Clave Security Review

Pashov Audit Group

Conducted by: Said, ubermensch, pontifex

November 2nd - November 11th

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **Abstract-Foundation/clave-contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Clave

Clave is a self-custodial smart wallet built on ZKsync that allows users to manage their on-chain assets with account abstraction. It enables transactions with any token for gas fees, offers customizable user features like nicknames, and simplifies asset transfers through link-based sharing.

5. Risk Classification

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

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - b2c88fc704b41430a5225f60416e86161628c178

fixes review commit hash - bbc9b2e09c4dc213681c99534246b9d43545bd67

Scope

The following smart contracts were in scope of the audit:

- `AccountFactory`
- `ClaveImplementation`
- `ClaveProxy`
- `EOAValidator`
- `PasskeyValidator`
- `ERC1271Handler`
- `ValidationHandler`
- `BatchCaller`
- `BatchCaller`
- `ClaveStorage`
- `Errors`
- `SignatureDecoder`
- `HookManager`
- `ModuleManager`
- `OwnerManager`
- `UpgradeManager`
- `ValidatorManager`
- `TokenCallbackHandler`
- `Auth`
- `LinkedList`
- `VerifierCaller`
- `ClaveRegistry`
- `BootloaderAuth`
- `SelfAuth`
- `HookAuth`
- `ModuleAuth`

7. Executive Summary

Over the course of the security review, Said, ubermensch, pontifex engaged with Clave to review Clave. In this period of time a total of **11** issues were uncovered.

Protocol Summary

Protocol Name	Clave
Repository	https://github.com/Abstract-Foundation/clave-contracts
Date	November 2nd - November 11th
Protocol Type	Account Abstraction

Findings Count

Severity	Amount
High	1
Medium	6
Low	4
Total Findings	**11**

Summary of Findings

ID	Title	Severity	Status
[H-01]	executeTransactionFromOutside does not increment the nonce	High	Resolved
[M-01]	Registered modules have access to critical functions	Medium	Acknowledged
[M-02]	removeHook should attempt to remove the context if it exists.	Medium	Resolved
[M-03]	Transaction fees can be underestimated due to early return	Medium	Resolved
[M-04]	Broken initial call functionality	Medium	Resolved
[M-05]	User can bypass deployment restrictions	Medium	Resolved
[M-06]	Use of constant authenticator data in WebAuthn Protocol	Medium	Acknowledged
[L-01]	Lack of payable fallback within ClaveImplementation	Low	Resolved
[L-02]	Lack of a maximum number check	Low	Acknowledged
[L-03]	Unexpected revert in validateTransaction due to incorrect hookData length	Low	Resolved
[L-04]	validateTransaction method reverts due to an invalid validator address	Low	Resolved

8. Findings

8.1. High Findings

[H-01] `executeTransactionFromOutside` does not increment the nonce

Severity

Impact: High

Likelihood: Medium

Description

`ClaveImplementation` implements `executeTransactionFromOutside`, which can be called by the wallet owner to manually execute a transaction as a fallback. However, this execution doesn't increment the nonce of the wallet.

```
function executeTransactionFromOutside(
        Transaction calldata transaction
    ) external payable override {
        // Check if msg.sender is authorized
        if (!_k1IsOwner(msg.sender)) {
            revert Errors.UNAUTHORIZED_OUTSIDE_TRANSACTION();
        }

        // Extract hook data from transaction.signature
        bytes[] memory hookData = SignatureDecoder.decodeSignatureOnlyHookData(
            transaction.signature
        );

        // Get the hash of the transaction
        bytes32 signedHash = transaction.encodeHash();

        // Run the validation hooks
        if (!runValidationHooks(signedHash, transaction, hookData)) {
            revert Errors.VALIDATION_HOOK_FAILED();
        }

        _executeTransaction(transaction);
    }
```

Consider a scenario where the execution operator is unresponsive, and the wallet owner decides to manually execute the transaction. When the operator becomes active again and processes the request, it can still execute the transaction because the nonce is not incremented when `executeTransactionFromOutside` is called, potentially causing unintended double execution.

Recommendations

Increment nonce inside `executeTransactionFromOutside` execution.

8.2. Medium Findings

[M-01] Registered modules have access to critical functions

Severity

Impact: High

Likelihood: Low

Description

Registered modules have access to several critical functions, such as `addModule` / `removeModule`, `r1AddOwner` / `k1AddOwner`, `r1RemoveOwner` / `k1RemoveOwner` / `resetOwners`, `r1AddValidator` / `k1AddValidator`, and `r1RemoveValidator` / `k1RemoveValidator` through `onlySelfOrModule` modifier. This excessive access could allow a maliciously registered module to hijack the wallet by changing its owner or validator.

Recommendations

Consider restricting the mentioned function to `onlySelf`, to avoid providing excessive access to the registered modules.

[M-02] `removeHook` should attempt to remove the context if it exists.

Severity

Impact: Medium

Likelihood: Medium

Description

Within the `runExecutionHooks` modifier, it iterates through all registered execution hooks, triggers the `preExecutionHook`, and sets the hook's `context`. Then, at the end of execution, it iterates again, triggers the `postExecutionHook` if the `context` exists, and finally deletes the context.

```
modifier runExecutionHooks(Transaction calldata transaction) {
        mapping
          (address => address) storage executionHooks = _executionHooksLinkedList();

        address cursor = executionHooks[AddressLinkedList.SENTINEL_ADDRESS];
        // Iterate through hooks
        while (cursor > AddressLinkedList.SENTINEL_ADDRESS) {
            // Call the preExecutionHook function with transaction struct
            bytes memory context = IExecutionHook(cursor).preExecutionHook
              (transaction);
            // Store returned data as context
            _setContext(cursor, context);

            cursor = executionHooks[cursor];
        }

        _;

        cursor = executionHooks[AddressLinkedList.SENTINEL_ADDRESS];
        // Iterate through hooks
        while (cursor > AddressLinkedList.SENTINEL_ADDRESS) {
            bytes memory context = _getContext(cursor);
            if (context.length > 0) {
                // Call the postExecutionHook function with stored context
                IExecutionHook(cursor).postExecutionHook(context);
                // Delete context
                _deleteContext(cursor);
            }

            cursor = executionHooks[cursor];
        }
    }
```

It is possible that during the execution of a function with the `runExecutionHooks` modifier calls the `removeHook` and remove one of the registered hooks. However, since the removed hook is no longer registered in `executionHooks`, its context will not be removed at the end of `runExecutionHooks` modifier's execution.

This could lead to unexpected behavior, especially if the removed hook is re-added in the future, as the old context could be used during some operations.

Recommendations

Remove the hook's context when `removeHook` is called.

[M-03] Transaction fees can be underestimated due to early return

Severity

Impact: Medium

Likelihood: Medium

Description

The `IAccount.validateTransaction` method besides validation `is also used during the gas fee estimation`. This way the protocol should `preserve as many steps as possible both for valid and invalid transactions`. In turn, the `ClaveImplementation.validateTransaction` function has an early return for the gas fee estimation calls. So many gas-consumed checks stay not estimated.

```
function _validateTransaction(
        bytes32 signedHash,
        Transaction calldata transaction
    ) internal returns (bytes4 magicValue) {
        if (transaction.signature.length == 65) {
            // This is a gas estimation
>>          return bytes4(0);
        }

        // Extract the signature, validator address and hook data from the
        // transaction.signature
        (
          bytesmemorysignature,
          addressvalidator,
          bytes[]memoryhookData
        ) = SignatureDecoder
            .decodeSignature(transaction.signature);

        // Run validation hooks
        bool hookSuccess = runValidationHooks
          (signedHash, transaction, hookData);

        if (!hookSuccess) {
>>          return bytes4(0);
        }

        bool valid = _handleValidation(validator, signedHash, signature);

        magicValue = valid ? ACCOUNT_VALIDATION_SUCCESS_MAGIC : bytes4(0);
    }
```

Recommendations

Consider following `IAccount` recommendations about the gas fee estimation and in case of obvious incorrect signature length making a signature look valid. As an example (https://code.zksync.io/tutorials/native-aa-multisig#prerequisites):

```
if (_signature.length != 130) {
            // Signature is invalid anyway, but we need to proceed with the
            // signature verification as usual
            // in order for the fee estimation to work correctly
            _signature = new bytes(130);

            // Making sure that the signatures look like a valid ECDSA signature
            // and are not rejected rightaway
            // while skipping the main verification process.
            _signature[64] = bytes1(uint8(27));
            _signature[129] = bytes1(uint8(27));
        }
```

The `hookSuccess` result can also be checked later.

```
bool hookSuccess = runValidationHooks
          (signedHash, transaction, hookData);

-       if (!hookSuccess) {
-           return bytes4(0);
-       }

        bool valid = _handleValidation(validator, signedHash, signature);

-       magicValue = valid ? ACCOUNT_VALIDATION_SUCCESS_MAGIC : bytes4(0);
```

[M-04] Broken initial call functionality

Severity

Impact: Medium

Likelihood: Medium

Description

Though the `ClaveImplementation.initialize` function can proceed an initial call with arbitrary `value`, the `payable` modifier is missed both in the `initialize` and in the `AccountFactory.deployAccount` functions. The `deployAccount` function also initializes an account with hardcoded `0` value. This way the initial call functionality is implemented only partially.

```
function deployAccount(
        bytes32 salt,
        bytes memory initializer
>>  ) external returns (address accountAddress) {
<...>
        // Initialize the account
        bool initializeSuccess;

        assembly ('memory-safe') {
            initializeSuccess := call(
                gas(),
                accountAddress,
>>              0,
                add(initializer, 0x20),
                mload(initializer),
                0,
                0
            )
        }
<...>
    function initialize(
        address initialK1Owner,
        address initialK1Validator,
        bytes[] calldata modules,
        Call calldata initCall
>>  ) public initializer {
<...>
        if (initCall.target != address(0)) {
>>          uint128 value = Utils.safeCastToU128(initCall.value);
            _executeCall(
              initCall.target,
              value,
              initCall.callData,
              initCall.allowFailure
            );
        }
    }
```

Recommendations

Consider declaring the `deployAccount` and `initialize` functions as `payable` and providing `msg.value` as the `value` parameter instead of hardcoded zero.

[M-05] User can bypass deployment restrictions

Severity

Impact: Medium

Likelihood: Medium

Description

The `deployAccount` function is intended to allow users to deploy their own smart accounts with controlled initialization parameters. It accepts user-provided `initializer` data, which is expected to call the `initialize` function of the deployed `claveProxy` contract to set up the account correctly.

```
function deployAccount(
    bytes32 salt,
    bytes memory initializer
) external returns (address accountAddress) {
    // Deployment logic
}
```

During deployment, a `claveProxy` contract is created:

```
(
  boolsuccess,
  bytesmemoryreturnData
) = SystemContractsCaller.systemCallWithReturndata(
    uint32(gasleft()),
    address(DEPLOYER_SYSTEM_CONTRACT),
    uint128(0),
    abi.encodeCall(
        DEPLOYER_SYSTEM_CONTRACT.create2Account,
        (
            salt,
            proxyBytecodeHash,
            abi.encode(implementationAddress),
            IContractDeployer.AccountAbstractionVersion.Version1
        )
    )
);
```

After deployment, the user-provided `initializer` calldata is passed directly to the deployed contract via a low-level call:

```
assembly ('memory-safe') {
    initializeSuccess := call(
        gas(),
        accountAddress,
        0,
        add(initializer, 0x20),
        mload(initializer),
        0,
        0
    )
}
```

The `initialize` function in the `claveProxy` contract includes checks to ensure that only authorized deployers can set the `initialK1Owner`:

```
if (thisDeployer != factory.deployer()) {
    if (initialK1Owner != thisDeployer) {
        revert Errors.NOT_FROM_DEPLOYER();
    }
}
```

However, because the `initializer` data is entirely user-controlled, an attacker can craft it to call a different function and then deploy his own clone of the `AccountFactory` contract to call the `initialize` function using it, setting any K1 owner address they want to bypass the check.

Recommendations

Restrict or validate the user-supplied `initializer` data to ensure it exclusively calls the `initialize` function with trusted parameters. Consider hardcoding the function selector to the `initialize` function within the `deployAccount` function to prevent manipulation.

[M-06] Use of constant authenticator data in WebAuthn Protocol

Severity

Impact: Medium

Likelihood: Medium

Description

The `PasskeyValidator` contract's `_validateSignature` function is responsible for verifying signatures derived from the WebAuthn protocol. This verification process constructs a message hash that includes the transaction hash, specific WebAuthn protocol strings, and a constant `AUTHENTICATOR_DATA`:

```
clientData = bytes.concat(
    bytes(ClIENT_DATA_PREFIX),
    challengeBase64,
    bytes(IOS_ClIENT_DATA_SUFFIX)
);

bytes32 message = _createMessage(AUTHENTICATOR_DATA, clientData);
```

The `AUTHENTICATOR_DATA` is defined as a constant with a hardcoded counter value of zero:

```
// hash of 'https://getclave.io' +
//(BE, BS, UP, UV) flags set + unincremented sign counter
bytes constant AUTHENTICATOR_DATA =

        hex'175faf8504c2cdd7c01778a8b0efd4874ecb3aefd7ebb7079a941f7be8897d411d0000000
```

The `authData` is composed of the following fields, concatenated in the order presented:

1. `rpIdHash` **(32 bytes):** SHA-256 hash of the Relying Party Identifier (RP ID).
2. `flags` **(1 byte):** Bit flags indicating user presence, user verification, and the inclusion of additional data.
3. `signCount` **(4 bytes):** Signature counter to prevent replay attacks.
4. `attestedCredentialData` **(variable length, optional):** Includes credential information; present during registration.
5. `extensions` **(variable length, optional):** Contains extension data; included if extensions are used.

By using a constant `AUTHENTICATOR_DATA` with a static counter, the contract fails to account for the dynamic nature of the authenticator's counter. Authenticators (such as secure enclaves or TPMs) increment this counter with each signing operation. Consequently, the signatures produced by authenticators will include a counter value that does not match the static value expected by the contract.

This mismatch causes all signature verifications to fail, effectively leading to a denial of service (DoS) for any user account operations that rely on signature validation through the `PasskeyValidator`. Users are then forced to rely on alternative validators or key owners, undermining the security and usability of the system.

Recommendations

Modify the contract to accept and process dynamic `authenticatorData` that includes the incrementing counter from the authenticator.

8.3. Low Findings

[L-01] Lack of payable `fallback` within `ClaveImplementation`

It is possible that, in some instances, a wallet may want to receive native ETH with a transaction that contains data. However, due to the lack of a `fallback payable` function in `ClaveImplementation`, an attempt to transfer native ETH with a transaction that contains data will always revert. Consider adding `fallback payable` inside `ClaveImplementation`.

[L-02] Lack of a maximum number check

`runValidationHooks` and `runExecutionHooks` iterate through the registered `validationHooks` and `executionHooks` to perform the defined operations. However, these operations could run out of gas if there are too many registered hooks, potentially leading to unexpected reverts and a poor user experience. Consider restricting the number of registered hooks within the wallet.

[L-03] Unexpected revert in `validateTransaction` due to incorrect `hookData` length

The `HookManager.runValidationHooks` function accepts `hookData` array with arbitrary length and returns `false` when `hookData.length != idx`. But in fact, the check only validates if `hookData.length` exceeds the number of validators. In turn, there can be an unexpected revert when `hookData.length` is less than the number of validators. Consider calling the next validator only if `idx < hookData.length` and returning `false` otherwise.

```
function runValidationHooks(
        bytes32 signedHash,
        Transaction calldata transaction,
        bytes[] memory hookData
    ) internal returns (bool) {
        mapping
          (address => address) storage validationHooks = _validationHooksLinkedList();

        address cursor = validationHooks[AddressLinkedList.SENTINEL_ADDRESS];
        uint256 idx = 0;
        // Iterate through hooks
        while (cursor > AddressLinkedList.SENTINEL_ADDRESS) {
            // Call it with corresponding hookData
            bool success = _call(
                cursor,
                abi.encodeWithSelector(
                    IValidationHook.validationHook.selector,
                    signedHash,
                    transaction,
>>                  hookData[idx++]
                )
            );

            if (!success) {
                return false;
            }

            cursor = validationHooks[cursor];
        }

        // Ensure that hookData is not tampered with
>>      if (hookData.length != idx) return false;

        return true;
    }
```

[L-04] `validateTransaction` method reverts due to an invalid `validator` address

An invalid `validator` address parameter can cause a revert in the `ClaveImplementation.validateTransaction` function instead of returning `bytes4(0)` on an incorrect signature because of `validAddress` modifier in the `AddressLinkedList.exists` view function. Though it is not obligatory, it is better to abstain from reverting in order to allow for fee estimation to work. Consider returning `false` in the `_handleValidation` when `validator <= SENTINEL_ADDRESS`.

```
<...>
    function _handleValidation(
        address validator,
        bytes32 signedHash,
        bytes memory signature
    ) internal view returns (bool) {
>>      if (_r1IsValidator(validator)) {
<...>
>>      } else if (_k1IsValidator(validator)) {
<...>
        }

        return false;
    }
<...>
    function _r1IsValidator(address validator) internal view returns (bool) {
        return _r1ValidatorsLinkedList().exists(validator);
    }

    function _k1IsValidator(address validator) internal view returns (bool) {
        return _k1ValidatorsLinkedList().exists(validator);
    }
<...>
    modifier validAddress(address value) {
        if (value <= SENTINEL_ADDRESS) {
>>          revert Errors.INVALID_ADDRESS();
        }
        _;
    }
<...>
    function exists(
        mapping(address => address) storage self,
        address value
>>  ) internal view validAddress(value) returns (bool) {
        return self[value] != address(0);
    }
```