AvatarWallet

Smart Contract Audit Report

Document Info

Project Name	Avatar Wallet
Project Website	https://www.avatarwallet.io/
Audit Period	17.10.2022 - 24.09.2022
Audit Result	Passed
Report Prepared By	Zhang Yatao
Changelog	24.10.2022 - Draft Report

Audit Scope

Project Repo	https://github.com/*PRIVATEREPO*/solidity_jwt
Audit Commit	22ee06f51681eed1a1019d3c515f7676ad257c9b

```
├─jwt
│   Base64.sol
│   Identity.sol
│   JWKS.sol
│   SolRsaVerify.sol
│   Strings.sol
│
├─proxy
│  │ CustomTransparentUpgradeableProxy.sol
│  │ CustomUUPSUpgradeable.sol
│  │ factory.sol
│  │
│  └─ERC1967
│       CustomERC1967Proxy.sol
│
├─testkit
│   GldNft.sol
│   GldToken.sol
│   TestUserWallet.sol
│
```

```
└─wallets
  │ Estimate.sol
  │ JwtAuthProxy.sol
  │ ManagementWallet.sol
  │ TestTransfer.sol
  │ UserWallet.sol
  │ UserWalletImplV1.sol
  │ UserWalletProxy.sol
  │
  └─interface
      IJwtAuth.sol
      IUserWallet.sol
```

Summary

Avatar Wallet is targeting to provide a seamless experience into the Metaverse. It is a decentralized and non-custody smart contract based wallet. The team implemented a novel algorithm to make sure only the user can touch the asset.

After our inspection and auditing according to industry standards, we found NO CRITICAL, HIGH risk problems. There are one MEDIUM risk and multiple LOW risk issues that need special attention in future operations. Please refer to the summary below and see the Audit Details for implementation details.

#	Finding	Risk Level	Comments from DigiFT
R1	Uncheck “audience” in JWT which can cause phishing problems.	MEDIUM	Solved. The “aud” parameter will be checked.
R2	Leakage of management private keys may lead to signature forge .	LOW	Risk accepted. The procedure of using a management wallet will be reinforced. And the transaction of updating the trust public key will be monitored.
R3	Implementation of base64 encoding is different from standard implementation .	LOW	Risk accepted. This is a work around of an internal bug. And the usage of base64 encoding won’t have an impact on security control.

Contract based wallet is the state of the art solution on keyless wallet. There are many challenges in implementing a set of safe contracts to provide such functions. Avatar Wallet did a great job on designing the architecture and balancing the security and gas efficiency. Here are some highlights in the implementation.

#	Finding
G1	There is no special wallet address that can interfere with a user's asset. The asset is transferred if and only if the user has sent their valid signature.
G2	User Wallet adopts the UUPS Proxy Upgrade approach, thus there is no privileged wallet address existing in the user's wallet contract. Even the upgrade request also needs to be signed by the user.
G3	The root of trust is provided by a third party rather than Avatar Wallet. And by the design, the root of trust and wallet service provider is plugable. As users have the choice to upgrade their wallet contract to change to wallet service provider and the root of trust.

Audit Detail

ID:	R1	File:	Identity.sol	Risk:	Medium
Finding:	Uncheck “audience” in JWT which can cause phishing problems.				

//Identity.sol: Line 51

```
  function verifyAndParseJwt(IJwtAuth.JWT memory _jwt)
      external
      view
      override
      returns (
          uint256,
          bytes memory,
          bytes memory,
          bytes memory,
          uint256
      )
   {
      (
          bytes memory aud,
          bytes memory email,
          bytes memory signature,
          bytes memory jti,
          uint256 expiration
      ) = parseOptimized(_jwt.payloadJson);

      require(
          keccak256(abi.encodePacked(_jwt.alg)) == keccak256(abi.encodePacked("RS256")),
          "Google must use RS256 algo"
      );
```

//ElessarLabs: This check is newly added. Make sure Google’s returning audience information is consistent with our prestore value.

```
      require(audiences[aud], "Audience not allowed");

      uint256 result = verifySignature(
          _jwt.kid,
          string(abi.encodePacked(_jwt.headerBase64, ".", _jwt.payloadJson.encode())),
          _jwt.signature
      );
      return (result, email, signature, jti, expiration);
   }
```

ID:	R2	File:	JWKS.sol	Risk:	Low
Finding:	Leakage of management private keys may lead to signature forge .				

//JWKS.sol: Line 34

```
  function addKey(
      string memory kid,
      bytes memory modulus,
      bytes memory exponent
   ) public {
```

//ElessarLabs: The keys and exponents consist of the public key to verify the trust party's signature. In case the admin or editor's wallet private leaked. There exists a possibility that a forged JWT request can pass the signature verification.

```
      require(msg.sender == admin || editors[msg.sender], "Only admin or editor can add key");
      keys[kid] = modulus;
      exponents[kid] = exponent;
      emit KeyAdded(kid, modulus, exponent);
   }
```

ID:	R3	File:	Base64.sol	Risk:	Low
Finding:	Implementation of base64 encoding is different from standard implementation.				

//Base64.sol: Line 7

```
  function encode(string memory _str) internal pure returns (string memory) {
    bytes memory data = bytes(_str);
    if (data.length == 0) return "";

    // load the table into memory
    string memory table = TABLE_ENCODE;
```

//ElessarLabs: The calculation of encoded length and padding function differ from solidity's reference implementation. Though this function is used in a non security critical area.

```
    // multiply by 4/3 rounded up
    uint256 encodedLen = 4 * ((data.length) / 3);

    if (data.length % 3 != 0) {
      encodedLen += ((data.length) % 3) + 1;
    }

    // add some extra buffer at the end required for the writing
    string memory result = new string(encodedLen + 32);
```

…..

```
       // padding with '='
      // switch mod(mload(data), 3)
      // case 1 { mstore(resultPtr, sub(mload(resultPtr), 1)) }
      // case 2 { mstore(resultPtr, sub(mload(resultPtr), 2)) }
      // case 1 { mstore(sub(resultPtr, 2), shl(240, 0x3d3d)) }
      // case 2 { mstore(sub(resultPtr, 1), shl(248, 0x3d)) }
    }

    return result;
  }
```

ID:	G1	File:	UserWalletProxy.sol	Risk:	N/A
Finding:	No special wallet address that can interfere with a user's asset				

//UserWalletProxy: Line 28

```
    function proxyInitialize(
        address logic_,
        address admin_,
        bytes memory data_
    ) external onlyAdmin {
        require(
```

//ElessarLabs: This only function call within the user wallet's contract that needs to be invoked by the admin. And this function can only be invoked once during initialization.

```
            StorageSlot.getBooleanSlot(_INITIALIZED_SLOT).value == false,
            "proxy already initialized"
        );
        assert(
            _IMPLEMENTATION_SLOT ==
                bytes32(uint256(keccak256("eip1967.proxy.implementation")) - 1)
        );

        _upgradeToAndCall(logic_, data_, false);
        require(_getAdmin() != address(0), "admin can't be address(0)");
        _changeAdmin(admin_);
        StorageSlot.getBooleanSlot(_INITIALIZED_SLOT).value = true;
    }
```

ID:	G2	File:	UserWalletImplV1.sol	Risk:	N/A
Finding:	User Wallet adopts the UUPS Proxy Upgrade approach. Users can initial contract upgrade on their own.				

```
//UserWalletImplV1.sol: Line 35

function _authorizeUpgrade(
        address _addr,
        address _refundAddress,
        address _refundToken,
        uint256 _refundAmount,
        bytes memory _jwt
    ) internal override {
        address[] memory _to = new address[](1);
        _to[0] = _addr;

        bytes[] memory _data = new bytes[](1);
        _data[0] = "";

        uint256[] memory _values = new uint256[](1);
        _values[0] = 0;

        bytes32 nonce = this.hashOrder(_to, _data, _values, _refundAddress, _refundToken,
_refundAmount);
        safeRefund(_refundAddress, _refundToken, _refundAmount);

        IJwtAuth.JWT memory jwt = abi.decode(_jwt, (IJwtAuth.JWT));

//ElessarLabs: This function only approves the user's invocation. And the user’s
signature will be checked.
        require(verifyJwt(jwt, bytes32(nonce)), "Invalid JWT, cannot upgrade the contract");
    }
```

ID:	G3	File:	BaseERC20.Sol	Risk:	N/A
Finding:	The root of trust is provided by a third party rather than Avatar Wallet. And by the design, the root of trust and wallet service provider is plugable.				

```
//UserWalletImplV1.sol:16
  function initialize(string memory _userId, address _jwtProxy) external initializer onlyProxy
{
      require(msg.sender == _getAdmin(), "Only admin can initialize");
      version = "1.0.0";
      bytes32 typeHash = keccak256(
         "EIP712Domain(string name,string version,uint256 chainId,address
verifyingContract)"
      );
      bytes32 hashedName = keccak256(bytes("UserWallet"));
      bytes32 hashedVersion = keccak256(bytes(version));
      _CACHED_CHAIN_ID = block.chainid;
      _CACHED_DOMAIN_SEPARATOR = _buildDomainSeparator(typeHash, hashedName,
hashedVersion);

      _TYPE_HASH = typeHash;
      _HASHED_VERSION = hashedVersion;
      _HASHED_NAME = hashedName;

      userId = bytes(_userId);

//ElessarLabs: This jwtProxy is the smart contract that provides verification. It can be
plugged in any Oauth alike service provider to provide the root of trust. By default, it is
using Google's Oauth service.
      jwtProxy = _jwtProxy;
   }
```