SMART CONTRACT AUDIT REPORT

for

INSTADAPP LABS

Prepared By: Shuxiao Wang

Hangzhou, China
Mar. 18, 2020

Document Properties

Client	InstaDApp Labs
Title	Smart Contract Audit Report
Target	InstaDApp Smart Accounts
Version	1.0
Author	Huaguo Shi, Chiachih Wu
Auditors	Huaguo Shi, Chiachih Wu
Reviewed by	Chiachih Wu
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0	Mar. 18, 2020	Huaguo Shi, Chiachih Wu	Final Release
1.0-rc2	Mar. 16, 2020	Huaguo Shi, Chiachih Wu	Status Update, Minor Findings Added
1.0-rc1	Mar. 10, 2020	Huaguo Shi, Chiachih Wu	Status Update
0.2	Mar. 05, 2020	Huaguo Shi, Chiachih Wu	Status Update
0.1	Mar. 03, 2020	Huaguo Shi	Initial Draft

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the **InstaDApp Smart Accounts** Smart Accounts design document and related smart contract source code, we in the report outline our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About InstaDApp Smart Accounts

InstaDApp is a DeFi portal that aggregates the major protocols using a smart wallet layer and bridge contracts, making it easy for users to make the best decisions about assets and execute previously complex transactions seamlessly.

The basic information of InstaDApp Smart Accounts is as follows:

Table 1.1: Basic Information of InstaDApp Smart Accounts

Item	Description
Issuer	InstaDApp Labs
Website	https://instadapp.io/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	Mar. 18, 2020

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit:

- https://github.com/InstaDApp/dsa-contracts (180509b)
- https://github.com/InstaDApp/sa-contracts/tree/peckshield-edits (ae050f2)

- https://github.com/InstaDApp/sa-contracts/tree/peckshield-edits (83902b7)
- https://github.com/InstaDApp/sa-contracts (bb76c0e)

1.2 About PeckShield

PeckShield Inc. [25] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [20]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [19], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this audit does not give any warranties on finding all possible security issues of the given smart contract(s), i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as an investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the InstaDApp Smart Accounts implementation. During the first phase of our audit, we studied the smart contract source code and ran our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	2	■ ■
Low	3	■ ■ ■
Informational	4	■ ■ ■ ■
Total	10	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 2 medium-severity vulnerability, 3 low-severity vulnerabilities, and 4 informational recommendations.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Missing Address Validation in changeMaster()	Business Logics	Resolved
PVE-002	Low	Missing Ether Amount Checks in deposit()	Data Processing Issues	Resolved
PVE-003	Medium	Flawed Upgrade Logic in InstaIndex	Business Logics	Resolved
PVE-004	High	Unprotected Initialization Interface in InstaIndex	Initialization and Cleanup	Confirmed
PVE-005	Info.	Possible Data Pollution in Deposit/Withdraw	Coding Practices	Confirmed
PVE-006	Info.	Missing Validation to the Origin While Building Smart Accounts	Coding Practices	Confirmed
PVE-007	Low	Missing Array Length Checks in InstaAccount	Data Processing Issues	Resolved
PVE-008	Info.	Flawed Linked List Implementations	Data Processing Issues	Resolved
PVE-009	Low	Missing Disable Function in staticConnectors	Behavioral Issues	Confirmed
PVE-010	Info.	Gas Optimization	Resource Management	Confirmed

Please refer to Section 3 for details.

3 | Detailed Results

3.1 Missing Address Validation in changeMaster()

- ID: PVE-001
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `contracts/registry/index.sol`
- Category: Business Logics[15]
- CWE subcategory: CWE-754 [10]

Description

InstaDApp Smart Accounts has a very important management authority, `master`, which can be used to add new accounts and change the check address. The `InstaIndex` contract provides the `changeMaster()` function to allow the current `master` to modify the privileged address to a new address.

```
48     /**
49      * @dev Change the Master Address.
50      * @param _newMaster New Master Address.
51      */
52     function changeMaster(address _newMaster) external isMaster {
53         require(_newMaster != master, "already-a-master");
54         require(_newMaster != address(0), "not-valid-address");
55         master = _newMaster;
56         emit LogNewMaster(_newMaster);
57     }
```

Listing 3.1: contracts/registry/index.sol

As shown in the above code snippets, `_newMaster` is validated against the current `master` and the zero address in line 53-54. However, if you enter a wrong address by mistake, you will never be able to take the management permissions back.

Recommendation The transition should be managed by the implementation with a two-step approach: `changeMaster()` and `updateMaster()`. Specifically, the `changeMaster()` function keeps the new address in the storage `newMaster` instead of modifying the `master` directly. The `updateMaster()`

function checks whether `newMaster` is the `msg.sender`, which means `newMaster` signs the transaction and verifies itself as the new `master`. After that, `master` could be replaced by `newMaster`. This had been addressed in the patched `contracts/registry/index.sol`.

```
    /**
     * @dev Change the Master Address.
     * @param _newMaster New Master Address.
     */
    function changeMaster(address _newMaster) external isMaster {
        require(_newMaster != master, "already-a-master");
        require(_newMaster != address(0), "not-valid-address");
        require(newMaster != _newMaster, "already-a-new-master");
        newMaster = _newMaster;
        emit LogNewMaster(newMaster);
    }

    /**
     * @dev update the Master Address.
     */
    function updateMaster() external {
        require(newMaster != address(0), "not-valid-address");
        require(msg.sender == newMaster, "not-master");
        master = newMaster;
        newMaster = address(0);
        emit LogUpdateMaster(master);
    }
```

Listing 3.2: contracts/registry/index.sol

3.2 Missing Ether Amount Checks in deposit()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `contracts/connectors/basic.sol`
- Category: Data Processing Issues [16]
- CWE subcategory: CWE-229 [5]

Description

The function `deposit()` only processes the ERC-20 assets but does not perform the necessary checks on ether. Specifically, if the `tokenAmt` is greater than `msg.value`, the book-keeping amount could be greater than the actual ether amount deposited into the `InstaAccount` instance.

```
        /**
        * @dev Deposit Assets To Smart Account.
        * @param erc20 Token Address.
        * @param tokenAmt Token Amount.
```

```
         * @param getId Get Storage ID.
         * @param setId Set Storage ID.
         */
        function deposit(address erc20, uint tokenAmt, uint getId, uint setId) public payable {
            uint amt = getUint(getId, tokenAmt);
            if (erc20 != getEthAddr()) {
                ERC20Interface token = ERC20Interface(erc20);
                amt = amt == uint(-1) ? token.balanceOf(msg.sender) : amt;
                token.transferFrom(msg.sender, address(this), amt);
            }
            setUint(setId, amt);
            emit LogDeposit(erc20, amt, getId, setId);
        }
```

Listing 3.3: contracts/connectors/basic.sol

Recommendation Check `msg.value` against `amt`. This had been addressed in the patched `contracts/connectors/basic.sol`.

```
        /**
         * @dev Deposit Assets To Smart Account.
         * @param erc20 Token Address.
         * @param tokenAmt Token Amount.
         * @param getId Get Storage ID.
         * @param setId Set Storage ID.
         */
        function deposit(address erc20, uint tokenAmt, uint getId, uint setId) public payable {
            uint amt = getUint(getId, tokenAmt);
            if (erc20 != getEthAddr()) {
                ERC20Interface token = ERC20Interface(erc20);
                amt = amt == uint(-1) ? token.balanceOf(msg.sender) : amt;
                token.transferFrom(msg.sender, address(this), amt);
            } else {
                require(msg.value == amt, "invalid-ether-amount");
             }
            setUint(setId, amt);
            emit LogDeposit(erc20, amt, getId, setId);
        }
```

Listing 3.4: contracts/connectors/basic.sol

3.3 Flawed Upgrade Logic in InstaIndex

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `contracts/registry/index.sol`
- Category:Business Logics [15]
- CWE subcategory: CWE-841 [11]

Description

The `addNewAccount()` function in `InstaIndex` is used to add new `InstaAccount` implementations along with the corresponding `InstaConnectors` and `check` contracts. However, the current `addNewAccount()` implementation has some flaws such that the `master's` incautious calls may destroy the whole system. Specifically, the `_newAccount` is not validated except checking for zero addresses. At least, the important variable, `version`, should be validated before adding it into the registry.

```
    function addNewAccount(address _newAccount, address _connectors, address _check)
        external isMaster {
        require(_newAccount != address(0), "not-valid-address");
        versionCount++;
        account[versionCount] = _newAccount;
        if (_connectors != address(0)) connectors[versionCount] = _connectors;
        if (_check != address(0)) check[versionCount] = _check;
        emit LogNewAccount(_newAccount, _connectors, _check);
    }
```

Listing 3.5: contracts/registry/index.sol

As shown in the code snippets, `version` is used to index the `connectors` mapping (line 136 and line 138) in `InstaIndex`. Also, the `check` mapping is indexed by the `version` (line 143).

```
124     function cast(
125         address[] calldata _targets,
126         bytes[] calldata _datas,
127         address _origin
128     )
129     external
130     payable
131     {
132         require(isAuth(msg.sender)  msg.sender == instaIndex, "permission-denied");
133         IndexInterface indexContract = IndexInterface(instaIndex);
134         bool isShield = shield;
135         if (!isShield) {
136             require(ConnectorsInterface(indexContract.connectors(version)).isConnector(
                    _targets), "not-connector");
137         } else {
138             require(ConnectorsInterface(indexContract.connectors(version)).
                    isStaticConnector(_targets), "not-static-connector");
```

```
        }
        for (uint i = 0; i < _targets.length; i++) {
            spell(_targets[i], _datas[i]);
        }
        address _check = indexContract.check(version);
```

Listing 3.6: contracts/account.sol

What if the `master` does not `addNewAccount()` by the order of different `InstaAccount` contracts' version? What if the author of a new `InstaAccount` contract set wrong version number for the latest implementation? The `cast()` may have abnormal behaviors such as using v1 connectors in v2 `InstaAccount`.

Recommendation Check the `version` of `_newAccount` against `versionCount`. Also, we should ensure `versionCount > 0` when we set `master` in the constructor (mentioned here). Otherwise, a new `InstaAccount` could be added by `addNewAccount()` without the essential initialization process. This had been addressed in the patched `contracts/registry/index.sol`.

```
    function addNewAccount(address _newAccount, address _connectors, address _check)
        external isMaster {
        require(versionCount > 0, "not-init-yet");
        require(_newAccount != address(0), "not-valid-address");
        require((versionCount+1) == AccountInterface(_newAccount).version, "not-valid-
            account-version");
        versionCount++;
        account[versionCount] = _newAccount;
        if (_connectors != address(0)) connectors[versionCount] = _connectors;
        if (_check != address(0)) check[versionCount] = _check;
        emit LogNewAccount(_newAccount, _connectors, _check);
    }
```

Listing 3.7: contracts/registry/index.sol

3.4 Unprotected Privileged Interface in InstaIndex

- ID: PVE-004
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `contracts/registry/index.sol`
- Category:Initialization and Cleanup [17]
- CWE subcategory: CWE-454 [8]

Description

The `setBasics()` function in `InstaIndex` contract is designed to be called once to initialize the first version of `InstaAccount` along with the `InstaList` and `InstaConnectors` contracts. However, there's no restriction to enter such as privileged function, which results in possible DoS attacks. Specifically, a malicious actor could send out front-running transactions calling the `setBasics()` function with an alternative `_master` whenever a legit `setBasics()` is identified in the tx pool. This enables bad actors to take-over any `InstaIndex` contract right after the deployment.

```
    function setBasics(
        address _master,
        address _list,
        address _account,
        address _connectors
    ) external {
        require(
            master == address(0) &&
            list == address(0) &&
            account[1] == address(0) &&
            connectors[1] == address(0) &&
            versionCount == 0,
            "already-defined"
        );
        master = _master;
        list = _list;
        versionCount++;
        account[versionCount] = _account;
        connectors[versionCount] = _connectors;
    }
```

Listing 3.8: contracts/registry/index.sol

Recommendation The `master` should be set as `msg.sender` in the constructor. With that, the `setBasics()` could be protected by `isMaster()` modifier to prevent it from being abused.

```
    constructor () public {
        master = msg.sender;
    }

    function setBasics(
        address _master,
        address _list,
        address _account,
        address _connectors
    ) external isMaster {
        require(
            list == address(0) &&
            account[1] == address(0) &&
            connectors[1] == address(0) &&
            versionCount == 0,
            "already-defined"
        );
        master = _master;
        list = _list;
        versionCount++;
        account[versionCount] = _account;
        connectors[versionCount] = _connectors;
```

```
    }
```

Listing 3.9: Revised contracts/registry/index.sol

As per discussion with InstaDApp Labs, the design of InstaDApp Smart Accounts allows anyone to deploy the `InstaIndex` contract (i.e., the deployer is not necessarily the `master`), which makes InstaDApp Smart Accounts an open framework for users and developers. Also, when the `InstaIndex` contract is deployed, the deployer might not have the knowledge of who gonna be the future `master`. On the other hand, the bad actor cannot make any benefit from launching the DoS attacks. Based on that, we provide some optimization suggestions to raise the cost of attacking `InstaIndex` through the `setBasics()` function. First, we can consider add some code to burn more gas in `setBasics()`.

```
    function setBasics(
        address _master,
        address _list,
        address _account,
        address _connectors
    ) external {
        require(
            master == address(0) &&
            list == address(0) &&
            account[1] == address(0) &&
            connectors[1] == address(0) &&
            versionCount == 0,
            "already-defined"
        );
        master = _master;
        list = _list;
        versionCount++;
        account[versionCount] = _account;
        connectors[versionCount] = _connectors;

        /* Burn gas to prevent DoS attack */
        for (uint256 i = 0; i < 22231; i++) {}
    }
```

Listing 3.10: contracts/registry/index.sol

Moreover, we can consider obfuscating the logic of `setBasics()`, which makes the attacker fail to identify the entry to `setBasics()` function. However, this simply violates the open source convention of Ethereum smart contracts.

3.5 Possible Data Pollution in Deposit/Withdraw

- ID: PVE-005
- Severity: Informational
- Likelihood: None
- Impact: None
- Target: `contracts/account.sol`
- Category:Coding Practices [13]
- CWE subcategory: CWE-621 [9]

Description

In the `InstaAccount` contract, the `cast()` function is designed to be used to call multiple functions in multiple connectors in one transaction. The `deposit()` and `withdraw()` functions are implemented as the basic connector since they are the two most common operations.

```
67      * @dev Deposit Assets To Smart Account.
68      * @param erc20 Token Address.
69      * @param tokenAmt Token Amount.
70      * @param getId Get Storage ID.
71      * @param setId Set Storage ID.
72      */
73     function deposit(address erc20, uint tokenAmt, uint getId, uint setId) public payable
        {
74         uint amt = getUint(getId, tokenAmt);
75         if (erc20 != getEthAddr()) {
76             ERC20Interface token = ERC20Interface(erc20);
77             amt = amt == uint(-1) ? token.balanceOf(msg.sender) : amt;
78             token.transferFrom(msg.sender, address(this), amt);
79         }
80         setUint(setId, amt);
81         emit LogDeposit(erc20, amt, getId, setId);
82     }
```

Listing 3.11: contracts/connectors/basic.sol

As shown in the above code snippets, after `transferFrom()`, the `setUint()` function is called to store the `amt` into the memory using `setId` as the index (line 81).

```
    /**
    * @dev Withdraw Assets To Smart Account.
    * @param erc20 Token Address.
    * @param tokenAmt Token Amount.
    * @param to Withdraw token address.
    * @param getId Get Storage ID.
    * @param setId Set Storage ID.
    */
    function withdraw(
        address erc20,
        uint tokenAmt,
        address payable to,
```

```
        uint getId,
        uint setId
    ) public payable {
        require(AccountInterface(address(this)).isAuth(to), "invalid-to-address");
        uint amt = getUint(getId, tokenAmt);
        if (erc20 == getEthAddr()) {
            amt = amt == uint(-1) ? address(this).balance : amt;
            to.transfer(amt);
        } else {
            ERC20Interface token = ERC20Interface(erc20);
            amt = amt == uint(-1) ? token.balanceOf(address(this)) : amt;
            token.transfer(to, amt);
        }
        setUint(setId, amt);
        emit LogWithdraw(erc20, amt, to, getId, setId);
    }
```

Listing 3.12: contracts/connectors/basic.sol

On the other hand, the `withdraw()` function retrieves the amount by calling the `getUint()` function with `getId` as the index. After that, the `amt` of assets are transferred. Here, we show an example of data pollution. If someone `deposit()` 10 tokens twice, she should have 20 tokens to `withdraw()`. However, in the current implementation, only 10 tokens would be book-kept in the memory (i.e., the second `setUint()` will overwrite the first `setUint()`.

Recommendation As per discussion with InstaDApp Labs, the `setUint()/getUint()` are the infrastructure provided to the developers. It means the developers need to implement their own ways to perform the operations such as `deposit()` and `withdraw()`. We recommend that the use cases of temporary memory should be clearly explained in the documentation. Especially, remind developers that, as we mention above, the potential risks of calling `deposit()` twice.

3.6 Missing Validation to the Origin While Building Smart Accounts

- ID: PVE-006
- Severity: Informational
- Likelihood: None
- Impact: None
- Target: `contracts/registry/index.sol`
- Category:Coding Practices [13]
- CWE subcategory: CWE-1041 [3]

Description

In the `InstaIndex` contract, the public function, `build()`, allows arbitrary users to create a smart `account` for a specific `_owner`. While building the `account`, the caller can specify the `accountVersion` and `_origin`, which indicates what version of `InstaAccount` contract to be used and where the account is created from. However, throughout the `build()` function, there's no logic to validate the `_origin`, leading to fabricated data being logged on the blockchain (line 167).

```
152     /**
153      * @dev Create a new Smart Account for a user.
154      * @param _owner Owner of the Smart Account.
155      * @param accountVersion Account Module version.
156      * @param _origin Where Smart Account is created.
157      */
158     function build(
159         address _owner,
160         uint accountVersion,
161         address _origin
162     ) public returns (address _account) {
163         require(accountVersion != 0 && accountVersion <= versionCount, "not-valid-account");
164         _account = createClone(accountVersion);
165         ListInterface(list).init(_account);
166         AccountInterface(_account).enable(_owner);
167         emit AccountCreated(msg.sender, _owner, _account, _origin);
168     }
```

Listing 3.13: contracts/registry/index.sol

Recommendation Validate the `_origin` before emitting logs. If it's not an valid `_origin`, log it as an unknown origin.

```
/**
 * @dev Create a new Smart Account for a user.
 * @param _owner Owner of the Smart Account.
 * @param accountVersion Account Module version.
 * @param _origin Where Smart Account is created.
 */
function build(
    address _owner,
    uint accountVersion,
    address _origin
) public returns (address _account) {
    require(accountVersion != 0 && accountVersion <= versionCount, "not-valid-account");
    _account = createClone(accountVersion);
    ListInterface(list).init(_account);
    AccountInterface(_account).enable(_owner);
    if ( _origin != msg.sender && _origin != tx.origin && !validOrigin[_origin] ) {
        emit AccountCreated(msg.sender, _owner, _account, address(0));
    } else {
        emit AccountCreated(msg.sender, _owner, _account, _origin);
    }
}
```

Listing 3.14: contracts/registry/index.sol

3.7 Missing Array Length Checks in InstaAccount

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `contracts/account.sol`
- Category: Data Processing Issues [16]
- CWE subcategory: CWE-130 [4]

Description

In the `InstaAccount` contract, the `cast()` cast plays an important role of calling connectors with corresponding input data. Specifically, multiple `_targets` and `_datas` are packed in two arrays and passed into `cast()` which issues the invocations to the functions in each connector one-by-one in one transaction (line 141).

```
118     /**
119      * @dev This is the main function, Where all the different functions are called
120      * from Smart Account.
121      * @param _targets Array of Target(s) to of Connector.
122      * @param _datas Array of Calldata(S) of function.
123     */
124     function cast(
125         address[] calldata _targets,
126         bytes[] calldata _datas,
127         address _origin
128     )
129     external
130     payable
131     {
132         require(isAuth(msg.sender)  msg.sender == instaIndex, "permission-denied");
133         IndexInterface indexContract = IndexInterface(instaIndex);
134         bool isShield = shield;
135         if (!isShield) {
136             require(ConnectorsInterface(indexContract.connectors(version)).isConnector(
                    _targets), "not-connector");
137         } else {
138             require(ConnectorsInterface(indexContract.connectors(version)).
                    isStaticConnector(_targets), "not-static-connector");
139         }
140         for (uint i = 0; i < _targets.length; i++) {
141             spell(_targets[i], _datas[i]);
142         }
143         address _check = indexContract.check(version);
144         if (_check != address(0) && !isShield) require(CheckInterface(_check).isOk(), "
                not-ok");
145         emit LogCast(_origin, msg.sender, msg.value);
146     }
```

Listing 3.15: contracts/account.sol

However, if the two arrays do not have an identical length, there will be a problem. For example, when `_datas.length < _targets.length`, the fallback function of the remaining targets might be invoked due to the `_data[i]` beyond `_datas.length` are undetermined (likely to be 0).

Recommendation Check the lengths of the two arrays. This had been addressed in the patched `contracts/account.sol` by validating `_targets.length == _datas.length`.

```
    /**
     * @dev This is the main function, Where all the different functions are called
     * from Smart Account.
     * @param _targets Array of Target(s) to of Connector.
     * @param _datas Array of Calldata(S) of function.
    */
    function cast(
        address[] calldata _targets,
        bytes[] calldata _datas,
        address _origin
    )
    external
    payable
    {
        require(isAuth(msg.sender)  msg.sender == instaIndex, "permission-denied");
        require(_targets.length == _datas.length, "array-length-invalid");
        IndexInterface indexContract = IndexInterface(instaIndex);
        bool isShield = shield;
        if (!isShield) {
            require(ConnectorsInterface(indexContract.connectors(version)).isConnector(
                _targets), "not-connector");
        } else {
            require(ConnectorsInterface(indexContract.connectors(version)).
                isStaticConnector(_targets), "not-static-connector");
        }
        for (uint i = 0; i < _targets.length; i++) {
            spell(_targets[i], _datas[i]);
        }
        address _check = indexContract.check(version);
        if (_check != address(0) && !isShield) require(CheckInterface(_check).isOk(), "
            not-ok");
        emit LogCast(_origin, msg.sender, msg.value);
    }
```

Listing 3.16: contracts/account.sol

3.8 Flawed Linked List Implementations

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: Medium
- Target: contracts/registry/connectors.sol, contracts/registry/list.sol
- Category: Data Processing Issues [16]
- CWE subcategory: CWE-237 [6]

Description

The list is used for book-keeping the _connectors in InstaConnectors contract. However, the removeFromList() could be improved in many ways. First, the storage slot of list[_connector] is not deleted after it is not linked to other connectors. Since some gas would be refunded by removing the storage slot, it is worth to add delete list[_connector] in the end of removeFromList(). Moreover, removeFromList() could be called twice (or more) with the same _connector while the second call would reset both the first and last pointers. Fortunately, removeFromList() is an internal function with sanity checks in its caller, disable(). There's no existing path to trigger this bug but we should always make each function block secure.

```
    function removeFromList(address _connector) internal {
        if (list[_connector].prev != address(0)) {
            list[list[_connector].prev].next = list[_connector].next;
        } else {
            first = list[_connector].next;
        }
        if (list[_connector].next != address(0)) {
            list[list[_connector].next].prev = list[_connector].prev;
        } else {
            last = list[_connector].prev;
        }
        count = sub(count, 1);

        emit LogDisable(_connector);
    }
```

Listing 3.17: contracts/registry/connectors.sol

Similar logic applies to addToList(). If a _connector is addToList() twice, the list would no longer track all the connectors. For example, we have $first \rightarrow A \leftrightarrow B \leftrightarrow C \leftrightarrow D \leftarrow last$ in the list and we add A again into list. D and A would be connected in line 98-99 and last would point to A in line 104, which results in $D \leftrightarrow A \leftarrow last$. However, first still points to A, which means we cannot track B, C, D from first.

```
96      function addToList(address _connector) internal {
97          if (last != address(0)) {
98              list[_connector].prev = last;
```

```
            list[last].next = _connector;
        }
        if (first == address(0)) {
            first = _connector;
        }
        last = _connector;
        count = add(count, 1);

        emit LogEnable(_connector);
    }
```

Listing 3.18: contracts/registry/connectors.sol

We have identified some other linked list implementations which have similar flaws. They are listed in the following:

```
    function addAccount(address _owner, uint64 _account) internal {
        if (userLink[_owner].last != 0) {
            userList[_owner][_account].prev = userLink[_owner].last;
            userList[_owner][userLink[_owner].last].next = _account;
        }
        if (userLink[_owner].first == 0) userLink[_owner].first = _account;
        userLink[_owner].last = _account;
        userLink[_owner].count = add(userLink[_owner].count, 1);
    }
```

Listing 3.19: contracts/registry/list.sol

```
    function removeAccount(address _owner, uint64 _account) internal {
        uint64 _prev = userList[_owner][_account].prev;
        uint64 _next = userList[_owner][_account].next;
        if (_prev != 0) userList[_owner][_prev].next = _next;
        if (_next != 0) userList[_owner][_next].prev = _prev;
        if (_prev == 0) userLink[_owner].first = _next;
        if (_next == 0) userLink[_owner].last = _prev;
        userLink[_owner].count = sub(userLink[_owner].count, 1);
        delete userList[_owner][_account];
    }
```

Listing 3.20: contracts/registry/list.sol

```
    function addUser(address _owner, uint64 _account) internal {
        if (accountLink[_account].last != address(0)) {
            accountList[_account][_owner].prev = accountLink[_account].last;
            accountList[_account][accountLink[_account].last].next = _owner;
        }
        if (accountLink[_account].first == address(0)) accountLink[_account].first =
            _owner;
        accountLink[_account].last = _owner;
        accountLink[_account].count = add(accountLink[_account].count, 1);
    }
```

Listing 3.21: contracts/registry/list.sol

```
    function removeUser(address _owner, uint64 _account) internal {
        address _prev = accountList[_account][_owner].prev;
        address _next = accountList[_account][_owner].next;
        if (_prev != address(0)) accountList[_account][_prev].next = _next;
        if (_next != address(0)) accountList[_account][_next].prev = _prev;
        if (_prev == address(0)) accountLink[_account].first = _next;
        if (_next == address(0)) accountLink[_account].last = _prev;
        accountLink[_account].count = sub(accountLink[_account].count, 1);
        delete accountList[_account][_owner];
    }
```

Listing 3.22: contracts/registry/list.sol

Recommendation Delete `list[_connector]` after removing `_connector` from the list. Also, validate `_connector` before adding it into the list or removing it from the list.

```
    function removeFromList(address _connector) internal {
        require( !(list[_connector].prev == address(0) && list[_connector].next ==
            address(0)), "not-in-list");
        if (list[_connector].prev != address(0)) {
            list[list[_connector].prev].next = list[_connector].next;
        } else {
            first = list[_connector].next;
        }
        if (list[_connector].next != address(0)) {
            list[list[_connector].next].prev = list[_connector].prev;
        } else {
            last = list[_connector].prev;
        }
        count = sub(count, 1);
        delete list[_connector];

        emit LogDisable(_connector);
    }
```

Listing 3.23: contracts/registry/connectors.sol

```
    function addToList(address _connector) internal {
        require( list[_connector].prev == address(0) && list[_connector].next == address
            (0), "already-in-list");
        if (last != address(0)) {
            list[_connector].prev = last;
            list[last].next = _connector;
        }
        if (first == address(0)) {
            first = _connector;
        }
        last = _connector;
        count = add(count, 1);

        emit LogEnable(_connector);
    }
```

Listing 3.24: contracts/registry/connectors.sol

Last but not the least, if the order of elements in the list doesn't matter, we suggest to replace all the linked lists with arrays, which makes the implementation simpler and easier to maintain.

In the patch, the issue is partially fixed by deleting the storage slot of `list[_connector]` after it is not linked to other connectors. But, the sanity checks before adding/removing an item to/from the list are not addressed in the patch. As mentioned earlier, those internal functions are guarded by the callers with sanity checks. It should be fine for now.

3.9 Missing Disable Function in staticConnectors

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `contracts/registry/connectors.sol`
- Category:Behavioral Issues [14]
- CWE subcategory: CWE-431 [7]

Description

In the `InstaConnectors` contract, it maintains two type of connector objects: `connectors` and `staticConnectors`. Our analysis found that the `staticConnectors` object does not have a `disable` function. It means that the administrator can never delete the existing `staticConnectors`. If there's an instance of `staticConnectors` which must be deleted, it can only be done by re-deploying the contract.

Recommendation Add `disableStatic(address _connector)` function in the `InstaConnectors` contract.

3.10 Gas Optimization

- ID: PVE-010
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `contracts/registry/index.sol`
- Category:Resource Management [18]
- CWE subcategory: CWE-920 [12]

Description

In `AddressIndex` contract, the `addNewAccount()` function validates the version of `_newAccount` after `versionCount++`, which is a waste of gas. Specifically, in the case that `AccountInterface(_newAccount).version()!= versionCount +1`, the EVM execution reverts after `versionCount++` (line 88) with extra gas comsumption for the `SSTORE` opcode which stores `versionCount+1` into storage.

```
/**
 * @dev Add New Account Module.
 * @param _newAccount The New Account Module Address.
 * @param _connectors Connectors Registry Module Address.
 * @param _check Check Module Address.
 */
function addNewAccount(address _newAccount, address _connectors, address _check) external isMaster {
    require(_newAccount != address(0), "not-valid-address");
    versionCount++;
    require(AccountInterface(_newAccount).version() == versionCount, "not-valid-version");
    account[versionCount] = _newAccount;
    if (_connectors != address(0)) connectors[versionCount] = _connectors;
    if (_check != address(0)) check[versionCount] = _check;
    emit LogNewAccount(_newAccount, _connectors, _check);
}
```

Listing 3.25: contracts/registry/index.sol

Recommendation Move the require statement before `versionCount++`, which not only saves some gas but also makes the function comply to the checks-effects-interactions conversion [2].

```
/**
 * @dev Add New Account Module.
 * @param _newAccount The New Account Module Address.
 * @param _connectors Connectors Registry Module Address.
 * @param _check Check Module Address.
 */
function addNewAccount(address _newAccount, address _connectors, address _check) external isMaster {
    require(_newAccount != address(0), "not-valid-address");
    require(AccountInterface(_newAccount).version() == versionCount+1, "not-valid-version");
    versionCount++;
    account[versionCount] = _newAccount;
    if (_connectors != address(0)) connectors[versionCount] = _connectors;
    if (_check != address(0)) check[versionCount] = _check;
    emit LogNewAccount(_newAccount, _connectors, _check);
}
```

Listing 3.26: contracts/registry/index.sol

3.11 Other Suggestions

Due to the fact that compiler upgrades might bring unexpected compatibility or inter-version consistencies, it is always suggested to use fixed compiler versions whenever possible. As an example, we highly encourage to explicitly indicate the Solidity compiler version, e.g., `pragma solidity 0.6.0;` instead of `pragma solidity ^0.6.0;`.

Moreover, we strongly suggest not to use experimental Solidity features or third-party unaudited libraries. If necessary, refactor current code base to only use stable features or trusted libraries. In case there is an absolute need of leveraging experimental features or integrating external libraries, make necessary contingency plans.

4 Conclusion

In this audit, we thoroughly analyzed the InstaDApp Smart Accounts documentation and implementation. The audited system does involve various intricacies in both design and implementation. The current code base is well organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

5 Appendix

5.1 Basic Coding Bugs

5.1.1 Constructor Mismatch

- Description: Whether the contract name and its constructor are not identical to each other.
- Result: Not found
- Severity: Critical

5.1.2 Ownership Takeover

- Description: Whether the set owner function is not protected.
- Result: Not found
- Severity: Critical

5.1.3 Redundant Fallback Function

- Description: Whether the contract has a redundant fallback function.
- Result: Not found
- Severity: Critical

5.1.4 Overflows & Underflows

- Description: Whether the contract has general overflow or underflow vulnerabilities [21, 22, 23, 24, 26].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [27] is an issue when code can call back into your contract and change state, such as withdrawing ETHs.
- Result: Not found
- Severity: Critical

5.1.6 Money-Giving Bug

- Description: Whether the contract returns funds to an arbitrary address.
- Result: Not found
- Severity: High

5.1.7 Blackhole

- Description: Whether the contract locks ETH indefinitely: merely in without out.
- Result: Not found
- Severity: High

5.1.8 Unauthorized Self-Destruct

- Description: Whether the contract can be killed by any arbitrary address.
- Result: Not found
- Severity: Medium

5.1.9 Revert DoS

- Description: Whether the contract is vulnerable to DoS attack because of unexpected `revert`.
- Result: Not found
- Severity: Medium

5.1.10 Unchecked External Call

- Description: Whether the contract has any external call without checking the return value.
- Result: Not found
- Severity: Medium

5.1.11 Gasless Send

- Description: Whether the contract is vulnerable to gasless send.
- Result: Not found
- Severity: Medium

5.1.12 Send Instead Of Transfer

- Description: Whether the contract uses send instead of transfer.
- Result: Not found
- Severity: Medium

5.1.13 Costly Loop

- Description: Whether the contract has any costly loop which may lead to Out-Of-Gas exception.
- Result: Not found
- Severity: Medium

5.1.14 (Unsafe) Use Of Untrusted Libraries

- Description: Whether the contract use any suspicious libraries.
- Result: Not found
- Severity: Medium

5.1.15 (Unsafe) Use Of Predictable Variables

- Description: Whether the contract contains any randomness variable, but its value can be predicated.
- Result: Not found
- Severity: Medium

5.1.16 Transaction Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: Not found
- Severity: Medium

5.1.17 Deprecated Uses

- Description: Whether the contract use the deprecated `tx.origin` to perform the authorization.
- Result: Not found
- Severity: Medium

5.2 Semantic Consistency Checks

- Description: Whether the semantic of the white paper is different from the implementation of the contract.
- Result: Not found
- Severity: Critical

5.3 Additional Recommendations

5.3.1 Avoid Use of Variadic Byte Array

- Description: Use fixed-size byte array is better than that of `byte[]`, as the latter is a waste of space.
- Result: Not found
- Severity: Low

5.3.2 Make Visibility Level Explicit

- Description: Assign explicit visibility specifiers for functions and state variables.
- Result: Not found
- Severity: Low

5.3.3 Make Type Inference Explicit

- Description: Do not use keyword `var` to specify the type, i.e., it asks the compiler to deduce the type, which is not safe especially in a loop.
- Result: Not found
- Severity: Low

5.3.4 Adhere To Function Declaration Strictly

- Description: Solidity compiler (version 0.4.23) enforces strict ABI length checks for return data from `calls()` [1], which may break the the execution if the function implementation does NOT follow its declaration (e.g., no return in implementing `transfer()` of ERC20 tokens).
- Result: Not found
- Severity: Low

References

[1] axic. Enforcing ABI length checks for return data from calls can be breaking. https://github.com/ethereum/solidity/issues/4116.

[2] ethereum. Security Considerations. https://solidity.readthedocs.io/en/v0.6.4/security-considerations.html#use-the-checks-effects-interactions-pattern.

[3] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[4] MITRE. CWE-130: Improper Handling of Length Parameter Inconsistency. https://cwe.mitre.org/data/definitions/130.html.

[5] MITRE. CWE-229: Improper Handling of Values. https://cwe.mitre.org/data/definitions/229.html.

[6] MITRE. CWE-237: Improper Handling of Structural Elements. https://cwe.mitre.org/data/definitions/237.html.

[7] MITRE. CWE-431: Missing Handler. https://cwe.mitre.org/data/definitions/431.html.

[8] MITRE. CWE-454: External Initialization of Trusted Variables or Data Stores. https://cwe.mitre.org/data/definitions/454.html.

[9] MITRE. CWE-621: Variable Extraction Error. https://cwe.mitre.org/data/definitions/621.html.

[10] MITRE. CWE-754: Improper Check for Unusual or Exceptional Conditions. https://cwe.mitre.org/data/definitions/754.html.

[11] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[12] MITRE. CWE-920: Improper Restriction of Power Consumption. https://cwe.mitre.org/data/definitions/920.html.

[13] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[14] MITRE. CWE CATEGORY: Behavioral Problems. https://cwe.mitre.org/data/definitions/438.html.

[15] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[16] MITRE. CWE CATEGORY: Data Processing Errors. https://cwe.mitre.org/data/definitions/19.html.

[17] MITRE. CWE CATEGORY: Initialization and Cleanup Errors. https://cwe.mitre.org/data/definitions/452.html.

[18] MITRE. CWE CATEGORY: Resource Management Errors. https://cwe.mitre.org/data/definitions/399.html.

[19] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[20] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[21] PeckShield. ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). https://www.peckshield.com/2018/04/22/batchOverflow/.

[22] PeckShield. New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-11239). https://www.peckshield.com/2018/05/18/burnOverflow/.

[23] PeckShield. New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-10706). https://www.peckshield.com/2018/05/10/multiOverflow/.

[24] PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). https://www.peckshield.com/2018/04/25/proxyOverflow/.

[25] PeckShield. PeckShield Inc. https://www.peckshield.com.

[26] PeckShield. Your Tokens Are Mine: A Suspicious Scam Token in A Top Exchange. https://www.peckshield.com/2018/04/28/transferFlaw/.

[27] Solidity. Warnings of Expressions and Control Structures. http://solidity.readthedocs.io/en/develop/control-structures.html.