CoinFabrik

HEX Audit

December 2019

By CoinFabrik

Contents

Introduction

CoinFabrik was asked to audit the contracts for the HEX project. Firstly, we will provide a summary of our discoveries and secondly, we will show the details of our findings.

Summary

The contracts audited are from the HEX repository at https://gitlab.com/bitcoinhex/contract-staging. The audit is based on the commit b0a5f8e5596d4fbb9b2ced67a2e982e2b1a50ce7 and updated to reflect changes until commit 67bd0093c5796b659fd27a813c431773ecfc7fef.

The audited contracts are:

- GlobalsAndUtility.sol
- HEX.sol
- StakeableToken.sol
- TransformableToken.sol
- UTXOClaimValidation.sol
- UTXORedeemableToken.sol

We analyzed the code for the following errors:

- Misuse of the different call methods: call.value(), send() and transfer()
- Integer rounding errors, overflow, underflow and related usage of SafeMath functions
- Outdated version of Solidity compiler
- Race conditions such as reentrancy attacks or front running
- Misuse of block timestamps
- Contract softlocking attacks (DoS)
- Potential gas cost of functions
- Missing or misused function qualifiers
- Fallback functions above the gas limit
- Fraudulent or poorly-written code
- Needlessly complex code and contract interactions
- Poor or nonexistent error handling
- Failure to use a withdrawal pattern
- Insufficient analysis of function input requirements
- Incorrect use of digital signatures

Severity Classification

The security risk categories are evaluated according to the following classification:

- **Critical:** The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for client's reputation or serious financial implications for clients and users.
- **Medium:** The issue puts a subset of users' sensitive information at risk, would be detrimental for the client's reputation if exploited, or is reasonably likely to lead to moderate financial impact.
- **Minor:** The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low-impact in view of the client's business circumstances.
- **Enhancement:** The issue does not pose an immediate threat to continued operation or usage, but is relevant for security best practices, software engineering best practices, or defensive redundancy.

Detailed Findings

Critical severity

No issue has been found.

Medium severity

No issue has been found.

Minor severity

Replay signature with modified parameters

The function *claimBtcAddress* is used to claim a BTC address and balance and assign Hearts to a claimant Ethereum address. It uses an ECDSA signature to validate the balance in BTC and some parameters like claimant and referrer address.

Only part of the referrer address is included in the signed message, which enables attackers to reuse a single time that valid signature to credit the referrer tokens to themselves.

The signed message is created in the function _*createClaimMessage* in UTXOClaimValidation.sol. There are two types of messages. One includes the *claimParamHashStr*:

```
return abi.encodePacked(
    BITCOIN_SIG_PREFIX_LEN,
    BITCOIN_SIG_PREFIX_STR,
    uint8(prefixStr.length) + ETH_ADDRESS_HEX_LEN +
            1 + CLAIM_PARAM_HASH_HEX_LEN,
    prefixStr,
    addrStr,
    "_",
    claimParamHashStr
);
```

The variable claimParamHashStr is created from claimParamHash which is a 32 byte variable created in function *claimBtcAddress* at UTXORedeemableToken.sol

```
bytes32 claimParamHash = 0;

if (claimToAddr != msg.sender) {
    /* Claimer did not send this, so claim params must be signed */
    claimParamHash = keccak256(
        abi.encodePacked(MERKLE_TREE_ROOT,
                autoStakeDays, referrerAddr)
    );
}
```

The variable *claimParamHashStr* is a hexadecimal representation from *claimParamHash*.

```
bytes memory claimParamHashStr = new bytes(CLAIM_PARAM_HASH_HEX_LEN);

_copyAsHexString(claimParamHashStr,
        claimParamHash, CLAIM_PARAM_HASH_BYTE_LEN);
```

The function _*copyAsHexString* will copy the first CLAIM_PARAM_HASH_BYTE_LEN from *claimParamHash* to *claimParamHashStr* as hexadecimal digits. From GlobalsAndUtility.sol we have the definitions

```
    uint8 internal constant CLAIM_PARAM_HASH_BYTE_LEN = 8;
    uint8 internal constant CLAIM_PARAM_HASH_HEX_LEN =
            CLAIM_PARAM_HASH_BYTE_LEN * 2;
```

A prefix of the first 8 bytes will be copied in the signed message. So if we produce a different *claimParamHash* that matches the first 8 bytes we will have a valid signature.

There are two inputs used to calculate *claimParamHash* that are controlled by the user:

- *autoStakeDays* which is limited between 350 days and 18200 days (around 50 years)
- *referrerAddr* that has no restriction

After MERKLE_TREE_ROOT is determined, the attacker can precalculate the sets of prefixes that he can control. They can wait for someone to send a transaction with a known prefix and send the malicious transaction.

As result of this attack the attacker will gain 20% of the minted tokens as referrer and the claimant can have its funds frozen for more days than originally intended.

We propose these two possible solutions:

- Use at least 16 bytes for prefix. The function *_createClaimMessage()* is only called from *claimBtcAddress()* so the increase in costs will be minimal to investors.

- Require *claimToAddr* to be the same as *msg.sender*. If the feature that allows to send transactions on behalf of someone else is not part of the UX then you can remove them from the contract without loss of functionality.

Using a computer with 8 Tesla V100 we can generate 13,590 million keccak hashes per second. Using 377,049 instances gives us a probability of 0.33% of finding a collision in 12 seconds, the average time that takes to create a block in Ethereum.

We decided to re-classify the severity as minor since the ability to create this type of signatures is not part of the user interface and the probability of finding a collision is very low with available hardware.

HEX team is evaluating how to resolve this issue. They have two branches implementing the suggestions. Because the feature is not critical to the project they haven't decided yet if they want to cut the feature or leave the risk in place.

At commit 8640000cba2c8180a656fee53ba3d83e01970652 the prefix length was increased to 12 bytes.

Excessive gas for computation

The function *_calcPayoutRewards* calculates the payout iterating the daily shares for the duration of the stake.

```
for (uint256 day = beginDay; day < endDay; day++) {
    payout += dailyData[day].dayPayoutTotal * stakeSharesParam /
                  dailyData[day].dayStakeSharesTotal;
}
```

It can iterate up to MAX_STAKE_DAYS times. Currently MAX_STAKE_DAYS is defined as 18200 days (around 50 years).

This function is not accessible to users but internally it is called to close a stake from *endStake* or *goodAccounting*. Running out of gas will cause the user to be unable to close the stake. Fail to close a stake will cause the owner to lose funds on late penalties.

The following table shows a simulation of minimum cost to execute the function with both a contract compiled with optimizations and without optimizations.

Iterations / days	Gas Unoptimized	Gas Optimized
100	103,713	62,340
200	184,213	101,540
500	425,777	219,204
1,000	828,277	415,204
2,000	1,633,227	807,204
5,000	4,048,277	1,983,204
10,000	8,073,277	3,943,204
18,200	14,674,277	7,157,604
20,000	16,123,277	7,863,204

For the current MAX_STAKE_DAYS of 18,200 the unoptimized contract requires at least 14.6 millions gas at the current block gas limit is 8 millions it will fail to execute and the optimized contract requires above 7.1 million gas.

Since 7.1 million gas is too close to the current block gas limit of 8 millions we suggest to lower the limit MAX_STAKE_DAYS to a more reasonable value, for example 10 years (around 3640 days).

> ***HEX team is aware that it is possible that future changes to the Ethereum blockchain might affect the gas consumption of some operations. We understand that currently this is not an issue but we wanted to raise awareness so the Ethereum community reach an agreement that satisfies all the parties involved.***
>
> ***At commit cd59207e95b2c020b1bb7099f68fdf4d13ecb6e9 due to opcode re-pricing included in the Istanbul fork MAX_STAKE_DAYS was lowered to 5555 days (approximately 15 years).***

The fallback function is not used and marked as payable

The fallback function is marked as payable but it is not used to track funds or to generate events when a deposit is made.

If it doesn't have a purpose it is better to remove the payable modifier to prevent unintentional sending funds to the contract.

> ***HEX team states that this is not an issue because it will only happen when the contracts are used in an undocumented manner and it is not a normal use case.***

Enhancements

Function name unrelated to functionality

The function *goodAccounting* in StakeableToken.sol doesn't reflect the functionality provided. We suggest to use a more proper name.

```
/**
 * @dev PUBLIC FACING: Removes a completed stake from the global pool,
 * distributing the proceeds of any penalty immediately. The staker must
 * still call endStake() to retrieve their stake return (if any).
 * @param stakerAddr Address of staker
 * @param stakeIndex Index of stake within stake list
 * @param stakeIdParam The stake's id
 */
function goodAccounting(address stakerAddr,
        uint256 stakeIndex, uint40 stakeIdParam) external
```

This issue has been acknowledged by the HEX team and they claim that within a business context the name makes sense. The function goodAccounting was renamed to stakeGoodAccounting at commit 200fa7bbcea5225c2c9e42cd7bc9a6dc57f30a26.

The documentation for function parameters is insufficient

We found the documentation to be very good for most functions and it helps to understand the code. But there are a couple of functions in StakeableToken.sol that have insufficient documentation for some of the parameters used.

For example in function _calcPayoutAndEarlyPenalty the parameters *pooledDayParam*, *stakedDaysParam*, *stakeSharesParam* all have the same description:

```
/**
 * @dev Calculates served payout and early penalty for early unstake
 * @param g Cache of stored globals
 * @param pooledDayParam Param from stake
 * @param stakedDaysParam Param from stake
 * @param servedDays Number of days actually served
 * @param stakeSharesParam Param from stake
 * @return 1: Payout in Hearts; 2: Penalty in Hearts
 */
function _calcPayoutAndEarlyPenalty(
    GlobalsCache memory g,
    uint256 pooledDayParam,
    uint256 stakedDaysParam,
    uint256 servedDays,
    uint256 stakeSharesParam
)
```

This was discussed with the HEX team and we are in agreement that documentation for private functions should be removed. It was removed at commit 1af0028b1d0f5eb21c7e2787fb1d892bc1e50b05.

Magic constants in some functions

The function *getDailyDataRange* use some numeric constants without giving them a name to make it more clear of their functionality.

```
uint256 v1 = uint256(dailyData[src].dayPayoutTotal);
uint256 v2 = uint256(dailyData[src].dayStakeSharesTotal) << 80;
uint256 v3 = uint256(dailyData[src].dayUnclaimedSatoshisTotal) << 160;
```

Similarly the function *_calcDailyRound* uses some constant without giving them a name

```
rs._payoutTotal = rs._allocSupplyCached * 10000 / 100448995;
```

We recommend to add a constant to GlobalsAndUtility.sol to make it easier to understand what data is being manipulated.

For example in function *_encodeClaimsValues* the constant SATOSHI_UINT_SIZE is similar situation:

```
uint256 v = _claimedBtcAddrCount << (SATOSHI_UINT_SIZE * 2);
    v |= _claimedSatoshisTotal << SATOSHI_UINT_SIZE;
    v |= _unclaimedSatoshisTotal;
```

This was discussed with the HEX team and we agreed it's not critical for the readers to have perfect understanding beyond the comments.

The function getDailyDataRange was renamed to dailyDataRange and it was modified to use the named constant HEART_UINT_SIZE at commit 4c1d5f2e99b093e9c2a6696f7e706d21c0e679cb.

Document implicit initializations

In function *_storeDailyDataBefore* uses the structure *RoundState* in memory. The field *_batchMintOrigin* might be used later without it being explicitly initialized.

Currently initialization of structs in memory is a feature that is not documented in the language specification. The solidity compiler will implicitly initialize memory structs fields to zero.

```
    RoundState memory rs;
    rs._allocSupplyCached = totalSupply() + g._lockedHeartsTotal;
```

We suggest to annotate places where an implicit initialization is required.

> ***This issue has been acknowledged by the HEX team. We agree that Solidity documentation is ambiguous but the behavior is consistent with the spirit of the docs. So _batchMintOrigin has been removed on commit 640906556dd14b2b57902185557b36f8d4251806.***

Observations

- Quantities used in various formulas are stored in units like uint72, uint96 and converted to uint256 before making arithmetic operations and converted back to the original type as needed.
  It prevents intermediate results to cause overflow in the formulas used currently in the contract. But it is possible that slight modifications of the formulas might cause unintended intermediate overflows.
  We suggest to include tests to make sure the intermediate results are safe using uint256, or use a library like SafeMath to check the results during execution.

- The function *_btcAddressIsValid* is used to validate if a BTC address and an amount of satoshis are included in the snapshot of the Bitcoin chain. It works as intended but how a leaf node is constructed is not well documented.
  It constructs a leaf node that will be checked for inclusion in a merkle tree.

  ```
    bytes32 merkleLeaf = bytes32(btcAddr)|bytes32(rawSatoshis);
  ```

  The variable *btcAddr* is declared as bytes20 so it will be left aligned once converted to bytes32 and since *rawSatoshis* is declared as uint256 it will be right aligned. Since they are smaller than their limit, they do not clash when combined together. But it would be good to clarify this on the documentation.

The format used was documented when modifications were made to prevent merkle proof attacks at commits 384fe129daf59f8e32cd16b1f516871c0f3e2e5b and dbbecf931b1a8aad399af88d032783744b386f53.

- The contracts require to be compiled with optimizations enabled because some operations are too expensive to run in mainnet with them disabled. When optimizations are enabled the compiler transforms the compiled bytecode by performing actions to lower gas usage, for example it removes redundancies and optimize access to contract storage. Using the optimization expose the contract to bugs in the optimization module (like 2019/03/26 solidity optimizer bug), but since current design requires optimizations it is a risk that seems worth taking.

Conclusion

We found three minor problems. Two of them could have an impact in the future, but are not a problem right now. The last one requires a determined attacker with knowledge of how signatures and hash functions work. We have determined that it is hard to exploit in currently available hardware and also it requires extra steps from users since it is not available in the user interface.

The contracts in Solidity were well written and properly documented.

Disclaimer

This audit report is not a security warranty, investment advice, or an approval of the HEX project since CoinFabrik has not reviewed its platform. Moreover, it does not provide a smart contract code faultlessness guarantee.