PRIVACY POOLS SMART CONTRACTS SECURITY AUDIT REPORT

OXORIO

MAY 24, 2024

CONTENTS

1 AUDIT OVERVIEW

1.1 PROJECT BRIEF

Title	Description
Client	Privacy Pools
Project name	Privacy Pools v1
Category	Private Transactions
Website	https://privacypools.com/
Repository	https://github.com/ProofOfInnocence/privacy-pools-v1
Initial Commit	e221f0b88e52fb5c214726e765997ef4067793a9
Final Commit	8ab7132877325e27b22053e974b3310d70b860b5
Network	Ethereum
Languages	Solidity, Circom
Lead Auditor	Alexander Mazaletskiy - am@oxor.io
Project Manager	Viktor Mikhailov - viktor@oxor.io

1.2 AUDITED FILES

The following table contains a list of the audited files. The scc tool was used to count the number of lines and assess complexity of the files.

	File	Lines	Blanks	Comments	Code	Complexity
1	contracts/ERC20PrivacyPool.sol	45	5	8	**32**	16%
2	contracts/ETHPrivacyPool.sol	39	4	7	**28**	18%
3	contracts/MerkleTreeWithHistory.sol	145	15	16	**114**	99%
4	contracts/PrivacyPool.sol	138	17	10	**111**	7%
5	membership-proof/circuits/proofOfInnocence.circom	188	29	45	**114**	70%
	Total	**555**	**70**	**86**	**399**	**53%**

Lines: The total number of lines in each file. This provides a quick overview of the file size and its contents.

Blanks: The count of blank lines in the file.

Comments: This column shows the number of lines that are comments.

Code: The count of lines that actually contain executable code. This metric is essential for understanding how much of the file is dedicated to operational elements rather than comments or whitespace.

Complexity: This column shows the file complexity per line of code. It is calculated by dividing the file's total complexity (an approximation of cyclomatic complexity that estimates logical depth and decision points like loops and conditional branches) by the number of executable lines of code. A higher value suggests greater complexity per line, indicating areas with concentrated logic.

1.3 PROJECT OVERVIEW

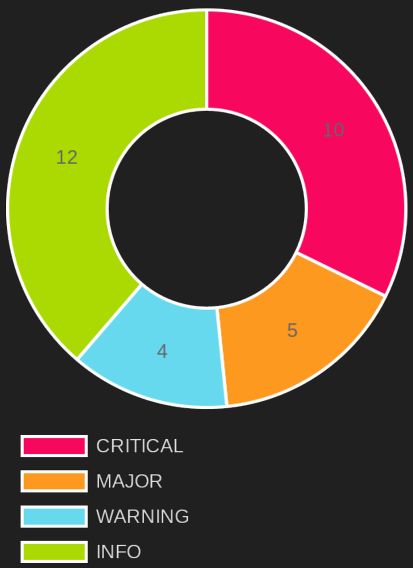
Privacy Pools is an advanced smart contract-based protocol designed to enhance privacy on public blockchains while complying with regulatory frameworks. This project builds on the groundwork laid by Tornado Cash but introduces critical innovations to address specific vulnerabilities associated with earlier privacy protocols. Notably, Tornado Cash facilitated anonymous transactions but struggled with misuse by bad actors, leading to regulatory scrutiny and sanctions. Privacy Pools addresses these issues by allowing users to produce zero-knowledge proofs that demonstrate whether their funds originate from legitimate sources, without revealing their entire transaction history. This mechanism is essential for separating legitimate from non-compliant financial activities.

The architecture of Privacy Pools leverages a novel concept where users can prove membership or exclusion from custom-defined sets of transactions, termed "association sets." These sets are constructed to reflect adherence to diverse regulatory standards or community norms, thus enabling a more nuanced approach to transaction validation. By using zero-knowledge proofs, Privacy Pools ensures that users can verify the legality of their transactions without compromising their privacy. This approach not only enhances user trust and safety but also aligns with global regulatory requirements, providing a sustainable solution to the challenge of maintaining privacy in decentralized financial systems.

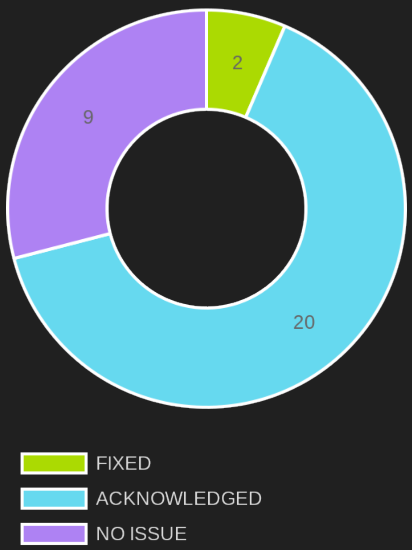
1.4 SUMMARY OF FINDINGS

The table below provides a comprehensive summary of the audit findings, categorizing each by status and severity level. For a detailed description of the severity levels and statuses of findings, see the Findings Classification Reference section.

Severity	TOTAL	NEW	FIXED	ACKNOWLEDGED	NO ISSUE
CRITICAL	10	0	1	3	6
MAJOR	5	0	1	4	0
WARNING	4	0	0	4	0
INFO	12	0	0	9	3
TOTAL	31	0	2	20	9



Issue distribution by severity



Issue distribution by status

This table provides an overview of the findings across the audited files, categorized by severity level. The table enables to quickly identify areas that require immediate attention and prioritize remediation efforts accordingly.

File	TOTAL	CRITICAL	MAJOR	WARNING	INFO
contracts/PrivacyPool.sol	**15**	3	1	3	8
membership-proof/circuits/proofOfInnocence.circom	**12**	8	1	0	3
contracts/ERC20PrivacyPool.sol	**7**	1	2	2	2
contracts/ETHPrivacyPool.sol	**4**	1	0	1	2
contracts/MerkleTreeWithHistory.sol	**4**	1	0	0	3

1.5 CONCLUSION

Despite the identified issues, most of them have been marked as `NO ISSUE` or `ACKNOWLEDGED`.

We would like to emphasize that items `C-02`, `C-04`, and `C-09`, which have been marked as `NO ISSUE`, refer to third-party code where verification takes place. However, the client-provided code is incomplete and not implemented. Most of its functionality is marked as `TODO`. Items `C-03`, `C-05`, `M-02`, `M-03`, `M-04`, and `M-05` also have an `ACKNOWLEDGED` status, but no solutions have been found for them at this time.

It should also be noted that the tests presented in the project do not cover all possible usage scenarios. At the same time, the values of `txMerkleRoot` and `allowedTxRecordsMerkleRoot` used in the tests coincide. This fact suggests insufficient testing of the project code.

In light of the above, we cannot recommend deploying the project to mainnet. We strongly recommend that all of the listed issues, especially those of `CRITICAL` and `MAJOR` severity, be addressed and that the project be re-audited.

2 FINDINGS REPORT

OXORIO

2.1 CRITICAL

C-01	A dishonest prover can manipulate the proof in `IsNum2Bits`
Severity	**CRITICAL**
Status	• FIXED

Location

File	Location	Line
proofOfInnocence.circom	template `IsNum2Bits`	19

Description

In the `IsNum2Bits` circuit, the template should return `1` if the size of the number in bitwise representation is less than `n`, and `0` otherwise. However, due to the lack of constraints, an attacker can manipulate the output value by altering the information in the expression:

```
out[i] <-- (in >> i) & 1;
```

This ultimately gives the attacker the ability to choose the tree in which to prove their transaction, regardless of whether the transaction is classified as "withdrawal" or "deposit".

```
component isDeposit = IsNum2Bits(240);
isDeposit.in <== publicAmount;


checkTxRecordsRoot.in[0] <== isDeposit.isLower*(allowedTxRecordsMerkleRoot -
txRecordsMerkleRoot) + txRecordsMerkleRoot;
```

This finding was discovered by the independent researcher Lev Soukhanov.

Recommendation

We recommend adding constraints on the value passed by the prover to prevent tampering with the final value.

Update

Client's response

Fixed in commit `8ab7132877325e27b22053e974b3310d70b860b5`.

C-02	Possible bypass of validation through invalid input for membership proof
Severity	**CRITICAL**
Status	• NO ISSUE

Location

File	Location	Line
proofOfInnocence.circom	template `Step (PoI)`	120
proofOfInnocence.circom	template `Step (PoI)`	160

Description

It is possible to submit input to the protocol that can bypass all current validations and convince the validator that the membership proof is valid.

For example, the protocol can accept a valid `step_in` but pass the following dataset as input.


```
[
  {
    "txRecordPathElements": [
      "0x14c12f8a123ec67839b093c9e6a7f4429e690f15bb5dda253bb1c57c3e08b708",
      "5499455416245728767758502084886479424423696064367889023130038198639417800794",
      "2600606430905529428182060339305082646353110793508472649153062931542918209074",
      "11224495635916644180335675565949106569141882748352237685396337327907709534945",
      "2399242030534463392142674970266584742013168677609861039634639961298697064915",
      "13182067204896548373877843501261957052850428877096289097123906067079378150834",
      "7106632500398372645836762576259242192202230138343760620842346283595225511823",
      "17857585024203959071818533000506593455576509792639288560876436361491747801924",
      "17278668323652664881420209773995988768195998574629614593395162463145689805534",
      "209436188287252095316293336871467217491997565239632454977424802439169726471",
      "6509061943359659796226067852175931816441223836265895622135845733346450111408",
      "6520190068409764223804901922836901806344965427489260575428680588368698845875",
      "630132332916246842688588724815186106224110394685542623398451000510712069429",
      "9206039871053115294588634296883085884325447977155589548409410370778004490583",
      "18779153312338976659887621628264397623589151531526416856563780429314827231828",
      "17363895409447993644572750069112567504016430687578426059851649285544630344678",
      "19286143925655010470313606340648478511795716951787385659135295148709476458035",
```

```
      "4388129095966650031202114509205416778945646941825467776717149885766859085576",
      "1349852001905976393525917581684850395969195070196158093124360497286644081937",
      "17368552900438243823879346548262086836538128269316713944042307770284532592541",
      "18985880316913054223420119281626297509988828082052104176513061368828929683625",
      "4579408122636409797983594817001151232592120036711857848633386522167572030422",
      "11449953026770181129414594616106672273569253702456243279842657156414101762953"
    ],
    "txRecordPathIndex": 0,
    "accInnocentCommitments": [
      "21663839004416932945382355908790599225266501822907911457504978515578255421292",
      "21663839004416932945382355908790599225266501822907911457504978515578255421292"
    ],
    "isLastStep": 1,
    "txRecordsMerkleRoot":
"14793069448869167018163244265359265004698140403231416118505165104987905632445",
    "allowedTxRecordsMerkleRoot":
"14793069448869167018163244265359265004698140403231416118505165104987905632445",
    "step_in":
"3065659258246033882340361795426780261220662006474317618385752630881570082324",
    "publicAmount": "30000000000000000",
    "outputsStartIndex": 0,
    "inputNullifier": [
      "3217299130395714092375048112053367579529963253380286404242381776442753918543",
      "10338137275517050404818090901831522193036630945261272796204366916420718187042"
    ],
    "inSignature": [
      "5609147781996121559202758336514417651087410452588524961110093556834322927839",
      "7684119861868953362809411662523568231537908702992356876997581542021110158881"
    ],
    "inPublicKey": [
      "4309230837725745671052431107707725589237023471362445678307480723432139544372",
      "4309230837725745671052431107707725589237023471362445678307480723432139544372"
    ],
    "inAmount": ["0", "0"],
    "inBlinding": [
      "155030504778121016616625776637504540871481615842940226898405754575560440661",
      "160684339370557899170855270602669397810112520220487561059681779990351273904"
    ],
    "inPathIndices": [0, 0],
    "outputCommitment": [
      "1",
      "55"
    ]
  }
]
```

The transaction set may consist of only one element (the starting deposit) and is labeled as `isLastStep = 1`, with only `inputNullifiers` in this list of valid data.

Since the element is labeled `isLastStep = 1`, the `txRecord` check in `merkleRoot` is not performed at `#120`.

As the UTXO inputs have `inAmount = 0`, the checks for `accInnocentCommitments` and `merkleProof` are skipped at `#160`.

The constructed `Proof` will be valid for the validator because the validator knows `step_in` and `step_out`, which will be like a hash from `inputNullifiers`.

Recommendation

We recommend revisiting the logic of using the `isLastStep` parameter as well as addressing issues related to ignoring merkle root checks.

Update

Client's response

There is a check in #150 to check if input nullifiers have correct inAmounts, So having the above prove just proves that the nullifiers with amount 0 are member of the inclusion but since they have 0 amount, this can't be from any valid withdrawal.

Oxorio's response:

This finding describes the scenario when the nova recursion consists of only 1 element and `isLastStep = 1`. In terms of the current audited scope, having a call to the special ASP service with specific validations is not a required condition. The main idea of this finding is to initiate a conversation and understand how validation in such a case is expected. We understand that validations can be implemented in the ASP system or in the relayer. However, this is an off-chain part of the protocol and is not included in the audited scope. Additionally, there is a lack of documentation describing how this is expected to work. In the current scope, this finding is valid.

Client's response:

Here is the verification process.

C-03	Limitation of merkle tree for withdrawals in `PrivacyPool`
Severity	**CRITICAL**
Status	• ACKNOWLEDGED

Location

File	Location	Line
PrivacyPool.sol	contract `PrivacyPool` > function `transact`	69
MerkleTreeWithHistory.sol	contract `MerkleTreeWithHistory` > function `_insert`	52

Description

In the `transact` function of the `PrivacyPool` contract, there is an insertion of `2` leaves with every deposit and withdrawal to the pool. The merkle tree has a limitation based on the merkle tree depth. For example, with a merkle tree depth of `20`, the maximal amount of leaves is `2**20 = 1048576`. Considering that with every deposit and withdrawal, there will be an insertion of `2` leaves, there is a maximal amount of deposits equal to `1048576 / 4 = 262144`. If the pool accumulates more than `262144` deposits over time, some users may be unable to withdraw their funds from the pool. Consider the following scenario:

1. A pool is deployed with no initial deposits, and the merkle tree depth level is `2**20`.
2. Over time, users deposit and withdraw their funds, resulting in a total of `262144 + 1` deposits, creating at least `524290` leaves in the merkle tree.
3. Subsequently, all users decide to withdraw funds, requiring `524290` leaves for withdrawals. However, due to limitations, only `1048576 - 524290 = 524286` leaves are available, allowing only `262143` users out of `262145` total deposits to withdraw funds.
4. The funds of the last 2 users will be blocked in the pool forever without the possibility to withdraw, as the `transact` function call will revert in the `_insert` call with the error message `Merkle tree is full. No more leaves can be added`.

It's worth noting that there is no validation on the amount of deposit or withdrawal. Therefore, a hacker can exploit this by calling the `transact` function in a loop with zero amounts, creating an enormous quantity of leaves in the merkle root. This could potentially block funds of users in the protocol, especially on a blockchain with low gas fees.

Recommendation

We recommend adding a minimal deposit amount, defining a maximal quantity of deposits, and reviewing the case when the merkle tree is full.

Update

Client's response

We will add a new function `escapeWithdraw` which will not add anything to merkle tree.

C-04	User de-anonymization risk
Severity	**CRITICAL**
Status	• NO ISSUE

Location

File	Location	Line
proofOfInnocence.circom	template `Step (PoI)`	-
PrivacyPool.sol	`PrivacyPool`	39

Description

In the protocol, the lack of on-chain verification requires the proving party to prove the validity of its input data to establish innocence. The process of proving innocence involves revealing the `step_in` value, which includes `accInnocentCommitments`, potentially exposing the user's identity through traceable commitment transactions.

Identity disclosure occurs as follows: the verifier can take all emitted `NewCommitment` events from the `PrivacyPool` contract and compute `poseidon(commitment, commitmentIndex)` to find a hash that matches the value provided by the prover. This way, the funds deposit transaction will be discovered.

Another disclosure is possible if the user requests an allow list through a service, as the circuit checks the validity of nullifiers, essentially disclosing that the requester has knowledge of `publicKey` and `blinding`. By doing so, the user is revealing themselves.

Recommendation

We recommend revising the protocol logic to ensure user privacy, as this is one of the main ideas behind the protocol.

Update

Client's response

The first `step_in`'s `accInnocentCommitments` will always be `zeroValue`, `zeroValue`. There is no check for this but this check will be done in the verification. We already do it in relayer.

C-05	Unverified transaction inclusion in `Step (PoI)`
Severity	**CRITICAL**
Status	• ACKNOWLEDGED

Location

File	Location	Line
[proofOfInnocence.circom]()	template `Step (PoI)`	117

Description

In the `Step (PoI)` template, the absence of explicit verification for transaction inclusion in both the `txRecordsMerkleRoot` and `allowedTxRecordsMerkleRoot` compromises the verification process. Although it checks whether the transaction is in either `txRecordsMerkleRoot` or `allowedTxRecordsMerkleRoot`, an attacker can specify a transaction that is in only one of the tries.

Recommendation

We recommend adding explicit checks to verify transaction inclusion in both tries to ensure the integrity of the verification process.

Update

Client's response

While it is possible that malicious `ASP` could provide a Merkle tree which is not a subset of a main Merkle tree, this seems to be out of scope of the protocol, as this both a self-evident malicious behavior, and doesn't allow anything than just using an normal `ASP` that allows every transaction.

Only potentially worrying scenario is a situation in which somehow the whole Merkle tree of an `ASP` is not checked, leading to covert bypass of the requirements.

C-06	Lack of nullifier uniqueness check in `Step (PoI)`
Severity	**CRITICAL**
Status	• NO ISSUE

Location

File	Location	Line
proofOfInnocence.circom	template `Step (PoI)`	32

Description

The `Step (PoI)` template lacks a nullifier uniqueness check. This allows an attacker to specify identical nullifiers (and hence commitments) as inputs. A corresponding check is present in the `Transaction` template to prevent this scenario.

For example, let's consider a scenario:

Since there is no check for the inclusion of `txRecord` in Merkle tries on the last step (when `isLastSpet == 1`), the attacker can take advantage of this and pass two identical valid commitments to the input, hence the nullifiers.

The `step_out` in the last step is the hash from the two nullifiers, and since the attacker was able to change one of the nullifiers, they have altered the output parameter that the verifier will use to verify the proof. Such a manipulated proof will be accepted as valid.

Recommendation

We recommend implementing a nullifier uniqueness check within the `Step (PoI)` circuit to prevent the passage of identical nullifiers.

Update

Client's response

We check if nullifiers match like this.

C-07	Missing output commitment validation in `Step (PoI)`
Severity	**CRITICAL**
Status	• NO ISSUE

Location

File	Location	Line
proofOfInnocence.circom	template `Step (PoI)`	32

Description

The `Step (PoI)` template does not validate `outputCommitments`, allowing users to pass any value, including those belonging to others.

For example, let us consider a scenario involving an attacker engaged in a sequence of transactions that incorporate illicit funds. In the proof process, they specify as an `outputCommitment` a transaction from a completely different account with an `inAmount` of `0` in the penultimate step. At the last step, there is no transaction inclusion check, even as an `accInnocentCommitments` check. As a result, a hash of two nullifiers will be returned as a `step_out` associated with a third party commitment.

Recommendation

We recommend introducing checks to validate `outputCommitments` to enhance system soundness.

Update

Client's response

`outputCommitment` is hashed with `txRecord` here, and then it is verified against either `txRecordsMerkleRoot` or `allowedTxRecordsMerkleRoot`.

C-08	Lack of sums validation in `Step (PoI)`
Severity	**CRITICAL**
Status	• NO ISSUE

Location

File	Location	Line
[proofOfInnocence.circom]()	template `Step (PoI)`	32

Description

In the `Step (PoI)` template, the absence of the constraint `sumIns + publicAmount === sumOuts;` compromises transaction integrity. While the `inAmount` is validated against the UTXO, the `publicAmount` is left unconstrained, allowing an attacker to insert arbitrary values into these signals.

For example, consider a scenario where the attacker modifies the `publicAmount` to any chosen value.

The attacker has the ability to set the `publicAmount` to a value less than `240` bits, causing the system to classify the transaction as a deposit. This allows the `txRecordHash` to be checked against the `allowedTxRecordsMerkleRoot` instead of the `txRecordsMerkleRoot`, making it possible for the attacker to prove the current transaction in the wrong set of transactions.

In addition, there is no mechanism in the last step of the recursion to ensure that the transaction is included in the Merkle tries. As a result, the `publicAmount` can be manipulated to any value without the need to prove inclusion in `allowedTxRecordsMerkleRoot`.

Recommendation

We recommend implementing checks to ensure the correctness of amounts.

Update

Client's response

This is an already existing transaction, so it is checked in `TC-Nova` part.

C-09	Partial transaction history acceptance in `Step (PoI)`
Severity	**CRITICAL**
Status	• NO ISSUE

Location

File	Location	Line
proofOfInnocence.circom	template `Step (PoI)`	32

Description

In the `Step (PoI)` template, the prover is given the flexibility to initiate the proof sequence from any transaction within the account history. This provision allows an actor who has deposited illicit funds to selectively start the verification of his transaction history from a mid-point, effectively obfuscating the initial transactions involving the deposit of those illicit funds. As a result, the actor can obscure the origin of the funds from the verifier, thereby claiming innocence in a manner that is contrary to the operational integrity designed into the protocol.

Example:
if the actor's transaction history has a length of `3`, he can start the proof with transaction number `2`, and hide transaction `1`, which was a deposit of illicit funds.

Recommendation

We recommend mandating the provision of a complete transaction history to ensure accountability and traceability within the system.

Update

Client's response

The first `step_in`'s `accInnocentCommitments` will always be `zeroValue`, `zeroValue`. There is no check for this but this check will be done in the verification. We already do it in relayer.

C-10	Fees may exceed the amount being sent in `PrivacyPool`
Severity	**CRITICAL**
Status	• ACKNOWLEDGED

Location

File	Location	Line
PrivacyPool.sol	contract `PrivacyPool` > function `calculatePublicAmount`	76
PrivacyPool.sol	contract `PrivacyPool` > function `calculatePublicAmount`	79
PrivacyPool.sol	contract `PrivacyPool` > function `calculatePublicAmount`	114
ETHPrivacyPool.sol	contract `ETHPrivacyPool.sol` > function `_processDeposit`	23
ETHPrivacyPool.sol	contract `ETHPrivacyPool.sol` > function `_processWithdraw`	35
ERC20PrivacyPool.sol	contract `PrivacyPool` > function `_processDeposit`	29
ERC20PrivacyPool.sol	contract `ERC20PrivacyPool.sol` > function `_processWithdraw`	41

Description

In the function `calculatePublicAmount` of the contract `PrivacyPool`, there is no validation ensuring that `_extData.extAmount` is greater than `_extData.fee`. This omission allows for the execution of deposit logic during withdrawals and vice versa.

For example, an attacker (malicious relayer) can potentially steal user funds during withdrawal processes by manipulating the values of `_extData.extAmount` and `_extData.fee` to achieve the expected `publicAmount`, but by changing the sign of `_extData.extAmount`, which results in the execution of deposit logic instead of withdrawal.

Relayer attack during the withdrawal process:

1. The user wishes to withdraw `0.059` ETH from the pool. According to the circuit logic and smart contract, this amount is reflected as `publicAmount = -0.059`. To ensure the correct operation of the circuit, the value `FIELD_SIZE - publicAmount` is used.

```
function calculatePublicAmount(int256 _extAmount, uint256 _fee) public pure returns
(uint256) {
    //...
```

```
    return (publicAmount >= 0) ? uint256(publicAmount) : FIELD_SIZE - uint256(-publicAmount);
  }
```

1. The user generates a proof.

```
  function verifyProof(Proof memory _args) public view returns (bool) {
    if (_args.inputNullifiers.length == 2) {
      return
        verifier2.verifyProof(
          _args.proof,
          [
            uint256(_args.root),
            _args.publicAmount,
            uint256(_args.extDataHash),
            uint256(_args.inputNullifiers[0]),
            uint256(_args.inputNullifiers[1]),
            uint256(_args.outputCommitments[0]),
            uint256(_args.outputCommitments[1])
          ]
        );
    } else {
      revert("unsupported input count");
    }
  }
```

1. The `calculatePublicAmount` function checks the values of `_extData.extAmount` and `_extData.fee`.

   However, this check does not take into account the possibility that `_extData.fee` may exceed `_extData.extAmount`.

```
  function calculatePublicAmount(int256 _extAmount, uint256 _fee) public pure returns
(uint256) {
    require(_fee < MAX_FEE, "Invalid fee");
    require(_extAmount > -MAX_EXT_AMOUNT && _extAmount < MAX_EXT_AMOUNT, "Invalid ext
amount");
    //...
  }
```

1. In the second step, the attacker(relayer) provides the user with an `extDataHash` generated from `extData`, where the values of `_extData.extAmount = 0` and `_extData.fee = 0.059.`

2. In the `calculatePublicAmount` function, the `publicAmount` value is calculated by subtracting `0.059` from `0`. This results in a negative value of `-0.059`.

```
function calculatePublicAmount(int256 _extAmount, uint256 _fee) public pure returns
(uint256) {
    //...
    int256 publicAmount = _extAmount - int256(_fee);
    //...
}
```

1. The check in the `_transact` function passes successfully.

```
function _transact(Proof memory _args, ExtData memory _extData) internal nonReentrant {
    //...
    require(_args.publicAmount == calculatePublicAmount(_extData.extAmount, _extData.fee),
"Invalid public amount");
    //...
}
```

1. Since the value of `_extData.extAmount` is positive, the attacker(relayer) triggers the `_processDeposit` function to make a deposit.

```
function _processDeposit(ExtData memory _extData) internal override {
    if (_extData.extAmount > 0) {
    require(msg.value == uint256(_extData.extAmount), "Invalid amount");
    require(uint256(_extData.extAmount) <= maximumDepositAmount, "amount is larger than
maximumDepositAmount");
    }
}
```

1. The `_processWithdraw` function fails to execute, even though the user initiated a withdrawal of `0.059` ETH.
2. If the value of `_extData.fee` is positive, the attacker(relayer) receives a commission of `0.059 ETH` to the specified relayer address (which can be any address).

```
function _processWithdraw(Proof memory _args, ExtData memory _extData) internal override {
    //...
    if (_extData.fee > 0) {
      SafeTransferLib.safeTransferETH(_extData.relayer, _extData.fee);
```

```
  }
}
```

1. According to the circuit, the transaction was successfully executed, and the corresponding `outputCommitments` were added to the Merkle Tree.
2. In reality, the attacker(relayer) appropriates the entire amount that the user intended to withdraw.

`Here` is a proof of concept of this attack.

Recommendation

We recommend revisiting these scenarios and adding validation to ensure that the fee does not exceed the payment amount. This can be done by setting a maximum fee amount (`maxFeeAmount`) or by adding other conditions to reduce the fee. It is also advisable to move the `if` operator

```
if (_extData.fee > 0) {
  SafeTransferLib.safeTransferETH(_extData.relayer, _extData.fee);
```

into the statement `if (_extData.extAmount < 0)` on the line 30.

Update

Client's response

The fix has been implemented in the new major version, which is being prepared for release.

2.2 MAJOR

M-01	Deposit amount logic inconsistency
Severity	**MAJOR**
Status	• FIXED

Location

File	Location	Line
proofOfInnocence.circom	template `Step (PoI)`	114

Description

Since in the `PrivacyPool` contract the maximum deposit amount is `2^248 - 1` and the circuit treats amounts above `2^240` as a withdrawal, this provokes treating a transaction with an amount greater than `2^240` but less than `2^248 - 1` as a withdrawal while it is a deposit, which is erroneous logic.

Recommendation

We recommend aligning the deposit amount logic between the contract and the circuit to avoid misinterpretation of transaction types and other logic.

Update

Client's response

Fixed in commit `8ab7132877325e27b22053e974b3310d70b860b5`.

M-02	Replay attack vulnerability in `PrivacyPool`
Severity	**MAJOR**
Status	• ACKNOWLEDGED

Location

File	Location	Line
PrivacyPool.sol		-

Description

In the `PrivacyPool` contract, there is a lack of proof uniqueness validation for each specific pool; the token address is not added to the proof. This allows executing replay attacks across pools with identical states, even with the original pools of the Tornado Cash. Consider the following scenario:

1. There is a deployment of 2 new pools; one pool has `USDC` tokens, another one has `WETH` tokens.
2. Alice is the first depositor to the pool; she initiates a deposit to the `USDC` pool for `100` `USDC` tokens.
3. Alice withdraws her `100` `USDC` from the `USDC` pool.
4. Alice creates a deposit to the `WETH` pool for `100` `WETH`, also like a first depositor.
5. Bob takes the withdrawal proof of Alice from the `USDC` pool and uses it, withdrawing Alice's `WETH`. Even though Bob wouldn't be able to withdraw the funds to himself, since `extDataHash` must be the same and there is no way to change the `recipient` address, Alice will lose a part of her money for the relayer fees. For example, if she has used a `fee` of `20` `USDC` tokens with the withdrawal from the `USDC` pool, Alice will lose `20` `WETH` tokens with Bob's withdrawal for the relayer fees.

This replay attack can be executed using historic values across all of the pools of the privacy pools protocol or any other protocol, which will be using only `amount`, `pubkey`, and `blinding` in the UTXO structure.

Recommendation

We recommend implementing mechanisms to ensure proof uniqueness and validate pool-specific parameters to prevent replay attacks across pools. This can be done by adding the token address of the pool to the UTXO.

Update

Client's response

Acknowledged, as long as we are doing the first transaction, the states cannot be the same. Additionally, both Tornado and Tornado Nova should be susceptible to this.

M-03	Unchecked transfers in `ERC20PrivacyPool`
Severity	**MAJOR**
Status	• ACKNOWLEDGED

Location

File	Location	Line
ERC20PrivacyPool.sol	contract `ERC20PrivacyPool` > function `_processDeposit`	31
ERC20PrivacyPool.sol	contract `ERC20PrivacyPool` > function `_processWithdraw`	38

Description

In the functions `_processDeposit` and `_processWithdraw` of the `ERC20PrivacyPool` contract, the `SafeERC20` library is not used.

Tokens not compliant with the `ERC20` specification could return `false` from the transfer functions call to indicate the transfer fails, while the calling contract would not notice the failure if the return value is not checked. Checking the return value is a requirement, as written in the EIP-20 specification:

> Callers `MUST` handle `false` from returns (bool success). Callers `MUST NOT` assume that `false` is never returned!

Recommendation

We recommend using the `SafeERC20` library implementation from `OpenZeppelin` and call `safeTransfer` or `safeTransferFrom` when transferring `ERC20` tokens.

Update

Client's response

Confirmed, we will fix this.

M-04	Actual token received amount isn't checked in `ERC20PrivacyPool`
Severity	**MAJOR**
Status	• ACKNOWLEDGED

Location

File	Location	Line
ERC20PrivacyPool.sol	contract `ERC20PrivacyPool` > function `_processDeposit`	31

Description

In the function `_processDeposit` of the `ERC20PrivacyPool` contract, there is no check on how many tokens the contract actually received after the transfer.

Some tokens may charge a transfer fee or, conversely, add some amount to the transfer. Such tokens become a problem for the protocol, as the real amount will be different from the one specified in the `transferFrom`.

Recommendation

We recommend considering the amount received for the transfer rather than relying on the amount specified in the `transferFrom` call.

Update

Client's response

Acknowledged, we don't plan to support such tokens.

Oxorio's response

We would like to point out that even the `USDT` has the option to include the fee in its contract code.

M-05	Shielded transfers are possible in the system
Severity	**MAJOR**
Status	• ACKNOWLEDGED

Description

During communication with the client, it was discovered that shielded transfers should be prohibited in the system, yet they remain possible. In the current implementation, receiving a shielded transfer blocks the ability to prove innocence, as the recipient cannot prove the transaction history of the funds received, which is known only to the sender.

Recommendation

We recommend blocking the possibility to send shielded transfers to ensure the system functions as expected.

Update

Client's response

No need to block shielded transactions, since if they want to provide membership proof, they should be using the protocol appropriate, additionally, in the future case where PoI is implemented for shielded transactions, we can use the same contracts.

2.3 WARNING

W-01	Missing validation of the `_maximumDepositAmount` in `PrivacyPool`
Severity	**WARNING**
Status	• ACKNOWLEDGED

Location

File	Location	Line
PrivacyPool.sol	contract `PrivacyPool` > `constructor`	64

Description

In the `constructor` of the `PrivacyPool` contract, there is no validation of the `_maximumDepositAmount` variable, and there is no setter function, which will allow changing the variable after the deployment. If the variable is set to a 0 value, all the deposits to the contract will be blocked.

Recommendation

We recommend adding an additional setter for the `_maximumDepositAmount` function and incorporating validation of the `_maximumDepositAmount` variable in the `constructor`.

Update

Client's response

The contract is immutable and has no governance, so we cannot have a setter as we can't have `onlyOwner` type of access control. Additionally, we see no need to assert `_maximumDepositAmount` to be non-zero as that case needs a redeployment only and should be handled by the deployers of the contracts.

W-02	Missing validations in `PrivacyPool`, `ERC20PrivacyPool`
Severity	**WARNING**
Status	• ACKNOWLEDGED

Location

File	Location	Line
PrivacyPool.sol	contract `PrivacyPool` > `constructor`	63
ERC20PrivacyPool.sol	contract `ERC20PrivacyPool` > `constructor`	24

Description

In the `constructor` of the `PrivacyPool` and `ERC20PrivacyPool` contracts, there is no validation of the `_token` and `_verifier2` contracts for the support of the correct interface, as well as whether these addresses are empty or not. If these addresses are provided incorrectly, there is no function to change their values, so the contracts will have to be redeployed.

Recommendation

We recommend using `ERC165Checker` for validating the interface support, as well as validating addresses for zero values.

Update

Client's response

These should be checked by the deployers of the contracts.

W-03	Relayer address can be zero in `ERC20PrivacyPool`, `ETHPrivacyPool`
Severity	**WARNING**
Status	• ACKNOWLEDGED

Location

File	Location	Line
ERC20PrivacyPool.sol	contract `ERC20PrivacyPool` > function `_processWithdraw`	42
ETHPrivacyPool.sol	contract `ETHPrivacyPool` > function `_processWithdraw`	36

Description

In the functions `_processWithdraw` of contracts `ERC20PrivacyPool`, `ETHPrivacyPool`, there is a check before the withdrawal:

```
require(_extData.recipient != address(0), "Can't withdraw to zero address");
```

However, there is no validation that the `relayer` address is not zero, allowing burning tokens with the incorrect input. For example, if the user decides to execute a withdrawal by himself, leaving the `relayer` address as the zero address but specifying the fee value as non-zero, the tokens will be lost.

Recommendation

We recommend adding validation for a zero address.

Update

Client's response

Acknowledged, in the usual operation, all transactions should be executed by relayers for privacy purposes, but if a user chose to leave the field empty they should be able to.

W-04	No minimal value of withdrawal in `PrivacyPool`
Severity	**WARNING**
Status	• ACKNOWLEDGED

Location

File	Location	Line
PrivacyPool.sol	contract `PrivacyPool` > `constructor`	59

Description

In the `constructor` of contract `PrivacyPool`, there is no limitation on the minimal value for withdrawal, which exposes risks of the DDoS attack, since any user can create a lot of withdrawal requests to the relayer.

Recommendation

We recommend adding a minimal value for the withdrawal.

Update

Client's response

We will add a new function `escapeWithdraw` which will not add anything to merkle tree.

2.4 INFO

I-01	Floating pragma, experimental encoder in `ETHPrivacyPool`, `ERC20PrivacyPool`, `PrivacyPool`
Severity	**INFO**
Status	• ACKNOWLEDGED

Location

File	Location	Line
ERC20PrivacyPool.sol		3
PrivacyPool.sol		3
ETHPrivacyPool.sol		3

Description

In `ETHPrivacyPool`, `ERC20PrivacyPool`, `PrivacyPool` contracts, there is a redundant declaration of `ABIEncoderV2`, which is present by default in the compiler starting from version `0.8.0`. Additionally, all contracts have a floating pragma.

Recommendation

We recommend removing the redundant declaration of `ABIEncoderV2` and specifying the compiler version to a fixed and recent version of the Solidity compiler.

Update

Client's response

Confirmed, will fix.

I-02	Unused code in `proofOfInnocence.circom`
Severity	**INFO**
Status	• ACKNOWLEDGED

Location

File	Location	Line
proofOfInnocence.circom		45
proofOfInnocence.circom		64
proofOfInnocence.circom		123
proofOfInnocence.circom		127
proofOfInnocence.circom		134

Description

In the mentioned locations, there are unused parts of the code.

Recommendation

We recommend removing the unused and commented code to keep the codebase clean.

Update

Client's response

Confirmed, will fix.

I-03	Usage of old Poseidon in `proofOfInnocence.circom`
Severity	**INFO**
Status	• ACKNOWLEDGED

Location

File	Location	Line
proofOfInnocence.circom		66

Description

In `proofOfInnocence.circom` circuit, the original Poseidon hasher is used, whereas Poseidon 2, a faster and more efficient version, is already available (Poseidon 2).

Recommendation

We recommend updating to Poseidon 2 instead of using the original version of Poseidon.

Update

Client's response

Tornado Nova uses it, thats why we use the same.

I-04	Unused imports in `Step (PoI)`
Severity	**INFO**
Status	• ACKNOWLEDGED

Location

File	Location	Line
proofOfInnocence.circom		7-8

Description

The `ProofOfInnocence.circom` file includes unused imports for `merkleTreeUpdater.circom` and `keypair.circom`, leading to unnecessary code complexity.

Recommendation

We recommend removing the unused imports to simplify the codebase.

Update

Client's response

Confirmed, will fix.

I-05	Inefficient gas usage in `MerkleTreeWithHistory`
Severity	**INFO**
Status	• ACKNOWLEDGED

Location

File	Location	Line
MerkleTreeWithHistory.sol	contract `MerkleTreeWithHistory` > function `zeros`	109

Description

In the function `zeros` of contract `MerkleTreeWithHistory`, the linear search approach is used to return a `bytes32` value based on the input index `i`. A binary search algorithm can significantly optimize gas usage by reducing the average number of comparisons needed to find the corresponding `bytes32` value for an index, especially within a sorted structure like the one presented.

Recommendation

We recommend refactoring the `zeros` function to implement a binary search algorithm. This change would enhance the function's gas efficiency by minimizing the number of conditional checks required to return the corresponding `bytes32` value.

Update

Client's response

This function is directly taken from Tornado, so acknowledged.

I-06	Use of custom errors for efficiency and improved information in `MerkleTreeWithHistory`
Severity	**INFO**
Status	• NO ISSUE

Location

File	Location	Line
MerkleTreeWithHistory.sol	contract `MerkleTreeWithHistory` > `constructor`	27
MerkleTreeWithHistory.sol	contract `MerkleTreeWithHistory` > `constructor`	28
MerkleTreeWithHistory.sol	contract `MerkleTreeWithHistory` > function `hashLeftRight`	43
MerkleTreeWithHistory.sol	contract `MerkleTreeWithHistory` > function `hashLeftRight`	44
MerkleTreeWithHistory.sol	contract `MerkleTreeWithHistory` > function `_insert`	54
PrivacyPool.sol	contract `PrivacyPool` > function `calculatePublicAmount`	77
PrivacyPool.sol	contract `PrivacyPool` > function `calculatePublicAmount`	78
PrivacyPool.sol	contract `PrivacyPool` > function `_transact`	109
PrivacyPool.sol	contract `PrivacyPool` > function `_transact`	113
PrivacyPool.sol	contract `PrivacyPool` > function `_transact`	114
PrivacyPool.sol	contract `PrivacyPool` > function `_transact`	115
ETHPrivacyPool.sol	contract `PrivacyPool` > function `_processDeposit`	24
ETHPrivacyPool.sol	contract `PrivacyPool` > function `_processDeposit`	25
ETHPrivacyPool.sol	contract `PrivacyPool` > function `_processWithdraw`	31
ERC20PrivacyPool.sol	contract `PrivacyPool` > function `_processDeposit`	28
ERC20PrivacyPool.sol	contract `PrivacyPool` > function `_processDeposit`	30
ERC20PrivacyPool.sol	contract `PrivacyPool` > function `_processWithdraw`	37

Description

In the mentioned locations, the `require` function is used to check the correctness of the data, but custom errors can be used instead.

Custom errors from `solc 0.8.4` are cheaper than revert strings (cheaper deployment cost and runtime cost when the revert condition is met).

Source:

> Starting from Solidity v0.8.4, there is a convenient and gas-efficient way to explain to users why an operation failed through the use of custom errors. Until now, you could already use strings to give more information about failures (e.g., revert("Insufficient funds.");), but they are rather expensive, especially when it comes to deploy cost, and it is difficult to use dynamic information in them.

Recommendation

We recommend using custom errors instead of `require`.

Update

Client's response

We think `require` is more readable.

I-07	No need to explicitly initialize variables with default values
Severity	**INFO**
Status	• NO ISSUE

Location

File	Location	Line
MerkleTreeWithHistory.sol	contract `MerkleTreeWithHistory`	23-24
MerkleTreeWithHistory.sol	contract `MerkleTreeWithHistory`	32
PrivacyPool.sol	contract `PrivacyPool` > function `_transact`	110
PrivacyPool.sol	contract `PrivacyPool` > function `_transact`	117

Description

In the mentioned locations, variables are assigned a default value.

If a variable is not initialized, it is assumed to have the default value (`0` for `uint`, `false` for `bool`, `address(0)` for `address`...). Explicitly initializing it with its default value is an anti-pattern and wastes gas.

Recommendation

We recommend removing the assignment of default values.

Update

Client's response

I think this way is more readable.

I-08	Redundant event emissions in `PrivacyPool`
Severity	**INFO**
Status	• NO ISSUE

Location

File	Location	Line
[PrivacyPool.sol]()	contract `PrivacyPool` > function `_transact`	123-134

Description

In the function `_transact` of contract `PrivacyPool`, the variables `_args.inputNullifiers`, `_args.outputCommitments`, and `nextIndex` are emitted in multiple events within the same transaction. These emissions occur first through individual `NewCommitment` and `NewNullifier` events for each input nullifier and output commitment, and subsequently in a collective `NewTxRecord` event. This redundant emission of variables across different events in the same transaction scope can lead to unnecessary gas consumption and data redundancy on the blockchain.

Recommendation

We recommend optimizing the event emissions to reduce redundancy and potential gas costs. Consider consolidating event logs or reviewing the necessity of emitting the same information through multiple events, particularly when they pertain to the same transaction context. This approach can enhance efficiency and clarity in contract events management.

Client's response

`NewTxRecord` event is crutial for ASP's and screning, `NewCommitmen`t and `NewNullifier` events are crucial for making a transaction.

I-09	Redundant storage padding in `PrivacyPool`
Severity	**INFO**
Status	• ACKNOWLEDGED

Location

File	Location	Line
PrivacyPool.sol	contract `PrivacyPool`	16

Description

In the contract `PrivacyPool`, the `__gap` variable is used for storage padding to prevent storage collisions, typically seen in upgradeable contracts. However, since `PrivacyPool` is not an upgradeable contract, this padding is unnecessary and may lead to confusion or misinterpretation regarding the contract's design and upgradeability.

Recommendation

We recommend removing the `__gap` variable to streamline the contract's storage layout, enhancing clarity and reducing potential misunderstandings about its upgradeability.

Update

Client's response

Confirmed, will fix.

I-10	Complex `require` logic consumes more gas in `PrivacyPool`
Severity	**INFO**
Status	• ACKNOWLEDGED

Location

File	Location	Line
PrivacyPool.sol	contract `PrivacyPool` > function `calculatePublicAmount`	78

Description

In the function `calculatePublicAmount` of contract `PrivacyPool`, instead of using the `&&` operator in a single `require` statement to check multiple conditions, it is better to use multiple `require` statements with `1` condition per `require` (saving `3` gas per `&&`).

Recommendation

We recommend using multiple `require` statements with `1` condition per `require`.

Update

Client's response

Confirmed, maybe will fix.

I-11	Redundant condition in `PrivacyPool`
Severity	**INFO**
Status	• ACKNOWLEDGED

Location

File	Location	Line
PrivacyPool.sol	`PrivacyPool` > function `verifyProof`	89

Description

In the function `verifyProof` of contract `PrivacyPool`, there is an `if (_args.inputNullifiers.length == 2)` statement, which is redundant since `inputNullifiers` is a fixed-size `bytes32[2]` array, making this condition always true.

```
struct Proof {
  ...
  bytes32[2] inputNullifiers;
  ...
}
```

Recommendation

Remove the redundant condition to improve code clarity, as the array's fixed size makes this check unnecessary.

Update

Client's response

Confirmed, maybe will fix.

I-12	Inefficient use of storage in `PrivacyPool`
Severity	**INFO**
Status	• ACKNOWLEDGED

Location

File	Location	Line
PrivacyPool.sol	`PrivacyPool` > function `_transact`	123-134

Description

In the function `_transact` of contract `PrivacyPool`, the `nextIndex` storage variable is read from storage multiple times to calculate indices for `NewCommitment` and `NewTxRecord` events. Each read operation from storage consumes more gas than reading from the stack. Given that `nextIndex` is incremented once per transaction (by `2`) in `_insert`, its value does not change during the following execution of `_transact`, making multiple storage reads unnecessary and gas-inefficient.

Recommendation

We recommend caching `nextIndex` in a local variable after the `_insert` function call. This approach involves a single read from storage, followed by subsequent reads from the much cheaper stack for calculating indices in event emissions. This optimization will reduce the function's gas consumption, enhancing overall efficiency.

Update

Client's response

Confirmed, will fix.

3 APPENDIX

3.1 DISCLAIMER

The audit makes no statements or warranties about the utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, regulatory regime for the business model, or any other statements about the fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

At the request of client, Oxorio consents to the public release of this audit report. The information contained in this audit report is provided "as is," without any representations or warranties whatsoever. Oxorio disclaims any responsibility for damages that may arise from or in relation to this audit report. Oxorio retains copyright of this report.

3.2 SECURITY ASSESSMENT METHODOLOGY

Oxorio's smart contract audit methodology is designed to ensure the security, reliability, and compliance of smart contracts throughout their development lifecycle. Our process integrates the Smart Contract Security Verification Standard (SCSVS) with our advanced techniques to address complex security challenges. For a detailed look at our approach, please refer to the full version of our methodology. Here is a concise overview of our auditing process:

1. Project Architecture Review

All necessary information about the smart contract is gathered, including its intended functionality and dependencies. This stage sets the foundation by reviewing documentation, business logic, and initial code analysis.

2. Vulnerability Assessment

This phase involves a deep dive into the smart contract's code to identify security vulnerabilities. Rigorous testing and review processes are applied to ensure robustness against potential attacks.

This stage is focused on identifying specific vulnerabilities within the smart contract code. It involves scanning and testing the code for known security weaknesses and patterns that could potentially be exploited by malicious actors.

3. Security Model Evaluation

The smart contract's architecture is assessed to ensure it aligns with security best practices and does not introduce potential vulnerabilities. This includes reviewing how the contract integrates with external systems, its compliance with security best practices, and whether the overall design supports a secure operational environment.

This phase involves a analysis of the project's documentation, the consistency of business logic as documented versus implemented in the code, and any assumptions made during the design and development phases. It assesses if the contract's architectural design adequately addresses potential threats and integrates necessary security controls.

4. Cross-Verification by Multiple Auditors

Typically, the project is assessed by multiple auditors to ensure a diverse range of insights and thorough coverage. Findings from individual auditors are cross-checked to verify accuracy and completeness.

5. Report Consolidation

Findings from all auditors are consolidated into a single, comprehensive audit report. This report outlines potential vulnerabilities, areas for improvement, and an overall assessment of the smart contract's security posture.

6. Reaudit of Revised Submissions

Post-review modifications made by the client are reassessed to ensure that all previously identified issues have been adequately addressed. This stage helps validate the effectiveness of the fixes applied.

7. Final Audit Report Publication

The final version of the audit report is delivered to the client and published on Oxorio's official website. This report includes detailed findings, recommendations for improvement, and an executive summary of the smart contract's security status.

3.3 FINDINGS CLASSIFICATION REFERENCE

3.3.1 Severity Level Reference

The following severity levels were assigned to the issues described in the report:

Title	Description
CRITICAL	Issues that pose immediate and significant risks, potentially leading to asset theft, inaccessible funds, unauthorized transactions, or other substantial financial losses. These vulnerabilities represent serious flaws that could be exploited to compromise or control the entire contract. They require immediate attention and remediation to secure the system and prevent further exploitation.
MAJOR	Issues that could cause a significant failure in the contract's functionality, potentially necessitating manual intervention to modify or replace the contract. These vulnerabilities may result in data corruption, malfunctioning logic, or prolonged downtime, requiring substantial operational changes to restore normal performance. While these issues do not immediately lead to financial losses, they compromise the reliability and security of the contract, demanding prioritized attention and remediation.
WARNING	Issues that might disrupt the contract's intended logic, affecting its correct functioning or making it vulnerable to Denial of Service (DDoS) attacks. These problems may result in the unintended triggering of conditions, edge cases, or interactions that could degrade the user experience or impede specific operations. While they do not pose immediate critical risks, they could impact contract reliability and require attention to prevent future vulnerabilities or disruptions.
INFO	Issues that do not impact the security of the project but are reported to the client's team for improvement. They include recommendations related to code quality, gas optimization, and other minor adjustments that could enhance the project's overall performance and maintainability.

3.3.2 Status Level Reference

Based on the feedback received from the client's team regarding the list of findings discovered by the contractor, the following statuses were assigned to the findings:

Title	Description
NEW	Waiting for the project team's feedback.

Title	Description
FIXED	Recommended fixes have been applied to the project code and the identified issue no longer affects the project's security.
ACKNOWLEDGED	The project team is aware of this finding. Recommended fixes for this finding are planned to be made. This finding does not affect the overall security of the project.
NO ISSUE	Finding does not affect the overall security of the project and does not violate the logic of its work.

3.4 ABOUT OXORIO

OXORIO is a blockchain security firm that specializes in smart contracts, zk-SNARK solutions, and security consulting. With a decade of blockchain development and five years in smart contract auditing, our expert team delivers premier security services for projects at any stage of maturity and development.

Since 2021, we've conducted key security audits for notable DeFi projects like Lido, 1Inch, Rarible, and deBridge, prioritizing excellence and long-term client relationships. Our co-founders, recognized by the Ethereum and Web3 Foundations, lead our continuous research to address new threats in the blockchain industry. Committed to the industry's trust and advancement, we contribute significantly to security standards and practices through our research and education work.

Our contacts:

- oxor.io
- ping@oxor.io
- Github
- Linkedin
- Twitter

THANK YOU FOR CHOOSING

OXORIO