CONSENSYS
Diligence

Omisego Morevp Audit

Date	January 2020
Lead Auditor	Alexander Wade
Co-auditors	Daniel Luca, Martin Ortner

1 Summary

ConsenSys Diligence conducted a security audit of OmiseGo's plasma framework contracts. The contracts are their implementation of More Viable Plasma (MoreVP), which is based on Minimal Viable Plasma (MVP). MoreVP aims to improve on Plasma's UX by getting rid of MVP's confirmation signatures in favor of a more involved exit game.

Diligence performed a secondary review of the plasma contracts following OmiseGo's implementation of fee transaction types as well as their inclusion of fixes from our initial review.

2 Audit Scope

Our review was concerned primarily with the smart contracts in OmiseGo's plasma-contracts repository. We began our review at commit **e13aaf759c979cf6516c1d8de865c9e324bc2db6**.

Our subsequent review began at commit **9d79e35811a483277d4cd8b06b1678efc9f33151**.

A complete list of Solidity files reviewed can be found in the appendix.

3 Key Observations/Recommendations

- The bulk of the code (~80%) is concerned with the MoreVP exit game. Of this code, large portions of many contracts are irrelevant to the intended behavior of the system: boilerplate and leftovers from unused extensibility features.
    - The inclusion of this code makes it difficult to understand many components. Code is spread across a sprawling file structure, and understanding individual features involves hopping between files frequently.
    - The unused code may have unintended side effects. External calls and delegatecalls are often made. Memory is frequently allocated without cause. Functions often have more parameters than they use. It may be that these affect the function of the contracts in some subtle way.
    - **Update:** Since our initial review, significant refactoring has removed much of the unused code initially found. In particular, the removal of unused parameters and features like the output guard handler made it easier to reason about the code (see 5.12).
- Many future features are planned, but not yet implemented. The extensibility features mentioned above are meant to support new features when they are released, but, crucially, will never serve a purpose in the existing system post-deployment. Assuming the system is deployed and initialized correctly, the extensibility features in the existing codebase will never be active.
    - Instead, future features will be added via the registration of new exit games and vaults. This process involves a quarantine period whereby users can ensure that new features are understood and audited before being used. The quarantine period is based on the minimum exit period, so that users are free to opt-out via exit before any new features become active.
    - Some future features are represented in the current system. Of note is plasma transaction fees, which are represented in the exit state transition verifier contract. This contract checks that the sum of the denominations of each input is greater than or equal to the sum of the denominations of each output. Should fees not be implemented, this representation is incorrect and could lead to invalid transactions exiting successfully.

    - **Update:** Since our initial review, transaction fees have been implemented and included in the smart contracts as first-class citizens. However, the contracts are still highly complex due to heavy use of abstractions and a complicated transaction decoding scheme. The potential to enable future transaction types and decoding schemes plays a large role in obfuscating the business logic of the contracts. This obfuscation is magnified by the codebase's aforementioned sprawling file structure and relative lack of code commenting. Further work should attempt to limit this sprawl and focus on making implementation details more clear.

- Because MoreVP does not use confirmation signatures, verifying a transaction's validity is nearly impossible in the resource-constrained environment of the EVM. To get around this limitation, MoreVP allows invalid transactions to be exited. In order to avoid losing funds, users must be sure that they are running the child chain watcher, and that it is correctly configured to notify them of byzantine scenarios.

    - As a safeguard to the potential exiting of invalid transactions, users can perform a mass exit. In this case, the gas cost required to exit each UTXO is a critically-important bottleneck. Should a mass exit be too resource-intensive, the network may be clogged up and invalid transactions may be exited successfully. Future work on this codebase should make additional steps to ensure that exit game implementations are as efficient as possible.

- **Update:** As with any highly-complex system, it is impossible to account for every possibility before launching. Our review was primarily concerned with the plasma smart contracts as the critical point of infrastructure, but left other important components nearly untouched. Of particular note is the implementation of the child chain watcher (and its integration with the plasma chain), which serves as a crucial safeguard for users during production.

    - Our review uncovered several issues in a highly complex codebase, and more were uncovered by OmiseGo's development team during the engagement. We highly recommend proceeding with caution: rather than pushing immediately for a full-scale production release, a testnet, public bug bounty, limited release, or a combination of all of these would allow OmiseGo to work out the kinks of the system before it reaches critical mass.

4 Security Specification

This section describes, **from a security perspective**, the expected behavior of the system under audit. It is not a substitute for documentation. The purpose of this section is to identify specific security properties that were validated by the audit team.

4.1 Actors

The relevant actors are as follows:

- **Operator:** Runs the child chain and submits child chain blocks to the `PlasmaFramework` contract.
- **Maintainer:** An address controlled by OmiseGo that has permissions to enable some extensibility features in the root chain contracts.
- **Deployer:** The address used to deploy the system's contracts. Following deployment, the deployer should revoke their permissions in some `Ownable` contracts.
- **User:** An EOA that has deposited ERC20 or Ether into `PlasmaFramework` vaults. Users hold assets in the child chain.
- **Watcher:** A node that observes properties of the child chain and root chain contracts and signals if a byzantine scenario is detected.

4.2 Trust Model

In any smart contract system, it's important to identify what trust is expected/required between various actors. For this audit, we established the following trust model:

Deployment and Initialization

Before the plasma chain can start submitting blocks to the root chain contract, it must be deployed and initialized correctly. That the contracts are correctly initialized is crucial. The safety of many system components rely on the revocation of permissions post-initialization, as well as the correct injection of parameters into each contract constructor.

- `PlasmaFramework.constructor` - `minExitPeriod`
    - The minimum exit period should be 1 week
- `PlasmaFramework.constructor` - vault and exit game immunities
    - `PlasmaFramework` should be initialized with 2 immunities for vaults, which should be filled during initialization by the erc20 and eth vaults.

    - `PlasmaFramework` should be initialized with 1 immunity for exit games, which should be filled during initialization by the `PaymentExitGame` contract, configured with each of the components mentioned below.

- `OutputGuardHandlerRegistry` and `SpendingConditionRegistry`
    - Following deployment, the owner of these contracts should revoke ownership by transferring permissions to the zero address.
    - Only one payment output type should be registered in `OutputGuardHandlerRegistry`.
    - Two spending conditions should be registered in `SpendingConditionRegistry`, with the same output type registered in `OutputGuardHandlerRegistry`, and two different transaction types. These spending conditions should be separately-deployed instances of `PaymentOutputToPaymentTxCondition.sol`.
    - **Update:** The `OutputGuardHandlerRegistry` was removed after refactoring suggested in 5.12.

- `PaymentExitGame.constructor` (args)
    - `ethVaultId` and `erc20VaultId` should be the deployed `EthVault.sol` and `ERC20Vault.sol` contracts. They should be different addresses. Each should be initialized with the correct deposit verifier contract.
    - `outputGuardHandlerRegistry`, `spendingConditionVerifier`, `stateTransitionVerifier`, and `txFinalizationVerifier` should be the deployed `OutputGuardHandlerRegistry.sol`, `SpendingConditionRegistry.sol`, `PaymentTransactionStateTransitionVerifier.sol`, and `TxFinalizationVerifier.sol`
    - **Update:** The `OutputGuardHandlerRegistry` was removed after refactoring suggested in 5.12.

User Behavior

The safety of the system relies in large part on vigilant monitoring and decisive action on the part of the system's users. Users should be running the child chain watcher, which monitors the plasma chain and main chain contracts to alert the user if an exit is needed. In the event of a byzantine operator or some discovered flaw, it is critical that users be able to exit quickly and correctly.

- The watcher should monitor registered exit games and vaults, and alert users if a new exit game is registered. Users should examine each registered exit game to ensure it complies with their expectations of the system.
- The watcher should be used by as many users as is feasible.
- In the event that an exit is needed, users must be able to coordinate and exit safely.

5 Issues

Each issue has an assigned severity:

- **Minor** issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- **Medium** issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- **Major** issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- **Critical** issues are directly exploitable security vulnerabilities that need to be fixed.

5.1 `Merkle.checkMembership` allows existence proofs for the same leaf in multiple locations in the tree Critical ✓ Addressed

Resolution

This was addressed in omisego/plasma-contracts#533 by including a check in `PosLib` that restricts transaction indices to between `0` and `2**16 - 1` inclusive. A subsequent change in omisego/plasma-contracts#547 ensured the passed-in index satisfied the recommended criterion.

Description

`checkMembership` is used by several contracts to prove that transactions exist in the child chain. The function uses a `leaf`, an `index`, and a `proof` to construct a hypothetical

root hash. This constructed hash is compared to the passed in `rootHash` parameter. If the two are equivalent, the proof is considered valid.

The proof is performed iteratively, and uses a pseudo-index (`j`) to determine whether the next proof element represents a "left branch" or "right branch":

code/plasma_framework/contracts/src/utils/Merkle.sol:L28-L41

```
uint256 j = index;
// Note: We're skipping the first 32 bytes of `proof`, which holds the size o
for (uint256 i = 32; i <= proof.length; i += 32) {
    // solhint-disable-next-line no-inline-assembly
    assembly {
        proofElement := mload(add(proof, i))
    }
    if (j % 2 == 0) {
        computedHash = keccak256(abi.encodePacked(NODE_SALT, computedHash, pro
    } else {
        computedHash = keccak256(abi.encodePacked(NODE_SALT, proofElement, com
    }
    j = j / 2;
}
```

If `j` is even, the computed hash is placed before the next proof element. If `j` is odd, the computed hash is placed after the next proof element. After each iteration, j is decremented by `j = j / 2` .

Because `checkMembership` makes no requirements on the height of the tree or the size of the proof relative to the provided `index` , it is possible to pass in invalid values for `index` that prove a leaf's existence in multiple locations in the tree.

Examples

By modifying existing tests, we showed that for a tree with 3 leaves, leaf 2 can be proven to exist at indices 2, 6, and 10 using the same proof each time. The modified test can be found here: https://gist.github.com/wadeAlexC/01b60099282a026f8dc1ac85d83489fd#file-merkle-test-js-L40-L67

```
it('should accidentally allow different indices to use the same proof', async () =>
  const rootHash = this.merkleTree.root;
```

```
  const proof = this.merkleTree.getInclusionProof(leaves[2]);

  const result = await this.merkleContract.checkMembership(
    leaves[2],
    2,
    rootHash,
    proof,
  );
  expect(result).to.be.true;

  const nextResult = await this.merkleContract.checkMembership(
    leaves[2],
    6,
    rootHash,
    proof,
  );
  expect(nextResult).to.be.true;

  const nextNextResult = await this.merkleContract.checkMembership(
    leaves[2],
    10,
    rootHash,
    proof,
  );
  expect(nextNextResult).to.be.true;
});
```

Conclusion

Exit processing is meant to bypass exits processed more than once. This is implemented using an “output id” system, where each exited output should correspond to a unique id that gets flagged in the `ExitGameController` contract as it’s exited. Before an exit is processed, its output id is calculated and checked against `ExitGameController`. If the output has already been exited, the exit being processed is deleted and skipped. Crucially, output id is calculated differently for standard transactions and deposit transactions: deposit output ids factor in the transaction index.

By using the behavior described in this issue in conjunction with methods discussed in issue 5.8 and issue 5.10, we showed that deposit transactions can be exited twice using

indices `0` and `2**16`. Because of the distinct output id calculation, these exits have different output ids and can be processed twice, allowing users to exit double their deposited amount.

A modified `StandardExit.load.test.js` shows that exits are successfully enqueued with a transaction index of `65536`: https://gist.github.com/wadeAlexC/4ad459b7510e512bc9556e7c919e0965#file-standardexit-load-test-js-L55

Recommendation

Use the length of the proof to determine the maximum allowed index. The passed-in index should satisfy the following criterion: `index < 2**(proof.length/32)`. Additionally, ensure range checks on transaction position decoding are sufficiently restrictive (see issue 5.10).

Corresponding issue in plasma-contracts repo: https://github.com/omisego/plasma-contracts/issues/546

5.2 Improper initialization of spending condition abstraction allows "v2 transactions" to exit using `PaymentExitGame` Major ✓ Addressed

Resolution
This was addressed in omisego/plasma-contracts#478 by requiring that `PaymentStartStandardExit` and `PaymentStartInFlightExit` check the exiting transaction's transaction type.

Description

`PaymentOutputToPaymentTxCondition` is an abstraction around the transaction signature check needed for many components of the exit games. Its only function, `verify`, returns `true` if one transaction (`inputTxBytes`) is spent by another transaction (`spendingTxBytes`):

code/plasma_framework/contracts/src/exits/payment/spendingConditions/PaymentOutputT
L69

```
function verify(
    bytes calldata inputTxBytes,
    uint16 outputIndex,
    uint256 inputTxPos,
    bytes calldata spendingTxBytes,
    uint16 inputIndex,
    bytes calldata signature,
    bytes calldata /*optionalArgs*/
)
    external
    view
    returns (bool)
{
    PaymentTransactionModel.Transaction memory inputTx = PaymentTransactionMod
    require(inputTx.txType == supportInputTxType, "Input tx is an unsupported

    PaymentTransactionModel.Transaction memory spendingTx = PaymentTransactior
    require(spendingTx.txType == supportSpendingTxType, "The spending tx is ar

    UtxoPosLib.UtxoPos memory utxoPos = UtxoPosLib.build(TxPosLib.TxPos(inputT
    require(
        spendingTx.inputs[inputIndex] == bytes32(utxoPos.value),
        "Spending tx points to the incorrect output UTXO position"
    );

    address payable owner = inputTx.outputs[outputIndex].owner();
    require(owner == ECDSA.recover(eip712.hashTx(spendingTx), signature), "Tx

    return true;
}
```

Verification process

The verification process is relatively straightforward. The contract performs some basic input validation, checking that the input transaction's `txType` matches `supportInputTxType`, and that the spending transaction's `txType` matches `supportSpendingTxType`. These values are set during construction.

Next, `verify` checks that the spending transaction contains an input that matches the position of one of the input transaction's outputs.

Finally, `verify` performs an EIP-712 hash on the spending transaction, and ensures it is signed by the owner of the output in question.

Implications of the abstraction

The abstraction used requires several files to be visited to fully understand the function of each line of code: `ISpendingCondition`, `PaymentEIP712Lib`, `UtxoPosLib`, `TxPosLib`, `PaymentTransactionModel`, `PaymentOutputModel`, `RLPReader`, `ECDSA`, and `SpendingConditionRegistry`. Additionally, the abstraction obfuscates the underlying spending condition verification primitive where used.

Finally, understanding the abstraction requires an understanding of how `SpendingConditionRegistry` is initialized, as well as the nature of its relationship with `PlasmaFramework` and `ExitGameRegistry`. The aforementioned `txType` values, `supportInputTxType` and `supportSpendingTxType`, are set during construction. Their use in `ExitGameRegistry` seems to suggest they are intended to represent different versions of transaction types, and that separate exit game contracts are meant to handle different transaction types:

code/plasma_framework/contracts/src/framework/registries/ExitGameRegistry.sol:L58-L78

```
/**
 * @notice Registers an exit game within the PlasmaFramework. Only the mainta
 * @dev Emits ExitGameRegistered event to notify clients
 * @param _txType The tx type where the exit game wants to register
 * @param _contract Address of the exit game contract
 * @param _protocol The transaction protocol, either 1 for MVP or 2 for MoreVP
 */
function registerExitGame(uint256 _txType, address _contract, uint8 _protocol)
    require(_txType != 0, "Should not register with tx type 0");
    require(_contract != address(0), "Should not register with an empty exit g
    require(_exitGames[_txType] == address(0), "The tx type is already registe
    require(_exitGameToTxType[_contract] == 0, "The exit game contract is alre
    require(Protocol.isValidProtocol(_protocol), "Invalid protocol value");
```

```
        _exitGames[_txType] = _contract;
        _exitGameToTxType[_contract] = _txType;
        _protocols[_txType] = _protocol;
        _exitGameQuarantine.quarantine(_contract);

        emit ExitGameRegistered(_txType, _contract, _protocol);
    }
```

Migration and initialization

The migration script seems to corroborate this interpretation:

code/plasma_framework/migrations/5_deploy_and_register_payment_exit_game.js:L109-L124

```
// handle spending condition
await deployer.deploy(
    PaymentOutputToPaymentTxCondition,
    plasmaFramework.address,
    PAYMENT_OUTPUT_TYPE,
    PAYMENT_TX_TYPE,
);
const paymentToPaymentCondition = await PaymentOutputToPaymentTxCondition.depl

await deployer.deploy(
    PaymentOutputToPaymentTxCondition,
    plasmaFramework.address,
    PAYMENT_OUTPUT_TYPE,
    PAYMENT_V2_TX_TYPE,
);
const paymentToPaymentV2Condition = await PaymentOutputToPaymentTxCondition.de
```

The migration script shown above deploys two different versions of `PaymentOutputToPaymentTxCondition`. The first sets `supportInputTxType` and `supportSpendingTxType` to `PAYMENT_OUTPUT_TYPE` and `PAYMENT_TX_TYPE`, respectively. The second sets those same variables to `PAYMENT_OUTPUT_TYPE` and `PAYMENT_V2_TX_TYPE`, respectively.

The migration script then registers both of these contracts in `SpendingConditionRegistry`, and then calls `renounceOwnership`, freezing the spending conditions registered permanently:

code/plasma_framework/migrations/5_deploy_and_register_payment_exit_game.js:L126-L135

```
console.log(`Registering paymentToPaymentCondition (${paymentToPaymentConditio
await spendingConditionRegistry.registerSpendingCondition(
    PAYMENT_OUTPUT_TYPE, PAYMENT_TX_TYPE, paymentToPaymentCondition.address,
);

console.log(`Registering paymentToPaymentV2Condition (${paymentToPaymentV2Cond
await spendingConditionRegistry.registerSpendingCondition(
    PAYMENT_OUTPUT_TYPE, PAYMENT_V2_TX_TYPE, paymentToPaymentV2Condition.addre
);
await spendingConditionRegistry.renounceOwnership();
```

Finally, the migration script registers a single exit game contract in `PlasmaFramework`:

code/plasma_framework/migrations/5_deploy_and_register_payment_exit_game.js:L137-L143

```
// register the exit game to framework
await plasmaFramework.registerExitGame(
    PAYMENT_TX_TYPE,
    paymentExitGame.address,
    config.frameworks.protocols.moreVp,
    { from: maintainerAddress },
);
```

Note that the associated `_txType` is permanently associated with the deployed exit game contract:

code/plasma_framework/contracts/src/framework/registries/ExitGameRegistry.sol:L58-L78

```
/**
 * @notice Registers an exit game within the PlasmaFramework. Only the maintai
 * @dev Emits ExitGameRegistered event to notify clients
 * @param _txType The tx type where the exit game wants to register
 * @param _contract Address of the exit game contract
 * @param _protocol The transaction protocol, either 1 for MVP or 2 for MoreVP
 */
function registerExitGame(uint256 _txType, address _contract, uint8 _protocol)
    require(_txType != 0, "Should not register with tx type 0");
    require(_contract != address(0), "Should not register with an empty exit g
    require(_exitGames[_txType] == address(0), "The tx type is already registe
    require(_exitGameToTxType[_contract] == 0, "The exit game contract is alre
    require(Protocol.isValidProtocol(_protocol), "Invalid protocol value");

    _exitGames[_txType] = _contract;
    _exitGameToTxType[_contract] = _txType;
    _protocols[_txType] = _protocol;
    _exitGameQuarantine.quarantine(_contract);

    emit ExitGameRegistered(_txType, _contract, _protocol);
}
```

Conclusion

Crucially, this association is never used. It is implied heavily that transactions with some `txType` must use a certain registered exit game contract. In fact, this is not true. When using `PaymentExitGame`, its routers, and their associated controllers, the `txType` is invariably inferred from the encoded transaction, not from the mappings in `ExitGameRegistry`. If initialized as-is, both `PAYMENT_TX_TYPE` and `PAYMENT_V2_TX_TYPE` transactions may be exited using `PaymentExitGame`, provided they exist in the plasma chain.

Recommendation

- Remove `PaymentOutputToPaymentTxCondition` and `SpendingConditionRegistry`
- Implement checks for specific spending conditions directly in exit game controllers. Emphasize clarity of function: ensure it is clear when called from the top level that a signature verification check and spending condition check are being performed.

- If the inferred relationship between `txType` and `PaymentExitGame` is correct, ensure that each `PaymentExitGame` router checks for its supported `txType`. Alternatively, the check could be made in `PaymentExitGame` itself.

Corresponding issue in plasma-contracts repo: https://github.com/omisego/plasma-contracts/issues/472

5.3 RLPReader - Leading zeroes allow multiple valid encodings and exit / output ids for the same transaction Major ✓ Addressed

Resolution
This was addressed in omisego/plasma-contracts#507 with the addition of checks to ensure primitive decoding functions in `RLPReader` (`toAddress` , `toUint` , `toBytes32`) do not decode lists. A subsequent change in omisego/plasma-contracts#476 rejects leading zeroes in `toUint` , and improves on size requirements for decoded payloads. Note that the scalar "0" should be encoded as `0x80` .

Description

The current implementation of RLP decoding can take 2 different `txBytes` and decode them to the same structure. Specifically, the `RLPReader.toUint` method can decode 2 different types of bytes to the same number. For example:

- `0x821234` is decoded to `uint(0x1234)`
- `0x83001234` is decoded to `uint(0x1234)`
- `0xc101` can decode to `uint(1)` , even though the tag specifies a short list
- `0x01` can decode to `uint(1)` , even though the tag specifies a single byte

As explanation for this encoding:

`0x821234` is broken down into 2 parts:

- `0x82` - represents `0x80` (the string tag) + `0x02` bytes encoded
- `0x1234` - are the encoded bytes

The same for `0x83001234` :

- `0x83` - represents `0x80` (the string tag) + `0x03` bytes encoded
- `0x001234` - are the encoded bytes

The current implementation casts the encoded bytes into a uint256, so these different encodings are interpreted by the contracts as the same number:

`uint(0x1234) = uint(0x001234)`

code/plasma_framework/contracts/src/utils/RLPReader.sol:L112

```
result := mload(memPtr)
```

Having different valid encodings for the same data is a problem because the encodings are used to create hashes that are used as unique ids. This means that multiple ids can be created for the same data. The data should only have one possible id.

The encoding is used to create ids in these parts of the code:

- Outputid.sol

code/plasma_framework/contracts/src/exits/utils/OutputId.sol:L18

```
return keccak256(abi.encodePacked(_txBytes, _outputIndex, _utxoPosValue));
```

code/plasma_framework/contracts/src/exits/utils/OutputId.sol:L32

```
return keccak256(abi.encodePacked(_txBytes, _outputIndex));
```

- ExitId.sol

code/plasma_framework/contracts/src/exits/utils/ExitId.sol:L41

```
bytes32 hashData = keccak256(abi.encodePacked(_txBytes, _utxoPos.value));
```

code/plasma_framework/contracts/src/exits/utils/ExitId.sol:L54

```
return uint160((uint256(keccak256(_txBytes)) >> 105).setBit(151));
```

- TxFinalizationVerifier.sol

code/plasma_framework/contracts/src/exits/utils/TxFinalizationVerifier.sol:L55

```
bytes32 leafData = keccak256(data.txBytes);
```

Other methods that are affected because they rely on the return values of these methods:

- `ExitId.sol`
  - getStandardExitId
  - getInFlightExitId
- `OutputId.sol`
  - computeDepositOutputId
  - computeNormalOutputId
- `PaymentChallengeIFENotCanonical.sol`
  - verifyAndDeterminePositionOfTransactionIncludedInBlock
  - verifyCompetingTxFinalized
- `PaymentChallengeStandardExit.sol`
  - verifyChallengeTxProtocolFinalized
- `PaymentStartInFlightExit.sol`
  - verifyInputTransactionIsStandardFinalized
- `PaymentExitGame.sol`
  - getStandardExitId
  - getInFlightExitId
- `PaymentOutputToPaymentTxCondition.sol`
  - verify

Recommendation

Enforce strict-length decoding for `txBytes`, and specify that `uint` is decoded from a 32-byte short string.

Enforcing a 32-byte length for `uint` means that `0x1234` should always be encoded as:

`0xa0000000000000000000000000000000000000000000000000000000000001234`

- `0xa0` represents the tag + the length: `0x80 + 32`

- `0000000000000000000000000000000000000000000000000000000000001234` is the number 32 bytes long with leading zeroes

Unfortunately, using leading zeroes is against the RLP spec:

https://github.com/ethereum/wiki/wiki/RLP

> *positive RLP integers must be represented in big endian binary form with no leading zeroes*

This means that libraries interacting with OMG contracts which are going to correctly and fully implement the spec will generate "incorrect" encodings for uints; encodings that are not going to be recognized by the OMG contracts.

Fully correct spec encoding: `0x821234`. Proposed encoding in this solution: `0xa0000000000000000000000000000000000000000000000000000000000001234`.

Similarly enforce restrictions where they can be added; this is possible because of the strict structure format that needs to be encoded.

Some other potential solutions are included below. Note that these solutions are not recommended for reasons included below:

1. Normalize the encoding that gets passed to methods that hash the transaction for use as an id:

This can be implemented in the methods that call `keccak256` on `txBytes` and should decode and re-encode the passed `txBytes` in order to normalize the passed encoding.

- a `txBytes` is passed
- the txBytes are decoded into structure: `tmpDecodedStruct = decode(txBytes)`
- the `tmpDecodedStruct` is re-encoded in order to normalize it: `normalizedTxBytes = encode(txBytes)`

This method is not recommended because it needs a Solidity encoder to be implemented and a lot of gas will be used to decode and re-encode the initial `txBytes`.

1. Correctly and fully implement RLP decoding

This is another solution that adds a lot of code and is prone to errors.

The solution would be to enforce all of the restrictions when decoding and not accept any encoding that doesn't fully follow the spec. This for example means that is should not accept uints with leading zeroes.

This is a problem because it needs a lot of code that is not easy to write in Solidity (or EVM).

5.4 Recommendation: Remove `TxFinalizationModel` and `TxFinalizationVerifier`. Implement stronger checks in `Merkle` Medium

Resolution
This was partially addressed in omisego/plasma-contracts#503, with the removal of several unneeded branches of logic in `TxFinalizationModel` (now renamed to `MoreVpFinalization`). A subsequent change in omisego/plasma-contracts#533 added a non-zero proof length check in `Merkle`. Note that `PaymentChallengeIFENotCanonical.respond` still calls `Merkle.checkMembership` directly, and lacks the typical transaction type protocol check made in `MoreVpFinalization.isStandardFinalized`.

Description

`TxFinalizationVerifier` is an abstraction around the block inclusion check needed for many of the features of plasma exit games. It uses a struct defined in `TxFinalizationModel` as inputs to its two functions: `isStandardFinalized` and `isProtocolFinalized`.

`isStandardFinalized` returns the result of an inclusion proof. Although there are several branches, only the first is used:

code/plasma_framework/contracts/src/exits/utils/TxFinalizationVerifier.sol:L19-L32

```
/**
* @notice Checks whether a transaction is "standard finalized"
* @dev MVP: requires that both inclusion proof and confirm signature is checke
* @dev MoreVp: checks inclusion proof only
*/
```

```
function isStandardFinalized(Model.Data memory data) public view returns (bool
    if (data.protocol == Protocol.MORE_VP()) {
        return checkInclusionProof(data);
    } else if (data.protocol == Protocol.MVP()) {
        revert("MVP is not yet supported");
    } else {
        revert("Invalid protocol value");
    }
}
```

`isProtocolFinalized` is unused:

code/plasma_framework/contracts/src/exits/utils/TxFinalizationVerifier.sol:L34-L47

```
/**
* @notice Checks whether a transaction is "protocol finalized"
* @dev MVP: must be standard finalized
* @dev MoreVp: allows in-flight tx, so only checks for the existence of the t
*/
function isProtocolFinalized(Model.Data memory data) public view returns (bool
    if (data.protocol == Protocol.MORE_VP()) {
        return data.txBytes.length > 0;
    } else if (data.protocol == Protocol.MVP()) {
        revert("MVP is not yet supported");
    } else {
        revert("Invalid protocol value");
    }
}
```

The abstraction used introduces branching logic and requires several files to be visited to fully understand the function of each line of code: `ITxFinalizationVerifier`, `TxFinalizationModel`, `TxPosLib`, `Protocol`, `BlockController`, and `Merkle`. Additionally, the abstraction obfuscates the underlying inclusion proof primitive when used in the exit game contracts. `isStandardFinalized` is not clearly an inclusion proof, and `isProtocolFinalized` simply adds confusion.

Finally, the abstraction may have ramifications on the safety of `Merkle.sol`. As it stands now, `Merkle.checkMembership` should never be called directly by the exit game

controllers, as it lacks an important check made in `TxFinalizationVerifier.checkInclusionProof`:

code/plasma_framework/contracts/src/exits/utils/TxFinalizationVerifier.sol:L49-L59

```
function checkInclusionProof(Model.Data memory data) private view returns (boo
    if (data.inclusionProof.length == 0) {
        return false;
    }

    (bytes32 root,) = data.framework.blocks(data.txPos.blockNum());
    bytes32 leafData = keccak256(data.txBytes);
    return Merkle.checkMembership(
        leafData, data.txPos.txIndex(), root, data.inclusionProof
    );
}
```

By introducing the abstraction of `TxFinalizationVerifier`, the input validation performed by `Merkle` is split across multiple files, and the reasonable-seeming decision of calling `Merkle.checkMembership` directly becomes unsafe. In fact, this occurs in one location in the contracts:

**code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFENot
L204**

```
function verifyAndDeterminePositionOfTransactionIncludedInBlock(
    bytes memory txbytes,
    UtxoPosLib.UtxoPos memory utxoPos,
    bytes32 root,
    bytes memory inclusionProof
)
    private
    pure
    returns(uint256)
{
    bytes32 leaf = keccak256(txbytes);
    require(
        Merkle.checkMembership(leaf, utxoPos.txIndex(), root, inclusionProof),
        "Transaction is not included in block of Plasma chain"
```

```
        );

        return utxoPos.value;
    }
```

Recommendation

1. Remove `TxFinalizationVerifier` and `TxFinalizationModel`
2. Implement a proof length check in `Merkle.sol`
3. Call `Merkle.checkMembership` directly from exit controller contracts:

- `PaymentChallengeIFEOutputSpent.verifyInFlightTransactionStandardFinalized`:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFEOut

```
require(controller.txFinalizationVerifier.isStandardFinalized(finalizationData
```

- `PaymentChallengeIFENotCanonical.verifyCompetingTxFinalized`:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFENot

```
require(self.txFinalizationVerifier.isStandardFinalized(finalizationData), "Fa
```

- `PaymentStartInFlightExit.verifyInputTransactionIsStandardFinalized`:

**code/plasma_framework/contracts/src/exits/payment/controllers/PaymentStartInFlightExit.
L308**

```
require(exitData.controller.txFinalizationVerifier.isStandardFinalized(finaliz
        "Input transaction is not standard finalized");
```

1. If none of the above recommendations are implemented, ensure that `PaymentChallengeIFENotCanonical` uses the abstraction `TxFinalizationVerifier` so that a length check is performed on the inclusion proof.

Corresponding issue in plasma-contracts repo: https://github.com/omisego/plasma-contracts/issues/471

5.5 Merkle - The implementation does not enforce inclusion of leaf nodes. Medium ✓ Addressed

Resolution
This was addressed in omisego/plasma-contracts#452 with the addition of leaf and node salts to the `checkMembership` function.

Description

A observation with the current Merkle tree implementation is that it may be possible to validate nodes other than leaves. This is done by providing `checkMembership` with a reference to a hash within the tree, rather than a leaf.

code/plasma_framework/contracts/src/utils/Merkle.sol:L9-L42

```
/**
 * @notice Checks that a leaf hash is contained in a root hash
 * @param leaf Leaf hash to verify
 * @param index Position of the leaf hash in the Merkle tree
 * @param rootHash Root of the Merkle tree
 * @param proof A Merkle proof demonstrating membership of the leaf hash
 * @return True, if the leaf hash is in the Merkle tree; otherwise, False
*/
function checkMembership(bytes32 leaf, uint256 index, bytes32 rootHash, bytes
    internal
    pure
    returns (bool)
{
    require(proof.length % 32 == 0, "Length of Merkle proof must be a multiple

    bytes32 proofElement;
    bytes32 computedHash = leaf;
    uint256 j = index;
    // Note: We're skipping the first 32 bytes of `proof`, which holds the si
    for (uint256 i = 32; i <= proof.length; i += 32) {
        // solhint-disable-next-line no-inline-assembly
        assembly {
```

```
                proofElement := mload(add(proof, i))
            }
            if (j % 2 == 0) {
                computedHash = keccak256(abi.encodePacked(computedHash, proofEleme
            } else {
                computedHash = keccak256(abi.encodePacked(proofElement, computedHa
            }
            j = j / 2;
        }

        return computedHash == rootHash;
    }
```

The current implementation will validate the provided "leaf" and return `true`. This is a known problem of Merkle trees https://en.wikipedia.org/wiki/Merkle_tree#Second_preimage_attack.

Examples

Provide a hash from within the Merkle tree as the `leaf` argument. The index has to match the index of that node in regards to its current level in the tree. The `rootHash` has to be the correct Merkle tree `rootHash`. The proof has to skip the necessary number of levels because the nodes "underneath" the provided "leaf" will not be processed.

Recommendation

A remediation needs a fixed Merkle tree size as well as the addition of a byte prepended to each node in the tree. Another way would be to create a structure for the Merkle node and mark it as `leaf` or `no leaf`.

Corresponding issue in plasma-contracts repo: https://github.com/omisego/plasma-contracts/issues/425

5.6 Maintainer can bypass exit game quarantine by registering not-yet-deployed contracts Medium ✓ Addressed

Resolution

This was addressed in commit 7669076be1dff47473ee877dcebef5989d7617ac by adding a check that registered contracts had nonzero `extcodesize`.

Description

The plasma framework uses an `ExitGameRegistry` to allow the maintainer to add new exit games after deployment. An exit game is any arbitrary contract. In order to prevent the maintainer from adding malicious exit games that steal user funds, the framework uses a "quarantine" system whereby newly-registered exit games have restricted permissions until their quarantine period has expired. The quarantine period is by default `3 * minExitPeriod`, and is intended to facilitate auditing of the new exit game's functionality by the plasma users.

However, by registering an exit game at a contract which has not yet been deployed, the maintainer can prevent plasma users from auditing the game until the quarantine period has expired. After the quarantine period has expired, the maintainer can deploy the malicious exit game and immediately steal funds.

Explanation

Exit games are registered in the following function, callable only by the plasma contract maintainer:

code/plasma_framework/contracts/src/framework/registries/ExitGameRegistry.sol:L58-L78

```
/**
 * @notice Registers an exit game within the PlasmaFramework. Only the mainta
 * @dev Emits ExitGameRegistered event to notify clients
 * @param _txType The tx type where the exit game wants to register
 * @param _contract Address of the exit game contract
 * @param _protocol The transaction protocol, either 1 for MVP or 2 for MoreVP
 */
function registerExitGame(uint256 _txType, address _contract, uint8 _protocol)
    require(_txType != 0, "Should not register with tx type 0");
    require(_contract != address(0), "Should not register with an empty exit g
    require(_exitGames[_txType] == address(0), "The tx type is already registe
    require(_exitGameToTxType[_contract] == 0, "The exit game contract is alre
```

```
        require(Protocol.isValidProtocol(_protocol), "Invalid protocol value");

        _exitGames[_txType] = _contract;
        _exitGameToTxType[_contract] = _txType;
        _protocols[_txType] = _protocol;
        _exitGameQuarantine.quarantine(_contract);

        emit ExitGameRegistered(_txType, _contract, _protocol);
    }
```

Notably, the function does not check the `extcodesize` of the submitted contract. As such, the maintainer can submit the address of a contract which does not yet exist and is not auditable.

After at least `3 * minExitPeriod` seconds pass, the submitted contract now has full permissions as a registered exit game and can pass all checks using the `onlyFromNonQuarantinedExitGame` modifier:

code/plasma_framework/contracts/src/framework/registries/ExitGameRegistry.sol:L33-L40

```
    /**
     * @notice A modifier to verify that the call is from a non-quarantined exit g
     */
    modifier onlyFromNonQuarantinedExitGame() {
        require(_exitGameToTxType[msg.sender] != 0, "The call is not from a regist
        require(!_exitGameQuarantine.isQuarantined(msg.sender), "ExitGame is quara
        _;
    }
```

Additionally, the submitted contract passes checks made by external contracts using the `isExitGameSafeToUse` function:

code/plasma_framework/contracts/src/framework/registries/ExitGameRegistry.sol:L48-L56

```
    /**
     * @notice Checks whether the contract is safe to use and is not under quarant
     * @dev Exposes information about exit games quarantine
```

```
 * @param _contract Address of the exit game contract
 * @return boolean Whether the contract is safe to use and is not under quara
 */
function isExitGameSafeToUse(address _contract) public view returns (bool) {
    return _exitGameToTxType[_contract] != 0 && !_exitGameQuarantine.isQuarant
}
```

These permissions allow a registered quarantine to:

1. Withdraw any users' tokens from `ERC20Vault`:

code/plasma_framework/contracts/src/vaults/Erc20Vault.sol:L52-L55

```
function withdraw(address payable receiver, address token, uint256 amount) ext
    IERC20(token).safeTransfer(receiver, amount);
    emit Erc20Withdrawn(receiver, token, amount);
}
```

1. Withdraw any users' ETH from `EthVault`:

code/plasma_framework/contracts/src/vaults/EthVault.sol:L46-L54

```
function withdraw(address payable receiver, uint256 amount) external onlyFromN
    // we do not want to block exit queue if transfer is unucessful
    // solhint-disable-next-line avoid-call-value
    (bool success, ) = receiver.call.value(amount)("");
    if (success) {
        emit EthWithdrawn(receiver, amount);
    } else {
        emit WithdrawFailed(receiver, amount);
    }
```

1. Activate and deactivate the `ExitGameController` reentrancy mutex:

code/plasma_framework/contracts/src/framework/ExitGameController.sol:L63-L66

```
function activateNonReentrant() external onlyFromNonQuarantinedExitGame() {
    require(!mutex, "Reentrant call");
```

```
    mutex = true;
}
```

code/plasma_framework/contracts/src/framework/ExitGameController.sol:L72-L75

```
function deactivateNonReentrant() external onlyFromNonQuarantinedExitGame() {
    require(mutex, "Not locked");
    mutex = false;
}
```

1. `enqueue` arbitrary exits:

code/plasma_framework/contracts/src/framework/ExitGameController.sol:L115-L138

```
function enqueue(
    uint256 vaultId,
    address token,
    uint64 exitableAt,
    TxPosLib.TxPos calldata txPos,
    uint160 exitId,
    IExitProcessor exitProcessor
)
    external
    onlyFromNonQuarantinedExitGame
    returns (uint256)
{
    bytes32 key = exitQueueKey(vaultId, token);
    require(hasExitQueue(key), "The queue for the (vaultId, token) pair is not
    PriorityQueue queue = exitsQueues[key];

    uint256 priority = ExitPriority.computePriority(exitableAt, txPos, exitId)

    queue.insert(priority);
    delegations[priority] = exitProcessor;

    emit ExitQueued(exitId, priority);
    return priority;
}
```

1. Flag outputs as “spent”:

code/plasma_framework/contracts/src/framework/ExitGameController.sol:L210-L213

```
function flagOutputSpent(bytes32 _outputId) external onlyFromNonQuarantinedExi
    require(_outputId != bytes32(""), "Should not flag with empty outputId");
    isOutputSpent[_outputId] = true;
}
```

Recommendation

`registerExitGame` should check that `extcodesize` of the submitted contract is non-zero.

Corresponding issue in plasma-contracts repo: https://github.com/omisego/plasma-contracts/issues/410

5.7 EthVault - Unused state variable Minor ✓ Addressed

Resolution
This was addressed in commit ea36f5ff46ab72ec5c281fa0a3dffe3bcc83178b.

Description

The state variable `withdrawEntryCounter` is not used in the code.

code/plasma_framework/contracts/src/vaults/EthVault.sol:L8

```
uint256 private withdrawEntryCounter = 0;
```

Recommendation

Remove it from the contract.

5.8 Recommendation: Add a tree height limit check to `Merkle.sol` Minor

Description

Each plasma block has a maximum of `2 ** 16` transactions, which corresponds to a maximum Merkle tree height of 16. The `Merkle` library currently checks that the proof is comprised of 32-byte segments, but neglects to check the maximum height:

code/plasma_framework/contracts/src/utils/Merkle.sol:L17-L23

```
function checkMembership(bytes32 leaf, uint256 index, bytes32 rootHash, bytes
    internal
    pure
    returns (bool)
{
    require(proof.length % 32 == 0, "Length of Merkle proof must be a multiple
```

Corresponding issue in plasma-contracts repo: https://github.com/omisego/plasma-contracts/issues/467

5.9 Recommendation: remove `IsDeposit` and add a similar getter to `BlockController` Minor ✓ Addressed

Resolution

This was addressed in commit 0fee13f7f084983139eb47636ff785ebea8a1c36 by removing the `IsDeposit` contract and replicating its functionality in `BlockController.sol`.

Description

The `IsDeposit` library is used to check whether a block number is a deposit or not. The logic is simple - if `blockNum % childBlockInterval` is nonzero, the block number is a deposit.

By including this check in `BlockController` instead, the contract can perform an existence check as well. The function in `BlockController` would return the same result as the `IsDeposit` library, but would additionally revert if the block in question does not exist:

```
function isDeposit(uint _blockNum) public view returns (bool) {
  require(blocks[_blockNum].timestamp != 0, "Block does not exist");
  return _blockNum % childBlockInterval != 0;
}
```

Note that this check is made at the cost of an external call. If the check needs to be made multiple times in a transaction, the result should be cached.

Corresponding issue in plasma-contracts repo: https://github.com/omisego/plasma-contracts/issues/466

5.10 Recommendation: Merge `TxPosLib` into `UtxoPosLib` and implement a `decode` function with range checks. Minor

Resolution

This was partially addressed in omisego/plasma-contracts#515 with the merging of `TxPosLib` and `UtxoPosLib` into `PosLib`. A subsequent change in omisego/plasma-contracts#533 implemented stricter range checks for block number and transaction index. Note that the maximum output index in `PosLib` is still 9999, well above the currently-supported maximum of "3". Additionally, `PosLib.encode` lacks an explicit range check on `txIndex` and `PosLib.decode` lacks an explicit range check on `outputIndex`.

Description

`TxPosLib` and `UtxoPosLib` serve very similar functions. They both provide utility functions to access the block number and tx index of a packed utxo position variable. `UtxoPosLib`, additionally, provides a function to retrieve the output index of a packed utxo position variable.

What they both lack, though, is sanity checks on the values packed inside a utxo position variable. By implementing a function `UtxoPosLib.decode(uint _utxoPos) returns`

`(UtxoPos)`, each exit controller contract can ensure that the values it is using make logical sense. The `decode` function should check that:

- `txIndex` is between 0 and `2**16`
- `outputIndex` is between 0 and 3

Currently, neither of these restrictions is explicitly enforced. As for `blockNum`, the best check is that it exists in the `PlasmaFramework` contract with a nonzero root. Since `UtxoPosLib` is a pure library, that check is better performed elsewhere (See issue 5.9).

Once implemented, all contracts should avoid casting values directly to the `UtxoPos` struct, in favor of using the `decode` function. Merging the two files will help with this.

Corresponding issue in plasma-contracts repo: https://github.com/omisego/plasma-contracts/issues/465

5.11 Recommendation: Implement additional existence and range checks on inputs and storage reads Minor

Resolution
This was partially addressed in omisego/plasma-contracts#524 and omisego/plasma-contracts#483. Not all recommended checks were included.

Description

Many input validation and storage read checks are made implicitly, rather than explicitly. The following compilation notes each line of code in the exit controller contracts where an additional check should be added.

Examples

1. `PaymentChallengeIFEInputSpent`:

- Check that `inFlightTx` has a nonzero input at the provided index:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFEInpu

```
require(ife.isInputPiggybacked(args.inFlightTxInputIndex), "The indexed input
```

- Check that each transaction is nonzero and is correctly formed:

**code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFEInpu
L101**

```
require(
    keccak256(args.inFlightTx) != keccak256(args.challengingTx),
    "The challenging transaction is the same as the in-flight transaction"
);
```

- Check that resulting `outputId` is nonzero:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFEInpu

```
bytes32 ifeInputOutputId = data.ife.inputs[data.args.inFlightTxInputIndex].out
```

- See issue 5.10:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFEInpu

```
UtxoPosLib.UtxoPos memory utxoPos = UtxoPosLib.UtxoPos(data.args.inputUtxoPos)
```

- See issue 5.9:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFEInpu

```
bytes32 challengingTxInputOutputId = data.controller.isDeposit.test(utxoPos.bl
```

- Check that `inputTx` is nonzero and well-formed:

**code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFEInpu
L128**

```
? OutputId.computeDepositOutputId(data.args.inputTx, utxoPos.outputIndex(), ut
: OutputId.computeNormalOutputId(data.args.inputTx, utxoPos.outputIndex());
```

- Check that output is nonzero:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFEInpu

```
WireTransaction.Output memory output = WireTransaction.getOutput(data.args.cha
```

- See issue 5.10:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFEInpu

```
UtxoPosLib.UtxoPos memory inputUtxoPos = UtxoPosLib.UtxoPos(data.args.inputUtx
```

- Check that `challengingTx` has a nonzero input at provided index:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFEInpu

```
data.args.challengingTxInputIndex,
```

2. `PaymentChallengeIFENotCanonical`**:**

- Check that each transaction is nonzero and is correctly formed:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFENot
L101

```
require(
    keccak256(args.inFlightTx) != keccak256(args.competingTx),
    "The competitor transaction is the same as transaction in-flight"
);
```

- See issue 5.10:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFENot

```
UtxoPosLib.UtxoPos memory inputUtxoPos = UtxoPosLib.UtxoPos(args.inputUtxoPos)
```

- See issue 5.9:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFENot

```
if (self.isDeposit.test(inputUtxoPos.blockNum())) {
```

- Check that `inputTx` is nonzero and well-formed:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFENot L110

```
    outputId = OutputId.computeDepositOutputId(args.inputTx, inputUtxoPos.outp
} else {
    outputId = OutputId.computeNormalOutputId(args.inputTx, inputUtxoPos.outpu
```

- Check that `inFlightTx` has a nonzero input at the provided index:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFENot L113

```
require(outputId == ife.inputs[args.inFlightTxInputIndex].outputId,
        "Provided inputs data does not point to the same outputId from the in-
```

- Check that output is nonzero:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFENot

```
WireTransaction.Output memory output = WireTransaction.getOutput(args.inputTx,
```

- Check that `competingTx` has a nonzero input at provided index:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFENot

```
args.competingTxInputIndex,
```

- Check that resulting position is nonzero:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFENot

```
uint256 competitorPosition = verifyCompetingTxFinalized(self, args, output);
```

- Check that `inFlightTxPos` is nonzero:

**code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFENot
L173**

```
require(
    ife.oldestCompetitorPosition > inFlightTxPos,
    "In-flight transaction must be younger than competitors to respond to non-
```

- See issue 5.10:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFENot

```
UtxoPosLib.UtxoPos memory utxoPos = UtxoPosLib.UtxoPos(inFlightTxPos);
```

- Check that block root is nonzero:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFENot

```
(bytes32 root, ) = self.framework.blocks(utxoPos.blockNum());
```

- Check that `inFlightTx` is nonzero and well-formed:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFENot

```
inFlightTx, utxoPos, root, inFlightTxInclusionProof
```

- See issue 5.10:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFENot

```
UtxoPosLib.UtxoPos memory competingTxUtxoPos = UtxoPosLib.UtxoPos(args.competi
```

3. `PaymentChallengeIFEOutputSpent`:

- Check that `inFlightTx` is nonzero and is well-formed:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFEOut

```
uint160 exitId = ExitId.getInFlightExitId(args.inFlightTx);
```

- See issue 5.10:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFEOut

```
UtxoPosLib.UtxoPos memory utxoPos = UtxoPosLib.UtxoPos(args.outputUtxoPos);
```

- Check that `inFlightTx` has a nonzero output at the provided index:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFEOut L63

```
require(
    ife.isOutputPiggybacked(outputIndex),
    "Output is not piggybacked"
);
```

- Check that bond size is nonzero:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFEOut

```
uint256 piggybackBondSize = ife.outputs[outputIndex].piggybackBondSize;
```

- See issue 5.10:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFEOut

```
UtxoPosLib.UtxoPos memory utxoPos = UtxoPosLib.UtxoPos(args.outputUtxoPos);
```

- See issue 5.10:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFEOut

```
UtxoPosLib.UtxoPos memory utxoPos = UtxoPosLib.UtxoPos(args.outputUtxoPos);
```

- Check that `challengingTx` is nonzero and is well-formed:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFEOut

```
uint256 challengingTxType = WireTransaction.getTransactionType(args.challengi
```

- Check that output is nonzero:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFEOut

```
WireTransaction.Output memory output = WireTransaction.getOutput(args.challeng
```

- Check that `challengingTx` has a nonzero input at provided index:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFEOut

```
args.challengingTxInputIndex,
```

4. `PaymentChallengeStandardExit`**:**

- See issue 5.10:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeStanda

```
UtxoPosLib.UtxoPos memory utxoPos = UtxoPosLib.UtxoPos(data.exitData.utxoPos);
```

- Check that `exitingTx` is nonzero and well-formed:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeStanda

```
.decode(data.args.exitingTx)
```

- Check that output is nonzero:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeStanda
L113

```
PaymentOutputModel.Output memory output = PaymentTransactionModel
    .decode(data.args.exitingTx)
    .outputs[utxoPos.outputIndex()];
```

- Check that `challengeTx` is nonzero and well-formed:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeStanda

```
uint256 challengeTxType = WireTransaction.getTransactionType(data.args.challer
```

- See issue 5.10:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeStanda

```
txPos: TxPosLib.TxPos(data.args.challengeTxPos),
```

- See issue 5.9:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeStanda

```
bytes32 outputId = data.controller.isDeposit.test(utxoPos.blockNum())
```

- Check that `challengeTx` has a nonzero input at provided index:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeStanda

```
args.inputIndex,
```

5. `PaymentPiggybackInFlightExit` **:**

- Check that `inFlightTx` is nonzero and well-formed:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentPiggybackInFligh

```
uint160 exitId = ExitId.getInFlightExitId(args.inFlightTx);
```

- Check that `inFlightTx` has a nonzero input at provided index:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentPiggybackInFligh

```
require(!exit.isInputPiggybacked(args.inputIndex), "Indexed input already pigg
```

- See issue 5.10:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentPiggybackInFligh

```
enqueue(self, withdrawData.token, UtxoPosLib.UtxoPos(exit.position), exitId);
```

- Check that `inFlightTx` is nonzero and is well-formed:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentPiggybackInFligh

```
uint160 exitId = ExitId.getInFlightExitId(args.inFlightTx);
```

- Check that `inFlightTx` has a nonzero output at provided index:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentPiggybackInFligh

```
require(!exit.isOutputPiggybacked(args.outputIndex), "Indexed output already p
```

- See issue 5.10:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentPiggybackInFligh

```
enqueue(self, withdrawData.token, UtxoPosLib.UtxoPos(exit.position), exitId);
```

6. `PaymentStartInFlightExit`:

- Check that `inFlightTx` is nonzero and is well-formed:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentStartInFlightExit.

```
exitData.exitId = ExitId.getInFlightExitId(args.inFlightTx);
```

- Check that the length of `inputTxs` is nonzero:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentStartInFlightExit.

```
exitData.inputTxs = args.inputTxs;
```

- See issue 5.10:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentStartInFlightExit.

```
utxosPos[i] = UtxoPosLib.UtxoPos(inputUtxosPos[i]);
```

- See issue 5.9:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentStartInFlightExit.

```
bool isDepositTx = controller.isDeposit.test(utxoPos[i].blockNum());
```

- Check that each `inputTxs` is nonzero and well-formed:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentStartInFlightExit.
L183

```
outputIds[i] = isDepositTx
    ? OutputId.computeDepositOutputId(inputTxs[i], utxoPos[i].outputIndex(), u
    : OutputId.computeNormalOutputId(inputTxs[i], utxoPos[i].outputIndex());
```

- Check that each output is nonzero:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentStartInFlightExit.

```
WireTransaction.Output memory output = WireTransaction.getOutput(inputTxs[i],
```

- Check that `inFlightTx` has nonzero inputs for all `i`:

**code/plasma_framework/contracts/src/exits/payment/controllers/PaymentStartInFlightExit.
L328**

```
exitData.inFlightTxRaw,
i,
```

- Check that each output is nonzero:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentStartInFlightExit.

```
PaymentOutputModel.Output memory output = exitData.inFlightTx.outputs[i];
```

7. `PaymentStartStandardExit` **:**

- See issue 5.10:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentStartStandardExi

```
UtxoPosLib.UtxoPos memory utxoPos = UtxoPosLib.UtxoPos(args.utxoPos);
```

- Check that output is nonzero:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentStartStandardExi

```
PaymentOutputModel.Output memory output = outputTx.outputs[utxoPos.outputIndex
```

- Check that timestamp is nonzero:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentStartStandardExi

```
(, uint256 blockTimestamp) = controller.framework.blocks(utxoPos.blockNum());
```

Corresponding issue in plasma-contracts repo: https://github.com/omisego/plasma-contracts/issues/463

5.12 Recommendation: Remove optional arguments and clean unused code Minor ✓ Addressed

Resolution
This was addressed in omisego/plasma-contracts#496 and omisego/plasma-contracts#503 with the removal of the output guard handler pattern, the simplification of the tx finalization check via `MoreVpFinalization`, and the removal of various unused function parameters and struct fields.

Description

Several locations in the codebase feature unused arguments, functions, return values, and more. There are two primary reasons to remove these artifacts from the codebase:

1. Mass exits are the primary safeguard against a byzantine operator. The biggest bottleneck of a mass exit is transaction throughput, so plasma rootchain implementations should strive to be as efficient as possible. Many unused features require external calls, memory allocation, unneeded calculation, and more.

2. The contracts are set up to be extensible by way of the addition of new exit games to the system. "Optional" or unimplemented features in current exit games should be removed for simplicity's sake, as they currently make up a large portion of the codebase.

Examples

- Output guard handlers
    - These offer very little utility in the current contracts. The main contract, `PaymentOutputGuardHandler`, has three functions:
        - `isValid` enforces that some "preimage" value passed in via calldata has a length of zero. This could be removed along with the unused "preimage" parameter.
        - `getExitTarget` converts a `bytes20` to `address payable` (with the help of `AddressPayable.sol`). This could be removed in favor of using `AddressPayable` directly where needed.

        - `getConfirmSigAddress` simply returns an empty address. This should be removed wherever used - empty fields should be a rare exception or an error, rather than being injected as unused values into critical functions.
    - The minimal utility offered comes at the price of using an external call to the `OutputGuardHandlerRegistry`, as well as an external call for each of the functions mentioned above. Overall, the existence of output guard handlers adds thousands of gas to the exit process.
    - Referenced contracts: `IOutputGuardHandler`, `OutputGuardModel`, `PaymentOutputGuardHandler`, `OutputGuardHandlerRegistry`

- Payment router arguments
    - Several fields in the exit router structs are marked "optional," and are not used in the contracts. While this is not particularly impactful, it does clutter and confuse the contracts. Many "optional" fields are referenced and passed into functions which do not use them. Of note is the crucially-important signature verification function, `PaymentOutputToPaymentTxCondition.verify`, where `StartExitData.inputSpendingConditionOptionalArgs` resolves to an unnamed parameter:

**code/plasma_framework/contracts/src/exits/payment/controllers/PaymentStartInFlightExit.
L332**

```
bool isSpentByInFlightTx = condition.verify(
    exitData.inputTxs[i],
    exitData.inputUtxosPos[i].outputIndex(),
    exitData.inputUtxosPos[i].txPos().value,
    exitData.inFlightTxRaw,
    i,
    exitData.inFlightTxWitnesses[i],
    exitData.inputSpendingConditionOptionalArgs[i]
);
require(isSpentByInFlightTx, "Spending condition failed");
```

**code/plasma_framework/contracts/src/exits/payment/spendingConditions/PaymentOutputT
L47**

```
function verify(
    bytes calldata inputTxBytes,
    uint16 outputIndex,
    uint256 inputTxPos,
    bytes calldata spendingTxBytes,
    uint16 inputIndex,
    bytes calldata signature,
    bytes calldata /*optionalArgs*/
```

The additional fields clutter the namespace of each struct, confusing the purpose of the other fields. For example, `PaymentInFlightExitRouterArgs.StartExitArgs` features two fields, `inputTxsConfirmSigs` and `inFlightTxsWitnesses`, the former of which is marked "optional". In fact, the `inFlightTxsWitnesses` field ends up containing the signatures passed to the spending condition verifier and `ECDSA` library:

**code/plasma_framework/contracts/src/exits/payment/routers/PaymentInFlightExitRouterAr
L24**

```
/**
* @notice Wraps arguments for startInFlightExit.
* @param inFlightTx RLP encoded in-flight transaction.
* @param inputTxs Transactions that created the inputs to the in-flight transa
* @param inputUtxosPos Utxos that represent in-flight transaction inputs. In t
* @param outputGuardPreimagesForInputs (Optional) Output guard pre-images for
* @param inputTxsInclusionProofs Merkle proofs that show the input-creating tr
* @param inputTxsConfirmSigs (Optional) Confirm signatures for the input txs.
* @param inFlightTxWitnesses Witnesses for in-flight transaction. In the same
* @param inputSpendingConditionOptionalArgs (Optional) Additional args for the
*/
struct StartExitArgs {
    bytes inFlightTx;
    bytes[] inputTxs;
    uint256[] inputUtxosPos;
    bytes[] outputGuardPreimagesForInputs;
    bytes[] inputTxsInclusionProofs;
    bytes[] inputTxsConfirmSigs;
    bytes[] inFlightTxWitnesses;
```

```
    bytes[] inputSpendingConditionOptionalArgs;
}
```

Corresponding issue in plasma-contracts repo: https://github.com/omisego/plasma-contracts/issues/457

5.13 Recommendation: Remove `WireTransaction` and `PaymentOutputModel`. Fold functionality into an extended `PaymentTransactionModel` Minor

Description

RLP decoding is performed on transaction bytes in each of `WireTransaction`, `PaymentOutputModel`, and `PaymentTransactionModel`. The latter is the primary decoding function for transactions, while the former two contracts deal with outputs specifically.

Both `WireTransaction` and `PaymentOutputModel` make use of `RLPReader` to decode transaction objects, and both implement very similar features. Rather than having a codebase with two separate definitions for `struct Output`, `PaymentTransactionModel` should be extended to implement all required functionality.

Examples

- `PaymentTransactionModel` should include three distinct decoding functions:
  - `decodeDepositTx` decodes a deposit transaction, which has no inputs and exactly 1 output.
  - `decodeSpendTx` decodes a spend transaction, which has exactly 4 inputs and 4 outputs.
  - `decodeOutput` decodes an output, which is a long list with 4 fields (`uint`, `address`, `address`, `uint`)

A mock implementation including `decodeSpendTx` and `decodeOutput` is shown here: https://gist.github.com/wadeAlexC/7820c0cd82fd5fdc11a0ad58a84165ae

OmiseGo may want to consider enforcing restrictions on the ordering of empty and nonempty fields here as well.

Corresponding issue in plasma-contracts repo: https://github.com/omisego/plasma-contracts/issues/456

5.14 ECDSA error value is not handled Minor ✓ Addressed

Resolution
This was addressed in commit 32288ccff5b867a7477b4eaf3beb0587a4684d7a by adding a check that the returned value is nonzero.

Description

The OpenZeppelin `ECDSA` library returns `address(0x00)` for many cases with malformed signatures:

contracts/cryptography/ECDSA.sol:L57-L63

```
if (uint256(s) > 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681
    return address(0);
}

if (v != 27 && v != 28) {
    return address(0);
}
```

The `PaymentOutputToPaymentTxCondition` contract does not explicitly handle this case:

**code/plasma_framework/contracts/src/exits/payment/spendingConditions/PaymentOutputT
L68**

```
address payable owner = inputTx.outputs[outputIndex].owner();
require(owner == ECDSA.recover(eip712.hashTx(spendingTx), signature), "Tx in r

return true;
```

Recommendation

Adding a check to handle this case will make it easier to reason about the code.

Corresponding issue in plasma-contracts repo: https://github.com/omisego/plasma-contracts/issues/454

5.15 No existence checks on framework block and timestamp reads

Minor ✓ Addressed

Resolution
This was addressed in commit c5e5a460a2082b809a2c45b2d6a69b738b34937a by adding checks that block root and timestamp reads return nonzero values.

Description

The exit game libraries make several queries to the main `PlasmaFramework` contract where plasma block hashes and timestamps are stored. In multiple locations, the return values of these queries are not checked for existence.

Examples

1. `PaymentStartStandardExit.setupStartStandardExitData`:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentStartStandardExi

```
(, uint256 blockTimestamp) = controller.framework.blocks(utxoPos.blockNum());
```

1. `PaymentChallengeIFENotCanonical.respond`:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentChallengeIFENot

```
(bytes32 root, ) = self.framework.blocks(utxoPos.blockNum());
```

1. `PaymentPiggybackInFlightExit.enqueue`:

code/plasma_framework/contracts/src/exits/payment/controllers/PaymentPiggybackInFligh

```
(, uint256 blockTimestamp) = controller.framework.blocks(utxoPos.blockNum());
```

1. `TxFinalizationVerifier.checkInclusionProof`:

code/plasma_framework/contracts/src/exits/utils/TxFinalizationVerifier.sol:L54

```
(bytes32 root,) = data.framework.blocks(data.txPos.blockNum());
```

Recommendation

Although none of these examples seem exploitable, adding existence checks makes it easier to reason about the code. Each query to `PlasmaFramework.blocks` should be followed with a check that the returned value is nonzero.

Corresponding issue in plasma-contracts repo: https://github.com/omisego/plasma-contracts/issues/463

5.16 BondSize - effectiveUpdateTime should be uint64 Minor

Description

In BondSize, the mechanism to update the size of the bond has a grace period after which the new bond size becomes active.

When updating the bond size, the time is casted as a `uint64` and saved in a `uint128` variable.

code/plasma_framework/contracts/src/exits/utils/BondSize.sol:L24

```
uint128 effectiveUpdateTime;
```

code/plasma_framework/contracts/src/exits/utils/BondSize.sol:L11

```
uint64 constant public WAITING_PERIOD = 2 days;
```

code/plasma_framework/contracts/src/exits/utils/BondSize.sol:L57

```
self.effectiveUpdateTime = uint64(now) + WAITING_PERIOD;
```

There's no need to use a `uint128` to save the time if it never will take up that much space.

Recommendation

Change the type of the `effectiveUpdateTime` to `uint64`.

```
- uint128 effectiveUpdateTime;
+ uint64 effectiveUpdateTime;
```

5.17 `PaymentExitGame` contains several redundant `plasmaFramework` declarations Minor

Description

`PaymentExitGame` inherits from both `PaymentInFlightExitRouter` and `PaymentStandardExitRouter`. All three contracts declare and initialize their own `PlasmaFramework` variable. This pattern can be misleading, and may lead to subtle issues in future versions of the code.

Examples

1. `PaymentExitGame` declaration:

code/plasma_framework/contracts/src/exits/payment/PaymentExitGame.sol:L18

```
PlasmaFramework private plasmaFramework;
```

1. `PaymentInFlightExitRouter` declaration:

code/plasma_framework/contracts/src/exits/payment/routers/PaymentInFlightExitRouter.so

```
PlasmaFramework private framework;
```

1. `PaymentStandardExitRouter` declaration:

code/plasma_framework/contracts/src/exits/payment/routers/PaymentStandardExitRouter.s

```
PlasmaFramework private framework;
```

Each variable is initialized in the corresponding file's constructor.

Recommendation

Introduce an inherited contract common to `PaymentStandardExitRouter` and `PaymentInFlightExitRouter` with the `PlasmaFramework` variable. Make the variable internal so it is visible to inheriting contracts.

5.18 BlockController - inaccurate description of childBlockInterval for submitDepositBlock Minor

Description

The Vault calls `submitDepositBlock` when a user deposits funds into the plasma chain. Each deposit transaction creates one deposit block on the plasma chain. The number of deposit blocks between two child blocks is limited by the `childBlockInterval`. For example, a `childBlockInterval` of `1` would not allow any deposit blocks, a `childBlockInterval` of `2` would allow one deposit block after each child block `[child][optional: deposit][child][optional: deposit]`.

code/plasma_framework/contracts/src/framework/BlockController.sol:L96-L114

```
/**
 * @notice Submits a block for deposit
 * @dev Block number adds 1 per submission; it's possible to have at most 'ch
 * @param _blockRoot Merkle root of the Plasma block
 * @return The deposit block number
 */
function submitDepositBlock(bytes32 _blockRoot) public onlyFromNonQuarantinedV
    require(isChildChainActivated == true, "Child chain has not been activated
    require(nextDeposit < childBlockInterval, "Exceeded limit of deposits per

    uint256 blknum = nextDepositBlock();
    blocks[blknum] = BlockModel.Block({
        root : _blockRoot,
        timestamp : block.timestamp
    });

    nextDeposit++;
    return blknum;
}
```

However, the comment at line 98 mentions the following:

> *[..] it's possible to have at most 'childBlockInterval' deposit blocks between two child chain blocks [..]*

This comment is inaccurate, as a `childBlockInterval` of `1` would not allow deposits at all (Note how `nextDeposit` is always `>=1`).

Remediation

The comment should read: [..] it's possible to have at most 'childBlockInterval -1' deposit blocks between two child chain blocks [..]. Make sure to properly validate inputs for these values when deploying the contract to avoid obvious misconfiguration.

5.19 PlasmaFramework - Can omit inheritance of VaultRegistry

Minor

Description

The contract `PlasmaFramework` inherits `VaultRegistry` even though it does not use any of the methods directly. Also `BlockController` inherits `VaultRegistry` effectively adding all of the needed functionality in there.

Remediation

`PlasmaFramework` does not need to inherit `VaultRegistry`, thus the import and the inheritance can be removed from `PlasmaFramework.sol`.

```
 import "./BlockController.sol";
 import "./ExitGameController.sol";
-import "./registries/VaultRegistry.sol";
 import "./registries/ExitGameRegistry.sol";

-contract PlasmaFramework is VaultRegistry, ExitGameRegistry, ExitGameControl
+contract PlasmaFramework is ExitGameRegistry, ExitGameController, BlockContro
     uint256 public constant CHILD_BLOCK_INTERVAL = 1000;

     /**
```

All tests still pass after removing the inheritance.

5.20 BlockController - maintainer should be the only entity to set new authority Minor ✓ Addressed

Resolution
This was addressed in commit 25c2560e3b2e40ce9a10c40da97c3f79afc2c641 with the removal of the `setAuthority` function.

Description

code/plasma_framework/contracts/src/framework/BlockController.sol:L69-L72

```
function setAuthority(address newAuthority) external onlyFrom(authority) {
    require(newAuthority != address(0), "Authority address cannot be zero");
    authority = newAuthority;
}
```

`deployer` initially sets the account that is allowed to submit new blocks as `authority`. `authority` can then set a new `authority` at will. In a system that is set-up and maintained by a `maintainer` role (multi-sig) that can upgrade certain parts of the system it is unexpected for another role to be able to pass along its permissions. The security specification notes that the `authority` role is only used to submit blocks:

> *Authority: EOA used exclusively to submit plasma block hashes to the root chain. The child chain assumes at deployment that the authority account has nonce zero and no transactions have been sent from it.*

However, **no transactions** might not be possible as `authority` is the only one to `activateChildChain`. Once activated, the child chain cannot be de-activated but the `authority` can change.

elixir-omg#managing-the-operator-address notes the following for `operator` aka `authority`:

> *As a consequence, the operator address must never send any other transactions, if it intends to continue submitting blocks. (Workarounds to this limitation are available, if there's such requirement.)*

Additionally, `setAuthority` should emit an event to allow participants to react to this change in the system and have an audit trial.

Remediation

Remove the `setAuthority` function, or clarify its intended purpose and add an event so it can be detected by users.

Corresponding issue in plasma-contracts repo: https://github.com/omisego/plasma-contracts/issues/403

Appendix 1 - Scope

Our initial review covered the following files:

File Name	
exits/interfaces/IOutputGuardHandler.sol	441f1302e9c56a
exits/interfaces/ISpendingCondition.sol	00c615d91f4b56
exits/interfaces/IStateTransitionVerifier.sol	a8a402a118795
exits/interfaces/ITxFinalizationVerifier.sol	47d1025d9d719
exits/models/OutputGuardModel.sol	46ef116b93bb41
exits/models/TxFinalizationModel.sol	8a5bbd3e8022e
exits/payment/controllers/PaymentChallengeIFEInputSpent.sol	277cac44c58fcc
exits/payment/controllers/PaymentChallengeIFENotCanonical.sol	cddc8ba53ccf99
exits/payment/controllers/PaymentChallengeIFEOutputSpent.sol	a5ce1510088b8
exits/payment/controllers/PaymentChallengeStandardExit.sol	a5a319545934d
exits/payment/controllers/PaymentPiggybackInFlightExit.sol	8eb01f55de028e
exits/payment/controllers/PaymentProcessInFlightExit.sol	6ba4a78b47995
exits/payment/controllers/PaymentProcessStandardExit.sol	20e5f5d30b3787
exits/payment/controllers/PaymentStartInFlightExit.sol	c6c5424ee37c61

File Name	
exits/payment/controllers/PaymentStartStandardExit.sol	4ebe197698627
exits/payment/outputGuardHandlers/PaymentOutputGuardHandler.sol	564e9ea7a3fb4(
exits/payment/PaymentExitDataModel.sol	d1e69011622fe6
exits/payment/PaymentExitGame.sol	f0b6b93c0a89e1
exits/payment/PaymentInFlightExitModelUtils.sol	33d3e5c065be8
exits/payment/PaymentTransactionStateTransitionVerifier.sol	e5cf8acf73b6ad
exits/payment/routers/PaymentInFlightExitRouterArgs.sol	c11e874a9e06fb
exits/payment/routers/PaymentInFlightExitRouter.sol	970fa3e62f1a56
exits/payment/routers/PaymentStandardExitRouterArgs.sol	bf16c27381f8c9
exits/payment/routers/PaymentStandardExitRouter.sol	42806bdfedae95
exits/payment/spendingConditions/PaymentOutputToPaymentTxCondition.sol	03e91d87e21ca
exits/registries/OutputGuardHandlerRegistry.sol	309a123160bbe
exits/registries/SpendingConditionRegistry.sol	3c3d474f0a9fcd
exits/utils/BondSize.sol	5b0d0d28374d8
exits/utils/ExitableTimestamp.sol	43c6aac2ffb2cb
exits/utils/ExitId.sol	7afda23a55bc86
exits/utils/OutputId.sol	92f09840ae6a9b
exits/utils/TxFinalizationVerifier.sol	fe3ed4518d03e0
framework/BlockController.sol	6739cfe1a0ee45
framework/ExitGameController.sol	80368067a6813
framework/interfaces/IExitProcessor.sol	e4c1d8af9e266f
framework/models/BlockModel.sol	b8189e31fa460f
framework/PlasmaFramework.sol	ab2f4972d01ca5
framework/Protocol.sol	19a3df96f1038b
framework/registries/ExitGameRegistry.sol	0f005fbde0fc38a
framework/registries/VaultRegistry.sol	b67f8e7bc05518
framework/utils/ExitPriority.sol	18b26af2160f3b

File Name	
framework/utils/PriorityQueue.sol	122b3e2f81de23
framework/utils/Quarantine.sol	eb3c6ca62779e
transactions/eip712Libs/PaymentEip712Lib.sol	484d1dc077895
transactions/outputs/PaymentOutputModel.sol	2cd78f5327a459
transactions/PaymentTransactionModel.sol	2901a612cba37
transactions/WireTransaction.sol	95919930e6213
utils/AddressPayable.sol	fbe6d6c78e748a
utils/Bits.sol	ecdb86c5001d0e
utils/FailFastReentrancyGuard.sol	af48169f434734
utils/IsDeposit.sol	d6968ebd0091e
utils/Merkle.sol	876dad4fb2edea
utils/OnlyFromAddress.sol	7c2992b12e768
utils/OnlyWithValue.sol	85bf439b5889f9
utils/RLPReader.sol	3fd2f65a4bdc0fc
utils/SafeEthTransfer.sol	056e0166a2e4ef
utils/TxPosLib.sol	e3338d37bdd83
utils/UtxoPosLib.sol	bf056fd54e5a8a
vaults/Erc20Vault.sol	0b71916cd9cef1
vaults/EthVault.sol	3502005fc37019
vaults/Vault.sol	9cf94dbbd859c7
vaults/verifiers/Erc20DepositVerifier.sol	deba9753470bc
vaults/verifiers/EthDepositVerifier.sol	5e53ed549695ec
vaults/verifiers/IErc20DepositVerifier.sol	bd9cc22d1669f8
vaults/verifiers/IEthDepositVerifier.sol	943c3ebddf7f85
vaults/ZeroHashesProvider.sol	6564cf101c4b92

Our subsequent review covered the following files:

File Name	
contracts/src/exits/fee/FeeClaimOutputToPaymentTxCondition.sol	6c
contracts/src/exits/fee/FeeExitGame.sol	17
contracts/src/exits/interfaces/ISpendingCondition.sol	3a
contracts/src/exits/interfaces/IStateTransitionVerifier.sol	a8
contracts/src/exits/payment/controllers/PaymentChallengeIFEInputSpent.sol	ce
contracts/src/exits/payment/controllers/PaymentChallengeIFENotCanonical.sol	33
contracts/src/exits/payment/controllers/PaymentChallengeIFEOutputSpent.sol	74
contracts/src/exits/payment/controllers/PaymentChallengeStandardExit.sol	e8
contracts/src/exits/payment/controllers/PaymentDeleteInFlightExit.sol	10
contracts/src/exits/payment/controllers/PaymentPiggybackInFlightExit.sol	fe7
contracts/src/exits/payment/controllers/PaymentProcessInFlightExit.sol	8c
contracts/src/exits/payment/controllers/PaymentProcessStandardExit.sol	1c
contracts/src/exits/payment/controllers/PaymentStartInFlightExit.sol	31
contracts/src/exits/payment/controllers/PaymentStartStandardExit.sol	19
contracts/src/exits/payment/PaymentExitDataModel.sol	d1
contracts/src/exits/payment/PaymentExitGameArgs.sol	77
contracts/src/exits/payment/PaymentExitGame.sol	93
contracts/src/exits/payment/PaymentInFlightExitModelUtils.sol	eb
contracts/src/exits/payment/PaymentTransactionStateTransitionVerifier.sol	64
contracts/src/exits/payment/routers/PaymentInFlightExitRouterArgs.sol	14
contracts/src/exits/payment/routers/PaymentInFlightExitRouter.sol	21
contracts/src/exits/payment/routers/PaymentStandardExitRouterArgs.sol	eb
contracts/src/exits/payment/routers/PaymentStandardExitRouter.sol	ea
contracts/src/exits/payment/spendingConditions/PaymentOutputToPaymentTxCondition.sol	e4
contracts/src/exits/registries/SpendingConditionRegistry.sol	b9
contracts/src/exits/utils/BondSize.sol	5b
contracts/src/exits/utils/ExitableTimestamp.sol	43

File Name	
contracts/src/exits/utils/ExitId.sol	80
contracts/src/exits/utils/MoreVpFinalization.sol	f25
contracts/src/exits/utils/OutputId.sol	92
contracts/src/framework/BlockController.sol	51
contracts/src/framework/ExitGameController.sol	ce
contracts/src/framework/interfaces/IExitProcessor.sol	e4
contracts/src/framework/models/BlockModel.sol	b8
contracts/src/framework/PlasmaFramework.sol	ab
contracts/src/framework/Protocol.sol	19
contracts/src/framework/registries/ExitGameRegistry.sol	83
contracts/src/framework/registries/VaultRegistry.sol	06
contracts/src/framework/utils/ExitPriority.sol	d6
contracts/src/framework/utils/PriorityQueue.sol	12
contracts/src/framework/utils/Quarantine.sol	eb
contracts/src/transactions/eip712Libs/PaymentEip712Lib.sol	89
contracts/src/transactions/FungibleTokenOutputModel.sol	31
contracts/src/transactions/GenericTransaction.sol	48
contracts/src/transactions/PaymentTransactionModel.sol	f62
contracts/src/utils/Bits.sol	ec
contracts/src/utils/FailFastReentrancyGuard.sol	8c
contracts/src/utils/Merkle.sol	72
contracts/src/utils/OnlyFromAddress.sol	7c
contracts/src/utils/OnlyWithValue.sol	85
contracts/src/utils/PosLib.sol	47
contracts/src/utils/RLPReader.sol	90
contracts/src/utils/SafeEthTransfer.sol	05
contracts/src/vaults/Erc20Vault.sol	59

File Name	
contracts/src/vaults/EthVault.sol	7b
contracts/src/vaults/Vault.sol	7a
contracts/src/vaults/verifiers/Erc20DepositVerifier.sol	3c
contracts/src/vaults/verifiers/EthDepositVerifier.sol	42
contracts/src/vaults/verifiers/IErc20DepositVerifier.sol	bd
contracts/src/vaults/verifiers/IEthDepositVerifier.sol	94

Appendix 2 - Disclosure

ConsenSys Diligence (“CD”) typically receives compensation from one or more clients (the “Clients”) for performing the analysis contained in these reports (the “Reports”). The Reports may be distributed through other means, including via ConsenSys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any Third-Party in any respect, including regarding the bugfree nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any Third-Party by virtue of publishing these Reports.

PURPOSE OF REPORTS The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of Solidity code and only the Solidity code we note as being within the scope of our review within this report. The Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other

areas beyond Solidity that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty.

CD makes the Reports available to parties other than the Clients (i.e., “third parties”) – on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

LINKS TO OTHER WEB SITES FROM THIS WEB SITE You may, through hypertext or other computer links, gain access to web sites operated by persons other than ConsenSys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites’ owners. You agree that ConsenSys and CD are not responsible for the content or operation of such Web sites, and that ConsenSys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that ConsenSys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. ConsenSys and CD assumes no responsibility for the use of third party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

TIMELINESS OF CONTENT The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice. Unless indicated otherwise, by ConsenSys and CD.