SXT Security Review

Pashov Audit Group

Conducted by: Ch_301, 0x37, Oblivionis, zark, seeques, Pascal, 0xAristos

March 31st 2025 - April 6th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **spaceandtimelabs/sxt-token**, **spaceandtimelabs/sxt-node-op-contracts**, **spaceandtimelabs/sxt-zkpay-contracts** repositories was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About SXT

SXT is a token with a distribution mechanism based on Merkle proofs, allowing users to claim tokens through lightweight, verifiable proofs. The scope also includes modules for payments (an engine handling fund transfers with extensive gas optimization) and staking (staking SXT tokens to nominate validators, manage thresholds, and emitting cross-chain messages).

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hashes:

- 73462f93d03a0cae29129db62b75615f96639c93
- de770e8107d3ec7a8a93cd4fad08c5806f49d68d
- cc275ea67004ca5073be8613a20a548d06dc733d

fixes review commit hashes:

- ffa98911b034c8b976a3830ff93938c7710efce7
- 2a4483339d59d138fce9504862899691db073aa3
- cc275ea67004ca5073be8613a20a548d06dc733d

Scope

The following smart contracts were in scope of the audit:

- `SXTDeployer`
- `SpaceAndTime`
- `SXTDistributorWithDeadline`
- `SXTTokenDistributor`
- `TokenSplitter`
- `SubstrateSignatureValidator`
- `StakingPool`
- `Staking`
- `SXTChainMessaging`
- `ZKPay`
- `ZKPayStorage`
- `AssetManagement`
- `Constants`
- `QueryLogic`
- `Utils`

7. Executive Summary

Over the course of the security review, Ch_301, 0x37, Oblivionis, zark, seeques, Pascal, 0xAristos engaged with Space and Time to review SXT. In this period of time a total of **21** issues were uncovered.

Protocol Summary

Protocol Name	SXT
Repository	https://github.com/spaceandtimelabs/sxt-token
Date	March 31st 2025 - April 6th 2025
Protocol Type	Token distribution and staking

Findings Count

Severity	**Amount**
Critical	2
Medium	4
Low	15
Total Findings	**21**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Accepting duplicate signatures from one signer	Critical	Resolved
[C-02]	Emptying ETH from ZKPay.sol	Critical	Resolved
[M-01]	Fulfillers exploit EIP-150 to bypass zkPayCallback	Medium	Resolved
[M-02]	Frontrunning fulfillQuery() to cancel queries	Medium	Resolved
[M-03]	Attacker can block refunds to the fulfiller's account	Medium	Resolved
[M-04]	stakingDelayStartTime Not Reset After claimUnstake()	Medium	Resolved
[L-01]	No gas limit safeguard allows gas draining	Low	Resolved
[L-02]	Using uninitialized value	Low	Resolved
[L-03]	Inconsistent timeout check	Low	Resolved
[L-04]	Missing check for max/min price from chainlink	Low	Acknowledged
[L-05]	Missing check the L2 sequencer uptime for the price	Low	Acknowledged
[L-06]	Not compliant with the EIP-191 standard	Low	Resolved
[L-07]	Users may fail to claim rewards	Low	Resolved
[L-08]	High gas price incentives for fulfillers cause value leakage	Low	Resolved
[L-09]	MIN_STAKING_AMOUNT too low	Low	Resolved

[L-10]	Lacking msg.data length check	Low	Resolved
[L-11]	fulfillmentTxGasLimit is missing	Low	Resolved
[L-12]	Replay attack In StakingPool.sol	Low	Resolved
[L-13]	Attacker can exploit unbounded gas limit on refund to grief fulfillers	Low	Resolved
[L-14]	fulfillQuery() gasUsed calculation misses L1 rollup fee	Low	Acknowledged
[L-15]	Unstake requests might remain unfulfilled for users	Low	Acknowledged

8. Findings

8.1. Critical Findings

[C-01] Accepting duplicate signatures from one signer

Severity

Impact: High

Likelihood: High

Description

The `validateMessage` function in `SubstrateSignatureValidator` is designed to verify that a message has been signed by a threshold of attestors. However, the function does not account for duplicate signatures from the same attestor. It simply counts how many times a signature maps to a known attestor, regardless of whether the signer has already been counted. Since the function lacks deduplication logic, it is possible to replay the same valid signature multiple times to artificially inflate the count of valid signatures.

```
for (uint256 i = 0; i < signaturesLength; ++i) {
            address recoveredAddress = ecrecover(message, v[i], r[i], s[i]);

            while
              (attestorIndex < attestorsLength && _attestors[attestorIndex] < recovere
                ++attestorIndex;
            }

            if
              (attestorIndex < attestorsLength && _attestors[attestorIndex] == recover
                ++validSignaturesCount;
            }

            if (validSignaturesCount == _threshold) return true;
        }
```

The result is that if a signature from a valid attestor is submitted multiple times (i.e., same v, r, and s), it will be counted multiple times toward meeting the threshold but in reality the rest of attestors would not have signed it.

For a quick PoC, modify the following functions of `SubstrateSignatureValidatorTest` and run `forge test --mt testValidateMessage`. As we can see, the threshold is `2` but only the first attestor has signed and his signature is repeated two times in order to meet the threshold.

```
function setUp() public {
        message = keccak256("test");

        attestors = new address[](2);
        attestors[0] = ecrecover(message, 27, bytes32(uint256(0x1)), bytes32
          (uint256(0x3)));
        attestors[1] = ecrecover(message, 27, bytes32(uint256(0x2)), bytes32
          (uint256(0x4)));

        validator = new SubstrateSignatureValidator(attestors, 2);
    }

    function testValidateMessage() public view {
        bytes32[] memory r = new bytes32[](2);
        r[0] = bytes32(uint256(0x1));
        r[1] = bytes32(uint256(0x1));

        bytes32[] memory s = new bytes32[](2);
        s[0] = bytes32(uint256(0x3));
        s[1] = bytes32(uint256(0x3));

        uint8[] memory v = new uint8[](2);
        v[0] = 27;
        v[1] = 27;

        assertEq(validator.validateMessage(message, r, s, v), true);
    }
```

Recommendations

Consider implementing a memory array inside the loop in `validateMessage` and check if the recovered signer has already been checked against the signature.

[C-02] Emptying ETH from `ZKPay.sol`

Severity

Impact: High

Likelihood: High

Description

An attacker can call `ZKPay::query()`, which is supposed to be used with ERC20 tokens, but pass `NATIVE_ADDRESS` and specify a large amount of ETH. Then, `handleQueryPayment` will assume that this amount came from `msg.value`, so it will think the sender actually sent that ETH to the contract.

```
function query(
        addressasset,
        uint248amount,
        QueryLogic.QueryRequestcalldataqueryRequest
    )
        external
        returns (bytes32 queryHash)
    {
        return _query(asset, amount, queryRequest);
    }

    function handleQueryPayment(
        mapping(address asset => PaymentAsset) storage _assets,
        address assetAddress,
        uint248 tokenAmount
    ) internal returns (uint248 actualAmountReceived, uint248 amountInUSD) {
        if (!isSupported(_assets, assetAddress, PaymentType.Query)) {
            revert AssetIsNotSupportedForThisMethod();
        }

        if (assetAddress == NATIVE_ADDRESS) {
@>              actualAmountReceived = tokenAmount;
        } else {
            uint256 balanceBefore = IERC20(assetAddress).balanceOf(address
              (this));
            IERC20(assetAddress).safeTransferFrom(msg.sender, address
              (this), tokenAmount);
            uint256 balanceAfter = IERC20(assetAddress).balanceOf(address
              (this));

            actualAmountReceived = uint248(balanceAfter - balanceBefore);
        }

        amountInUSD = convertToUsd(_assets, assetAddress, actualAmountReceived);
    }
```

In the same transaction, the attacker cancels their query and withdraws the entire ETH amount. They haven't deposited a single wei, but a `QueryPayment` for a large ETH amount was created, and now they're able to withdraw it.

Recommendations

Consider reverting if the simple `ZKPay::query` function is called with `asset == NATIVE_ADDRESS`.

```
function query(
      addressasset,
      uint248amount,
      QueryLogic.QueryRequestcalldataqueryRequest
    )
        external
        returns (bytes32 queryHash)
    {
+       require(asset != NATIVE_ADDRESS);
        return _query(asset, amount, queryRequest);
    }
```

8.2. Medium Findings

[M-01] Fulfillers exploit `EIP-150` to bypass `zkPayCallback`

Severity

Impact: High

Likelihood: Low

Description

`fulfillQuery` calls the callback function of `ZKPayClient` near the end of its execution, which is done within a try-catch block:

```
try IZKPayClient
          (queryRequest.callbackClientContractAddress).zkPayCallback(
            queryHash, results, queryRequest.callbackData
        ) {
            emit CallbackSucceeded
              (queryHash, queryRequest.callbackClientContractAddress);
        } catch {
            emit CallbackFailed
              (queryHash, queryRequest.callbackClientContractAddress);
        }
```

According to EIP150, a call gets allocated 63/64 bits of the gas, and the entire 63/64 parts of the gas is burnt up If this call runs out of gas. This creates an attack vector: if the user's `zkPayCallback` consumes a significant amount of gas, the fulfiller can specify just enough gas to cause the `zkPayCallback` to run out of gas, while ensuring that the remaining 1/64 gas is sufficient to complete `settleQueryPayment`. This allows a malicious fulfiller to bypass the `zkPayCallback`.

Recommendations

If the callback consumes little gas, no changes are necessary. If the callback consumes a lot of gas, a possible mitigation is to allow users specify a minimum gasleft() before executing `zkPayCallback`.

[M-02] Frontrunning `fulfillQuery()` to cancel queries

Severity

Impact: Medium

Likelihood: Medium

Description

In ZKPay, users send one query request, the related merchant monitors this event and the related oracle will fulfill this request via `fulfillQuery` and get the related fee.

The problem here is that the query's owner can cancel this request at any time. According to the sponsor's information, this contract will be deployed on Ethereum. On Ethereum, the malcious users can monitor the `fulfillQuery` transaction and frontrun this transaction to cancel this query. After this frontrun, the malicious users can get back the pay for the query and also malicious users can get the query's result via monitoring the on-chain data.

```
function cancelQuery(bytes32 queryHash) external nonReentrant {
        QueryLogic.QueryPayment storage payment = _queryPayments[queryHash];
        if (payment.source != msg.sender) revert OnlyQuerySourceCanCancel();

        QueryLogic.cancelQuery(_queryPayments, _queryNonces, queryHash);
    }

    function fulfillQuery(
      bytes32queryHash,
      QueryLogic.QueryRequestcalldataqueryRequest,
      bytescalldataqueryResult
    )
        external
        nonReentrant
        returns (uint248 gasUsed)
    {
    }
```

Recommendations

Once the merchant monitors this query, the merchant can trigger one transaction to accept this query to lock this query. Then nobody cannot cancel this before this query is expired.

[M-03] Attacker can block refunds to the fulfiller's account

Severity

Impact: Medium

Likelihood: Medium

Description

The current `fulfillQuery` pays refund gas to the query source. If the refund fails, the entire flow will revert. This means malicious users can always reject the completion of queries:

```
QueryLogic.sol::L243
            if (refundAmount > 0) {
                // slither-disable-next-line low-level-calls
                (
                  boolrefundSuccess,

                ) = payment.source.call{value: refundAmount}("")
                if (!refundSuccess) revert NativeRefundFailed();
            }
```

Attacker can:

0. Attacker pre-deploy a contract that can freely adjust whether the receive function reverts, initially setting it to not revert.
1. Attacker post a valid query on the EVM ZKPay contract with a non-zero request fee.
2. Offchain resolver finds the valid request, and trying to fill the query.
3. Attacker switch the contract to revert any `receive()`/`fallback()` call.
4. resolver's tx will fail.

Recommendations

A possible mitigation is to wrap the origin token and send it to the user.

[M-04] `stakingDelayStartTime` Not Reset After `claimUnstake()`

Severity

Impact: Medium

Likelihood: Medium

Description

The `stakingDelayStartTime` is initialized when `CollaborativeStaking::initiateUnstake()` is called, but it is never cleared or reset when the corresponding unstaking process is completed via `claimUnstake()`.

```
function claimUnstake() external onlyRole(STAKER_ROLE) {
        emit ClaimUnstake();

        IStaking(STAKING_ADDRESS).claimUnstake();
    }
```

As a result, even after the users have: 1. Staked funds, 2. Initiated unstaking, 3. Successfully claimed the unstaked tokens,

They are still subject to a staking delay before being able to stake again. This happens because the `withStakingDelay` modifier enforces the delay based on `stakingDelayStartTime`, which remains set.

This vulnerability exists when the `STAKING_DELAY_LENGTH` is longer than the `UNSTAKING_UNBONDING_PERIOD` defined in `Staking.sol`. In this case, the users have to wait an additional `STAKING_DELAY_LENGTH` even after the unstaking process is fully completed, which is unnecessary and can disrupt staking workflows and, also, delay participation in the staking.

Recommendations

Consider resetting the `stakingDelayStartTime` in the `claimUnstake()` function to `0`.

8.3. Low Findings

[L-01] No gas limit safeguard allows gas draining

The user can spend all the gas limit here.

```
File: ZKPay.sol#fulfillQuery()

         try IZKPayClient
           (queryRequest.callbackClientContractAddress).zkPayCallback(
            queryHash, results, queryRequest.callbackData
        ) {
```

In other words, once the oracle fulfills the query, there is no check to save the oracle from spending all the gas limit. The callback contract can be any contract; The `ZKPay.sol#fulfillQuery()` function will not revert because the check in `QueryLogic.sol#settleQueryPayment()` accepts any amount different from the paid amount and the actual `gasUsed`.

```
File: QueryLogic.sol#settleQueryPayment()

            if (payoutAmount > payment.amount) {
            payoutAmount = payment.amount;
        }
```

To resolve this, you can add an acceptable difference and check against it.

[L-02] Using uninitialized value

In the `constructor` of the `SubstrateSignatureValidator` contract, the call to `_updateAttestors(attestors)` is made before the `_threshold` value is set via `_updateThreshold(threshold)`. However, `_updateAttestors` checks that `attestors.length >= _threshold`, relying on a state variable that has not yet been initialized.

```
constructor(address[] memory attestors, uint16 threshold) Ownable
      (msg.sender) {
        _updateAttestors(attestors);
        _updateThreshold(threshold);
    }
```

As we can see, `_updateAttestors` is being called before `_updateThreshold`, `_updateAttestors` performs a check using `_threshold` which is uninitialised.

```
function _updateAttestors(address[] memory attestors) internal {
        // ...
@>        if
  (attestors.length < _threshold) revert AttestorsLengthLessThanThreshold();

        // ...
    }
```

To mitigate this issue, consider calling `_updateThreshold` before `_updateAttestors`.

[L-03] Inconsistent timeout check

In ZKPay, users will query some requests with one timeout. Before the timeout expires, the merchant oracle can resolve this request and fulfill this via `fulfillQuery`. If the timeout expires, the merchant oracle is disallowed to fulfill one expried request. And anyone can help the request owner to cancel this request.

The problem here is that we do not have one clear definition of `timeout`. This will cause that when the block.timestamp equals `queryRequest.timeout`, the merchant is allowed to fulfill this query. However, malicious users can frontrun to cancel this query directly. Because when the block.timestamp equals `queryRequest.timeout`, anyone is allowed to cancel one expired query.

```
function cancelExpiredQuery
      (bytes32 queryHash, QueryLogic.QueryRequest calldata queryRequest)
        external
        nonReentrant
    {
        _validateQueryRequest(queryHash, queryRequest);
        if
          (block.timestamp < queryRequest.timeout || queryRequest.timeout == 0) {
            revert QueryHasNotExpired();
        }

        QueryLogic.cancelQuery(_queryPayments, _queryNonces, queryHash);
    }
    function fulfillQuery(
      bytes32queryHash,
      QueryLogic.QueryRequestcalldataqueryRequest,
      bytescalldataqueryResult
    )
        external
        nonReentrant
        returns (uint248 gasUsed)
    {
        if
          (block.timestamp > queryRequest.timeout && queryRequest.timeout != 0) {
            revert QueryTimeout();
        }
    }
```

Make sure the edge case can work for `cancelExpiredQuery` and `fulfillQuery`. Function `fulfillQuery` and `cancelExpiredQuery` should not be executed on the same block.

[L-04] Missing check for max/min price from chainlink

In function `_validatePriceFeed`, we will fetch the price from the chainlink aggreagator. In chainlink, there is one min/max answer. In some edge cases, the returned value may not the actual price, e.g. return min answer if the actual price is less than min answer.

```
function _validatePriceFeed
      (PaymentAsset memory paymentAsset) internal view {
        if (paymentAsset.priceFeed == NATIVE_ADDRESS || !Utils.isContract
          (paymentAsset.priceFeed)) {
            revert InvalidPriceFeed();
        }
        (, int256 answer,, uint256 updatedAt,) = AggregatorV3Interface
          (paymentAsset.priceFeed).latestRoundData();

        if (answer == 0) {
            revert InvalidPriceFeed();
        }

        // slither-disable-next-line timestamp
        if
          (updatedAt + paymentAsset.stalePriceThresholdInSeconds < block.timestamp) {
            revert InvalidPriceFeed();
        }
    }
```

Checking https://docs.chain.link/data-feeds#check-the-latest-answer-against-reasonable-limits, we need to set one reasonable min/max value for in-scope token. If the return price reaches our set min/max value, we should revert here to avoid using the stale price.

[L-05] Missing check the L2 sequencer uptime for the price

In function `_validatePriceFeed`, we will fetch the price's feed from chainlink's feed. When we try to fetch price from Layer2, it's suggested that we should check the sequencer's uptime and pass the grace period and then fetch the price.(https://docs.chain.link/data-feeds/l2-sequencer-feeds)

According to the sponsor's input, contracts may be deployed on Base chain. So if the contract is deployed on Base chain, we need to consider checking the sequencer's uptime.

```
function _validateSxtFulfillUnstake(
        address staker,
        uint248 amount,
        uint64 sxtBlockNumber,
        bytes32[] calldata proof,
        bytes32[] calldata r,
        bytes32[] calldata s,
        uint8[] calldata v
    ) internal view returns (bool isValid) {
        bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encodePacked
          (uint256(uint160(staker)), amount))));
        bytes32 rootHash = MerkleProof.processProof(proof, leaf);
        bytes32 messageHash =
            keccak256(abi.encodePacked(
              abi.encodePacked

            )

        isValid =
            ISubstrateSignatureValidator(
              SUBSTRATE_SIGNATURE_VALIDATOR_ADDRESS
            ).validateMessage(messageHash, r, s, v
    }
```

Add sequencer's uptime check if the deployed chain is Base chain.

[L-06] Not compliant with the `EIP-191` standard

In function `_validateSxtFulfillUnstake`, we will generate the message hash according to the ERC-191. According to the https://eips.ethereum.org/EIPS/eip-191, the proposed format is like as below:

```
"\x19Ethereum Signed Message:\n" + len(message).
```

In function `_validateSxtFulfillUnstake`, we use 36 as one fixed message length. However, when we check these three paramters: `bytes32 rootHash`, `uint64 sxtBlockNumber`, `block.chainid(uint256)`, the total byte length should be 72.

This will cause the message hash is not compliant with the EIP-191 standard.

```
function _validateSxtFulfillUnstake(
        address staker,
        uint248 amount,
        uint64 sxtBlockNumber,
        bytes32[] calldata proof,
        bytes32[] calldata r,
        bytes32[] calldata s,
        uint8[] calldata v
    ) internal view returns (bool isValid) {
        bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encodePacked
          (uint256(uint160(staker)), amount))));
        bytes32 rootHash = MerkleProof.processProof(proof, leaf);
        bytes32 messageHash =
            keccak256(abi.encodePacked(
              abi.encodePacked

            )

        isValid =
            ISubstrateSignatureValidator(
              SUBSTRATE_SIGNATURE_VALIDATOR_ADDRESS
            ).validateMessage(messageHash, r, s, v
    }
```

Calcualte the correct bytes length for the message hash.

[L-07] Users may fail to claim rewards

In SXTDistributorWithDeadline, users can claim SXT token via the merkel proof. If users don't claim the rewards by the deadline, the owner will withdraw back the remaining SXT token in this contract.

The problem here is that there is some time conflict in function `claim` and `withdraw`. If function `claim`, users can claim their expected rewards when the block.timestamp equals `END_TIME`. However, in function `withdraw`, the admin can withdraw the remaining SXT token directly when block.timestamp equals `END_TIME`.

For example:

1. END_TIME = timestamp X.
2. In timestamp X, the owner withdraws the remaining SXT token via `withdraw` function.
3. In timestamp X, users claim rewards. Although users' claim apply the check `block.timestamp > END_TIME`, users' claim will fail because there is not left funds in the contract.

```
function claim(
      uint256index,
      addressaccount,
      uint256amount,
      bytes32[]calldatamerkleProof
    ) public override {
        if (block.timestamp > END_TIME) revert ClaimWindowFinished();
        super.claim(index, account, amount, merkleProof);
    }

    function withdraw() external onlyOwner {
        if (block.timestamp < END_TIME) revert NoWithdrawDuringClaim();

        address tokenAddress = this.token();

        uint256 balance = IERC20(tokenAddress).balanceOf(address(this));
        IERC20(tokenAddress).safeTransfer(msg.sender, balance);
    }
```

Give one clear definition for `END_TIME`. We should not trigger `claim` and `withdraw` in the same block.

[L-08] High gas price incentives for fulfillers cause value leakage

`settleQueryPayment()` uses `tx.gasprice` to obtain the gas price used by the fulfiller. It consists of two gas prices:

- base fee, as set by the system.
- priority fee, which is tipped to the miner and defined by the user.

This gas price is used to calculate the `refundAmount` to refund the user:

```
uint248 usedGasInWei = uint248(gasUsed * tx.gasprice);
        uint248 usedGasInPaymentToken = AssetManagement.convertNativeToToken
          (_assets, payment.asset, usedGasInWei);

        (address payoutAddress, uint248 fee) = ICustomLogic
          (customLogicContractAddress).getPayoutAddressAndFee();
        uint248 feeInPaymentToken = AssetManagement.convertUsdToToken
          (_assets, payment.asset, fee);
        payoutAmount = usedGasInPaymentToken + feeInPaymentToken;

        if (payoutAmount > payment.amount) {
            payoutAmount = payment.amount;
        }

        refundAmount = payment.amount - payoutAmount;
```

In the current `fulfillQuery()`, there is no limit on the gas price used by the fulfiller. If the fee required for executing a job is variable and there is a large amount of fee remaining, the fulfiller can always choose an extremely high priority fee to get their transaction included faster while reducing the refund amount. This significantly reduces the refund that should have been returned to the user, resulting in a major value leak for the entire system.

One possible solution is to allow users to specify a maximum gas price.

[L-09] `MIN_STAKING_AMOUNT` too low

During a `Staking::stake()` call, a `Staked` event is emitted to inform SXT nodes. The staking contract defines a `MIN_STAKING_AMOUNT` to prevent users from making extremely small stakes to spam the SXT chain. However, SXT token has 18 decimals, but the current `MIN_STAKING_AMOUNT` is only set to 100, which is not sufficient to stop malicious users from spamming.

Increase the `MIN_STAKING_AMOUNT` to some value, like 100e18.

[L-10] Lacking `msg.data` length check

In ZKPay, the merchant oracles will finish the query via `fulfillQuery` and get the related fees. The fees will include the query resolve fee from `getPayoutAddressAndFee` and the related transaction's gas fee.

The gas calculation is based on this caluclation `gasUsed = uint248(initialGas - gasleft() + (16 * msg.data.length) + ACCOUNTING_FIXED_GAS_OVERHEAD);`. In normal cases, the `msg.data` will include the expected parameters for `bytes32 queryHash, QueryLogic.QueryRequest calldata queryRequest, bytes calldata queryResult`. One normal `msg.data`'s length can be evaluated based on the specific `bytes32 queryHash, QueryLogic.QueryRequest calldata queryRequest, bytes calldata queryResult`.

The problem here is that malicious users can manipulate the calldata to expand the padding with 0 value. This will lead that the `msg.data`'s length will be far larger than the expected `msg.data`'s length.

The function `fulfillQuery` should be triggered by users' trusted merchant oracle. However, in Etherum, users can monitor the memory pool and frontrun this `fulfillQuery`.

When the attacker manipulates to expand the `msg.data` with 0x00, gas used that the attacker uses will increase, too. But we should notice that in https://eips.ethereum.org/EIPS/eip-2028, the non-zero byte's gas is 16, and zero byte gas is 4. So the attackers will consume 4 gas, however, the query requester has to pay 16 gas for the expanded part.

```
function fulfillQuery(
        bytes32queryHash,
        QueryLogic.QueryRequestcalldataqueryRequest,
        bytescalldataqueryResult
    )
        external
        nonReentrant
        returns (uint248 gasUsed)
    {
        gasUsed = uint248(initialGas - gasleft() +
          (16 * msg.data.length) + ACCOUNTING_FIXED_GAS_OVERHEAD);
    }
```

Add the related gas limit for function `zkPayCallback`.

[L-11] `fulfillmentTxGasLimit` is missing

In ZKPay, users send one query request. In this query request, users will define the `fulfillmentTxGasLimit`, the maximum amount of gas to use for the callback.. This is one important parameter. The merchant oracle will evaluate the total gas consume that the oracle needs to spend. The oracle will expect the users' query payment should be higher than gas consume value for `fulfillQuery` and the execution fee.

The problem here is that when we trigger `fulfillQuery`, we donot have the related gas limit for the `zkPayCallback`. The malicious users may spend a lot more gas than `fulfillmentTxGasLimit`. This will decrease the merchant's profit a lot.

For example:

1. Merchant A's query fee is 2U.
2. Alice requests one query for merchant A, and the `fulfillmentTxGasLimit` is 1000. Alice pays 2.2U. 2U is for the query fee, and 0.2U is used to cover the gas fees.
3. Merchat A checks this event, and evaluates the gas consumption according to `fulfillmentTxGasLimit`, thinks the total gas for `fulfillQuery` will be

covered by 0.2U. Merchant A's oracle will try to resolve this query and fulfill this query.
4. In Alice's `zkPayCallback`, we can consume lots of gas. Then the merchant's profit will drop a lot, because we expect 0.2U to cover the gas.

The problem here is that the query's owner can cancel this request at any time. According to the sponsor's information, this contract will be deployed on Ethereum. On Ethereum, the malcious users can monitor the `fulfillQuery` transaction and frontrun this transaction to cancel this query. After this frontrun, the malicious users can get back the pay for the query and also malicious users can get the query's result via monitoring the on-chain data.

```
function fulfillQuery(
      bytes32queryHash,
      QueryLogic.QueryRequestcalldataqueryRequest,
      bytescalldataqueryResult
    )
        external
        nonReentrant
        returns (uint248 gasUsed)
    {
        try IZKPayClient
          (queryRequest.callbackClientContractAddress).zkPayCallback(
            queryHash, results, queryRequest.callbackData
        ) {
            emit CallbackSucceeded
              (queryHash, queryRequest.callbackClientContractAddress);
        } catch {
            emit CallbackFailed
              (queryHash, queryRequest.callbackClientContractAddress);
        }
    }
```

Add the related gas limit for function `zkPayCallback`.

[L-12] Replay attack In `StakingPool.sol`

The `StakingPool.sol` contract is designed to handle multiple `Staking.sol` contracts, e.g., SXT and USDC, because the owner can use `StakingPool.sol#addStakingContract()` for that.

When the user calls `Staking.sol#claimUnstake()`, the SXT Chain will perform a callback to fulfill the unstake request by triggering `Staking.sol#sxtFulfillUnstake()`. It will validate the signature by sub-calling to `_validateSxtFulfillUnstake()`

```
bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encodePacked
          (uint256(uint160(staker)), amount))));
        bytes32 rootHash = MerkleProof.processProof(proof, leaf);
        bytes32 messageHash = keccak256(abi.encodePacked(
          abi.encodePacked

        )

        isValid = ISubstrateSignatureValidator(
          SUBSTRATE_SIGNATURE_VALIDATOR_ADDRESS
        ).validateMessage(messageHash, r, s, v
```

A replay attack is possible if we have multiple staking contracts. An attacker can use the same inputs from the SXT Chain to fulfill an unstake request in the second staking contract. Note: The SXT token has 18 decimals however, the USDC token has 6 decimals. If the fulfilled amount in the SXT staking contract is 1 SXT, the attacker can unstake (steal) 1000000000000 USDC.

To resolve this issue, you need to make the `messageHash` unique to the stacked token

```
bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encodePacked
          (uint256(uint160(staker)), amount))));
        bytes32 rootHash = MerkleProof.processProof(proof, leaf);
-        bytes32 messageHash = keccak256(abi.encodePacked
- ("\x19Ethereum Signed Message:\n36", rootHash, sxtBlockNumber, block.chainid));
+        bytes32 messageHash = keccak256(abi.encodePacked
+ ("\x19Ethereum Signed Message:\n36", rootHash, sxtBlockNumber, block.chainid, TOKEN_

        isValid = ISubstrateSignatureValidator(
          SUBSTRATE_SIGNATURE_VALIDATOR_ADDRESS
        ).validateMessage(messageHash, r, s, v
```

[L-13] Attacker can exploit unbounded gas limit on refund to grief fulfillers

The current refund logic directly performs a low-level call to the refund address. However, the gas consumed by this call is not paid by the user. Therefore, malicious users can always add malicious logic in the refund callback to waste fulfillers‘ gas.

```
if (payment.asset == NATIVE_ADDRESS) {
            if (payoutAmount > 0) {
                // slither-disable-next-line low-level-calls
                (bool paid,) = payoutAddress.call{value: payoutAmount}("");
                if (!paid) revert NativePaymentFailed();
            }
```

One solution is to always send wrapped origin tokens to the user, while another solution is to limit the gas used by the call.

[L-14] `fulfillQuery()` `gasUsed` calculation misses L1 rollup fee

The `gasUsed` of `fulfillQuery` is designed to always consume the full execution gas.

```
//QueryFulfillment.t.sol::L206
    function testFulfillQueryGasAccounting() public {
        uint256 initialGas = gasleft();
        uint248 usedGas = zkpay.fulfillQuery
          (_queryHash, _queryRequest, "results");
        uint248 expectedGasUsed = uint248(initialGas - gasleft());

        int248 shouldBeZero = int248(expectedGasUsed) - int248(usedGas);
        assertEq(shouldBeZero, 0, "FIXED_GAS_OVERHEAD is off by");
    }
```

The issue here is that the gas paid by the user is always equal to the execution gas fee, but when ZKPay is used on Ethereum L2, the L1 data fee paid by the resolver is not reimbursed.

For example, the current L1 fee on the OP chain is calculated as:

`l1FeeScaled = baseFeeScalar*l1BaseFee*16 + blobFeeScalar*l1BlobBaseFee` `estimatedSizeScaled = max(minTransactionSize * 1e6, intercept + fastlzCoef*fastlzSize)` `l1Cost = estimatedSizeScaled * l1FeeScaled / 1e12`

https://docs.optimism.io/app-developers/transactions/fees#fjord

Recommendations:

Calculate additional L1 gas based on the specific deployment chain.

[L-15] Unstake requests might remain unfulfilled for users

The current `SubstrateSignatureValidator.sol` only records the instantaneous Attestor set. If a user's sxtFulfillUnstake fails at a particular moment(for example, pool has sufficient balance), and the Attestor set changes afterward, the user may never be able to withdraw their tokens. These tokens cannot be withdrawn again because the SXT chain cannot determine if the withdrawal was successful, and the user cannot reapply for the unstake.

A possible solution is to record the mapping of the sxtBlockNumber to the attest set in SubstrateSignatureValidator.sol, allowing users to withdraw using the mapping from the block at the time of the request.

Another solution is to allow the staking pool to internally record the user's unsettled tokens.