Tornado Cash
Smart Contracts Audit.
Final Version

Mikhail Vladimirov and Dmitry Khovratovich

19th November 2019

This document is the audit of Tornado Cash smart contract set performed by ABDK Consulting.

1. Introduction

We've been asked to review the Tornado codebase in a public repo. Based on our findings, the authors made a number of changes, with the most recent commit being d0e31.

No critical (immediately exploitable with obvious security loss) **issues** were found during the review. All other significant issues were fixed. Some minor issues were left for the future releases or ignored.

2. ERC20Mixer.sol

2.1 Fixed Issues

- By passing a receiver address that rejects ether transfers, one may attack relayer tricking it to spend gas but not get any fee in exchange. Such attack will have zero cost for attacker. Consider sending refund to relayer in case an attempt to send it to the recipient failed. Another option is to deny at relayer withdrawal requests with non-zero refund in case recipient is a contract, because refunding contracts does not make much sense. Interestingly, it is not possible in general to tell for sure, whether transaction will be successful, or not. One may develop special smart contract that accepts ether when transaction is executed off-chain (during estimateGas), but rejects ether when executing on-chain. There is number of block and transaction attributes, such as hash of current block, that are not properly emulated during "estimateGas" execution, and could be used to detect, whether contract is invoked on-chain or off-chain.

  Resolved as: in the case of failure the relayer now gets all the refund.

- In case this method returned less than 32 bytes, it will read from memory after the actual returned data. In case returned data is all zeros, but memory after the data contains some non-zero bytes, this will treat failed transfer as a successful one. In case more than 32 bytes were returned, only first 32 bytes of returned data will be checked. The same applies to other assembly inlines.

  Resolved as: data length check added.

2.2 Unresolved Minor Issues

- Token should be probably of type IERC20 fo readability.

  > **Comment:** *we don't work with IERC20 interface anyhow so it's not necessary. Moreover it's more convenient to work with address type in case of `.call`*

- It is bad practice to use `msg.*` predefined variables in functions that are neither "public" nor "external", because such usage introduces implicit parameter and makes code harder to read.

  > **Comment**: *We think our approach is cheaper and more reliable - we ensure that we never pass incorrect msg.value as _processWithdraw arg.*

- This and similar utility function should probably be moved to a library. Moving to a library would clearly separate low-level utility code from high-level business logic code. And this function is not too small nor too simple. It is actually quite sophisticated, and uses advanced features such as ABI encoder and inline assembly.

  > **Comment**: *those functions are used only in one place and this functionality it too small.*

3. ETHMixer.sol

3.1 Fixed Issues

- In case "`_receiver`" will reject ether, transaction will be rolled back, and `relayer`, who already paid for gas, will not get anything in exchange. This may be used to attack relayers, and such attack would cost nothing for the attacker. Consider maintaining internal ether balances of recipient addresses inside the smart contract and add failed withdrawals to these

balances, so recipient will be able to take the ether later himself. Alternatively, you can send everything to the relayer.

Resolved as: relayer will implement a black list.

4. Mixer.sol

4.1 Resolved Architecture Issues

- Internal structure of the proof is implementation details and may change in future. Consider passing the whole proof as single “bytes” parameter and split into parts inside using abi.decode. The public part of the proof (“input”) should be passed as separate variables of proper type to the withdraw function, which should pack it to an array of field elements (or anything else) internally. This keeps the contract API more clear to a user, and it is not changed if another proof system is selected. The current API is for Groth16 only.

  Resolved as: proper data structures are used, input parameters are passed of proper types.

- Currently “public” function is defined in base contract, and it calls “internal” function defined in inherited contracts. Better architecture would be to have “public” functions in inherited contracts, with different set of parameters and different “payable” status, and common parts in “internal” function defined in base contract, which “public” public functions would call. The same idea applies to “deposit” function as well.

  > **Comment**: *That was our first implementation and results to more less readable code.*

4.2 Fixed Major Flaws

- Here an external contract is called in the ERC20 case, and the contract state is updated afterwards. This is the reentrancy attack pattern. Consider changing the state beforehand.

  Resolved as: non-reentrancy modifiers are used.

4.3 Minor Issues Scheduled for Future Releases

- This should be checked to be field elements to avoid malleability. It is a safer practice to make all necessary checks and assertions as early as possible so that internal functions can safely assume they are working with the right types.
- Functionality of `MerkleTreeWithHistory` looks more like library, than like base contract. Consider refactoring.
- The refund seems to be needed only for certain implementations. Moving public function to inherited contracts, and moving all common parts into “internal” function will let make this parameter optional. Also, this method

should probably check that _refund equals to msg.value, and should probably transfer refund to the recipient. Delegating this to inherited smart contract, especially those that do not need refund functionality at all is error-prone.

- Currently, `MiMCSponge` is a library being called via expensive DELEGATECALL, but `MerkleTreeWithHistory` is a base contract whose code is inlined. This is suboptimal, because MiMCSponge is called once per tree level, i.e. 20 times per insertions. It would be better to inline MiMCSponge code into MerkleTreeWithHistory, and then turn MerkleTreeWithHistory into a library. This way there will be only one DELEGATECALL per insertion.
- Storing commitments does not prevent accidental deposits with the same nullifier but different randomness. Adding the tree position to the nullifier hash generation would solve this problem better.
- In Ethereum, common name for operator is “owner”. Using a different word confuses readers.
- Leafindex should probably be indexed as well. Also including timestamp into logged events is usually redundant, because all logged events are bound to corresponding blocks these blocks already have timestamps.
- This comment is a bit misleading. Whereas tokens should be approved beforehand, Ether should be added to the message value in this transaction.
- This should be payable for certain implementations only. Consider refactoring. Probably, “`deposit`” function should be made abstract, and common deposit logic should be moved to internal function.
- This check could be made much more efficient if user would provide index of the root inside “`_roots`”.The root that was the latest at the moment user was constructing the proof may be not the latest at the moment proof is being verified by smart contract. Remember that there is a relayer between user and smart contract.
- Also, current approach does not scale, as increasing root history size will increase verification gas cost. Providing root index would address scaleability issue.
- It is uncommon for Solidity to start function names with underscore (“_”).
- `updateVerifier, disableVerifierUpdate, changeOperator` should probably log some event. There is no obvious reason to make these functions “external” rather than “public”.
- Name is confusing. Actually this is the amount of asset in single Mixer transfer. Consider renaming to something like “transferAmount”. Also, making this a compile-time constant will make the smart contract more gas-efficient.

4.4 Fixed Minor Issues

- Word “`receiver`” usually means a device that receives radio signals. Consider renaming to “recipient”.
- This relies on the knowledge of implementation details of MerkleTreeWithHistory. Consider refactoring to make “`_insert`” function to return index of just inserted leaf.
- Maybe equality should be allowed for the token case.

- This function duplicates a getter for `nullifierHashes`. Also In "withdraw" function nullifier was called "nullifierHash". Inconsistent naming makes code harder to read.
- parameter name is confusing, because there is no such thing as "account" anywhere else in the contract.
- In Ethereum, events are usually named via nouns, such as "Withdrawal". Also address to should probably be indexed as well.
- `_verifier` should have type IVerifier.
- There is no range check for `_denomination`. Consider requiring that it is non-zero.
- Inverting this storage field (renaming to "depositsDisabled") would make smart contract cheaper to deploy while not affecting usage costs.
- `toggleDeposits` function should probably log some event. This is not a one-time function according to the documentation comment. And for users it is important whether deposits are currently enabled or not.There is no obvious reason to make this function "external" rather than "public".Also result of the execution of this function depends on current state. This is error-prone. If operator address is controlled by two persons (for redundancy) and vulnerability is discovered, both operators will hurry to disable deposits, and the second attempt to disable deposits will effectively re-enable them. Consider passing desired state as a parameter.
- This function should be abstract, i.e. should be have body at all. Currently it does have empty body. See documentation for details.
- There is no code that sends money directly to the `operator`, so he does not have to be payable.
- In Ethereum it is common to use `bytes32` data type for hashes to distinguish them from numbers. Also the name is confusing. This map actually does not store nullifier hashes, but only boolean flags telling what hashes were already used. Consider renaming to something like "usedNullifiers" or "spentNullifiers".

5. MerkleTreeWithHistory.sol

5.1 Fixed Moderate Issues

- The `MiMCSponge` hash function is collision-resistant only for field elements as inputs. The field membership assertions should be there for each publicj function that uses it (for example in `hashLeftRight`, `_insert` functions).

  Resolved as: checks added.

5.2 Minor Issues Scheduled for Future Releases

- Library `Hasher` should be in separate file. Also name is too generic. Consider renaming.
- Contract MerkleTreeWithHistory could be turned into library. There is common pattern for this. The library defines a structure, calling contract defines storage variables of the structure type, and then passes these variables to the library by reference, so the library may read and update them. See, for example, this library, that implements Red-Black tree.
- Turning this into compile-time constant will make access cheaper. Unfortunately, configurable compile-time constants are not possible yet

(though there is change request for this). Currently, they may be approximated by abstract pure functions used in base contract and implemented in inherited contract, whose implementations just return constant values.
- It seems that history size is more likely to be customized than the number of levels, while history size is compile-time constant, but number of levels is not.
- This function is expensive and does not scale, as it reads up to 100 values from the storage. Consider adding another parameter: root history position to check. Authors' comment: In most cases user provides the latest root and providing index as argument will be more expensive.
- This may read uninitialized roots.
- This function may return uninitialized roots.
- This function could be turned into "`pure`" in case zero values will become compile-time constants.

Also the following issues should be resolved in the third party code:
- `MIMCSponge` function will be called via DELEGATECALL opcode, which is quite expensive. Consider changing to "internal" for embedding function code into the calling smart contract. **Note**: this requires a Solidity code of `MIMCSponge`, which is not ready yet.
- As long as the third parameter of "MiMCSponge" is always zero, hardcoding it to zero inside "MiMCSponge" could make "MiMCSponge" functions cheaper.

5.3 Fixed Minor Issues

- In Ethereum it is common to use `bytes32` data type for hashes to distinguish then from numbers.
- Some zero_value (valid commitment values) can be chosen maliciously. Consider using a hash of something and make it a constant, or use the block hash.
- Name is confusing. One may think that this storage variable contains current root hash, while actually it contains only the index of current root hash inside "`_roots`" array.
- `_roots` could be turned into fixed-size array that will be cheaper to use. Making this public will provide more standard read access to the roots history.
- `uint8` for `ROOT_HISTORY_SIZE` looks unnecessary limiting. Consider changing to a wider type.
- This code is confusing because it looks like a simple assignment, while actually it returns value. Good old "return" statement would be more readable.
- The name is misleading as this is the BN254 prime subgroup order. It should be made named compile-time constant.
- This code is confusing because it looks like a simple assignment, while actually it returns value. Good old "return" statement would be more readable.
- This function may return outdated or uninitialized slots from "`_filled_subtrees`".