HYTOPIA wallet Security Review

Pashov Audit Group

Conducted by: Dan Ogurtsov, __141345__

December 13th 2023 - December 16th 2023

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 **wallet** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About wallet

The HYTOPIA wallet manages user accounts on the blockchain. It has functionalities such as multi-signer authority model, gasless transactions and session authorization. The wallet conforms to several standards including EIP-1271 for smart contract signatures, EIP-4337 for account abstraction, and EIP-1967 for beacon proxies. This allows HYTOPIA to create, transact, and manage users' accounts without holding total custody over their assets.

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 **- 0d38699f148577f137e709ada64bc47fdd65924f**

fixes review commit hash **- 533d5fdb12385aa50f7ce7c60f0d17d33ea7d953**

Scope

The following smart contracts were in scope of the audit:

- `BeaconProxyFactory`
- `WalletProxyFactory`
- `utils/Bytes`
- `utils/Signatures`
- `modules/Versioned/Versioned`
- `modules/SessionCalls/SessionCalls`
- `modules/SessionCalls/SessionCallsStorage`
- `modules/SessionCalls/SessionCallsStructs`
- `modules/PreauthorizedCalls/PreauthorizedCalls`
- `modules/PreauthorizedCalls/PreauthorizedCallsStorage`
- `modules/PreauthorizedCalls/PreauthorizedCallsStructs`
- `modules/Main/Main`
- `modules/Main/MainStorage`
- `modules/ERC1271/ERC1271`
- `modules/Controllers/Controllers`
- `modules/Controllers/ControllersStorage`
- `modules/Calls/Calls`
- `modules/Calls/CallsStructs`
- `interfaces/**`

7. Executive Summary

Over the course of the security review, Dan Ogurtsov, __141345__ engaged with HYTOPIA to review wallet. In this period of time a total of **7** issues were uncovered.

Protocol Summary

Protocol Name	wallet
Repository	https://github.com/hytopiagg/wallet
Date	December 13th 2023 - December 16th 2023
Protocol Type	smart contract wallet

Findings Count

Severity	**Amount**
High	3
Medium	2
Low	2
Total Findings	**7**

Summary of Findings

ID	Title	Severity	Status
[H-01]	Missing payable function	High	Resolved
[H-02]	Threshold conflicts with removing controllers	High	Resolved
[H-03]	Wallet initialize frontrun	High	Resolved
[M-01]	RestrictedFunction could miss come cases	Medium	Resolved
[M-02]	Working with signatures - nonce and expiry	Medium	Resolved
[L-01]	CallRequestPreauthorization additional checks	Low	Acknowledged
[L-02]	addControllers() does not check existing controllers	Low	Resolved

8. Findings

8.1. High Findings

[H-01] Missing `payable` function

Severity

Impact: Medium, wallet will not receive native tokens but expected

Likelihood: High, because the use of msg.value and sending native token is a frequent operation

Description

`_call()` could send native token via `.call{ value: _callRequest.value }`, however, none of the contracts can receive native token, including Main.sol, PreauthorizedCalls.sol, SessionCalls.sol, Calls.sol, Controllers.sol.

```
File: contracts\modules\Calls\Calls.sol
80:     function _call
  (CallsStructs.CallRequest calldata _callRequest) internal returns (bytes memory) {
81:         (
  boolsuccess,
  bytesmemoryresult
) = _callRequest.target.call{ value: _callRequest.value }(_callRequest.data
82:
```

Recommendations

- Add payable modifier to the related functions
- Add `receive()` function

[H-02] Threshold conflicts with removing controllers

Severity

Impact: High, attack vector to capture the wallet

Likelihood: Medium, relevant not for all sets of controllers

Description

There is a drawback to using absolute weights - sometimes we should decrease the Threshold before removing a controller. Decreasing Threshold is a strong voting rules adjustment, and can be maliciously used by any controller.

Imagine we have a set of controllers A=50, B=50. TotalWeight=100, Threshold=100. Then, imagine user B asks to exit.

```
function _removeController(address _controller) internal {
        ControllersStorage.layout().totalWeights -= ControllersStorage.layout
          ().weights[_controller];
        if (
            ControllersStorage.layout().totalWeights == 0
                || ControllersStorage.layout
                  ().totalWeights < ControllersStorage.layout().threshold
        ) {
            revert ThresholdImpossible();
        }
        ControllersStorage.layout().weights[_controller] = 0;
    }
```

`_removeController()` will revert, as `totalWeight` will update to 50, but `threshold=100` is not updated, and we require that `totalWeight >= threshold`.

So, signers should have two transactions:

1. `updateControlThreshold()` to 50
2. `removeControllers()` removing B

If user A signs TX 1 - then user B can capture the wallet.

1. Controller B receives a signature from A
2. Controller B makes an atomic multicall = `updateControlThreshold()` to 50 + `removeControllers()` **removing controller A**. This last call requires only one signature, as the threshold was changed to 50.

As a result, Controller B managed to remove Controller A and capture the wallet.

Recommendations

This issue is hard to fix. Some new rules should be introduced to deal with this scenario. If `totalWeight` falls below the current `threshold`, this delta voting power **can be** somehow distributed among other controllers. **Or**, `totalWeight` can be allowed to fall only to the current `threshold`. Consider the following idea. Imagine `totalWeight` of 100 should be decreased by 20, when the current threshold is 95.

1. 5/20 can be decreased without distribution. `totalWeight` is changed to 95.
2. 15/20 can be ignored. As a result, `totalWeight` is 95 and `threshold` is 95, which means 100% of consensus required.

Also, it must be noted that the same thing is relevant to `updateControllerWeight()` as it can be used to decrease weight for a given controller, and `totalWeight` is decreased there as well.

[H-03] Wallet initialize frontrun

Severity

Impact: High, attack to secretly own new wallets of other users

Likelihood: Medium, frontrun required

Description

The deploy script does 2 transactions to deploy a wallet:

1. deploy contract
2. call initialize() (which accepts the first controller) (foundry, every call is a transaction)

```
address _newWalletAddr = _factory.createProxy(_testWalletSalt);
        Main(payable(_newWalletAddr)).initialize(deployer);
```

Wallet deployments can be tracked and awaited. An attacker can frontrun and call `initialize()` with malicious input.

One of the ideas:

1. Attacker frontruns, sets `Controller=Attacker`

2. Attacker updates implementation (`Main.upgradeToAndCall()`), to the malicious implementation
3. Sets back `controller=TargetUser`
4. The user keeps operation with this wallet, with the malicious implementation
5. Implementation has some malicious function to withdraw all tokens to the attacker. The attacker waits for some balance on the wallet, and invokes the attack function.

Recommendations

Ensure the `initialize()` is called in the same transaction as the wallet deployment.

8.2. Medium Findings

[M-01] `RestrictedFunction` could miss come cases

Severity

Impact: High, because the fund in the contract could all be stolen, much beyond allowed amount.

Likelihood: Low, because it only applies to cases when MAGIC_CONTRACT_ALL_FUNCTION_SELECTORS being used, and not all tokens have this problem.

Description

Only `approve` and `setApprovalForAll` are considered restricted function, however it is possible that the token allowance could be chaged by other functions: Such as RNDR, has `increaseApproval()/decreaseApproval()` function, which is not standard ERC20 functions.

```
File: contracts\modules\SessionCalls\SessionCalls.sol
448:     function _isRestrictedFunction
  (bytes4 _functionSelector) private pure returns (bool isRestricted_) {
449:         if (
450:
               _functionSelector == IERC20.approve.selector || _functionSelector == IE
451:                 || _functionSelector == IERC721.setApprovalForAll.selector
452:                 || _functionSelector == IERC1155.setApprovalForAll.selector
453:         ) {
454:             isRestricted_ = true;
455:         }
456:     }
```

```
// https://etherscan.io/address/0x1a1fdf27c5e6784d1cebf256a8a5cc0877e73af0#code
  function increaseApproval(address _spender, uint _addedValue) public returns
    (bool) {
    allowed[msg.sender][_spender] = allowed[msg.sender][_spender].add
      (_addedValue);
    emit Approval(msg.sender, _spender, allowed[msg.sender][_spender]);
    return true;
  }

  function decreaseApproval
    (address _spender, uint _subtractedValue) public returns (bool) {
    uint oldValue = allowed[msg.sender][_spender];
    if (_subtractedValue > oldValue) {
      allowed[msg.sender][_spender] = 0;
    } else {
      allowed[msg.sender][_spender] = oldValue.sub(_subtractedValue);
    }
    emit Approval(msg.sender, _spender, allowed[msg.sender][_spender]);
    return true;
  }
```

Recommendations

- just remove `MAGIC_CONTRACT_ALL_FUNCTION_SELECTORS` option, or restrict it to privileged roles

[M-02] Working with signatures - nonce and expiry

Severity

Impact: High, some signers can extract personal benefits

Likelihood: Low, required mistakes and unique parity of voting power

Description

There are some not protected attack vectors.

- if controllers signed two transactions, msg.sender can decide on the ordering of these two transactions, which one two prioritize
- if some signers refused to sign, but some other signers signed - their signatures can be used months later, for some modified set of controller, or signers who refused can find a moment to sign when they see it beneficial

Some simplified example:

1. We have 3 controllers: A, B, C, each with weights 30, 30, 30
2. New proposal - update the weight of C to 40
3. A signs, B signs, C refuses to sign. Controller C saves the signatures of A and B.
4. After one year, Controller C asks to replace Controller C with his/her other wallet D. All controllers sign the multiple transactions with the result that we have 3 controllers - A, B, and D.
5. Controllers D and C belong to the same signer. Controller D signs the previously saved transaction of updating the weight of C to 40.
6. Now Controller C appears in the list of controllers. Now it is the set of controllers A, B, C, D - with weights of 30, 30, 30, 40.
7. It is not something that controllers A and B signed for.

There are two problems why it is possible:

1. Currently, `nonce` in inputs is used as a salt. Multiple verified transactions with the same nonce can exist. Now it is not compared with any storage nonce value. It should be some storage value nonce incremented by one on every call. If a new transaction with the inputted nonce 6 is signed, it means it can only be written when the active nonce in the storage is 6.

2. There is no `Expiry` argument in inputs which would be a part of _inputHash. It is a timestamp in the future which is compared to the current block.timestamp. If inputs with the expiry of 1 day are signed by some controllers, their signatures will not be used in one month.

Recommendations

1. Add some global `nonce` in the storage, e.g. at `ControllersStorage.layout()`, and make it increment on every call. Every function should check that the inputted `_nonce` is equal to this

`ControllersStorage.layout().nonce` `meetsControllersThreshold()` is a perfect place for this logic.

2. Add some `expiredAt` arguments in inputs, add it to hash

8.3. Low Findings

[L-01] CallRequestPreauthorization additional checks

`PreauthorizedCalls.preauthorizeCall()` accepts `CallRequestPreauthorization` as input. Now only .maxCalls checked Consider additionally checking that `lastCallTimestamp != 0` because it is an expected condition for further usage of `preauthorizedCall()`.

[L-02] addControllers() does not check existing controllers

`Controller.addControllers()` accepts the list of new controllers. There can be two mistakes in the input:

- some of the new controllers were previously added
- the list of new controllers has duplicates

In this case `totalWeights` will be increased, but `weights[_controller]` will be just rewritten to the last value. As a result, `totalWeights` will exceed the sum of weights[] for all controllers.

`_addController()` should check that the current weight of the new controller is zero before the new weight is written.