OKX AA钱包原理分析

OKX AA钱包使用流程

下载 OKX App ， 前往 Web3 钱包- 钱包管理- 添加账户 中，添加智能合约账户

- 注：OKX在帮用户生成AA钱包，只是根据“私钥钱包地址”计算出了对应的“AA钱包地址”。只有当用户使用该AA钱包发起第一笔交易时，OKX会帮用户在该“AA钱包地址”上部署合约，示例如下：

1. 准备一个私钥钱包：0xeba81661003435B5Fa9a35358F334F9C58912ce2

2. 在OKX App上添加AA钱包，计算得到AA地址：0xd16919FfbB44a442B982b0734bd44A6966642BD0

3. 【第一次】使用AA钱包进行转账：
- OKX官方地址 首先会该地址上部署合约：
https://polygonscan.com/tx/0x6eebd58b436b06c358ffb53984f191867ba7c2680b28e138297ab718108e9148

- OKX官方地址 调用AA钱包，执行转账操作：
https://polygonscan.com/tx/0x6eebd58b436b06c358ffb53984f191867ba7c2680b28e138297ab718108e9148

4. 补充：以上操作是在polygon链执行的，所以暂时只有Polygon链部署了AA钱包合约（其他链流程相同）

技术分析（以polygon链为例）

- EntryPoint合约（用户交互的入口）：
https://polygonscan.com/address/0xdc5319815cdaac2d113f7f275bc893ed7d9ca469#contracts

- SmartAccountProxyFactory（用于创建AA钱包Proxy合约，被EntryPoint调用）：https://polygonscan.com/address/0x81e11c4701c5189b0122ef42daf1ff3d453d968e#code
- SmartAccountProxy示例（即用户的AA钱包，本质上是一个proxy合约）：https://polygonscan.com/address/0xd16919ffbb44a442b982b0734bd44a6966642bd0#code
- SmartAccount（各个AA钱包proxy指向的Impl合约--Singleton）：https://polygonscan.com/address/0x3dbeb76d9d9444d7db9dcf3799e17acd247f8fac#code

AA钱包地址的生成

```
AA钱包合约是通过create2()部署的，因此其部署地址计算规则同create2():
AA钱包地址= keccak256( 0xFF+ SmartAccountProxyFactory合约地址 + salt ++ keccak256(SmartAccountProxy合约的createion code))[12:]
```

- 上述公式中，salt 由用户私钥钱包地址及一个随机数来决定（参考下方getAddress()中的salt2）。通过修改salt，同一私钥钱包可以生成多个AA钱包
- 因此，该地址是可以预计算出来的

代码中的计算流程

- 具体流程可参考SmartAccountProxyFactory合约的getAddress()，如下图所示，
  - 参数_safeSingleton为SmartAccount合约地址
  - 参数initializer为"0x36b14535000000000000000000000000+用户的私钥钱包地址"，0x36b14535表示的是 Initialize(address)方法的selector（以上述钱包地址为例，initializer就是"0x36b14535000000000000000000000000eba81661003435b5fa9a35358f334f9c58912ce2"）--每个用户对应唯一的一个initializer

- 参数slat：生成AA钱包地址时用的随机数

```
     */
    function getAddress(
        address _safeSingleton,
        bytes memory initializer,
        uint256 salt
    ) public view returns (address) {
        //copied from deployProxyWithNonce
        bytes32 salt2 = keccak256(
            abi.encodePacked(keccak256(initializer), salt)
        );
        bytes memory deploymentData = abi.encodePacked(
            type(SmartAccountProxy).creationCode
        );
        return
            Create2.computeAddress(
                bytes32(salt2),
                keccak256(deploymentData),
                address(this)
            );
    }
}
```

- 下图中computeAddress()的流程，同create2操作符

```
    */
    function computeAddress(
        bytes32 salt,
        bytes32 bytecodeHash,
        address deployer
    ) internal pure returns (address addr) {
        /// @solidity memory-safe-assembly
        assembly {
            let ptr := mload(0x40) // Get free memory pointer

            // |                   | ↓ ptr ...  ↓ ptr + 0x0B (start) ...  ↓ ptr + 0x20 ...  ↓ ptr + 0x40 ...   |
            // |-------------------|---------------------------------------------------------------------------|
            // | bytecodeHash      |                                                        CCCCCCCCCCCC...CC |
            // | salt              |                                      BBBBBBBBBBBB...BB                    |
            // | deployer          | 000000...0000AAAAAAAAAAAAAAAAAA...AA                                      |
            // | 0xFF              |            FF                                                             |
            // |-------------------|---------------------------------------------------------------------------|
            // | memory            | 000000...00FFAAAAAAAAAAAAAAAAAA...AABBBBBBBBBBBB...BBCCCCCCCCCCCC...CC |
            // | keccak(start, 85) |            ↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑ |

            mstore(add(ptr, 0x40), bytecodeHash)
            mstore(add(ptr, 0x20), salt)
            mstore(ptr, deployer) // Right-aligned with 12 preceding garbage bytes
            let start := add(ptr, 0x0b) // The hashed data starts at the final garbage byte which we will set to 0xff
            mstore8(start, 0xff)
            addr := keccak256(start, 85)
        }
    }
```

AA钱包的部署

- 由OKX 官方账号发起交易
- 具体流程可参考SmartAccountProxyFactory合约的 `deployProxyWithNonce`，如下图所示

- 只需要保证参数相同，就可以将合约部署到之前计算出的AA钱包地址上

```
    /// @param saltNonce Nonce that will be used to generate the salt to calcu
    function deployProxyWithNonce(
        address _singleton,
        bytes memory initializer,
        uint256 saltNonce
    ) internal returns (SmartAccountProxy proxy) {
        // If the initializer changes the proxy address should change too. Has
        bytes32 salt = keccak256(
            abi.encodePacked(keccak256(initializer), saltNonce)
        );
        bytes memory deploymentData = abi.encodePacked(
            type(SmartAccountProxy).creationCode
        );
        // solhint-disable-next-line no-inline-assembly
        assembly {
            proxy := create2(
                0x0,
                add(0x20, deploymentData),
                mload(deploymentData),
                salt
            )
        }
        require(address(proxy) != address(0), "Create2 call failed");
        walletWhiteList[address(proxy)] = true;
    }

```

AA钱包的使用

1. 方式一：用户通过OKX钱包使用AA钱包时，OKX会帮用户组装交易数据、发起交易，进而调用AA钱包合约的。
2. 用户只需要对交易内容签名、不需要主动发起交易，因此也不需要直接支付gas费（准确来说，是OKX会直接从AA钱包中扣除相应金额的token作为gas费）
3. 方式二(不可行）：用户自行调用AA钱包合约（即SmartAccountProxy）中的各个方法，用户虽然是AA钱包的owner，但实际基本无法直接调用该AA钱包，主要原因是，通过分析Smart Account合约中的各个write方法可知：
4. enableModule() disableModule() updateImplement() execTransactionRevertOnFail() 都使用了authorized修饰符，使得它们只能被execTransactionXXXX()间接调用
5. 另外，如下方的原理分析所示，execTransactionXXXX()中有权限校验，仅允许OKX 官方白名单地址通过EntryPoint合约来调用
6. 因此，要想调用各个write方法，入口都是由OKX官方控制的

```
contract SelfAuthorized {
    function requireSelfCall() private view {
        require(msg.sender == address(this), "GS031");
    }

    modifier authorized() {
        // 保证函数只能被自己调用---》即只能通过execTransactionXXX方法来间接调用，从而保证安全（多用于多签钱包）
```

```
        requireSelfCall();
        _;
    }
}
```

原理分析

- 在SmartAccount合约中，提供了多个`execTransactionFromXXX`用于帮助执行各种操作，如下图所示

```
78
79      function execTransactionFromEntrypoint(
80          address to,
81          uint256 value,
82          bytes calldata data
83      ) public onlyEntryPoint {
84          executeWithGuard(to, value, data);
85      }
86
87      function execTransactionFromEntrypointBatch(
88          ExecuteParams[] calldata _params
89      ) external onlyEntryPoint {
90          executeWithGuardBatch(_params);
91      }
92
93      function execTransactionFromEntrypointBatchRevertOnFail(
94          ExecuteParams[] calldata _params
95      ) external onlyEntryPoint {
96          execTransactionBatchRevertOnFail(_params);
97      }
98
99      function execTransactionFromModule(
100         address to,
101         uint256 value,
102         bytes calldata data,
103         Enum.Operation operation
104     ) public override {
105         IStorage(EntryPoint).validateModuleWhitelist(msg.sender);
106
107         if (operation == Enum.Operation.Call) {…
109         } else {…
115         }
116     }
117 }
```

- 为了防止未经授权的用户调用钱包，合约中各方法会首先对msg.sender进行校验，只有EntryPoint合约自身或者EntryPoint合约中设置的白名单地址才可以（如上图红框所示，即使用户自己也是无法调用的）
- 另外，默认情况下，用户通过EntryPoint合约操作AA钱包时，调用的是handleOps()，如下图所示，只有OKX官方的白地址才可调用，进一步保障了钱包的安全（调用过程中也会校验用户签名）

```
    }

    function handleOps(
        UserOperation[] calldata ops,
        address payable beneficiary
    ) public override(EntryPoint0_4, IEntryPoint) {
        uint256 opslen = ops.length;

        if (!officialBundlerWhiteList[msg.sender]) {
            require(
                unrestrictedBundler && msg.sender == tx.origin,
                "called by illegal bundler"
            );
            require(opslen == 1, "only support one op");
        }

        UserOpInfo[] memory opInfos = new UserOpInfo[](opslen);
        uint256 collected;
        unchecked {
            for (uint256 i = 0; i < opslen; ++i) {
                try this.handleOp(i, ops[i], opInfos[i], address(0)) returns (
                    uint256 gasUsed
                ) {
                    collected += gasUsed;
                } catch (bytes memory revertReason) {
                    emit HandleUserOpRevertReason(
                        ops[i].sender,
                        ops[i].nonce,
                        revertReason
                    );
                }
            }
            _compensate(beneficiary, collected);
        }
    }
```

仅官方白名单地址可调用

补充：OKX 无私钥钱包（MPC钱包）

- OKX无私钥钱包的核心为MPC技术。无私钥钱包创建时私钥碎片被分成了 3 片，分别存储在 OKX 服务器，用户设备和 iCloud 或 Google Drive。交易签名时使用其中 2 份私钥碎片，即可计算出完整签名并使用钱包
- OKX 无私钥钱包的算法的源码：https://github.com/okx/threshold-lib （私钥碎片的计算、存储等流程）