Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Versa Wallet

Veridise Inc.

August 23, 2023

► **Prepared For:**

Versa Lab
https://www.versawallet.io/

► **Prepared By:**

Yanju Chen
Himanshu
Bryan Tan

► **Contact Us:** contact@veridise.com

► **Version History:**

Aug. 23, 2023 V1

Contents

1 Executive Summary

From Jul. 26, 2023 to Aug. 10, 2023, Versa Lab engaged Veridise to review the security of their Versa Wallet, an on-chain, extensible wallet smart contract supporting EIP-4337* account abstraction. Veridise conducted the assessment over 6 person-weeks, with 3 engineers reviewing code over 2 weeks from commits `de6c674 - 56b82dd`. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers as well as extensive manual code review.

Project summary. The security assessment covered the Solidity smart contract implementation of the Versa Wallet, including the core wallet smart contract as well as several implementations of *plugins* that extend the wallet functionality. A Versa wallet has two "roles" in which user operations can be executed: a *normal validator* is able to perform *normal executions* that execute standard external calls from the wallet (with some limitations); and *sudo validators* can execute normal executions as well as *sudo executions* that are used to make arbitrary external calls or delegatecalls from the wallet (including calls to customize the wallet). The plugins are categorized into three major types: a *validator* is a custom smart contract for authenticating and authorizing user operations; a *hooks* contract can trigger custom callbacks on normal executions; and a *module* contract can execute arbitrary calls as the wallet. The validator implementations in the security assessment include an ECDSA signature validator, a multi-signature validator, and a session key-based validator.

Code assessment. The Versa Wallet developers provided the source code of the Versa Wallet contracts for review. To aid the Veridise auditors, the developers also provided a documentation website† that demonstrates the intended behavior of the project. The source code also contained some documentation in the form of READMEs and documentation comments on functions and storage variables. The source code is mostly original, but with some parts based on the account abstract reference implementation‡ and the Safe Contracts§ project.

The source code contained a test suite, which the Veridise auditors noted provides test coverage of common behaviors of the smart contracts. Several files in the source code also indicate that the developers use linting and static analysis tools such as Solhint. During the audit, the Versa Wallet developers made several additions to the code. This is because the Versa Wallet developers wanted to include an additional contract `SessionkeyValidator` in the audit scope. Consequently, the Veridise auditors extended the duration of the audit by two days.

Summary of issues detected. The audit uncovered 24 issues, 2 of which are assessed to be of high or critical severity by the Veridise auditors. Specifically, a malicious hooks contract can make its callbacks unremovable (V-VERS-VUL-002), and a reentrancy vulnerability allows a

* Also known as "Account Abstraction Using Alt Mempool": https://eips.ethereum.org/EIPS/eip-4337
† https://versawallet.gitbook.io/versa-docs/
‡ https://github.com/eth-infinitism/account-abstraction
§ https://github.com/safe-global/safe-contracts/

module to make itself unable to be disabled by the wallet (V-VERS-VUL-001). The Veridise auditors also identified several medium-severity issues, including a potential replay attack vector caused by validators not being included in the signed hashes (V-VERS-VUL-005), as well as an issue where ERC20 tokens with fees and/or interest may cause inconsistencies in a VersaVerifyingPaymaster's bookkeeping (V-VERS-VUL-004). The Veridise auditors further identified 8 low-severity issues, 8 warnings, and 2 informational findings.

The Versa Wallet developers resolved all of the reported issues.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Versa Wallet	`de6c674 - 56b82dd`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Jul. 26 - Aug. 10, 2023	Manual & Tools	3	6 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Resolved
Critical-Severity Issues	0	0
High-Severity Issues	2	2
Medium-Severity Issues	4	4
Low-Severity Issues	8	8
Warning-Severity Issues	8	8
Informational-Severity Issues	2	2
TOTAL	24	24

Table 2.4: Category Breakdown.

Name	Number
Logic Error	8
Data Validation	6
Maintainability	6
Replay Attack	2
Reentrancy	1
Usability Issue	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of Versa Wallet's smart contracts. In our audit, we sought to answer questions such as:

- How can compromised or malicious hooks contracts harm a wallet?
- Are there exploitable vulnerabilities in the signature parsing logic?
- Does the wallet correctly validate normal executions?
- Does the verifying paymaster omit any checks when validating its signatures?
- Are all data that need to be validated included as part of the hashes computed in the signature schemes?
- Are session key signatures correctly validated?
- Does the session key validator correctly restrict calls to the ones specified in the corresponding session key?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a combination of human experts and automated program analysis & testing tools. In particular, we conducted our audit with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, we leveraged our custom smart contract analysis tool Vanguard, as well as the open-source tool Slither. These tools are designed to find instances of common smart contract vulnerabilities, such as reentrancy and uninitialized variables.
- *Fuzzing/Property-based Testing.* We also leverage fuzz testing to determine if the protocol may deviate from the expected behavior. To do this, we formalize the desired behavior of the protocol as [V] specifications and then use our fuzzing framework OrCa to determine if a violation of the specification can be found.

Scope. The scope of this audit is limited to the `contracts` folder of the source code provided by the Versa Wallet developers, which contains the smart contract implementation of the Versa Wallet. During the audit, the Veridise auditors referred to external packages and services used by the Versa Wallet but assumed that they have been implemented correctly.

Methodology. The Veridise auditors inspected the provided tests, read the Versa Wallet documentation, and reviewed relevant documents such as the EIP-4337 specification. They then began a manual audit of the code assisted by both static analyzers and automated testing. During the audit, the Veridise auditors regularly met with the Versa Wallet developers to ask questions about the code.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-VERS-VUL-001	Malicious hook can avoid removal	High	Fixed
V-VERS-VUL-002	Reentrancy into disableModule() can make a modu	High	Acknowledged
V-VERS-VUL-003	Hooks that dynamically change hasHooks() may b	Medium	Acknowledged
V-VERS-VUL-004	Recorded token balances may not match actual am	Medium	Acknowledged
V-VERS-VUL-005	Signed hashes do not include validator address	Medium	Fixed
V-VERS-VUL-006	Instant transaction signatures have no expiry	Medium	Won't Fix
V-VERS-VUL-007	SpendingLimitHooks cannot account for spending	Low	Intended Behavior
V-VERS-VUL-008	resetTimeIntervalMinutes is assumed to be nonze.	Low	Fixed
V-VERS-VUL-009	lastResetTimeMinutes rounding logic not applied.	Low	Intended Behavior
V-VERS-VUL-010	ECDSA, MultiSigValidators will not be initializ...	Low	Intended Behavior
V-VERS-VUL-011	_getValidationIntersection reverts if both vali...	Low	Fixed
V-VERS-VUL-012	Missing length checks in _validateMultipleSessi...	Low	Fixed
V-VERS-VUL-013	isValidSignature does not check validity times ...	Low	Intended Behavior
V-VERS-VUL-014	validateUserOp() does not call _checkNormalExec.	Low	Intended Behavior
V-VERS-VUL-015	_enableHooks() bitfield check may be incorrect	Warning	Fixed
V-VERS-VUL-016	_isWalletInited() default implementation is err...	Warning	Fixed
V-VERS-VUL-017	Non-compliance in some ERC165 implementations	Warning	Acknowledged
V-VERS-VUL-018	Executor.execute() should explicitly check for ...	Warning	Won't Fix
V-VERS-VUL-019	Normal execution operation type check is error-...	Warning	Fixed
V-VERS-VUL-020	replace() allows special values to be inserted ...	Warning	Fixed
V-VERS-VUL-021	Signature length check does not capture actual ...	Warning	Fixed
V-VERS-VUL-022	Error-prone assumption that msg.sender and wall.	Warning	Fixed
V-VERS-VUL-023	Out of date doc comment in executeTransactionFr.	Info	Fixed
V-VERS-VUL-024	createAccount() reverts if wallet already exists	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-VERS-VUL-001: Malicious hook can avoid removal

Severity	High	**Commit**	de6c674
Type	Logic Error	**Status**	Fixed
File(s)	`HookManager.sol`		
Location(s)	disableHooks()		

A hooks on a Versa wallet can be disabled by calling the wallet's `HooksManager.disableHooks()` method. This will update the `beforeTxHooks` and `afterTxHooks` mappings, which store the hooks to execute before and after a transaction (respectively). The `_disableHooks()` function determines which mapping(s) to update by inspecting the bit field returned by the `hasHooks()` method.

If a hooks contract returns 0 from `hasHooks()`, then it will not be removed from `beforeTxHooks` and `afterTxHooks`.

```
function _disableHooks(address prevBeforeTxHook, address prevAfterTxHooks, address
    hooks) internal {
    // Try to clear wallet configurations
    try IHooks(hooks).clearWalletConfig() {
        emit DisabledHooks(hooks);
    } catch {
        emit DisabledHooksWithError(hooks);
    }
    // Remove hooks from exsiting linked list
    uint256 hasHooks = IHooks(hooks).hasHooks();
    if (hasHooks >> 128 == 1) {
        beforeTxHooks.remove(prevBeforeTxHook, hooks);
    }
    if (uint128(hasHooks) == 1) {
        afterTxHooks.remove(prevAfterTxHooks, hooks);
    }
}
```

Snippet 4.1: Definition of `_disableHooks()`

Impact A malicious hooks can exploit this vulnerability so that the before- and after-transaction callbacks will continue to be triggered even after the hooks is disabled. For example, a hooks contract can return an appropriate non-zero value in `hasHooks()` before it is initialized. As part of its initialization, it can change some storage variables so that `hasHooks()` will now return zero.

This vulnerability can prevent sudo validators from stopping malicious hooks from triggering on normal executions. For example, even after a malicious hook is disabled, the following behaviors could still occur:

- A hooks can always revert in one of its callbacks, causing a denial-of-service problem on normal executions.
- A malicious hooks could use the before-transaction callback to perform frontrunning attacks when a normal execution is used.

Note that hooks callbacks are not executed on sudo executions.

Recommendation Change `_disableHooks()` to always remove the hooks contract from the `beforeTxHooks` and `afterTxHooks` (if it is contained in them).

4.1.2 V-VERS-VUL-002: Reentrancy into disableModule() can make a module un-disableable

Severity	High	**Commit**	de6c674
Type	Reentrancy	**Status**	Acknowledged
File(s)	`ModuleManager.sol`		
Location(s)	_disableModule()		

The `disableModule()` method of a Versa wallet (inherited from the `ModuleManager` base class) can be called through a sudo execution to disable a module. This method will first attempt to call the `clearWalletConfig()` on the module to execute any module-specific logic to clear the configuration related to this wallet, and then it will remove the module from the wallet's list of enabled modules.

```
function _disableModule(address prevModule, address module) internal {
    try IModule(module).clearWalletConfig() {
        emit DisabledModule(module);
    } catch {
        emit DisabledModuleWithError(module);
    }
    modules.remove(prevModule, module);
}
```

Snippet 4.2: Definition of `_disableModule()`, which is the internal implementation of `disableModule()`

However, `_disableModule()` is theoretically prone to a reentrancy attack that allows a module to make itself un-disableable. For example, a malicious module could implement `clearWalletConfig()` in a way that causes the following behavior:

1. A sudo execution calls `disableModule()`, which then will make an external call to the module's `clearWalletConfig()` method.
2. Within the module's `clearWalletConfig()`, the module can use the wallet's `execTransactionFromModule()` method to invoke `disableModule()` again.
3. The nested call to `disableModule()` can invoke `clearWalletConfig()` on the module again; in this case, the module can be implemented in a way that it does not do anything on the second invocation to `clearWalletConfig()`.
4. The nested call to `disableModule()` will successfully remove the module from the module list and return.
5. Finally, control flow returns to the original call to `disableModule()`. Then the `modules.remove(prevModule, module)` line will be executed again, but this will cause a revert because the module has already been removed.

Impact If a module is malicious or becomes compromised, it can use this attack to make itself unremovable from the wallet. An attacker can exploit this vulnerability to increase the damage of an attack, and perhaps prevent a wallet from being recovered by its legitimate users.

Recommendation There are many possible mitigations, each with different trade-offs. We list some of them here:

- Add a reentrancy guard to related functions, e.g., `_disableModule`. This solution will prevent the exact attack described here, but it cannot prevent the attacker from performing similar attacks using `execTransactionFromModule()`.
- Implement a mechanism to disable `execTransactionFromModule()` from being called when a module is being disabled. This should prevent this attack as well as similar attacks, but at the cost of increasing the complexity of the `ModuleManager` logic.
- Remove the module from the module list before calling `clearWalletConfig()`. This will ensure that the module will no longer have access to the wallet functionalities before any module-controlled code is executed. However, this change will require some API changes in the modules, since the reference implementation of `clearWalletConfig()` currently requires the calling wallet to have the module on the enabled module list.

Developer Response The developers acknowledged the problem but do not intend to add any mitigations for the following reason:

> Modules are a security risk since they can execute arbitrary transactions. A malicious module can completely take over a Versa wallet like using `delegatecall` to change the wallet implementation and change the code behavior as whatever they like. So we think add restrictions to avoid `un-disableable` module might not help. To mitigate this issue, each module should be well audited to be added to Versa wallet. Additionally, we will provide our users with warnings to make them aware of the risks associated with using third-party modules and advise them to use modules with caution.

4.1.3 V-VERS-VUL-003: Hooks that dynamically change hasHooks() may be executed incorrectly

Severity	Medium	**Commit**	de6c674
Type	Logic Error	**Status**	Acknowledged
File(s)	HooksManager		
Location(s)	_beforeTransaction()		

When a user operation is executed with a normal validator, the "before transaction hooks" callbacks are executed before the actual operation, and the "after transaction hooks" callbacks are executed after the actual operation. This is done through the `HooksManager._beforeTransaction()` (resp. `HooksManager._afterTransaction()`) function, which will iterate through the hooks stored in `beforeTxHooks` (resp. `afterTxHooks`) and execute each hooks' callback. The actual entries in `beforeTxHooks` and `afterTxHooks` are created in `HooksManager.enableHooks()`, which will inspect the `hasHooks()` method of a hooks contract.

If the value of a hooks contract's `hasHooks()` function changes after the hooks is enabled for a wallet, then the `beforeTxHooks` and `afterTxHooks` will become inconsistent with the new value of `hasHooks()`.

```
/**
 * @dev Loop through the beforeTransactionHooks list and execute all before
    transaction hooks.
 * @param to The address of the transaction recipient.
 * @param value The value of the transaction.
 * @param data The data of the transaction.
 * @param operation The type of operation being performed.
 */
function _beforeTransaction(address to, uint256 value, bytes memory data, Enum.
    Operation operation) internal {
    address addr = beforeTxHooks[AddressLinkedList.SENTINEL_ADDRESS];
    while (uint160(addr) > AddressLinkedList.SENTINEL_UINT) {
        IHooks(addr).beforeTransaction(to, value, data, operation);
        addr = beforeTxHooks[addr];
    }
}
```

Snippet 4.3: Definition of `_beforeTransaction`

```
uint256 hasHooks = IHooks(hooks).hasHooks();
if (hasHooks >> 128 == 1) {
    beforeTxHooks.add(hooks);
}
if (uint128(hasHooks) == 1) {
    afterTxHooks.add(hooks);
}
```

Snippet 4.4: Snippet in `_enableHooks()` that adds a hooks to the appropriate lists.

Impact Such a situation may arise if a hooks contract is upgradeable or acting maliciously. For example, if a hooks originally only requires to be run before a transaction, but it is upgraded to

also run after the transaction, then the `afterTxHooks` will not be updated unless a sudo execution disables and enables the hooks again. This can result in bugs in the hooks if it requires the after-transaction code to run.

Recommendation The developers could adopt one or more possible mitigations depending on what the intended behavior of hooks should be in this scenario:

- To allow for upgradeable hooks or other hooks whose `hasHooks()` value can dynamically change, the developers could change the `HooksManager` to use a single variable to store *all* enabled hooks. Then, `_beforeTransaction` and `_afterTransaction` can be changed so that they iterate over all hooks and check the `hasHooks()` to determine whether the hooks has a before (resp. after) callback.
- If it is not intended for `hasHooks()` to change dynamically, the developers should clearly document this assumption and warn end-users to avoid hooks that are upgradeable.

Developer Response The developers noted that "Hooks are not supposed to change their supported hooks type dynamically" and have added a documentation comment on `_enableHooks` noting as such.

4.1.4 V-VERS-VUL-004: Recorded token balances may not match actual amount transferred

Severity	Medium	**Commit**	de6c674
Type	Logic Error	**Status**	Acknowledged
File(s)	`VersaVerifyingPaymaster.sol`		
Location(s)	_postOp()		

The `VersaVerifyingPaymaster` is an EIP-4337 paymaster that allows an account to pay for user operations using ERC-20 tokens. It collects the token fees in the `_postOp()` method, which is called after the user operation is executed. To charge the fees, the paymaster will call `token.safeTransferFrom(...)` to transfer `actualTokenCost` tokens from the account to the paymaster and then increase the `balances[token]` mapping entry by `actualTokenCost`.

The balance update logic assumes that the amount of tokens transferred is `actualTokenCost`, but the amount may be different in practice. For example, some tokens may charge fees, accrue interest, or rebase, which could result in the `actualTokenCost` differing from the actual amount transferred.

```
    if (mode != PostOpMode.postOpReverted) {
        token.safeTransferFrom(account, address(this), actualTokenCost);
        balances[token] += actualTokenCost;
        emit UserOperationSponsored(account, address(token), actualTokenCost);
    }
```

Snippet 4.5: Relevant lines in `_postOp()`

Impact The `balances` mapping is used in the `withdrawTokensTo()` method, which the owner of the paymaster can call to withdraw any fees that have been paid. If an entry in `_token` is inconsistent with the actual number of tokens, then `withdrawTokensTo()` is likely to revert (either when subtracting `_amount` or in the `_token.safeTransfer()` call).

```
function withdrawTokensTo(IERC20Metadata _token, address _target, uint256 _amount)
    external onlyOwner {
    balances[_token] -= _amount;
    _token.safeTransfer(_target, _amount);
}
```

Snippet 4.6: Definition of `withdrawTokensTo()`

Recommendation Some potential mitigations include:

- To deal with fees or interest, the code could be changed so that it keeps track of the `token.balanceOf(...)` before and after the transfer. The `balances[token]` entry should then be increased by the difference between the recorded values.
- To deal with rebasing tokens, the code may need to be modified to also account for changes in the token's total supply.

- The paymaster contract could store a list of tokens that are allowed to be used for payments (and can be dynamically adjusted by the owner). This would also have the advantage of allowing the owner to revoke supported tokens.

Developer Response The developers stated that "[t]his paymaster service is managed by us, and we will only allow the use of sufficiently secure ERC20 tokens to pay for gasfee."

4.1.5 V-VERS-VUL-005: Signed hashes do not include validator address

Severity	Medium	**Commit**	56b82dd
Type	Replay Attack	**Status**	Fixed
File(s)		`SignatureHandler.sol`	
Location(s)		splitUserOpSignature()	

The signature format defined by the Versa wallet requires the address of the target validator to be stored in the first 20 bytes of the signature data. When the `ECDSAValidator` and `MultiSignatureValidator` parse the signature data using the `splitUserOpSignature` function, the validator address is not included in the hash. This may introduce a small risk of replay attacks on user operations, where an attacker could keep the signature the same but modify the validator address.

```
if (splitedSig.signatureType == INSTANT_TRANSACTION) {
    splitedSig.signature = userOp.signature[INSTANT_SIG_OFFSET:];
    splitedSig.hash = userOpHash;
} else if (splitedSig.signatureType == SCHEDULE_TRANSACTION) {
    // ...
        bytes memory extraData = abi.encode(
        splitedSig.validUntil,
        splitedSig.validAfter,
        splitedSig.maxFeePerGas,
        splitedSig.maxPriorityFeePerGas
    );
    splitedSig.hash = keccak256(abi.encode(userOpHash, extraData));
}
```

Snippet 4.7: Relevant lines in `splitUserOpSignature()`

Impact The signature scheme for `splitUserOpSignature()` is used by `ECDSAValidator` and `MultiSigValidator`. An attacker could exploit the missing validator address check to perform replay attacks in specific scenarios. For example, consider a situation where:

- The wallet has two `MultiSigValidators` enabled.
  - One of the `MultiSigValidators` is a validator that requires two signatures, with guardians Alice and Bob.
  - One of the `MultiSigValidators` is a validator that requires three signatures, with guardians Alice, Bob, Charlie, and Eve.
- Alice, Bob, and Charlie sign a user op to be used with the second validator. They submit it to an Ethereum client's user op mempool, but it is rejected by the validation step because Charlie's signature is invalid.

If the attacker observes the user op in the user operation mempool, then they can replace the first validator's address with the second validator's address and then resubmit the operation to the mempool. This would then allow the method to be executed using the first validator, even though Charlie did not provide a valid signature and Alice and Bob meant for the user op to only be executed through the second validator with Charlie's approval.

Recommendation Include the `validatorAddress` as part of the hash.

4.1.6 V-VERS-VUL-006: Instant transaction signatures have no expiry

Severity	Medium	**Commit**	56b82dd
Type	Replay Attack	**Status**	Won't Fix
File(s)	`SignatureHandler.sol`		
Location(s)	splitUserOpSignature()		

The signature scheme described in `SignatureHandler.sol` describes two types of transactions: instant transactions and scheduled transactions. While scheduled transactions include fields for EIP-4337 `validUntil` and `validAfter` timestamps, instant transactions do not. Consequently, there is no way to attach expiry times to instant transactions, so instant transactions are always valid once signed.

```
if (splitedSig.signatureType == INSTANT_TRANSACTION) {
    splitedSig.signature = userOp.signature[INSTANT_SIG_OFFSET:];
    splitedSig.hash = userOpHash;
} else if (splitedSig.signatureType == SCHEDULE_TRANSACTION) {
```

Snippet 4.8: Relevant lines in `splitUserOpSignature()`. The hash to be signed only consists of the `userOpHash` provided by the EIP-4337 entrypoint contract.

Impact Due to the missing timestamp expiry logic, instant transactions may be more vulnerable to replay attacks. For example, consider a scenario where a user operation is submitted to an Ethereum client's user operation mempool, but the operation is rejected at the validation step (e.g., for reasons such as not enough gas). An attacker can save the operation and replay it at a later date, assuming that the signer has already "given up" on the operation and that the validator is still enabled on the wallet. Such an attack could be performed even after a long period of time has passed, and the signer no longer intends for the operation to be executed.

Recommendation Include the `validUntil` and `validAfter` dates as part of the signature for instant transactions. For "instant" transactions, signers should set the `validAfter` to occur in the near future (e.g., within 30 minutes).

Developer Response The developers noted that "instant transactions" are meant to emulate transactions sent by an externally-owned account. The developers do not intend to extend the instant transaction type with timestamps. Instead, the developers suggest that users should submit a newer user operation with the same nonce and a higher gas value in order to revoke previous user operations.

4.1.7 V-VERS-VUL-007: SpendingLimitHooks cannot account for spending on extra methods added by custom ERC20 tokens

Severity	Low	**Commit**	de6c674
Type	Logic Error	**Status**	Intended Behavior
File(s)	`SpendingLimitHooks.sol`		
Location(s)	_checkERC20TokenSpendingLimit()		

The `SpendingLimitHooks` contract is a hooks contract that can be used to impose a "spending limit" on normal executions that will transfer native currency and/or ERC20 tokens. The spending limits for ERC20 tokens are implemented in the `_checkERC20TokenSpendingLimit()` function, which checks the selector of the ERC20 method call and reverts if the call would put the wallet above the spending limit for that token. However, the spending limit is only applied to standard ERC20 methods such as `transfer()` and `transferFrom()`, and it will not be applied to extra or custom methods that the token may also support.

```
function _checkERC20TokenSpendingLimit(address _wallet, address _token, bytes
    calldata _data) internal {
    ...
        if (methodSelector == TRANSFER || methodSelector == INCREASE_ALLOWANCE) {
            ...
        } else if (methodSelector == TRANSFER_FROM) {
            ...
        } else if (methodSelector == APPROVE) {
            ...
        }
    ...
}
```

Snippet 4.9: Relevant code in `_checkERC20TokenSpendingLimit()`

Impact If the token contract supports custom methods that transfers tokens or adjusts approvals, then normal executions can evade the spending limit by calling these custom methods.

Recommendation Some mitigations, each with different trade-offs, include:

- Add an `else` case that will revert, so that the only allowed calls are to one of the methods handled in one of the `if` branches. This will prevent normal executions from attempting to evade the spending limit checks by calling custom methods, but it may also prevent users from using some features of the tokens.
- Allow calls to other methods and clearly document the limitations of the `SpendingLimitHooks` contracts.

Developer Response The developers opted to allow calls to other methods and documented the limitations of the `SpendingLimitHooks`.

4.1.8 V-VERS-VUL-008: resetTimeIntervalMinutes is assumed to be nonzero but not checked

Severity	Low	**Commit**	de6c674
Type	Data Validation	**Status**	Fixed
File(s)	SpendingLimitHooks.sol		
Location(s)	_setSpendingLimit()		

In the `_setSpendingLimit()` function, the value of `config.resetTimeIntervalMinutes` is assumed to be nonzero, but there is no check that is it is indeed nonzero. If a value of zero is used, then the function may revert when the value of `spendingLimitInfo.lastResetTimeMinutes` is calculated, since the `config.resetTimeIntervalMinutes` is used as the right-hand-side of a modulus operation. We note that other methods such as `getSpendingLimitInfo()`, which perform similar calculations, do have checks that `_config.resetTimeIntervalMinutes` is nonzero.

```
if (_config.resetBaseTimeMinutes > 0) {
    require(
        _config.resetBaseTimeMinutes <= currentTimeMinutes,
        "SpendingLimitHooks: resetBaseTimeMinutes can not greater than
    currentTimeMinutes"
    );
    spendingLimitInfo.lastResetTimeMinutes =
        currentTimeMinutes -
        ((currentTimeMinutes - _config.resetBaseTimeMinutes) % _config.
    resetTimeIntervalMinutes);
} else if (spendingLimitInfo.lastResetTimeMinutes == 0) {
  // ...
}
```

Snippet 4.10: Relevant lines in `_setSpendingLimit()`

Recommendation Add code that handles the situation when `_config.resetTimeIntervalMinutes` is zero.

4.1.9 V-VERS-VUL-009: lastResetTimeMinutes rounding logic not applied in one of the cases

Severity	Low	**Commit**	de6c674
Type	Logic Error	**Status**	Intended Behavior
File(s)	`SpendingLimitHooks.sol`		
Location(s)	_setSpendingLimit()		

The `SpendingLimitHooks` hooks contract can be used to set an "allowance" or "spending limit" of a given currency over a period of time. It can optionally be configured to periodically reset the spending limit.

A spending limit reset may occur when a sudo validator calls the `_setSpendingLimit()` function to enable or reconfigure a spending limit for a specified currency. There are two cases in this function that may adjust the `lastResetTimeMinutes`; however, in the case where the `_config.resetBaseTimeMinutes == 0` and `spendingLimitInfo.lastResetTimeMinutes == 0`, the `lastResetTimeMinutes` will be directly set to the `currentTimeMinutes`. This is inconsistent with the code in the other case.

```
function _setSpendingLimit(SpendingLimitSetConfig memory _config) internal {
        // ...
        if (_config.resetBaseTimeMinutes > 0) {
          require(
              _config.resetBaseTimeMinutes <= currentTimeMinutes,
              "SpendingLimitHooks: resetBaseTimeMinutes can not greater than
    currentTimeMinutes"
          );
          spendingLimitInfo.lastResetTimeMinutes =
              currentTimeMinutes -
              ((currentTimeMinutes - _config.resetBaseTimeMinutes) % _config.
    resetTimeIntervalMinutes);
      } else if (spendingLimitInfo.lastResetTimeMinutes == 0) {
          spendingLimitInfo.lastResetTimeMinutes = currentTimeMinutes;
      }
      spendingLimitInfo.resetTimeIntervalMinutes = _config.resetTimeIntervalMinutes;
      spendingLimitInfo.allowanceAmount = _config.allowanceAmount;
      _updateSpendingLimitInfo(_config.tokenAddress, spendingLimitInfo);
        // ...
```

Snippet 4.11: Relevant lines in `_setSpendingLimit()`

Furthermore, the case above is also inconsistent with the `getSpendingInfo()` method, which returns a `SpendingLimitInfo` struct with the following fields:

- The `lastResetTimeMinutes`, if nonzero, stores the last time the spending limit was reset.
- The `resetTimeIntervalMinutes`, if nonzero, stores the time that must pass after the `lastResetTimeMinutes` before the spending limit can be reset.

In `getSpendingInfo()`, the `SpendingLimitInfo` will first be retrieved from storage, and then its `lastResetTimeMinutes` to be checked to determine if the spending limit can be reset. If so, the `lastResetTimeMinutes` will be set to:

```
lastResetTimeMinutes + c * resetTimeIntervalMinutes
```


where `c` is the number of full intervals passed since the `lastResetTimeMinutes` (note that the actual formula in the code can be shown to be equivalent to this form).

```
function getSpendingLimitInfo(address _wallet, address _token) public view returns (
    SpendingLimitInfo memory) {
    SpendingLimitInfo memory spendingLimitInfo = _tokenSpendingLimitInfo[_wallet][
    _token];
    uint32 currentTimeMinutes = uint32(block.timestamp / 60);
    if (
        spendingLimitInfo.resetTimeIntervalMinutes > 0 &&
        spendingLimitInfo.lastResetTimeMinutes + spendingLimitInfo.
    resetTimeIntervalMinutes <= currentTimeMinutes
    ) {
        spendingLimitInfo.spentAmount = 0;
        spendingLimitInfo.lastResetTimeMinutes =
            currentTimeMinutes -
            ((currentTimeMinutes - spendingLimitInfo.lastResetTimeMinutes) %
                spendingLimitInfo.resetTimeIntervalMinutes);
    }
    return spendingLimitInfo;
}
```

Snippet 4.12: Definition of `getSpendingInfo()`

Impact If a sudo validator calls `setSpendingLimit()` without specifying a `resetBaseTimeMinutes` or a `lastResetTimeMinutes`, then the `lastResetTimeMinutes` will be set to the current time. Consequently, the spending limit will not be reset until the next interval.

Recommendation The developers may want to consider whether this behavior is intended and adjust the `else if` case accordingly.

Developer Response The developers noted that this is intended behavior and have added comments in the relevant parts of the code explaining why this is the case.

4.1.10 V-VERS-VUL-010: ECDSA, MultiSigValidators will not be initialized when re-enabled

Severity	Low	**Commit**	56b82dd
Type	Logic Error	**Status**	Intended Behavior
File(s)	`ECDSAValidator.sol, MultiSigValidator.sol`		
Location(s)	_clear()		

The `ECDSAValidator` and `MultiSigValidator` implement the `_clear()` method as a no-op, and comments on the corresponding files indicate that this is intended behavior. However, this is unexpected behavior from a user's perspective because a user may want to re-enable a validator with a new set of signers. As an example, consider the following scenario:

1. User of the wallet disables an ECDSA or MultiSigValidator. Because the `_clear()` method is a no-op, this does not clear any existing signers.
2. The user enables the previously disabled validator for the wallet, but supplies a new set of signers as initialization data.
3. Because the validator is already "initialized", the `_init()` method is **not** executed, and the new signers are ignored.

Impact Since the new signers are ignored, a user may unintentionally create access control problems when re-enabling a validator. For example, if the user intends to revoke signing permission by disabling a validator, and then attempts to re-enable the validator with a new set of signers, then the old signers may still submit user operations for the wallet even though their access should have been revoked.

Recommendation Implement the `_clear()` method to remove all existing signers.

Developer Response The developers stated that this is intended behavior:

> If an user want to re-enable an `ECDSAValidator` or a `MultiSigValidator`, our front-end will let the user to set a new signer / set of signers. If the new signers do not match the initial ones, our front-end will batch a `setSigner` or `resetGuardians` transaction to update user's new signers which in some cases may save some gas as some slot read and write can be avoid.

4.1.11 V-VERS-VUL-011: _getValidationIntersection reverts if both validUntil times are 0

Severity	Low	Commit	56b82dd
Type	Logic Error	Status	Fixed
File(s)	SessionkeyValidator.sol		
Location(s)	_getValidationIntersection()		

The function `_getValidationIntersection()` is used to calculate the "intersection" of the validity intervals of two sessions. The end of the two sessions are provided as the `validUntil1` and `validUntil2` arguments, respectively. A session may have a `validUntil` value set to 0 to indicate a validity interval with no expiry time.

```
function _getValidationIntersection(
    uint48 validUntil1,
    uint48 validunitil2,
    uint48 validAfter1,
    uint48 validAfter2
) internal pure returns (uint48 validUntil, uint48 validAfter) {
    if (validUntil1 != 0 && validunitil2 != 0) {
        validUntil = validUntil1 < validunitil2 ? validUntil1 : validunitil2;
    } else {
        validUntil = validUntil1 > validunitil2 ? validUntil1 : validunitil2;
    }
    validAfter = validAfter1 > validAfter2 ? validAfter1 : validAfter2;
    require(validUntil >= validAfter, "SessionKeyValidator: invalid validation
    duration");
}
```

Snippet 4.13: Definition of `_getValidationIntersection()`

```
struct Session {
        // ...
    // The timestamp when the session is expired, 0 for infinite
    uint48 validUntil;
    // ...
}
```

Snippet 4.14: Definition of struct `Session`

A require statement at the end of the function checks that the interval is nonempty (by asserting that the end time of the validity interval cannot come before the start time). However, if both `validUntil` and `validUntil2` are both zero, then the condition will evaluate to false, causing the function to revert. This appears to be unintended: the intersection of the validity intervals of two sessions with a `validUntil` of 0 (i.e., infinite) should correspond to a valid interval with an infinite end time.

Impact The `_getValidationIntersection()` function is used when validating a batched execution. Thus, if the list of sessions has two adjacent sessions that both have a `validUntil` time of 0, then the function will revert and cause the user operation to be rejected.

Recommendation Modify `_getValidationIntersection` to account for the case where both `validUntil` values are 0.

4.1.12 V-VERS-VUL-012: Missing length checks in _validateMultipleSessions()

Severity	Low	**Commit**	56b82dd
Type	Data Validation	**Status**	Fixed
File(s)	SessionkeyValidator.sol		
Location(s)	_validateMultipleSessions()		

The `_validateMultipleSessions()` function is invoked when the `SessionkeyValidator` is used to validate the signature of a call to `VersaWallet.batchNormalExecute()`. One of the checks performed during this validation is to ensure that the correct amount of information is provided as part of the signature. As indicated by the loop, the `data`, `proof`, `session`, `rlpCalldata`, `to`, and `value` arrays are assumed to be the same length. However, only the lengths of `to`, `session`, and `proof` are checked to be equal.

```
function _validateMultipleSessions(
    /* ... other parameters ... */,
    bytes32[][] memory proof,
    Session[] memory session,
    bytes[] memory rlpCalldata,
    address[] memory to,
    uint256[] memory value,
    bytes[] memory data
) internal returns (uint48 validUntil, uint48 validAfter) {
        require(
            to.length == session.length && session.length == proof.length,
            "SessionKeyValidator: invalid batch length"
        );
        address paymaster = _parsePaymaster(userOp.paymasterAndData);
        for (uint256 i = 0; i < data.length; i++) {
            // ...
            _validateSession(
                /* ... other parameters ... */
                proof[i],
                session[i],
                rlpCalldata[i],
                paymaster,
                to[i],
                value[i],
                data[i]
            );
        }
```

Snippet 4.15: Relevant lines in `_validateMultipleSessions()`

Impact The submitter of the user operation may append extra `data`, `session`, `value`, `rlpCalldata` or other fields, but the validator will silently ignore the extra data and accept the user operation instead of rejecting it. This could potentially result in the operation performing an action that the user does not intend to be performed.

Furthermore, `batchNormalExecute()` will check that the lengths of `to`, `value`, and `data` are equal. If the validator accepts a user operation in which the lengths of `to`, `value`, and `data` are not equal,

then `batchNormalExecute()` will revert during the actual execution of the user operation. This will waste the gas fees paid by the wallet or paymaster.

```
function batchNormalExecute(
    address[] memory to,
    uint256[] memory value,
    bytes[] memory data,
    Enum.Operation[] memory operation
) external onlyFromEntryPoint {
    _checkBatchDataLength(to.length, value.length, data.length, operation.length);
    // ...
}

function _checkBatchDataLength(
    uint256 toLen,
    uint256 valueLen,
    uint256 dataLen,
    uint256 operationLen
) internal pure {
    require(toLen == valueLen && dataLen == operationLen && toLen == dataLen, "Versa:
     invalid batch data");
}
```

Snippet 4.16: Relevant lines in `VersaWallet`

Recommendation Add the missing length checks to `_validateMultipleSessions()`.

Developer Response The developers added a check for `rlpCalldata`, but they did not add the remaining checks for the following reason:

> We don't check `to, value, data`'s length to be equal here as they are checked at `VersaWallet.batchNormalExecute()` . The user operation will be rejected during the bundler `estimateGas` call if a transaction has invalid data length as `estimateGas` will simulate the transaction and revert on the invalid data length. So the user operation will not able to be actually sent to the bundler mempool.

The auditors note that EIP-4337 only requires bundlers to simulate validation, but not execution; so this issue may occur with any bundler that does not simulate the execution.

4.1.13 V-VERS-VUL-013: isValidSignature does not check validity times of scheduled transactions

Severity	Low	**Commit**	56b82dd
Type	Data Validation	**Status**	Intended Behavior
File(s)	`ECDSAValidator.sol`		
Location(s)	isValidSignature()		

A Versa wallet supports the ERC-1271 standard for on-chain signature validation. This is implemented in the `CompatibilityFallbackHandler.isValidSignature()` method, which delegates the actual signature validation to a validator implementation.

```
function isValidSignature(
    bytes32 _hash,
    bytes calldata _signature
) public view override returns (bytes4 magicValue) {
    address validator = address(bytes20(_signature[0:20]));
    require(
        ValidatorManager(msg.sender).getValidatorType(validator) == ValidatorManager.
    ValidatorType.Sudo,
        "Only sudo validator"
    );
    bool isValid = IValidator(validator).isValidSignature(_hash, _signature[20:], msg
    .sender);
    return isValid ? EIP1271_MAGIC_VALUE : bytes4(0xffffffff);
}
```

Snippet 4.17: Definition of `CompatibilityFallbackHandler.isValidSignature()`.

The signature format used by an `ECDSAValidator` (as defined in `SignatureHandler.sol`) defines two transaction "types": instant transactions and scheduled transactions. The latter require the signature to contain the `validUntil` and `validAfter` validity times (corresponding to the timestamps specified by EIP-4337).

However, the `ECDSAValidator` implementation of `isValidSignature()` currently assumes that the actual validity times are 0 when validating the signature. For scheduled transactions, this would be weaker than the check performed for user operations.

```
function isValidSignature(bytes32 hash, bytes calldata signature, address wallet)
    external view returns (bool) {
    uint256 validUntil;
    uint256 validAfter;
    address signer = _signers[wallet];
    uint256 validationData = _validateSignature(signer, signature, hash, validUntil,
    validAfter);
    return validationData == 0 ? true : false;
}
```

Snippet 4.18: Definition of `ECDSAValidator.isValidSignature()`. The `_validateSignature()` call will return the packed validation format described by EIP-4337. But since the `validUntil` and `validAfter` values are set to 0 here, a successful validation will also have the validity times set to 0.

Impact When an `ECDSAValidator` is used as an ERC-1271 signature validator, the `isValidSignature()` method will accept signatures of scheduled transactions that are already expired.

Recommendation Require users of the ERC-1271 interface to provide the full signature data described in `SignatureHandler.sol` so that fields such as `validUntil` and `validAfter` can be validated.

Developer Response The developers noted that:

> The `isValidSignature` function is for ERC-1271 support to verify whether a signature on a behalf of a given contract is valid. It's not meat to verify 4337 transactions but messages generated by third-party dapps. For example, an user need to sign the OpenSea Term of Service and Privacy Policy and other related messages to connect to Opensea. So we provide this function to verify signature signed by the wallet owner. The validity times is an option for these dapps to be added to the message rather than for the `isValidSignature` function to enforce.

4.1.14 V-VERS-VUL-014: validateUserOp() does not call _checkNormalExecute()

Severity	Low	Commit	56b82dd
Type	Data Validation	Status	Intended Behavior
File(s)	VersaWallet.sol		
Location(s)	validateUserOp()		

The `VersaWallet.validateUserOp()` function is called by the Ethereum client and EIP-4337 entrypoint to check whether a user operation should be executed. However, it is missing a call to the `_checkNormalExecute()` operation, which restricts normal executions to specific targets and operation types.

```
function validateUserOp(
    UserOperation calldata userOp,
    bytes32 userOpHash,
    uint256 missingAccountFunds
) external override onlyFromEntryPoint returns (uint256 validationData) {
    address validator = _getValidator(userOp.signature);
    _validateValidatorAndSelector(validator, bytes4(userOp.callData[0:4]));
    validationData = IValidator(validator).validateSignature(userOp, userOpHash);
    _payPrefund(missingAccountFunds);
}
```

Snippet 4.19: Definition of `validateUserOp()`

```
/**
 * @dev A normal execution has following restrictions:
 * 1. Cannot selfcall, i.e., change wallet's config
 * 2. Cannot call to an enabled plugin, i.e, change plugin's config or call wallet
    from plugin
 * 3. Cannot perform a delegatecall
 * @param to The address to which the transaction is directed.
 * @param _operation The operation type of the transaction.
 */
function _checkNormalExecute(address to, bytes memory data, Enum.Operation _operation
    ) internal view {
    require(
        (to != address(this) || (to == address(this) && data.length == 0)) &&
            !isValidatorEnabled(to) &&
            !isHooksEnabled(to) &&
            !isModuleEnabled(to) &&
            _operation != Enum.Operation.DelegateCall,
        "Versa: operation is not allowed"
    );
}
```

Snippet 4.20: Definition of `_checkNormalExecute()`

Impact This currently should have limited impact in terms of the correctness of the access controls, as normal executions will result in a call to `_checkNormalExecute()`. However, if a normal execution validates successfully but reverts due to `_checkNormalExecute()`, then any gas paid by the wallet or paymaster will have been wasted.

Recommendation Modify `validateUserOp()` so that `_checkNormalExecute()` will be called when the user operation is a normal execution.

Developer Response The developers do not intend to insert the check for the following reason:

> The user operation described below will be rejected by the bundler during `eth_estimateUserOperationGas` call, as `eth_estimateUserOperationGas` will simulate the user operation and revert on any unmet conditions. We will chose bundler that has the simulation implementation to avoid actual gas lost.

4.1.15 V-VERS-VUL-015: _enableHooks() bitfield check may be incorrect

Severity	Warning	**Commit**	de6c674
Type	Maintainability	**Status**	Fixed
File(s)	`HooksManager.sol`		
Location(s)	_enableHooks()		

When a Versa wallet enables a hooks contract in `HooksManager.enableHooks()`, it checks the `hasHooks()` method of the hooks to determine when the hooks' callbacks should be executed. The `hasHooks()` method returns a bit field (as a `uint256`) where:

- the least significant bit is asserted if the hook should be executed after a transaction
- bit 129 (assuming the most significant bit is "bit 256") is asserted if the hook should be executed before a transaction

However, the way these checks are implemented may be prone to errors, especially if the developers intend to support additional flags in the bit field:

- When checking the flag indicating an after-transaction hook, the lower 128 bits are compared with the integer 1. This means that if bits 2 to 128 are used, then the comparison will fail.
- Similarly, when checking the flag indicating a before-transaction hook, the upper 128 bits are compared with the integer 1. This means that if bits 130 to 256 are used, then the comparison will fail.

```
uint256 hasHooks = IHooks(hooks).hasHooks();
if (hasHooks >> 128 == 1) {
    beforeTxHooks.add(hooks);
}
if (uint128(hasHooks) == 1) {
    afterTxHooks.add(hooks);
}
```

Snippet 4.21: Location in `_enableHooks()` where `hasHooks()` bitfield is used.

Recommendation To make the code more robust, the additional bits should be cleared before performing the comparison:

- Change `hasHooks >> 128` to `(hasHooks >> 128) & 1`
- Change `uint128(hasHooks)` to `hasHooks & 1`

4.1.16 V-VERS-VUL-016: _isWalletInited() default implementation is error-prone

Severity	Warning	**Commit**	de6c674
Type	Maintainability	**Status**	Fixed
File(s)	`BaseHooks.sol, BaseValidator.sol`		
Location(s)	_isWalletInited()		

The `BaseHooks` and `BaseValidator` contracts provide some convenience functions for implementing hooks and validators, respectively. The two base contracts provide a default implementation of `_isWalletInited()` that always returns false.

The `_isWalletInited()` method is used in `initWalletConfig()` to determine whether the initialization code should be invoked; and in `clearWalletConfig()` to determine whether the deinitialization code should be invoked.

```
function initWalletConfig(bytes memory _data) external onlyEnabledHooks {
    if (!_isWalletInited(msg.sender)) {
        _init(_data);
        emit InitWalletConfig(msg.sender);
    }
}

/**
 * @dev Clears the wallet configuration. Triggered when disabled by a wallet
 */
function clearWalletConfig() external onlyEnabledHooks {
    if (_isWalletInited(msg.sender)) {
        _clear();
        emit ClearWalletConfig(msg.sender);
    }
}

function _isWalletInited(address wallet) internal view virtual returns (bool) {}
```

Snippet 4.22: The relevant functions in `BaseHooks`. The ones in `BaseValidator` are similar.

The default implementation of `_isWalletInited()` is error-prone, since it will always cause the wallet configuration to be initialized when the hooks or validator is enabled, but never cleared when the hooks or validator is disabled. This can cause a developer that forgets to override `_isWalletInited()` to introduce bugs into their hooks or validator implementation.

Recommendation Remove the default implementation of `_isWalletInited()` and make it pure virtual, so that subclasses of `BaseHooks` and `BaseValidator` will be forced to implement the method.

4.1.17 V-VERS-VUL-017: Non-compliance in some ERC165 implementations

Severity	Warning	**Commit**	de6c674
Type	Logic Error	**Status**	Acknowledged
File(s)	`BaseValidator.sol`, `BaseHooks.sol`		
Location(s)	supportsInterface()		

The ERC-165 standard specifies a `supportInterface()` method that returns true when the called smart contract supports a given function interface. `BaseValidator` and `BaseHooks` implement the `IERC165` interface, but they do not return true (as required by the specification) when the ERC-165 interface ID is provided as the argument.

```
contract BaseValidator{
        function supportsInterface(bytes4 interfaceId) external pure returns (bool) {
            return interfaceId == type(IValidator).interfaceId;
      }
}
```

Snippet 4.23: Definition of `BaseValidator.supportsInterface()`

```
contract BaseHooks{
        function supportsInterface(bytes4 _interfaceId) external pure returns (bool)
    {
            return _interfaceId == type(IHooks).interfaceId;
      }
}
```

Snippet 4.24: Definition of `BaseHooks.supportsInterface()`

Impact Third-party smart contracts which dynamically check support for ERC-165 using the algorithm suggested in the specification may falsely mark hooks and validators as not supporting ERC-165.

Recommendation Change the implementations of `supportsInterface()` so that they return true when passed the ERC165 interface ID. To help with testing, the developers may want to consider adding test cases that use OpenZeppelin's `ERC165Checker` library to perform the check.

Developer Response The developer noted that they only implemented `supportsInterface()` to allow the manager classes to dynamically check whether an address is a hooks or validator contract, so they think there is no need to fix this issue at this time.

4.1.18 V-VERS-VUL-018: Executor.execute() should explicitly check for the normal call operation

Severity	Warning	**Commit**	de6c674
Type	Maintainability	**Status**	Won't Fix
File(s)	`Executor.sol`		
Location(s)	execute()		

The `execute()` helper function is used to implement a call supported by the Versa wallet. Currently, this only includes two types of calls: a normal external call and a delegatecall. The way the function is implemented currently is error-prone: if the developer adds an operation type to the enum `Enum.Operation` but forgets to add the corresponding case in `execute()`, then any operation of that new type will be executed as if it is a normal call.

```
function execute(...) internal returns(bool success){
    if (operation == Enum.Operation.DelegateCall) {
        // delegatecall
    }
    else {
        // call
    }
}
```

Snippet 4.25: Relevant lines in `execute()`

Recommendation Change the `else` case to an `else if (operation == Enum.Operation.Call) { ... }` and add an additional `else` case that reverts.

4.1.19 V-VERS-VUL-019: Normal execution operation type check is error-prone

Severity	Warning	**Commit**	56b82dd
Type	Maintainability	**Status**	Fixed
File(s)	VersaWallet.sol		
Location(s)	_checkNormalExecute()		

The `_checkNormalExecute()` function is used to restrict normal executions (i.e., user operations made with a normal validator). One of these restrictions is that the call cannot use be a delegatecall. Since the only other type of operation supported by a Versa wallet is a normal external call, this means that normal executions can only access normal calls.

However, the way this is currently implemented is error-prone, as it checks that the operation is ***not*** a delegatecall and permits all other types of calls. If the developers add additional types of operation in the future, they may forget to update `_checkNormalExecute()`. This can lead to access control vulnerabilities.

```
function _checkNormalExecute(address to, bytes memory data, Enum.Operation _operation
    ) internal view {
    require(
        (to != address(this) || (to == address(this) && data.length == 0)) &&
            !isValidatorEnabled(to) &&
            !isHooksEnabled(to) &&
            !isModuleEnabled(to) &&
            _operation != Enum.Operation.DelegateCall,
        "Versa: operation is not allowed"
    );
}
```

Snippet 4.26: Implementation of `_checkNormalExecute()`

```
abstract contract Enum {
    enum Operation {
        Call,
        DelegateCall
    }
}
```

Snippet 4.27: The two types of calls supported by a Versa wallet.

Recommendation Instead of a "allow by default" policy that checks for operations to deny, the developers should use a "deny by default" policy that checks for operations to allow. Specifically, the developers should change `operator != Enum.Operation.DelegateCall` to `operator == Enum.Operation.NormalCall`. This can help prevent the developers from accidentally introducing access control vulnerabilities as the wallet implementation is updated.

4.1.20 V-VERS-VUL-020: replace() allows special values to be inserted into the list

Severity	Warning	**Commit**	de6c674
Type	Data Validation	**Status**	Fixed
File(s)	`AddressLinkedList.sol`		
Location(s)	replace()		

The `AddressLinkedList` library implements a set data structure of nonzero addresses, as a circular linked list on top of a mapping. It has the following properties:

- Any key that has a value of `address(0)` is not contained in the set.
- The special key `SENTINEL_ADDRESS` (equal to `address(1)`) has an address value pointing to the head of the linked list.
- The last address in the linked list has a value of `SENTINEL_ADDRESS`.

The `replace()` function will replace an existing entry with a new one (that does not already exist). However, the `replace()` function can be successfully called with the zero address or the `SENTINEL_ADDRESS`. This can violate some of the above properties of the `AddressLinkedList`.

```
function replace(mapping(address => address) storage self, address oldAddr, address
    newAddr) internal {
    require(isExist(self, oldAddr), "address not exists");
    require(!isExist(self, newAddr), "new address already exists");

    address cursor = SENTINEL_ADDRESS;
    while (true) {
        address _addr = self[cursor];
        if (_addr == oldAddr) {
            address next = self[_addr];
            self[newAddr] = next;
            self[cursor] = newAddr;
            self[_addr] = address(0);
            return;
        }
        cursor = _addr;
    }
}
```

Snippet 4.28: Definition of `replace()`

```
function isExist(mapping(address => address) storage self, address addr) internal
    view returns (bool) {
    return self[addr] != address(0) && uint160(addr) > SENTINEL_UINT;
}
```

Snippet 4.29: Definition of `isExist()`

As a concrete example, the following sequence of function calls can result in an address being inserted twice:

1. `add(list, addr1)`
2. `add(list, addr2)`
3. `replace(list, addr1, address(0))`

4. `add(list, addr2)`

This occurs because the `replace()` function does not check that the `newAddr` parameter is nonzero, so that the `lastAddr` will actually be replaced with the zero address. Furthermore, it is also possible to replace an existing entry with the `SENTINEL_ADDRESS`.

Impact Currently, this has no impact since the `replace()` function is not used. However, if `replace()` is used and this issue is triggered, then several invariants will be violated, including:

- Replacing an existing address with the zero address will "remove" all items after that address, as they cannot be reached by iterating over the `SENTINEL_ADDRESS` key. However, the "removed" addresses will still exist in the sense that `isExist()` will evaluate to true on them. Furthermore, the new "last" address will not have the value `SENTINEL_ADDRESS`.
- Replacing an existing address with the `SENTINEL_ADDRESS` also leads to `size()` of the list being calculated incorrectly.

An invariant violation can result in bugs. For example, it is possible to cause an infinite loop in `replace()`, causing transactions to use up all gas and revert. One such sequence of calls that demonstrate the issue is:

1. `add(list, addr1)`
2. `add(list, addr2)`
3. `replace(list, addr1, address(0))`
4. `add(list, addr2)`
5. `add(list, addr3)`
6. `add(list, addr1)`
7. `add(list, addr4)`
8. `replace(list, addr4, addr5)`

Recommendation Add the `onlyAddress(newAddr)` modifier to prevent the zero address and the `SENTINEL_ADDRESS` from being inserted.

4.1.21 V-VERS-VUL-021: Signature length check does not capture actual property

Severity	Warning	**Commit**	56b82dd
Type	Data Validation	**Status**	Fixed
File(s)	`ECDSAValidator.sol`		
Location(s)	validateSignature()		

The first few lines of the `validateSignature()` method check that the signature attached to the user operation matches the length needed for the signature scheme described in `SignatureHandler.sol`. For the `ECDSAValidator`, the scheme requires that:

1. The signature field of an instant transaction is exactly 86 bytes long.
2. The signature field of a scheduled transaction is exactly 162 bytes long.

However, the actual check that is performed is that (1) the signature type is either instant or scheduled; and (2) the length is equal to either 86 bytes or 162 bytes. The actual behavior is not logically equivalent to the intended behavior.

Impact This could result in bugs if the signature is an instant transaction with a signature of length 162 bytes or a scheduled transaction with a signature of length 86 bytes. In the current code, this has no impact due to the following reasons:

1. An instant transaction with signature length 162 will cause a revert, as the OpenZeppelin ECDSA library requires an ECDSA signature that is exactly 65 bytes in length, but the actual signature passed in will be 141 bytes in length.
2. A scheduled transaction with signature length 86 will cause a revert, as `splitUserOpSignature()` will go out-of-bounds when indexing into the user operation signature field.

Recommendation Despite having no impact, we recommend that the behavior is corrected in case the developers intend to change some of the signature parsing logic in the future. Specifically, the developers should change the length check so that it checks the property "(signature type is instant transaction implies length is 86) and (signature type is scheduled transaction implies length is 162)".

4.1.22 V-VERS-VUL-022: Error-prone assumption that msg.sender and wallet are the same

Severity	Warning	Commit	de6c674
Type	Maintainability	Status	Fixed
File(s)	MultiSigValidator.sol		
Location(s)	_revokeGuardian(), _addGuardian()		

The `MultiSigValidator` keeps track of its signers ("guardians") in the `_guardians` mapping, where an address `g` is a guardian of a wallet `w` if and only if `_guardians[g][w]` is set to true. When the `_addGuardian()` and `_revokeGuardian()` functions are used to update `_guardians`, the actual wallet key that is used is `msg.sender` rather than the `wallet` parameter. Thus, the code assumes that `msg.sender` equals `wallet`, though this assumption is neither enforced nor stated anywhere.

```
function _addGuardian(address wallet, address guardian) internal {
    require(!_isGuardian(wallet, guardian), "Guardian is already added");
    require(guardian != wallet && guardian != address(0), "Invalid guardian address")
    ;
    WalletInfo storage info = _walletInfo[wallet];
    info.guardianCount++;
    _guardians[guardian][msg.sender] = true;
    emit AddGuardian(wallet, guardian);
}

function _revokeGuardian(address wallet, address guardian) internal {
    require(_isGuardian(wallet, guardian), "Not a valid guardian");
    WalletInfo storage info = _walletInfo[wallet];
    _guardians[guardian][msg.sender] = false;
    info.guardianCount--;
    emit RevokeGuardian(wallet, guardian);
}
```

Snippet 4.30: Definitions of `_addGuardian()` and `_revokeGuardian()`

Impact This currently has no impact as the `msg.sender` will be the same as `wallet` on all paths to `_addGuardian()` and `_revokeGuardian()`. However, a future refactor may break this assumption, which can result in bugs where the wrong address is added to or removed from the guardian set.

Recommendation Change `msg.sender` to `wallet` to match the assumption in the code.

4.1.23 V-VERS-VUL-023: Out of date doc comment in executeTransactionFromModule()

Severity	Info	**Commit**	de6c674
Type	Maintainability	**Status**	Fixed
File(s)	`ModuleManager.sol`		
Location(s)	executeTransactionFromModule()		

The `execTransactionFromModule()` method has a documentation comment stating:

```
1 * @notice Subclasses must override '_isPluginEnabled' to ensure the plugin is enabled
      .
```

However, there is no function named `_isPluginEnabled` in the code base.

Recommendation To avoid confusion, update the documentation comment to be consistent with the code.

4.1.24 V-VERS-VUL-024: createAccount() reverts if wallet already exists

Severity	Info	**Commit**	de6c674
Type	Usability Issue	**Status**	Fixed
File(s)	`VersaAccountFactory.sol`		
Location(s)	createAccount()		

`createAccount()` reverts if there is an attempt to create an account at an address where some account already exists. This is inconsistent with EIP-4337, which states:

> If the factory does use `CREATE2` or some other deterministic method to create the wallet, it's expected to return the wallet address even if the wallet has already been created. This is to make it easier for clients to query the address without knowing if the wallet has already been deployed, by [. . .]

```
function createAccount(/* ... */) public returns(address){
    // ...
    require(addr.code.length == 0, "Versa factory: account already created");
    // ...
}
```

Snippet 4.31: Relevant lines in `createAccount()`

Recommendation Return the address if it already exists, as suggested.