Versa

Audit

Presented by:

OtterSec contact@osec.io

Woosun Song procfs@osec.io

Matteo Oliva matt@osec.io

Robert Chen r@osec.io

Contents

Appendices

01 | Executive Summary

Overview

Versa engaged OtterSec to perform an assessment of the `Wallet` program. This assessment was conducted between August 18th and September 9th, 2023. For more information on our auditing methodology, see Appendix B.

Key Findings

Over the course of this audit engagement, we produced 7 findings in total.

In particular, we identified a medium risk vulnerability within the paymaster implementation that enables users to bypass fees in a revert caused by a failure to transfer the amount to the paymaster. This subsequently triggers the execution of the post-operation function again, where the set mode skips the transfer function (OS-VRS-ADV-00). We also advised against using a revert mechanism accompanied by an error string, primarily due to the challenge of decoding non-string data when using `abi.decode` (OS-VRS-ADV-01). Additionally, we highlighted an issue allowing the spending limit to be set to zero, which may be misleading as zero also denotes uninitialized data (OS-VRS-ADV-02).

We also provided suggestions regarding the absence of a zero address check on the signer address in the ECDSA validator (OS-VRS-SUG-00) and emphasized the importance of memory alignment for the free memory pointer (OS-VRS-SUG-01). Furthermore, we brought to attention various coding best practices, including the removal of dead code and adherence to naming conventions, to be integrated into the code base to enhance code efficiency and optimization (OS-VRS-SUG-02), (OS-VRS-SUG-03).

02 | Scope

The source code was delivered to us in a git repository at github.com/VersaLab/versa-contract. This audit was performed against commit 56b82dd.

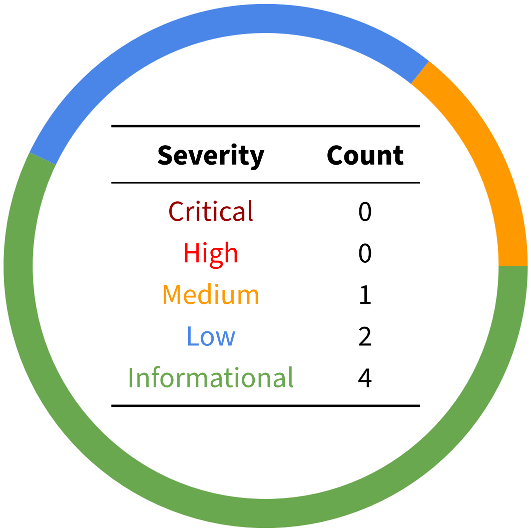
A brief description of the programs is as follows.

Name	Description
wallet-contracts	The versa wallet incorporates ERC-4337, implementing account abstraction. This design vastly enhances extensibility, enabling the utilization of various plugins to assist with identity verification, enhancing security, and adding custom logic to transactions.

03 | Findings

Overall, we reported 7 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will help mitigate future vulnerabilities.



Severity	Count
Critical	0
High	0
Medium	1
Low	2
Informational	4

04 | Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-VRS-ADV-00	Medium	Resolved	Fee payment in `VersaVerifyingPaymaster` may be bypassed due to an unhandled revert condition and the absence of checks on the sender's balance and allotted allowance.
OS-VRS-ADV-01	Low	Resolved	Inability to decode string with `abi.decode` when non-string data is returned.
OS-VRS-ADV-02	Low	Resolved	The spending limit being set to zero may be misleading.

OS-VRS-ADV-00 [med] | Fee Bypass In Paymaster Implementation

Description

The vulnerability arises in `_postOp` within `VersaVerifyingPaymaster`, where the user is not charged the fees in a particular instance involving a revert condition.

paymaster/VersaVerifyingPaymaster.sol SOLIDITY

```solidity
function _postOp(PostOpMode mode, bytes calldata context, uint256 actualGasCost)
    ↪  internal override {
    /* ... */
    if (mode != PostOpMode.postOpReverted) {
        token.safeTransferFrom(account, address(this), actualTokenCost);
        balances[token] += actualTokenCost;
        emit UserOperationSponsored(account, address(token), actualTokenCost);
    }
}
```

`_postOp` is designed to manage post-operation actions, including charging the user. This callback function is invoked with an argument called `mode`, encompassing three modes: `success`, `reverted`, and `postOpReverted`. The first two modes signify the transaction execution's success and failure, respectively. The last mode indicates a failure within `_postOp`. In the event of a revert, the function fails, resulting in its subsequent invocation with the mode set as `postOpReverted`.

The issue arises when `transferFrom` is called to transfer `ERC-20` tokens from the user's account to the contract. `transferFrom` may fail for various reasons, such as:

1. Insufficient balance in the user's account.
2. Insufficient approval for the contract to spend the user's tokens.
3. Being blacklisted or frozen, preventing token transfers.

When `transferFrom` fails, a revert occurs, resulting in `_postOp` being called again with `postOpReverted`. This second call will not invoke `transferFrom`, effectively allowing the transaction to finish without reverting, bypassing fee payment.

Proof of Concept

SOLIDITY

```solidity
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.19;

contract Test1 is TestSetup {
    function testIssue1() public {
        // create a valid signature
        {
            VersaVerifyingPaymaster.PaymasterData
                memory paymasterData = VersaVerifyingPaymaster.PaymasterData(
                    IERC20Metadata(address(token)),
                    VersaVerifyingPaymaster.SponsorMode.GAS_AND_FEE,
                    uint48(block.timestamp) + 1000 days,
                    0,
                    1e18 /* 1:1 */,
                    ""
                );
            UserOperation memory userOp = UserOperation(
                /* ... */
            );
            signPaymasterAndData(userOp, paymasterData);
            userOp.paymasterAndData = packPaymasterAndData(
                address(paymaster),
                address(paymasterData.token),
                uint8(paymasterData.mode),
                paymasterData.validUntil,
                paymasterData.fee,
                paymasterData.exchangeRate,
                paymasterData.signature
            );
            signValidatorAndData(userOp, address(validator));
            UserOperation[] memory ops = new UserOperation[](1);
            ops[0] = userOp;
            entrypoint.handleOps(ops, payable(user));
            require(user.balance > 0, "exploit unsuccessful");
        }
    }
}
```

1. A user initiates a transaction to interact with the contract and triggers `\_postOp`.
2. During the execution of `\_postOp`, the contract attempts to transfer `ERC-20` tokens from the wallet.
3. `transferFrom` fails for any of the reasons listed above, resulting in a revert.
4. When the transaction reverts due to the `transferFrom` failure, it triggers `\_postOp` again with the `postOpReverted` mode.
5. Due to the mode being set to `postOpReverted`, no `ERC-20` transfers are performed, allowing the user to bypass fees.
6. The `EntryPoint` contract rebates all gas to a user-specified address. This step is effectively equivalent to theft of funds from the paymaster, as the paymaster pays the `EntryPoint`, the `EntryPoint` pays the user, but the user pays nothing.

Remediation

Ensure that both the approved allowance and the wallet's balance are adequate to cover `actualTokenCost`. Additionally, it would be advisable to discontinue the paymaster service when tokens are paused or when wallets are blacklisted.

Patch

This issue was mitigated by hardening the off-chain paymaster service.

1. The paymaster checks the ERC20 token balance of the wallet prior to signing.
2. The paymaster checks if the signed transaction includes an `ERC20.approve` message.
3. The paymaster only signs fee subsidization with ERC-20 tokens adhering to the standard. This minimizes the possibility of `ERC20.transferFrom` failing.

OS-VRS-ADV-01 [low] | Failure In Decoding Error Message

Description

The issue arises in `executeAndRevert` within `Executor`, where the function tries to revert with the actual string containing the error message when the execution of a particular call fails.

common/Executor.sol SOLIDITY

```solidity
    /**
     * Execute a call but also revert if the execution fails.
     * The default behavior of the Safe is to not revert if the call fails,
     * which is challenging for integrating with ERC4337 because then the
     * EntryPoint wouldn't know to emit the UserOperationRevertReason event,
     * which the frontend/client uses to capture the reason for the failure.
     */
    function executeAndRevert(address to, uint256 value, bytes memory data,
    ↪   Enum.Operation operation) internal {
        bool success = execute(to, value, data, operation, type(uint256).max);

        bytes memory returnData = getReturnData(type(uint256).max);
        // Revert with the actual reason string
        if (!success) {
            if (returnData.length < 68) revert();
            assembly {
                returnData := add(returnData, 0x04)
            }
            revert(abi.decode(returnData, (string)));
        }
    }
```

Typically, in the event of a contract execution failure, an error message is returned as an encoded hexadecimal string, following the pattern of an `ABI` encoded error data. However, it is essential to recognize that this convention is not universally followed. Some contracts may return arbitrary data or implement custom error-handling procedures. Therefore, the assumption that the return data from a failed call is consistently a string is inaccurate, and utilizing `abi.decode` on `returnData` may result in failures, which in turn may impact the contract's functionality.

Remediation

Eliminate the error string upon a revert to mitigate any potential adverse effects on contract functionality from decoding non-string data.

Patch

Fixed in dd47282.

OS-VRS-ADV-02 [low] | Discrepancy In Spending Limit

Description

Setting a spending limit to zero is not viable, as the value zero is used as an indicator of uninitialized data rather than a limitation of zero. For instance, `_checkNativeTokenSpendingLimit` only performs checks if the allowance amount is non-zero, resulting in a zero allowance amount to exhibit identical behavior to infinite allowance.

contracts/plugin/hooks/SpendingLimitHooks.sol SOLIDITY

```solidity
function _checkNativeTokenSpendingLimit(address _wallet, uint256 _value) internal {
    SpendingLimitInfo memory spendingLimitInfo = getSpendingLimitInfo(_wallet,
    ↪  address(0));

    // Check if there is a spending limit set for this wallet
    if (spendingLimitInfo.allowanceAmount > 0) {
        // Update the spent amount with the transaction value
        spendingLimitInfo.spentAmount += _value;
        _checkAmountAndUpdate(address(0), spendingLimitInfo);
    }
}
```

Thus, utilizing the value zero as a sentinel may result in misinterpretation or unexpected behavior. To maintain clarity and ensure that spending limits are effectively enforced, set a minimum limit greater than zero.

Remediation

Ensure that a minimum amount of one is specified for the spending limit to avoid any confusion. We also recommend documenting this behavior to prevent any potential confusion.

Patch

Fixed in ee54c9d by elaborating documentation.

05 General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may lead to security issues in the future.

ID	Description
OS-VRS-SUG-00	Zero address check on the signer address in `ECDSAValidator` absent.
OS-VRS-SUG-01	Absence of memory alignment in `getReturnData`.
OS-VRS-SUG-02	Dead code in the code base may impact code readability.
OS-VRS-SUG-03	Suggestions regarding best practices and improved code readability.

OS-VRS-SUG-00 | Zero Address Checks

Description

In `validateSignature` within `ECDSAValidator`, the signer address passed to `_validateSignature` is not checked to be non-zero. The value may become zero if the signer entry for the key `_userOp.sender` is uninitialized.

contracts/plugin/validator/ECDSAValidator.sol SOLIDITY

```solidity
function validateSignature(
    UserOperation calldata _userOp,
    bytes32 _userOpHash
) external view returns (uint256 validationData) {
    [...]
    validationData = _validateSignature(
        _signers[_userOp.sender],
        splitedSig.signature,
        splitedSig.hash,
        splitedSig.validUntil,
        splitedSig.validAfter
    );
}
```

Remediation

Ensure `_signers[_userOp.sender]` is non-zero. It should be noted that the current implementation is not exploitable due to checks in `_validateSignature`, but it is advised to follow best practices in case of future code additions.

Patch

Fixed in 9b3a981.

OS-VRS-SUG-01 | Memory Alignment

Description

`getReturnData` utilizes inline assembly in a manner that deviates from `YUL` specifications. Specifically, the free pointer `ptr`, which indicates the next area of memory to be allocated, is not guaranteed to be aligned by `0x20`. For instance, if a `maxLen` of `0xf` is given, the alignment of the free pointer will be broken, as it is incremented by `0x2f`.

common/Executor.sol SOLIDITY

```solidity
// get returned data from last call or calldelegate
function getReturnData(uint256 maxLen) internal pure returns (bytes
    ↪  memoryreturnData) {
    assembly {
        let len := returndatasize()
        if gt(len, maxLen) {
            len := maxLen
        }
        let ptr := mload(0x40)
        mstore(0x40, add(ptr, add(len, 0x20)))
        mstore(ptr, len)
        returndatacopy(add(ptr, 0x20), 0, len)
        returnData := ptr
    }
}
```

As `YUL` assumes free memory alignment, the free pointer altered by `getReturnData` may introduce inconsistencies in other parts of the code.

Remediation

Ensure the alignment of free memory by rounding up allocation length to the nearest multiple of 0x20.

Patch

Fixed in c509054.

OS-VRS-SUG-02 | Dead Code

Description

In `SessionKeyValidator`, `_getChainId` remains unutilized throughout the entire code base. The presence of such code may negatively impact the code base's readability and maintainability.

plugin/validator/SessionKeyValidator.sol SOLIDITY

```solidity
function _getChainId() internal view returns (uint256 id) {
    // solium-disable-next-line security/no-inline-assembly
    assembly {
        id := chainid()
    }
}
```

Remediation

Ensure to eliminate any obsolete code to adhere to the best coding practices.

Patch

Fixed in b1f08c8.

OS-VRS-SUG-03 | Code Maturity

Description

The following suggestions are geared toward best coding practices and optimization:

1. Naming conventions should be standardized, ensuring consistency. For example, between `SessionKeyValidator` and `SessionkeyValidator` or `validUntil1` and `validuntil2`.
2. In `SpendingLimitsHook`, both the wallet address and `msg.sender` are asserted to be equal, although these two variables are utilized interchangeably. We recommend unifying their usage for the sake of clarity.

Remediation

Ensure consistent adherence to naming conventions throughout the project, and utilize a single variable to represent a specific value rather than employing multiple variables for the same purpose.

Patch

Fixed in efff5bf.

A | Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings can be found in the General Findings section.

Critical

Vulnerabilities that immediately lead to loss of user funds with minimal preconditions

Examples:

- Misconfigured authority or access control validation
- Improperly designed economic incentives leading to loss of funds

High

Vulnerabilities that could lead to loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions
- Exploitation involving high capital requirement with respect to payout

Medium

Vulnerabilities that could lead to denial of service scenarios or degraded usability.

Examples:

- Malicious input that causes computational limit exhaustion
- Forced exceptions in normal user flow

Low

Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions

Informational

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants
- Improved input validation

B | Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the implementation of the program requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of sum, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to get a comprehensive understanding of the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.