L2Labs

Smart Contracts

Security Assessment

February 3rd, 2021

By:
Sheraz Arshad @ CertiK
sheraz.arshad@certik.org

Camden Smallwood @ CertiK
camden.smallwood@certik.org

Dan She @ CertiK
dan.she@certik.org

Disclaimer

CertiK reports are not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. These reports are not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts CertiK to perform a security review.

CertiK Reports do not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

CertiK Reports should not be used in any way to make decisions around investment or involvement with any particular project. These reports in no way provide investment advice, nor should be leveraged as investment advice of any sort.

CertiK Reports represent an extensive auditing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK’s position is that each company and individual are responsible for their own due diligence and continuous security. CertiK’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

What is a CertiK report?

- A document describing in detail an in depth analysis of a particular piece(s) of source code provided to CertiK by a Client.
- An organized collection of testing results, analysis and inferences made about the structure, implementation and overall best practices of a particular piece of source code.
- Representation that a Client of CertiK has completed a round of auditing with the intention to increase the quality of the company/product's IT infrastructure and or source code.

Overview

Project Summary

Project Name	L2labs: ZkSwap - Smart Contracts
Description	Smart contracts portion of the zkSwap repository
Platform	Ethereum; Solidity, Yul
Codebase	GitHub Repository
Commits	1. 1fb223956953eed471a7bb8736cebd26849f703c 2. d70b461d16d2f010e32301d15920d18a405a808d

Audit Summary

Delivery Date	Feb. 03 , 2020
Method of Audit	Static Analysis, Manual Review
Consultants Engaged	3
Timeline	Dec. 14, 2020 - Feb. 03, 2021

Vulnerability Summary

Total Issues	61
Total Critical	4
Total Major	1
Total Medium	1
Total Minor	5
Total Informational	50

Executive Summary

The CertiK team audited the codebase from December 14th, 2020 through January 8th, 2021. The code that was modified was found to be well-written in regards to the original implementation, but multiple issues of varying severity were found. Over the course the audit, we also performed a mathematical verification on the `PlonkCore` contract, as well as an additional audit on L2Labs' Gas Mining contracts.

The Gas Mining codebase was found to be well-written, with the exception of two cases in the `StakingRewards` contract where token-to-eth conversion was found to be valid but inefficient. Refere ce SRW-01 and SRW-02 for more information.

`PlonkCore` provides a verifier for zero knowledge proof based on PLONK with the help of precompiled contracts in EVM. Four libraries or contracts are implemented in PlonkCore.sol: `PairingsBn254` , `TranscriptLibrary` , `Plonk4VerifierWithAccessToDNext` and `VerifierWithDeserialize` .

The proof is read as an array of `uint256` and parsed by `VerifierWithDeserialize` . `Plonk4VerifierWithAccessToDNext` validates parsed proof and generate challenges from inputs by calling `TranscriptLibrary` . Then it calculates polynomial commitments and calls `PairingsBn254` to checks if the elliptic curve pairing holds. If the pairing holds, the proof is valid.

This report is mainly focusing on the implementation and security of the verifier. As an on-chain verifier, PlonkCore checks proofs from off-chain provers. Therefore, the correctness of the prove-verify process is not determined by PlonkCore alone. It also depends on the prover.

Most of terminologies and conventions in this report follow the original PLONK paper.

Files In Scope

ID	Contract	Location
BYT	Bytes.sol	contracts/contracts/Bytes.sol
CON	Config.sol	contracts/contracts/Config.sol
DPF	DeployFactory.sol	contracts/contracts/DeployFactory.sol
EVE	Events.sol	contracts/contracts/Events.sol
GOV	Governance.sol	contracts/contracts/Governance.sol
OPE	Operations.sol	contracts/contracts/Operations.sol
OWN	Ownable.sol	contracts/contracts/Ownable.sol
PTM	PairTokenManager.sol	contracts/contracts/PairTokenManager.sol
PLK	PlonkCore.sol	contracts/contracts/PairTokenManager.sol
PRO	Proxy.sol	contracts/contracts/Proxy.sol
STO	Storage.sol	contracts/contracts/Storage.sol
TKI	TokenInit.sol	contracts/contracts/TokenInit.sol
UGR	UpgradeGatekeeper.sol	contracts/contracts/UpgradeGatekeeper.sol
VFR	Verifier.sol	contracts/contracts/UpgradeGatekeeper.sol
VFE	VerifierExit.sol	contracts/contracts/UpgradeGatekeeper.sol
ZSC	ZkSync.sol	contracts/contracts/ZkSync.sol
ZSB	ZkSyncCommitBlock.sol	contracts/contracts/ZkSyncCommitBlock.sol
ZSE	ZkSyncExit.sol	contracts/contracts/ZkSyncExit.sol
TOK	BasicToken.sol	contracts/BasicToken.sol
OWD	Owned.sol	contracts/Owned.sol
PAU	Pausable.sol	contracts/Pausable.sol
RDR	RewardsDistributionRecipient.sol	contracts/RewardsDistributionRecipient.sol
SRW	StakingRewards.sol	contracts/StakingRewards.sol

PlonkCore Reviews

CERTIK

VerifierWithDeserialize

`VerifierWithDeserialize` generates structed proof from public inputs and serialized proof.

- `serialized_proof[:8]` : wire polynomial commitments $[a]*1$, $[b]*1$, $[c]*1$, and $[d]*1\in \mathbb{G}_1$.
- `serialized_proof[8:10]` : grand product (permutation polynomial comment) $[Z]*1\in \mathbb{G}*1$.
- `serialized_proof[10:18]` : quotient polynomial commitments $[t_1]*1$, $[t_2]*1$, $[t_3]*1$ and $[t_4]*1\in \mathbb{G}_1$.
- `serialized_proof[18:22]` : wire evaluations $\bar{a}$, $\bar{b}$, $\bar{c}$ and $\bar{d}$.
- `serialized_proof[22]` : wire value $\bar{d}_{z\omega}$.
- `serialized_proof[23]` : grand product evaluation $\bar{Z}_{\omega}$.
- `serialized_proof[24]` : quotient polynomial evaluation $\bar{t}$.
- `serialized_proof[25]` : linearization polynomial evaluation $\bar{r}$.
- `serialized_proof[26:29]` : permutation polynomial evaluation $\bar{s}*{\sigma_1}$, $\bar{s}*{\sigma_2}$ and $\bar{s}_{\sigma_3}$.
- `serialized_proof[29:31]` : $[W*\mathcal{z}]*1$.
- `serialized_proof[31:]` : $[W*{\mathcal{z}\omega}]*1$.

Plonk4VerifierWithAccessToDNext

`Plonk4VerifierWithAccessToDNext` applies the PLONK algorithm with gate constraint equation

$$q_a(x)a(x) + q_b(x)b(x) + q_c(x)c(x) + q_d(x)d(x) + q_m(x)a(x)b(x) + q_{const}(x) + q_{d_{next}}d(\omega x) = 0.$$

Differnt from the original PLONK algorithm, it introduces two more terms $q_d(x)d(x)$ and $q*{d*{next}}d(x\omega)$.

- `evaluate_lagrange_poly_out_of_domain` (never called)
- `batch_evaluate_lagrange_poly_out_of_domain`
  - Calculate Lagrange polynomial evaluation $L_i(z) = \frac{(z^n-1)\omega^{i-1}}{(z-w^{i-1})n}$.
- `evaluate_vanishing`
  - Calculate zero polynomial evaluation $Z_H(z) = z^n-1$.
- `verify_at_z`
  - `lhs` = $Z_H(z)\bar{t}$
  - `rhs` = $\bar{r} + \sum w_iL_i(z) - \alpha\bar{Z}\omega(\bar{a}+\beta \bar{s}*{\sigma_1}+\gamma)(\bar{b}+\beta \bar{s}*{\sigma_2}+\gamma)(\bar{c}+\beta \bar{s}*{\sigma_3}+\gamma)(\bar{d}+\gamma) - L_1(z)\alpha^2$
  - Verify the quotient polynomial evaluation by checking `lhs == rhs` .

- `reconstruct_d`
  - $[D]*1 = v([q*{const}]*1 + *bar{a}[q_a]*1 + \bar{b}[q_b]*1 + *bar{c}[q_c]*1 + \bar{d}[q_d]*1 + *bar{a}*bar{b}[q_m]*1 + \bar{d}*{z*omega}[q*{d*{next}}]*1) + A + B + C$
  - $A = v(\alpha (\bar{a}+\beta z+\gamma)(\bar{b}+\beta k_1z+\gamma)(\bar{c}+\beta k_2z+\gamma)(\bar{d}+\beta k_3z+\gamma) + \alpha^2 L_1(z))[Z]_1$
  - $B = - \alpha\beta v\bar{Z}*omega(*bar{a}+*beta*bar{s}*{\sigma_1}+\gamma)(\bar{b}+\beta\bar{s}*{*sigma_2}+*gamma)(*bar{c}+*beta*bar{s}*{\sigma_3}+\gamma)[s*{*sigma_4}]*1$
  - $C = uv^9[Z]_1$
- `verify_commitments`
  - `reconstruct_d` --> $[D]_1$
  - $[F]*1 = [D]*1 + \sum*{i=1}^4z^{i-1}[t_i]*1 + v^2[a]*1 + *v^3[b]*1 + v^4[c]*1 + *v^5[d]*1 + v^6[s*{*sigma_1}]*1 + v^7[s*{*sigma_2}] + *v^8[s*{\sigma_3}] +uv^{10}[d]_1$
  - $[E]*1 = (*bar{t} + v*bar{r} + v^2*bar{a} + *v^3*bar{b} + *v^4*bar{c} + *v^5*bar{d} + *v^6*bar{s}*{\omega_1} + v^7\bar{s}*{*omega_2} + *v^8*bar{s}*{\omega_3} + uv^9\bar{z}*omega + *uv^{10}*bar{d}*{z\omega})[1]_1$
  - Check BN254 pairing:

$$e(z[W_z]_1 + uz\omega[W_{z\omega}]_1 + [F]_1 - [E]_1, [1]_2) \cdot e(-[W_z]_1 - u[W_{z\omega}], [x]_2) == 1$$

- `verify_initial`
  - Generate challenges using `TranscriptLibrary`:
    - permutation challenges β and γ
    - Quotient challenge α
    - Evaluation challenge $\mathcal{Z}$
    - Opening challenge v
    - Multipoint evaluation challenge u
  - `batch_evaluate_lagrange_poly_out_of_domain`
  - `verify_at_z`
- `verify`
  - `verify_initial`
  - `verify_commitments`

TranscriptLibrary

`TranscriptLibrary` applies Fiat-Shamir transform and generates challenges for the verifier. It makes the verification process non-interactive.

- Update transcript

- `state_0` : `keccak256(0 | old_state_0 | old_state_1 | value)`
- `state_1` : `keccak256(1 | old_state_0 | old_state_1 | value)`
- `challenge` : `uint256(keccak256(2 | state_0 | state_1 | counter))`

PairingsBn254

`PairingsBn254` provides operations for the elliptic curve over finite fields.

- Barreto-Naehrig curve $y^2 = x^3 + b$
  - $b = 3$.
  - $q = 36u^4 + 36u^3 + 24u^2 + 6u + 1$ with $u = 4965661367192848881$.
  - $r = 36u^4 + 36u^3 + 18u^2 + 6u + 1$ with $u = 4965661367192848881$.
  - $G_1 = (1, 2)$.
  - $G_2 = (11559732032986387107991004021392285783925812861821192530917403151452391805634i +\
    10857046999023057135944570762232829481370756359578518086990519993285655852781,\
    40823678758634336813322034031454355683168513275934012081057410762141200935 31i +\
    84956539231234314176049732474892724384181905872636001487702806493069581019 30)$
- `pow` : Performing big number modulo exponential calculation by calling the precompiled contract at 0x05 in EVM.
- `point_add_into_dest` and `point_sub_into_dest` : Performing BN256 add calculation by calling the precompiled contract at 0x06 in EVM.
- `point_mul_into_dest` : Performing BN256 scalar multiply calculation by calling the precompiled contract at 0x07 in EVM.
- `pairing` : Performing BN256 pairing by calling the precompiled contract at 0x08 in EVM.

PlonkCore Findings

General

- [INFO] Recommend using `uint256` for all `uint`.

PairingsBn254

- [INFO] `negate`: recommend checking if `self.Y < q_mod`.
- [INFO] `point_sub_into_dest`: recommend checking if `p2.Y < q_mod`.

Plonk4VerifierWithAccessToDNext

- [INFO] `batch_evaluate_lagrange_poly_out_of_domain`: considering function `evaluate_vanishing` has already been implemented, we can call it to simplify the following lines

```
            PairingsBn254.Fr memory one = PairingsBn254.new_fr(1);
            PairingsBn254.Fr memory tmp_1 = PairingsBn254.new_fr(0);
            PairingsBn254.Fr memory tmp_2 = PairingsBn254.new_fr(domain_size);
            PairingsBn254.Fr memory vanishing_at_z = at.pow(domain_size);
            vanishing_at_z.sub_assign(one);
```

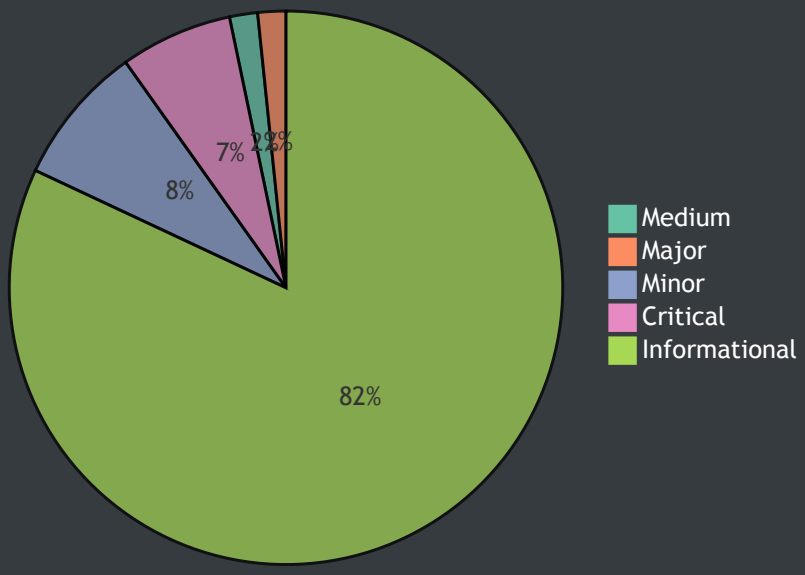
to

```
            PairingsBn254.Fr memory tmp_1 = PairingsBn254.new_fr(0);
            PairingsBn254.Fr memory tmp_2 = PairingsBn254.new_fr(domain_size);
            PairingsBn254.Fr memory vanishing_at_z = evaluate_vanishing(domain_size, at);
```

References

- An Introduction to Pairing-Based Cryptography
- Pairing-Friendly Curves
- Quadratic Arithmetic Programs: from Zero to Hero
- Why and How zk-SNARK Works: Definitive Explanation
- Understanding PLONK
- PlonK: Permutations over Lagrange-bases for Oecumenical Noninteractive Arguments of Knowledge
- Plonk Unrolled for Ethereum
- Constant-Size Commitments to Polynomials and Their Applications

Findings



ID	Title	Type	Severity	Resolved
BYT-01	Unlocked Compiler Version	Language Specific	Informational	
BYT-02	Explicitly returning local variable	Gas Optimization	Informational	
BYT-03	Return Variable Utilization	Gas Optimization	Informational	
BYT-04	Return Variable Utilization	Gas Optimization	Informational	
CON-01	Unlocked Compiler Version	Language Specific	Informational	
EVE-01	Unlocked Compiler Version	Language Specific	Informational	
GOV-01	Unlocked Compiler Version	Language Specific	Informational	
GOV-02	`constructor` with empty body	Language Specific	Informational	

GOV-03	`require` statement will never evaluate to `false`	Dead Code	Informational	
GOV-04	Unnecessary `getTokenAddress` function	Gas Optimization	Informational	
GOV-05	`changeGovernor` does not check if the provided address is non-zero	Volatile Code	Medium	✓
GOV-06	address value is not checked against zero	Volatile Code	Minor	✓
GOV-07	address is not checked against zero value	Logical Issue	Minor	✓
OPE-01	Unlocked Compiler Version	Language Specific	Informational	
OPE-02	Usage of `uint` alias instead of `uint256`	Language Specific	Informational	
OPE-03	Redundant Variable Initialization	Coding Style	Informational	
OPE-04	Redundant Statements	Dead Code	Informational	
OWN-01	Unlocked Compiler Version	Language Specific	Informational	
OWN-02	Function call can be substituted with a `modifier`	Language Specific	Informational	
PTM-01	Unlocked Compiler Version	Language Specific	Informational	
PTM-02	Ineffectual `require` statement	Gas Optimization	Informational	
PTM-03	Explicitly returning a local variable	Gas Optimization	Informational	
PRO-01	Unlocked Compiler Version	Language Specific	Informational	
PRO-02	Inefficient storage read	Gas Optimization	Informational	
PRO-03	Function calls can be substituted with a `modifier`	Language Specific	Informational	
PRO-04	Usage of `uint` alias instead of `uint256`	Language Specific	Informational	
STO-01	Unlocked Compiler Version	Language Specific	Informational	
STO-02	Usage of `uint` alias instead of `uint256`	Language Specific	Informational	
STO-03	Inefficient storage layout	Gas Optimization	Informational	
STO-04	Inefficient storage layout	Gas Optimization	Informational	
STO-05	Documentation discrepancy	Coding Style	Informational	
STO-06	Documentation Discrepancy	Coding Style	Informational	

TKI-01	Unlocked Compiler Version	Language Specific	Informational	↻
TKI-02	Explicitly returning a local variable	Gas Optimization	Informational	↻
UGR-01	Unlocked Compiler Version	Language Specific	Informational	↻
UGR-02	Usage of `uint` alias instead of `uint256`	Language Specific	Informational	↻
UGR-03	Function call can be subsituted with a `modifier`	Language Specific	Informational	↻
UGR-04	`else` clause is not needed	Gas Optimization	Informational	↻
UGR-05	Use of `uint64` as local variable	Gas Optimization	Informational	↻
UGR-06	Inefficient storage read	Gas Optimization	Informational	↻
ZSC-01	Unlocked Compiler Version	Language Specific	Informational	↻
ZSC-02	Redundant Statements	Dead Code	Informational	↻
ZSC-03	Function call can be substituted with a `modifier`	Coding Style	Informational	↻
ZSC-04	No access restriction on function `setGenesisRootAndAddresses`	Logical Issue	Critical	✓
ZSC-05	Unsafe Addition	Arithmetic	Minor	✓
ZSC-06	Unsafe subtraction	Arithmetic	Minor	✓
ZSC-07	Redundant Variable Initialization	Coding Style	Informational	↻
ZSC-08	TODO comments	Coding Style	Informational	↻
ZSC-09	Use of low level `call` without specifying gas	Volatile Code	Minor	✓
ZSB-01	Unlocked Compiler Version	Language Specific	Informational	↻
ZSB-02	Redundant Variable Initialization	Coding Style	Informational	↻
ZSB-03	Function call can be substituted with a `modifier`	Coding Style	Informational	↻
ZSB-04	`else` block is not needed	Coding Style	Informational	↻
ZSB-05	Function call can be substituted with a `modifier`	Coding Style	Informational	↻
ZSB-06	Inefficient storage read	Gas Optimization	Informational	↻
ZSB-07	Inefficient usage of `memory`	Gas Optimization	Informational	↻

ZSE-01	Unlocked Compiler Version	Language Specific	Informational	⏲
ZSE-02	`checkLpL1Balance` also burns tokens	Logical Issue	Critical	✓
ZSE-03	`lpExit` can be called by anyone	Logical Issue	Critical	✓
ZSE-04	`updateBalance` can be called by anyone to increase their token balance	Logical Issue	Critical	✓
ZSE-05	Incorrect code	Logical Issue	Major	✓
SRW-01	Inefficient conversion of token to ETH amount	Arithmetic	Minor	✓
SRW-02	Inefficient conversion of token to ETH amount	Arithmetic	Minor	✓

BYT-01: Unlocked Compiler Version

Type	Severity	Location
Language Specific	Informational	Bytes.sol L1

Description:

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation:

We advise that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.5.0` the contract should contain the following line:

```
pragma solidity 0.5.0;
```

Alleviation:

No alleviations.

BYT-02: Explicitly returning local variable

Type	Severity	Location
Gas Optimization	Informational	Bytes.sol L129

Description:

The function `slice` on the aforementioned line declares and explicitly returns a `bytes memory` local variable, which increases the overall cost of gas.

Recommendation:

Since named return variables can be declared in the signature of a function, consider refactoring to remove the local variable declaration and explicit return statement in order to reduce the overall cost of gas.

Alleviation:

No alleviations.

BYT-03: Return Variable Utilization

Type	Severity	Location
Gas Optimization	Informational	Bytes.sol L271

Description:

The linked function declarations contain explicitly named `return` variables that are not utilized within the function's code block.

Recommendation:

We advise that the linked variables are either utilized or omitted from the declaration.

Alleviation:

No alleviations.

BYT-04: Return Variable Utilization

Type	Severity	Location
Gas Optimization	Informational	Bytes.sol L240

Description:

The linked function declarations contain explicitly named `return` variables that are not utilized within the function's code block.

Recommendation:

We advise that the linked variables are either utilized or omitted from the declaration.

Alleviation:

No alleviations.

CON-01: Unlocked Compiler Version

Type	Severity	Location
Language Specific	Informational	Config.sol L1

Description:

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation:

We advise that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.5.0` the contract should contain the following line:

```
pragma solidity 0.5.0;
```

Alleviation:

No alleviations.

EVE-01: Unlocked Compiler Version

Type	Severity	Location
Language Specific	Informational	Events.sol L1

Description:

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation:

We advise that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.5.0` the contract should contain the following line:

```
pragma solidity 0.5.0;
```

Alleviation:

No alleviations.

GOV-01: Unlocked Compiler Version

Type	Severity	Location
Language Specific	Informational	Governance.sol L1

Description:

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation:

We advise that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.5.0` the contract should contain the following line:

```
pragma solidity 0.5.0;
```

Alleviation:

No alleviations.

GOV-02: `constructor` with empty body

Type	Severity	Location
Language Specific	Informational	Governance.sol L42

Description:

The aforementioned lines declares a constructor with empty body which can be removed to increase legibility of the codebase.

Recommendation:

We advise to remove the empty constructor on the aforementioned line.

Alleviation:

No alleviations.

GOV-03: `require` statement will never evaluate to `false`

Type	Severity	Location
Dead Code	Informational	Governance.sol L118

Description:

The `require` statement on the aforementioned line will never evaluate to `false` as a non-zero tokenId will always be less than `MAX_AMOUNT_OF_REGISTERED_TOKENS` due to the restriction on `L72`. Additionally, the check `tokenId <= MAX_AMOUNT_OF_REGISTERED_TOKENS` can be replaced with `tokenId < MAX_AMOUNT_OF_REGISTERED_TOKENS` as tokenId will never be equal to max value due to check on `L72`.

Recommendation:

We advise to remove the `require` statement on the aforementioned line as it will never evaluate to `false`.

Alleviation:

No alleviations.

GOV-04: Unnecessary `getTokenAddress` function

Type	Severity	Location
Gas Optimization	Informational	Governance.sol L123

Description:

The `getTokenAddress` function in the `Governance` contract is unnecessary, as it only retrieves a value from the public `tokenAddresses` mapping state variable.

Recommendation:

As the `tokenAddresses` state variable is declared as `public`, we recommend removing from the body of the function instead of storing it in a local variable as it is gas efficient:

```
governance.tokenAddresses(tokenId);
```

Alleviation:

No alleviations.

GOV-05: `changeGovernor` does not check if the provided address is non-zero

Type	Severity	Location
Volatile Code	Medium	Governance.sol L59

Description:

The function `changeGovernor` on the aforementioned line receives parameter `_newGovernor` which is set as a new governor of the contract. The function does not perform the check for `_newGovernor` parameter that it is not a zero address. A zero-address set as governor will result in unwanted behaviour of the `Governance` contract.

Recommendation:

We advise to add a `require` check in the body of the function which checks that the `_newGovernor` address is not zero.

```
    require(
       _newGovernor != address(0),
       "zero address is passed as _newGovernor"
    );
```

Alleviation:

Alleviations were applied as of commit hash `d70b461d16d2f010e32301d15920d18a405a808d`.

GOV-06: address value is not checked against zero

Type	Severity	Location
Volatile Code	Minor	Governance.sol L69

Description:

The function `addToken` on aforementioned line recieves _token parameter and it is not checked against zero value of address.

Recommendation:

We advise to check `_token` value on the aformentioned line against `address(0)`.

```
require(
   _token != address(0),
   "address cannot be zero"
);
```

Alleviation:

Alleviations were applied as of commit hash `d70b461d16d2f010e32301d15920d18a405a808d`.

GOV-07: address is not checked against zero value

Type	Severity	Location
Logical Issue	Minor	Governance.sol L85

Description:

The `setValidator` function in the `Governance` contract does not verifiy if its supplied `_validator` parameter is non-zero.

Recommendation:

We advise to check the value of `_validator` against `address(0)` .

```
    require(
       _validator != address(0),
       "_validator cannot be zero"
    );
```

Alleviation:

The team responded with "even if a zero address is used, there is no way to sign a tx with the zero address. no reason to fix." rendering this exhibit ineffectual.

OPE-01: Unlocked Compiler Version

Type	Severity	Location
Language Specific	Informational	Operations.sol L1

Description:

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation:

We advise that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.5.0` the contract should contain the following line:

```
pragma solidity 0.5.0;
```

Alleviation:

No alleviations.

OPE-02: Usage of `uint` alias instead of `uint256`

Type	Severity	Location
Language Specific	Informational	Operations.sol L58, L102, L207

Description:

The aforementioned line use `uint` to declare 256-bit unsigned integers. Although, `uint` is an alias for `uint256` and both represent the same underlying integer allocation. It is advisable that for clean coding practices the complete form uint256 should be used instead of the alias `uint` .

Recommendation:

We advise to use `uint256` instead of alias `uint` on the aforementioned lines.

Alleviation:

No alleviations.

OPE-03: Redundant Variable Initialization

Type	Severity	Location
Coding Style	Informational	Operations.sol L66, L109, L213

Description:

All variable types within Solidity are initialized to their default "empty" value, which is usually their zeroed out representation. Particularly:

- `uint` / `int` : All `uint` and `int` variable types are initialized at `0`
- `address` : All `address` types are initialized to `address(0)`
- `byte` : All `byte` types are initialized to their `byte(0)` representation
- `bool` : All `bool` types are initialized to `false`
- `ContractType` : All contract types (i.e. for a given `contract ERC20 {}` its contract type is `ERC20`) are initialized to their zeroed out address (i.e. for a given `contract ERC20 {}` its default value is `ERC20(address(0))`)
- `struct` : All `struct` types are initialized with all their members zeroed out according to this table

Recommendation:

We advise that the linked initialization statements are removed from the codebase to increase legibility.

Alleviation:

No alleviations.

OPE-04: Redundant Statements

Type	Severity	Location
Dead Code	Informational	Operations.sol L33, L31, L35, L48

Description:

The linked statements do not affect the functionality of the codebase and appear to be either leftovers from test code or older functionality.

Recommendation:

We advise that they are removed to better prepare the code for production environments.

Alleviation:

No alleviations.

OWN-01: Unlocked Compiler Version

Type	Severity	Location
Language Specific	Informational	Ownable.sol L1

Description:

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation:

We advise that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.5.0` the contract should contain the following line:

```
pragma solidity 0.5.0;
```

Alleviation:

No alleviations.

OWN-02: Function call can be substituted with a `modifier`

Type	Severity	Location
Language Specific	Informational	Ownable.sol L44

Description:

The function call on the aforementioned line can be substituted with a `modifier` to increase the legibility of the code.

Recommendation:

We advise to introduce a `modifier` that can be declared in `Ownable` contract and then be used on the aforementioned and also in the `Proxy` contract as mentioned in the exhibit `PRO-03`.

Alleviation:

No alleviations.

PTM-01: Unlocked Compiler Version

Type	Severity	Location
Language Specific	Informational	PairTokenManager.sol L1

Description:

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation:

We advise that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.5.0` the contract should contain the following line:

```
pragma solidity 0.5.0;
```

Alleviation:

No alleviations.

PTM-02: Ineffectual `require` statement

Type	Severity	Location
Gas Optimization	Informational	PairTokenManager.sol L43

Description:

The `require` statement on the aforementioned line will never evaluate to `false` because every non-zero tokenId will always be less than `PAIR_TOKEN_START_ID + MAX_AMOUNT_OF_PAIR_TOKENS` because of the check on `L27`.

Recommendation:

We advise to remove the `require` statement on the aforementioned line as it will never evaluate to `false` and is ineffectual.

Alleviation:

No alleviations.

PTM-03: Explicitly returning a local variable

Type	Severity	Location
Gas Optimization	Informational	PairTokenManager.sol L40

Description:

The function `validatePairToken` on the aforementioned line declares and explicitly returns a `uint16` local variable, which increases the overall cost of gas.

Recommendation:

Since named return variables can be declared in the signature of a function, consider refactoring to remove the local variable declaration and explicit return statement in order to reduce the overall cost of gas.

Alleviation:

No alleviations.

PRO-01: Unlocked Compiler Version

Type	Severity	Location
Language Specific	Informational	Proxy.sol L1

Description:

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation:

We advise that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.5.0` the contract should contain the following line:

```
pragma solidity 0.5.0;
```

Alleviation:

No alleviations.

PRO-02: Inefficient storage read

Type	Severity	Location
Gas Optimization	Informational	Proxy.sol L22

Description:

The aforementioned line performs storage read by calling function `getTarget` and getting a `target` address yet the same target address is available in the local variable `target`. As reading from storage is significantly expensive compared to reading from local variables, we advise that the `target` local variable be used in place of the function call `getTarget()`.

Recommendation:

We advise to use the `target` local variable in place of storage read through function call `getTarget` as former is cheaper in gas cost.

```
(bool initializationSuccess, ) = target.delegatecall(
            abi.encodeWithSignature("initialize(bytes)",
targetInitializationParameters)
);
```

Alleviation:

No alleviations.

PRO-03: Function calls can be substituted with a `modifier`

Type	Severity	Location
Language Specific	Informational	Proxy.sol L60, L116, L123, L130, L136

Description:

The function calls on the aforementioned lines can be substituted with a `modifier` to increase the legibility and quality of the codebase.

Recommendation:

We advise to substitute the function calls with `modifier` on the aforementioned lines.

```
    modifier onlyMaster() {
       requireMaster(msg.sender);
       _;
    }
```

Alleviation:

No alleviations.

PRO-04: Usage of `uint` alias instead of `uint256`

Type	Severity	Location
Language Specific	Informational	Proxy.sol L108, L111

Description:

The aforementioned line use `uint` to declare 256-bit unsigned integers. Although, `uint` is an alias for `uint256` and both represent the same underlying integer allocation. It is advisable that for clean coding practices the complete form uint256 should be used instead of the alias `uint` .

Recommendation:

We advise to use `uint256` instead of alias `uint` on the aforementioned lines.

Alleviation:

No alleviations.

STO-01: Unlocked Compiler Version

Type	Severity	Location
Language Specific	Informational	Storage.sol L1

Description:

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation:

We advise that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.5.0` the contract should contain the following line:

```
pragma solidity 0.5.0;
```

Alleviation:

No alleviations.

STO-02: Usage of `uint` alias instead of `uint256`

Type	Severity	Location
Language Specific	Informational	Storage.sol L1

Description:

The contract uses `uint` to declare 256-bit unsigned integers. Although, `uint` is an alias for `uint256` and both represent the same underlying integer allocation. It is advisable that for clean coding practices the complete form uint256 should be used instead of the alias `uint`.

Recommendation:

We advise to use `uint256` instead of alias `uint` in all of occurrences in the contract.

Alleviation:

No alleviations.

STO-03: Inefficient storage layout

Type	Severity	Location
Gas Optimization	Informational	Storage.sol L18

Description:

The storage variable of type `bool` on the aforementioned line can be tight packed with variable `verifier` on `L25` by placing the aforementioned variable declaration on `L23` which will result in both of the variable occupying only one storage slot.

Recommendation:

We advise to place aforementioned storage variable on `L23` so it could be packed with the variable `verifier`.

```
    /// @notice Flag indicates that upgrade preparation status is active
    /// @dev Will store false in case of not active upgrade mode
    bool public upgradePreparationActive;

    /// @notice Verifier contract. Used to verify block proof and exit proof
    Verifier internal verifier;
```

Alleviation:

No alleviations.

STO-04: Inefficient storage layout

Type	Severity	Location
Gas Optimization	Informational	Storage.sol L94

Description:

The storage variable declared on the aforementioned line can be tight packed with variable `verifier` on `L25` be placing before it. This will result in both of the variables occupying only one storage slot resulting in reduced gas cost.

Recommendation:

We advise that the variable on the aforementioned be moved before variable `verifier` so they can be tight packed. Additionally, if the last exhibit is followed and all three variables are placed next to each other then all of them will occupy only one 32-byte storage slot as their combined size will be 22 bytes.

```
    /// @notice Flag indicates that upgrade preparation status is active
    /// @dev Will store false in case of not active upgrade mode
    bool public upgradePreparationActive;

    /// @notice Flag indicates that exodus (mass exit) mode is triggered
    /// @notice Once it was raised, it can not be cleared again, and all users must
exit
    bool public exodusMode;

    /// @notice Verifier contract. Used to verify block proof and exit proof
    Verifier internal verifier;
```

Alleviation:

No alleviations.

STO-05: Documentation discrepancy

Type	Severity	Location
Coding Style	Informational	Storage.sol L81

Description:

The comment on the aforementioned line describes the struct member `amount` which is actually not specified in the struct definition.

Recommendation:

We advise to remove the comment on the aforementioned line to increase legibility.

Alleviation:

No alleviations.

STO-06: Documentation Discrepancy

Type	Severity	Location
Coding Style	Informational	Storage.sol L59, L61

Description:

The comments on the aforementioned lines describe struct members which are actually not present in the struct definition.

Recommendation:

We advise to remove the comments on the aforementioned line to increase the legibility and quality of the code.

Alleviation:

No alleviations.

TKI-01: Unlocked Compiler Version

Type	Severity	Location
Language Specific	Informational	TokenInit.sol L1

Description:

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation:

We advise that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.6.2` the contract should contain the following line:

```
pragma solidity 0.5.0;
```

Alleviation:

No alleviations.

TKI-02: Explicitly returning a local variable

Type	Severity	Location
Gas Optimization	Informational	TokenInit.sol L4

Description:

The function `getTokens` on the aforementioned line declares and explicitly returns a `address[] memory` local variable, which increases the overall cost of gas.

Recommendation:

Since named return variables can be declared in the signature of a function, consider refactoring to remove the local variable declaration and explicit return statement in order to reduce the overall cost of gas.

Alleviation:

No alleviations.

UGR-01: Unlocked Compiler Version

Type	Severity	Location
Language Specific	Informational	UpgradeGatekeeper.sol L1

Description:

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation:

We advise that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.5.0` the contract should contain the following line:

```
pragma solidity 0.5.0;
```

Alleviation:

No alleviations.

UGR-02: Usage of `uint` alias instead of `uint256`

Type	Severity	Location
Language Specific	Informational	UpgradeGatekeeper.sol L1

Description:

The contract uses `uint` to declare 256-bit unsigned integers. Although, `uint` is an alias for `uint256` and both represent the same underlying integer allocation. It is advisable that for clean coding practices the complete form uint256 should be used instead of the alias `uint`.

Recommendation:

We advise to use `uint256` instead of alias `uint` in all of the ocurrences in the contract.

Alleviation:

No alleviations.

UGR-03: Function call can be subsituted with a `modifier`

Type	Severity	Location
Language Specific	Informational	UpgradeGatekeeper.sol L52, L62, L76, L89, L105

Description:

The function calls on the aforementioned lines can be substituted with a `modifier` to increase the legibility of the codebase.

Recommendation:

We advise to use `modifier` in place of the function calls. The `modifier` declaration mentioned in the exhibit `OWN-02` will be inherited by this contract and can be directly used on the relevant functions.

Alleviation:

No alleviations.

UGR-04: `else` clause is not needed

Type	Severity	Location
Gas Optimization	Informational	UpgradeGatekeeper.sol L97

Description:

The `else` clause on the aforementioned line is not needed and can be replaced with the statement `return false`.

Recommendation:

We advise to remove the `else` clause on the aforementioned line and replace with the statement `return false`.

Alleviation:

No alleviations.

UGR-05: Use of `uint64` as local variable

Type	Severity	Location
Gas Optimization	Informational	UpgradeGatekeeper.sol L111

Description:

The aforementioned line uses `uint64` as local variable which is inefficient as EVM works with 32-byte words and data-packing only happens in the storage. Local variables and variables or stored in memory are not packed. EVM has to do additional work to convert `uint64` to `uint256` and then work with it which results in additional gas cost.

Recommendation:

We advise to change the type of variable `i` from `uint64` to `uint256` it will be cheaper to use.

Alleviation:

No alleviations.

UGR-06: Inefficient storage read

Type	Severity	Location
Gas Optimization	Informational	UpgradeGatekeeper.sol L111

Description:

The aforementioned line performs storage read of `managedContracts.length` on each iteration of the surrounding `for` loop which will cost significant gas if the `for` loop executes a number of times.

Recommendation:

We advise to store the `managedContracts.length` in a local variable and then use it in the `for` loop as reading from local variable will be cheaper compared to repeatedly reading from storage.

```
uint256 length = managedContracts.length;
for (uint64 i = 0; i < length; i++) {...}
```

Alleviation:

No alleviations.

ZSC-01: Unlocked Compiler Version

Type	Severity	Location
Language Specific	Informational	ZkSync.sol L1

Description:

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation:

We advise that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.5.0` the contract should contain the following line:

```
pragma solidity 0.5.0;
```

Alleviation:

No alleviations.

ZSC-02: Redundant Statements

Type	Severity	Location
Dead Code	Informational	ZkSync.sol L27

Description:

The linked statements do not affect the functionality of the codebase and appear to be either leftovers from test code or older functionality.

Recommendation:

We advise that they are removed to better prepare the code for production environments.

Alleviation:

No alleviations.

ZSC-03: Function call can be substituted with a `modifier`

Type	Severity	Location
Coding Style	Informational	ZkSync.sol L31, L52, L249, L266, L319

Description:

The function calls on the aforementioned lines can be substituted with a `modifier` to increase the legibility of the codebase.

Recommendation:

We advise to substitute the function calls on the aforementioned lines with a `modifier`.

```
modifier onlyWhenActive() {
   requireActive();
   _;
}
```

The usage of `modifier` with the function would as followed.

```
function func_name() onlyWhenActive {...}
```

Alleviation:

No alleviations.

ZSC-04: No access restriction on function `setGenesisRootAndAddresses`

Type	Severity	Location
Logical Issue	Critical	ZkSync.sol L142

Description:

The function `setGenesisRootAndAddresses` on the aforementioned line can be called by anyone as it has no access restriction. This enables anyone call this either direct on the implementation or through `proxy` contract and update the storage variables of `zkSyncCommitBlockAddress` and `zkSyncExitAddress` . These storage variables are used to delegate calls to `ZkSyncCommitBlock` and `ZkSyncExit` contracts, respectively. A malicious actor can update these addresses to point a malicous contract and update the storage of both implementation contract or `proxy` contract.

Recommendation:

We advise to either make the aforementioned function `internal` and call it in the `initialize` function with appropriate arguments or move the body of `setGenesisRootAndAddresses` function to `initialize` function.

```
    function initialize(bytes calldata initializationParameters) external {
            initializeReentrancyGuard();

            (
                address _governanceAddress,
             address _verifierAddress,
             address _verifierExitAddress,
          address _pairManagerAddress,
          bytes32 _genesisRoot,
          address _zkSyncCommitBlockAddress,
          address _zkSyncExitAddress
            ) = abi.decode(initializationParameters, (address, address, address, address,
    bytes32, address, address));

            verifier = Verifier(_verifierAddress);
            verifierExit = VerifierExit(_verifierExitAddress);
            governance = Governance(_governanceAddress);
            pairmanager = UniswapV2Factory(_pairManagerAddress);

       setGenesisRootAndAddresses(_genesisRoot, _zkSyncCommitBlockAddress,
    _zkSyncExitAddress);
    }
```

```
    function setGenesisRootAndAddresses(bytes32 _genesisRoot, address
    _zkSyncCommitBlockAddress, address _zkSyncExitAddress) internal {
            blocks[0].stateRoot = _genesisRoot;
            zkSyncCommitBlockAddress = _zkSyncCommitBlockAddress;
            zkSyncExitAddress = _zkSyncExitAddress;
    }
```

Alleviation:

Alleviations were applied as commit hash `d70b461d16d2f010e32301d15920d18a405a808d` by making sure that the aforementioned state variables are not set twice.

ZSC-05: Unsafe Addition

Type	Severity	Location
Arithmetic	Minor	ZkSync.sol L217, L238

Description:

The aforementioned lines perform unsafe addition which can result in overflow of `uint128` value if a large amount is added.

Recommendation:

We advise to use `add` function from `SafeMath` library which will revert the transaction in case of overflow.

Alleviation:

The team respponded that "balanceToWithdraw is gauranteed to not overflow uint128, because processOnchainWithdrawals uses SafeMath when adding to it." rendering this exhibit ineffectual.

ZSC-06: Unsafe subtraction

Type	Severity	Location
Arithmetic	Minor	ZkSync.sol L195

Description:

The aforementioned lines perform unsafe subtraction which can result in underflow of `uint128` if a large amount is subtracted.

Recommendation:

We recommend to use `sub` function from `SafeMath` library which will revert the transaction in case of underflow.

Alleviation:

The team responded with `it's possible to overflow. tolerable risk because deposit assets are limited by governance, and unlikely to see overflows.` rendering this exhibit ineffectual.

ZSC-07: Redundant Variable Initialization

Type	Severity	Location
Coding Style	Informational	ZkSync.sol L196, L201, L270, L278-L280

Description:

All variable types within Solidity are initialized to their default "empty" value, which is usually their zeroed out representation. Particularly:

- `uint` / `int` : All `uint` and `int` variable types are initialized at `0`
- `address` : All `address` types are initialized to `address(0)`
- `byte` : All `byte` types are initialized to their `byte(0)` representation
- `bool` : All `bool` types are initialized to `false`
- `ContractType` : All contract types (i.e. for a given `contract ERC20 {}` its contract type is `ERC20`) are initialized to their zeroed out address (i.e. for a given `contract ERC20 {}` its default value is `ERC20(address(0))`)
- `struct` : All `struct` types are initialized with all their members zeroed out according to this table

Recommendation:

We advise that the linked initialization statements are removed from the codebase to increase legibility.

Alleviation:

No alleviations.

ZSC-08: TODO comments

Type	Severity	Location
Coding Style	Informational	ZkSync.sol L124, 179

Description:

The aforementioned lines contain `TODO` comments which can be removed to increase the quality of codebase for production environment.

Recommendation:

We advise to remove the `TODO` comments from the aforementioned lines.

Alleviation:

No alleviations.

ZSC-09: Use of low level `call` without specifying gas

Type	Severity	Location
Volatile Code	Minor	ZkSync.sol L257

Description:

The aforementioned line uses a low-level `call` function and forwards all the available gas which can enable a malicious actor re-enter the contract to exploit any vulnerability.

Recommendation:

We advise to either use `transfer` method which forwards only `23000` gas which is not enough to re-enter the contract or explicitly specify the gas in the function low enough such that a malicious actor is not able to re-enter the contract.

```
msg.sender.trasnfer(_amount);
```

```
(bool success, ) = msg.sender.call.value(_amount).gas(23000)("");
```

Alleviation:

The team pointed out that the function is marked `non-Rentrant` and hence rentrancy is not possible, rendering this exhibit ineffectual as it was incorrectly identified.

ZSB-01: Unlocked Compiler Version

Type	Severity	Location
Language Specific	Informational	ZkSyncCommitBlock.sol L1

Description:

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation:

We advise that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.5.0` the contract should contain the following line:

```
pragma solidity 0.5.0;
```

Alleviation:

No alleviations.

ZSB-02: Redundant Variable Initialization

Type	Severity	Location
Coding Style	Informational	ZkSyncCommitBlock.sol L200-L202, L208-L209, L264, L419, L97

Description:

All variable types within Solidity are initialized to their default "empty" value, which is usually their zeroed out representation. Particularly:

- `uint` / `int` : All `uint` and `int` variable types are initialized at `0`
- `address` : All `address` types are initialized to `address(0)`
- `byte` : All `byte` types are initialized to their `byte(0)` representation
- `bool` : All `bool` types are initialized to `false`
- `ContractType` : All contract types (i.e. for a given `contract ERC20 {}` its contract type is `ERC20`) are initialized to their zeroed out address (i.e. for a given `contract ERC20 {}` its default value is `ERC20(address(0))`)
- `struct` : All `struct` types are initialized with all their members zeroed out according to this table

Recommendation:

We advise that the linked initialization statements are removed from the codebase to increase legibility.

Alleviation:

No alleviations.

ZSB-03: Function call can be substituted with a `modifier`

Type	Severity	Location
Coding Style	Informational	ZkSyncCommitBlock.sol L42, L71

Description:

The function calls on the aforementioned lines can be substituted with a `modifier` to increase the legibility of the codebase.

Recommendation:

We advise to substitute the function calls on the aforementioned lines with a `modifier`.

```
modifier onlyWhenActive() {
   requireActive();
   _;
}
```

The usage of `modifier` with the function would as followed.

```
function func_name() onlyWhenActive {...}
```

Alleviation:

No alleviations.

ZSB-04: `else` block is not needed

Type	Severity	Location
Coding Style	Informational	ZkSyncCommitBlock.sol L128-L129

Description:

The `else` block on the aforementioned line is redundant and can be replaced with a `return false;` statement.

Recommendation:

We advise to replace the `else` block with a `return false;` statement to increase legibility and quality of the codebase.

Alleviation:

No alleviations.

ZSB-05: Function call can be substituted with a `modifier`

Type	Severity	Location
Coding Style	Informational	ZkSyncCommitBlock.sol L44, L73, L93

Description:

The function calls on the aforementioned lines can be substituted with a `modifier` to increase the legibility of the codebase.

Recommendation:

We advise to substitute the function calls on the aforementioned lines with a `modifier`.

```
modifier onlyValidator() {
   governance.requireActiveValidator(msg.sender);
   _;
}
```

The usage of `modifier` would be as followed.

```
function func_name() onlyValidator {...}
```

Alleviation:

No alleviations.

ZSB-06: Inefficient storage read

Type	Severity	Location
Gas Optimization	Informational	ZkSyncCommitBlock.sol L99

Description:

The aforementioned line reads and uses the storage varible `totalBlocksCommitted`, which can be replaced by the local variable `blocksCommited` as it already stores the same value. The reading from storage is significantly gas costly compared to reading from local variable and hence we suggest that the `totalBlocksCommitted` be replaced with `blocksCommited` on the aforementioned line.

Recommendation:

We advise to use the local variable `blocksCommited` in place of the storage variable `totalBlocksCommitted` on the aforementioned line.

Alleviation:

No alleviations.

ZSB-07: Inefficient usage of `memory`

Type	Severity	Location
Gas Optimization	Informational	ZkSyncCommitBlock.sol L100

Description:

The aforementioned line copies a struct type in a `memory` variable which is inefficient as the read operation on `memory` variable is only performed twice and it will be cheaper to declare the memory location of the variable as `storage`. Reading from `storage` twice will be less costly compared to copying the whole struct in memory and then using it.

Recommendation:

We recommend to change the memory location of the variable declaration from `memory` to `storage` as it will consume less gas.

```
    Block storage revertedBlock = blocks[i];
```

Alleviation:

No alleviations.

ZSE-01: Unlocked Compiler Version

Type	Severity	Location
Language Specific	Informational	ZkSyncExit.sol L1

Description:

The contract has unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Recommendation:

We advise that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version `v0.5.0` the contract should contain the following line:

```
pragma solidity 0.5.0;
```

Alleviation:

No alleviations.

ZSE-02: `checkLpL1Balance` also burns tokens

Type	Severity	Location
Logical Issue	Critical	ZkSyncExit.sol L42

Description:

The function `checkLpL1Balance` on the aforementioned line has a confusing name which sugggests that it only checks the balance yet it also burns the balance. As the function has `public` visibility and it can be called by anyone resulting in unintentional lost of tokens through burning. This function is also called in `lpExit` function and its visibility can be changed to `internal`.

Recommendation:

We advise to the change the visibility of the function on aforementioned line to `internal` so a user does not call it by mistake and unintentionally burn his token balance.

```
function checkLpL1Balance(address pair, uint128 _lpL1Amount) internal {...}
```

Alleviation:

Alleviations were applied as of commit `d70b461d16d2f010e32301d15920d18a405a808d`.

ZSE-03: `lpExit` can be called by anyone

Type	Severity	Location
Logical Issue	Critical	ZkSyncExit.sol L96

Description:

The function on the aforementioned line can be called by anyone where a malicious actor can provide values to the function resulting in loss of funds for the users.

Recommendation:

We advise to restrict the access of function to an authorized entity or introduce a code change where only a `msg.sender` is allowed to successfully call this function and exit.

Alleviation:

The team responded that `lpExit` function is restricted to addressed specified in _addresses[0], which is the address trying to exit associated assets rendering this exhibit ineffectual.

ZSE-04: `updateBalance` can be called by anyone to increase their token balance

Type	Severity	Location
Logical Issue	Critical	ZkSyncExit.sol L36

Description:

The function `updateBalance` on the aforementioned line has `public` visibility and can be called by anyone to increase their token balance. It is also caleld in the function `lpExit` and that seems to be its intended functionality. The visibility of this function can be changed from `public` to `internal` so a malicious actor could not increase its balance by calling it.

Recommendation:

We advise to change the visibility of the aforementioned function from `public` to `internal` .

```
    function updateBalance(uint16 _tokenId, uint128 _out) internal {}
```

Alleviation:

Alleviations were applied as of commit hash `d70b461d16d2f010e32301d15920d18a405a808d` .

ZSE-05: Incorrect code

Type	Severity	Location
Logical Issue	Major	ZkSyncExit.sol L69

Description:

The aforementioned line assigns `address(1)` to `_token1` when the tokenId is 0. It is a wrong assigment as in the case of ETH when tokenId is 0, the address used is `address(0)`.

Recommendation:

We advise to change the assignment on the aforementioned line from `address(1)` to `address(0)`.

```
    _token1 = address(0);
```

Alleviation:

Alleviations were applied as of commit hash `d70b461d16d2f010e32301d15920d18a405a808d`.

SRW-01: Inefficient conversion of token to ETH amount

Type	Severity	Location
Arithmetic	Minor	contracts/StakingRewards.sol L84-L87

Description:

The `withdrawableETH` function in the `StakingRewards` contract calculates the total `amount` of withdrawable ETH using unnecessary extra multiplication and division operations:

```
    return amount.mul(1e27)
                 .div(_totalSupply)
                 .mul(_totalFundETH)
                 .div(1e27);
```

Recommendation:

Consider performing multiplication before division in order to prevent integer truncation and save on the overall cost of gas:

```
    return amount * _totalFundETH / _totalSupply;
```

Alleviation:

The relvant code part was removed as of commit hash `ebae49054e81de66f12a23af9d2cb440258edfc4` rendering this exhibit ineffectual.

SRW-02: Inefficient conversion of token to ETH amount

Type	Severity	Location
Arithmetic	Minor	contracts/StakingRewards.sol L109-L113

Description:

The `_stake` function in the `StakingRewards` contract calculates the `amount` from `amountETH` using unnecessary extra multiplication and division operations:

```
    amount = amountETH
                      .mul(1e27)
                      .mul(_totalSupply)
                      .div(_totalFundETH)
                      .div(1e27);
```

Recommendation:

Consider performing multiplication before division in order to prevent integer truncation and save on the overall cost of gas:

```
    amount = amountETH * _totalSupply / _totalFundETH;
```

Alleviation:

The relvant code part was removed as of commit hash `ebae49054e81de66f12a23af9d2cb440258edfc4` rendering this exhibit ineffectual.

Appendix

Finding Categories

Gas Optimization

Gas Optimization findings refer to exhibits that do not affect the functionality of the code but generate different, more optimal EVM opcodes resulting in a reduction on the total gas cost of a transaction.

Arithmetic

Arithmetic exhibits entail findings that relate to mishandling of math formulas, such as overflows, incorrect operations etc.

Logical Issue

Logical Issue findings are exhibits that detail a fault in the logic of the linked code, such as an incorrect notion on how `block.timestamp` works.

Control Flow

Control Flow findings concern the access control imposed on functions, such as owner-only functions being invoke-able by anyone under certain circumstances.

Volatile Code

Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases that may result in a vulnerability.

Data Flow

Data Flow findings describe faults in the way data is handled at rest and in memory, such as the result of a `struct` assignment operation affecting an in-memory `struct` rather than an in-storage one.

Language Specific

Language Specific findings are issues that would only arise within Solidity, i.e. incorrect usage of `private` or `delete`.

Coding Style

Coding Style findings usually do not affect the generated byte-code and comment on how to make the codebase more legible and as a result easily maintainable.

Inconsistency

Inconsistency findings refer to functions that should seemingly behave similarly yet contain different code, such as a `constructor` assignment imposing different `require` statements on the input variables than a setter function.

Magic Numbers

Magic Number findings refer to numeric literals that are expressed in the codebase in their raw format and should otherwise be specified as `constant` contract variables aiding in their legibility and maintainability.

Compiler Error

Compiler Error findings refer to an error in the structure of the code that renders it impossible to compile using the specified version of the project.

Dead Code

Code that otherwise does not affect the functionality of the codebase and can be safely omitted.