Clave Security Review

Pashov Audit Group

Conducted by: Said, ast3ros, pontifex

December 23th 2024 - December 26th 2024

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 - 3e5cb6784ad067ecba23758aa39ab78468026f5d

fixes review commit hash - 1aba0968925632b57bccbfc07bc488687f3e8937

Scope

The following smart contracts were in scope of the audit:

- `ClaveImplementation`
- `BatchCaller`
- `TimestampAsserter`
- `TimestampAsserterLocator`
- `ClaveStorage`
- `Errors`
- `SessionLib`
- `ValidatorManager`
- `SessionKeyValidator`

7. Executive Summary

Over the course of the security review, Said, ast3ros, pontifex engaged with Clave to review Clave. In this period of time a total of **8** issues were uncovered.

Protocol Summary

Protocol Name	Clave
Repository	https://github.com/Abstract-Foundation/clave-contracts
Date	December 23th 2024 - December 26th 2024
Protocol Type	Account Abstraction Wallet

Findings Count

Severity	Amount
Medium	4
Low	4
Total Findings	**8**

Summary of Findings

ID	Title	Severity	Status
[M-01]	SessionKeyValidator is not working on ZkSync mainnet	Medium	Resolved
[M-02]	sessionCounter check not enforced when removing SessionKeyValidator	Medium	Resolved
[M-03]	SessionKeyValidator module cannot be removed properly	Medium	Resolved
[M-04]	validateFeeLimit does not properly limit the transaction fee	Medium	Resolved
[L-01]	Leaving transient storage slots with nonzero values	Low	Acknowledged
[L-02]	sessionStatus and sessionState does not consider expiresAt	Low	Resolved
[L-03]	Incorrect parameter extraction for dynamic types over 32 bytes	Low	Acknowledged
[L-04]	Not restricting SessionKeyValidator as the transaction target	Low	Resolved

8. Findings

8.1. Medium Findings

[M-01] `SessionKeyValidator` is not working on ZkSync mainnet

Severity

Impact: Low

Likelihood: High

Description

The SessionKeyValidator contract fails to function on ZkSync mainnet due to the `TimestampAsserterLocator.locate` function explicitly reverting on chainId `324` (ZkSync mainnet). This breaks all session key functionality for mainnet deployments.

```
function locate() internal view returns (ITimestampAsserter) {
    ...
    if (block.chainid == 324) { // @audit revert if the chain is ZkSync mainnet
      revert("Timestamp asserter is not deployed on ZKsync mainnet yet");
    }
    ...
  }
```

Recommendations

Update the `TimestampAsserterLocator` with the mainnet address.

[M-02] `sessionCounter` check not enforced when removing `SessionKeyValidator`

Severity

Impact: Medium

Likelihood: Medium

Description

When users remove `SessionKeyValidator`, it will ensure that `sessionCounter` is 0 before triggering `removeModuleValidator` and `removeHook` to complete the module removal. The `sessionCounter` check is necessary to prevent the use of active session keys from the past if the module is installed again later.

```
function disable() external {
    if (_isInitialized(msg.sender)) {
      // Here we have to revoke all keys, so that if the module
      // is installed again later, there will be no active sessions from the
      // past.
      // Problem: if there are too many keys, this will run out of gas.
      // Solution: before uninstalling, require that all keys are revoked
      // manually.
>>>   require(sessionCounter[msg.sender] == 0, "Revoke all keys first");
      IValidatorManager(msg.sender).removeModuleValidator(address(this));
      IHookManager(msg.sender).removeHook(address(this), true);
    }
  }
```

However, if `disable` is reverted due to the `sessionCounter` check, the module removal still succeeds because the success status of the `disable` call is not checked.

```
function _removeModule(address module) internal {
        _modulesLinkedList().remove(module);

>>>     (bool success, ) = module.excessivelySafeCall(
            gasleft(),
            0,
            0,
            abi.encodeWithSelector(IInitable.disable.selector)
        );
        (success); // silence unused local variable warning

        emit RemoveModule(module);
    }
```

Recommendations

When removing a module, consider adding an option (additional param) to check if the `disable` call is successful. Alternatively, move the `sessionCounter[msg.sender] == 0` check to `init` instead.

[M-03] `SessionKeyValidator` module cannot be removed properly

Severity

Impact: Medium

Likelihood: Medium

Description

When the module is removed, `disable` will be called after the module has been removed from the registered list.

```
function _removeModule(address module) internal {
        _modulesLinkedList().remove(module);

>>>     (bool success, ) = module.excessivelySafeCall(
            gasleft(),
            0,
            0,
            abi.encodeWithSelector(IInitable.disable.selector)
        );
        (success); // silence unused local variable warning

        emit RemoveModule(module);
    }
```

Inside `SessionKeyValidator`, it attempts to trigger `removeModuleValidator` and `removeHook`, which are restricted functions that can only be called via self-call or by registered modules.

```
function disable() external {
    if (_isInitialized(msg.sender)) {
      // Here we have to revoke all keys, so that if the module
      // is installed again later, there will be no active sessions from the
      // past.
      // Problem: if there are too many keys, this will run out of gas.
      // Solution: before uninstalling, require that all keys are revoked
      // manually.
      require(sessionCounter[msg.sender] == 0, "Revoke all keys first");
>>>   IValidatorManager(msg.sender).removeModuleValidator(address(this));
>>>   IHookManager(msg.sender).removeHook(address(this), true);
    }
  }
```

```
function removeModuleValidator(address validator) external onlySelfOrModule {
        _removeModuleValidator(validator);
    }
```

```
function removeHook
      (address hook, bool isValidation) external override onlySelfOrModule {
        _removeHook(hook, isValidation);
    }
```

But since `SessionKeyValidator` is no longer registered as a module, the calls will silently fail, as `removeModule` doesn't enforce the `disable` call to succeed. This will wrongly give the impression to users that `SessionKeyValidator` is no longer registered as a hook and module validator.

Recommendations

Consider moving `disable` call before removing the module from the list.

[M-04] `validateFeeLimit` does not properly limit the transaction fee

Severity

Impact: Medium

Likelihood: Medium

Description

`validateFeeLimit` will be called within `SessionKeyValidator`'s `validationHook` to ensure that the caller's provided `maxFeePerGas` and `gasLimit` is within the usage limit.

```
function validationHook(
    bytes32signedHash,
    Transactioncalldatatransaction,
    bytescalldatahookData
  ) external {
    if (hookData.length == 0) {
      // There's no session data so we aren't validating anything
      return;
    }
    (bytes memory signature, address validator, ) = abi.decode
      (transaction.signature, (bytes, address, bytes[]));
    if (validator != address(this)) {
      // This transaction is not meant to be validated by this module
      return;
    }
    (
      SessionLib.SessionSpecmemoryspec,
      uint64[]memoryperiodIds
    ) = abi.decode(
      hookData,
      (SessionLib.SessionSpec, uint64[])
    );
    require(spec.signer != address(0), "Invalid signer (empty)");
    bytes32 sessionHash = keccak256(abi.encode(spec));
    // this generally throws instead of returning false
    sessions[sessionHash].validate(transaction, spec, periodIds);
    (
      addressrecoveredAddress,
      ECDSA.RecoverErrorrecoverError,

    ) = ECDSA.tryRecover(signedHash, signature
    if
      (recoverError != ECDSA.RecoverError.NoError || recoveredAddress == address(0)) {
      return;
    }
    require(recoveredAddress == spec.signer, "Invalid signer (mismatch)");
    // This check is separate and performed last to prevent gas estimation
    // failures
>>> sessions[sessionHash].validateFeeLimit(transaction, spec, periodIds[0]);

    // Set the validation result to 1 for this hash, so that isValidSignature
    // succeeds
    uint256 slot = uint256(signedHash);
    assembly {
      tstore(slot, 1)
    }
  }
```

However, `validateFeeLimit` incorrectly checks the fee when `paymaster` is configured, instead of when it is not.

```
function validateFeeLimit(
    SessionStorage storage state,
    Transaction calldata transaction,
    SessionSpec memory spec,
    uint64 periodId
  ) internal {
    // This is split from `validate` to prevent gas estimation failures.
    // When this check was part of `validate`, gas estimation could fail due to
    // fee limit being smaller than the upper bound of the gas estimation binary
    // search.
    // By splitting this check, we can now have this order of operations in
    // `validateTransaction`:
    // 1. session.validate()
    // 2. ECDSA.tryRecover()
    // 3. session.validateFeeLimit()
    // This way, gas estimation will exit on step 2 instead of failing, but will
    // still run through
    // most of the computation needed to validate the session.

    // TODO: update fee allowance with the gasleft/refund at the end of
    // execution
>>> if (transaction.paymaster != 0) {
      // If a paymaster is paying the fee, we don't need to check the fee limit
      uint256 fee = transaction.maxFeePerGas * transaction.gasLimit;
      spec.feeLimit.checkAndUpdate(state.fee, fee, periodId);
    }
  }
```

This will cause the fee limit to not work properly, requiring the session key caller to provide a fee without any limit.

Recommendations

Update the `checkAndUpdate` condition to the following:

```solidity
  function validateFeeLimit(
    SessionStorage storage state,
    Transaction calldata transaction,
    SessionSpec memory spec,
    uint64 periodId
  ) internal {
    // This is split from `validate` to prevent gas estimation failures.
    // When this check was part of `validate`, gas estimation could fail due to
    // fee limit being smaller than the upper bound of the gas estimation binary
    // search.
    // By splitting this check, we can now have this order of operations in
    // `validateTransaction`:
    // 1. session.validate()
    // 2. ECDSA.tryRecover()
    // 3. session.validateFeeLimit()
    // This way, gas estimation will exit on step 2 instead of failing, but will
    // still run through
    // most of the computation needed to validate the session.

    // TODO: update fee allowance with the gasleft/refund at the end of
    // execution
-    if (transaction.paymaster != 0) {
+    if (transaction.paymaster == 0) {
      // If a paymaster is paying the fee, we don't need to check the fee limit
      uint256 fee = transaction.maxFeePerGas * transaction.gasLimit;
      spec.feeLimit.checkAndUpdate(state.fee, fee, periodId);
    }
  }
```

8.2. Low Findings

[L-01] Leaving transient storage slots with nonzero values

The Security considerations from the EIP-1153 standard do not recommend leaving nonzero values in transient storage slots (https://eips.ethereum.org/EIPS/eip-1153#security-considerations).

```
function handleValidation
    (bytes32 signedHash, bytes memory signature) external view returns (bool) {
    // This only succeeds if the validationHook has previously succeeded for
    // this hash.
    uint256 slot = uint256(signedHash);
    uint256 hookResult;
    assembly {
>>    hookResult := tload(slot)
    }
    require(
      hookResult==1,
      "Can'tcallthisfunctionwithoutcallingvalidationHook"
    );
    return true;
  }
```

[L-02] `sessionStatus` and `sessionState` does not consider `expiresAt`

When `sessionStatus` and `sessionState` are called by the user to get the provided session key status, it will return `Active`, even if the session has already expired. This returned value could mislead users, implying that the session key can still be used. Consider adding an additional state (e.g., `Expired`), or returning `Closed` if the session has already expired.

[L-03] Incorrect parameter extraction for dynamic types over 32 bytes

The `checkAndUpdate` function incorrectly extracts calldata parameters when handling dynamic types larger than 32 bytes. The function assumes all parameters can be extracted using a fixed 32-byte offset, but this fails for dynamic types that use a more complex encoding scheme.

```
function checkAndUpdate(
    Constraint memory constraint,
    UsageTracker storage tracker,
    bytes calldata data,
    uint64 period
  ) internal {
    uint256 index = 4 + constraint.index * 32;
    bytes32 param = bytes32
    //(data[index:index + 32]); // @audit Incorrect assumption that parameter can be e
    Condition condition = constraint.condition;
    bytes32 refValue = constraint.refValue;

    ...
  }
```

Let's consider the following example:

This is the function signature:

```
contract Example {
    function exampleFunction
      (string memory text, uint256[] memory numbers) external {}
}
```

There are two cases:

1. First case when the text is less than 32 bytes - hello word is included in 32-byte word.

Input: `hello`, [1,2,3]

```
-
-   0x303430f7                                                         // function signat
-
-   0000000000000000000000000000000000000000000000000000000000000040 // offset for text
-
-   0000000000000000000000000000000000000000000000000000000000000080 // offset for numb
-
-   0000000000000000000000000000000000000000000000000000000000000005 // length of strin
- 68656c6c6f000000000000000000000000000000000000000000000000000000 // hello
-
-   0000000000000000000000000000000000000000000000000000000000000003 // length of the a
- 0000000000000000000000000000000000000000000000000000000000000001 // 1
- 0000000000000000000000000000000000000000000000000000000000000002 // 2
- 0000000000000000000000000000000000000000000000000000000000000003 // 3
```

2. Second case when the text is more than 32 bytes - The data spans multiple 32-byte words.

Input: `helloooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo oo`, [1,2,3]

```
-
-  0x303430f7                                                       // function signat
-
-  0000000000000000000000000000000000000000000000000000000000000040 // offset for text
-
-  00000000000000000000000000000000000000000000000000000000000000c0 // offset for numb
-
-  0000000000000000000000000000000000000000000000000000000000000048 // length of strin
- 68656c6c6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f // text
- 6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f6f // text
- 6f6f6f6f6f6f6f6f000000000000000000000000000000000000000000000000 // text
-
-  0000000000000000000000000000000000000000000000000000000000000003 // length of the a
- 0000000000000000000000000000000000000000000000000000000000000001 // 1
- 0000000000000000000000000000000000000000000000000000000000000002 // 2
- 0000000000000000000000000000000000000000000000000000000000000003 // 3
```

In the second case, the text cannot be extracted correctly using one index like in `checkAndUpdate` function. It leads to misinterpretation of the actual parameter values with the following impact:

- Session key constraints can be bypassed
- Functions with dynamic parameters become unusable with session keys when constraints are set on the dynamic parameters.

Handle the dynamic types correctly in the `checkAndUpdate` function using multiple 32-byte words.

[L-04] Not restricting `SessionKeyValidator` as the transaction target

Within `SessionLib.validate`, it checks `transaction.to` to ensure it is not set to `msg.sender`, preventing callers from abusing session keys to administer the smart account. However, it allows `transaction.to` to be set to `SessionKeyValidator`, which could enable callers to create additional session keys with unrestricted calls if allowed.

```
function validate(
    SessionStorage storage state,
    Transaction calldata transaction,
    SessionSpec memory spec,
    uint64[] memory periodIds
  ) internal {
    // Here we additionally pass uint64[] periodId to check allowance limits

    // periodIds[0] is for fee limit,
    // periodIds[1] is for value limit,
    // periodIds[2:] are for call constraints, if there are any.
    // It is required to pass them in
    //(instead of computing via block.timestamp) since during validation
    // we can only assert the range of the timestamp, but not access its value.

    require(state.status[msg.sender] == Status.Active, "Session is not active");
    TimestampAsserterLocator.locate().assertTimestampInRange(0, spec.expiresAt);

    address target = address(uint160(transaction.to));

    if (transaction.paymasterInput.length >= 4) {
      bytes4 paymasterInputSelector = bytes4(transaction.paymasterInput[0:4]);
      require(
        paymasterInputSelector != IPaymasterFlow.approvalBased.selector,
        "Approval based paymaster flow not allowed"
      );
    }

    if (transaction.data.length >= 4) {
      // Disallow self-targeting transactions with session keys as these have
      // the ability to administer
      // the smart account.
>>>   require(address(uint160
  (transaction.to)) != msg.sender, "Can not target self");

   // ...
  }
```

Consider reverting if `transaction.to` is set to the `SessionKeyValidator` address.