Fun Wallet review

Prepared by	Date
blockdev	June 27 to June 29, 2023

About Fun Wallet

Protocol for ERC-4337 based wallets and paymasters. It integrates web2 frameworks and even allows sending tokens to Twitter handles!

About Jiru LLC

We do smart contract and ZK code review to find and fix bugs. Currently, we are serving Ethereum Foundation and Spearbit.

Review Summary

This is a 3 day review of ERC-4337 based wallet and paymaster implementation. Several issues have been already identified which were not fixed at the time. They have been excluded from the report.

This is not a full security review. The goal of this quick review is to have more eyes on the code to surface bugs with a full audit planned in future.

The auditor make no warranties regarding the security of the code and do not warrant that the code is free from defects. By deploying or using the code, Fun and users of the contracts agree to use the code at their own risk.

Scope

The following contracts at commit 109fd8f01623ad192b14a4910c91c68148fac7ee are under scope:

```
├── deployer
│   ├── Create3Deployer.sol
│   ├── FunWalletFactory.sol
│   └── WalletInit.sol
├── modules
│   ├── Module.sol
│   └── actions
│       ├── AaveWithdraw.sol
│       ├── ApproveAndExec.sol
│       └── ApproveAndSwap.sol
├── oracles
│   ├── FeePercentOracle.sol
│   ├── TokenPriceOracle.sol
│   ├── TwitterOracle.sol
│   └── UniswapPriceOracle.sol
├── paymaster
│   ├── BasePaymaster.sol
│   ├── GaslessPaymaster.sol
│   └── TokenPaymaster.sol
├── utils
│   ├── CallLib.sol
│   ├── DataLib.sol
│   ├── HashLib.sol
│   ├── MultiCall.sol
│   └── TestToken.sol
├── validations
│   ├── RoleBasedAccessControl.sol
│   ├── UserAuthentication.sol
│   └── Validation.sol
└── wallet
    ├── FunWallet.sol
    ├── WalletExec.sol
    ├── WalletFee.sol
    ├── WalletModules.sol
    ├── WalletState.sol
    └── WalletValidation.sol
```

Findings Summary

Findings

[H-01] Target amount returned by `getTokenValueOfEth()` is incorrect

`getTokenValueOfEth()` calculates the target amount as:

```
return (((ethAmount * uint256(price) * (10 ** tokenDecimals)) / (10 **
chainlinkAggregatorDecimals))) / WEI_IN_ETH;
```

This is incorrect. At the time of review, a PR is up for review which updates this to:

```
return ((ethAmount * WEI_IN_ETH * (10 ** tokenDecimals))) / ((10 **
chainlinkAggregatorDecimals) * uint256(price));
```

This only works when the decimal used by Chainlink aggregator is 18.

Recommendation

Update it to:

```
return (ethAmount * (10 ** chainlinkAggregatorDecimals) * (10 **
tokenDecimals))/(uint256(price) * WEI_IN_ETH);
```

Here's the reasoning:

1. For `10**tokenDecimals`, you get `price/(10**chainlinkAggregatorDecimals)` ether.
2. So for 1 ether you get, `(10**tokenDecimals * 10**chainlinkAggregatorDecimals)/price` tokens.
3. For `ethAmount/WEI_IN_ETH` ether, you get `(ethAmount/WEI_IN_ETH)*(10**tokenDecimals * 10**chainlinkAggregatorDecimals)/price` tokens.
4. Simplifying the above, we get:

   ```
   (ethAmount * (10**tokenDecimals) * (10**chainlinkAggregatorDecimals)) /
   (uint256(price) * WEI_IN_ETH)
   ```

You can also test it out by having a test where you multiply chainlink price by 10, and increase `chainlinkAggregatorDecimals` by 1. The above formula returns the correct price,

but the PR formula does not.

[H-02] Issues with `UniswapPriceOracle`

The team already knows that the Uniswap Oracle as used in the protocol is not protected from pool manipulation attacks. To fetch the price of a token against `WETH`, the contract considers the pool with more liquidity between Sushiswap and UniswapV2. However, if there is low liquidity in both the pools, it can still return incorrect price as it's costly to arbitrage low liquidity pools.

Recommendation

- Add pool manipulation protection by using TWAP oracle.
- Even with a TWAP oracle, consider the duration over which price average will be taken.
- The longer the duration, the resistant it is from manipulation, but the higher deviation against the actual price. So there is a trade off.
- Prefer using sufficiently decentralized Chainlink oracles over Uniswap oracle protect against these attacks and especially for low-liquidity pools.

[H-03] Signatures can be replayed

As a good practice, signatures should be used in a way that the message includes a nonce stored in the verifier contract. After verification, the nonce should be incremented so that the same signature cannot be used again.

Recommendation

Create a nonce variable in every contract where signature is verified. Increment it as soon as the current value is used.

[H-04] `depositToEntryPoint()` doesn't transfer the native token

`depositToEntryPoint()` calls `depositTo()` with `0` value:

```
_entryPoint.depositTo(address(this));
```

Recommendation

Update it to:

```
_entryPoint.depositTo{value: amount}(address(this));
```

[H-05] `WalletInit.commit()` can be called by anyone

`commit()` can be called by anyone to set `commit[commitKey]` to any value.

```
function commit(bytes32 commitKey, bytes32 hash) public {
        require(commits[commitKey].expirationTime < block.timestamp, "FW018");
        uint256 commitExpiration = block.timestamp + 10 minutes;
        commits[commitKey] = Commit(commitExpiration, hash);
        emit CommitHash(commitKey, hash, commitExpiration);
}
```

`commits` is used to validate salt in `validateSalt()` function and hence that functionality is now broken:

```
require(keccak256(abi.encode(seed, owner, initializerCallData)) ==
commits[commitKey].hash, "FW019");
```

`validateSalt()` transaction can be frontrun by an attacker who can set any value for the key in `commits` making this check pass or fail.

Recommendation

Determine who should have access to `commit()` and add proper access control.

[M-01] Validation contracts should consider native token amount

Validation contracts `RoleBasedAccessControl` and `RoleBasedAccessControl` ignore the native token value to be sent along the user operation while validating it:

```
(address target, , bytes memory data) = DataLib.getCallData(userOp.callData);
isValidAction(target, data, userOp.signature, bytes32(0));
```

For a user operation with attached value, like for transferring native token or to call a payable function, an attacker can send a higher or lower value.

Recommendation

Update `isValidAction()` to consider native token value, and update the above snippet to:

```
(address target, uint256 value, bytes memory data) =
DataLib.getCallData(userOp.callData);
isValidAction(target, value, data, userOp.signature, bytes32(0));
```

[L-01] `validateMerkleRoot()` can prove any node in the merkle tree

`validateMerkleRoot()` doesn't validate the proof length or doesn't hash the leaf. So `validateMerkleTree()` returns true for any intermediate node and its path to the root. However, the current invocations of this function are safe because the leaves are hashed before passing into this function.

Recommendation

Leave a comment on this function specifying this fact so that it's not called in unsafe contexts.

[G-01] `feeOracle` variable can be made internal or private

`feeOracle` can be fetched using `getFeeOracle()`.

Recommendation

Consider making `feeOracle` private or internal.

[G-02] try/catch in `_transferEthFromEntrypoint()` can be simplified

`_transferEthFromEntrypoint()` catches the revert in `_entryPoint` and then reverts itself with a specific error message.

```
try _entryPoint.withdrawTo(payable(recipient), amount) {} catch {
    revert("FW505");
}
```

This hides the reason with `_entryPoint` reverted and also costs more gas.

Recommendation

Update it as:

```
_entryPoint.withdrawTo(payable(recipient), amount);
```

When the call is successful, the behavior remains the same. On a revert, this also reverts while preserving the underlying revert reason. Additionally, this article is a good starting point to consider the unintended consequences of using try/catch, although it doesn't affect the current usages, mainly because the external calls enclosed are made to trusted addresses and the catches are generic.

[G-03] `UniswapPriceOracle` can avoid many external staticcalls

`UniswapPriceOracle._getLatestPrice()` makes 5 external calls for `WETH` and factory addresses. These variables are immutable in UniswapV2 and SushiswapV2 router, so they can be fetched and stored as immutable in the constructor.

Recommendation

Add a constructor and fetch `WETH` and factory address from `sushiswapRouterAddress` and `0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D`. For safety, you can revert if `WETH` addresses from both the routers don't match.

[G-04] zero slot manipulation can be avoided

Solidity ensures that the memory slot 0x60 is always `0` unless you assembly to manipulate it. `WalletModules.getPermitHash()` saves the value at the zero slot to restore it back. This is unnecessary as the value there is always `0`.

Recommendation

Apply this diff:

```
assembly {
    let temp1 := mload(0x40)
-       let temp2 := mload(0x60)

    mstore(0x0, _token)
    mstore(0x14, _to)
    mstore(0x28, amount)
    mstore(0x48, nonce)

    _hash := keccak256(0x0, 0x68)

    mstore(0x40, temp1)
-       mstore(0x60, temp2)
+       mstore(0x60,
0x0000000000000000000000000000000000000000000000000000000000000000)
}
```

[I-01] `_validateSignatureECDSA()` can use try/catch to avoid staticcall and assembly

This function relies of low level staticcall and assembly:

```
function _validateSignatureECDSA(bytes memory signature, bytes32 userOpHash,
bytes32 userId) internal view returns (uint256 sigTimeRange) {
    bytes memory call =
abi.encodeWithSelector(this.subValidateSignatureECDSA.selector, signature,
userOpHash, userId);
    (bool success, bytes memory returnData) = address(this).staticcall(call);
    if (success) {
        assembly {
            sigTimeRange := mload(add(returnData, 0x20))
        }
    } else {
        return 1;
    }
}
```

It can be simplified to avoid these low level constructions.

Recommendation

Consider using this instead:

```
function _validateSignatureECDSA(bytes memory signature, bytes32 userOpHash,
bytes32 userId) internal view returns (uint256 sigTimeRange) {
        try
UserAuthentication(address(this)).subValidateSignatureECDSA(signature,
userOpHash, userId) returns (uint256 _sigTimeRange) {
                return _sigTimeRange;
        } catch {
                return 1;
        }
}
```

[I-02] `getTokenValueOfEth()` incorrect Natspec

`tokenDecimals` in `getTokenValueOfEth(address aggregatorOrToken, uint256 ethAmount, uint8 tokenDecimals)` represents the decimal value in the target token. However, its natspec is defined as: `* @param tokenDecimals The number of decimals used by the price feed.`

Recommendation

Update it as: `* @param tokenDecimals The number of decimals used by the specified token.`

[I-03] Validations are not aggregated correctly

WalletValidation.sol#L140 combines the result of all the `authenticateUserOp()` calls through a boolean OR:

```
sigTimeRange |= IValidation(_validations[i]).authenticateUserOp(userOp,
userOpHash, "");
```

`sigTimeRange` follows the format as specified by EIP-4337:

> The return value MUST be packed of authorizer, validUntil and validAfter timestamps. - `authorizer` - 0 for valid signature, 1 to mark signature failure. Otherwise, an address of an authorizer contract. This ERC defines "signature aggregator" as authorizer.
>
> - `validUntil` is 6-byte timestamp value, or zero for "infinite". The UserOp is valid only up to this time.
> - `validAfter` is 6-byte timestamp. The UserOp is valid only after this time.

boolean OR doesn't preserve the correct values of `validUntil` and `validAfter`.

Recommendation

Ideally, it should take the max of `validAfter` and min of `validUntil`. However, the team clarified that currently these values are assumed to be `0`. Consider leaving a comment to convey why the OR operation is correct currently and when it should be updated.

[I-04] Outdated comment on `updateEntryPoint()`

`updateEntryPoint()` 's Natspec says:

```
/***
...
* @dev This function can only be called by the current entry point or owner of
the contract.
...
*/
function updateEntryPoint(IEntryPoint _newEntryPoint) external override {
        _requireFromEntryPoint();
        ...
}
```

It can only be called by the entrypoint itself because of `_requireFromEntryPoint()`. There is no owner of `FunWallet` in the traditional sense. The ownership is handled by validations which can have any custom logic.

Recommendation

Update the comment:

```
-* @dev This function can only be called by the current entry point or owner of
the contract.
+* @dev This function can only be called by the current entry point.
```