SALUS SECURITY

MAR 2024

CODE SECURITY ASSESSMENT

INK FINANCE

Overview

Project Summary

- Name: Ink Finance - Incremental Audit
- Platform: EVM-compatible chains
- Language: Solidity
- Repository:
  - https://github.com/Ink-Finance-Inc/v2-governance-core
  - https://github.com/Ink-Finance-Inc/v3-economy-core
- Audit Range: See Appendix - 1

Project Dashboard

Application Summary

Name	Ink Finance - Incremental Audit
Version	v1
Type	Solidity
Dates	May 29 2024
Logs	May 29 2024

Vulnerability Summary

Total High-Severity issues	7
Total Medium-Severity issues	3
Total Low-Severity issues	0
Total informational issues	1
Total	11

Contact

E-mail: support@salusec.io

Risk Level Description

High Risk	The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for clients' reputations or serious financial implications for clients and users.
Medium Risk	The issue puts a subset of users' sensitive information at risk, would be detrimental to the client's reputation if exploited, or is reasonably likely to lead to a moderate financial impact.
Low Risk	The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or defense in depth.

Content

Introduction

1.1 About SALUS

At Salus Security, we are in the business of trust.

We are dedicated to tackling the toughest security challenges facing the industry today. By building foundational trust in technology and infrastructure through security, we help clients to lead their respective industries and unlock their full Web3 potential.

Our team of security experts employ industry-leading proof-of-concept (PoC) methodology for demonstrating smart contract vulnerabilities, coupled with advanced red teaming capabilities and a stereoscopic vulnerability detection service, to deliver comprehensive security assessments that allow clients to stay ahead of the curve.

In addition to smart contract audits and red teaming, our Rapid Detection Service for smart contracts aims to make security accessible to all. This high calibre, yet cost-efficient, security tool has been designed to support a wide range of business needs including investment due diligence, security and code quality assessments, and code optimisation.

We are reachable on Telegram (https://t.me/salusec), Twitter (https://twitter.com/salus_sec), or Email (support@salusec.io).

1.2 Audit Breakdown

The objective was to evaluate the repository for security-related issues, code quality, and adherence to specifications and best practices. Possible issues we looked for included (but are not limited to):

- Risky external calls
- Integer overflow/underflow
- Transaction-ordering dependence
- Timestamp dependence
- Access control
- Call stack limits and mishandled exceptions
- Number rounding errors
- Centralization of power
- Logical oversights and denial of service
- Business logic specification
- Code clones, functionality duplication

1.3 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release and does not give any warranties on finding all possible security issues with the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues.

Findings

2.1 Summary of Findings

ID	Title	Severity	Category	Status
1	Vulnerable ECDSA library	High	Configuration	Pending
2	Cross-chain replay attacks are possible with isValidSignV1()	High	Cryptography	Pending
3	Users can create proposal and make it accepted immediately	High	Access Control	Pending
4	Missing modifier for approveTo() allows attacker to steal any asset	High	Access Control	Pending
5	Malicious users can modify the userInfo of other users	High	Data Validation	Pending
6	Signature verification in verifyUser can be bypassed	High	Access Control	Pending
7	Arbitrary upgrade vulnerability	High	Access Control	Pending
8	Anyone can create a proposal through InkMainDAO	Medium	Access Control	Pending
9	The staking engine cannot be updated using ContractUpgradeAgent	Medium	Business Logic	Pending
10	Missing validation of member in TreasuryManagerAgent	Medium	Business Logic	Pending
11	Missing zero address checks	Informational	Data Validation	Pending

2.2 Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

1. Vulnerable ECDSA library

Severity: High	Category: Configuration
Target: - contracts/utils/SignManager.sol	

Description

A vulnerability found in the ECDSA library developed by the OpenZeppelin team. The OpenZeppelin team published a security advisory on GitHub on August 22nd, 2022. According to the vulnerability, recover and tryRecover functions are vulnerable, as those functions accept the standard signature format and compact signature format. (EIP-2098) The functions ECDSA.recover and ECDSA.tryRecover are vulnerable to some sort of signature malleability because they accept compact EIP- 2098 signatures in addition to the traditional 65-byte signature format.

This is only an issue for the functions that take a single byte argument, and not the functions that take r, v, s or r, vs as separate arguments. Potentially affected contracts are those that implement signature reuse or replay protection by marking the signature itself as used, rather than the signed message or a nonce included in it. A user can take a signature that has already been submitted, submit it again in a different form, and bypass this protection.
Affected Versions: >= 4.1.0 < 4.7.3

package.json:L29

```
"@openzeppelin/contracts": "^4.4.2",
```

It was observed that SignManager contracts were using the vulnerable ECDSA library to recover signatures.

Recommendation

It is recommended to update the @openzeppelin/contracts package version to 4.7.3 or higher to fix this finding.

2. Cross-chain replay attacks are possible with isValidSignV1()

Severity: High	Category: Cryptography

Target:
- contracts/utils/SignManager.sol

Description

contracts/utils/SignManager.sol:L103-L114

```
function isValidSignV1(
    uint256 projectID,
    bytes memory signature_,
    bytes32 signedCont_
) external view override returns (bool) {
    address requiredProjectSigner = _projectSigner[projectID];

    address recovered = ECDSA.recover(signedCont_, signature_);
    return recovered == requiredProjectSigner;
}
```

The function isValidSignV1() is used to validate signatures and ensure that recovered address from ECDSA is requiredProjectSigner which is stored in mapping.

contracts/utils/SignManager.sol:L51-L68

```
function packForProject1(
    address target,
    address token,
    uint256 amount,
    uint256 batch,
    uint256 endTime
) external view override returns (bytes32) {
    if (endTime <= block.timestamp) {
       revert("endTime is not valid");
    }
    bytes32 packaged = keccak256(
        abi.encodePacked(target, token, batch, amount, endTime)
    );
    bytes32 withPrefixed = prefixed(packaged);
    return withPrefixed;
}
```

However, there is a lack of critical safeguards in the signature hash calculation to prevent replay attacks on different chain IDs and the same smart contract.
This could lead to a cross-chain replay vulnerability in the BIGA contract when deployed on multiple chains with the same validator address. could exploit a single signature across multiple chains, resulting in asset losses for the project.

Recommendation

It is recommended that signatures follow the EIP712 standard and include the chainId as a signature domain to prevent cross-chain replay attacks.

3. Users can create proposal and make it accepted immediately

Severity: High	Category: Access Control
Target: - contracts/daos/InkMainDAO.sol	

Description

contracts/daos/InkMainDAO.sol

```
function runProposal(
    NewProposalInfo calldata proposal,
    address proposer,
    bytes calldata data
) external {
    // only admin
    bytes32 proposalID = _runProposal(proposal, true, msg.sender, data);
    IProposalHandler(_proposalHandlerAddress).decideProposal(
        proposalID,
        proposer,
        true,
        data
    );
}
```

The runProposal() function allows the caller to create a proposal and immediately decide the proposal's outcome with an "agree" status.

Malicious users could exploit this function to create malicious proposals and immediately execute their agents, potentially causing unpredictable and significant impacts on the DAO system.

Recommendation

It is recommended to add permission control to the runProposal() function, e.g. adding the onlyAdmin modifier.

4. Missing modifier for approveTo() allows attacker to steal any asset

Severity: High	Category: Access Control
Target: - contracts/ucv/PayrollUCV.sol	

Description

contracts/ucv/PayrollUCV.sol:L253-L265

```
function approveTo(
    address spender,
    address token,
    uint256 tokenType,
    uint256 tokenID,
    uint256 amount
) external override {
    if (tokenType == 721) {
        IERC721(token).approve(spender, tokenID);
    } else if (tokenType == 20) {
        IERC20(token).approve(spender, amount);
    }
}
```

The approveTo() function allows the caller to approve the contract's tokens to any person.

However, the function lacks appropriate access control. Thus unauthorized token approval may cause the contract to lose all its tokens.

Recommendation

It is recommended to add appropriate access control to the approveTo() function.

5. Malicious users can modify the userInfo of other users

Severity: High	Category: Data Validation

Target:
- contracts/cores/KYCVerifyManager.sol

Description

contracts/cores/KYCVerifyManager.sol:L200-L223

```
function verifyUserXYZBatch(string memory zone, VerifyInfo[] memory infos)
    external
{
    UserKV[] memory kvs = new UserKV[](infos.length);
    for (uint256 i = 0; i < infos.length; i++) {
        kvs[i].key = infos[i].accountType;
        kvs[i].user = infos[i].wallet;
        kvs[i].typeID = keccak256("account_info");
        kvs[i].data = abi.encode(
            kvs[i].key,
            infos[i].account,
            infos[i].data
        );
        emit KYCVerified(
            zone,
            infos[i].wallet,
            infos[i].accountType,
            infos[i].account,
            infos[i].data
        );
    }

    IIdentity(_identityManager).batchSetUserKVs(zone, kvs);
}
```

The verifyUserXYZBatch() function allows the caller to pass an array of “infos” for multiple users to be verified.

However, this function lacks access control and validation of the “infos” array. This could potentially allow malicious users to maliciously manipulate other users' UserKVs by calling this function, resulting in incorrect user information records in IdentityManager.

Recommendation

It is recommended to implement reasonable access control for this function or verify that the authenticated user is authorized by signature.

6. Signature verification in verifyUser can be bypassed

Severity: High	Category: Access Control

Target:
- contracts/cores/KYCVerifyManager.sol

Description

contracts/cores/KYCVerifyManager.sol:L86-L112

```
function verifyUser(
    bytes memory signature,
    string memory zone,
    string memory accountType,
    string memory account,
    string memory data /*, address testWallet */
) external {
    address wallet = msg.sender;

    bytes32 signData = keccak256(
        abi.encodePacked(zone, accountType, account, wallet, data)
    );
    address actualSigner = _getSigner(signature, signData);
    if (valiedSign[signData] != 0) {
        revert INK_ERROR(4002);
    }
    if (actualSigner != _signer) {
        revert INK_ERROR(4003);
    }

    valiedSign[signData] = 1;

    _verifyUser(zone, wallet, accountType, account, data);
}
```

By observing the above code, it can be noticed that the prerequisite for user verification is obtaining the signature corresponding to the signer.

contracts/cores/KYCVerifyManager.sol:L114-L138

```
function daoVerifyUser(
    string memory zone,
    string memory accountType,
    string memory account,
    string memory data /*, address testWallet */
) external {
    address wallet = msg.sender;
    bytes32 signData = keccak256(
        abi.encodePacked(zone, accountType, wallet, data)
    );

    if (valiedSign[signData] != 0) {
       revert INK_ERROR(4002);
    }
    if (!startsWith(zone, "DAO_")) {
       revert INK_ERROR(4004);
   }

    valiedSign[signData] = 1;
```

```
    _verifyUser(zone, wallet, accountType, account, data);
}
```

In the daoVerifyUser() function, the same logic is executed without the need to verify if the wallet has the signature of the signer.
This means that users can bypass the signature check of the verifyUser() function by using the daoVerifyUser() function.

Recommendation

It is recommended to add appropriate access control to this function, e.g. only dao calls are allowed.

7. Arbitrary upgrade vulnerability

Severity: High	Category: Access Control

Target:
- contracts/products/funds/FundManager.sol

Description

contracts/products/funds/FundManager.sol:L1369-L1377

```
function updateContract(address proxyAddress, address newImplement)
    external
{
    (, address newBeaconAddress) = IFactoryManager(_factoryManager)
        .getBeaconKeyAndAddress(proxyAddress, newImplement);
    InkBeaconProxy(payable(proxyAddress)).upgradeTo(
        address(newBeaconAddress)
    );
}
```

The updateContract() function has the ability to upgrade a contract, but the function lacks the appropriate access control.

Since the contract may be the privileged address of other proxy contracts. This could lead to an attacker being able to maliciously upgrade these proxy contracts using this function.

Recommendation

It is recommended to add appropriate access control to this function.

8. Anyone can create a proposal through InkMainDAO

Severity: Medium	Category: Access Control

Target:
- contracts/daos/InkMainDAO.sol

Description

In Ink Finance, the proposal creation process should be: committee - dao - proposalHandler.

contracts/committee/TheBoard.sol:L46-L73

```
function newProposal(
    NewProposalInfo calldata proposal,
    bool commit,
    bytes calldata data
) external override returns (bytes32 proposalID) {
    if (!_hasDutyToOperate(DutyID.PROPOSER, _msgSender())) {
        revert INK_ERROR(1024);
    }

    if (!_isPledgeForManager(_msgSender())) {
        revert INK_ERROR(1023);
    }
    ...
    IProposalHandler proposalHandler = IProposalHandler(getParentDAO());
    proposalID = proposalHandler.newProposal(
        proposal,
        commit,
        _msgSender(),
        data
    );
}
```

In committee, there will be access control for the caller. Only people who have PROPOSER duty and have already participated in a pledge can create a proposal.

contracts/daos/InkMainDAO.sol:L249-L257

```
function newProposal(
    NewProposalInfo calldata proposal,
    bool commit,
    address proposer,
    bytes calldata data
) public override returns (bytes32 proposalID) {
    // only committee
    return _runProposal(proposal, commit, proposer, data);
}
```

But in InkMainDAO, due to the lack of access control (onlyCommittee modifier). Users can bypass the committee check by calling newProposal in the dao directly.

This can lead to malicious users creating illegal proposals.

Recommendation

It is recommended to implement proper access control in newProposal().

9. The staking engine cannot be updated using ContractUpgradeAgent

Severity: Medium	Category: Business Logic

Target:
- contracts/agents/ContractUpgradeAgent.sol

Description

contracts/bases/BaseDAO.sol:L339-L355

```
function updateContract(
    uint256 updateType,
    address proxyAddress,
    address newImplement
) external override onlyAgent {
    if (updateType == 1) {
        (, address newBeaconAddress) = IFactoryManager(_factoryAddress)
            .getBeaconKeyAndAddress(proxyAddress, newImplement);
        InkBeaconProxy(payable(proxyAddress)).upgradeTo(
            address(newBeaconAddress)
        );
    } else if (updateType == 2) {
        // if type == 2 it's update staking engine, we don't need proxyAddress to
upgrade
        IEconomyEngineV1Factory(_economyEngineFactcory)
            .upgradeStakingEngineByDao(_stakingAddr, newImplement);
    }
}
```

When the contract needs to be upgraded, you need to use an agent to call updateContract() to do it.

contracts/agents/ContractUpgradeAgent.sol:L54-L75

```
function exec(bytes32 proposalID) external override onlyCallFromDAO {
    bytes32 typeID;
    bytes memory bytesData;

    (typeID, bytesData) = IProposalHandler(getAgentDAO()).getProposalKvData(
        proposalID,
        ORIGINAL_PROXY
    );
    address proxyAddress = abi.decode(bytesData, (address));

    (typeID, bytesData) = IProposalHandler(getAgentDAO()).getProposalKvData(
        proposalID,
        NEW_IMPLEMENTATION
    );
    address newImplementation = abi.decode(bytesData, (address));

    IDAO(getAgentDAO()).updateContract(1, proxyAddress, newImplementation);
```

```
    _executed = true;
}
```

But in the ContractUpgradeAgent contract, it is calling the updateContract() function with a hard-coded updateType.
This will cause the staking engine cannot be upgraded.

Recommendation

It is recommended to read the updateType from the proposal to avoid using hardcoding.

10. Missing validation of member in TreasuryManagerAgent

Severity: Medium	Category: Business Logic

Target:
- contracts/agents/TreasuryManagerAgent.sol

Description

In TreasuryManagerAgent, validation of members is missing in multiple places.

a) During the agent's preExec period, the parameters of the proposal should be validated.

contracts/agents/TreasuryManagerAgent.sol:L29-L40

```
function preExec(bytes32 proposalID)
    external
    override
    returns (bool success)
{
    // _verifyManagers(proposalID, _MD_SIGNER);
    // _verifyManagers(proposalID, _MD_OPERATOR);
    // _verifyManagers(proposalID, _MD_AUDITOR);
    // _verifyManagers(proposalID, _MD_INVESTOR);
    success = true;
}
```

However, the highlighted portion of the code above is commented out, which could result in an unauthorized member accidentally becoming a privileged role.

b) During the execution period of the agent, the _setMemberDuties() function is called to add duties to the member in the dao.

contracts/agents/TreasuryManagerAgent.sol:L29-L40

```
function _setMemberDuties(bytes32 dutyID, bytes memory memberBytes) internal
{
    ...
    // require(_isPledgeForManager(members), "not enough pledge for manager");
    for (uint256 i = 0; i < members.length; i++) {
        IDAO(getAgentDAO()).addDuty(members[i], dutyID);
    }
}
```

However, there is a lack of validation that the member is eligible for the privileged role (the highlighted part of the code above).

Recommendation

It is recommended to uncomment part of the code for a full validation of the parameters of the proposal.

2.3 Informational Findings

11. Missing zero address checks

Severity: Informational	Category: Data Validation
Target: - contracts/daos/InkMainDAO.sol	

Description

It is considered a security best practice to verify addresses against the zero address during initialization or setting. However, this precautionary step is absent for address variables “_signManager”.

contracts/daos/InkMainDAO.sol:L1446-L1449

```
function updateSignerManager(address newSigner) external {
    require(_msgSender() == _ownerAddress, "only owner");
    _signManager = newSigner;
}
```

Recommendation

Consider adding zero address checks for address variables “_signManager”.

Appendix

Appendix 1 - Files in Scope

We audited the commit 2e80942 that introduced new features to the Ink-Finance-Inc/v2-governance-core repository.

File	SHA-1 hash
ContractUpgradeAgent.sol	7d52846220edf188492d035cfbd500178c07a2e5
DAOInfoSetupAgent.sol	38b03fb76cebd8967b87c29339633a8d81a318c4
IncomeManagerSetupAgent.sol	317c3bcb8d6a2590d70185c23111dad9cc14347c
InvestmentManagementSetupAgent.sol	df567bf897de804f53f6d45cdeac2f3275d65167
TreasuryManagerAgent.sol	29509d242a12b9e87fb286fa966fc4472dfff388
BaseCommittee.sol	64c80e299e375c8515dcee7998edb446c4908677
BaseDAO.sol	34e8f56866347535b02e29d52f680ca87d6a4d56
BaseUCV.sol	76eae2f429d3d51f99ce1e6f905cb70a35f5dd6b
InvestmentCommittee.sol	59a8da89aab2ca2c50878e3dd66ee70abd00e526
TheBoard.sol	6b00fc28b5a09f67def1860926e9540053968d8f
ThePublic.sol	37417fd78b53dc3e8921c47b1b60562491e06ebf
TreasuryCommittee.sol	02df294b0c30268617eba344cdcbb5e3a1575ab6
IdentityManager.sol	8993f10ce6866605f6a0cc7d37bb87f8ce411818
IdentityManagerV2.sol	e1f130bcc98613ae3e456097ae5a9041d4470d99
KYCVerifyManager.sol	c42213035423e8961edce11bfdccc56f4bb68f17
InkMainDAO.sol	796079465a5ba5838da8703e967c215fcf2bc755
MasterDAO.sol	bade029343934afcc7c39ceda40a2f8b6d54b57c
FundManager.sol	7cdb494a87afe8e7a7e7d93fb342a30498b410b1
EscrowManager.sol	f0ffaf083902ed5e96a55268b70fefb5b382e170
InkFund.sol	32d04561503b25cbdb2ef45918e14c9004363964
ProposalHandler.sol	8292bf63abfd727cd5737516bd97776432f11cf7
InkBadgeERC20.sol	b3c7884b1c689ed5e9c6376259c477d7a0010d02

InvestmentUCVManager.sol	cb748ddf8ddc4d0cd9a77cdd73c2bc445ba00101
PayrollUCV.sol	37cb2602eb7e4b61028b4dc570ad14a7e1d98cbd
PayrollUCVManager.sol	e94f2f2a35c74b90f9cb0295bb1ccd7084b00d48
TreasuryIncomeManager.sol	509296e13b891d6c79ecf7bad076ea34bdf88ef4
FactoryManager.sol	2f436cfec1193c1378d10dce788dc03f2e1976cd
SignManager.sol	2817666cf07026e2515218d2552177f95dae1d31

and the commit c527546 that introduced new features to the Ink-Finance-Inc/v3-economy-core repository.

File	SHA-1 hash
EconomyAddressesProvider.sol	29e4820e7194b4670e7e470094e71449beb889eb
EconomyEngineV1Factory.sol	fbd612ff06a2a5bc6b465d3d27e631b9d9cee391