



Smart Contract Security Audit Report



Table Of Contents

1 Executive Summary	_____
2 Audit Methodology	_____
3 Project Overview	_____
3.1 Project Introduction	_____
3.2 Vulnerability Information	_____
4 Code Overview	_____
4.1 Contracts Description	_____
4.2 Visibility Description	_____
4.3 Vulnerability Summary	_____
5 Audit Result	_____
6 Statement	_____

1 Executive Summary

On 2024.05.16, the SlowMist security team received the SoulWallet team's security audit application for SoulWallet, developed the audit plan according to the agreement of both parties and the characteristics of the project, and finally issued the security audit report.

The SlowMist security team adopts the strategy of "white box lead, black, grey box assists" to conduct a complete security test on the project in the way closest to the real attack.

The test method information:

Test method	Description
Black box testing	Conduct security tests from an attacker's perspective externally.
Grey box testing	Conduct security testing on code modules through the scripting tool, observing the internal running status, mining weaknesses.
White box testing	Based on the open source code, non-open source code, to detect whether there are vulnerabilities in programs such as nodes, SDK, etc.

The vulnerability severity level information:

Level	Description
Critical	Critical severity vulnerabilities will have a significant impact on the security of the DeFi project, and it is strongly recommended to fix the critical vulnerabilities.
High	High severity vulnerabilities will affect the normal operation of the DeFi project. It is strongly recommended to fix high-risk vulnerabilities.
Medium	Medium severity vulnerability will affect the operation of the DeFi project. It is recommended to fix medium-risk vulnerabilities.
Low	Low severity vulnerabilities may affect the operation of the DeFi project in certain scenarios. It is suggested that the project team should evaluate and consider whether these vulnerabilities need to be fixed.
Weakness	There are safety risks theoretically, but it is extremely difficult to reproduce in engineering.
Suggestion	There are better practices for coding or architecture.

2 Audit Methodology

The security audit process of SlowMist security team for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using automated analysis tools.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that was considered during the audit of the smart contract:

Serial Number	Audit Class	Audit Subclass
1	Overflow Audit	-
2	Reentrancy Attack Audit	-
3	Replay Attack Audit	-
4	Flashloan Attack Audit	-
5	Race Conditions Audit	Reordering Attack Audit
6	Permission Vulnerability Audit	Access Control Audit
		Excessive Authority Audit
7	Security Design Audit	External Module Safe Use Audit
		Compiler Version Security Audit
		Hard-coded Address Security Audit
		Fallback Function Safe Use Audit
		Show Coding Security Audit
		Function Return Value Security Audit
		External Call Function Security Audit

Serial Number	Audit Class	Audit Subclass
7	Security Design Audit	Block data Dependence Security Audit
		tx.origin Authentication Security Audit
8	Denial of Service Audit	-
9	Gas Optimization Audit	-
10	Design Logic Audit	-
11	Variable Coverage Vulnerability Audit	-
12	"False Top-up" Vulnerability Audit	-
13	Scoping and Declarations Audit	-
14	Malicious Event Log Audit	-
15	Arithmetic Accuracy Deviation Audit	-
16	Uninitialized Storage Pointer Audit	-

3 Project Overview

3.1 Project Introduction

Soul Wallet is an Ethereum cross-L2 social recovery wallet. It focuses on providing a secure and user-friendly experience with features like social recovery, passkey signing, and compatibility with the most popular L2s.

This audit covers the underlying Core protocol and the upper-layer wallet contracts of the Soul wallet. The Core protocol mainly implements interfaces that comply with the EIP4337 standard and provides management interfaces for extension modules. The upper-layer wallet contracts inherit the Core protocol and implement various extension modules, including social recovery, 2FA, upgradeability, signature verification, owner management, hook functionality, etc.

3.2 Vulnerability Information

The following is the status of the vulnerabilities found in this audit:

NO	Title	Category	Level	Status
N1	The gas usage of uninstalling the external is fixed at 1M	Others	Information	Acknowledged
N2	Allow reinstallation of old extensions that have been uninstalled	Design Logic Audit	Information	Acknowledged
N3	Optimizable hook signature verification logic	Design Logic Audit	Information	Acknowledged
N4	Potential DoS risks of hook verification	Denial of Service Vulnerability	Information	Acknowledged
N5	Unnecessary extra loops	Design Logic Audit	Information	Acknowledged
N6	Changing 2FA is not checking if there is 2FA that is not applied	Design Logic Audit	Suggestion	Acknowledged
N7	The applyChange2FA operation did not check whether pending2FAAddr existed	Design Logic Audit	Low	Fixed
N8	Crypto2FAHook only works with current AA wallets	Others	Information	Acknowledged
N9	Potential risk of account takeover	Design Logic Audit	Medium	Fixed
N10	Redundant unchecked usage	Others	Suggestion	Fixed
N11	Risks of arbitrarily upgrading account	Authority Control Vulnerability Audit	Information	Acknowledged
N12	The return value of the validateUserOp function does not conform to the specification	Others	Suggestion	Acknowledged

NO	Title	Category	Level	Status
N13	Missing checks on L2 oracle availability	Design Logic Audit	Low	Acknowledged
N14	Potential risk of Paymaster funds being drained	Design Logic Audit	Critical	Fixed
N15	Incorrect fee amount charged	Design Logic Audit	Medium	Fixed
N16	Potential risks of not using real-time prices	Design Logic Audit	Medium	Fixed
N17	Potential risks of one-time maximum approval	Design Logic Audit	Suggestion	Acknowledged
N18	Oracle updatedAt checks can be optimized	Design Logic Audit	Suggestion	Acknowledged

4 Code Overview

4.1 Contracts Description

Audit Version:

https://github.com/SoulWallet/soul-wallet-contract/tree/preliminary_audit

commit: fd9d0ce5572826ebf9e6842b5316977e17316ac2

The audit does not include the libraries and dev directory

<https://github.com/SoulWallet/soulwallet-core>

commit: 2941b40f06387054321806905ce27845286035d4

The main network address of the contract is as follows:

The code was not deployed to the mainnet.

4.2 Visibility Description

The SlowMist Security team analyzed the visibility of major contracts during the audit, the result as follows:

SoulWalletCore			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	EntryPointManager
isValidSignature	Public	-	-
_decodeSignature	Internal	-	-
validateUserOp	Public	Payable	-

EOAValidator			
Function Name	Visibility	Mutability	Modifiers
supportsInterface	External	-	-
Init	External	-	-
Delnit	External	-	-
_packHash	Internal	-	-
_isOwner	Private	-	-
validateSignature	External	-	-
validateUserOp	External	-	-

AccountExecute			
Function Name	Visibility	Mutability	Modifiers
executeUserOp	External	Payable	-
_executeUserOp	Internal	Can Modify State	-

Authority			
Function Name	Visibility	Mutability	Modifiers
_onlyModule	Internal	-	-

Authority			
_onlySelfOrModule	Internal	-	-
fallbackManagementAccess	Internal	-	-
pluginManagementAccess	Internal	-	-
ownerManagementAccess	Internal	-	-
executorAccess	Internal	-	-
validatorManagementAccess	Internal	-	-

EntryPointManager			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
entryPoint	External	-	-
_onlyEntryPoint	Internal	-	-

FallbackManager			
Function Name	Visibility	Mutability	Modifiers
<Receive Ether>	External	Payable	-
_setFallbackHandler	Internal	Can Modify State	-
<Fallback>	External	Payable	-
setFallbackHandler	External	Can Modify State	-

HookManager			
Function Name	Visibility	Mutability	Modifiers
isInstalledHook	External	-	-
_isInstalledHook	Internal	-	-

HookManager			
_isSupportsHookInterface	Internal	-	-
_installHook	Internal	Can Modify State	-
_uninstallHook	Internal	Can Modify State	-
uninstallHook	External	Can Modify State	-
listHook	External	-	-
_nextHookSignature	Private	-	-
_prelsValidSignatureHook	Internal	-	-
_preUserOpValidationHook	Internal	Can Modify State	-

ModuleManager			
Function Name	Visibility	Mutability	Modifiers
_moduleMapping	Internal	-	-
_isAuthorizedModule	Internal	-	-
_isInstalledModule	Internal	-	-
isInstalledModule	External	-	-
_isSupportsModuleInterface	Internal	-	-
_installModule	Internal	Can Modify State	-
_uninstallModule	Internal	Can Modify State	-
uninstallModule	External	Can Modify State	-
listModule	External	-	-
executeFromModule	External	Can Modify State	-

OwnerManager			
Function Name	Visibility	Mutability	Modifiers
_ownerMapping	Internal	-	-
_isOwner	Internal	-	-
isOwner	External	-	-
_addOwner	Internal	Can Modify State	-
addOwner	External	Can Modify State	-
_removeOwner	Internal	Can Modify State	-
removeOwner	External	Can Modify State	-
_resetOwner	Internal	Can Modify State	-
_clearOwner	Internal	Can Modify State	-
resetOwner	External	Can Modify State	-
listOwner	External	-	-

StandardExecutor			
Function Name	Visibility	Mutability	Modifiers
execute	External	Payable	-
executeBatch	External	Payable	-

ValidatorManager			
Function Name	Visibility	Mutability	Modifiers
_isInstalledValidator	Internal	-	-
_isSupportsValidatorInterface	Internal	-	-
_installValidator	Internal	Can Modify State	-

ValidatorManager			
_uninstallValidator	Internal	Can Modify State	-
uninstallValidator	External	Can Modify State	-
listValidator	External	-	-
_isValidSignature	Internal	-	-
_validateUserOp	Internal	Can Modify State	-

HookInstaller			
Function Name	Visibility	Mutability	Modifiers
installHook	External	Can Modify State	-

ModuleInstaller			
Function Name	Visibility	Mutability	Modifiers
installModule	External	Can Modify State	-

ValidatorInstaller			
Function Name	Visibility	Mutability	Modifiers
installValidator	External	Can Modify State	-

DefaultCallbackHandler			
Function Name	Visibility	Mutability	Modifiers
onERC721Received	External	-	-
onERC1155Received	External	-	-
onERC1155BatchReceived	External	-	-
supportsInterface	External	-	-

DefaultCallbackHandler			
<Receive Ether>	External	Payable	-
<Fallback>	External	Payable	-

ERC1271Handler			
Function Name	Visibility	Mutability	Modifiers
_encodeRawHash	Internal	-	-
getChainId	Public	-	-

SoulWalletHookManager			
Function Name	Visibility	Mutability	Modifiers
_installHook	Internal	Can Modify State	-
installHook	External	Can Modify State	-

SoulWalletModuleManager			
Function Name	Visibility	Mutability	Modifiers
installModule	External	Can Modify State	-
_isSupportsModuleInterface	Internal	-	-
_addModule	Internal	Can Modify State	-

SoulWalletOwnerManager			
Function Name	Visibility	Mutability	Modifiers
_addOwners	Internal	Can Modify State	-
addOwners	External	Can Modify State	-
resetOwners	External	Can Modify State	-

SoulWalletUpgradeManager			
Function Name	Visibility	Mutability	Modifiers
_upgradeTo	Internal	Can Modify State	-

SoulWalletValidatorManager			
Function Name	Visibility	Mutability	Modifiers
installValidator	External	Can Modify State	-

AaveUsdcSaveAutomation			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	Ownable
depositUsdcToAave	Public	Can Modify State	onlyBot
depositUsdcToAaveBatch	Public	Can Modify State	onlyBot
addBot	Public	Can Modify State	onlyOwner
removeBot	Public	Can Modify State	onlyOwner

ClaimInterest			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	Ownable
claimInterest	Public	Can Modify State	-
getChainId	Public	-	-
changeSigner	Public	Can Modify State	onlyOwner
withdraw	Public	Can Modify State	onlyOwner
deposit	Public	Can Modify State	-
incrementNonce	Public	Can Modify State	-

SoulWalletFactory			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	Ownable
_calcSalt	Private	-	-
createWallet	External	Can Modify State	-
proxyCode	External	-	-
_proxyCode	Private	-	-
getWalletAddress	Public	-	-
deposit	Public	Payable	-
withdrawTo	Public	Can Modify State	onlyOwner
addStake	External	Payable	onlyOwner
unlockStake	External	Can Modify State	onlyOwner
withdrawStake	External	Can Modify State	onlyOwner

Crypto2FAHook			
Function Name	Visibility	Mutability	Modifiers
supportsInterface	External	-	-
Init	External	Can Modify State	-
DelInit	External	Can Modify State	-
prelsValidSignatureHook	External	-	-
preUserOpValidationHook	External	-	-
initiateChange2FA	External	Can Modify State	-
applyChange2FA	External	Can Modify State	-
cancelChange2FA	External	Can Modify State	-

UpgradeModule			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
inited	Internal	-	-
_init	Internal	Can Modify State	-
_delnit	Internal	Can Modify State	-
upgrade	External	Can Modify State	-
requiredFunctions	External	-	-

BaseModule			
Function Name	Visibility	Mutability	Modifiers
inited	Internal	-	-
_init	Internal	Can Modify State	-
_delnit	Internal	Can Modify State	-
sender	Internal	-	-
Init	External	Can Modify State	-
Delnit	External	Can Modify State	-
supportsInterface	External	-	-

ERC20Paymaster			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	BasePaymaster
setNativeAssetOracle	External	Can Modify State	onlyOwner
setToken	External	Can Modify State	onlyOwner

ERC20Paymaster			
withdrawToken	External	Can Modify State	onlyOwner
updatePrice	External	Can Modify State	-
isSupportToken	Public	-	-
_validatePaymasterUserOp	Internal	Can Modify State	-
_validateConstructor	Internal	-	-
_decodeApprove	Private	-	-
_postOp	Internal	Can Modify State	-
fetchPrice	Internal	-	-

SoulWalletDefaultValidator			
Function Name	Visibility	Mutability	Modifiers
validateUserOp	External	-	-
validateSignature	External	-	-
_packSignatureHash	Internal	-	-
_pack1271SignatureHash	Internal	-	-
_isOwner	Private	-	-
recover	Internal	-	-
supportsInterface	Public	-	-
Init	External	Can Modify State	-
Delnit	External	Can Modify State	-

SoulWallet			
Function Name	Visibility	Mutability	Modifiers

SoulWallet			
<Constructor>	Public	Can Modify State	EntryPointManager
initialize	External	Can Modify State	initializer
_uninstallValidator	Internal	Can Modify State	-
isValidSignature	Public	-	-
_decodeSignature	Internal	-	-
validateUserOp	Public	Payable	-
upgradeTo	External	Can Modify State	-
upgradeFrom	External	-	-

4.3 Vulnerability Summary

[N1] [Information] The gas usage of uninstalling the external is fixed at 1M

Category: Others

Content

In the protocol, the wallet supports some external trusted extensions to enrich the functionality of AA. In Core, these extensions are managed through the installation/uninstallation functions of the HookManager, ModuleManager, and ValidatorManager contracts. When installing these extensions, the Manager calls the Init function of the extension for initialization. When uninstalling these extensions, the Manager calls the Delnit function of the extension to clear data. When calling the Delnit function, the gas of the transaction is limited to 1M, which means that if the Delnit function is too complex, it will cause the transaction to fail, resulting in the inability to uninstall the extension. Therefore, when installing extensions, it is necessary to specifically check whether the gas consumption of the Delnit function exceeds expectations.

Code location:

soulwallet-core/contracts/base/HookManager.sol#L130

soulwallet-core/contracts/base/ModuleManager.sol#L133

soulwallet-core/contracts/base/ValidatorManager.sol#L83

```
function _uninstallHook(address hookAddress) internal virtual override {
    ...
    (bool success,) =
        hookAddress.call{gas: 1000000 /* max to 1M gas */ }
        (abi.encodeWithSelector(IPluggable.DeInit.selector));
    ...
}

function _uninstallModule(address moduleAddress) internal virtual override {
    ...
    (bool success,) =
        moduleAddress.call{gas: 1000000 /* max to 1M gas */ }
        (abi.encodeWithSelector(IPluggable.DeInit.selector));
    ...
}

function _uninstallValidator(address validator) internal virtual override {
    ...
    (bool success,) =
        validator.call{gas: 1000000 /* max to 1M gas */ }
        (abi.encodeWithSelector(IPluggable.DeInit.selector));
    ...
}
```

Solution

N/A

Status

Acknowledged

[N2] [Information] Allow reinstallation of old extensions that have been uninstalled

Category: Design Logic Audit

Content

In the protocol, the wallet supports some external trusted extensions to enrich the functionality of AA. In Core, these extensions are managed through the installation/uninstallation functions of the FallbackManager, HookManager, ModuleManager, OwnerManager, and ValidatorManager contracts. When installing these extensions, the Manager calls the Init function of the extension for initialization. When uninstalling these

extensions, the Manager calls the Delnit function of the extension to clear data. However, the protocol allows users to reinstall extensions that have been uninstalled. If the extension does not completely clear its residual data during uninstallation, it will cause the protocol to misuse previous data, which may lead to unknown security risks.

Code location:

soulwallet-core/contracts/base/HookManager.sol#L78-L137

soulwallet-core/contracts/base/ModuleManager.sol#L87-L139

soulwallet-core/contracts/base/ValidatorManager.sol#L49-L89

```
function _installHook(address hookAddress, bytes memory initData, uint8
capabilityFlags)
    internal
    virtual
    override
{
    ...
}

function _uninstallHook(address hookAddress) internal virtual override {
    ...
}

function _installModule(address moduleAddress, bytes memory initData, bytes4[]
memory selectors)
    internal
    virtual
    override
{
    ...
}

function _uninstallModule(address moduleAddress) internal virtual override {
    ...
}

function _installValidator(address validator, bytes memory initData) internal
virtual override {
    ...
}

function _uninstallValidator(address validator) internal virtual override {
```

```
...
}
```

Solution

It is recommended that when installing extensions, they should be audited and properly handle old data in their Delnit function.

Status

Acknowledged

[N3] [Information] Optimizable hook signature verification logic

Category: Design Logic Audit

Content

In the HookManager contract, `_preIsValidSignatureHook` and `_preUserOpValidationHook` are used to verify the `hookSignatures` passed in by the user. They compare the hook addresses in the `preIsValidSignatureHook/preUserOpValidationHook` list with the `_hookAddr` decoded from `_nextHookSignature` to determine the `currentHookSignature` for the external call. If the decoded `_hookAddr` is not in the `preIsValidSignatureHook/preUserOpValidationHook` list, then `currentHookSignature` will be 0. However, it's important to note that even if `currentHookSignature` is 0 and `_hookAddr` is not in the list, the protocol will still initiate an external call to the current hookAddress. This means that if the user-passed `_hookAddr` is the 0 address, a hook call will still be initiated. If the user-defined hook contract doesn't verify the signature, it could lead to the risk of unexpected hooks. Moreover, each hook call may not invoke all the hook addresses in the list, which could lead to the hook call reverting even if the user passes in the correct hook address and signature because the loop hasn't correctly matched yet.

Code location: soulwallet-core/contracts/base/HookManager.sol#L239,L307

```
function _preIsValidSignatureHook(bytes32 hash, bytes calldata hookSignatures)
internal view virtual {
    ...
    bytes memory callData =
        abi.encodeWithSelector(IHook.preIsValidSignatureHook.selector, hash,
currentHookSignature);
    assembly ("memory-safe") {
        let result := staticcall(gas(), hookAddress, add(callData, 0x20),
```

```

mload(callData), 0x00, 0x00)
    if iszero(result) {
        mstore(0x00, 0x00000000)
        return(0x00, 0x20)
    }
}

hookAddress = preIsValidSignatureHook[hookAddress];
}

if (_hookAddr != address(0)) {
    revert INVALID_HOOK_SIGNATURE();
}
}

function _preUserOpValidationHook(
    ...
) internal virtual {
    ...
    bytes memory callData = abi.encodeWithSelector(
        IHook.preUserOpValidationHook.selector, userOp, userOpHash,
missingAccountFunds, currentHookSignature
    );
    assembly ("memory-safe") {
        let result := call(gas(), hookAddress, 0, add(callData, 0x20),
mload(callData), 0x00, 0x00)
        if iszero(result) {
            mstore(0x00, SIG_VALIDATION_FAILED)
            return(0x00, 0x20)
        }
    }

    hookAddress = preUserOpValidationHook[hookAddress];
}
if (_hookAddr != address(0)) {
    revert INVALID_HOOK_SIGNATURE();
}
}
}

```

Solution

It is recommended to only proceed with the hook call when the `_hookAddr` matches a hook address in the `preIsValidSignatureHook/preUserOpValidationHook` list.

Status

Acknowledged; After communicating with the project team, they stated that this is the expected design. When

users utilize hook extensions, they must use all of the installed hook extensions and cannot selectively use specific hook extensions.

[N4] [Information] Potential DoS risks of hook verification

Category: Denial of Service Vulnerability

Content

In the HookManager contract, the `_preIsValidSignatureHook/_preUserOpValidationHook` functions are used to verify the hookSignatures passed in by the user. They use a while loop to compare the hook addresses in the `preIsValidSignatureHook/preUserOpValidationHook` list with the `_hookAddr` decoded from `_nextHookSignature` to determine the currentHookSignature for the external call. It's important to note that the loop starts from `preIsValidSignatureHook[AddressLinkedList.SENTINEL_ADDRESS]` and eventually returns to `SENTINEL_ADDRESS`. This means that if the `hookSignatures` parameter passed in by the user involves multiple hook calls, then the encoded `_hookAddr` must follow the loop order; otherwise, some valid hook verifications may not be executed. In fact, when users perform hook signature encoding off-chain, they may not be able to ensure that the order is consistent with the list, which could lead to denial of service risks.

Code location:

soulwallet-core/contracts/base/HookManager.sol#L221

soulwallet-core/contracts/base/HookManager.sol#L288

```
function _preIsValidSignatureHook(bytes32 hash, bytes calldata hookSignatures)
internal view virtual {
    address _hookAddr;
    uint256 _cursorFrom;
    uint256 _cursorEnd;
    (_hookAddr, _cursorFrom, _cursorEnd) = _nextHookSignature(hookSignatures,
_cursorEnd);

    mapping(address => address) storage preIsValidSignatureHook =
AccountStorage.layout().preIsValidSignatureHook;
    address hookAddress =
preIsValidSignatureHook[AddressLinkedList.SENTINEL_ADDRESS];
    while (uint160(hookAddress) > AddressLinkedList.SENTINEL_UINT) {
        bytes calldata currentHookSignature;
        if (hookAddress == _hookAddr) {
            currentHookSignature = hookSignatures[_cursorFrom:_cursorEnd];
            // next
```

```

        _hookAddr = address(0);
        if (_cursorEnd > 0) {
            (_hookAddr, _cursorFrom, _cursorEnd) =
_nextHookSignature(hookSignatures, _cursorEnd);
        }
    } else {
        currentHookSignature = hookSignatures[0:0];
    }
    ...
}

function _preUserOpValidationHook(
    ...
) internal virtual {
    address _hookAddr;
    uint256 _cursorFrom;
    uint256 _cursorEnd;
    (_hookAddr, _cursorFrom, _cursorEnd) = _nextHookSignature(hookSignatures,
_cursorEnd);

    mapping(address => address) storage preUserOpValidationHook =
AccountStorage.layout().preUserOpValidationHook;
    address hookAddress =
preUserOpValidationHook[AddressLinkedList.SENTINEL_ADDRESS];
    while (uint160(hookAddress) > AddressLinkedList.SENTINEL_UINT) {
        bytes calldata currentHookSignature;
        if (hookAddress == _hookAddr) {
            currentHookSignature = hookSignatures[_cursorFrom:_cursorEnd];
            // next
            _hookAddr = address(0);
            if (_cursorEnd > 0) {
                (_hookAddr, _cursorFrom, _cursorEnd) =
_nextHookSignature(hookSignatures, _cursorEnd);
            }
        } else {
            currentHookSignature = hookSignatures[0:0];
        }
        ...
    }
}

```

Solution

It is recommended to first extract all `_hookAddr` through `_nextHookSignature`, and then filter the valid `_hookAddr` for execution.

Status

Acknowledged; After communicating with the project team, they stated that users must pass in the hook addresses in the order of the list. The potential DoS risk caused by not passing them in order is expected.

[N5] [Information] Unnecessary extra loops

Category: Design Logic Audit

Content

In the HookManager contract, the `_preIsValidSignatureHook/_preUserOpValidationHook` functions are used to verify the hookSignatures passed in by the user. They use a while loop to compare the hook addresses in the `preIsValidSignatureHook/preUserOpValidationHook` list with the `_hookAddr` decoded from `_nextHookSignature` to determine the currentHookSignature for the external call. The while loop only stops when the hookAddress equals SENTINEL_ADDRESS, which means that even if there is only one hook that needs to be called, the while loop will still traverse the entire list, which is unnecessary.

Code location:

soulwallet-core/contracts/base/HookManager.sol#L221

soulwallet-core/contracts/base/HookManager.sol#L288

```
function _preIsValidSignatureHook(bytes32 hash, bytes calldata hookSignatures)
internal view virtual {
    address _hookAddr;
    uint256 _cursorFrom;
    uint256 _cursorEnd;
    (_hookAddr, _cursorFrom, _cursorEnd) = _nextHookSignature(hookSignatures,
_cursorEnd);

    mapping(address => address) storage preIsValidSignatureHook =
AccountStorage.layout().preIsValidSignatureHook;
    address hookAddress =
preIsValidSignatureHook[AddressLinkedList.SENTINEL_ADDRESS];
    while (uint160(hookAddress) > AddressLinkedList.SENTINEL_UINT) {
        bytes calldata currentHookSignature;
        if (hookAddress == _hookAddr) {
            currentHookSignature = hookSignatures[_cursorFrom:_cursorEnd];
            // next
            _hookAddr = address(0);
            if (_cursorEnd > 0) {
                (_hookAddr, _cursorFrom, _cursorEnd) =
_nextHookSignature(hookSignatures, _cursorEnd);
            }
        }
    }
}
```

```

        } else {
            currentHookSignature = hookSignatures[0:0];
        }
        ...
    }

    function _preUserOpValidationHook(
        ...
    ) internal virtual {
        address _hookAddr;
        uint256 _cursorFrom;
        uint256 _cursorEnd;
        (_hookAddr, _cursorFrom, _cursorEnd) = _nextHookSignature(hookSignatures,
        _cursorEnd);

        mapping(address => address) storage preUserOpValidationHook =
AccountStorage.layout().preUserOpValidationHook;
        address hookAddress =
preUserOpValidationHook[AddressLinkedList.SENTINEL_ADDRESS];
        while (uint160(hookAddress) > AddressLinkedList.SENTINEL_UINT) {
            bytes calldata currentHookSignature;
            if (hookAddress == _hookAddr) {
                currentHookSignature = hookSignatures[_cursorFrom:_cursorEnd];
                // next
                _hookAddr = address(0);
                if (_cursorEnd > 0) {
                    (_hookAddr, _cursorFrom, _cursorEnd) =
_nextHookSignature(hookSignatures, _cursorEnd);
                }
            } else {
                currentHookSignature = hookSignatures[0:0];
            }
            ...
        }
    }
}

```

Solution

It is recommended to break the loop when `_hookAddr` is the 0 address.

Status

Acknowledged; After communicating with the project team, they stated that this is the expected design. When users utilize hook extensions, they must use all of the installed hook extensions and cannot selectively use specific hook extensions.

[N6] [Suggestion] Changing 2FA is not checking if there is 2FA that is not applied

Category: Design Logic Audit**Content**

In the Crypto2FAHook contract, users can set a new 2FA address through the `initiateChange2FA` function. It first sets the new 2FA address as `pending2FAAddr`, and can only formally update it through the `applyChange2FA` function after the operation lock time expires. However, during this process, it does not check whether a `pending2FAAddr` already exists. This means that users can directly overwrite `pending2FAAddr` with a new 2FA without canceling the operation, even if a `pending2FAAddr` has already been set.

Code location: `soul-wallet-contract/contracts/hooks/2fa/Crypto2FAHook.sol#L57`

```
function initiateChange2FA(address new2FA) external {
    User2FA storage _user2fa = user2FA[msg.sender];
    require(_user2fa.initialized, "User not initialized");
    _user2fa.pending2FAAddr = new2FA;
    _user2fa.effectiveTime = block.timestamp + TIME_LOCK_DURATION;
}
```

Solution

If this is not the intended design, it is recommended to check that `effectiveTime` must be 0 when performing the `initiateChange2FA` operation.

Status

Acknowledged; The project owner said that user can overwrite pending 2fa when call `initiateChange2fa` again.

[N7] [Low] The `applyChange2FA` operation did not check whether `pending2FAAddr` existed**Category: Design Logic Audit****Content**

In the Crypto2FAHook contract, users can formally update the preset 2FA address to `wallet2FAAddr` through the `applyChange2FA` function. However, during this process, there is no check on whether `pending2FAAddr` exists or whether `effectiveTime` is greater than 0. This will allow users to directly update the 2FA address incorrectly to the 0 address through the `applyChange2FA` function without performing the `initiateChange2FA` operation.

Code location: `soul-wallet-contract/contracts/hooks/2fa/Crypto2FAHook.sol#L64`

```
function applyChange2FA() external {
    User2FA storage _user2fa = user2FA[msg.sender];
    require(block.timestamp >= _user2fa.effectiveTime, "Time lock not expired");
    _user2fa.wallet2FAAddr = _user2fa.pending2FAAddr;
    _user2fa.pending2FAAddr = address(0);
    _user2fa.effectiveTime = 0;
}
```

Solution

It is recommended to check that pending2FAAddr is not the 0 address or that effectiveTime must be greater than 0 when performing the applyChange2FA operation.

Status

Fixed; Fixed in commit b70cdec5421a58381441798819e535932036863f

[N8] [Information] Crypto2FAHook only works with current AA wallets

Category: Others

Content

In the Crypto2FAHook contract, the preUserOpValidationHook function is called by SoulWallet to verify if the hook signature is valid. The signature data depends on the userOpHash of the user's UserOperation, which is constructed by EntryPoint. It guarantees that the userOpHash passed in each time preUserOpValidationHook is called will be different. This ensures that even though preUserOpValidationHook does not implement replay protection for signatures, it can still avoid the risk of signature replay. Other external protocols interfacing with this contract need to use it with caution.

The same is true for the preIsValidSignatureHook function

Code location: soul-wallet-contract/contracts/hooks/2fa/Crypto2FAHook.sol#L48

```
function preUserOpValidationHook(
    PackedUserOperation calldata userOp,
    bytes32 userOpHash,
    uint256 missingAccountFunds,
    bytes calldata hookSignature
) external view override {
    (userOp, userOpHash, missingAccountFunds, hookSignature);
    address recoveredAddress =
        userOpHash.toEthSignedMessageHash().recover(hookSignature);
```

```
require(recoveredAddress == user2FA[msg.sender].wallet2FAAddr, "Crypto2FAHook:  
invalid signature");  
}
```

Solution

N/A

Status

Acknowledged

[N9] [Medium] Potential risk of account takeover

Category: Design Logic Audit

Content

A social recovery module is built into the protocol to allow users to reset the owner of their wallet after losing their private key. Users can install this module through SoulWalletModuleManager. During module installation, the user's data is initialized through the Init function. In the social recovery module, it requires users to set the guardianHash and delayPeriod at the time of installation. However, it's important to note that when users load this module, they may not have completed confirming the guardian addresses, so they may pass in GuardianData with all 0 values.

Unfortunately, during social recovery, when the v value of the passed-in guardianSignature is 2, it will skip the signature check for a specified number of signatures. Although the skipped count is subtracted before the final threshold check, when the user's initial GuardianData are all 0 values, `guardianData.threshold` will necessarily be 0. Since GuardianData are all 0 values, it's easy to construct malicious data to pass `scheduleRecovery`, and once the time delay expires, the user's account can be taken over through `executeRecovery`.

Code location: soul-wallet-contract/contracts/modules/socialRecovery/base/BaseSocialRecovery.sol#L106

```
function setGuardian(bytes32 newGuardianHash) external {  
    address wallet = _msgSender();  
    socialRecoveryInfo[wallet].guardianHash = newGuardianHash;  
    _increaseNonce(wallet);  
    emit GuardianSet(wallet, newGuardianHash);  
}
```

Solution

It is recommended to strictly limit the user's passed-in guardianHash to not be a hash of 0 values during initialization, or to remove the signature check logic when the v value is 2.

Status

Fixed; Fixed in commit 4213c373ce85efeea4967130502aa736c092e1c4

[N10] [Suggestion] Redundant unchecked usage

Category: Others

Content

In the BaseSocialRecovery contract, the `_verifyGuardianSignature` function is used to check if the guardian's signature is valid. It checks each guardian one by one through a for loop to ensure that the valid signatures meet the threshold. Inside the loop, it uses unchecked for loop increments to reduce gas consumption. However, it's important to note that after Solidity 0.8.22, for loop increments are unchecked by default. Therefore, explicitly using unchecked for loop increments is unnecessary.

Reference: <https://soliditylang.org/blog/2023/10/25/solidity-0.8.22-release-announcement/>

Code location: soul-wallet-contract/contracts/modules/socialRecovery/base/BaseSocialRecovery.sol#L360

```
function _verifyGuardianSignature(
    ...
) internal view {
    ...
    for (uint256 i = 0; i < guardiansLen;) {
        ...
        unchecked {
            i++; // see Note line 223
        }
    }
    ...
}
```

Solution

When using Solidity versions above 0.8.22, it is not recommended to explicitly use unchecked for for loop increments.

Status

Fixed; Fixed in commit 043c99b5a4083f5788111f087a1498b180a42652

[N11] [Information] Risks of arbitrarily upgrading account**Category: Authority Control Vulnerability Audit****Content**

The protocol has a built-in upgradable module that allows users to optionally upgrade their accounts. The implementation contract address is fixed at the time of UpgradeModule deployment and cannot be changed. Users can directly upgrade by calling the upgrade function of the UpgradeModule contract. Unfortunately, anyone can call the upgrade function of the UpgradeModule contract and pass in a specified SoulWallet. Once a user's SoulWallet has this module installed, any user can arbitrarily call the upgrade function to upgrade the SoulWallet that has this module installed, without the user's consent.

Code location: contracts/modules/upgrade/UpgradeModule.sol#L33

```
function upgrade(address wallet) external override {
    require(_initiated[wallet] != 0, "not initiated");
    require(!_upgraded[wallet] == false, "already upgraded");
    IUpgradable(wallet).upgradeTo(newImplementation);
    _upgraded[wallet] = true;
}
```

Solution

It is recommended to only allow the upgrade function to call the upgradeTo of `msg.sender` instead of any passed-in wallet.

Status

Acknowledged; After communicating with the project team, they stated that a user installing this extension implies consent to upgrade their contract to the new implementation contract. Therefore, users should carefully consider when installing extensions.

[N12] [Suggestion] The return value of the validateUserOp function does not conform to the specification

Category: Others

Content

The soul wallet implements the `validateUserOp` interface for `EntryPoint` to call. The EIP4337 standard requires that the return value of the `validateUserOp` function must include `authorizer`, `validUntil`, and `validAfter`. The soul wallet's `validateUserOp` function calls an external trusted `EOAValidator` or `SoulWalletDefaultValidator` module for signature verification. However, the `EOAValidator` module only returns `SIG_VALIDATION_SUCCESS`, which is the value 0, upon successful validation. The `SoulWalletDefaultValidator` returns a value of 0 or the data carried in the user's `validatorSignature` upon successful validation. This does not comply with the EIP4337 specification. The same is true for the `validatePaymasterUserOp` function of the `ERC20Paymaster` contract. Despite this, it does not affect the execution of `handleOps`.

Code location:

soul-wallet-contract/contracts/SoulWallet.sol#L131

soulwallet-core/contracts/base/ValidatorManager.sol#L167

soulwallet-core/contracts/validators/EOAValidator.sol#L89

```
function validateUserOp(PackedUserOperation calldata userOp, bytes32 userOpHash,
bytes calldata validatorSignature)
    external
    view
    override
    returns (uint256 validationData)
{
    ...
    return _isOwner(recoveredAddr) ? SIG_VALIDATION_SUCCESS :
SIG_VALIDATION_FAILED;
}
```

Solution

It is recommended to implement the return value of the `validateUserOp` function according to the EIP4337 standard to avoid future compatibility issues.

Status

Acknowledged

[N13] [Low] Missing checks on L2 oracle availability

Category: Design Logic Audit

Content

In most L2 protocols, the sequencer is primarily responsible for rolling up L2 transactions. When the sequencer is down, transactions and data on L2 cannot be processed correctly. Therefore, when using the Chainlink oracle, it is necessary to check whether the current L2 sequencer status is available. Chainlink provides `sequencerUptimeFeed` to check the availability of the current L2 sequencer status.

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

Code location: `soul-wallet-contract/contracts/paymaster/ERC20Paymaster.sol#L223`

```
function fetchPrice(IOracle _oracle) internal view returns (uint192 price) {
    (uint80 roundId, int256 answer,, uint256 updatedAt, uint80 answeredInRound) =
    _oracle.latestRoundData();
    require(answer > 0, "Paymaster: Chainlink price <= 0");
    require(updatedAt >= (block.timestamp - 2 days), "Paymaster: Incomplete
round");
    require(answeredInRound >= roundId, "Paymaster: Stale price");
    price = uint192(int192(answer));
}
```

Solution

It is recommended to check whether the sequencer is available when using Chainlink to obtain prices in L2.

Status

Acknowledged

[N14] [Critical] Potential risk of Paymaster funds being drained

Category: Design Logic Audit

Content

In the `ERC20Paymaster` contract, the functionality of paying on behalf of user wallets is implemented. The `_validatePaymasterUserOp` function checks if the wallet balance is sufficient to pay the fee when a user creates a wallet and sets its `sponsorWalletCreation` state to true. Finally, after the `EntryPoint` completes the execution operation, it calls the `postOp` function of the paymaster to make the actual payment. The paymaster

checks through the `_validateConstructor` function whether there is an operation in the user's execution that approves sufficient fees to the paymaster, so that the paymaster can recover the fees from the user's wallet. It should be noted that the user can transfer away the pre-stored fees in the user's wallet during the execution of the user operation by the EntryPoint through `innerHandleOp`, causing the `postOp` operation on the paymaster by the `innerHandleOp` function of the EntryPoint to revert due to insufficient funds in the wallet. This means that the `innerHandleOp` execution fails, but the `_executeUserOp` function of the EntryPoint will call the `postOp` function of the paymaster again after the `innerHandleOp` execution fails. Unfortunately, at this time, the `PostOpMode` state passed in is `postOpReverted`. In the `_postExecution` function of the EntryPoint, it checks if the `PostOpMode` is in the `postOpReverted` state and will not proceed with the `postOp` operation. This leads to the risk that the paymaster pays the fees for operations such as wallet creation for the user but is unable to recover the fees from the user's account. Malicious users can exploit this to drain the funds in the ERC20Paymaster contract.

Code location: `soul-wallet-contract/contracts/paymaster/ERC20Paymaster.sol#L126-L133`

```
function _validatePaymasterUserOp(PackedUserOperation calldata userOp, bytes32,
uint256 requiredPreFund)
    internal
    override
    returns (bytes memory context, uint256 validationResult)
{
    ...
    if (userOp.initCode.length != 0) {
        require(requiredPreFund < MAX_ALLOW_SPONSOR_FUND_ACTIVE_WALLET,
"Paymaster: maxCost too high");
        require(ERC20Token.balanceOf(sender) >= tokenRequiredPreFund, "Paymaster:
not enough balance");
        _validateConstructor(userOp, token, tokenRequiredPreFund);
        sponsorWalletCreation = true;
    } else {
        ERC20Token.safeTransferFrom(sender, address(this), tokenRequiredPreFund);
        sponsorWalletCreation = false;
    }

    return (abi.encode(sender, token, costOfPost, cachedPrice,
tokenRequiredPreFund, sponsorWalletCreation), 0);
}
```

Solution

It is recommended to complete the fee collection in the `_validatePaymasterUserOp` function, and use `_postOp` only for refunding excess fees.

Status

Fixed; Fixed in commit 49d33af1b24e612a8496f807d4a23f5f2bb842f5

[N15] [Medium] Incorrect fee amount charged

Category: Design Logic Audit

Content

In the ERC20Paymaster contract, the `_postOp` function is used to calculate the actually consumed gas fee for execution and refund the previously overcharged fees. However, when `sponsorWalletCreation` is true, the transferred fee amount is `tokenRequiredPreFund`, not the actual `tokenRequiredFund`. Moreover, if the previously collected fee `tokenRequiredPreFund` is less than the actual fee `tokenRequiredFund`, ERC20Paymaster will not charge the user again, which may lead to insufficient fee collection.

Code location: soul-wallet-contract/contracts/paymaster/ERC20Paymaster.sol#L206-L214

```
function _postOp(PostOpMode mode, bytes calldata context, uint256 actualGasCost,
uint256 actualUserOpFeePerGas)
    internal
    override
{
    ...
    uint256 tokenRequiredFund =
        (actualGasCost + costOfPost) * supportedToken[token].priceMarkup *
cachedPrice / (1e18 * PRICE_DENOMINATOR);
    if (sponsorWalletCreation) {
        // if sponsor during wallet creation, charge the actual amount
        IERC20Metadata(token).safeTransferFrom(sender, address(this),
tokenRequiredPreFund);
    } else if (sponsorWalletCreation == false && tokenRequiredPreFund >
tokenRequiredFund) {
        // refund unused precharge token
        IERC20Metadata(token).safeTransfer(sender, tokenRequiredPreFund -
tokenRequiredFund);
    }
    ...
}
```

Solution

If this is not the intended design, it is recommended to strictly verify and collect the actual fees, refunding any excess and supplementing any shortfall.

Status

Fixed; Fixed in commit 210c26ac387e65ad2204a04a4af32312136e27b4

[N16] [Medium] Potential risks of not using real-time prices

Category: Design Logic Audit

Content

In the `_validatePaymasterUserOp` function of the ERC20Paymaster contract, due to the restrictions on global storage access imposed by EIP4337, the cached price `previousPrice` is used for calculating user fees instead of the real-time price. Any user can update the cached price through the `updatePrice` function. ERC20Paymaster updates the price at the end of the `_postOp` operation to meet usage requirements. However, it should be noted that the `tokenRequiredFund` value calculated at this time still uses the outdated cached price, which will lead to inaccurate fee calculation results, causing ERC20Paymaster to potentially overcharge or undercharge fees.

Code location: soul-wallet-contract/contracts/paymaster/ERC20Paymaster.sol#L115-L121

```
function _postOp(PostOpMode mode, bytes calldata context, uint256 actualGasCost,
uint256 actualUserOpFeePerGas)
    internal
    override
{
    ...
    uint256 tokenRequiredFund =
        (actualGasCost + costOfPost) * supportedToken[token].priceMarkup *
cachedPrice / (1e18 * PRICE_DENOMINATOR);
    if (sponsorWalletCreation) {
        // if sponsor during wallet creatation, charge the acutal amount
        IERC20Metadata(token).safeTransferFrom(sender, address(this),
tokenRequiredPreFund);
    } else if (sponsorWalletCreation == false && tokenRequiredPreFund >
tokenRequiredFund) {
        // refund unused precharge token
        IERC20Metadata(token).safeTransfer(sender, tokenRequiredPreFund -
tokenRequiredFund);
    }
}
```

```
    }  
    // update oracle  
    uint192 latestTokenPrice = fetchPrice(supportedToken[token].tokenOracle);  
    uint192 nativeAssetPrice = fetchPrice(nativeAssetOracle);  
    supportedToken[token].previousPrice =  
        nativeAssetPrice * uint192(supportedToken[token].tokenDecimals) /  
    latestTokenPrice;  
    emit UserOperationSponsored(sender, token, tokenRequiredFund, actualGasCost);  
}
```

Solution

It is recommended to update `previousPrice` before calculating `tokenRequiredFund` in the `_postOp` function to avoid the above risks.

Status

Fixed; Fixed in commit 4ea42482115c9a5dd51250a88904aa298aaee465. However, it should be noted that if the protocol undercharged fees previously, it will not require users to make up the difference again.

[N17] [Suggestion] Potential risks of one-time maximum approval

Category: Design Logic Audit

Content

In the protocol, the `AaveUsdcSaveAutomation` contract is used to provide the functionality of depositing USDC tokens into AAVE. During contract initialization, it approves an allowance of `uint256.max` to AAVE, allowing users to make deposits to AAVE through the `AaveUsdcSaveAutomation` contract. Despite approving the maximum allowance, if a large number of users utilize the contract, this allowance will eventually be exhausted in the future. Since this contract cannot be upgraded again, it poses the risk of rendering the `AaveUsdcSaveAutomation` contract unusable.

Code location: `soul-wallet-contract/contracts/automation/AaveUsdcSaveAutomation.sol#L36`

```
constructor(address _owner, address _usdcAddr, address _aaveUsdcPoolAddr)  
Ownable(_owner) {  
    usdcToken = IERC20(_usdcAddr);  
    aave = IAaveV3(_aaveUsdcPoolAddr);  
    usdcToken.approve(address(aave), 2 ** 256 - 1);  
}
```

Solution

It is recommended to approve AAVE during each deposit instead of a one-time approval to address the issue of allowance exhaustion.

Status

Acknowledged

[N18] [Suggestion] Oracle updatedAt checks can be optimized

Category: Design Logic Audit

Content

In the ERC20Paymaster contract, the fetchPrice function is used to fetch the price from the Chainlink oracle. It checks whether the oracle price update time updatedAt is within two days. It should be noted that the Chainlink oracle has different heartbeat intervals for different tokens. When the time interval exceeds the preset heartbeat interval, an update will be triggered. Therefore, it is not reasonable to perform a two-day interval check for all tokens.

Code location: soul-wallet-contract/contracts/paymaster/ERC20Paymaster.sol#L226

```
function fetchPrice(IOracle _oracle) internal view returns (uint192 price) {  
    ...  
    require(updatedAt >= (block.timestamp - 2 days), "Paymaster: Incomplete  
round");  
    ...  
}
```

Solution

It is recommended to check updatedAt based on the heartbeat intervals of different tokens.

Status

Acknowledged

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002406040001	SlowMist Security Team	2024.05.16 - 2024.06.04	Passed

Summary conclusion: The SlowMist security team uses a manual and SlowMist team's analysis tool to audit the project, during the audit work we found 1 critical risk, 3 medium risks, 2 low risks, 5 suggestions, and 7 information. All the findings were fixed or acknowledged. The code was not deployed to the mainnet.

6 Statement

SlowMist issues this report with reference to the facts that have occurred or existed before the issuance of this report, and only assumes corresponding responsibility based on these.

For the facts that occurred or existed after the issuance, SlowMist is not able to judge the security status of this project, and is not responsible for them. The security audit analysis and other contents of this report are based on the documents and materials provided to SlowMist by the information provider till the date of the insurance report (referred to as "provided information"). SlowMist assumes: The information provided is not missing, tampered with, deleted or concealed. If the information provided is missing, tampered with, deleted, concealed, or inconsistent with the actual situation, the SlowMist shall not be liable for any loss or adverse effect resulting therefrom. SlowMist only conducts the agreed security audit on the security situation of the project and issues this report. SlowMist is not responsible for the background and other conditions of the project.



Official Website
www.slowmist.com



E-mail
team@slowmist.com



Twitter
[@SlowMist_Team](https://twitter.com/SlowMist_Team)



Github
<https://github.com/slowmist>