shieldify

Etherspot

Gas Tank Paymaster Module

SECURITY REVIEW

Date: 28 September 2025

CONTENTS

1. **About Shieldify Security** 3
2. **Disclaimer** 3
3. **About Etherspot – Gast Tank Paymaster Module** 3
4. **Risk classification** 3
   - 4.1 Impact 4
   - 4.2 Likelihood 4
5. **Security Review Summary** 4
   - 5.1 Protocol Summary 4
   - 5.2 Scope 4
6. **Findings Summary** 5
7. **Findings** 6

1. About Shieldify

Positioned as the first hybrid Web3 Security company, Shieldify shakes things up with a unique subscription-based auditing model that entitles the customer to unlimited audits within its duration, as well as top-notch service quality thanks to a disruptive 6-layered security approach. The company works with very well-established researchers in the space and have secured multiple millions in TVL across protocols, also can audit codebases written in Solidity, Vyper, Rust, Cairo, Move and Go.

Learn more about us at shieldify.org.

2. Disclaimer

This security review does not guarantee bulletproof protection against a hack or exploit. Smart contracts are a novel technological feat with many known and unknown risks. The protocol, which this report is intended for, indemnifies Shieldify Security against any responsibility for any misbehavior, bugs, or exploits affecting the audited code during any part of the project's life cycle. It is also pivotal to acknowledge that modifications made to the audited code, including fixes for the issues described in this report, may introduce new problems and necessitate additional auditing.

3. About Etherspot - Gas Tank Paymaster Module

Etherspot is a top-notch Account & Chain Abstraction infrastructure designed to help developers create an unparalleled cross-chain user experience for their blockchain protocols on Ethereum and EVM-compatible chains.

The **CredibleAccountModule** is a dual-purpose ERC-7579 module that functions as both a validator and a hook for smart accounts, enabling secure session key management with resource locking and token balance validation.

This module implements session-based authentication where users can create time-limited session keys with locked token amounts. It validates user operations against session parameters and ensures sufficient unlocked token balances through pre/post execution hooks.

The **ResourceLockValidator** is a validator module for ERC-7579 smart accounts that enables secure session key management through resource locking mechanisms and Merkle proofs for batched authorizations.

This validator implements dual-mode signature verification, supporting both direct ECDSA signatures and Merkle proof-based validations. It extracts resource lock data from user operation call data and validates operations against predefined resource constraints, enabling efficient batch authorization of multiple resource locks through Merkle tree structures.

4. Risk Classification

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

4.1 Impact

- **High** – results in a significant risk for the protocol's overall well-being. Affects all or most users
- **Medium** – results in a non-critical risk for the protocol affects all or only a subset of users, but is still unacceptable
- **Low** – losses will be limited but bearable – and covers vectors similar to griefing attacks that can be easily repaired

4.2 Likelihood

- **High** – almost certain to happen and highly lucrative for execution by malicious actors
- **Medium** – still relatively likely, although only conditionally possible
- **Low** – requires a unique set of circumstances and poses non-lucrative cost-of-execution to rewards ratio for the actor

5. Security Review Summary

The security review lasted 11 days, with a total of 176 hours dedicated to the audit by two researchers from the Shieldify team.

Overall, the code is well-written. The audit report contributed by identifying one Critical, three High, six Medium and eight Low severity issues. The most critical findings are related to the possibility of the session key owner draining the smart wallet or impersonating another session key owner for the same smart wallet, a lack of refund when cancelling an invoice, and the user not paying for tx gas.

The Etherspot team has done a great job with their test suite and provided exceptional support, and promptly implemented all of the suggested recommendations from the Shieldify researchers.

5.1 Protocol Summary

Project Name	**Etherspot - Gas Tank Paymaster Module**
Repository	etherspot-modular-accounts
Type of Project	Account Abstraction, ERC-7579, EIP-712
Audit Timeline	11 days
Review Commit Hash	9019f2a78c36e74bdb1df4029672998cb4631162
Fixes Review Commit Hash	4d4babe5ae1db9f690ade6f8f68ca4c5f4089ffb

5.2 Scope

The following smart contracts were in the scope of the security review:

File	nSLOC
src/modules/validators/CredibleAccountModule.sol	545
src/invoice_manager/TokenManager.sol	76
src/invoice_manager/InvoiceManager.sol	297
src/invoice_manager/SolverManager.sol	95
src/paymaster/GasTankPaymaster.sol	377
src/paymaster/utils/UniswapHelper.sol	96
Total	**1486**

6. Findings Summary

The following number of issues have been identified, sorted by their severity:

- **Critical** and **High** issues: **4**
- **Medium** issues: **6**
- **Low** issues: **8**
- **Info** issues: **1**

ID	Title	Severity	Status
[C-01]	`SessionKey` Owner Can Drain the Smart Wallet	Critical	Fixed
[H-01]	`SessionKey` Owner Can Impersonate Another Session Key Owner for the Same Smart Wallet	High	Fixed
[H-02]	Paid Tokens by Smart Contract Wallets Are Not Refunded When Cancelling Invoice	High	Fixed
[H-03]	Users Can Escape Paying for the TX Gas	High	Fixed
[M-01]	Session with Duplicate Tokens Can Not Be Settled	Medium	Fixed
[M-02]	Solver Can Receive Zero Settlement When Fee Exceeds Token Amount	Medium	Fixed
[M-03]	Solver Offboarding Can Block Invoice Settlement	Medium	Fixed
[M-04]	The `_postOp()` Function in `GasTank` Is Not Handling Gas Penalty in Calculations	Medium	Fixed
[M-05]	Insufficient Check in `updateCachedPrice()` Can Lead to Read Stale Prices as Valid Prices	Medium	Fixed
[M-06]	The `minSwapAmount` Check Is Made for the Minimum Amount Instead of the Exact Amount	Medium	Fixed

[L-01]	The `isInvoiceSettleable()` Function in `InvoiceManager` Is Not Checking the Credit Balance	Low	Fixed
[L-02]	Missing Emitting `CredibleAccountModule_InvoiceManagerUpdated` Event When Calling `configure()` Function in `CredibleAccountModule`	Low	Fixed
[L-03]	Redundant Token Credit Validation Can Cause Unnecessary Reverts	Low	Fixed
[L-04]	The `_validatePaymasterUserOp()` Function in `GasTank` Will Revert When Adding the Exact Minimum `PostOp` Gas	Low	Fixed
[L-05]	The `setSwapRouter()` in `GasTank` Is Not Revoking the Previous Approval	Low	Fixed
[L-06]	Incorrect Event Emission at `updateCachedPrice()` in `GasTank`	Low	Fixed
[L-07]	Not Checking for Sequencer Down for L2 Networks like Arbitrum	Low	Fixed
[L-08]	Using `transfer` Instead of `call` When Sending ETH in `withdrawAllNative()` Function in `GasTank`	Low	Fixed
[I-01]	Slippage Configuration Limitation	Info	Fixed

7. Findings

[C-01] `SessionKey` Owner Can Drain the Smart Wallet

Severity

Critical Risk

Description

`CredibleAccountModule` is designed to allow time-based restricted authorization for `sessionKey`, where it gives `sessionKey` the ability to transfer tokens on behalf of the main SCW. This session is restricted to a given period, for a given `sessionKey`, and a given set of Locked tokens with an amount for each one.

So, in order for the SCW to be safe, each `sessionKey` can only take the tokens that are in its session, and not exceed the amount locked. This means that the `sessionKey` owner can drain the SCW totally.

The `sessionKeys` have the ability to call two functions:

- `ERC20::approve()` (Any ERC20 token)
- `CredibleAccountModule::claim()` (Only `CredibleAccountModule` contract as target)

The problem is that when approving token there is no check weather the `sessionKey` is approving the `CredibleAccountModule` as the Spender or he is Approving another address, so the `sessionKey` owner can easily drain the SCW by approving tokens to himself with full approval (no check for the amount too), and take the tokens himself (via `transferFrom()` function, since he is a spender).

Location of Affected Code

File: CredibleAccountModule.sol#L763

```
function _validateSingleCall(bytes calldata _callData) internal view
    returns (bool) {
     // code
>>   if (selector == IERC20.approve.selector) return true;
     if (target != address(this)) return false; // If not approve call
        must call this contract
     return true;
}
```

File: CredibleAccountModule.sol#L780

```
function _validateBatchCall(bytes calldata _callData) internal view
    returns (bool) {
     Execution[] calldata execs = ExecutionLib.decodeBatch(_callData[
        EXEC_OFFSET:]);
     for (uint256 i; i < execs.length; ++i) {
        // code
>>      if (selector == IERC20.approve.selector) continue;
        if (execs[i].target != address(this)) return false; // If not
           approve call must call this contract
     }
     return true;
}
```

Impact

Any `sessionKey` owner can completely drain the SCW from all tokens, even tokens that are not locked or tokens that are not whitelisted in `InvoiceManager`.

Proof of Concept

- SCW has a `sessionKey` activated
- The SCW give that `sessionKey` the ability to claim `100 USDT` and `1 WETH`
- The SCW owns USDT, WETH, WBTC, and USDC
- The `sessionKey` enters its active period
- The `sessionKey` made a transaction for calling approve USDT for the full balance, WETH, WBTC, USDC, USDT
- He approved for himself, where he made himself the spender
- Transactions have been executed on SCW
- The `sessionkey` took all the SCW balance for himself

Recommendation

In `_validateSingleCall()`, we should extract the spender address and check that it is approving the `CredibleAccountModule` address to be the spender. The same should be made for

`_validateBatchCall()` by checking that the Execution that is calling approve is with `callData`, making the `CredibleAccountModule` address as the spender.

Team Response

Fixed.

[H-01] `SessionKey` Owner Can Impersonate Another Session Key Owner for the Same Smart Wallet

Severity

High Risk

Description

The Smart Wallet (Etherspot Wallet) can enable multiple sessions at the same time, it can have N count active sessions.

When validating the signature via `CredibleAccountModule::validateUserOp()`, we are making sure that the signer is the SCW and it owns that `sessionKey`.

If the function we are calling is `CredibleAccountModule::claim()`, we pass `sessionKey` for this function to be consumed. However, there is no check to verify if the `sessionKeySigner` that signed the message signed for `claim()` using their `sessionKey`, or for another `sessionKey`. This will allow other `sessionKey` owners to sign messages that consume others' `sessionKey` s for the same Smart Wallet.

Location of Affected Code

File: CredibleAccountModule.sol#L759-L766

```
function _validateSelector(bytes4 _selector) internal pure returns (
    bytes4) {
    return (_selector == this.claim.selector || _selector == IERC20.
        approve.selector) ? _selector : bytes4(0);
}

// code

function _validateSingleCall(bytes calldata _callData) internal view
    returns (bool) {
     (address target,, bytes calldata execData) = ExecutionLib.
         decodeSingle(_callData[EXEC_OFFSET:]);
     bytes4 selector = _validateSelector(bytes4(execData[0:4]));
     if (selector == bytes4(0)) return false;
     if (selector == IERC20.approve.selector) return true;
     if (target != address(this)) return false; // If not approve call
         must call this contract
     return true;
}
```

Impact

Consumption of the session by unauthorized parties

Proof of Concept

- There is a Smart Wallet (SCW) that has two `sessionKey` s (SK1, SK2)
- SK2 signed a message for making a `claim()` function, passing parameters as `sessionKey` equals `SK1`
- `CredibleAccountModule::validateUserOp()` is called
  - The `msg.sender` is the SCW | check passed
  - The signed `sessionKey` is owned by that SCW | check passed
  - We are calling `claim()` function on the `CredibleAccountModule` contract | check passed
- validation will pass successfully, and the Smart Wallet will execute the transaction
- Now executing `claim()` providing `sessionKey` as `SK1`
- All checks will be passed, the sender will be the SCW, and it owns the `sessionKey`
- This will end up claiming the SK1 tokens, without SK1 signing, but SK2 is the one who signed the message

Recommendation

Since we only allow two functions to be called, `ERC20::approve()` and `CredibleAccountModule::claim()` , we should make sure that if the function is `claim()` , we check the `sessionKey` provided to match the `sessionKeySigner` for the message.

Team Response

Fixed.

[H-02] Paid Tokens by Smart Contract Wallets Are Not Refunded When Cancelling Invoice

Severity

High Risk

Description

When processing the claiming process, the smart wallet pays for the amount of token locked and sends it to the `InvoiceManager` contract. In order for the claiming to occur, the session should not have expired, and it should not have already claimed tokens.

After sending tokens, we call `InvoiceManager::creditTokensToInvoice()` , which increases `creditedAmount` for that token.

File: CredibleAccountModule.sol#L731-L732

```
function claim(address _sessionKey, address _token, uint256 _amount)
    external nonReentrant returns (bool) {
     // code
     if (!sd.live || block.timestamp < sd.validAfter || block.timestamp >
        sd.validUntil) {
         revert CredibleAccountModule_SessionKeyDoesNotExist(_sessionKey);
     }
     // code
     for (uint256 i; i < tokenLength;) {
         if (tokens[i].token == _token) {
             // code
>>           IERC20(_token).safeTransferFrom(wallet, invoiceManager,
    _amount);
>>           IInvoiceManager(invoiceManager).creditTokensToInvoice(
    _sessionKey, _token, _amount);
             return true;
         }
         unchecked {
             ++i;
         }
     }
     // code
}
```

File: src/invoice_manager/InvoiceManager.sol#L221

```
function creditTokensToInvoice(address _sessionKey, address _token,
    uint256 _amount) external onlyRole(CREDIBLE_ACCOUNT_ROLE) {
     //code

     // 4. If checks passed, then attribute to the invoice
>>   tokenData[tokenIndex].creditedAmount = newCreditedAmount;
     emit TokensCreditedToInvoice(_sessionKey, _token, _amount);
}
```

In order for the Invoice to get settled, all tokens in that Invoice should be credited so that the settlement process can occur.

File: InvoiceManager.sol#L159-L165

```
function settleInvoice(address _sessionKey) external
    onlySettlerOrLinkedWallet(_sessionKey) nonReentrant {
     // code
     for (uint256 i; i < tokensLength; ++i) {
>>   if (tokens[i].creditedAmount != tokens[i].amount) {
              revert IM_InvoiceNotFullyCredited(
                  _sessionKey, tokens[i].token, tokens[i].amount, tokens[i
                     ].creditedAmount
              );
          }
     }

     // code
}
```

So, in case the session expired and there were still one or more tokens not getting claimed. We will not be able to claim them from `CredibleAccountModule` or credit them on `InvoiceManager`.

Admins can handle this, where they can delete any Invoice they want in case of any error occurring, or unfulfillment, but the problem is that when deleting the Invoice, the SCW does not refund the money it paid.

File: InvoiceManager.sol#L232-L245

```
function cancelInvoice(address _sessionKey, string calldata _reason)
    external onlyRole(SETTLER_ROLE) {
     Invoice storage invoice = invoices[_sessionKey];
     if (invoice.createdAt == 0) revert IM_InvoiceNotFound();

     bytes32 bidHash = invoice.data.bidHash;
     address solver = invoice.data.solver;

     delete invoices[_sessionKey];
     delete invoiceTokenData[_sessionKey];
     delete bidHashToSessionKey[bidHash];
     _removeSolverInvoice(solver, _sessionKey);

     emit InvoiceCancelled(_sessionKey, _reason);
}
```

Since the Invoice is for cross-chain payment, after it is settled and sent to Solver. It will handle the payment for the SCW on the destination chain. In case of non-full settlement, the funds should be refunded to the sender on the Source chain.

Location of Affected Code

File: InvoiceManager.sol#L232-L245

```
function cancelInvoice(address _sessionKey, string calldata _reason)
    external onlyRole(SETTLER_ROLE) {
     Invoice storage invoice = invoices[_sessionKey];
     if (invoice.createdAt == 0) revert IM_InvoiceNotFound();

     bytes32 bidHash = invoice.data.bidHash;
     address solver = invoice.data.solver;

     delete invoices[_sessionKey];
     delete invoiceTokenData[_sessionKey];
     delete bidHashToSessionKey[bidHash];
     _removeSolverInvoice(solver, _sessionKey);

     emit InvoiceCancelled(_sessionKey, _reason);
}
```

Impact

Smart Wallets will not take the money they paid in case the Invoice got cancelled and they paid part of it

Recommendation

We should go for the invoice locked tokens, and in case it is credited, the `amount` should be returned to the original `SmartWallet` (it is stored in the `InvoiceData` struct.

Team Response

Fixed.

[H-03] Users Can Escape Paying for the TX Gas

Severity

High Risk

Description

The current implementation of Paymaster is not taking the amount of gas paid by the paymaster for the tx execution within the same user tx, or before it. It is collected after the execution of the tx.

- `GasTankPaymaster::_validatePaymasterUserOp()` is checking that the verified signer accepts paying for that user
- `GasTankPaymaster::_postOp()` calculates the amount of gas used and emits an event for this

No taking of funds occurs after `_postOp()`, it is designed that Admins will claim the funds by processing `GasTankPaymaster_UserOperationSponsored` event, where they will call `GasTank::repaySponsoredTransaction()` to take ERC20 from the user.

GasTankPaymaster.sol#L361-L368

```
function repaySponsoredTransaction(address _from, uint256 _amount)
    external onlyOwner {
     if (_from == address(0)) revert GasTankPaymaster_InvalidAddress();
     if (_amount == 0) revert GasTankPaymaster_InvalidAmount();
     if (balances[_from] < _amount) revert
        GasTankPaymaster_InsufficientBalance(_from, _amount);
     balances[_from] -= _amount;
     balances[feeReceiver] += _amount;
     emit GasTankPaymaster_RepaySponsoredTransaction(_from, _amount);
}
```

But the problem is that the user can simply withdraw all his balance from the `GasTank` and escape paying for the tx gas. where anyUser can call `gasTankWithdraw()` at any time and withdraw all their balance.

Location of Affected Code

File: GasTankPaymaster.sol#L347-L353

```
function gasTankWithdraw(uint256 _amount) external {
     uint256 currentBalance = balances[msg.sender];
     if (currentBalance < _amount) revert
        GasTankPaymaster_InsufficientBalance(msg.sender, _amount);
     SafeERC20.safeTransfer(token, msg.sender, _amount);
     balances[msg.sender] = currentBalance - _amount; // Safe due to check
         above
     emit GasTankPaymaster_Withdrawn(msg.sender, _amount);
}
```

Impact

- Users can escape paying for the gas paid by the GasTank

Proof of Concept

- UserA has deposited an amount of tokens into the GasTank
- UserA has submitted an ERC4337 transaction
- off-chain system caught the tx, and will execute it
- Tx has been executed successfully, and GasTank paid for it
- User immediately call `GasTank::gasTankWithdraw()` and withdraw all his tokens
- Admins can't repay the fees to the feeReceiver

Recommendation

To mitigate the risk, we should prevent the user from withdrawing his balance in case he has some transactions that have not been paid for. It should be like this.

- Add a mapping (user => isWithdrawPaused)
- `_validatePaymasterUserOp()` will check that `isWithdrawPaused` is false
- `_postOp()` will pause the user (making isWithdrawPaused true)

- `repaySponsoredTransaction()` will reset the user pausing (making isWithdrawPaused false)
- and `gasTankWithdraw()` should prevent withdrawing if user => isWithdrawPaused is true.

This will ensure the user does not withdraw his balance and escape from paying, but this will only work if it is intended for the user to pay for the tx from the same chain. If cross-chain gas is intended to be used where the user can pay from GasTank on Polygon and execute on Arbitrum, for example, then the gas should be taken before executing, or having a system that prevents this can be complex.

Team Response

Fixed.

[M-01] Session with Duplicate Tokens Can Not Be Settled

Severity

Medium Risk

Description

When creating sessions, users can put the same token twice, where there is no check in either `CredibleAccountModule::enableSession()` or `InvoiceManager::createInvoice()` that prevents this behaviour.

The problem is that when claiming, both functions that are called `CredibleAccountModule::claim()` and `InvoiceManager::creditTokensToInvoice()` are catching the first appearance of the token only, so in case the token is repeated, the second lock of these tokens can't be reached.

So all sessions that are made by duplicating one token in `LockedTokens` will be acceptable when enabling the session and creating the invoice, but when finalizing it (claiming all sessions for settling the invoice), it will be impossible, as the second instance of the token will not get reached.

Location of Affected Code

File: CredibleAccountModule.sol#L723

```
function claim(address _sessionKey, address _token, uint256 _amount)
    external nonReentrant returns (bool) {
     // code

     for (uint256 i; i < tokenLength;) {
>>     if (tokens[i].token == _token) {
>>       if (tokens[i].claimedAmount != 0) revert
    CredibleAccountModule_TokenAlreadyClaimed(_sessionKey, _token);
             // code
         }
         // code
       }
     }
     // code
}
```

File: InvoiceManager.sol#L202-L206

```
function creditTokensToInvoice(address _sessionKey, address _token,
    uint256 _amount) ... {
  // code
  for (uint256 i; i < tokenData.length; ++i) {
>>    if (tokenData[i].token == _token) {
            tokenFound = true;
            tokenIndex = i;
            break;
      }
  }

  if (!tokenFound) revert IM_TokenNotFoundInInvoice(_sessionKey, _token);

  // code
}
```

Impact

Inability to settle sessions with a duplicate token.

Recommendation

- We should prevent token duplication either in `CredibleAccountModule::enableSession()` or at `InvoiceManager::createInvoice()`.
- If this is a supported behaviour, we go and search if the token is the same and is claimed instead of reverting in both `CredibleAccountModule::claim()` and `InvoiceManager::creditTokensToInvoice()`

Team Response

Fixed.

[M-02] Solver Can Receive Zero Settlement When Fee Exceeds Token Amount

Severity

Medium Risk

Description

The `settleInvoice()` function finalizes invoice settlement by calling `_processTokenTransfers()`, which deducts protocol fees and sends the remaining balance to the solver. Inside `_processTokenTransfers()`, the fee is calculated with `_calculateFeeForToken()`. If the fee exceeds the token amount, the function sets `pulseFee = amount`, leaving the solver with `solverAmount = 0`.

This situation results in the entire token payment being sent to the fee receiver, while the solver receives nothing, even though the solver successfully completed the work. The vulnerability arises because no validation exists during invoice creation to ensure that the token amount is always greater than the expected protocol fee.

Location of Affected Code

File: InvoiceManager.sol

```
function _processTokenTransfers( address _sessionKey, address solver,
    InvoiceTokenData[] storage tokens, uint256 tokensLength, uint256
    invoicePulseFee ) internal {
  // code
  uint256 pulseFee = _calculateFeeForToken(token, invoicePulseFee);
  if (pulseFee > amount) {
      pulseFee = amount; // entire amount sent as fee
  }
  uint256 solverAmount = amount - pulseFee;
  // code
}
```

Impact

If an invoice is created with a very small token amount, or if fees are later set higher than the amount, solvers receive zero settlement.

Proof of Concept

1. Create an invoice with `amount = 1` token unit.
2. If `_calculateFeeForToken()` returns `5` units as the minimum fee, the contract enforces `pulseFee = 1`.
3. Result: Solver receives `0`, fee receiver receives `1`.

Recommendation

Introduce a minimum amount validation at invoice creation to guarantee that `amount > expected fee`, ensuring solvers are never left unpaid.

Team Response

Fixed.

[M-03] Solver Offboarding Can Block Invoice Settlement

Severity

Medium Risk

Description

The `settleInvoice()` function requires that the solver linked to the invoice is still active by calling `_isSolverActive(solver)`. If the solver has been offboarded after the invoice was created, settlement reverts with `SM_SolverInactive()`. This creates a situation where invoices tied to previously active solvers can no longer be settled, even though work was completed and funds were credited. The vulnerability arises because solver activity is checked at settlement time instead of at invoice creation time.

Location of Affected Code

File: InvoiceManager.sol

```
function settleInvoice(address _sessionKey) external
    onlySettlerOrLinkedWallet(_sessionKey) nonReentrant {
      Invoice storage invoice = invoices[_sessionKey];
      if (invoice.createdAt == 0) revert IM_InvoiceNotFound();

      bytes32 bidHash = invoice.data.bidHash;
      address solver = invoice.data.solver;
      if (!_isSolverActive(solver)) revert SM_SolverInactive(); //@audit

      InvoiceTokenData[] storage tokens = invoiceTokenData[_sessionKey];
      uint256 tokensLength = tokens.length;
      uint256 invoicePulseFee = invoice.pulseFee;

      // code
}
```

Impact

Invoices belonging to solvers who were offboarded after invoice creation become permanently unclaimable. This can freeze funds within the contract, preventing both solvers and fee receivers from being paid.

Recommendation

When offboarding solvers, check if they have pending invoices in `solverInvoices`. If pending invoices exist, block new invoice assignments but allow settlement of existing ones. Once all outstanding invoices are settled, remove the solver completely.

Team Response

Fixed.

[M-04] The `_postOp()` Function in `GasTank` Is Not Handling Gas Penalty in Calculations

Severity

Medium Risk

Description

After executing the user Operation, call `_postExecution()` on the `EntryPoint`. the EntryPoint calls `Paymaster::postOp()` for handling gas paying.

`EntryPoint::_postExecution()` is fired with `actualGas` parameter, which is the amount of gas used in all pref operations, as well as the gas used for making the user tx call.

File: EntryPoint.sol#L356-L359

```
function innerHandleOp( bytes memory callData, UserOpInfo memory opInfo,
    bytes calldata context ) external returns (uint256 actualGasCost) {
     uint256 preGas = gasleft();
     // code

     IPaymaster.PostOpMode mode = IPaymaster.PostOpMode.opSucceeded;
     if (callData.length > 0) {
          bool success = Exec.call(mUserOp.sender, 0, callData,
             callGasLimit);
          if (!success) {
               // code
               mode = IPaymaster.PostOpMode.opReverted;
          }
     }

     unchecked {
          uint256 actualGas = preGas - gasleft() + opInfo.preOpGas;
>>        return _postExecution(mode, opInfo, context, actualGas);
     }
}
```

In `_postExecution()`, we call the `Paymaster::postOp()` and calculate the exact amount of gas used in executing this function. And in case the amount of gas consumed for `postOp` or the `call` made for the User Operation was greater than the limit, the difference is refunded to the user, the remaining is refunded back to the user, but with a penalty of \codex{10%}.

File: EntryPoint.sol#L726

```
function _postExecution(IPaymaster.PostOpMode mode, UserOpInfo memory
    opInfo, bytes memory context, uint256 actualGas) private returns (
    uint256 actualGasCost) {
     uint256 preGas = gasleft();
     unchecked {
         // code
>>       actualGas += preGas - gasleft();

         // Calculating a penalty for unused execution gas
         {
             uint256 executionGasLimit = mUserOp.callGasLimit + mUserOp.
                paymasterPostOpGasLimit;
             uint256 executionGasUsed = actualGas - opInfo.preOpGas;
             // this check is required for the gas used within EntryPoint
                and not covered by explicit gas limits
             if (executionGasLimit > executionGasUsed) {
                 uint256 unusedGas = executionGasLimit - executionGasUsed;
>>               uint256 unusedGasPenalty = (unusedGas * PENALTY_PERCENT)
    / 100;
                 actualGas += unusedGasPenalty;
             }
         }

         actualGasCost = actualGas * gasPrice;
         uint256 prefund = opInfo.prefund;
         if (prefund < actualGasCost) {
           // code
         } else {
>>           uint256 refund = prefund - actualGasCost;
>>           _incrementDeposit(refundAddress, refund);
             bool success = mode == IPaymaster.PostOpMode.opSucceeded;
             emitUserOperationEvent(opInfo, success, actualGasCost,
                actualGas);
         }
     } // unchecked
}
```

- `executionGasLimit` is taken by calldata gasLimit (user Operation) and postOp for paymaster
- `executionGasUsed` is the actual gas subtracting `preOpGas`, so it will include the `UserOp` gas as well as postOp gas, including some other steps in EntryPoint

As we can see, there is a penalty applied by \codex{10%} where we increase the amount of gas used with that penalty, which increases as the limit goes above the actual amount used. The refund process is done after that.

The problem comes in our `GasTankPaymaster` where the `actualGas` cost is calculated based on the `actualGas` value parameter, which is passed to it, without taking into consideration that the penalty can occur. So the amount taken from the user will be less than the amount actually taken from the paymaster in case the Call limit was much larger than the actual used, and the taken postOp was near the actual used.

Location of Affected Code

File: GasTankPaymaster.sol#L472

```
function _postOp(PostOpMode mode, bytes calldata context, uint256
    actualGasCost, uint256 actualUserOpFeePerGas) internal override {
     (address userOpSender, uint256 preChargeNative) = abi.decode(context,
         (address, uint256));
     uint256 priceForTopUp;
     // code
     // Calculate total gas cost in native currency (wei)
>>   uint256 totalGasCostWei = actualGasCost + (paymasterConfig.postOpCost
     * actualUserOpFeePerGas);
     emit GasTankPaymaster_UserOperationSponsored( ... );
}
```

Impact

- Paymaster will pay more than it receives from the user
- This can be used in a malicious way where users submit a large gas limit, where the transaction will take a small gas, leading to an increase in loss for the Paymaster, and the user will only pay the `actualGas` used for their call

Proof of Concept

- UserA submitted a transaction where he made callGasLimit 1,000,000 and `postOp()` gasLimit is 100,000
- actualGas used for his tx was `100,000`
- EntryPoint when calling `postOp()` passed the actual gas, which is `100,000`
- When calculating `totalGasCostWei`, we used the actual gas and added the preconfigured postOp cost
- postOp gasCost was as the gasLimit (100,000), so no significant penalty will occur
- In the EntryPoint, the actualGas will be increased by \codex{10%} for the not used gas for the call, which is `1,000,000 - 100,000 = 900,000`, so it will be `90,000` gas increased, which will be multiplied by the gasCost.
- The Paymaster will end paying this amount without taking it from the user

Recommendation

We should subtract the amount of total `preFunding` we made from the `totalGasUsed`, and then apply the penalty increase. But this can only occur if we have the actual prefunding we did. In the context, we pass `preChargeNative`, but it is not the same as the actual `preFund`, we should either make another variable for prefunding, or make preChargeNative the same as prefunding

Team Response

Fixed.

[M-05] Insufficient Check in `updateCachedPrice()` Can Lead to Read Stale Prices as Valid Prices

Severity

Medium Risk

Description

In `GasTankPaymaster`, when updating the price of a trading asset, in case the price was already new, and the time passed from the last update does not exceed the MaxAge, we return the current cached price.

This is an acceptable behaviour, and in case we pass, we read from the main ChainLink oracles to store the new price.

The problem comes in the first check, where we are only checking the `cachedAge` against the native age without checking for the tokenMaxAge too. So, in case the ERC20 token `maxAge` differs from the native currency age (this depends on the Feed and can change from blockchain to another), the tokenPrice will be read as valid, although it is a stale price as `cacheAge` is greater than the MaxAge for the token.

Location of Affected Code

File: GasTankPaymaster.sol#L391-L393

```
function updateCachedPrice(bool force) public returns (uint256) {
    GasTankPaymasterConfig storage gtpConfig = paymasterConfig;
    uint256 cacheAge = block.timestamp - gtpConfig.cachedPriceTimestamp;
>>  if (!force && cacheAge <= gtpConfig.nativeMaxAge && gtpConfig.
   cachedTokenPrice > 0) {
        return gtpConfig.cachedTokenPrice;
    }
    // Try to get and validate prices
    (bool success, uint256 newPrice) = _tryGetFreshPrice(gtpConfig);
    if (success) {
        // code
    } else {
      // code
    }
}
```

Impact

The `cachedTokenPrice` will be read as a valid price, although `cacheAge` can be greater than `gtpConfig.tokenMaxAge`, leading to incorrect calculations when performing a top-up.

Proof of Concept

- native Oracle has a MaxAge of 1 day
- token oracle has a MaxAge of 1 hour

- The last update happened 6 hours ago
- Now the price should be considered stale as the `tokenMaxAge` (1 hour) is smaller than the cachedAge (6 hours)
- Calling `updateCachedPrice()` will return `cachedTokenPrice` without reading the new price from Chainlink Oracle

Recommendation

In the if condition, we should add another restriction so that `cacheAge <= gtpConfig.tokenMaxAge`, so that we only return the cachedPrice if neither Token nor native has stale prices

Team Response

Fixed.

[M-06] The `minSwapAmount` Check Is Made for the Minimum Amount Instead of the Exact Amount

Severity

Medium Risk

Description

When making topUp for the Paymaster, we swap the ERC20 token for the WETH token to then fund the Paymaster with ETH in the EntryPoint.

There is a minimum amount of tokens to be swapped, which is added when configuring the Paymaster, where we can not swap an amount smaller than this amount.

We apply a slippage check that ensures the value is not too far from the value read by oracles, but when checking for the minimum amount of WETH, we will take it instead of the expected amount, leading to incorrect checks and some correct swaps will be rejected in this case

Location of Affected Code

File: UniswapHelper.sol#L57-L70

```
function _maybeSwapTokenToWeth(IERC20 tokenIn, uint256 amountToSwap,
    uint256 quote) internal returns (uint256) {
     // code
     // uint256 expectedOutput = tokenToWei(amountToSwap, quote);
>>   uint256 amountOutMin = addSlippage(tokenToWei(amountToSwap, quote),
    uniswapHelperConfig.slippage);
     // Compare expected output (in wei) vs minSwapAmount (also in wei)
>>   if (amountOutMin < uniswapHelperConfig.minSwapAmount) {
         return 0;
     }
     return swapToken(
         address(tokenIn), address(wrappedNative), amountToSwap,
            amountOutMin, uniswapHelperConfig.uniswapPoolFee
     );
}
```

Impact

Valid output value can be rejected by the minimum amount check, although the expected amount is greater than the `UniHelperConfig.minSwapAmount`

Recommendation

We should make the check against the expected amount instead of the minimum amount. The expected amount is the value coming from `tokenToWei()`.

Team Response

Fixed.

[L-01] The `isInvoiceSettleable()` Function in `InvoiceManager` Is Not Checking the Credit Balance

Severity

Low Risk

Description

The `InvoiceManager::isInvoiceSettleable()` is a view function that is designed to know whether the Invoice is settleable or not.

In order for the Invoice to be successfully settled, we call `settleInvoice()` and the Invoice should have all its Locked Tokens credited with the full amount in order for it to be settled.

File: InvoiceManager.sol#L159-L165

```
function settleInvoice(address _sessionKey) external
    onlySettlerOrLinkedWallet(_sessionKey) nonReentrant {
     // code

     for (uint256 i; i < tokensLength; ++i) {
>>.    if (tokens[i].creditedAmount != tokens[i].amount) {
             revert IM_InvoiceNotFullyCredited(
                 _sessionKey, tokens[i].token, tokens[i].amount, tokens[i
                    ].creditedAmount
             );
         }
     }
     // code
}
```

Our function is not making this check, it is only checking whether the Solver is active or not and whether there is enough balance for paying or not, without checking the state or the credited balance in the Invoice. So it can return that an Invoice can be successfully settled, although it will not when calling `settleInvoice()`

Location of Affected Code

File: InvoiceManager.sol#L364-L378

```
function isInvoiceSettleable(address _sessionKey) external view returns (
    bool) {
     Invoice storage invoice = invoices[_sessionKey];
     if (invoice.createdAt == 0) return false;
     if (!_isSolverActive(invoice.data.solver)) return false;

     InvoiceTokenData[] storage tokenData = invoiceTokenData[_sessionKey];
     uint256 tokenDataLength = tokenData.length;

     for (uint256 i; i < tokenDataLength; ++i) {
         if (IERC20(tokenData[i].token).balanceOf(address(this)) <
            tokenData[i].amount) {
             return false;
         }
     }
     return true;
}
```

Impact

Incorrect return values from the view function will affect off-chain systems integrating with the contract

Recommendation

We should make another check to make sure that all tokens have been credited, by checking that `creditedAmount` equals `amount` for all tokens

Team Response

Fixed.

[L-02] Missing Emiting `CredibleAccountModule_InvoiceManagerUpdated` Event When Calling `configure()` Function in `CredibleAccountModule`

Severity

Low Risk

Description

When changing the `invoiceManager`, we emit an event that the address of the invoice manager has been changed. This is done when changing it via the `CredibleAccountModule::setInvoiceManager()` function. But when changing the invoice manager through `CredibleAccountModule::configure()`, the event is not emitted.

Location of Affected Code

File: CredibleAccountModule.sol#L107-L116

```
function configure(address _resourceLockValidator, address
    _invoiceManager) external onlyRole(DEFAULT_ADMIN_ROLE) {
     // code
     resourceLockValidator = _resourceLockValidator;
     invoiceManager = _invoiceManager; // <<-- @audit emiting event is
        missing
}
```

File: CredibleAccountModule.sol#L118-L125

```
function setInvoiceManager(address _invoiceManager) external onlyRole(
    DEFAULT_ADMIN_ROLE) {
     // code
     address current = invoiceManager;
     invoiceManager = _invoiceManager;
>>   emit CredibleAccountModule_InvoiceManagerUpdated(current,
    _invoiceManager);
}
```

Impact

Not emitting events when changing critical variables

Recommendation

We should emit `CredibleAccountModule_InvoiceManagerUpdated` when changing the invoiceManager via `configure()` function, or we can make another event for `configure()` to show that both `resourceLockValidator` and `invoiceManager` addresses have been changed

Team Response

Fixed.

[L-03] Redundant Token Credit Validation Can Cause Unnecessary Reverts

Severity

Low Risk

Description

The `creditTokensToInvoice()` function validates that credited tokens for an invoice do not exceed the expected amount. It performs two checks:

1. It reverts with `IM_TokenOverCredited` if `newCreditedAmount > tokenData[tokenIndex].amount`
   .

2. It reverts with `IM_TokenAlreadyCredited` if `tokenData[tokenIndex].creditedAmount == tokenData[tokenIndex].amount`.

The second check is redundant because if the credited amount has already reached the target, any additional credit would already fail the first condition (`newCreditedAmount > amount`). This creates unnecessary complexity and may lead to confusion when interpreting revert reasons.

Location of Affected Code

File: InvoiceManager.sol

```
function creditTokensToInvoice(address _sessionKey, address _token,
    uint256 _amount) external onlyRole(CREDIBLE_ACCOUNT_ROLE) {
    // code

    if (newCreditedAmount > tokenData[tokenIndex].amount) {
        revert IM_TokenOverCredited(_sessionKey, _token, tokenData[
            tokenIndex].amount, newCreditedAmount);
    }
    if (tokenData[tokenIndex].creditedAmount == tokenData[tokenIndex].
        amount) {
        revert IM_TokenAlreadyCredited(_sessionKey, _token);
    }

    // code
}
```

Impact

Increases code complexity and makes audits/maintenance harder without providing meaningful functional protection.

Recommendation

Remove the redundant `AlreadyCredited` check to simplify logic and avoid confusion.

Team Response

Fixed.

[L-04] The `_validatePaymasterUserOp()` Function in `GasTank` Will Revert When Adding the Exact Minimum `PostOp` Gas

Severity

Low Risk

Description

In our `GasTankPaymaster`, there is a configuration for the minimum gas limit for the `postOp()` function, which is stored in `postOpCost`

File: GasTankPaymaster.sol#L80

```
/**
 * @notice Configuration struct for the paymaster
 * @param tokenUsdFeed Oracle for token/USD price feed
 * @param nativeUsdFeed Oracle for native token/USD price feed
 * @param minEPBalance Minimum ETH balance to maintain in EntryPoint
 * @param cachedPriceTimestamp Timestamp of the last price update
 * @param tokenMaxAge Maximum age of token cached price before it's
    considered stale
 * @param nativeMaxAge Maximum age of native cached price before it's
    considered stale
 * @param postOpCost Gas cost for post-operation processing
 * @param cachedTokenPrice Cached token price to avoid frequent oracle
    calls
 * @param markup Price markup applied to oracle prices (in
    PRICE_DENOMINATOR units)
 * @param minVSTokenBalance Minimum token balance required for verifying
    signer to attempt top-up
 */
struct GasTankPaymasterConfig {
    IOracle tokenUsdFeed;
    IOracle nativeUsdFeed;
    uint128 minEPBalance;
    uint48 cachedPriceTimestamp;
    uint48 tokenMaxAge;
    uint48 nativeMaxAge;
>>  uint48 postOpCost;
    uint256 cachedTokenPrice;
    uint256 markup;
    uint256 minFeeReceiverTokenBalance;
}
```

In `_validatePaymasterUserOp()`, we are making sure that the amount of GasLimit put by the user is not smaller than this value, but the problem is that the check enforces the GasLimit to be greater than the `Config::postOpCost`, and is not accepted if the GasLimit is passed equal to `postOpCost`

.

This will result in the validation process failing even when putting GasLimit with the correct value, which is the value in the Paymaster config itself.

Location of Affected Code

File: GasTankPaymaster.sol#L436-L438

```
function _validatePaymasterUserOp( PackedUserOperation calldata userOp,
    bytes32, /*userOpHash*/ uint256 requiredPreFund ) internal view
    override whenNotPaused returns (bytes memory context, uint256
    validationData) {
     // code
     // Calculate comprehensive charge information
     uint256 maxFeePerGas = userOp.unpackMaxFeePerGas();
     uint256 refundPostopCost = paymasterConfig.postOpCost;
>>   if (refundPostopCost >= userOp.unpackPostOpGasLimit()) {
         revert GasTankPaymaster_PostOpGasLimitTooLow();
     }
     // code
}
```

Impact

The correct `postOpGasLimit` value will be rejected in the validation process.

Recommendation

We should use `>` instead of `>=` so that in case PostOpGasLimit equals `postOpCost`, the if block escapes, and tx not reverted with `PostOpGasLimitTooLow` error.

Team Response

Fixed.

[L-05] The `setSwapRouter()` in `GasTank` Is Not Revoking the Previous Approval

Severity

Low Risk

Description

When setting up our paymaster, we call `_initUniswapHelper()`, which adds the swapRouter address. We make full approval for the router for this token, so that it can pull assets from the GasTank without approving each tx.

File: UniswapHelper.sol#L43

```
function _initUniswapHelper( IERC20 _token, IERC20 _wrappedNative,
    ISwapRouter _uniswap, UniswapHelperConfig memory _uniswapHelperConfig
    ) internal {
     // code
     // Approve router for token
>>   _token.approve(address(_uniswap), type(uint256).max);
     // code
     initialized = true;
}
```

This router can be changed via `setSwapRouter()`, and when changing it, we make the approval for the new router. But the issue is that we are not revoking the previous approvals from the previous Uniswap Router, leaving the previous router also has approvals.

Location of Affected Code

File: GasTankPaymaster.sol#L308-L313

```
function setSwapRouter(ISwapRouter _swapRouter) external onlyOwner {
    if (address(_swapRouter) == address(0)) revert
        GasTankPaymaster_InvalidAddress();
    uniswap = _swapRouter;
>>  token.approve(address(uniswap), type(uint256).max);
    emit GasTankPaymaster_SwapRouterUpdated(address(_swapRouter));
}
```

Impact

Leaving the max approval for the previous router, which will not be used.

Recommendation

We should check if the swapRouter already has values. We should revoke this approval by approving `0` tokens in `setSwapRouter()` before approving the new router.

Team Response

Fixed.

[L-06] Incorrect Event Emission at `updateCachedPrice()` in `GasTank`

Severity

Low Risk

Description

When updating the price of the token and the native, in case of successful updating of the price, we are emitting `GasTankPaymaster_TokenPriceUpdated` event, which takes the current price as the first parameter and the previous price as the second parameter. But we always put the previous price as `0` instead of storing the previous `cachedPrice` that was before, making the event emit with an incorrect value for `previousPrice`.

Location of Affected Code

File: GasTankPaymaster.sol#L399

```
function updateCachedPrice(bool force) public returns (uint256) {
    // code
    if (success) {
        gtpConfig.cachedTokenPrice = newPrice;
        gtpConfig.cachedPriceTimestamp = uint48(block.timestamp);
>>      emit GasTankPaymaster_TokenPriceUpdated(newPrice, 0, gtpConfig.
   cachedPriceTimestamp);
        return newPrice;
     } else {
          emit GasTankPaymaster_OracleUpdateFailed();
          return 0; // Signal failure
     }
}
```

Impact

Incorrect event emission data can lead to mistakes in the monitoring process.

Recommendation

We should store the previous `gtpConfig.cachedTokenPrice` and use it in the `previousPrice` parameter slot instead of the constant `0`.

Team Response

Fixed.

[L-07] Not Checking for Sequencer Down for L2 Networks like Arbitrum

Severity

Low Risk

Description

The Current system is designed to work in different chains, and A lot of L2 networks are supported, including Arbitrum.

In Arbitrum L2, there is a sequencer that manages transaction ordering, execution, etc... But this sequencer can become unavailable. When it goes to work again, it handles all queued transactions before and submits them at once, which affects protocols that rely on Token Prices, the liquidation process, etc...

Ref: https://docs.chain.link/data-feeds/l2-sequencer-feeds

Our Price check is not adding the sequencer check, it only relies on Data Feeds. So in case the sequencer goes down and comes back to work, there can be a mistake in the calculations of the Prices, or some users take advantage over the others.

Location of Affected Code

File: GasTankPaymaster.sol#L388-L405

```
function updateCachedPrice(bool force) public returns (uint256) {
    GasTankPaymasterConfig storage gtpConfig = paymasterConfig;
    uint256 cacheAge = block.timestamp - gtpConfig.cachedPriceTimestamp;
    if (!force && cacheAge <= gtpConfig.nativeMaxAge && gtpConfig.
        cachedTokenPrice > 0) {
        return gtpConfig.cachedTokenPrice;
    }
    // Try to get and validate prices
    (bool success, uint256 newPrice) = _tryGetFreshPrice(gtpConfig);
    if (success) {
        gtpConfig.cachedTokenPrice = newPrice;
        gtpConfig.cachedPriceTimestamp = uint48(block.timestamp);
        emit GasTankPaymaster_TokenPriceUpdated(newPrice, 0, gtpConfig.
            cachedPriceTimestamp);
        return newPrice;
    } else {
        emit GasTankPaymaster_OracleUpdateFailed();
        return 0; // Signal failure
    }
}
```

Impact

Incorrect calculations in case the sequencer goes down

Recommendation

We can use the Sequencer feed, which will be an optional and we will add it only to the L2 that is supported to have Sequencer feed by ChainLink. And if it exists, we check for it.

NOTE: Adding the sequencer should be made with `GRACE_PERIOD`, in case it goes down and then is worked on again. More information on how to add it can be found here.

Team Response

Fixed.

[L-08] Using `transfer` Instead of `call` When Sending ETH in `withdrawAllNative()` Function in `GasTank`

Severity

Low Risk

Description

When sending native ETH through the `GasTank::withdrawAllNative()` function, we use `transfer()` instead of a native low-level call. This is not the best practice when transferring ETH, as the transfer only uses `2300` gas, which is only enough for event emitting and not for fallback logic.

Keeping in consideration that `to` can be a smart contract wallet, there can be a fallback logic there. For example, SafeWallet can consume > `2300` or the first time accessing it. If any smart wallet implements a fallback logic, the tx will fail.

https://help.safe.global/en/articles/40813-why-can-t-i-transfer-eth-from-a-contract-into-a-safe

Location of Affected Code

File: GasTankPaymaster.sol#L741

```
function withdrawAllNative(address payable _to) external onlyOwner {
    uint256 nativeBalance = address(this).balance;
    uint256 wNativeBalance = wrappedNative.balanceOf(address(this));
    if (nativeBalance > 0) {
>>      _to.transfer(nativeBalance);
        emit GasTankPaymaster_NativeWithdrawn(_to, nativeBalance);
    }
    if (wNativeBalance > 0) {
        SafeERC20.safeTransfer(wrappedNative, _to, wNativeBalance);
        emit GasTankPaymaster_WrappedNativeWithdrawn(_to, wNativeBalance)
            ;
    }
}
```

Impact

The receiver of the native token can't receive tokens if it is a Smart Contract wallet.

Recommendation

We should use `call()` instead of `transfer()` for transfering ETH

Team Response

Fixed.

[I-01] Slippage Configuration Limitation

Severity

Info Risk

Description

The `uniswapHelperConfig.slippage` parameter is stored as a `uint8`, which restricts the valid range to 0–255. Within the `addSlippage()` function, slippage is applied as:

```
amount * (10000 - slippage) / 10000
```

This design caps the maximum effective slippage tolerance at approximately 2.55% (when slippage = 255) and the minimum at 0%.

For comparison, most DEXs typically allow users to configure slippage tolerances in the range of 0.5%–5%, sometimes higher for volatile or low-liquidity pairs. This limitation may result in failed swaps under normal market conditions if the required tolerance exceeds 2.55%.

Location of Affected Code

File: UniswapHelper.sol#L62

Impact

This issue does not pose a direct security risk. However, the capped slippage tolerance (max ~2.55%) may be unintentionally restrictive compared to typical DEX configurations (~0.5%–5%). As a result, swaps could fail in certain market conditions, but the impact is limited to usability rather than security.

Recommendation

If higher slippage tolerances are intended to be supported, consider storing slippage as a larger data type (`uint16`), so the full basis-point range (`0-10000`) can be represented.

Team Response

Fixed.

shieldify

Thank you!