Kyokan/FourthState

Security Assessment: Final Report

Minimal Viable Plasma - Rootchain Contract

February 28, 2019

Prepared For:

Matthew Slipper | Kyokan

mslipper@kyokan.io

Prepared By:

Alexander Wade | Authio

alex@authio.org

Paul Vienhage | Authio

paulv@authio.org

Alex Towle | Authio

alext@authio.org

Contents

1. **Executive Summary** 2
2. **Scope** 5
3. **System Overview** 6
4. **Findings** 9
5. **Recommendations** 13

1. Executive Summary

In December, Kyokan asked us to do a code review of FourthState's *Minimal Viable Plasma (Plasma MVP)* rootchain contract. We assessed the associated smart contract, and enclose in this document the findings and feedback provided to those involved.

The primary security properties Plasma MVP is meant to provide is that a UTXO entitles its owner to withdraw the entirety of its value to the mainchain, and that the owner will not be stopped from doing so as a result of (i) fraudulent challenges or (ii) successful exits of invalid transactions. Our assessment primarily targeted the interactions between challenges and exits, as well as the underlying finalization queue.

A preliminary review mapped the rootchain contract's insufficient use of SafeMath and relative lack of input validation to a large class of issues. We uncovered two important vulnerabilities that allowed: (i) the deposit exit queue to be permanently frozen via malicious deposit exit, and (ii) invalid transactions accepted as exits via unchecked overflow. Aside from adding SafeMath and performing input validation, our recommendation was to standardize and restrict the range of accepted RLP-encoded inputs, as our conclusion was that the wide range was exacerbating the input validation problem and represented an attack surface too large to meaningfully cover.

Subsequent efforts targeted properties of the FourthState implementation that differed from the original specification. Our analysis of the change made to add a `committedFee` parameter to the deposit and transaction exit functions revealed several inconsistencies, including an issue that allowed an operator to successfully challenge any exit with malicious blocks. These findings resulted in our recommendation to remove the fee mismatch challenge and `committedFee` entirely, and favor a more rigorous validation process for submitted transactions.

This assessment focused on the rootchain smart contract, `PlasmaMVP.sol`. The implementation of the child chain is out of scope. We strongly recommend Kyokan continue studying these features, as their integration into the final environment is crucial and should be assessed as part of a holistic review.

Update: *Following several rounds of revision, we performed a final review of the contract's progress, the results of which are summarized below.*

The `PlasmaMVP` contract acts as a root of trust for a plasma sidechain, safeguarding the security properties of the system for its users. It allows the operator to submit blocks and allows users to make deposits, exit transactions, and challenge invalid exits. To that end, it functions as a sort of virtual machine that simulates the properties of the plasma chain.

The contract operates from the Ethereum mainnet, meaning it is unable to directly see the vast majority of the plasma chain's data. Instead, it relies on a relatively small amount of information to build an accurate picture of the state of the sidechain at any given moment. As a result, `PlasmaMVP` relies heavily on the contract correctly interpreting the data it receives. In order to make strong statements about the security properties `PlasmaMVP` enforces, it is necessary to ensure that this interpretation is as unambiguous as possible.

The latest contracts represent a significant improvement on the initial iteration. The several commits added to address previous recommendations have significantly reduced the aforementioned variance in interpretation:

- **`9f75cd2a60f45aa1cafb95b1a4221184af873391`** implemented range checks on `committedFee` inputs, restricting an edge case that allowed an attacker to freeze the deposit exit queue.
- **`2922c20e722348c1c3662190500e442851b2c30d`** removed the `challengeFeeMismatch` function, simplifying challenges by folding the fee mismatch challenge functionality into `challengeExit`.
- **`ec7e0dd0639c194f4e35132402f256015b2d647e`** restricted acceptable signers in part by mitigating silent failure issues in the ECDSA library, which allowed bypassing of signature checks in some cases.
- **`51f51e6fa12c9ff8d0446217fbe73979886d911d`** implemented strict validation checks for transaction positions and data formats and extended SafeMath coverage to the entirety of the `PlasmaMVP` file.
- **`2a41203ea25d85001323606cf3266858a7f8852f`** restricted the transaction exit position calculation by requiring its result to fit in a uint128.

We recommend that further iteration on the `PlasmaMVP` contract follow the same principles embodied by previous recommendations. That is, the contract should strike an appropriate balance between strictness and simplicity. Some possible avenues for simplification are:

- Consolidate `startDepositExit` and `startFeeExit` into `startTransactionExit`
- Reduce the number of scenarios in which a mass exit is required to maintain security properties
- Minimize submission of redundant data to exits and challenges

This security review should be treated as a single part of a holistic review. We recommend that FourthState and Kyokan pursue similar evaluations of the other components in the Plasma MVP system. Of considerable importance are the following:

- The contract's deployment process
- The deployment of the plasma network
- The operator's key management
- The implementation of the Plasma MVP client, in particular the input validation practiced in each client

Finally, we recommend that care be taken to maintain code clarity and consistency of commenting and documentation. The significant changes introduced by audit fixes have not entirely been represented in the code's documentation or commenting.

2. Scope

The following material was provided as reference for FourthState's Plasma MVP specification:

1. Minimal Viable Plasma: https://ethresear.ch/t/minimal-viable-plasma/426
2. FourthState Labs Plasma Research: https://github.com/FourthState/plasma-research
3. Plasma Vulnerability: Sybil Txs -> Drained Contract: https://ethresear.ch/t/plasma-vulnerabiltity-sybil-txs-drained-contract/1654

The scope of the audit is limited to the Solidity files located in the FourthState/plasma-mvp-rootchain repository. The audit began at commit `74047018dd36b0bf76adbb9b350e45aa285f1974`.

Update: *FourthState addressed our recommendations in commits* `9f75cd2a60f45aa1cafb95b1a4221184af873391`, `2922c20e722348c1c3662190500e442851b2c30d`, `ec7e0dd0639c194f4e35132402f256015b2d647e`, `51f51e6fa12c9ff8d0446217fbe73979886d911d`, **and** `2a41203ea25d85001323606cf3266858a7f8852f`.

We reviewed the changes associated with our recommendations and provided FourthState with an updated list of recommendations.

3. System Overview

3.1 Plasma Transactions

I. A transaction is an RLP-encoded list of fields referencing up to two inputs, two outputs, and a transaction fee. It is uniquely represented by the tuple (`blockNum`, `txIndex`), which references the transaction at `txIndex` in plasma block `blockNum`.
   A. Each transaction must have at least 1 input and 1 output, and can have a maximum of 2 of each.
   B. A transaction contains a fee, which the owner of the first input agrees to pay to the operator.

II. Each transaction can have 1 or 2 inputs of any of the following types:
   A. A utxo input is a reference to an unspent output in a previous block. Its position is specified by the above tuple. A utxo input is characterized by a nonzero `blockNum`. A utxo input must also include the confirm signature associated with the output being spent.
      1. The `blockNum` must reference some previous plasma block.
      2. The `txIndex` can be a minimum of 0, and must be strictly less than the number of transactions in the block it references.
      3. The `oIndex` must be 0 or 1.
      4. The `depositNonce` must be 0, as a utxo is not a deposit.
   B. A deposit input is created directly from the rootchain, and is a special case of input with all fields set to 0 except the `depositNonce`. A deposit input does not need a confirm signature.
   C. A fee input is another special case of input in which the operator treats the fee created in another transaction as a spendable output. Only `blockNum` and `txIndex` are nonzero in a fee input. A fee input does not need a confirm signature.
      1. The `blockNum` must reference the block number in which the fee was created.
      2. The `txIndex` is a special case: 2**16 - 1. A standard utxo cannot have this index, as it is reserved for fee inputs.

III. Each transaction can have 1 or 2 outputs, which reference the address of the new owner, and the denomination included in the output.

IV. The sum of the fee and output denominations must equal the sum of the denominations of the inputs.

3.2 Block Submission

I. The operator, set at initialization, commits plasma blocks using `submitBlock(bytes32[] headers, uint[] txnsPerBlock, uint[] feePerBlock, uint blockNum)`, which allows one or more plasma blocks to be added to a list of all submitted plasma blocks.
II. Each block is uniquely identified by its block number, and is stored as a collection of fields:
    A. A `bytes32` block header, which is the merkle root of all transactions in the block
    B. The number of transactions recorded in the block
    C. The amount of fees collected by the operator across all transactions in the block
    D. The time at which the block was submitted to the Ethereum blockchain
III. Each block submitted increments a counter, `lastCommittedBlock`.

3.3 Deposits

I. Users are able to enter the plasma chain directly from the rootchain contract by using `deposit(address owner)`, which creates a new deposit owned by `owner`.
II. A list of unspent deposits is maintained by the operator for the plasma chain, each of which can be used as an input for a transaction on the plasma chain.
III. Each deposit is uniquely identified by the `depositNonce` at time of creation.
IV. The set of all deposits is maintained by the rootchain contract, associating each deposit with:
    A. The owner address
    B. The denomination of ETH used to create the deposit
    C. The Ethereum timestamp when the deposit was created
    D. The Ethereum block number when the deposit was created

3.4 Exits

I. The contract maintains a list of submitted exit transactions, as well as a separate list of submitted deposit exits. Both types of exits store:
   A. The amount being exited
   B. The fee committed to by the exiting user
   C. The Ethereum timestamp when the exit was created
   D. The owner of the exit
   E. The position on the child chain being exited (block number, tx index, output index, deposit nonce)
   F. An enum representing the exit's current state (NonExistance, Pending, Challenged, Finalized)

II. Exits are processed in a priority queue. Priority for standard exits is calculated by the tuple (`blockNum`, `txIndex`, `oIndex`), while deposit exits use the `depositNonce`.

III. Exits can be finalized once they reach the front of their priority queue and are older than the queue's challenge period.
   A. The deposit exit queue has a challenge period of 5 days
   B. The standard exit queue has a challenge period of 1 week

3.5 Challenges

I. Exits can be challenged while in their corresponding exit queue. There are two types of challenges, each with their own success criteria:
   A. A fee mismatch challenge will prove the existence of a spend that contradicts an exit's committed fee. A successful fee mismatch challenge sets an exit's state to `NonExistant`, allowing it to be re-opened.
   B. An inclusion challenge will prove the existence of a confirmed spend of an exiting output. A successful inclusion challenge sets an exit's state to `Challenged`. It cannot be reopened.

II. Each challenge must provide a proof of inclusion of a transaction in a position sequentially after the exit's transaction.

III. A successful challenge rewards the sender with the `minExitBond`, 200000 wei.

4. Findings

Insufficient Enforcement of Requirements for Transactions:

A generic transaction on a plasma chain should contain at most two inputs, two outputs, and a fee. A transaction is considered *valid* if (i) the sum of the outputs and the fee is equal to the sum of the inputs, and (ii) each utxo input should include a valid confirmation signature confirming the spend of the input. A transaction's validity only qualifies it to be included in a plasma block.

Once included in a plasma block, a transaction is only considered *confirmed* once its owner broadcasts a confirmation signature. This threshold prevents malicious behavior from the plasma owner, as the user is able to withhold it to ensure the security of their funds.

The following findings result from improper verification procedure applied to transactions:

4.1 Lack of signature validation in `challengeFeeMismatch` allows operator to successfully challenge any exit (Critical Severity)

The creator of an exit must commit to a fee of arbitrary value which may or may not match the amount recorded on the plasma chain. The `challengeFeeMismatch` function allows the operator to dispute these claims by proving that a transaction was included in a submitted block that contains a different value for the fee. The sole requirement for the challenge's success is inclusion in a submitted block.

Relying on block inclusion alone does not sufficiently prove a transaction confirmation. Exits should provide confirmation signatures of each input, and challenges should provide a confirmation signature that spends the attempted exit.

Update: *This issue was addressed in commit* **`2922c20e722348c1c3662190500e442851b2c30d`** *with the removal of the* `challengeFeeMisMatch` *function.*

Exit Queue Blocking:

Both the transaction and deposit exit queues are finalized the same way: the queue is looped over in order of priority, with each exit being subjected to the same criteria for finalization. That criteria requires that (i) the exit was created more than 1 week ago, (ii) the rootchain contract must contain the funds necessary to exit, (iii) at least 80000 gas remains in the transaction. The finalization loop requires that each subsequent exit be successfully finalized before proceeding to the next exit, meaning that if the exit at the front of the queue cannot be finalized for some reason, the queue is effectively unable to process any exits with a lower priority.

The following finding presents a method by which the contract's deposit exit queue can be permanently frozen:

4.2 Arbitrary fee commitments allow malicious actor to freeze deposit exit queue (Critical Severity)

The creator of a deposit exit must to commit to a fee of arbitrary value, and can commit to a fee greater than the value being exited. A malicious actor is then able to use `startDepositExit` to construct a deposit exit that cannot be challenged, but which will inevitably cause an underflow in the deposit finalization loop which is caught and reverted by SafeMath. Each time the finalization loop is called for the deposit queue, the same deposit will be at the front of the queue and will cause the transaction to fail - permanently halting the finalization of lower-priority deposits.

Update: *This issue was addressed as of commit* `9f75cd2a60f45aa1cafb95b1a4221184af873391`*, by requiring that committed fees for deposit exits be less than the exiting amount.*

Exiting invalid transactions:

Transaction exit ordering uses the following formula for exit priority calculation between transactions in the same plasma block: `1000000 * txPos[0] + 10 * txPos[1] + txPos[2]`, where `txPos` is a `uint256[3]` representing the position of the UTXO being exited. The fields in `txPos` represent, in order, (i) the transaction's block number on the child chain, (ii) the transaction's index in its block, and (iii) the index of the output being exited. The result of the priority calculation is assumed to fit entirely within a `uint128`, as it must be concatenated with another `uint128` representing the time at which the exit can be successfully finalized. A critical feature of the priority calculation is that its right 128 bits must be unique, as collisions in this calculation would imply it is possible to exit two different transactions with the same level of priority.

A valid transaction has values for `txPos` that are assumed to fall within the following range: (i) the blocknumber, `txPos[0]`, must be under `10 ** 40` (a higher blocknumber will overflow into the left 128 bits of the priority calculation), (ii) the transaction index, `txPos[1]`, is assumed to be under `2 ** 16` (the maximum depth of the plasma chain's state tree), and (iii) the output index, `txPos[2]`, must be either `0` or `1` (a maximum of 2 outputs can be created per transaction, making the output indices `0` or `1`).

The following finding presents a method by which spent transactions can be successfully exited by value submissions outside this range:

4.3 Overflow in exit position allows sender to exit spent transactions unchallenged (High Severity)

The acceptable range of `txPos` for valid transactions is assumed to hold true, but is not explicitly checked for compliance. In the absence of such a check, users are free to submit values anywhere in the uint256 range for each of the fields in `txPos`. Combined with a priority calculation that does not utilize SafeMath, users are also able to provide input to txPos that causes overflows. This is not immediately harmful, as the `startTransactionExitHelper` function ensures both (i) that `txPos[0]` corresponds to an actual, submitted plasma block, and (ii) that `txPos[1]` represents a valid transaction index within that block. `txPos[2]`, though, does not necessarily need to be 0 or 1. The only requirement on the value of `txPos[2]` is that `2 * txPos[2]` must be either `0` or `2`, as this calculation provides an offset representing the position of the new owner and denomination of the UTXO

being exited. Because this calculation does not use SafeMath, the user is free to submit a value for `txPos[2]` that is not 0 or 1, but still satisfies `2 * txPos[2] ∈ {0, 2}`. Outside of `0` and `1`, this requirement is satisfied by both `(uint(-1) / 2) + p`, where p is an integer such that `p ∈ [-5, +2]`.

This range of values presents several interesting vectors for invalid transaction exit finalization. For any given choice of p, there exists the following requirement: `msg.sender` must equal `txList[12 + 2*p]`, where `txList` is a list of `RLPItems` derived from an RLP-encoded transaction. Additionally, the exit will be created with value equal to: `txList[13 + 2*p]`. Within the ranges of the RLP-encoded transaction, we see an interesting opportunity for the various owners of the transaction's component inputs and outputs:

- For `p ∈ {-3, 0}`, the corresponding requirement is: `msg.sender` is equal to the owner of input `1`, or `2` respectively. The value of the exit then becomes the first 32 bytes of the confirm signature of input `1`, or `2` respectively.
- For `p ∈ {1, 2}`, the corresponding requirement is: `msg.sender` is equal to the new owner of output `1`, or `2` respectively. The value of the exit then becomes the value of output `1`, or `2` respectively.
  In essence, not only can each pair of spent outputs for a transaction can be exited by their owners, but also each pair of inputs for a transaction can be exited for substantial reward by their original owners.

The result of submitting these values is that even a spent transaction can be exited in several different ways with no possible challenge: the `challengeExit` function checks the input `exitingTxPos` against a completely valid transaction included in a completely valid block, so submitting the same invalid value for `exitingTxPos[2]` will only work if a signed, submitted, validated transaction on the plasma chain has an invalid output index, grounds for a mass exit. The severity of this attack is mitigated slightly by the fact that the only possible invalid values set the pending exit to a very low priority, such that any legitimate exit will be processed first. The spent transaction can be completely exited as long as no valid exits have been created within the past week, meaning a queue of valid exits must be constantly maintained in order to prevent the finalization of the malicious exit.

Update: *This finding was directly addressed in commit* **`2922c20e722348c1c3662190500e442851b2c30d`** *with the addition of SafeMath calculation of exit position and a bounds check on* `txPos[2]`.

5. Recommendations

Decrease number of transaction exit and challenge patterns:
The rootchain contract contains several functions that perform similar purposes, but do not sufficiently emphasize modularity; the following recommendations are provided as a means to simplify the rootchain contract in order to reduce the number of edge cases:

5.1 Remove arbitrary fee commitments and fee challenges. Remove fee exits

A “fee mismatch” refers to a potential scenario allowed by the rootchain contract; that on exiting a deposit or transaction, the exiting user must provide a fee commitment to be paid to the operator in exchange for the original submission of an unconfirmed deposit or transaction to the plasma chain. The “mismatch” refers to the potential for the user to report the submission of a transaction with a different fee than the value recorded in the plasma chain. Note that because the submission of a transaction does not require publishing a confirmation signature for the transaction, the fee agreed to by the user is not signed until a confirmation signature is present.

We found the presence of the fee commitment to be highly subject to manipulation by the operator. The paradigm of the confirm signature used to commit to a transaction is important to maintain throughout the contract. By removing arbitrary fee inputs in `submitBlock`, `startDepositExit`, and `startTransactionExit`, fees do not need to be a trusted source of data. Rather, in each exit function mentioned, they can be parsed directly from the transaction bytes. Additionally, removing `challengeFeeMismatch` entirely and standardizing fee exit procedure to require a process like `challengeExit` will restrict the number of edge cases significantly.

As for the changes to the specification itself, our recommendation is to treat the fee as a sort of third output, and subject each to the same exit as a normal transaction. This would allow the removal of `startFeeExit` as well.

Update: *With the consolidation of fee mismatch challenges into the* `challengeExit` *function, the challenge process has been considerably simplified and presents a smaller attack surface. Fee commitments are now restricted to a reasonable range, and the process of challenging a fee mismatch now closely follows the process for challenging a standard transaction.*

5.2 Use SafeMath throughout all contracts, and practice input sanitization where possible

Many discovered vulnerabilities in the rootchain contract were a direct result of the inconsistent use of SafeMath. The rootchain contract keeps record of a fairly abstract form of data, and ingests a similarly abstract form of data for most external functions. The inputs in several functions have massive ranges for acceptable values, which makes it all the more important to ensure not only that values are generally within expected ranges, but also that operations using those values are performed safely and without side effects.

Update: *The main contract now features overflow and underflow-safe operations, but the* `BytesUtil` *and* `TMSimpleMerkleTree` *libraries still contain a few locations where unprotected operations are allowed. Changes to the* `RLPReader` *library and* `PlasmaMVP` *helper functions have introduced much more rigorous validation of raw transaction inputs.*

5.3 Standardize allowed RLP formats for each data type

The RLPReader library is used to digest and interpret parts of transactions submitted to exit and challenge functions, and provides several functions to translate from RLP encoded data to several native Solidity types: (i) bool, (ii) address, (iii) uint256, and (iv) bytes. These functions interpret some RLP data rather loosely which lends itself to unspecified behavior. In addition to implementing safe arithmetic and input sanitization checks as mentioned above, the severity of a wide class of potential issues can be mitigated significantly by making stronger statements about exactly how RLP encoded data should be interpreted. For example, the transaction exit vulnerability above described how, with certain inputs to `txPos[2]`, the exit's amount could be manipulated to interpret part of the inputs' confirm signatures as uints. By requiring that any payload size interpreted by the `toUint` function have a length of exactly 32, the confirm signature could never have been interpreted as a uint. Additionally, standard interpretations for certain RLP encodings will reduce ambiguity for users and developers attempting to use the system. The following lists a few suggested changes to increase the relative specificity of the `RLPReader` library:

- **Standardize uint representation:**
    - Encoding type: string, length 32 bytes
    - First byte ∈ [0x80, 0xa0]
    - Encoded size: 33 bytes
    - Use for: block number, transaction index, output index, deposit nonce, transaction denominations, transaction fees

- **Standardize address representation:**
    - Encoding type: string, length 20 bytes
    - First byte is always 0x94
    - Encoded size: 21 bytes
    - Use for: input owners, output owners
- **Standardize signature representation:**
    - Encoding type: long string, length 65 bytes
    - First two bytes: 0xb8 0x41
    - Encoded size: 67 bytes
    - Use for: input confirm signatures, transaction confirm signatures
- **Standardize transaction data representation:**
    - Encoding type: long list, payload length: 581 bytes (0x0245)
    - First bytes: 0xf9 0x02 0x45
    - Encoded size: 584 bytes
    - Contents, in order: [blocknum1, txindex1, oindex1, depositnonce1, owner1, confsig1, blocknum2, txindex2, oindex2, depositnonce2, owner2, confsig2, newowner1, denomination1, newowner2, denomination2, txfee]
- **Standardize transaction signatures representation:**
    - Encoding type: long list, payload length: 134 bytes (0x86)
    - First bytes: 0xf8 0x86
    - Encoded size: 136 bytes
    - Contents, in order: [txsig1, txsig2]
- **Standardize transaction representation:**
    - Encoding type: long list, payload length: 720 bytes (0x02d0)
    - First bytes: 0xf9 0x02 0xd0
    - Encoded size: 723 bytes
    - Contents, in order: [transaction data, transaction signatures]

Update: *The canonical form of a transaction was agreed to and formalized, and changes to the contract were made to allow for validation of a raw transaction's canonical form. The documentation and code comments should be updated to reflect the changes made.*

5.4 Reject non-canonical signature form

The signature library used doesn't throw on signatures which have the form (r, -s mod n), and so accepts a non-canonical signature form. Given that potential signature malleability may have significant side effects for the rootchain contract, the best precaution against this would be to include a check that s < n/2 in the ECDSA library.

5.5 Avoid use of memory at 0x00

The RLPReader.toRlpItem converts an input bytes array to its RLP representation, which it then returns. When the input bytes array has length 0, the returned RLP representation will have a memPtr at 0x00. In the case memory at 0x00 is used, the RLP representation should avoid this common location and instead return a pointer to the input bytes in memory.

Update: *The* `RLPReader` *library no longer allows memory pointers to 0x00.*