve` before performing a `transferFrom` if the function is called by the owner of the tokens (`from`).

If a token that allows `transferFrom` with no previous allowance when the msg.sender is the owner, like Openzeppelin's ERC721(or an ERC20 token with similar implementation), an

attacker can call the contract passing a list of assets in possession of the vault. These assets will then become part of his escrow and are withdrawable through release.

Recommendation:
Consider applying the lock as the first thing in the function. Also consider keeping escrowed values in different contracts (one per escrow or at least per user) as this greatly diminishes the possibility of one user being able to claim other's tokens. Lastly, consider adding a whitelist of trusted assets, as rogue ERC20 implementations have been previously used in attacks (Spankchain case).

Amended [02-06-2020]
The issue was fixed and is no longer present in commit `f4d6848f97b04cf61ed01250ef0272ddc0014fdd`.

Major

3. Escrow.sol: All range checks (referred to as batch transfers or checks in the codebase) do not check the last token in the range

---

When a range of tokens is passed to the functions below, the last item of the range (`highTokenId`) is not included in the batch actions performed:
- `_transferBatch()`: The last token will not be transfered, and will remain in the escrow contract.
- `_areAnyInBatchEscrowed()`: The last item will not be checked, the function could return false if only the last token in the range is escrowed.
- `_areAllInBatchEscrowed`: The function will not check the last token, so it can return true even though the last item of the list is not escrowed.

These are internal functions called by core functions of the contract: `callBackEscrow()` and `release()`;

Recommendation
Ensure all items in the range are iterated through in these functions.

Immutable's comment:
"This is intended - ranges/batches in our systems are bottom inclusive, top exclusive (think i = start; i < end; i++)."

4. Escrow.sol: Wrong Security assumptions

In the `callbackEscrow` function, there's the following comment:

```
// escrow the assets with this contract as the releaser
// trusted contract, no re-entrancy risk
```

This is not true because any call to an external contract should be considered unprotected, including for tokens. Any contract could adhere to the ERC20 interface but implement something completely different.

Specially here, for the same reasons as #2, `_existsAndEscrowed` calls arbitrary contracts, therefore it isn't safe to assume that it's not reentrant.

Finally, the current locks prefevent only calls within the same function, but reentrancies are still allowed from different functions. For example, the external call on `callbackEscrow` on `Escrow.sol` could reenter to the function `release` on the same contract.

Recommendation
Consider using the same mutex for all functions. Do not account ERC20 external calls as trusted (unless the tokens were deployed by Immutable, and there is a restriction on tokens accepted in vaults).

Function `Escrow` does not currently implement a mutex (though it does perform an external call to an ERC20 token). Consider implementing a mutex in it unless all tokens are trusted.

Amended [02-06-2020]
The issue was fixed and is no longer present in commit `f4d6848f97b04cf61ed01250ef0272ddc0014fdd`.

5. Beacon.sol: Callers could choose to let the commit expire

If a single caller is awaiting randomness results for a given block, he could check the results once the block is available, and wait for it to expire for unfavorable results (and then recommit it indefinitely).

Though the issue is alleviated by the fact that several parties might be waiting for the result, the exploit will still be possible in instances where a single party is expecting (the parties could deliberately choose blocks that do not yet have a commit in low usage hours, and hope that nobody else uses it).

Immutable's comment:
"This is known and intended - our model generally relies on Immutable to be making these callbacks on behalf of users"

Minor

6. PurchaseProcessor.sol: _getSigner is vulnerable to ecrecover malleability

---

There is a known vulnerability in ecrecover that allow for malleable signatures. The implementation of `_getSigner` does not account for that.

Recommendation
Implement a check in order to prevent malleable signatures from being used. Also consider using Openzeppelin's `ECDSA.sol`, that has the check implemented:

```
        // EIP-2 still allows signature malleability for ecrecover().
Remove this possibility and make the signature
        // unique. Appendix F in the Ethereum Yellow paper
(https://ethereum.github.io/yellowpaper/paper.pdf), defines
        // the valid range for s in (281): 0 < s < secp256k1n ÷ 2 + 1, and
for v in (282): v ∈ {27, 28}. Most
        // signatures from current libraries generate a unique signature
with an s-value in the lower half order.
        //
        // If your library generates malleable signatures, such as s-values
in the upper range, calculate a new s-value
        // with
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141 - s1 and
flip v from 27 to 28 or
        // vice versa. If your library also generates signatures with 0/1
for v instead 27/28, add 27 to v to accept
```

```
        // these malleable signatures as well.
        if (uint256(s) >
0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0) {
            revert("ECDSA: invalid signature 's' value");
        }

        if (v != 27 && v != 28) {
            revert("ECDSA: invalid signature 'v' value");
        }
```

Amended [02-06-2020]
The issue was fixed and is no longer present in commit `f4d6848f97b04cf61ed01250ef0272ddc0014fdd`.

7. Escrow.sol: Return values of ERC20 transfer and transferFrom ignored

The return values of `transferFrom` in `escrow()` and `transfer` in `release()` do not account for the return value. Some token implementations only return false for failed transactions and do not revert the transaction (BAT for example), in case this happens both the `escrow()` and the `release()` functions will succeed execution even with a failed token transfer.

Recommendation
Check return values of all token transfers.

Amended [02-06-2020]
The issue was fixed and is no longer present in commit `f4d6848f97b04cf61ed01250ef0272ddc0014fdd`.

8. PurchaseProcessor.sol: Signature nonce implementation vulnerable to transaction order dependence

---

Currently there are nonces as part of the signature as best practices dictate, but they are only verified for replays (if the nonce has already been used). The ordering of nonces is not required, and this would allow for a relayer to sort the user's transactions and therefore interfere with the outcome of them (i. e.: choosing which of the payments are processed first).

This also disables the user from burning unsent transactions (by sending another one with a higher nonce), and therefore unused signatures remain valid indefinitely.

Recommendation
Check sequence of nonces when verifying the signatures.

Immutable's comment:
"This is known and intended - we have deliberately chosen to prefer horizontal scalability here. The ordering of payments is not something that should be relied on by users."

9. Escrow.sol and CreditCardEscrow.sol: Function destroy makes tokens inaccessible, but they still remain in possession of the Escrow

---

Function `Destroy` does not actually destroy the tokens, and it still leaves all tokens in possession of the escrow contract. Though they are irrecoverable once `destroy` is called (because the vaults are deleted), this breaks an invariant of the escrow contract (Sum of tokens owned by the smart contract will be larger than the sum of all escrowed vaults).

This can also adversely affect the `totalSupply` of the token (accounting for tokens out of circulation).

Recommendation
Burn tokens on vault destruction, as this will update the `totalSupply` of the token and will keep the balance of the Escrow contract in line with what is escrowed in it.

Immutable's comment:
"This is known and intended - unfortunately, different tokens have different burn implementations, some prohibit burning etc. In fact, the core Gods Unchained contract prohibits burning until cards are tradable, and therefore the only available option is to leave them in the contract. However, this exacerbates the issue described in 1: if a malicious callback was used to setApprovalForAll (for example), then even supposedly 'deleted' assets would be able to be taken from escrow."

10. Beacon.sol: GetCurrentBlock could become uncallable

The forwarding chain could get so big that it isn't able to fit in a single block. This is only an issue if the calling contracts have hardcoded commitBlocks.

Recommendation
This unlikely to happen and maybe with small to no side effects, so the team should consider the trade-offs of fixing or leaving as is.

Immutable's comment:
"This is known and intended, we're comfortable with it."

11. Escrow.sol: Iterations over lists/batches of tokens could deplete the transaction available gas and cause it to fail

The multiple iterations over the same array could be larger than the block limit.

Recommendation
Consider implementing limits on the size of lists/batches to ensure the loops are always executable, and do not result in funds locked in/out the Escrow.

Amended [02-06-2020]
The issue was fixed and is no longer present in commit `f4d6848f97b04cf61ed01250ef0272ddc0014fdd`.

Notes

12. PurchaseProcessor.sol: _validateOrderPaymentMatch is currently not validating match of currencies between the order and the payment

Function `_validateOrderPaymentMatch` is not currently validating if order.currency equals `payment.currency`, it's instead matching it to `Currency.USDCents`:

```
        require(
            order.currency == Currency.USDCents,
            "IM:PurchaseProcessor: receipt currency must match"
        );
```

This currently has no adverse effects since the function is only called from `_payUsdPricedInUsd`, and is therefore being reported as a note.

Recommendation
Consider amending `_validateOrderPaymentMatch` to check if `order.currency` and `payment.currency` match, and also using it in the other currencies' payment functions.

Amended [02-06-2020]
The issue was fixed and is no longer present in commit `f4d6848f97b04cf61ed01250ef0272ddc0014fdd`.

13. Escrow.sol: Function _checkVault is not implemented

The function `_checkVault` is not implemented, and is not called anywhere in the in-scope contracts, it can be safely removed.

Amended [02-06-2020]
The issue was fixed and is no longer present in commit `f4d6848f97b04cf61ed01250ef0272ddc0014fdd`.

14. Beacon.sol: Unnecessary check for overflows

Line 39: `require(commitBlock >= block.number, "IM:Beacon: must not overflow");` is unnecessary because of use of safe math - Same in line 89.

Amended [02-06-2020]
The issue was fixed and is no longer present in commit `f4d6848f97b04cf61ed01250ef0272ddc0014fdd`.

SOLIDIFIED

Audit Report for Immutable on May 27th, 2020.

Closing Summary

Immutable's platform contracts contain eleven issues, with two of them being critical, and three of major severity, along with several areas of note.

We recommend all issues are amended, while the notes are up to Immutable's discretion, as they mainly refer to improving the operation of the smart contract and best practices.

Update [02-06-2020]
All issues were fixed and are no longer present in commit `f4d6848f97b04cf61ed01250ef0272ddc0014fdd`.

Disclaimer

Solidified audit is not a security warranty, investment advice, or an endorsement of the Immutable platform 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.