TABLE OF CONTENTS

1 INTRODUCTION **2**

1.1 Results presentation 2

1.2 Context 2

1.3 Service management 3

2 EXECUTIVE SUMMARY **4**

2.1 Summary 4

2.2 Vulnerabilities and recommendations summary 5

2.3 Remarks 6

3 SMART CONTRACT AUDIT **7**

3.1 Context 7

3.2 Security analysis 8

3.2.1 Architecture and dependencies 8

3.2.2 Access control 8

3.2.3 Data storage 9

3.2.4 Gas usage 9

3.2.5 Business logic 10

3.2.6 Development practices 12

4 APPENDIX **13**

4.1 Approach presentation 13

4.2 Results presentation 14

4.3 Risks terminology 17

TOUR CBX, 20ème ETAGE, 1 PASSERELLE DES REFLETS, 92913 PARIS LA DEFENSE CEDEX • TEL +33 1 41 91 77 77
INTRINSEC SECURITE • SAS AU CAPITAL DE 2 760 680 € • SIREN: 812 535 284 • APE: 6203Z

1 INTRODUCTION

In this section we summarize your needs as we understood them during our different interviews, in order to ensure that the mission is in line with your expectations. This allows us to precisely adapt the service regarding your priorities in terms of IT security.

1.1 RESULTS PRESENTATION

Each section details the actions performed, the results obtained, our comments on these points, and possible exploitation scenarios discovered.

More synthetically, four types of informative frameworks appear in the report. They are detailed in section 4.2 Results presentation - page 14.

1.2 CONTEXT

Crédit Agricole Corporate and Investment Bank (« CA-CIB ») is Crédit Agricole's corporate and investment banking entity. Founded in 2004, CA-CIB is active in a broad range of capital markets, investment banking and financing activities. In a goal to further develop their activities, CA-CIB and Skandinaviska Enskilda Banken AB (« SEB ») is launching a blockchain based financial project. Therefore, they wish to evaluate the security of both the governance and application smart contracts running on the blockchain. This document is the Governance part of the audit.

This type of audit requires a high level of expertise and knowledge about Ethereum security, and Intrinsec has this competence thanks to consultants with experience in the Ethereum blockchain ecosystem. Therefore, the application of a complete and recognized methodology detailed below will allow them to leverage their expertise to thoroughly assess the security of these smart contracts.

This report presents the results of the governance smart contract audit.

We draw your attention to the fact that the service was performed in strict loyalty and fairness and that necessary reservations have to be taken into account while analyzing the results (especially in terms of exhaustiveness), forced by the deadlines and the testing conditions (e.g. initial premise for an attack scenario...).

1.3 SERVICE MANAGEMENT

Intrinsec performed the service based on the following elements:

Scope	Governance smart contracts
Restrictions	No denial of service
Preliminary information	• Source code of the smart contracts • RPC of the testnet • Access to the front-end of SATURN and the block explorer of the testnet • Documentation
Date of completion	October to December 2022

2 EXECUTIVE SUMMARY

2.1 SUMMARY

The overall security level of the audited smart contracts is good.

Indeed, despite one identified vulnerability, the associated risk is low and **the probability of occurrence of an attack by these vectors remains very low**.

The vulnerability is related to **a lack of protection against reentrancy attacks**. Exploitation of this vulnerability could lead to the loss of funds from the governance contract, but the prerequisites and steps necessary for an exploitation are very important.

Finally**, no immediate attack scenarios were identified during the audit**.

A detailed action plan is available in section 2.2 Vulnerabilities and recommendations summary – page 5.

2.2 VULNERABILITIES AND RECOMMENDATIONS SUMMARY

The following table presents a summary of the vulnerabilities and related actions to take in order to remove or mitigate the previously detailed risks. ***Recommendations of the highest priority numbers must be applied first.*** *The priority is calculated according to the severity, the complexity and the estimated work/cost for each recommendation (for more details, please refer to the appendix 4.2 Results presentation - page 14).*

Ref.	Page	Scope	Vulnerability	Severity (/4)	Recommendation	Estimated complexity	Estimated workload/cost	Estimated priority (/4)
VULN-01	11	Governance smart contracts	Reentrency attack on the Pledge contract	2	Use the nonReentrant modifier for every sensitive function	Low	Low	3

2.3 REMARKS

Ref.	Page	Scope	Remark
REM-01	8	Smart contracts	Use a locked solidity version
REM-02	9	Smart contracts	Sort variables by types
REM-03	12	Smart contracts	Use an explicit type

3 SMART CONTRACT AUDIT

Smart contract audits are based on the Smart Contract Security Verification Standard (SCSVS). This reference, divided into 14 sections, gathers a set of verification points aiming at checking the good practices, the architecture and the security level of a contract. The rest of the report details the analysis performed during the audit, based on this standard and on an internal standard.

3.1 CONTEXT

The objective of the audit was to audit the security of the Governance smart contracts

Here is the hashes of all smart contract source code files used:

Filename	Sha256 hash
AuditorGovernance.sol	ab99c7520f7a2f866cd83699e425aac849c1c61d5e4227e8d43b0de4edfb5980
CarbonFootprint.sol	9acb1656c735a62d92994e85f6fb7d16648e2c447d7b824a0a7bf2cbfb54621a
Governance.sol	2d6ec37ef8d94eaba57f4276996bd2bca36920b3d6a0990f13a79c4f94ff978f
GovernanceLocalNetwork.sol	8d8e1fe482324e19105442f95b56450c08a79f1b228ecc80db2ef0f700bc1948
GovernanceTesting.sol	9e04c197a2456ec297c0129ab5ed93b78bb16537bfd02c6c016ab38261c4f870
ImprovementProposal.sol	cfa99375f0b1a186fa143d7c6bd2e7db9b69ef7de8fb2b512466848c5376ab05
NodeDelegation.sol	59201d82e8864ad796956f4fa72e0686346fdb0b7e05e82803c82711b24bb597
PledgeContract.sol	9c43d73096910c3fa622d6900addaf74abac6999603d2d13032a5861ad131a42
Utils/ReentrancyGuard.sol	aa73590d5265031c5bb64b5c0e7f84c44cf5f8539e6d8606b763adac784e8b2e
Intf/IAuditorGovernance.sol	817418e2557178958f5253cfffb949931cac41cb194f2cc4a277203dc22818b8
ICarbonFootprint.sol	c441c00701e0e25e58d0227e802cd71719a43e2f6e9f523281ab7366fc3f2efa
IImprovementProposal.sol	2dd9d715b55e93b316df89b6a9b8045c1d2c51132e6c99829a3205cd84d57146
INodeDelegation.sol	0f1096cba7d64e6056789d15a318afb5bab3d7ee3b7bec5ccaeebdc3abcf0aab
IPledgeContract.sol	aecab9f1ed8d74073bd0bf4194fb3519ec59fb4422be7dea0c5973a43c09ae1a

3.2 SECURITY ANALYSIS

3.2.1 Architecture and dependencies

3.2.1.1 *Dependencies*

The *governance* project (that contains the smart contracts) has two dependencies:

package	version	Last version
@openzeppelin/contracts[1]	^4.3.2	4.8.0
@saturn-chain/smart-contract[2]	^1.0.18	1.0.19

✓	All version ranges provided in the project configuration lead to the use of the latest available versions.

3.2.1.2 *Solidity*

All smart contracts are developed using the *Solidity* language. The version used by the compiler is defined in the source code using a pragma directive.

For the audited contracts, the version is defined by:

```
pragma solidity >=0.7.0 <0.9.0;
```

REM-01	Use a locked solidity version

To ensure consistency in the compilation of smart contracts, we recommend using a locked version with the latest version enabled.

Example using the latest version when this report is written:

```
pragma solidity 0.8.17;
```

3.2.2 Access control

Role management

✓	All steps and conditions have been verified in the source code and no issue has been identified.

Role verification

✓	No issue has been identified while auditing the source code of the contracts.

[1] https://www.npmjs.com/package/@openzeppelin/contracts
[2] https://www.npmjs.com/package/@saturn-chain/smart-contract

3.2.3 Data storage

The contracts hold information regarding the bond register and all trade contracts. Some of this information may be confidential in an ordinary trade but the trade off to publicly expose this information is explicitly accepted by the participants.

	Data stored in the blockchain are controlled.

3.2.4 Gas usage

Storage optimization

Solidity contracts have contiguous 32 bytes (256 bit) slots used for storage. If a variable we are trying to pack exceeds the 32 bytes limit of the current slot, it gets stored in a new one. The struct *AuditorStatus* from the contract *Auditor-Governance* is declared in the order below:

```
// AuditorGovernance.sol line 24
struct AuditorStatus {
    bool registered;
    uint256 votes;
    bool approved;
    uint256 registeredAtBlock;
    uint256 statusUpdateBlock;
    mapping(address => NodeVote) voters;
    uint256 minPledgeAtLastAudit;
    uint256 lastAuditAtBlock;
}
```

A *bool* is internally represented by a *uint8* (1 byte). 32 continuous *bool* can be packed in a 32 bytes slot.

You current struct size is 8 x 32 bytes (256 bytes), it can be optimized in the order below:

```
struct AuditorStatus {
    bool registered;                    // 1 slot (32 bytes)
    bool approved;                      // 1 bytes included in the previous slot
    uint256 votes;                      // 1 slot (32 bytes)
    uint256 registeredAtBlock;          // 1 slot (32 bytes)
    uint256 statusUpdateBlock;          // 1 slot (32 bytes)
    uint256 minPledgeAtLastAudit;       // 1 slot (32 bytes)
    uint256 lastAuditAtBlock;           // 1 slot (32 bytes)
    mapping(address => NodeVote) voters; // 1 slot (32 bytes)
}
```

The optimized struct size is now 7 x 32 bytes (224 bytes).

REM-02	Sort variables by types
As detailed before, we recommend declaring little variables (uint8, bool…) together in order to add it in the same 32 bytes slot.	

- https://docs.soliditylang.org/en/v0.8.17/internals/layout_in_storage.html
- https://docs.soliditylang.org/en/v0.8.17/internals/layout_in_memory.html

3.2.5 Business logic

3.2.5.1 *Contract whitelisting mechanism*

Reentrancy attacks

A reentrancy attack occurs when a function makes an external call to another untrusted contract, for instance during following a deposit. The untrusted contract could then make a call to a function of the initial contract. If the contract fails to update its state before the call to the untrusted contract and without proper protection, it could enable an attacker to drain the contract's funds.

In the audit contracts, the *ReentrancyGuard* of *Openzepplin* is used to add a protection layer. This library exposes a modifier trying to prevent reentrancy attacks by ensuring that there cannot be a nested call to a function with this modifier.

Among all functions that might be vulnerable to reentrancy attacks, we identified a possible but unlikely attack vector in the *PledgeContract* of the carbon footprint project.

Auditors have to commit a pledge in crypto to offer their service. This payment is made using the *pledge* function of the *PledgeContract* smart contract. If an auditor is rejected, its pledge is confiscated, as shown in the *onAuditorRejected* function :

```
/** @notice called when an auditor is rejected
    * and is implemented by confiscating the pledge
    */
function onAuditorRejected(address _auditor) internal override {
    confiscatePledge(_auditor);
}
```

The *confiscatePledge* function moves the pledge of the auditor from the *pledgesAmountsByAuditor* variable to the *totalConfiscatedAmount* one.

An allowed user is then able to call the *createTransfer* function to initiate a transfer of confiscated pledge. Initiating a transfer creates a new object containing the amount of pledge to be transferred and it removes this amount from the *totalConfiscatedAmount* variable:

```
function createTransfer(address payable _target, uint256 _amount)
    public
    override
{
    require(
        canSenderOperateTransfer(),
        "not allowed to create a transfer of confiscated pledge"
    );
```

```
    // ensure there are enough confiscated amount
    require(totalConfiscatedAmount >= _amount, "not enough funds");

    TransferTx storage t = transfers[nbTransfers];
    t.index = nbTransfers++;
    t.target = _target;
    t.amount = _amount;
    totalConfiscatedAmount -= _amount;
}
```

If the transfer gets enough votes, the *executeTransfer* function can be called. This function executes the transfer and set it as completed:

```
function executeTransfer(uint256 _index) public override nonReentrant {
    require(
        canSenderOperateTransfer(),
        "not allowed to execute a transfer of confiscated pledge"
    );

    require(_index < nbTransfers, "invalid index");
    TransferTx storage t = transfers[_index];
    require(!t.completed, "cannot execute the transfer");
    // test that the nb of approvals is greater than the number of nodes / 2 +1
    require(hasEnoughVote(t.nbApprovals), "not enough approvals");

    t.target.transfer(t.amount);
    t.completed = true;
}
```

Finally, it is possible to cancel a transfer that has not been completed using the *cancelTransfer* function. This function returns the amount that was to be transferred to the variable *totalConfiscatedAmount*.

VULN-01	Vulnerability: **Reentrency attack on the *Pledge* contract**		
Confirmed	Impact: **Major**	Exploit difficulty: **Difficult**	Severity: **2 / 4** 

Regarding the reentrancy attack, the *executeTransfer* is safe because it uses the *nonReentrant* modifier. However, we identified that if the transfer is made to a malicious contract, this contract could call the *cancelTransfer* function that is not protected by the *nonReentrant* modifier. Because the completed attribute of the transfer is only set after performing the transfer, a call to *cancelTransfer* would reset the *totalConfiscatedAmount* variable to its initial value despite the transfer having been done. This could enable an attacker to drain the contract's funds.

Because of the prerequisites to exploit this flaw, the probability of occurrence is low. Indeed, the transfer mechanism is protected by a voting system, limiting the possibility to automate this attack.

VULN-01	Recommendation: **Use the nonReentrant modifier for every sensitive function**	
Estimated complexity: **Low**	Estimated workload/cost: **Low**	Estimated priority: **3 / 4**

We recommend adding the *nonReentrant* modifier to the *cancelTransfer* function to prevent this attack.

This last solution has been implemented during the audit in the f92c0ba8ac53ae5a01d937ba919d540f6a0bf5f4 commit.

3.2.6 Development practices

3.2.6.1 *Explicit typing*

Even if by default solidity considers the *uint* type as a *uint256*, we recommend using the explicit size for a type.

REM-03	Use an explicit type

While most contracts explicitly use *uint256*, the following contracts and interfaces do not follow this best practice:

- ImprovementProposal.sol
- IAuditorGovernance.sol
- ICarbotFootprint.sol
- IImprovementProposal.sol
- IpledgeContract.sol

4 APPENDIX

4.1 APPROACH PRESENTATION

Below are the main standards used by Intrinsec for the security assessment, in addition to your own standards:

- OWASP framework: web application security;
- ISO/IEC-27001 : security management;
- ISO/IEC-27002 : best practices;
- 40 Essential measures for a healthy network: https://www.ssi.gouv.fr/uploads/2013/01/guide_hygiene_v1-2-1_en.pdf;
- SANS Top 20 critical controls: https://www.sans.org/media/critical-security-controls/critical-controls-poster-2016.pdf;
- Best practices and recommendations from editors and Intrinsec gathered by experience over the years.

4.2 RESULTS PRESENTATION

The content of each section describes the actions taken, the obtained results, our comments regarding these points and the potential exploitation scenarios.

Intrinsec consultants assess **the overall security level** based on the vulnerabilities found on the audited scope.

An audited scope with a very good security level will have a score close to 10/10. On the contrary, an audited scope with a low security level, affected by several critical vulnerabilities will have a score close to 0/10.

The security levels given according to the scores are detailed in the table below:

Overall security level for the audited scope	
	[9-10]/10 – Very good
	[7-9[/10 – Good
	[4-7[/10 – Medium
	[0-4[/10 – Low

	The **positive elements** identified during the assessment (corresponding to either failed attacks or secure configuration) are presented with this green table.

All the vulnerabilities or security issues found on the audited scope are highlighted using another more thorough table. For instance:

VULN-XX	Vulnerability: **[Example] Vulnerability title**		
Confirmed	Impact: **Major**	Exploit difficulty: **High**	Severity: **3 / 4** 

- Title: title of the vulnerability (attack type, protocol, etc.)
- Impact:
  - Minor: no direct consequences on the security of the audited information system
  - Important: few consequences on specific points of the audited information system
  - Major: limited consequences on a portion of the audited information system
  - Critical: extended consequences on the whole audited information system
- Exploit difficulty:
  - Easy: easily exploitable without any particular tool
  - Moderate: the exploitation is easy and possible with only basic skills and publicly available tools
  - High: public vulnerabilities exploitation needing information systems security skills and simple tools development
  - Hard: exploitation of non-public vulnerabilities needing information systems security expertise and specific tools development
- Severity: global note from 1 to 4 taking into account the impact and the exploit difficulty. It is based on the matrix presented in the Annex 4 of the PASSI reference:

Impact \ Exploit difficulty	Hard	High	Moderate	Easy
Minor	Minor (1/4)	Minor (1/4)	Important (2/4)	Major (3/4)
Important	Minor (1/4)	Important (2/4)	Important (2/4)	Major (3/4)
Major	Important (2/4)	Major (3/4)	Major (3/4)	Critical (4/4)
Critical	Important (2/4)	Major (3/4)	Critical (4/4)	Critical (4/4)

To be confirmed: the vulnerability seems to be present but confirmation requires either high privileges on the system or a dangerous exploitation attempt.

Confirmed: the vulnerability is present and has been confirmed through exploitation.

Recommendations associated with identified vulnerabilities are presented in the following way:

VULN-XX	Recommendation: **[Example] Recommendation title**	
Estimated complexity: **Low**	Estimated workload/cost: **Medium**	Estimated priority: **3 / 4**
Recommendation content: actions to do, settings to change, source code to edit, commands to run…		

Complexity and **estimated workload/cost** are meant as guidelines, and based on the knowledge of the scope during the tests. They do not represent, under any circumstances, the exact cost for the correction of one vulnerability.

Priority is calculated according to the estimated severity, complexity and workload/cost for each vulnerability. The higher the priority, the sooner the vulnerability should be fixed.

Remarks are presented in the following way:

REM-XX	[Example] Title

Remarks highlight those aspects not considered as vulnerabilities but which could be improved and, as a result, strengthen the security level of the scope:

- *Confirmation of a suspected issue (e.g. Is the "xx" account legitimate on the application?)*
- *Improvement actions with limited effects (e.g.: Your password policy requires 8 characters with 2 different character sets but a state-of-the-art policy should include 3 sets, etc.)*

Remarks improve the general action plan's readability without losing information.

4.3 RISKS TERMINOLOGY

The tables of risks synthesis provide an analysis aiming to evaluate the risks we identified in terms of impact and probability of likelihood.

Payment process bypassing	
Risk level : Critical	
Impact : Low – Medium – High – **Critical**	**Probability of occurrence :** Minimal – **Significant** – Strong – Maximum
Technical impact (DICT): impact on the payments integrity **Business impact**: direct financial losses	**Exposure factor**: the application is freely accessible on the Internet **Exploitation difficulty**: computer specialist attacker using publicly available tools
Description: By exploiting a flaw in the verification of the data integrity during the order validation, a user can modify the amount of the purchase, without any particular technical skill.	
Associated vulnerability	VULN-XX : Purchasing process bypass

The following terminology is used within the tables:

Risk level:

Low	Sensitive	Critical	Strategic

- Low: the organization will overcome the impacts without problem.
- Sensitive: the organization will overcome the impacts despite some difficulty.
- Critical: the organization will overcome the impacts with significant difficulties.
- Strategic: the organization could not overcome the impacts (its survival is threatened).

Impact:

- Minor: the organization can overcome the impacts without any problem.
- Important: the organization can overcome the impacts despite some difficulty.
- Major: the organization can overcome the impacts with significant difficulties.
- Critical: the survival of the organization is threatened.

Probability of occurrence:

- Minimal: Unknown vulnerability exploitation and limited exposure.
- Significant:
  - Advanced attack techniques with limited exposure.
  - Unknown vulnerability exploitation with medium exposure.
- Strong:
  - Attack techniques accessible to all with medium exposure.
  - Advanced attack techniques with high exposure.
- Maximum: Attack techniques accessible to all with high exposure.

Examples of exposure:

- High exposure: website accessible to anyone on the Internet.
- Medium exposure: authenticated section of a website (without account creation feature), internal collaborator network, CITRIX access, VPN collaborator access.
- Limited exposure: DMZ, isolated network, administration VLAN, dedicated VPN access.

Examples of attacks:

- Attack techniques accessible to all: manipulation of a parameter in a URL, access to network share, etc.
- Advanced attack techniques: SQL injection, Cross-Site Scripting, ARP, brute force, CSRF, etc.