

 IPSPProtocol	 RED ALERT LABS IoT Security	Reference: IPSP.RAL.SC-2024
		Version: 1.0
TARGET OF EVALUATION C.S.P.N		
Page 1 / 40		

Authors	Paul Gédéon, Jean-Loïc Mugnier
Current Version	1.0
Date	23/12/2024

	Sponsor	Evaluator
Organisations	Red Alert Labs as part of the CAMPUS Cyber initiative	RAL
	IPSProtocol as part of the CAMPUS Cyber initiative	IPSProtocol

 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	Page 2 / 40

Contents

- 1. Introduction..... 3**
 - 1.1 Document Goal..... 3**
 - 1.2 Security Target Identification 3**
 - 1.3 Identification of the System to be Evaluated..... 4**
 - 1.4 References 4**
 - 1.5 Definitions and Abbreviations 4**
- 2. Target Description 6**
 - 2.1 General Description of the System to be Evaluated 6**
 - 2.2 Description of the System Internal Execution Environment 9**
 - 2.3 Description of the Users Usage of the System 15**
 - 2.4 Dependency Description..... 20**
 - 2.5 Description of a typical user 21**
 - 2.6 Description of the Evaluation Scope..... 23**
- 3. Description of the technical operating environment..... 26**
 - 3.1 Asset to protect 26**
 - 3.3 Description of Environmental Assumptions..... 28**
 - 3.4 Threats Description 29**
 - 3.5 Security Functions Description 33**
 - 3.6 Coverage Matrix 36**
- Conclusions 40**

Figures

- Figure 1 : Example of Smart Contract Deployment with Truffle..... 14
- Figure 2 : High-Level Workflow of Deployment and Interaction with a Smart Contract..... 16
- Figure 2 : Description of the Total Scope Related to Smart Contract Execution 25
- Figure 3 : Description of the Evaluation Scope 25

 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	Page 3 / 40

1. Introduction

1.1 Document Goal

This document is a draft security target that can be used in the evaluation of products based on smart contracts, following the CSPN methodology outlined by ANSSI. The target is structured according to the description provided in section 4.2 of the [ANSSI_CSPN_CER_P_02] standard.

1.2 Security Target Identification

Smart contracts are publicly available software deployed and replicated across Ethereum client databases that operate within the same network. To mitigate the risk of a single point of failure, Ethereum supports multiple client implementations written in different programming languages, all adhering to the same protocol interface. Prominent examples include Geth (written in Go), Erigon (written in Go/Rust), Nethermind (written in C#), and Besu (written in Java). These diverse clients enhance the resilience and decentralization of the network.

Ethereum clients run the blockchain protocol on nodes hosted on servers or personal computers. Smart contracts, deployed on the Ethereum network, perform a variety of functions, including data storage and manipulation, executing computations, and interacting with other contracts within the blockchain ecosystem. This interoperability is a cornerstone of the decentralized finance (DeFi) and Web3 environments, enabling composability between smart contracts.

Beyond these functions, smart contracts facilitate automation of processes, the creation of crypto assets (such as ERC20 or ERC721 standards), implementation of decentralized governance systems, financial services and many more. These contracts are executed by the Ethereum Virtual Machine (EVM), a runtime environment operating on each node. The EVM executes and validates transactions, ensures the integrity of the network's state, and propagates an updated distributed ledger to peer nodes in the network.

The Ethereum blockchain operates under a consensus mechanism called Proof of Stake (PoS), which replaced Proof of Work in September 2022. PoS enhances the network's energy efficiency and security by using validators who stake Ether to propose and validate new blocks.

Once deployed, smart contracts are immutable, meaning their code and operations cannot be altered. This immutability ensures that their actions are transparently and verifiably recorded on the blockchain ledger, fostering trust, accountability, and reliability in their execution. By design, this transparency underpins the core principles of Ethereum, enabling a secure, decentralized, and tamper-proof ecosystem.

 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	
Page 4 / 40	

1.3 Identification of the System to be Evaluated

Editing Organisation	Ethereum Foundation
Link to Organisation	https://ethereum.foundation/
Sponsor	NA
System Commercial Name	EVM Smart Contract
Evaluation Version	0.8.20
System Category	Blockchain, Ethereum, Crypto asset, Smart Contract

1.4 References

Code	Reference	Name and Source
	ANSSI_CSPN_CER_P_02	Criteria for First-Level Security Certification Evaluation https://www.ssi.gouv.fr/uploads/2015/01/anssi-cspn-cer-p-02-criteres_pour_evaluation_en_vue_d_une_cspn_v4.0.pdf
[1]	ISO/IEC 30107-3	ISO/IEC 30107-3 :2017 <i>Testing and Reporting</i>
[2]	ISO/IEC 30107-1	ISO/IEC 30107-1 :2016 Framework
[3]	CSPN	Certificat de sécurité de premier niveau
[4]	RGPD	Regulation (EU) 2016/679 of the European Parliament and of the Council of April 27, 2016, on the protection of natural persons regarding the processing of personal data and the free movement of such data, repealing Directive 95/46/EC. 27 Avril 2016
[5]	SCSVSv2	https://smartcontractsecurity.eu/smart-contract-security-verification-standard-version-2-scsvsv2/
[6]	Ethereum Guide	https://ethereum.org/en/developers/docs/smart-contracts/security/
[7]	Consensys	https://consensys.github.io/smart-contract-best-practices/
[8]	SWC	https://swcregistry.io/
[9]	Solidity	https://docs.soliditylang.org/en/latest/bugs.html

1.5 Definitions and Abbreviations

Name	Abbrev.	Definition
National Cybersecurity Agency of France (ANSSI)	ANSSI	ANSSI is tasked with defending the information systems of the State and providing advice and support to government administrations and operators of vital importance.

 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	
Page 5 / 40	

Red Alert Labs	RAL	Security Evaluation Laboratory for Connected Objects (IoT)
International Standard Organisation	ISO	International Standardization Organization
Règlement Général de Protection des données	RGPD	[4]
Security Function Requirement	SFR	Security function enabling the implementation of a countermeasure against potential attacks.
Target Of Evaluation	TOE	Product to be evaluated by the laboratory within the scope defined with the client.
Smart Contract	SC	A programmable digital contract that automates the terms and conditions of an agreement between parties without requiring a trusted third party. These contracts are executed on a blockchain, ensuring their immutability and transparency.
EVM Blockchain		A decentralized and distributed state machine that records the history of all transactions carried out on its platform. Data blocks are cryptographically linked to form an uninterrupted chain. Blocks are added to the main chain according to the consensus rules followed by the blockchain's validators.
Decentralized Network		A computing system operating without a central authority controlling transactions and records. Data is managed by a network of distributed nodes in a decentralized manner.
Ethereum	ETH	Ethereum represents an instance of a public blockchain that implements the Proof of Stake consensus algorithm. Smart contracts are written in Solidity, a Turing-complete programming language. These smart contracts are executed within the EVM (Ethereum Virtual Machine).
Virtual Machine	VM	A computing system simulating the environment of a physical machine. It provides abstraction of hardware resources such as memory, processor, and input/output.
Ethereum Virtual Machine	EVM	A virtual machine simulating a computing environment, operating on the decentralized Ethereum network. It provides an abstraction layer for the execution of "smart contracts," playing a central role in automating transactions and implementing business logic on the Ethereum blockchain.
Decentralized Applications	dApps	Applications running on blockchain networks, enabling them to operate without a central authority. They use smart contracts to automate transactions.

 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	
Page 6 / 40	

ERC20 standard	ERC20	A standard on the Ethereum network that defines a common set of rules for tokens.
ERC721 standard	ERC721	A standard on the Ethereum network enabling the creation of non-fungible tokens (NFTs).
Merkle Patricia Trie	Trie	An optimized data structure implemented by Ethereum, known as the Merkle Patricia Trie, is fundamental to the blockchain's efficient storage and retrieval of data. It underpins key components of Ethereum, such as the state, transaction, and receipt tries, enabling fast verification and secure, tamper-proof storage.

2. Target Description

2.1 General Description of the System to be Evaluated

Introduction to the Evaluation of Ethereum Smart Contracts

In the context of a CSPN (Certification de Sécurité de Premier Niveau) evaluation, the focus is on assessing the security and robustness of Ethereum-based smart contracts. These contracts, introduced by the Ethereum Foundation, leverage blockchain technology to execute decentralized, trustless, and transparent services. This evaluation aims to ensure the integrity, security, and functionality of smart contracts, which play a foundational role in Ethereum's ecosystem and its decentralized applications (dApps).

The Foundation of Blockchain Technology

Ethereum builds upon the decentralized ledger innovation pioneered by Bitcoin, as described in the 2009 white paper by "Satoshi Nakamoto." Bitcoin introduced a peer-to-peer payment system that eliminated the need for a central authority, relying instead on a decentralized ledger and consensus algorithm to ensure transaction validity and immutability. Once confirmed, transactions recorded on the Bitcoin blockchain are irreversible, providing a high level of trust and finality.

Ethereum extends this foundational concept, transitioning from a simple value transfer network to a distributed state machine. By introducing the Turing-complete programming language Solidity, Ethereum enables developers to define and execute sophisticated logic on the blockchain through smart contracts. These contracts are run within the Ethereum Virtual Machine (EVM), allowing Ethereum to facilitate not just crypto asset transactions but also decentralized applications and consensus-driven software execution.

Ethereum's Role in Enabling Smart Contract Functionality

Ethereum's programmability allows for the development of a vast range of decentralized applications, offering unparalleled transparency and trust. Unlike traditional Web2 services, where APIs are proprietary

 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	Page 7 / 40

and opaque, smart contracts on Ethereum provide complete visibility into the code being executed. This allows users to independently verify the functionality of a service before using it, eliminating the need to trust third-party intermediaries.

Applications of Smart Contracts

Over time, Ethereum's smart contracts have revolutionized various industries, enabling:

- **Payment Automation:** Secure and trustless recurring payments.
- **Asset Management:** Handling crypto assets through wallets and platforms.
- **Non-Fungible Tokens (NFTs):** Creating unique digital assets.
- **Decentralized Finance (DeFi):** Supporting trading, lending, and staking protocols via decentralized exchanges (DEXs) and other mechanisms.

Challenges with Smart Contracts

Despite their transformative potential, smart contracts introduce unique security challenges due to their:

- **Immutability:** Once deployed, the contract logic cannot be changed.
- **Accessibility:** Smart contracts are publicly accessible 24/7, exposing their transparent code to potential attackers.
- **Lack of Confidentiality:** Unlike traditional systems, smart contracts cannot hide sensitive implementation details.

To mitigate these risks, rigorous security assessments are essential. Techniques such as static code analysis, formal verification, fuzz testing, and audits are employed to safeguard assets managed by smart contracts and to ensure the integrity of the crypto infrastructure they interact with.

Core Smart Contract Types in Ethereum

Ethereum and Solidity provide developers with several foundational types of smart contracts, each designed to promote modularity, reusability, and adherence to software engineering best practices. These contract types facilitate efficient development while reducing the overall costs associated with deploying new contracts.

1. Abstract Contracts

Abstract contracts serve as templates that define the structure and required functions for other contracts but do not implement any functionality themselves. They are typically used as a base for inheritance, encouraging consistent and standardized implementations.

2. Interfaces

Interfaces are contracts that specify function signatures without providing their implementations. This allows contracts to communicate with one another while ensuring compatibility and consistency. Interfaces are particularly valuable for creating interoperable systems, such as token standards like ERC-20 and ERC-721.

 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	Page 8 / 40

3. *Standard Contracts*

These are fully implemented contracts ready for deployment. They encapsulate complete logic and functionality, ranging from simple payment contracts to complex financial protocols.

4. *Libraries*

Libraries are a specialized type of contract used to store reusable code. They contain pure functions that do not modify state variables or interact with Ether balances, making them ideal for utility operations such as mathematical calculations or cryptographic hashing. By reusing library functions, developers can minimize deployment costs and optimize contract functionality.

Functional Perspective on Smart Contracts

From a functional standpoint, smart contracts are used to create digital representations of assets, such as utility tokens, non-fungible tokens (NFTs), and other programmable financial instruments. These assets often act as digital twins, mirroring real-world financial products and enabling new applications in decentralized environments.

1. *Representing Crypto Assets*

- **Utility Tokens:** Representing access rights or platform-specific functions.
- **Non-Fungible Tokens (NFTs):** Representing unique assets, such as artwork, collectibles, or virtual property.

2. *Complex Financial Products*

As Ethereum evolves, smart contracts are increasingly used to define mechanisms for sophisticated financial instruments, such as bonds, stocks, and derivatives. These implementations allow financial products to be transparently managed and executed on the blockchain.


3. *Decentralized Services*

Smart contracts enable decentralized services that interact with these digital representations. For instance:

Automated market makers (AMMs) facilitate decentralized exchanges.

Lending platforms use contracts to manage loan terms and collateral.

The key advantage of decentralized services lies in their transparency and trustlessness. Smart contracts not only execute services but also allow users to verify how these services work, as all logic is publicly accessible. This contrasts sharply with traditional financial services delivered via opaque Web2 APIs, where users often have limited visibility or control over the processes governing their assets.

 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	Page 9 / 40

Advantages of Decentralized Smart Contracts

1. **Transparency:** Users can audit and verify the contract code before engaging, ensuring trust in the system.
2. **Immutability:** Once deployed, the core logic cannot be altered, reducing the risk of tampering.
3. **Consensus-Driven Operation:** Smart contracts enable decentralized consensus on the execution of services, eliminating reliance on centralized entities.
4. **Open Access:** Anyone can interact with Ethereum-based contracts, democratizing access to financial and digital services.

2.2 Description of the System Internal Execution Environment

Ethereum, as a decentralized blockchain platform, operates a robust system that enables the deployment, execution, and interaction of smart contracts. This environment is supported by multiple components, including the Ethereum Virtual Machine (EVM), the Solidity programming language, and the gas mechanism. Together, these elements create a secure and efficient execution layer, enabling decentralized applications (dApps) and smart contracts to function reliably across a global network of nodes. Below is a detailed exploration of the critical components of Ethereum's internal execution environment.

Solidity

Solidity is a Turing-complete, high-level programming language specifically designed for developing smart contracts on Ethereum. It allows developers to define the contract's logic using constructs such as loops, arrays, dictionaries, and conditional clauses. Once written, Solidity code is compiled into bytecode—a low-level, machine-readable format executed by the EVM.

The compilation process is deterministic, meaning the same Solidity code will always produce the same bytecode. This property is essential for security and trust, as it enables users to verify that the deployed bytecode accurately represents the developer's original code. This verification process is particularly important for blockchain explorers, which provide a human-readable representation of deployed contracts to ensure transparency.

The Application Binary Interface (ABI)

The ABI defines the contract's interface, detailing its function signatures, variables, and events in a standardized JSON format. This interface allows users, applications, and other smart contracts to interact with the contract seamlessly. Key points include:

- **Shared Usage:** The ABI and the contract's address must be shared with users to enable interaction.
- **Blockchain Explorers:** Tools like Etherscan verify a contract's source code by matching it to the on-chain bytecode, thanks to Solidity's deterministic compilation.

 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	Page 10 / 40

Transactions and Contract Deployment

Smart contracts are deployed to the blockchain via Ethereum transactions. The deployment transaction carries the compiled bytecode to the blockchain, where it is validated by the consensus mechanism. Upon successful validation, the contract is assigned a unique address, which enables programmatic interactions by users and applications.

Each participating Ethereum node stores a copy of the contract's bytecode in its local database, ensuring the code remains accessible across the network. Additionally, the contract's hash (or fingerprint) is stored in the state trie, a critical data structure in Ethereum that maintains the network's consensus and transaction integrity.

Contract State and Storage

The state of a smart contract, including its variable values and balances, is stored within a structured and tightly integrated architecture that ensures persistence, immutability, and efficient execution. Ethereum relies on key components such as the Ethereum Virtual Machine (EVM), Solidity, and the gas mechanism to manage computational resources, enabling smart contracts to operate seamlessly in a decentralized environment.

Smart contract data is stored in a modified Merkle Patricia Trie, commonly referred to as the storage trie, which is unique to each contract. This trie structure ensures data integrity and facilitates efficient data access. The storage trie is linked to the state trie, a higher-level structure that tracks the overall state of all accounts and contracts on the blockchain. Any changes to a smart contract's state, such as updates to variable values or balances, are validated through the consensus process when new blocks are added to the blockchain. This ensures consistency, transparency, and the immutability of the blockchain state.

In addition to the storage trie, smart contracts also rely on the account trie. The bytecode of the smart contract and its state variables are stored within this trie, but they are maintained in separate sections to ensure clarity and streamlined access. Because the account trie is directly linked to the blockchain, all stored data inherits the immutability of the blockchain. However, operations involving persistent storage incur higher gas costs due to the permanence of the data.

For intermediate computations, the EVM provides temporary memory, which is used exclusively during transaction execution. Unlike persistent storage, temporary memory is cleared immediately after the transaction concludes and is less gas-intensive. This transient nature makes it ideal for short-term processing without affecting the contract's permanent state.

State modifications in a smart contract occur exclusively within the EVM during transaction execution. These modifications are provisional until the transaction successfully concludes. Once finalized, the updated state is propagated to the blockchain's database and integrated into the account trie, ensuring that the changes are synchronized across all Ethereum nodes.

 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	Page 11 / 40

Smart contracts also emit events to signal significant state changes or actions. These events are recorded in transaction logs, which are stored separately from the contract's state. Events provide an efficient mechanism for external applications to track and respond to blockchain activity without requiring continuous queries. They enhance integration with external systems, such as decentralized applications (dApps) and blockchain explorers, by providing structured and accessible on-chain data.

Smart contracts utilize efficient data structures, including mappings for key-value storage and arrays for sequential data management. These structures are indexed within the account trie for optimized access and retrieval. Transaction logs, which detail execution outcomes and emitted events, further facilitate seamless interaction with external systems.

This structured approach to contract state and storage management ensures that smart contracts maintain consistency, transparency, and reliability within the decentralized Ethereum ecosystem. By integrating robust storage mechanisms with the execution environment of the EVM, Ethereum smart contracts provide a secure and efficient framework for decentralized applications.

The Ethereum Virtual Machine (EVM)

The EVM is a stack-based virtual machine designed to execute the bytecode generated from Solidity or other high-level languages. It operates as a decentralized execution environment, ensuring consistent behavior across all Ethereum nodes.

The EVM uses a 256-bit stack for computation and includes temporary memory that is not persisted across transactions. Every operation (opcode) executed by the EVM has an associated gas cost, which fluctuates based on network demand. This mechanism ensures fairness and prevents abuse, as more complex operations consume more resources and are therefore more expensive to execute.

Successful transactions processed by the EVM result in state changes that are stored in the blockchain, while failed or reverted transactions leave the state unchanged. This robust execution model ensures the reliability and integrity of smart contract interactions.

EVM Context

The EVM enables seamless interaction with smart contracts, allowing users to retrieve variable values or execute functions. While accessing a variable's value is straightforward and free, modifying a smart contract's state requires a transaction, which incurs a gas fee.

Smart contracts also have access to built-in functions provided by Solidity and the EVM to manage execution data that isn't explicitly part of the contract. These include:

- msg.sender: The address of the entity interacting with the smart contract.
- msg.data: The raw input data sent with the transaction or call.
- msg.value: The amount of Ether sent with the transaction.
- msg.block: The current block number.

 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	Page 12 / 40

These built-in functions provide developers with relevant execution context, enabling efficient and secure contract behavior. For example, `msg.sender` is commonly used to verify user permissions, while `msg.value` facilitates payment functionalities within the contract.

Gas

Gas is a fundamental component of the Ethereum ecosystem that ensures security and incentivizes validator participation. Every EVM operation (opcode) has a specific gas cost, requiring users to pay for the computational resources consumed when executing smart contract code.

Gas serves two main purposes:

Mitigating Abuse: By requiring payment for every computation, gas discourages malicious activities, such as infinite loops or denial-of-service (DoS) attacks.

Resource Monetization: Validators are incentivized to maintain the network, as they are compensated with transaction fees and block rewards.

Since the introduction of EIP-1559, transaction fees are split into two components:

Base Fee: Burned and permanently removed from circulation.

Tip Fee: Rewarded to validators as an incentive for including transactions in a block.

This mechanism aligns costs with computational demand, ensuring fairness for users while maintaining network stability and sustainability.

Function Selector in Solidity and the EVM

Function selectors are essential for linking function calls in Solidity smart contracts to their corresponding execution in the EVM. When a contract is compiled, its source code is transformed into bytecode, and the function selector is derived from the first 4 bytes of the Keccak-256 hash of the function signature.

For example, the function `transfer(address,uint256)` generates the hash `a9059cbb2ab09eb219583f4a59a5d0623ade346d962bcd4e46b11da047c9049b`, and its selector is `0xa9059cbb`. This selector uniquely identifies the function to be executed.

While Solidity's compiler prevents selector collisions within the same contract, issues can arise in complex systems, such as multi-contract dApps or proxy-based architectures. Selector collisions—where two functions generate the same selector—can lead to unintended behaviors, such as incorrect function execution.

Proper design and thorough testing are crucial to mitigate these risks and ensure that function selectors work as intended.

 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	Page 13 / 40

Function Selector Collisions

Function selector collisions occur when two distinct function signatures produce the same 4-byte selector. For example:

```
workMyDirefulOwner(uint256,uint256)
```

```
join_tg_invmru_haha_fd06787(address,bool)
```

Both generate the same selector, despite having entirely different functionalities. If such a collision occurs, unintended functions may be called, resulting in deviations from the expected behavior. This is particularly problematic when critical functions, like `transfer(address,uint256)`, are involved.

To avoid collisions, developers must carefully design and audit smart contracts, particularly in multi-contract systems or proxy environments. Using unique function names and ensuring proper testing can prevent vulnerabilities associated with selector collisions.

Deployment

Smart contracts, once compiled, are deployed to the blockchain through an Ethereum transaction. During deployment, the contract's bytecode is stored on the blockchain and assigned a unique address, enabling users and applications to interact with the contract.

Deployment includes an initialization phase, where constructors define key parameters such as roles, balances, and immutable variables. These configurations establish the contract's internal environment. Once deployed, the bytecode becomes immutable, securing the contract's logic from modification.

Despite this immutability, the contract's state can still be updated if functions explicitly allow such changes. For example, administrative functions might enable parameter adjustments or state updates.

Smart contracts can be deployed manually via web applications like Remix (an online IDE) or programmatically using developer tools. In either case, the deploying wallet must have sufficient gas tokens, such as Ether (ETH), to cover deployment costs.

It's critical to ensure that the wallet used for deployment is secure, as it not only pays the gas fees but may also hold elevated privileges within the contract. If the private key is compromised, attackers can misuse these privileges, leading to security breaches. Verifying the contract's bytecode against the source code using tools like Sourcify or Etherscan further ensures trust in the deployed contract.

	Reference: IPSP.RAL.SC-2024
	Version: 1.0
	
TARGET OF EVALUATION C.S.P.N	Page 14 / 40

```

> npx truffle deploy --network goerli

Compiling your contracts...
=====
> Compiling @openzeppelin\contracts\interfaces\IERC165.sol
> Compiling @openzeppelin\contracts\token\ERC20\ERC20.sol
> Compiling @openzeppelin\contracts\token\ERC20\IERC20.sol
> Compiling @openzeppelin\contracts\token\ERC20\extensions\IERC20Metadata.sol
> Compiling @openzeppelin\contracts\utils\Context.sol
> Compiling @openzeppelin\contracts\utils\introspection\IERC165.sol
> Compiling .\contracts\GFICrypto.sol
> Compiling .\contracts\Math.sol
> Compiling .\contracts\interfaces\IBurnRedeemable.sol
> Compiling .\contracts\interfaces\IBurnableToken.sol
> Compiling .\contracts\interfaces\IRankedMintingToken.sol
> Compiling .\contracts\interfaces\IStakingToken.sol
> Compiling .\contracts\test\BadBurner.sol
> Compiling .\contracts\test\Burner.sol
> Compiling .\contracts\test\GFICryptoRank100001.sol
> Compiling .\contracts\test\GFICryptoRank25mm1.sol
> Compiling .\contracts\test\GFICryptoRank5001.sol
> Compiling .\contracts\test\RevertingBurner.sol
> Compiling abdk-libraries-solidity\ABDKMath64x64.sol
> Artifacts written to C:\Users\paulg\Downloads\GFI-Crypto\build\contracts
> Compiled successfully using:
   = solc: 0.8.17+commit.8df45f5f, Emscripten, clang

```

Figure 1 : Example of Smart Contract Deployment with Truffle

 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	Page 15 / 40

2.3 Description of the Users Usage of the System

Smart Contracts: The Foundation of Web3

Smart contracts are the cornerstone of the Web3 ecosystem. Unlike traditional Web2 applications hosted on private servers, smart contracts are decentralized software deployed and stored on the blockchain. On public blockchains such as Ethereum, smart contracts are publicly accessible and executed within the Ethereum Virtual Machine (EVM). This separation of execution environments provides a secure and isolated sandbox for running smart contract logic while ensuring immutability and transparency.

Smart Contract Deployment and Confirmation Process

The creation, deployment, and confirmation of a smart contract on the Ethereum blockchain follow a well-defined process designed to ensure reliability, consistency, and accessibility across the decentralized network. This process involves the following key stages:

Smart Contract Development

The process begins with the developer writing the smart contract code using a specialized programming language such as Solidity. This code encapsulates the business logic and rules that the smart contract will enforce. It defines the contract's behavior and includes the functions and conditions that will be automatically executed in response to user interactions. Ensuring the accuracy and security of this code is critical, as the deployed contract will be immutable.

Deployment of the Smart Contract

The deployment phase submits the compiled smart contract to the blockchain. This involves creating a transaction containing the contract's bytecode and any required initialization data. The transaction is broadcast to the Ethereum network, where it is processed by validators (in a Proof-of-Stake system). Validators group the deployment transaction with other pending transactions, execute the contract's initialization logic, and update the blockchain state. The new block containing the following elements is then constructed:

- The transaction data, including the smart contract's bytecode.
- The updated state reflecting the deployment.
- Metadata, such as the hash of the previous block and the Merkle root of the included transactions.

The deployment process assigns the smart contract a unique address, enabling users and applications to interact with it.

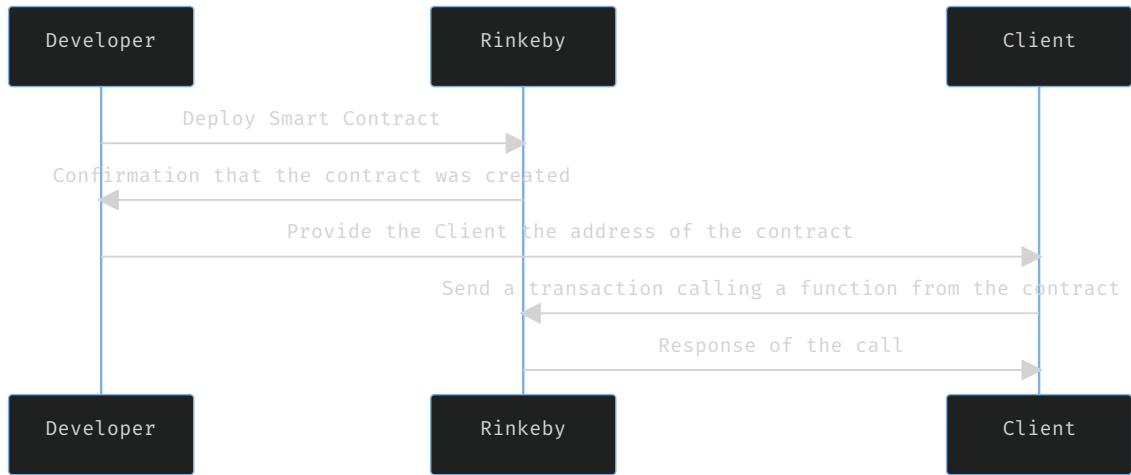


Figure 2 : High-Level Workflow of Deployment and Interaction with a Smart Contract

However, it is also possible to deploy smart contracts using another smart contract, commonly referred to as a "Factory." A Factory contract enables the deployment of multiple instances of the same smart contract template, streamlining the process for creating and deploying similar services. In such cases, smart contracts often utilize the **CREATE2** opcode, which provides a predictable contract address based on specific parameters, even before deployment.

Block Propagation

Once the block containing the deployed smart contract is created, it is propagated throughout the Ethereum network. The validator responsible for the block sends it to neighboring nodes, which validate its contents. If the block is valid, each node updates its local copy of the blockchain and propagates the block to other nodes. This propagation continues until a majority, ideally all, network nodes have validated and integrated the block into their local copies of the blockchain. At this stage, the smart contract becomes accessible and ready for interaction.

Confirmation of Deployment

Although the block containing the smart contract is added to the blockchain, additional confirmations are typically required to reduce the risk of blockchain reorganization. These confirmations involve subsequent blocks being added to the chain, solidifying the position of the block containing the smart contract. A smart contract is only considered fully deployed and reliable for use after achieving the required number of confirmations, ensuring its permanence and integrity.

 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	Page 17 / 40

Interaction with the Smart Contract

After confirmation, a smart contract becomes available for interaction. Users and web applications can send transactions to invoke the contract's functions. These transactions are processed by the Ethereum client and executed within the Ethereum Virtual Machine (EVM), which interprets the deployed logic defined in the contract. The interaction varies depending on the type of operation:

- **Read Operations:**

These interactions query the contract's state without modifying it. As they do not require consensus or alter the blockchain, they are free of gas costs and are typically faster than write operations.

- **Write Operations:**

These interactions modify the contract's state and require the user to pay gas fees. The state changes are processed within the EVM and then propagated across all nodes in the Ethereum network, ensuring consistency and immutability.

State changes resulting from interactions are validated by the network and included in the blockchain. Events emitted during these interactions enable external applications to track significant changes or updates in the contract's state. This provides a seamless interface for monitoring and responding to on-chain activity.

Permissionless Composability

Smart contracts can call functions in other contracts, a feature known as permissionless composability. This capability allows developers to integrate existing contracts and protocols into new applications without requiring off-chain authorization. It is a key driver of the rapid growth and innovation in the decentralized ecosystem, enabling modular and interoperable dApps.

Error Handling

Handling unexpected scenarios is critical to ensuring robust contract execution. Developers can implement error-handling mechanisms such as:

- **require:** Verifies that certain conditions are met before continuing execution. If the condition fails, the transaction is halted.
- **revert:** Rolls back the current transaction and undoes any state changes if an unexpected condition occurs.
- **try/catch:** Used when interacting with external smart contracts. This ensures that even if an external call fails, the remaining execution of the transaction can continue as intended.

These mechanisms help maintain the integrity and predictability of smart contract operations.

 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	Page 18 / 40

Events

Smart contracts can emit events, which are structured logs defined by an event name and associated parameters. Events are stored in the blockchain within the transaction trie and are validated through the consensus process. These logs allow external applications and decentralized systems to monitor blockchain activity efficiently.

Events provide a mechanism for external systems to track specific actions or changes in the contract's state. They enhance the transparency and usability of the Ethereum ecosystem by enabling dApps to react to on-chain activity in real time, without the need for continuous querying of the blockchain.

The Ethereum Virtual Machine (EVM)

The EVM is a decentralized computation environment specifically designed to execute smart contract bytecode. It operates as a stack machine with the following characteristics:

256-Bit Words: Optimized for cryptographic operations, such as Keccak-256 hashing and secp256k1 signatures.

Stack Depth: Supports a depth of 1024 items, ensuring sufficient space for complex computations.

Temporary Memory: Used during transaction execution but discarded afterward, keeping it lightweight.

Development Tools and Libraries

Web3 Libraries:

To interact with smart contracts, developers use Web3 libraries tailored to their programming language of choice. These libraries provide a standardized interface for managing accounts, deploying contracts, and sending transactions. Popular libraries include:

- **web3.js:** For JavaScript/TypeScript.
- **web3.py:** For Python.
- **web3j:** For Java.

Frameworks:

Development frameworks extend the capabilities of Web3 libraries, offering enhanced developer environments with features such as multi-chain setups, blockchain explorer integration, and debugging tools.

- **Hardhat:** A modern framework for JavaScript and TypeScript, replacing Truffle.
- **Apeworx:** A Python-based framework, succeeding Brownie.
- **Foundry:** Rust-based and highly innovative, enabling Solidity-native testing and development.

 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	Page 19 / 40

These tools streamline the development process, allowing for modular and efficient contract creation.

Blockchain Types

Smart contracts can be deployed on three types of blockchains, each catering to different needs:

1. **Public Blockchains:** Open and transparent, allowing unrestricted participation and interaction (e.g., Ethereum mainnet).
2. **Private Blockchains:** Restricted to authorized participants, offering enhanced privacy and security for enterprise use cases.
3. **Consortium Blockchains:** Governed by multiple organizations, balancing decentralization with controlled access.

Although the deployment process is consistent across these environments, public blockchains emphasize openness and decentralization, aligning with the ethos of Web3.

Smart Contracts: The Backbone of Decentralized Applications

1. **Asset Representation:**

Smart contracts with well-defined and encapsulated logic can represent digital assets and manage all associated features and functions. For example:

- a. **ERC-20 Tokens:** Represent fungible assets, such as cryptocurrencies or utility tokens.
- ERC-721 Tokens:** Represent unique, non-fungible assets like NFTs (Non-Fungible Tokens).

These standards enable consistent functionality for assets, making them interoperable across various decentralized applications.

2. **Composability:**

Smart contracts are inherently public and accessible on a permissionless blockchain, enabling seamless integration and interoperability. Developers can build upon existing smart contracts and services without requiring off-chain authorization, provided the existing code permits such interactions. This composability fosters innovation and allows decentralized applications to leverage each other's functionalities, creating an interconnected ecosystem.

 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	Page 20 / 40

2.4 Dependency Description

Smart contracts rely on several core dependencies to function correctly and interact effectively with the blockchain environment. These dependencies apply universally to all smart contracts and are foundational to their proper operation and security.

Core Dependencies

1. Wallets

Wallets play a critical role in signing transactions that interact with smart contracts. They securely manage private keys, which are required to authorize interactions and deployments. The integrity and security of wallet software directly impact the safety and reliability of the entire smart contract ecosystem.

2. Ethereum Virtual Machine (EVM)

The EVM serves as the execution environment for smart contracts. It interprets the bytecode and ensures decentralized execution across the network. The security and correctness of the EVM are foundational to the trustworthiness of smart contracts, as it governs their behavior and state changes.

3. Solidity Compiler (Solc) and Version

The Solidity compiler converts human-readable Solidity code into bytecode that the EVM can execute. The choice of the compiler version affects compatibility and security. Using an outdated or untested compiler version can introduce vulnerabilities, making careful version management essential for secure development.

4. Smart Contract Programming Language

A language like Solidity, or any other that compiles to EVM-compatible bytecode, is crucial for creating smart contracts. The choice of programming language significantly influences the flexibility, expressiveness, and security of the contract's implementation. For example, vulnerabilities in Vyper's reentrancy protection mechanisms previously allowed hackers to exploit the Curve protocol, highlighting the importance of robust and well-tested language features for ensuring contract security.

5. Blockchain and RPC Nodes

Blockchain nodes are essential for interacting with the blockchain. They enable querying the state, submitting transactions, and receiving network updates. Examples include Ethereum clients like Geth and Besu or hosted services such as Infura, which act as intermediaries between wallets, applications, and the blockchain.

 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	Page 21 / 40

6. Blockchain Data Structures (Account Trie and Storage Trie)

The account trie stores the contract's bytecode and balances, while the storage trie holds the state variables unique to the contract. These data structures are vital for ensuring immutability, integrity, and efficient access to smart contract data. Changes to these structures could significantly impact smart contract functionality.

5. Gas Fees and Gas Token (Ether)

Gas fees, paid in the network's native cryptocurrency (e.g., Ether for Ethereum), are required to execute transactions, including deploying and interacting with smart contracts. Smart contracts also have the capability to manage the gas they provide to external contracts during side calls. This introduces a potential vulnerability: if the gas costs for specific opcodes evolve, some applications may fail to execute properly, rendering them unavailable.

Efficient gas management within smart contracts is crucial, as it directly impacts their accessibility, scalability, and overall performance. Although gas fees have been significantly reduced in some ecosystems, proper handling of gas allocation and usage remains essential to ensure the reliability and sustainability of smart contract-based applications.

6. Consensus Mechanism

The consensus mechanism (e.g., Proof of Stake) underpins the reliability and immutability of the blockchain. It ensures that state changes resulting from smart contract interactions are validated and finalized across the network. This process is directly related to the blockchain's capacity to prevent reorganizations. A strong consensus mechanism minimizes the risk of reorganizations, thereby enabling finality, consistency, and security within the environment, which are critical for the proper functioning of smart contracts.

2.5 Description of a typical user

1. Core/Protocol Developers (or Base Layer Developers):

These personnel are responsible for the architecture, development, and maintenance of the blockchain's core protocol. They operate on the foundational source code of the underlying blockchain technology (e.g., Ethereum, Solana, Cosmos). Their work is fundamental, establishing the operational rules and constraints of the entire blockchain, which consequently directly influences the capabilities and limitations of all deployed smart contracts.

 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	Page 22 / 40

Key Responsibilities:

- **Protocol Code Development and Maintenance:** This includes the design, implementation, testing, and updating of the code governing consensus mechanisms, transaction processing, network security protocols, and other core functionalities.
- **Integration of Precompiled Contracts with the EVM (or Equivalent):** They are responsible for seamlessly integrating these precompiled contracts into the execution environment, ensuring correct invocation and interaction with other parts of the system.
- **Protocol Enhancement Implementation (Upgrades):** This involves proposing, developing, and deploying protocol upgrades to enhance scalability, security, introduce new features, or address identified issues.
- **Critical Issue and Defect Resolution:** These personnel are responsible for responding to and resolving critical network incidents, such as attacks or forks, to maintain service continuity and network integrity.
- **Research and Development:** This includes investigating and evaluating emerging technologies and methodologies to improve the blockchain's performance, security posture, and overall functionality.

Essential Competencies:

- Proficiency in low-level programming languages (e.g., C++, Go, Rust) suitable for performance-critical systems.
- Comprehensive understanding of blockchain principles, including consensus algorithms, cryptographic techniques, and distributed networking concepts.
- Knowledge of protocol-level security considerations, including potential vulnerabilities and common attack vectors.
- Proven ability to contribute to complex, distributed software development projects, including collaborative coding practices and version control methodologies.

2. Smart Contract Developers:

These personnel develop the business logic of decentralized applications (DApps) by designing, implementing, testing, and deploying smart contracts on an existing blockchain platform.

Key Responsibilities:

- **Smart Contract Design and Development:** This encompasses translating functional and security requirements into executable code using languages such as Solidity, Vyper, or Move.
- **Integration with External Services:** This includes integrating and leveraging external services from third-party providers, which are not under the control of the developing entity. Interactions can involve both smart contracts developed and controlled by the entity and those developed by external parties.
- **Unit and Integration Testing:** Rigorous testing is conducted to verify the correct functionality and security properties of the smart contracts.
- **Code Optimization:** This involves optimizing code for efficiency and minimizing resource consumption (e.g., gas costs on Ethereum).

 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	Page 23 / 40

- **Smart Contract Security Implementation:** Adherence to secure coding practices and implementation of security mechanisms to mitigate potential vulnerabilities.

Essential Competencies:

- Expertise in smart contract programming languages (e.g., Solidity, Vyper, Move), including syntax, semantics, and best practices.
- Thorough understanding of the target execution environment (e.g., Ethereum Virtual Machine (EVM)) and its specific constraints and capabilities, such as gas management and computational limits.
- Knowledge of common smart contract vulnerabilities and effective mitigation strategies.

3. Front-End Developers (DApp):

These personnel develop the user interfaces (DApps) that facilitate user interaction with deployed smart contracts.

Key Responsibilities:

- **User Interface Development:** This includes designing and implementing intuitive and user-friendly interfaces for interacting with smart contract functionalities.
- **Smart Contract Integration:** This involves establishing secure and reliable communication channels with deployed smart contracts using libraries such as Web3.js or Ethers.js. Interactions may occur with both internally and externally developed smart contracts, necessitating appropriate trust assumptions regarding data integrity and origin.
- **User Experience (UX) Management:** This includes optimizing the application's usability, performance, and accessibility.
- **User Interaction Security:** Implementing security measures to protect against client-side attacks and ensure the integrity of user interactions.

Essential Competencies:

- Proficiency in web development technologies, including HTML, CSS, and JavaScript.
- Familiarity with blockchain interaction libraries (e.g., Web3.js, Ethers.js).
- Understanding of web application security principles, including prevention of cross-site scripting (XSS) and cross-site request forgery (CSRF) attacks.

2.6 Description of the Evaluation Scope

The TOE comprises Solidity smart contracts, with a primary focus on those compiled with version 0.8.0 of the Solidity compiler. This version is chosen to ensure relevance for both newly developed contracts and upgrades leveraging recent EVM advancements. However, the evaluation will also consider general security principles and vulnerabilities applicable across multiple compiler versions. This approach provides a comprehensive assessment addressing both specific and general security concerns.

 	Reference: IPSP.RAL.SC-2024 Version: 1.0
TARGET OF EVALUATION C.S.P.N	Page 24 / 40

2.6.1 Evaluation Focus and Exclusions

The evaluation prioritizes the following aspects:

- **ERC-20 Implementation:** This evaluation heavily focuses on the implementation of the ERC-20 token standard within the smart contracts. The analysis will delve into:
 - **Conformance to Standards:** Verification of the contract's adherence to the ERC-20 specifications, including the presence and correct implementation of mandatory functions, events, and data structures.
 - **Security Implications:** Examination of potential security vulnerabilities associated with the ERC-20 implementation. This includes considering known ERC-20 vulnerabilities and assessing whether the evaluated contracts have implemented appropriate mitigations.
 - **Core ERC-20 Functions:** Detailed analysis of the correct implementation of ERC-20 functions like transfer, balanceOf, allowance, approve, and transferFrom.
- **General Security Principles:** In addition to the ERC-20 focus, the evaluation will address broader security principles and known vulnerabilities applicable to smart contracts in general (e.g., reentrancy attacks, integer overflows/underflows).
- **Smart Contract Code:** Analysis of the Solidity source code and its compiled bytecode, including business logic, control flow, data structures, and security mechanisms implemented within the contract.

Exclusions from the Evaluation:

The evaluation excludes the following:

- **EVM Internals:** The internal workings of the EVM, including opcodes, gas metering mechanisms, and execution model, are outside the scope. These aspects are considered part of the underlying platform and are assumed to function correctly.
- **Ethereum Client Code:** The analysis will not cover the Ethereum client code responsible for managing smart contract data within different tries (state, storage, code, and receipt tries). These are part of the Ethereum client implementation and are assumed to be secure.
- **Network-Level Security:** Security aspects related to the Ethereum network itself, such as consensus mechanisms, network protocols, and node security, are outside the scope.
- **Source Code Management:** The management and storage of the smart contract's source code (e.g., using Git repositories) are not part of this evaluation. The focus is on the deployed bytecode.

ERC-20 Standard Details

The ERC-20 token standard defines a common interface for fungible tokens, enabling easy exchange and integration with other applications. This evaluation will thoroughly analyze the implementation of the ERC-20 standard within the smart contracts.

Out of Scope for ERC Evaluation

The evaluation will not cover the following aspects related to ERCs:

- **ERC Proposal Process:** The evaluation will not assess the validity or appropriateness of the ERC-20 standard itself. The focus is on the implementation of the existing, accepted standard, not on the process by which it was created or approved.
- Future ERCs: The evaluation will only consider the finalized and widely adopted ERC-20 standard at the time of the evaluation. New or draft ERCs will not be included.

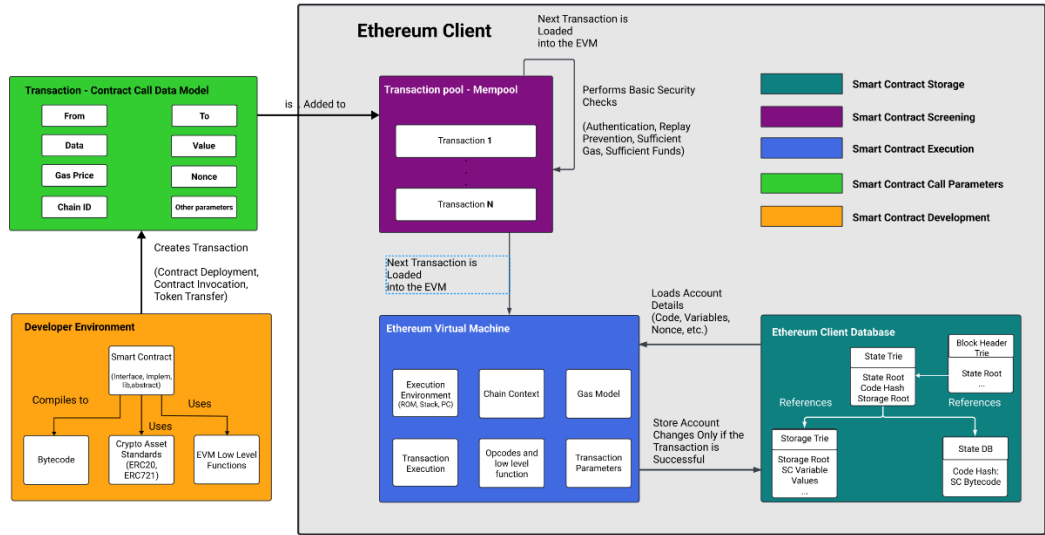


Figure 3 : Description of the Total Scope Related to Smart Contract Execution

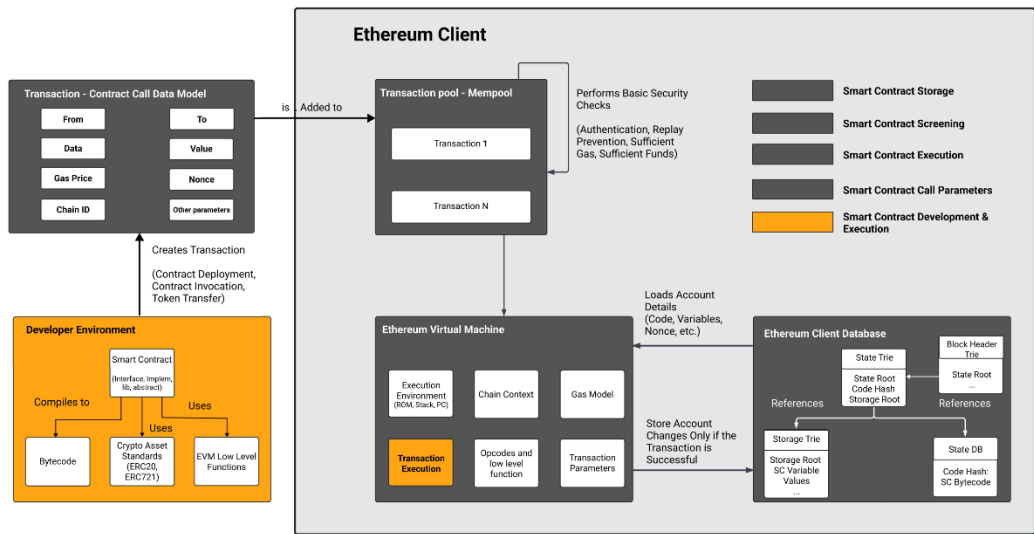


Figure 4 : Description of the Evaluation Scope

 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	Page 26 / 40

3. Description of the technical operating environment

3.1 Asset to protect

Core Smart Contract Assets (Common to All Contracts)

These assets are present in virtually all smart contracts, regardless of the specific application or type of crypto asset they manage:

- **Contract Bytecode:** The deployed bytecode represents the executable logic of the contract. Its integrity is paramount to ensure the contract functions as intended. Any unauthorized modification of the bytecode could lead to unexpected behavior, vulnerabilities, or loss of funds.
 - **Security Objective: Integrity:** The bytecode must not be modified after deployment except through authorized upgrade mechanisms (if any).
- **Contract State Data:** The contract's state, stored as variables on the blockchain, is a critical asset. This data reflects the current condition of the contract and its managed assets. Protected state data includes:
 - **Contract Variables:** Variables that store information relevant to the contract's operation, such as ownership information, configuration parameters, and application-specific data.
 - **Security Objectives: Integrity:** Contract variables must not be modified by unauthorized actors.
 - **Mappings and Data Structures:** Complex data structures used within the contract to store and manage information, such as user balances, lists of participants, or auction details.
 - **Security Objectives: Integrity:** These data structures must not be corrupted or Integrity: These data structures must remain secure and free from corruption or manipulation by unauthorized actors.
 - **Availability:** It is crucial that these data structures scale effectively with the usage of the protocol. Gas consumption should remain manageable to avoid making the dApps unfeasible or impractical for users.
- **Event Logs:** Event logs emitted by the smart contract are considered assets. These logs provide an auditable record of the contract's activity and are essential for tracking transactions and state changes. Their integrity is important for accountability and dispute resolution.
 - **Security Objective: Integrity:** Event logs must accurately reflect the actions performed by the contract and must not be forgeable or modifiable.

 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	
Page 27 / 40	

Asset	Storage / Access Location	Confidentiality	Integrity	Authenticity
A.1. Native Token	Smart Contract / EOA	No	Yes	Yes
A2. Contract Code in Bytecode	Account Trie / EVM	No	Yes	Yes
A3. Contract Data	Account Trie / EVM	No	Yes	Yes
A4. User Public Key	Transaction Trie (tx sig)	No	Yes	Yes
A5. User Private Key	Secure Wallet	Yes	Yes	Yes
A6. Source Code	Privat Repository e.g. GitHub)	Yes	Yes	Yes
A7. Events	Transaction Trie / EVM	No	Yes	No
A8. Gas	EVM	No	No	No

Analyzing the Specificities of ERC20

Assets Associated with Functional Aspects

This section identifies the business-oriented assets relevant to an ERC-20 utility token. To ensure a robust and practical evaluation, the analysis will be based on the ERC-20 interface defined by the Ethereum Foundation and its widely adopted implementation provided by OpenZeppelin.

Rationale for OpenZeppelin Implementation:

The OpenZeppelin implementation serves as a de facto standard within the Ethereum ecosystem due to its widespread adoption. Notably, the OpenZeppelin contracts have undergone audits by the security firms: Levelk, Alchemy, and Certora (which performed formal verification using their Certora tool). You can find detailed information about these audits in the OpenZeppelin repository: [link to OpenZeppelin audits](#).

Scope of Evaluation :

While OpenZeppelin offers additional contracts for functionalities like access control, proxies, and governance, this evaluation focuses solely on the core ERC-20 functionality and its associated assets.

Maturity of OpenZeppelin Contracts:

Before proceeding with the analysis of more complex contracts, it is essential to evaluate the maturity of the core ERC-20 contracts offered by OpenZeppelin. This evaluation is particularly relevant due to OpenZeppelin's established presence in the ecosystem and the first-time application of CSPN methodology to smart contracts within this context.

 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	
Page 28 / 40	

Assets	Storage / Access Location	Confidentiality	Integrity	Authenticity
A9. User Balances	Account Trie / EVM	No	Yes	Yes
A10. User Allowances	Account Trie / EVM	No	Yes	Yes
A11. Token Supply	Account Trie / EVM	No	Yes	Yes
A12. Name	Account trie / EVM	No	Yes	Yes
A13. Symbol	Account Trie / EVM	No	Yes	Yes

3.3 Description of Environmental Assumptions

Assumptions	Description
H1	No group of related entities holds more than 66% of the staked funds on the Ethereum network, and the consensus algorithms are considered secure.
H2	Smart contracts are deployed on a decentralized EVM-compatible network.
H3	We assume that none of the known vulnerabilities in the infrastructure nor in the clients' implementations are exploitable.
H4	The majority of the network uses the trusted and default EVM implementation.
H5	The cryptographic algorithms used, provided they comply with ANSSI recommendations, are considered secure.
H6	The evaluation involves smart contracts operating within the scope of the public Ethereum blockchain.
H7	User identity is not stored in the smart contract and, therefore, does not require protection.
H8	Solidity version is 0.8.0 or above. Refer to the changes that have significant impacts, such as the elimination of overflow and underflow vulnerabilities. Reference here .
H9	While we acknowledge some attack vectors resulting from integration within DeFi, we will not extensively explore all the different types of attack vectors. Instead, we highlight the concern of otherwise secure smart contracts becoming vulnerable due to integration. This could be addressed in a CSPN analysis focused on dApps.

 IPSPProtocol	 RED ALERT LABS IoT Security	Reference: IPSP.RAL.SC-2024
		Version: 1.0
TARGET OF EVALUATION C.S.P.N		Page 29 / 40

3.4 Threats Description

Although smart contracts deployed with older Solidity versions are still operational, this evaluation prioritizes modern versions and the associated challenges for new projects and those undertaking migrations. Migrating to newer Solidity versions is generally considered a best practice due to ongoing enhancements in the language and the EVM.

Threat ID	Threat	Description	Assurance Level
T.01	Floating Pragma	Contracts should be deployed with the same compiler version and options they were thoroughly tested with. Locking the pragma ensures that contracts are not accidentally deployed using an outdated compiler version, which could introduce bugs affecting system integrity.	Basic
T.02	Unchecked Call Return Value	The return value of a message call is not verified. Execution continues even if the called contract throws an exception. If the call fails accidentally or an attacker forces it to fail, it may cause unexpected behavior in subsequent program logic.	Substantial
T.03	Unprotected Ether Withdrawal	Missing or insufficient access controls in gas token management can allow malicious parties to withdraw part or all Ether from a contract account. This issue is sometimes caused by inadvertently exposing initialization functions or incorrectly naming constructors, leaving them callable by anyone.	Substantial
T.04	Update Proxy Logic	If a malicious user gains access to the keys responsible for upgrading the proxy, it could lead to the loss of all funds within the contract and any funds owned by the contract.	Substantial
T.05	Unprotected SELFDESTRUCT Instruction	Missing or insufficient access controls may allow malicious parties to trigger the self-destruction of a contract.	Substantial
T.06	Reentrancy	One major risk of calling external contracts is that they may gain control over the execution flow. In a reentrancy attack, a malicious contract repeatedly calls back into the caller contract before the initial function call is completed, causing undesired interactions.	Substantial
T.07	State Variable Default Visibility	Explicitly declaring visibility makes it easier to detect incorrect assumptions about who can access a variable, reducing the risk of unintended data exposure.	Basic
T.08	Uninitialized Storage Pointer	Uninitialized local storage variables can point to unexpected storage locations in the contract, potentially causing intentional or unintentional vulnerabilities.	Substantial
T.09	Assert Violation	The assert() function in Solidity is meant to validate invariants. A failing assert() indicates one of two issues: (1) A bug allows the contract to enter an invalid state, or (2) assert() is misused, such as for input validation instead of validating internal invariants.	Substantial
T.10	Use of Deprecated Solidity Functions	Using deprecated functions and operators in Solidity reduces code quality and may introduce side effects or compilation errors in newer compiler versions.	Basic
T.11	Delegatecall to Untrusted Callee	The delegatecall instruction executes code at a target address in the context of the calling contract. This can be dangerous, as	Basic

		Reference: IPSP.RAL.SC-2024
		Version: 1.0
TARGET OF EVALUATION C.S.P.N		
Page 30 / 40		

		untrusted target code can modify storage, change the caller's balance, or introduce vulnerabilities in the contract logic.	
T.12	DoS with Failed Call	External calls may fail accidentally or deliberately, potentially causing a Denial of Service (DoS) in the contract. Isolating external calls in separate transactions can minimize impact, especially for payments, where user withdrawal patterns should replace direct transfers.	Substantial
T.13	Transaction Order Dependence (TOD)	Transaction order on Ethereum is determined by miners, who prioritize higher gas prices. Exploiting this can lead to race conditions where an attacker anticipates and front-runs transactions, leading to undesired outcomes, such as stealing rewards or bypassing authorization.	Substantial
T.14	Authorization via tx.origin	Using tx.origin for authorization can be exploited if an authorized account interacts with a malicious contract. The malicious contract can pass the authorization check using tx.origin, as it reflects the original sender of the transaction.	Basic
T.15	Block Values as a Proxy for Time	Values like block.timestamp and block.number are unreliable as precise time indicators. Miners can manipulate timestamps within certain bounds, and block times are variable, making these values unsuitable for time-critical logic.	Substantial
T.16	Signature Malleability	Signatures can be slightly altered without access to the private key and still remain valid. For example, modifying the v, r, or s values can create new valid signatures, potentially allowing attackers to replay previously signed messages or bypass authorization mechanisms.	High
T.17	Incorrect Constructor Name	Before Solidity 0.4.22, constructors were defined by naming a function the same as its containing contract. If the name didn't match the contract name (e.g., due to reuse or renaming), the function became callable by anyone, risking unauthorized actions like resetting ownership.	Basic
T.18	Shadowing State Variables	Ambiguous variable naming across inherited contracts or within the same contract can lead to unintended behavior. Multiple instances of variables with the same name can cause confusion and security vulnerabilities in complex contract systems.	Basic
T.19	Weak Sources of Randomness from Chain Attributes	Using chain attributes like block.timestamp or blockhash for randomness is insecure, as miners can manipulate these values within certain limits. This weakness can be exploited in applications like games or lotteries, where reliable randomness is crucial.	Basic
T.20	Missing Protection Against Signature Replay Attacks	Without safeguards, signatures can be replayed across transactions if the contract implement a signature verification mechanism. Effective implementation should track processed message hashes and ensure each is only used once to prevent unauthorized actions or duplicate processing.	Substantial
T.21	Lack of Proper Signature Verification	Poor implementation of signature verification may assume a signed message is valid if its msg.sender matches the address of the signer. Such assumptions can be exploited, especially when proxies relay transactions, leading to bypassed authorization or compromised integrity.	Substantial

 IPSPProtocol	 RED ALERT LABS IoT Security	Reference: IPSP.RAL.SC-2024
		Version: 1.0
TARGET OF EVALUATION C.S.P.N		Page 31 / 40

T.22	Improper Data Validation	The require() function should validate external inputs or returned data. Incorrect conditions or insufficient validation may indicate a bug in the contract or overly restrictive logic, resulting in unexpected contract behavior.	Basic
T.23	Write to Arbitrary Storage Location	If attackers gain access to arbitrary storage locations, they can overwrite sensitive data like ownership details or authorization fields, bypassing security measures and corrupting the contract state.	Substantial
T.24	Incorrect Inheritance Order	Solidity's multiple inheritance can cause ambiguity if inheritance order is neglected. Improper ordering can lead to unexpected behavior in derived contracts, particularly when resolving function overrides or priority between parent contracts.	Basic
T.25	Insufficient Gas on Subcalls	Insufficient gas provided during subcalls can lead to "gas grieving" attacks. Attackers can censor transactions by providing just enough gas to execute part of the function but not enough for the subcall, causing transaction failures or service disruption.	Substantial
T.26	Arbitrary Jump with Function Type Variable	Solidity supports function types, which can store references to functions. If a function type variable is arbitrarily modified, attackers can execute unintended or malicious code. Using low-level operations like mstore may allow attackers to redirect function calls, leading to unauthorized state changes or vulnerabilities.	High
T.27	Block Gas Limitation and DDoS	Contracts relying on extensive computations or growing state sizes may exceed block gas limits, rendering them unusable. Logic complexity and state growth can make execution prohibitively expensive or impossible, causing Denial of Service (DoS) conditions that disrupt contract functionality and operations.	Substantial
T.28	Typographical Errors	Simple mistakes, such as += instead of +=, can introduce valid but unintended logic. For instance, reinitializing a variable instead of incrementing it can lead to unexpected behavior. Recent Solidity versions have deprecated the unary + operator to mitigate such risks.	Basic
T.29	Right-To-Left-Override Control Character (U+202E)	Malicious actors can use Unicode characters like Right-To-Left-Override (U+202E) to manipulate text rendering. This can disguise the true logic of a contract, misleading users about its actual implementation and intent.	Basic
T.29	Unexpected Ether Balance	Contracts assuming specific Ether balances may fail when forced Ether transfers occur (e.g., via selfdestruct or mining rewards). This can lead to unexpected behaviors, such as Denial of Service (DoS), making the contract unusable in certain scenarios.	Basic
T.30	Hash Collisions with Multiple Variable-Length Arguments	Using abi.encodePacked() with multiple variable-length arguments may result in hash collisions. Attackers can manipulate the order of elements to generate the same encoding, bypassing signature verification or authorization checks.	Substantial
T.31	Message Call with Hardcoded Gas Amount	Functions like transfer() and send() use a fixed gas amount of 2300, which can fail if gas costs increase (e.g., after a hard fork). This behavior can break contracts relying on these functions for value transfers, as demonstrated after the EIP-1884 upgrade.	Basic
T.32	Private Data On-Chain	Private variables in Solidity are only private at the source code level. Due to the transparent nature of Ethereum, they can be read	Basic

		Reference: IPSP.RAL.SC-2024
		Version: 1.0
TARGET OF EVALUATION C.S.P.N		
Page 32 / 40		

		directly from contract storage using blockchain analysis tools, making sensitive data accessible to attackers.	
T.33	EVM Upgrades	As the EVM evolves, changes (e.g., account abstraction) can break existing assumptions about contracts. For instance, EOAs with associated code could invalidate checks like <code>address(msg.sender).code.length == 0</code> . Such changes may introduce reentrancy risks or require costly dApp migrations and audits.	Substantial
T.34	Event Emission Inconsistencies	Missing or inconsistent event emissions can hinder transparency and make it difficult for users, auditors, or off-chain applications to track state changes. This issue is critical for ERC20/ERC721/ERC1155 contracts, where events like Transfer or Approval are essential for compliance and interoperability.	Substantial
T.35	Incorrect Fallback/Receive Implementation	Improperly implemented fallback or receive functions can lead to unintended Ether acceptance or rejection. Contracts without proper receive functions may fail to accept direct Ether transfers, while overly permissive fallback functions can unintentionally expose the contract to DoS or excessive gas consumption.	Substantial
T.36	Function Selector Collisions	When multiple functions share the same selector (due to Solidity's four-byte selector hashing), unintended functions may be executed, especially in upgradeable or proxy contracts. This is a technical issue that can break expected functionality and lead to vulnerabilities.	Substantial
T.37	ERC20 Approval Race Condition	The approve and transferFrom pattern in ERC20 contracts is vulnerable to a race condition. An approved spender can front-run a transaction reducing or revoking their allowance by executing a transfer while the original allowance is still valid, leading to unauthorized token usage.	High
T.38	Malicious Delegatecall Usage	The delegatecall instruction can be exploited if the target address is malicious or modified unexpectedly. Since delegatecall executes code in the context of the calling contract, attackers can manipulate storage or state, causing critical vulnerabilities or unauthorized actions.	High
T.39	Math Errors	Mathematical operations, such as addition, subtraction, multiplication, or division, may cause rounding errors due to the lack of floating-point precision in Solidity. These issues can result in unintended state changes, exploits, or inaccuracies.	Substantial
T.40	Key Mismanagement in Development Environments	Poor management of private keys or seed phrases in development environments can lead to their exposure. Such exposure can compromise wallets, allowing attackers to steal funds or impersonate privileged accounts.	High
T.41	Ice Phishing via Malicious Transaction Flows	Malicious actors can design transaction flows that appear legitimate but trick users into unintentionally approving harmful actions. This can lead to asset theft or unauthorized contract interactions.	High

 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	Page 33 / 40

In this analysis, we focus exclusively on threats intrinsic to the ERC20 token itself, and its technical features as enabled by the EVM and solidity programming language, independent of its integration or external usage. As a result, reentrancy attacks are excluded, since an ERC20 token does not invoke external contracts. Similarly, threats such as oracle price manipulation, non-standard ERC implementations, or vulnerabilities tied to proxy patterns fall outside the scope of this analysis.

It is also essential to step back and recognize that the majority of risks and the complexity of assessing smart contract security do not solely originate from the contract itself but from its interactions with millions of other smart contracts, each with unique characteristics. This is known as integration risk. Even when a smart contract is not explicitly designed for integration, the permissionless nature of public blockchains in Web3 allows any two incompatible protocols to be connected, potentially leading to exploitation. We will explore this further in the conclusion, particularly when evaluating the comprehensiveness of CSPN analyses conducted solely on smart contracts

3.5 Security Functions Description

Security Function ID	Security Function	Description	Category
SF.1	Proper Identification and Use of All Smart Contracts	Ensure that all smart contracts in the system are registered, their purposes and functions are well-documented, and they are used as designed. This includes systematically reviewing newly added or updated contracts.	Architecture and Design
SF.2	Security Assumptions in Design	Ensure that security assumptions specific to smart contracts are addressed during the design phase. Risk analysis and threat modeling are mandatory.	Architecture and Design
SF.3	Secure Deployment Workflows	Establish secure deployment workflows to ensure that developers and operators understand security policies and follow procedures to reduce vulnerabilities during deployment.	Policies and Procedures
SF.4	Updates Tested on Local Forks	Ensure that updates are first tested on a local fork of the main network to verify compatibility and observe behavior under updated conditions.	Upgradeability
SF.5	Backward Compatibility Ensured	Verify that all contract functionalities work as intended after an update, ensuring backward compatibility and predictable behavior.	Upgradeability
SF.6	Time-Lock Mechanisms for Critical Actions	Employ time-lock mechanisms to delay critical actions, giving users and stakeholders an opportunity to review and potentially challenge unauthorized or harmful operations.	Upgradeability
SF.7	Well-Defined Roles and Privileges	Establish robust access control systems with specific roles and privileges for users and	Access Control

 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	
Page 34 / 40	

		contracts, ensuring access only to authorized entities.	
SF.8	Access Restricted to Privileged Users and Contracts	Restrict access to privileged users and contracts using strict role-based access control (RBAC).	Access Control
SF.9	External Calls Considered for Abuse Cases	Evaluate external calls to and from other contracts for potential abuse and authorize them accordingly.	Communications
SF.10	Secure Use of Third-Party Libraries	Ensure third-party libraries are secure and adopt state-of-the-art practices (e.g., OpenZeppelin).	Communications
SF.11	Safe and Predictable Arithmetic Operations	Protect against vulnerabilities related to arithmetic operations such as overflow/underflow and ensure precision in fixed-point arithmetic.	Arithmetic
SF.12	Prevent Contract Availability Manipulation	Ensure the contract logic prevents influence on availability by managing loops, avoiding costly operations, and mitigating DoS attacks.	Denial of Service
SF.13	Stored Data Identification and Protection	Identify and protect all data stored in the contract, ensuring no sensitive data is exposed on-chain.	Data on Blockchain
SF.14	No Plaintext Secrets in Blockchain	Ensure no sensitive data is stored in plaintext on the blockchain.	Data on Blockchain
SF.15	Avoid Data Misrepresentation Vulnerabilities	Ensure the smart contract is not vulnerable to data misrepresentation (e.g., homoglyph attacks, improper mapping keys).	Data on Blockchain
SF.16	Optimized Gas Usage	Optimize gas consumption for efficiency and security, preventing gas exhaustion vulnerabilities.	Gas Usage and Limits
SF.17	Adherence to Smart Contract Gas Limitations	Avoid hardcoded gas values in function calls and consider dynamic adjustments.	Gas Usage and Limits
SF.18	Modular and Auditable Code Structure	Ensure clear, modular, and auditable code to reduce vulnerabilities and improve maintainability.	Code Quality
SF.19	Removal of Unused Code	Remove unnecessary code to reduce the attack surface and improve maintainability.	Code Quality
SF.20	Best Practices for Naming Variables and Functions	Follow best practices for naming conventions to avoid ambiguity or issues like variable shadowing.	Code Quality
SF.21	Comprehensive Static and Dynamic Testing	Perform static and dynamic testing (e.g., formal verification, symbolic execution) to validate against specifications.	Test Coverage
SF.22	Secure Handling of Elevated Privileges	Use multi-signature wallets or strict access control for elevated privileges to prevent unauthorized actions.	Access Control

 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	
Page 35 / 40	

SF.23	Validation of External Call Return Values	Validate external call responses to prevent logic flaws in cases of failure or malicious behavior.	Communications
SF.24	Enforcement of Delegatecall Restrictions	Restrict delegatecall use to trusted addresses, preventing state manipulation or malicious execution.	Communications
SF.25	Prevention of Signature Replay Attacks	Include nonce tracking or timestamps for signed messages to prevent replay attacks where an attacker reuses a previously valid signature to perform unauthorized actions.	Cryptographic Integrity
SF.26	Protection Against Integer Overflow and Underflow	Use safe arithmetic libraries or Solidity 0.8+ built-in checks to prevent overflow or underflow during arithmetic operations, ensuring numerical correctness in contract logic.	Arithmetic
SF.27	Hash Collision Prevention in Encoded Data	Avoid using functions like abi.encodePacked() with multiple variable-length arguments to prevent hash collisions that could lead to signature bypass or unintended behavior in function calls.	Cryptographic Integrity
SF.28	Keystore Management for Development Environments	Prohibit plaintext storage of private keys or seed phrases in development environments; use encrypted keystore files.	Cryptographic Integrity
SF.29	Transaction Simulation Software	Use simulation tools to preview wallet transactions, ensuring correctness and avoiding ice phishing attacks.	Cryptographic Integrity
SF.30	Proxy Upgradeability Standards	Maintain consistent variable layouts for proxies, enforce access control for upgrades, and adhere to proxy documentation standards.	Upgradeability
SF.31	Running Security Tools in Development Lifecycle	Integrate static analysis, fuzzing, and formal verification tools during the smart contract development lifecycle to identify and mitigate vulnerabilities early.	Development Practices
SF.32	Monitoring and Alerting	Deploy monitoring systems for smart contract activity and real-time alerts for anomalies.	Operations
SF.33	Arithmetic Precision and Safe Operations	Validate order of operations and magnitude to maintain precision; use safe arithmetic libraries.	Arithmetic

 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	Page 36 / 40

3.6 Coverage Matrix

3.6.1 Threats and Assets

Introduction to the Threats x Assets Analysis

The following matrices present the coverage of threats to sensitive assets. The letters A, I, C, and Au represent the requirements for Availability, Integrity, Confidentiality, and Authenticity, respectively.

Direct Risks

These refer to threats inherent to the ERC20 standard implementation itself or when interacting exclusively with Externally Owned Accounts (EOAs), which lack executable code (pre-Account Abstraction). These risks focus on vulnerabilities within the ERC20 contract as defined by its specification and are addressed in the first matrix.

Indirect Risks

These arise when users interact with the ERC20 token using smart contracts or integrate it into decentralized applications (dApps). Such interactions can introduce additional vulnerabilities based on the integration logic, composability, and behaviors of the interacting contracts. These risks are explored in the second matrix, emphasizing threats that emerge not from the ERC20 contract itself but from its role within larger systems.

Why This Analysis Is Necessary

In a public, permissionless blockchain like Ethereum, anyone can deploy a smart contract and interact with your ERC20 token or integrate it into another protocol. This permissionless composability means that the security of your ERC20 token may be influenced by the logic and behavior of third-party contracts or protocols, including those that already exist or have yet to be created. This dynamic nature adds significant complexity to the analysis. A thorough evaluation of both direct and indirect risks is essential to identify and mitigate potential vulnerabilities, thereby safeguarding the token's integrity and the broader ecosystem in which it operates.

Direct Threats

	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13
T01	I											
T13								I, Au	I, Au			
T22		I						I	I			
T27	A											
T32		C										
T37									I,Au			

 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	
Page 37 / 40	

Indirect Threats

	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13
T01	I											
T02								I, Au				
T03								I, Au	I, Au			
T04		I							I			
T05		I						I	I			
T06	A											
T07		C										
T08									I,Au			
T09		I						I				
T10	I	I										
T11		I						I,Au				
T12								A				
T13								I,Au	I,Au			
T14		I,Au						I,Au	I,Au			
T15								I,Au				
T16								I,Au	I,Au			
T17												
T18		I						I				
T19								I,Au				
T20												
T21								I,Au				
T22		I						I				
T23								I				
T24								I,A				
T25								A				
T26								I				
T27								A	A			
T28		I						I				
T29		I,Au						I,Au				
T30								I,Au	I,Au			
T31								A				
T32								C,I				
T33		I										
T34												
T35								A	A			

 	Reference: IPSP.RAL.SC-2024											
	Version: 1.0											
TARGET OF EVALUATION C.S.P.N									Page 38 / 40			

T36								I,Au				
T37								I,Au				
T38		I						I,Au				

These threats cannot be addressed internally within the scope of a single smart contract or crypto asset. Establishing them requires a deep understanding of the existing systems, protocols, and logic used in DeFi ecosystems, including decentralized exchanges, lending protocols, staking systems, decentralized autonomous organizations (DAOs), smart contract wallets, and more. Each of these systems comes with its own unique complexities and specificities, making it crucial to consider their nuances during integration.

As such, these threats fall under the integrator responsibility framework outlined in the conclusion, emphasizing that the security of the broader ecosystem relies on the responsible and secure integration of these components.

3.6.2 Threats and Security Fonction Matrix

As previously explained, every crypto asset can interact with and depend on other smart contracts as it integrates into the DeFi ecosystem. However, from a security perspective, it is neither scalable nor efficient to incorporate security mechanisms into a crypto asset based on potential interactions with other protocols or smart contracts. This inefficiency stems from the technical limitation of smart contracts themselves, as they cannot detect or respond to vulnerabilities or hacks in other smart contracts during the course of a transaction. For instance, this limitation becomes evident in scenarios such as the token's involvement in a liquidity pool.

As a result, the analysis of security functions can only focus on the internal functionalities of the smart contract itself and is limited to covering direct threats.

The following matrix presents the coverage of threats addressed by the security functions:

 IPSPProtocol 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	
Page 39 / 40	

	FS1	FS2	FS3	FS4	FS5	FS6	FS11	FS12	FS14	FS15	FS16	FS17	FS18	FS22	FS27	SF31	SF36	SF39
T01		X	X										X					
T13						X												
T20																		
T22										X			X					
T27											X							
T32									X									
T37																		

Some risks highlighted in this analysis cannot be fully mitigated within the current blockchain framework, exposing inherent challenges in the design of systems like Ethereum, particularly regarding the public mempool. A significant example is the threat represented by T37 (Transaction Order Dependence), which underscores vulnerabilities arising from the ordering of transactions in the mempool, where the sequence of transactions can critically influence outcomes.

The Problem with Transaction Ordering

On the Ethereum mainnet, transaction ordering is heavily influenced by financial incentives. The current block-building process is outlined in the Proposed Builder Separation (PBS) proposal. While PBS aims to introduce clearer distinctions between roles in the block-building process, much of the activity remains off-chain.

In this model, transaction ordering is neither deterministic nor guaranteed to be fair; instead, it is driven by financial incentives associated with Maximal Extractable Value (MEV) opportunities. Initially perceived as an inefficiency, MEV has evolved into a sophisticated ecosystem involving public and private mempools. Specialized entities known as block builders select transactions based on profitability.

 IPSPProtocol 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	Page 40 / 40

Although many Ethereum validators now leverage systems like MEV Boost, which outsources block-building to these specialized builders, this introduces new risks and complexities. Transaction ordering in such systems can be manipulated for financial gain, often to the detriment of users. For additional insights on MEV Boost adoption and the associated centralization risks, visit mevboost.pics.

Direct Impact on ERC20 Token Users

In the context of ERC20 tokens, these dynamics can lead to significant vulnerabilities. For example:

Allowance Manipulation: A user grants another user an allowance to spend their tokens. Subsequently, the original user submits a transaction to revoke this allowance.

Transaction Ordering Exploitation: The spender, monitoring the public mempool, detects the pending revocation. They then submit a transaction to utilize the allowance, attaching a higher gas fee to ensure their transaction is processed before the revocation.

This type of exploitation, facilitated by transaction order manipulation, directly impacts ERC20 token users, allowing unauthorized actions through what is commonly referred to as a "mempool race condition".

Future Mitigation Plans

The Ethereum ecosystem recognizes this concern and has ongoing plans to address these challenges over the mid to long term.

Conclusions

Responsibility, Security, and Risk in Web3 Ecosystems

Web3 introduces unique challenges for security and responsibility due to the **permissionless composability** of its systems. In this environment, smart contracts and crypto assets can interact transparently by design. While access control mechanisms can limit function access, they undermine the core value of public blockchains: composability. However, this feature also introduces risks that must be addressed in a structured way. To evaluate security comprehensively, risks should be categorized into three main types:

 IPSPProtocol 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	Page 41 / 40

1. **Self-Analysis:** Focused on the internal and technical mechanisms of the smart contract.
2. **Integration Risks:** Analyzed in conjunction with other ecosystem components.
3. **Systemic Risks:** Broader risks, often financial, associated with the interconnected nature of crypto assets, such as ERC20 tokens.

Independent Smart Contract Analysis

The first level of analysis examines the **internal mechanisms and technical aspects** of smart contracts, inherent to programming languages like Solidity and Vyper and execution environments such as the Ethereum Virtual Machine (EVM). This includes:

- Evaluating the correctness of the **business logic** and ensuring the security of the code itself.
- Adherence to standards like ERC20, covering proper allowance handling, arithmetic operations, and low-level call management.
- Addressing limitations in Solidity, such as the lack of standard mechanisms for mathematical calculations (e.g., compliance with IEEE 754 for Floating-Point Arithmetic).

Complexity in Composable Systems

Crypto assets are designed to function as part of interconnected systems of **interdependent protocols and smart contracts**. However, the crypto asset creator cannot control how other entities integrate or interact with the asset, especially in contexts that may not have been anticipated during the asset's design. This creates a need for meticulous tracking of integrations and dependencies to ensure compatibility and security.

Importantly, two or more **secure smart contracts**, when evaluated independently, can create vulnerabilities when integrated. This composability introduces complexities that require evaluation not only at the unitary level but also as part of a broader system.

Integration Risks

The transparent and composable nature of blockchain ecosystems extends security concerns beyond standalone analysis. Integration risks arise when:

1. **Protocols Interact:**
 - a. Protocols that are secure independently can create vulnerabilities when combined.

 IPSPProtocol 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	Page 42 / 40

- b. **Example:** The interaction between Uniswap V1 and ERC777, where unforeseen integration issues introduced vulnerabilities despite both protocols being secure individually.

2. **Behavior in Context:**

- a. Crypto assets can introduce new execution paths and vulnerabilities based on their integration with decentralized applications (dApps) or other systems.
- b. Smart contracts must be analyzed in their **usage context**, as many vulnerabilities stem from composability rather than the asset itself.

3. **Lack of Behavioral Guarantees:**

- a. Ensuring that integrating protocols behave as intended is **technically infeasible** in decentralized environments.
- b. Poorly designed or malicious integrators can create vulnerabilities, impacting otherwise secure crypto assets and their associated systems.

Systemic Risks

Systemic risks arise due to cascading failures or unrelated events within the ecosystem. Examples include:

1. **Liquidity Pool Vulnerabilities:**

- a. A token in a DEX liquidity pool may face indirect risks if the pool is compromised. For instance:
 - i. If one of the two tokens in the pool is exploited, the pool could be drained, impacting all users holding either token.
 - ii. A large-scale sale of hacked tokens could crash the crypto asset's price, impacting all holders.

2. **Market Instability:**

- a. A compromised liquidity pool could lead to significant price drops for the token, causing widespread losses.
- b. If the liquidity pool was the primary market for the asset, its illiquidity could erode user trust and create a feedback loop of panic and sell-offs.

These examples underscore the interconnected nature of decentralized ecosystems, where failures in one protocol can ripple across others, even when the affected asset or contract is not directly at fault.

 IPSPProtocol 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	Page 43 / 40

Challenges of Responsibility in Web3

The Web3 ecosystem has adopted a **shared responsibility model**, where:

- The **crypto asset creator** ensures the security of the asset's implementation.
- The **integrator** ensures secure interactions within their protocol.

However, this model faces significant limitations:

1. **Uncontrolled Integrations:**
 - a. Any entity can integrate crypto assets or protocols, and there is no mechanism for creators to validate these integrations.
2. **Pseudo-Anonymity:**
 - a. Integrators can remain pseudo-anonymous, making it impossible to assign responsibility in the event of an exploit.
3. **Undefined Responsibility Layers:**
 - a. In a shared responsibility framework, the lack of a clear reasoning layer for assigning blame leaves gaps in accountability.

Towards a Comprehensive Security Framework

To align with the security expectations of regulated industries—such as **banking**, where entities are fully responsible for their products in all scenarios—it is necessary to expand the scope of analysis beyond unitary risks. This requires incorporating both **integration risks** and **systemic risks** into the evaluation.

Indirect Threats Analysis

Because crypto assets are rarely used in isolation, their security must be evaluated in conjunction with other protocols and systems. An **indirect threats analysis** includes:

- Understanding **natural usage patterns**, where crypto assets are integrated into larger ecosystems.
- Evaluating the cascading effects of these integrations, particularly in composable systems.

This approach ensures a more comprehensive assessment of security, reflecting the realities of Web3 usage.

 IPSPProtocol 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	Page 44 / 40

While direct threat analysis evaluates the standalone security of a crypto asset, it does not account for the broader risks associated with integration and systemic events in a permissionless and composable environment. To establish a robust security framework, it is crucial to incorporate these additional dimensions into the evaluation process.

As demonstrated in the threat analysis, the majority of risks originate from external threats. Evaluating the Target of Evaluation (TOE) requires a comprehensive assessment of indirect threats. This includes analyzing how new smart contracts or decentralized applications interact with all existing and live smart contracts and dApps. To ensure thoroughness, the evaluation must also consider potential future and yet-to-be-developed solutions, as the risk of integration remains a critical factor.

However, this approach is not scalable, especially given the immutable nature of smart contracts. Once deployed, vulnerabilities discovered post-deployment must be identified promptly to allow applications to assess, address, and deploy updated versions. This process often involves migrating users and operations from the older, vulnerable version to a secure one, resulting in significant operational disruptions and a degraded user experience. This limitation underscores the importance of proactive and dynamic threat assessments to minimize such impacts and enhance the overall security posture of blockchain ecosystems.

This challenge reflects a broader issue within current blockchain technology, which lacks built-in security mechanisms to empower smart contract developers to define and control risks within a well-defined and limited scope. Introducing systems that provide dynamic analysis could mitigate these risks, enabling developers to maintain control over their crypto asset's behavior while still leveraging the permissionless composability that is a defining strength of the ecosystem.

Regarding the CSPN methodology, it demonstrates significant potential for delivering precise security assessments by clearly identifying direct threats and defining corresponding security functions to address them. This approach aligns with industry practices adopted by auditing firms but currently lacks standardized guidelines and processes that are broadly recognized and validated by the Web3 community.

The CSPN framework offers a unique opportunity to unify industry stakeholders around a key process and framework for generating relevant and standardized results. Such standardization is essential, especially considering the widespread risks and significant impact of hacks and scams within the industry. By providing clear, actionable, and standardized assessments, CSPN can enhance the security landscape and establish itself as a cornerstone methodology for evaluating the security of blockchain ecosystems globally.

 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	Page 45 / 40

Form references : IPSP.RAL.SC-2023

Version : 1.0

Written by :

Role	Name	Date (jj/m/aaaa)	Signature
Evaluateur sécurité de produit IoT	GEDEON Paul	10/02/2023	

Approved by :

Fonction	Nom	Date (jj/m/aaaa)	Signature
RAL Lab Director	KHALIL Ayman	16/06/2021	
IPSPProtocol Lead	MUGNIER Jean-Loïc	23/12/2024	

Historique du formulaire :

Version	Rédacteur	Date (jj/m/aaaa)	Modification
0.1	GEDEON Paul	18/04/2023	Focus on the creation and conceptualization of the smart contracts.
0.2	GEDEON Paul	25/05/2023	Emphasis on analyzing and improving the design and functionality of smart contracts.
0.3	GEDEON Paul	09/11/2023	Address comments and recommendations provided by the working group (WG).
0.4	MUGNIER Jean-Loïc	20/09/2024	Enhance and expand the introductory context to provide a clearer foundation for analysis. Develop analytical frameworks for evaluating various use case scenarios.
0.5	MUGNIER Jean-Loïc	15/10/2024	Translate to english.
0.6	MUGNIER Jean-Loïc	29/10/2024	Develop the threats, security functions, and analysis framework, addressing both direct and indirect threats.
0.7	MUGNIER Jean-Loïc	20/12/2024	Expand the coverage of security functions and conduct a comprehensive analysis of the relationships between threats, assets, and security functions.

 	Reference: IPSP.RAL.SC-2024
	Version: 1.0
TARGET OF EVALUATION C.S.P.N	Page 46 / 40

1.0 MUGNIER Jean- 23/12/2024 Conclusions and final review
Loïc

END OF THE DOCUMENT
