OZYS
Account-Abstraction(2Q)
Security Analysis Report

Prepared by

78ResearchLab

RedSpider

Sep 13, 2024

TABLE OF CONTENTS

PROJECT OVERALL

About Project

In the second phase of the audit, the focus is on enhancing the Account Abstraction (AA) system, with particular improvements in gas fee handling through the VerifyingTokenPaymaster. This phase introduces advanced features where the Paymaster can prepay gas fees on behalf of users who may not have the required tokens at the time of transaction initiation, and then collect the fees during the `postOp` phase. The token and trade path are made flexible through the use of `paymasterData`. This phase is also tailored for blockchains like Silicon and Klaytn, which operate under the assumption that there is no MEV (Maximum Extractable Value) risk.

Target Summary

Name	Account-Abstraction 2Q
Website	https://ozys.io/
Repository	
Commit	b8047807d374935f97fd79d66c5bebdbe9f6fe6e
Network	Silicon, Klaytn
Languages	Solidity
Method	Source code auditing
Timeline	Jul 1, 2024 ~ Jul 5, 2024

SCOPE

The audit primarily focuses on the following:

- VerifyingTokenPaymaster:
  Ensuring proper gas fee management, where the Paymaster can prepay gas and collect fees from the user during the post-operation phase, with a flexible token and trade path setup via `paymasterData`.
- Seedless Account:
  Verifying the contract logic that allows users to manage accounts without private keys, ensuring secure and user-friendly account operations.
- General Contract Review:
  Additional review of the changes in Paymaster and account interactions to ensure seamless integration and functionality on Silicon and Klaytn blockchains.

Source code

Name	commit
Account-Abstraction 2Q	b8047807d374935f97fd79d66c5bebdbe9f6fe6e

```
├── contracts
│   ├── core
│   │   ├── BaseAccount.sol
│   │   ├── BasePaymaster.sol
│   │   ├── EntryPointSimulations.sol
│   │   ├── EntryPoint.sol
│   │   ├── Helpers.sol
│   │   ├── NonceManager.sol
│   │   ├── SenderCreator.sol
│   │   ├── StakeManager.sol
│   │   └── UserOperationLib.sol
│   ├── external
│   │   ├── interfaces
│   │   │   ├── IUniV3.sol
│   │   │   ├── IV3Estimator.sol
│   │   │   ├── IWebAuthn.sol
│   │   │   └── IWETH9.sol
│   │   ├── ms
│   │   │   ├── ExternalSigner.sol
│   │   │   ├── MSAccountFactory.sol
│   │   │   └── MSAccount.sol
│   │   ├── paymaster
│   │   │   └── VerifyingTokenPaymaster.sol
│   │   ├── seedless
│   │   │   ├── SeedlessAccountFactory.sol
```

```
|   |   |   └── SeedlessAccount.sol
|   |   └── utils
|   |       └── SafeTransferLib.sol
|   ├── interfaces
|   |   ├── IAccountExecute.sol
|   |   ├── IAccount.sol
|   |   ├── IAggregator.sol
|   |   ├── IEntryPointSimulations.sol
|   |   ├── IEntryPoint.sol
|   |   ├── INonceManager.sol
|   |   ├── IPaymaster.sol
|   |   ├── IStakeManager.sol
|   |   └── PackedUserOperation.sol
|   ├── package.json
|   ├── samples
|   |   ├── bls
|   |   |   ├── BLSAccountFactory.sol
|   |   |   ├── BLSAccount.sol
|   |   |   ├── BLSHelper.sol
|   |   |   ├── BLSSignatureAggregator.sol
|   |   |   ├── IBLSAccount.sol
|   |   |   └── lib
|   |   |       ├── BLSOpen.sol
|   |   |       └── hubble-contracts
|   |   |           └── contracts
|   |   |               └── libs
|   |   |                   ├── BLS.sol
|   |   |                   ├── BNPairingPrecompileCostEstimator.sol
|   |   |                   └── ModExp.sol
|   |   ├── callback
|   |   |   └── TokenCallbackHandler.sol
|   |   ├── LegacyTokenPaymaster.sol
|   |   ├── SimpleAccountFactory.sol
|   |   ├── SimpleAccount.sol
|   |   ├── TokenPaymaster.sol
|   |   ├── utils
|   |   |   ├── IOracle.sol
|   |   |   ├── OracleHelper.sol
|   |   |   └── UniswapHelper.sol
|   |   └── VerifyingPaymaster.sol
|   └── utils
|       └── Exec.sol
```

RISK CLASSIFICATION

Severity

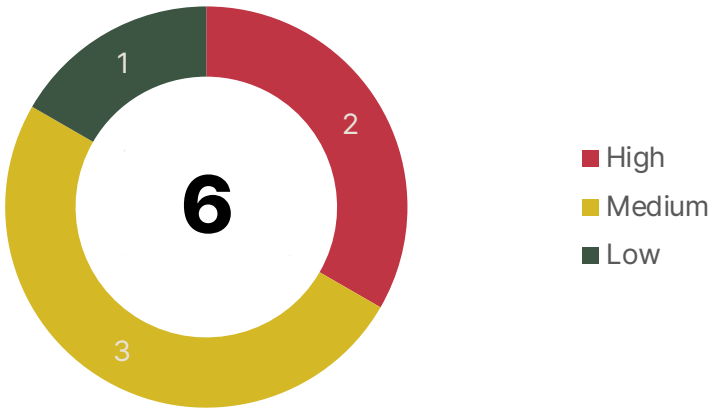
Our risk classification is based on Severity Categorization of code4ena.

> High ●
>
> Assets can be stolen, lost, compromised directly or indirectly via a valid attack path (e.g. Malicious Input Handling, Escalation of privileges, Arithmetic).

> Medium ●
>
> Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.

> Low ●
>
> Assets are not at risk. User mistake, misuse of privileges, governance risk fall under this grade.

FINDINGS BREAKDOWN



Severity	Acknowledged	fixed	Total
● High	1	1	2
● Medium	0	3	3
● Low	0	1	1
			6

* Fixed : Risk is fixed by Ozys.
* Acknowledged : Ozys has recognized the risk but has not addressed it, as it poses only a minor impact.

FINDINGS

HIGH

H-01. Gas verification is required in VerifyingTokenPaymaster: _validatePaymasterUserOp

Acknowledged

IMPACT

The bundler can lose its asset as `_validatePaymasterUserOp` does not verify if the `sender` has sufficient tokens to cover the gas payment.

DESCRIPTION

`_validatePaymasterUserOp` is a function that verifies whether the `sender` meets the necessary conditions to use the Paymaster before the `UserOp` is executed. In the Account-Abstraction model, the `validate*` functions must be simulated off-chain before execution, including a check to ensure that the `sender` has enough tokens to cover the gas used for the transaction.

```
function _validatePaymasterUserOp(PackedUserOperation calldata userOp, bytes32
userOpHash, uint256)
    internal
    view
    override
    returns (bytes memory context, uint256 validationResult)
{
        // ...
}
```

File 1 : VerifyingTokenPaymaster.sol Function: _validatePaymasterUserOp

However, `_validatePaymasterUserOp` does not utilize the `maxCost` parameter or perform any validation, which leaves room for malicious actors (griefers) to intentionally waste the bundler's gas.

A griefer could exploit this in the following way:

1. The griefer sets WETH as the token to pay for gas in the Paymaster and ensures a sufficient balance is available.
2. They initiate a `UserOp` transaction that transfers the WETH balance to another account owned by the griefer.
3. Since there is no separate off-chain validation process, the `UserOp` gets executed, but when the bundler attempts to retrieve gas fees via `postOp`, the WETH balance is already depleted, causing the transaction to revert.
4. The griefer repeats this process to drain the bundler's gas.

RECOMMENDATIONS

In `_validatePaymasterUserOp`, ensure that the `sender` holds enough tokens to cover the `maxCost`.

STATUS Acknowledged

Followings are the conversation we had with Ozys about this issue.

Ozys: Hello!

There are some details we didn't fully explain regarding the issue and issues #8829 and #8838, so we would like to share additional information.

The primary use case we envision for the VerifyingTokenPaymaster is:

- A user with no on-chain activity
- Becomes eligible for a Launchpad airdrop and is able to claim rewards
- Without having to deposit ETH into the account contract, the user can claim the reward using a prefund at the prePayment stage
- We would then collect a portion of the reward through postOp.Thus, the current signing policy of the internal VerifyingTokenPaymaster signer module is very limited as follows:
- The calldata in the userOperation must be a method recognized by our internal system
- If the userOperation affects the sender's (token) balance, the signature is denied
- At launch, it will only sign if the userOperation calldata contains `LaunchpadAirdrop.claim + approve(paymaster)`.

Regarding the case you shared in the issue:

- Since this use case assumes the user does not have assets to pay at the prePayment stage, in most cases, validation itself will likely be impossible.
- From the bundler's perspective, it will only execute handleOps if it has confirmed through a separate simulation that it won't incur a loss from the submitted userOperation.
- With the modified paymaster logic (which reverts if there are insufficient tokens to collect), the bundler will either not throw the transaction or it will remain in a pending state indefinitely (or be rejected).

We would appreciate your thoughts on this matter!

78ResearchLab: Hello,

Could you explain how the bundler verifies the simulation? If you plan to use the `_validatePaymasterUserOp` function with the RPC Debug API for simulation, wouldn't you need gas checks in the function logic?

Ozys: Hello!

The base code we are using for the bundler is eth-infinitism's bundler. When a userOperation is received, pre-validation is performed before adding it to the mempool, where user signature validation and paymaster signature validation take place at the following locations:

- https://github.com/eth-infinitism/bundler/blob/852d8d90bf3049f9b9484d9d32c6aa4d98e88150/packages/bundler/src/modules/ExecutionManager.ts#L41
- https://github.com/eth-infinitism/bundler/blob/852d8d90bf3049f9b9484d9d32c6aa4d98e88150/packages/validation-manager/src/ValidationManager.ts#L194Though more work is still needed,
- https://github.com/eth-infinitism/bundler/blob/852d8d90bf3049f9b9484d9d32c6aa4d98e88150/packages/bundler/src/modules/BundleManager.ts#L87This or another part will further implement a simulation through `simulateHandleOps` and `estimateGas`, ensuring that only transactions unlikely to revert are processed by the bundler.

78ResearchLab: Hello, In order to accurately verify the simulation in the bundler, I would like to clarify my understanding of the verification process:

- During the frontend `estimateGas` stage, if a revert occurs, the transaction is dropped (with `estimateGas` eventually handled by the bundler's `simulateHandleOp`).
- When the bundler creates a bundle, it calls `validateUserOp` to simulate validation. If a revert occurs, the bundle is not created, and the transaction is dropped from the mempool.

However, this process would not prevent a griefer from using opcodes like `timestamp` when executing `executeOp` to transfer tokens owed to the paymaster during the postOp phase. (Currently, the Paymaster signer module only signs limited cases, but it may allow general account use in the future). Is there any separate logic in place to validate against this scenario?

If I have misunderstood anything, please let me know so I can review further. Thank you!

Ozys: Hello! Could you explain more about the case where you mentioned using opcodes like `timestamp` during `executeOp`?
Additionally, as mentioned, in the bundler's simulation logic:

- https://github.com/eth-infinitism/bundler/blob/852d8d90bf3049f9b9484d9d32c6aa4d98e88150/packages/bundler/src/modules/BundleManager.ts#L87There will be further implementation of `EntryPointSimulations.simulateHandleOp` to validate state overrides or a simple `estimateGas` call to check for reverts before dropping the transaction from the mempool.

Regarding future general account use in the Paymaster signer module,

- We will add verification logic to ensure MSAccount (accounts deployed by our system)

addresses are checked by the verifying signer module.
- (We do not yet plan to release this version of the Paymaster signer module publicly!)

Thank you!

78ResearchLab: Hello, I considered a scenario where a griefer creates a malicious contract when calling external contracts through `executeOp` in the account.
Assume the timestamp during the bundler's simulation is 1, and the timestamp when the simulated transaction is executed is 3. The griefer creates a malicious contract that transfers all the tokens owed to the paymaster to another wallet if the timestamp is 2 or later. At the time of the bundler's simulation, the tokens exist, so there is no revert, but by the time the transaction is executed, the tokens are gone, causing a revert.
That's why I suggested adding logic to verify the user's balance in the `_validatePaymasterUserOp` process (since bundlers ban deterministic opcodes like `timestamp` during the `validateUserOp` call). However, if you want users with zero balance to still claim tokens and pay for gas through `claim + approve`, this logic would need special handling. I'm not sure which approach is best at the moment. What do you think?

Ozys: Hello! Thank you for the explanation! If I understand correctly, the scenario is something like this:

- Griefer 4337 account: account A
- Griefer's another account: account B

1. **Time 1**: Account A approves tokens for Account B
2. **Time 2**: Account A requests a signature from the paymaster and sends a userOperation to the bundler
3. **Time 3**: The bundler simulates and registers the operation in the mempool
4. **Time 4**: Account B calls `token.transferFrom(accountA, accountB, amount)`
5. **Time 5**: The bundler calls `EntryPoint.handleOps`, which reverts, causing the bundler to lose gas.

In this situation:

- At **Time 5**, the bundler would run another simulation (estimate, check for reverts) and execute the transaction.
- The calldata would be restricted to `approve(paymaster)` and `claim`, ensuring that no tokens claimed in the userOperation could be moved to another location except the paymaster (assuming the account contract is MSAccount).
- Since we don't expect MEV in silicon, the likelihood of malicious actions by the bundler is low. The chances that `transferFrom` is executed before `uo` in the same block seem low.

Please let me know your thoughts!

78ResearchLab: Hello, The example using `timestamp` might not have been very clear, so I thought of

another scenario where gas fees could be avoided.

(Assume this is an MSAccount that allows transactions other than just `claim + approve`.)

1. The griefer prepares Account 1 and Account 2.
2. They create a transaction in Account 1 to transfer tokens from Account 2 and send a gas-intensive transaction from Account 2 with tokens secured.
3. The bundler bundles these two transactions together, and since neither has issues individually, the simulation does not cause a revert. (For accounts not using paymaster, the validation process collects gas fees upfront, preventing this attack.)
4. When the transaction is actually processed, the transaction sent from Account 2 reverts because there are no tokens.

Without pre-securing the gas in `_validatePaymasterUserOp`, I don't think this attack can be prevented. Even if only `claim + approve` transactions are allowed for now, if you plan to support more diverse transactions in the future, I think some measures should be taken. I'd appreciate your thoughts on this.

Ozys: Hello, I understand the example you shared! As you mentioned, without securing the gas upfront in `_validatePaymasterUserOp`, it will be challenging to prevent such attacks.
Thus, we plan to keep restricting the userOperation calldata to `claim + approve` when using the VerifyingTokenPaymaster.
If, in the future, there is a need to pay gas fees with tokens for various transactions,

- We will develop and use a standard TokenPaymaster that collects the tokens upfront at the prePayment stage and refunds the remaining amount to the user during postOp.

Thank you!

H-02. SeedlessAccount: Attacker can take over the assets and the account because _validateSignature returns an inccorect value

Fixed

IMPACT

An attacker can execute transactions with a SeedlessAccount without the user's signature. They can steal deposited assets, and since it is Upgradeable, they can upgrade the contract and take over the account.

DESCRIPTION

The `_decodeAndValidateSignature` function of the SeedlessAccount contract returns true if signature verification is successful. Therefore, the `_validateSignature` function of SeedlessAccount returns `SIG_VALIDATION_FAILED` when the signature verification is successful and `SIG_VALIDATION_SUCCESS` when it fails.

```
function _validateSignature(PackedUserOperation calldata userOp, bytes32 userOpHash)
internal override virtual returns (uint256 validationData) {
@>  if (_decodeAndValidateSignature(abi.encodePacked(userOpHash), userOp.signature))
@>      return SIG_VALIDATION_FAILED;
@>  return SIG_VALIDATION_SUCCESS;
}
```

File 2 : erc4337-contract/contracts/external/seedless/SeedlessAccount.sol#L120-125 Function: _validateSignature

When a user tries to execute transactions by calling `handleOps` of the EntryPoint, the `validateUserOp` function of the SeedlessAccount contract is called to verify the signature. This function calls `_validateSignature` to get the result of the signature verification.

Anyone can call `handleOps` of the EntryPoint, and if the paymaster data is not provided, the paymaster signature can be ignored since it is not used.

```
function handleOps(
@>  PackedUserOperation[] calldata ops,
    address payable beneficiary
) public nonReentrant {
    uint256 opslen = ops.length;
    UserOpInfo[] memory opInfos = new UserOpInfo[](opslen);

    unchecked {
        for (uint256 i = 0; i < opslen; i++) {
            UserOpInfo memory opInfo = opInfos[i];
            (
@>              uint256 validationData,
                uint256 pmValidationData
@>          ) = _validatePrepayment(i, ops[i], opInfo);
@>          _validateAccountAndPaymasterValidationData(
                i,
```

```
@>              validationData,
                pmValidationData,
@>              address(0)
            );
        }

        ...
    }
}

function _validatePrepayment(
    uint256 opIndex,
    PackedUserOperation calldata userOp,
    UserOpInfo memory outOpInfo
)
    internal
    returns (uint256 validationData, uint256 paymasterValidationData)
{
    ...

    bytes memory context;
@>  if (mUserOp.paymaster != address(0)) {
        (context, paymasterValidationData) = _validatePaymasterPrepayment(
            opIndex,
            userOp,
            outOpInfo,
            requiredPreFund
        );
    }
    ...
}
```

File 3 : erc4337-contract/contracts/core/EntryPoint.sol#L174-205, L616-674 Function: _validateSignature, _validatePrepayment

If the user's signature verification result `validationData` is 0 (`SIG_VALIDATION_SUCCESS`), it passes `_validateAccountAndPaymasterValidationData`. Otherwise, it reverts.

```
function _validateAccountAndPaymasterValidationData(
    uint256 opIndex,
@>  uint256 validationData,
    uint256 paymasterValidationData,
    address expectedAggregator
) internal view {
@>  (address aggregator, bool outOfTimeRange) = _getValidationData(
        validationData
    );
@>  if (expectedAggregator != aggregator) {
        revert FailedOp(opIndex, "AA24 signature error");
    }
    if (outOfTimeRange) {
        revert FailedOp(opIndex, "AA22 expired or not due");
    }
```

```
    ...
}

function _getValidationData(
    uint256 validationData
) internal view returns (address aggregator, bool outOfTimeRange) {
@>  if (validationData == 0) {
@>      return (address(0), false);
    }
    ValidationData memory data = _parseValidationData(validationData);
    // solhint-disable-next-line not-rely-on-time
    outOfTimeRange = block.timestamp > data.validUntil || block.timestamp <
data.validAfter;
    aggregator = data.aggregator;
}
```

File 4 : erc4337-contract/contracts/core/EntryPoint.sol#L562-607 Function: _validateAccountAndPaymasterValidationData, _getValidationData

Therefore, an attacker can execute arbitrary transactions with a SeedlessAccount. They can steal assets deposited in the SeedlessAccount contract, and since the SeedlessAccount contract is Upgradeable, they can upgrade the contract to change the logic and take over the account.

RECOMMENDATIONS

Return `SIG_VALIDATION_SUCCESS` when signature validation is successful.

```
function _validateSignature(PackedUserOperation calldata userOp, bytes32 userOpHash)
internal override virtual returns (uint256 validationData) {
    if (_decodeAndValid Fixed nature(abi.encodePacked(userOpHash), userOp.signature))
-       return SIG_VALIDATION_FAILED;
+       return SIG_VALIDATION_SUCCESS;
-   return SIG_VALIDATION_SUCCESS;
+   return SIG_VALIDATION_FAILED;
}
```

STATUS Fixed

Ozys: Modified the return value of `_validateSignature`.

Modified in commit `8b5f7501f03b06b7522ed97fd050dfa7d6979ddd` .

MEDIUM

M-01. VerifyingTokenPaymaster: _postOp may fail due to duplicate approvals

Fixed

IMPACT

Some tokens cannot be used at VerifyingTokenPaymaster.

DESCRIPTION

Some ERC20 tokens like USDT apply Approval race protection. These tokens revert the transaction if there is an existing allowance when calling approve. Therefore, if an allowance is set, it must first be set to 0 before calling approve again.

In the `_postOp` function, the tokens from the account are swapped into native tokens using a Uniswap V3 based DEX. The token provided by the user as `tokenPayment.token` is the same as `tokenPayment.path[0]`. After transferring the user's tokens into the contract, `SafeTransferLib.safeApprove` is called and starts to swap from `tokenPayment.path[0]` by `swapNeg`.

In `swapNeg`, `SafeTransferLib.safeApprove` is called again to adjust the allowance. If `tokenPayment.path[0]` is USDT, `SafeTransferLib.safeApprove` will fail because it has already increased the allowance.

```
function _postOp(PostOpMode, bytes calldata context, uint256 actualGasCost, uint256
actualUserOpFeePerGas)
    internal
    override
{
    (address sender, bytes32 userOpHash, TokenPayment memory tokenPayment) =
abi.decode(context, (address, bytes32, TokenPayment));

    uint256 actualETHNeeded = actualGasCost + (additionalPostOpCost *
actualUserOpFeePerGas);
    uint256 userBal = IERC20(tokenPayment.token).balanceOf(sender);
    uint256 amount = userBal;

    IUniV3 router = IUniV3(tokenPayment.router);
    IWETH9 weth = IWETH9(router.WETH9());
    if(tokenPayment.token == address(weth)){
        if(userBal >= actualETHNeeded) amount = actualETHNeeded;
        SafeTransferLib.safeTransferFrom(tokenPayment.token, sender, address(this),
amount);
    }
    else{
```

```
        uint256[] memory amounts = getAmountsIn(actualETHNeeded, tokenPayment.path,
tokenPayment.pool);
        if(userBal >= amounts[0]) amount = amounts[0];
        SafeTransferLib.safeTransferFrom(tokenPayment.token, sender, address(this),
amount);
@>      SafeTransferLib.safeApprove(tokenPayment.token, tokenPayment.router, amount);

        uint256 pathLen = tokenPayment.path.length;
@>      for (uint256 i = 0; i < pathLen - 1; i++) {
@>          swapNeg(router, tokenPayment.pool[i], tokenPayment.path[i], amounts[i],
tokenPayment.path[i + 1], amounts[i + 1]);
        }
    }

    uint256 bal = weth.balanceOf(address(this));
    weth.withdraw(bal);
    entryPoint.depositTo{value: address(this).balance}(address(this));

    emit UserOperationSponsored(
        userOpHash,
        sender,
        actualETHNeeded,
        amount,
        bal,
        tokenPayment
    );
}

function swapNeg(IUniV3 router, address pool, address tokenIn, uint256 amountIn, address
tokenOut, uint256 amountOut) private {
@>  SafeTransferLib.safeApprove(tokenIn, address(router), amountIn);
    router.exactOutputSingle(
        IUniV3.ExactOutputSingleParams({
            tokenIn: tokenIn,
            tokenOut: tokenOut,
            fee: IUniV3(pool).fee(),
            recipient: address(this),
            deadline: block.timestamp + 100,
            amountOut: amountOut,
            amountInMaximum: amountIn,
            sqrtPriceLimitX96: 0
        })
    );
}
```

File 5 : erc4337-contract/contracts/external/paymaster/VerifyingTokenPaymaster.sol#L96-152 Function: _postOp, swapNeg

Additionally, since `ExactOutputSingleParams` is used during swaps, not all of the input tokens may be used in the swap. This can leave some allowance remaining after the swap. If an allowance remains, the approve will fail the next time `_postOp` is called, causing the transaction to be reverted.

RECOMMENDATIONS

SafeTransferLib.sol appears to be from the solady library. The same library provides `SafeTransferLib.safeApproveWithRetry` to handle ERC20 tokens with approval race protections. This function handles approvals without return values similarly to `SafeTransferLib.safeApprove`, and resets the allowance to 0 before re-approving if the first approve fails.

Additionally, since approve is called in `swapNeg`, there's no need to call approve after transferring the user's tokens.

```
function _postOp(PostOpMode, bytes calldata context, uint256 actualGasCost, uint256
actualUserOpFeePerGas)
    internal
    override
{
    ...
    else{
        uint256[] memory amounts = getAmountsIn(actualETHNeeded, tokenPayment.path,
tokenPayment.pool);
        if(userBal >= amounts[0]) amount = amounts[0];
        SafeTransferLib.safeTransferFrom(tokenPayment.token, sender, address(this),
amount);
-       SafeTransferLib.safeApprove(tokenPayment.token, tokenPayment.router, amount);

        uint256 pathLen = tokenPayment.path.length;
        for (uint256 i = 0; i < pathLen - 1; i++) {
            swapNeg(router, tokenPayment.pool[i], tokenPayment.path[i], amounts[i],
tokenPayment.path[i + 1], amounts[i + 1]);
        }
    }

    uint256 bal = weth.balanceOf(address(this));
    weth.withdraw(bal);
    entryPoint.depositTo{value: address(this).balance}(address(this));

    emit UserOperationSponsored(
        userOpHash,
        sender,
        actualETHNeeded,
        amount,
        bal,
        tokenPayment
    );
}

function swapNeg(IUniV3 router, address pool, address tokenIn, uint256 amountIn, address
tokenOut, uint256 amountOut) private {
-   SafeTransferLib.safeApprove(tokenIn, address(router), amountIn);
+   SafeTransferLib.safeApproveWithRetry(tokenIn, address(router), amountIn);
    router.exactOutputSingle(
        IUniV3.ExactOutputSingleParams({
```

```
            tokenIn: tokenIn,
            tokenOut: tokenOut,
            fee: IUniV3(pool).fee(),
            recipient: address(this),
            deadline: block.timestamp + 100,
            amountOut: amountOut,
            amountInMaximum: amountIn,
            sqrtPriceLimitX96: 0
        })
    );
}
```

STATUS Fixed

Ozys: Removed unnecessary `safeApprove` call and modified to use `safeApproveWithRetry` instead.

Modified in commit `ebd94b86299f887c8e2cdf0751826d31cef9807c`.

M-02. VerifyingTokenPaymaster: If the user does not have enough ERC20 tokens, the transaction is reverted Fixed

IMPACT

The transaction fails due to attempting to swap with an incorrect amount. The behavior is different when paying the gas fee with WETH and with ERC20 tokens.

DESCRIPTION

When `tokenPayment.token` is WETH, if the WETH in the wallet is less than the required amount, it just takes the WETH in the wallet. It accepts the transaction even if there is insufficient WETH. The policy is to treat insufficient tokens as a loss.

However, if a user uses ERC20 tokens other than WETH, it is handled differently. The user can pay the gas fee with ERC20 tokens other than WETH, which are swapped to WETH through a UniV3-based DEX.

Suppose the user pays the gas fee by A tokens. If the A tokens in the wallet are less than the required amount `amounts[0]`, it just takes the A tokens in the wallet. Although the received A tokens are less than `amounts[0]`, the `swapNeg` function uses the original values in the `amounts` array to swap. This means it tries to swap using more tokens than the wallet provided.

```
function _postOp(PostOpMode, bytes calldata context, uint256 actualGasCost, uint256
actualUserOpFeePerGas)
    internal
    override
{
    (address sender, bytes32 userOpHash, TokenPayment memory tokenPayment) =
abi.decode(context, (address, bytes32, TokenPayment));

@>  uint256 actualETHNeeded = actualGasCost + (additionalPostOpCost *
actualUserOpFeePerGas);
    uint256 userBal = IERC20(tokenPayment.token).balanceOf(sender);
@>  uint256 amount = userBal;

    IUniV3 router = IUniV3(tokenPayment.router);
    IWETH9 weth = IWETH9(router.WETH9());
    if(tokenPayment.token == address(weth)){
        if(userBal >= actualETHNeeded) amount = actualETHNeeded;
        SafeTransferLib.safeTransferFrom(tokenPayment.token, sender, address(this),
amount);
    }
    else{
@>      uint256[] memory amounts = getAmountsIn(actualETHNeeded, tokenPayment.path,
tokenPayment.pool);
```

```
@>      if(userBal >= amounts[0]) amount = amounts[0];
@>      SafeTransferLib.safeTransferFrom(tokenPayment.token, sender, address(this),
amount);
        SafeTransferLib.safeApprove(tokenPayment.token, tokenPayment.router, amount);

        uint256 pathLen = tokenPayment.path.length;
        for (uint256 i = 0; i < pathLen - 1; i++) {
@>          swapNeg(router, tokenPayment.pool[i], tokenPayment.path[i], amounts[i],
tokenPayment.path[i + 1], amounts[i + 1]);
        }
    }

    uint256 bal = weth.balanceOf(address(this));
    weth.withdraw(bal);
    entryPoint.depositTo{value: address(this).balance}(address(this));

    emit UserOperationSponsored(
        userOpHash,
        sender,
        actualETHNeeded,
        amount,
        bal,
        tokenPayment
    );
}

function swapNeg(IUniV3 router, address pool, address tokenIn, uint256 amountIn, address
tokenOut, uint256 amountOut) private {
    SafeTransferLib.safeApprove(tokenIn, address(router), amountIn);
    router.exactOutputSingle(
        IUniV3.ExactOutputSingleParams({
            tokenIn: tokenIn,
            tokenOut: tokenOut,
            fee: IUniV3(pool).fee(),
            recipient: address(this),
            deadline: block.timestamp + 100,
@>          amountOut: amountOut,
@>          amountInMaximum: amountIn,
            sqrtPriceLimitX96: 0
        })
    );
}
```

File 6 : erc4337-contract/contracts/external/paymaster/VerifyingTokenPaymaster.sol#L96-152 Function: _postOp, swapNeg

RECOMMENDATIONS

When `userBal < amounts[0]`, it should swap only the amount of tokens equal to `userBal` to handle the shortage in the same way as WETH. The `amounts` should be recalculated based on the input token amount `userBal`.

STATUS Fixed

Ozys: Modified the policy to accurately receive the amount of gas fee from the wallet, and revert the transaction if the wallet has less than required.

Modified in commit `0e9a13e26d5a9ee27bb38cef91dfedc5fafebd23`.

M-03. VerifyingTokenPaymaster: Cannot pay gas with ERC20 because of missing receive function for WETH.withdraw

Fixed

IMPACT

The feature to pay gas with ERC20 always fails.

DESCRIPTION

To pay gas fees with ERC20, `_postOp` receives WETH from the user or other ERC20 tokens, swaps them for WETH, and converts it to ETH. To convert WETH to ETH, the VerifyingTokenPaymaster contract needs a `receive` function to accept the native token. However, the VerifyingTokenPaymaster does not have a `receive` function, causing the `WETH.withdraw` call to fail and `_postOp` to always fail.

```
function _postOp(PostOpMode, bytes calldata context, uint256 actualGasCost, uint256
actualUserOpFeePerGas)
    internal
    override
{
    ...

    uint256 bal = weth.balanceOf(address(this));
@>  weth.withdraw(bal);
    entryPoint.depositTo{value: address(this).balance}(address(this));

    emit UserOperationSponsored(
        userOpHash,
        sender,
        actualETHNeeded,
        amount,
        bal,
        tokenPayment
    );
}
```

File 7 : erc4337-contract/contracts/external/paymaster/VerifyingTokenPaymaster.sol#L96-136 Function: _postOp

RECOMMENDATIONS

Define the `receive` function in the VerifyingTokenPaymaster.

STATUS

Fixed

Ozys: Added the `receive` function and modified the contract to not use a Proxy due to gas limit

issue.

Fixed in commit `4244e860f4b552da82b092da5dd20bf6fc753cbc` .

● LOW

L-01. VerifyingTokenPaymaster: Implemented as UUPSUpgradeable but inherited contract is not upgradeable

Fixed

IMPACT

The inherited contract is not upgradeable.

DESCRIPTION

The VerifyingTokenPaymaster contract is implemented as UUPSUpgradeable but inherits the non-upgradeable BasePaymaster contract. Although the BasePaymaster contract does not define storage variables, the Ownable contract, which is also inherited, does. Without using a storage gap, adding storage variables to the parents contracts could break the storage layout.

Additionally, since Ownable is not upgradeable, the BasePaymaster contract initializes the Ownable contract in its constructor. As a result, the owner of the logic contract remains as the deployer.

```
abstract contract BasePaymaster is IPaymaster, Ownable {
    IEntryPoint public immutable entryPoint;

    uint256 internal constant PAYMASTER_VALIDATION_GAS_OFFSET =
UserOperationLib.PAYMASTER_VALIDATION_GAS_OFFSET;
    uint256 internal constant PAYMASTER_POSTOP_GAS_OFFSET =
UserOperationLib.PAYMASTER_POSTOP_GAS_OFFSET;
    uint256 internal constant PAYMASTER_DATA_OFFSET =
UserOperationLib.PAYMASTER_DATA_OFFSET;

@>  constructor(IEntryPoint _entryPoint) Ownable(msg.sender) {
        _validateEntryPointInterface(_entryPoint);
        entryPoint = _entryPoint;
    }
    ...
```

File 8 : erc4337-contract/contracts/core/BasePaymaster.col#L16-26 Function: BasePaymaster contract

RECOMMENDATIONS

Either make all inherited contracts upgradeable or be careful with the storage layout when upgrading. Also, remove the owner of the logic contract from the constructor of the VerifyingTokenPaymaster contract.

STATUS Fixed

Ozys: Removed the owner in the constructor. Will pay attention to the storage layout during updates.
Fixed in commit `5a834cb7b0b17c2f70cddd4a8563491065cab687`.

ABOUT 78ResearchLab

78ResearchLab is a offensive security corporation offering security auditing, penetration testing, education to enterprises, national organizations, and laboratories with the goal of making safe and convenience digital world. We have our own proprietary technology from system/security analysis and projects on various industries. We are working with the top technical experts who have won prizes in global Realword Hacking Competition/CTF, reported numerous security vulnerabilities, and have 10 years of experience in the information security.
Learn more about us at https://www.78researchlab.com/.

ABOUT RED SPIDER

Red Spider offers cyber security research and auditing service with our new R&D technologies and customized solution in IoT, OS, Web3.
Learn more about red spider at https://www.78researchlab.com/redSpider.html.