Evaluating a Smart Contract Security Through the CSPN Methodology

A structured and standardized security evaluation framework for smart contracts using the ANSSI "First Level Security Certification" (Certification de Sécurité de Premier Niveau) methodology.

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

Evaluator
Red Alert Labs (RAL) – CAMPUS Cyber Initiative
IPSProtocol – CAMPUS Cyber Initiative

Distribution List

Name	**Organization**
GEDEON Paul	RAL
MUGNIER Jean-Loïc	IPSProtocol
CHRISTOFI Maria	BNP Paribas
MOUILLE Stéfane	CLR Labs
DEMASI Thibault	CLR Labs
BENJAMIN Thomas	Orange Cyberdéfense
PELFRESNE Christophe	BANQUE DE FRANCE
SENOT Olivier	DOCAPOSTE
Elyes Lehtihet	French Cybersecurity Agency (ANSSI)
GAINIER Fabien	CAMPUS CYBER
MIS Jean-Michel	Fédération Blockchain
GORNA Karolina	Ledger
PELFRESNE Christophe	Banque de France
VENTUZELO Patrick	Fuzzing Labs
LE COQ AKLI	Gendarmerie
BENOIT Thomas	SETINSTONE
LE DOUARON Alexandre	Lab Banque De France
FAURE Frédéric	Banque de France
CAMUS Laurent	French Prudential Supervision and Resolution Authority (ACPR)

Contents

Figures

Figures Licenses

Figure	Creator	License
Figure 1	IPSProtocol	CC BY-SA 4.0
Figure 2	IPSProtocol	CC BY-SA 4.0
Figure 3	IPSProtocol	CC BY-SA 4.0

1. Introduction

2. 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 4.1 of the [ANSSI_CSPN_CER_P_02] standard.

3. 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.

4. Identification of the System to be Evaluated

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

5. References

Code	Reference	Name and Source
	ANSSI_CSPN_CER_P_02	Criteria for First-Level Security Certification Evaluation https://cyber.gouv.fr/sites/default/files/document/ANSSI-CSPN-CER-P-02%20Criteres_pour_evaluation_en_vue_d%27une_CSPN_v5.0.pdf
[1]	**EN 17640:2022**	Cybersecurity Evaluation Methodology for ICT Products
[2]	ISO/IEC 15408-1:2022	Information technology – Security techniques – Evaluation criteria for IT security – Part 1: Introduction and general model
[3]	CSPN	Certificat de Sécurité de premier Niveau
[5]	Ethereum Guide	https://ethereum.org/en/developers/docs/smart-contracts/security/
[6]	Consensys	https://consensys.github.io/smart-contract-best-practices/
[7]	SWC	https://swcregistry.io/
[8]	Solidity	https://docs.soliditylang.org/en/latest/bugs.html

[9]	BNB Chain Web3 Security Framework	https://github.com/bnb-chain/avengerdao/tree/main/Web3%20Security%20Frameworks

6. Definitions and Abbreviations

Name	Abbreviation	Definition
National Cybersecurity Agency of France	ANSSI	The Agence Nationale de la Sécurité des Systèmes d'Information (ANSSI) is responsible for defending the information systems of the State and providing advice, support, and incident response to government administrations and operators of vital importance. As the national authority on cybersecurity, ANSSI also plays a key role in setting cybersecurity standards, promoting best practices, and ensuring compliance with security regulations to enhance the resilience of critical digital infrastructures in France.
Red Alert Labs	RAL	Security Evaluation Laboratory for Connected Objects (IoT)
International Standard Organisation	ISO	International Standardization Organization
General Data Protection Regulation	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
Target Of Evaluation	TOE	Product to be evaluated by the laboratory within the scope defined with the client.
Ethereum Virtual Machine	EVM	The Ethereum Virtual Machine (EVM) is a decentralized computing environment that executes smart contracts on the Ethereum blockchain. It acts as an isolated, sandboxed runtime that ensures deterministic execution of code across all network nodes. The EVM provides an abstraction layer that enables developers to deploy and automate complex business logic, facilitating trustless transactions, decentralized applications (dApps), and interoperability within the Ethereum ecosystem.

Decentralized Applications	dApps	Applications running on blockchain networks, enabling them to operate without a central authority. They use smart contracts to automate transactions.
ERC20 standard	ERC20	A standard on the Ethereum network that defines a common set of rules for tokens.
ERC721 standard	NFT or 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.

7. Target Description

8. Target of Evaluation - General Description

8.1.1. Target of Evaluation (TOE) Definition

The Target of Evaluation (TOE) is a smart contract implementing the ERC-20 token standard, designed for deployment on Ethereum and EVM-compatible blockchains. The ERC-20 standard defines a set of functions and event mechanisms that enable token transfers, balance inquiries, and delegated transaction approvals, providing a foundational framework for fungible digital asset representation in decentralized ecosystems.

This implementation follows the widely adopted OpenZeppelin ERC-20 contract, a security-audited reference implementation recognized for its industry best practices. It adheres to the Ethereum Foundation's ERC-20 specification, ensuring compatibility with wallets, decentralized applications (dApps), and smart contract integrations.

8.1.2. Establishing a CSPN Baseline for Smart Contract Security

As this CSPN methodology is applied to smart contracts for the first time, the evaluation deliberately focuses on an ERC-20 token implementation to establish a clear and structured approach. ERC-20 was chosen due to its widespread adoption, standardized interface, and relative simplicity, making it an ideal candidate for defining a reproducible security assessment framework.

The blockchain security auditing landscape has historically produced varied results, influenced by methodological differences among auditors. By applying CSPN certification principles, this evaluation aims to introduce a structured, repeatable, and transparent security assessment methodology for on-chain assets. This exercise will provide a baseline for future CSPN evaluations of more complex smart contract systems, helping to improve the standardization and comparability of security assessments within the blockchain ecosystem.

8.1.3. Functional Scope of the TOE

The evaluation focuses on the fundamental functionalities of the ERC-20 standard, which serves as the technical framework for fungible tokens in decentralized ecosystems. ERC-20 tokens enable seamless asset transfers, token management, and interoperability across wallets, exchanges, and decentralized applications (dApps).

This assessment ensures that the contract adheres to its intended functional specifications and that each feature is implemented correctly and securely. The following core functionalities are evaluated:

8.1.4. Token Transfers – Secure and Direct Asset Movement

ERC-20 tokens allow users to send and receive tokens between Ethereum accounts. This is facilitated through the *transfer*() function, which ensures that:

- The sender has sufficient token balance before executing the transaction.
- Tokens are deducted from the sender's balance and credited to the recipient's balance in an atomic operation (meaning it either completes fully or fails).
- The transaction emits a Transfer event, creating an auditable record on the blockchain.

8.1.5. Delegated Transfers – Third-Party Token Spending

The ERC-20 standard includes functions that enable users to authorize third parties (other accounts, EOA or smart contracts) to transfer tokens on their behalf. This functionality is implemented through:

- Approval via the *approve* function – A token holder grants another address permission to spend a specified number of tokens on their behalf.
- Delegated Transfers via *transferFrom* function – The authorized account executes the transfer within the approved limits.
- Querying Account Allowance via the *allowance* function– Provides a way to check how many tokens a third party is authorized to spend.

This feature is commonly used in decentralized exchanges (DEXs), lending protocols, and automated smart contract interactions. The evaluation ensures that these functions correctly enforce spending limits, preventing unauthorized withdrawals.

8.1.6. Balance Management – Querying Token Holdings

The ERC-20 *balanceOf*() function allows any user to check the token balance of a given Ethereum address. This function is critical for:

- Users who need to verify their token holdings in a wallet or dApp.
- dApps that display user balances in their interfaces.
- Smart contracts that require balance verification before executing transactions.

This function is designed to be read-only, meaning it does not modify the blockchain state and does not require gas fees.

8.1.7. Allowance Management – Controlling Delegated Token Spending

Since delegated transfers allow third-party spending, the ERC-20 standard includes mechanisms to manage and restrict allowances to prevent unintended transactions:

The *allowance*() function enables users and applications to verify the number of tokens that a third party is permitted to spend.

increaseAllowance() and *decreaseAllowance*() allow token holders to adjust spending limits, mitigating risks associated with excessive or outdated approvals.

The evaluation ensures that all approval changes follow correct authorization rules, preventing overwriting attacks or inconsistencies.

8.1.8. Token Supply Integrity – Maintaining a Fixed Total Supply

The totalSupply() function defines the maximum number of tokens that exist for a given ERC-20 contract. This evaluation verifies that:

- The total supply is correctly initialized when the contract is deployed.
- The supply remains immutable after deployment.
- No function allows unauthorized token creation.

8.1.9. *Token Burning – Permanent Removal of Tokens from Circulation*

The ERC-20 tokens standard implements a burning mechanism that allows tokens to be permanently removed from circulation by sending it to the zero address (0x0), reducing supply. The *burn*() function ensures:

- A token holder can voluntarily destroy tokens, reducing the total supply.

8.1.10. Event Logging – Ensuring Auditability and Transparency

Smart contracts use on-chain events to log transactions and state changes. In ERC-20, two key events ensure traceability and security:

- Transfer Event – Emitted whenever tokens are transferred, providing a verifiable record of all token movements.
- Approval Event – Emitted when an address grants spending authorization to another party, allowing for real-time monitoring of allowances.

These event logs provide historical records that can be accessed by blockchain explorers, dApps, and auditors to verify compliance and detect anomalies.

8.1.11. Adherence to ERC-20 Standard

This ERC-20 implementation follows the Ethereum Foundation's official specification and does not introduce any modifications, extensions, or deviations from the standard. The evaluation focuses on ensuring:

- Correct implementation of ERC-20 functions.
- Security best practices to prevent common vulnerabilities.
- Strict compliance with the expected behaviour of ERC-20 tokens.

By adhering to a well-established standard, this implementation remains interoperable across exchanges, wallets, and smart contract platforms, ensuring a consistent and predictable user experience.

8.1.12. Security Assumptions & Constraints

To maintain a clear evaluation scope, the following assumptions and constraints apply:

Immutability: Once deployed, the smart contract cannot be altered or upgraded. No proxy patterns or upgrade mechanisms are included.

Standard Compliance: The contract follows the standard ERC-20 specification without modifications, ensuring compatibility with Ethereum-based dApps and DeFi protocols.
Standalone Deployment: The evaluation focuses solely on this contract and does not assess interactions with external contracts, governance modules, or administrative controls.

Blockchain Infrastructure Assumptions: The Ethereum network, consensus mechanism, and EVM execution correctness are assumed to be secure and are out of scope.

No Additional Features: The evaluation is limited to the ERC-20 core specification and excludes additional functionalities such as:

- Pausing, Blacklisting, or Administrative Controls.
- Governance Extensions (e.g., voting mechanisms).
- Custom Fee or Tax Mechanisms.

9. Description of the System Internal Execution Environment

Ethereum, is a decentralized blockchain platform. It 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.

9.1.1. Role of the EVM in Smart Contract Execution

The Ethereum Virtual Machine (EVM) is the decentralized execution environment responsible for processing Ethereum transactions and executing smart contract logic. It ensures consistent and deterministic execution across all Ethereum nodes, maintaining network integrity and consensus over the accounts state. Smart contracts are compiled into EVM bytecode, which the virtual machine executes in response to transaction calls.

The EVM operates on a stack-based architecture with 256-bit word size, ensuring compatibility with cryptographic operations. It manages temporary memory, persistent storage, and gas metering, enforcing secure and efficient execution of smart contract functions.

9.1.2. Gas and Resource Management

Every EVM operation (opcode) has an associated gas cost, which serves two primary purposes:

- Mitigation of Malicious Activities: By requiring gas fees for execution, the network prevents infinite loops, denial-of-service (DoS) attacks, and inefficient computations.
- Economic Incentive for Validators: Gas fees compensate validators for processing transactions and securing the network.

Ethereum transactions incur execution costs, measured in gas, to prevent resource abuse and compensate network validators. This mechanism ensures economic security, protecting the blockchain from denial-of-service (DoS) attacks and incentivizing efficient computation.

Every EVM operation (opcode) has a fixed gas cost, defining the computational resources required to execute a transaction. The total gas fee a user pays is determined by:

Total Cost = Gas Used × Effective Gas Price

Where:

- Gas Used: The amount of computational work performed.
- Effective Gas Price: The total gas price paid per unit of gas.

9.1.3. Smart Contract Interaction and Execution Context

Smart contracts interact with external entities via transactions, which contain function calls and execution parameters. The EVM provides built-in execution context variables, enabling smart contracts to access transaction metadata, including:

- msg.sender: Address of the entity calling the contract function (typically a user or another contract).
- msg.value: Amount of ETH sent with the transaction (if applicable).
- msg.data: Raw input data accompanying the transaction, used for function selection and parameter passing.
- block.timestamp: Unix timestamp of the current block, often used for time-based logic.

These built-in functions facilitate secure contract behaviour, ensuring that transactions are correctly authenticated and processed within the expected execution environment.

For CSPN analysis, the gas and fee model introduce specific security implications

Security Consideration	**Risk**	**Mitigation**
DoS Prevention via Gas Costs	Contracts with poor logic, such as inefficient loops, can lead to excessive gas consumption, making their execution costly and potentially unsustainable in the long run.	Use gas-efficient logic and validate external calls.
Reverted Transactions & Fee Implications	Users incur base fees even if a transaction fails, except in specific cases.	Ensure proper input validation to minimize failed transactions.
Block Space Competition	During network congestion, high gas fees can prevent low-value transactions from being processed.	Smart contract logic should account for unordered execution since transaction sequencing is not guaranteed.

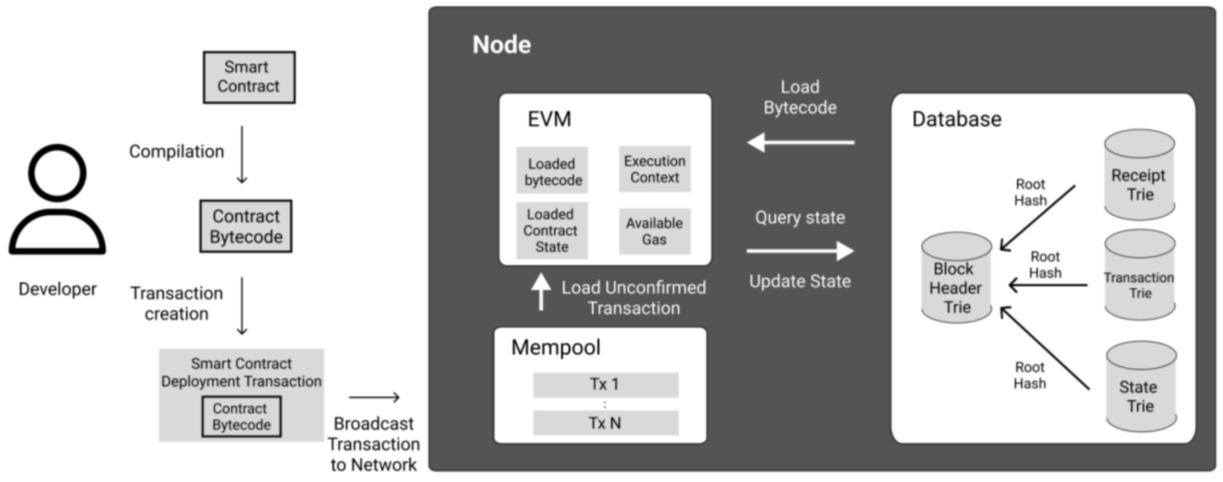

Figure 1: Overview of Smart Contract Deployment and Transaction Execution Components

9.1.4. Smart Contract Storage, Data Structure and Security Assumptions

9.1.5. Ethereum Account Model and State Management

Ethereum maintains two primary types of accounts, both recorded within the State Trie, a cryptographically structured database ensuring global state consistency and integrity:

- Externally Owned Accounts (EOAs)
  - Controlled through private keys.
  - Capable of initiating transactions and holding ETH.
  - Do not contain executable code (at least until EIP 7702)

- Contract Accounts
  - Contain compiled smart contract bytecode that defines executable logic.
  - Maintain persistent state variables that are stored independently from the contract's bytecode.
  - Can only execute functions when invoked by an external transaction or another smart contract; they cannot independently initiate transactions.

Each account entry in the State Trie consists of the following fields:

- Nonce:
  - For EOAs, tracks the number of transactions sent (prevents replay attacks).
  - For Contract Accounts, tracks the number of contracts created by the contract using the CREATE and CREATE2 opcode (increments with each deployment).
- Balance: Stores the amount of ETH held by the account.

- Storage Root: References the Storage Trie, where a contract's persistent state variables are stored (relevant only for Contract Accounts).
- Code Hash: Contains a cryptographic fingerprint of the smart contract's bytecode.
    - For EOAs, this field is empty since they do not contain executable code.
    - For Contract Accounts, this reference ensures that bytecode remains immutable after deployment.

9.1.6. Smart Contract Storage and the Storage Trie

The Storage Trie is a dedicated data structure unique to each Contract Account, where its state variables (persistent data) are stored.

- State variables are mapped as 256-bit key-value pairs, ensuring efficient indexing and retrieval.
- The Storage Root recorded in the State Trie serves as a cryptographic commitment to the contract's current state.

The use of a Storage Trie provides critical security properties:

- Data Integrity: Any modification to a contract's storage alters the Storage Root, making unauthorized tampering detectable.
- Efficient Verification: The Storage Trie enables efficient state validation without requiring full contract storage replication across all Ethereum nodes.

9.1.7. Ethereum's Cryptographic Integrity Model

Ethereum employs a Modified Merkle Patricia Trie (MPT) to ensure the security and consistency of on-chain data. It organizes blockchain data across multiple trie structures, each serving a distinct purpose:

- State Trie: Tracks the latest state of all Ethereum accounts.
- Transaction Trie: Stores all transactions included within a block, ensuring transaction integrity.
- Receipts Trie: Logs execution outcomes, including emitted events and gas usage.

Each block header contains:

- State Root Hash: Captures the latest verified state of all Ethereum accounts.
- Transaction Root Hash: A cryptographic hash linking all transactions within the block.
- Receipts Root Hash: Ensures execution results are provably linked to the block, preventing manipulation.

These cryptographic commitments ensure:

- Tamper Resistance: Any unauthorized modification to stored data results in a different trie root hash, rendering the block invalid under Ethereum's consensus rules.
- Network-Wide Consistency: All Ethereum nodes independently compute the same blockchain state based on the latest valid transactions and storage updates.
- Deterministic Execution: The Ethereum Virtual Machine (EVM) guarantees that identical contract logic produces the same output across all nodes, reinforcing trust and consistency within the network.

By leveraging structured cryptographic storage, deterministic execution, and decentralized verification mechanisms, Ethereum ensures the integrity, consistency, and resilience of smart contracts and their state across its global decentralized network

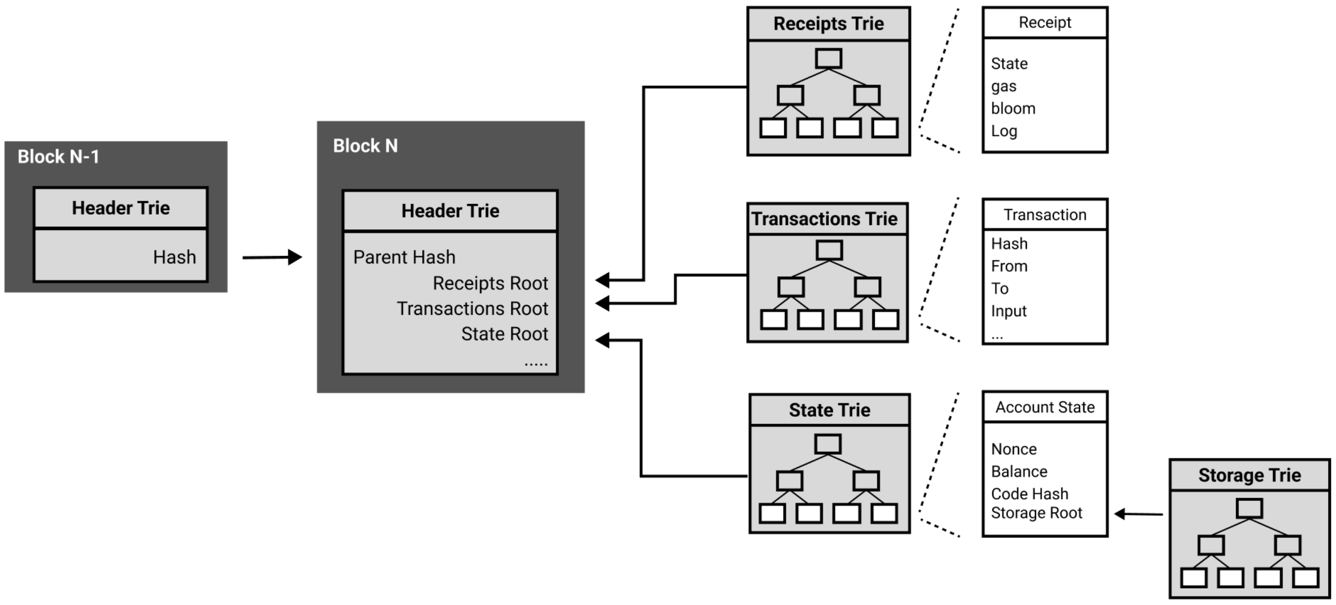

Figure 2: Ethereum Trie Structure Representation

10. Description of the Users Usage of the System

Smart contracts operate in a decentralized environment where multiple participants play distinct roles in their implementation, execution, interaction, provisioning and governance. Each actor contributes to the functionality, security, and evolution of the system. Below is an overview of these roles and their responsibilities.

10.1.1. Smart Contract Developers

Smart contract developers are responsible for writing, testing, compiling and deploying the smart contract bytecode on the blockchain. Smart contract bytecode and state being stored in the blockchain dedicated database for accounts. They use programming languages such as Solidity (Ethereum) to define the contract's logic, conditions for execution, and security mechanisms.

When deploying a contract, the developer creates a transaction containing the contract's bytecode, which is processed by the network and assigned a unique address. This transaction can fail if not properly optimized, resulting in wasted transaction fees (gas costs).

Key responsibilities:

- Writing secure contract logic to prevent vulnerabilities like reentrancy attacks and correct math calculations given the current limitations on integer representations.
- Optimizing gas costs to reduce transaction fees.
- Managing contract access control, incident response mechanisms, upgradeability (if applicable) through proxy contracts or governance mechanisms.
- Using security tools such static analysis, fuzzing, formal verification tools to detect vulnerabilities before deployment.
- Conducting internal security reviews and test coverage prior to third-party audits to ensure a baseline level of security.

Example: A developer deploying a decentralized finance (DeFi) lending contract must ensure that users cannot withdraw more assets than they deposited by implementing strict access controls and mathematical validation checks.

10.1.2. Validators

Validators are responsible for selecting transactions from the mempool, building blocks, and proposing them to the network for consensus. They prioritize transactions based on gas fees, typically including the most rewarding transactions first, which influences the ordering of smart contract interactions and can impact execution outcomes, particularly in DeFi and MEV-sensitive applications.

Before including a transaction in a block, validators perform several security verifications to ensure compliance with the protocol:

- Signature authentication to verify the sender's identity.
- Nonce verification to prevent replay attacks and ensure correct transaction ordering.
- Gas limit checks to confirm that the transaction has enough gas to execute fully.
- Sufficient funds verifications.

Validators also provide critical APIs for smart contract developers:

- Mempool access APIs allow dApps and monitoring tools to track pending transactions.
- Debug APIs help developers diagnose failed transactions by providing detailed failure reasons such as out-of-gas errors, invalid opcode usage, or smart contract reverts. This API is fundamental as the failure reason is not provided in the execution result returned by the node.

In addition to proposing blocks, validators are responsible for verifying transactions by re-executing all transactions in a proposed block before the network reach consensus on the validity of the block. They store and manage blockchain state, which include the accounts state such as wallets and smart contracts state. This ensures smart contracts can access relevant data during execution as well as precompiled contracts.

Different types of validator nodes provide varying levels of access to historical blockchain data, which can affect how smart contracts retrieve and process past transactions:

- Full nodes store complete blockchain state and recent transaction history but prune older data, limiting deep historical queries. As a consequence, obtaining older smart contract execution data may require using an archive node.
- Archive nodes retain all historical states of the blockchain, allowing full access to past transactions and smart contract executions. This is particularly useful for on-chain governance, historical pricing data in DeFi, and compliance auditing.
- Light nodes store only the most recent block headers, meaning they cannot directly query past contract states without relying on full or archive nodes for verification.

Additionally, Validators operate the Ethereum Virtual Machine (EVM), the execution environment responsible for processing smart contract logic. The EVM ensures that contracts run in a secure, isolated environment, preventing them from directly accessing or modifying blockchain infrastructure beyond their designated execution scope. This isolation guarantees deterministic execution, meaning that the same input always produces the same output, ensuring consistency across all validator nodes. By enforcing strict computational limits through gas fees and a clear mapping between opcode and their associated cost in terms of gas, the EVM mitigates infinite loops and excessive resource consumption, preserving network stability validators availability.

Validators also provide transaction cost estimation via their API, allowing users to anticipate gas fees, which fluctuate based on network demand and congestion.

From a smart contract perspective, validator node logic directly influences execution and ordering, state verification and immutability, and data availability, making them a crucial component in blockchain security and performance.

10.1.3. End Users

End users engage with smart contracts by initiating transactions through wallets or decentralized applications (dApps). These platforms enable users to sign transactions securely, which are then broadcast to the blockchain network. Upon inclusion in a block, these transactions trigger the execution of specific functions within the targeted smart contracts. Users are responsible for covering associated transaction fees, known as gas fees, to compensate validators for processing and confirming their transactions. This interaction allows users to participate in various decentralized services, such as token transfers, decentralized finance (DeFi) activities, governance voting or any other crypto asset service.

Relation with smart contracts:

- Sending transactions to interact with smart contracts (e.g., token transfers, NFT purchases, governance voting et protocol interactions).
- Paying gas fees to compensate validators for processing their transactions.
- Control the permissions given to smart contracts that will manage their assets. Eg: token approvals for DeFi protocol. If a smart contract is vulnerable, users risk losing their funds due to exploits or unintended behaviour.

Other Smart Contracts (Composability & Interoperability)

Smart contracts can interact with other contracts permissionlessly by design, meaning any deployed contract can call another contract's functions unless restricted. This allows for protocol composability but also creates security challenges.

Relation with smart contracts:

- Executing external function calls between smart contracts.
- Enabling automated interactions between protocols (e.g., lending protocols integrating with liquidity pools).

Example: A decentralized exchange (DEX) contract may interact with a stablecoin contract to facilitate swaps. If the stablecoin contract has vulnerabilities, it could affect the entire exchange system.

10.1.4. Decentralized Organizations (DAOs & Multi-Signature Wallets)

Decentralized organizations use on-chain governance to manage smart contracts collectively. Instead of a single owner, multiple participants control decisions through voting mechanisms or smart contract multi-signature wallets, or off chain MPC wallets.

Relation with smart contracts:

- Deploying and managing governance logic.
- Deploying and managing smart contracts as a collective entity.
- Using wallets for fund management.
- Voting on protocol upgrades and parameter changes in governance contracts.

10.1.5. Blockchain Protocol Developers (Precompiled Contracts & EVM Upgrades)

Blockchain protocol developers are responsible for maintaining and upgrading the execution environment in which smart contracts are executed. Also, they implement precompiled contracts to optimize complex operations and introduce new opcodes enabling new capabilities that can be used by smart contracts.

Relation with smart contracts:

- Implementing precompiled contracts for efficient cryptographic functions (e.g., ecRecover for signature verification) that are executed by the node instead of being executed in the EVM.
- Adjusting gas pricing for opcodes, which affects smart contract execution costs and might break smart contract logic in the case smart contracts fix the amount of gas that an external contract is allowed to use during an external call.
- Upgrading the Ethereum Virtual Machine (EVM) to improve security and execution efficiency.

11. Dependency Description

11.1.1. Dependencies

The evaluated ERC-20 contract inherits from standardized OpenZeppelin interfaces and utility contracts, ensuring compliance with the ERC-20 token standard without modifying its core functionality. The following dependencies are integrated during compilation:

Dependency	Type	Description	Security Considerations

IERC20	Interface	Defines the core ERC-20 standard functions (e.g., *transfer*(), *approve*(), *balanceOf*()).	No security risk – Interface only, no executable logic.
IERC20Metadata	Interface	Extends ERC-20 with metadata functions (*name*(), *symbol*(), *decimals*()).	No security risk – Interface only.
Context	Utility Contract	Provides execution context utilities (e.g., msg.sender, msg.data).	No security risk – Utility only.
IERC20Errors	Interface (ERC-6093 Draft)	Defines standardized error messages for ERC-20 function failures.	No security risk – Interface only.

Key Observations

- All dependencies used are interfaces or utility contracts—no external logic is executed, nor external libraries deployed on chain.
- The evaluated ERC-20 contract follows OpenZeppelin's standard implementation, without modifications, ensuring no deviations from the industry-standard security model.
- All dependencies are bundled during compilation and deployed with the contract, meaning no external contract calls occur at runtime.

11.1.2. Security Implications of Immutable Dependencies

Once deployed, a smart contract cannot be modified or updated, ensuring that external dependencies do not introduce vulnerabilities post-deployment. However, if a contract relies on an on-chain dependency deployed behind a proxy pattern, it remains upgradeable. This can include Solidity libraries or other contracts.

Using upgradeable dependencies is strongly discouraged unless the project fully controls and manages these dependencies. Uncontrolled upgrades can alter dependency behavior, potentially introducing vulnerabilities that compromise the security of the audited smart contract.

Unlike traditional software that can receive patches, all code in a non-upgradeable contract is bundled into the deployed bytecode, preventing from dynamically altering behavior.

12. Description of a typical user

Smart contracts operate within a decentralized and highly interconnected ecosystem, engaging a broad range of users with distinct interaction models and security considerations. Given the varied nature of smart contract interactions, this section identifies the principal user categories and analyses their security implications.

The following user groups are considered the primary stakeholders of the system under evaluation:

1. End Users (Investors and Participants)
2. Centralized Platforms Managing Assets for Third Parties
3. Decentralized Application (dApp) Integrators

This classification ensures a comprehensive assessment of the security risks associated with each type of user and their interactions with the smart contract system.

12.1.1. End Users (Investors and Participants)

12.1.2. Description

End users interact directly with smart contracts using self-custodial wallets, such as hardware or software wallets, where they retain full control and responsibility for managing their private keys. Their primary activities include:

- Acquiring, selling, and transferring tokens via decentralized exchanges (DEXs).
- Engaging in DeFi activities, including staking, lending, yield farming, and airdrop participation.
- Transferring tokens to other wallets.
- Bridging assets across different blockchains.

These users interact with smart contracts either programmatically or through web applications that facilitate transactions via self-custodial wallets. Such interfaces provide access to decentralized services like DEXs, lending protocols, and on-chain financial tools.

Their security depends not only on the integrity of the smart contracts they interact with but also on their own operational security practices. This includes securing private keys, using safe browsing environments, ensuring software integrity, and critically assessing the security posture of smart contract and web application before interaction.

12.1.3. Security Considerations

- **Phishing Attacks**:
  End users may be targeted by malicious actors through fraudulent websites, social engineering, or compromised interfaces that aim to extract private keys or make use sign unintended actions.
  - *Mitigation:* Using advanced simulation enforcing mechanisms, using secure wallet and vetted interfaces, scam databases, domain verification tools for decentralized applications, and enhanced user education on phishing threats.

- **Malicious Contract Approvals**:
  Users may inadvertently grant excessive permissions to smart contracts, enabling unauthorized asset transfers.
  Mitigation: Limited token approvals, regular allowance granting and revoking, monitoring of cotnracts and social feeds for compromised contracts, and interactions only with audited and trusted contracts.

- **Vulnerable Contracts:**
  Users may incur financial losses if a smart contract in which their funds are deposited contains vulnerabilities or gets compromised. Various attack vectors and security flaws could be exploited to compromise assets, including reentrancy attacks, arithmetic calculation errors, logic errors, or improperly implemented access controls.
  *Mitigation***:** Conduct thorough due diligence on the security of projects, including reviewing comprehensive audits, active bug bounty programs, multiple independent security assessments, and verifying the legitimacy and expertise of the development team behind the protocol.

12.1.4. Centralized Platforms Managing Assets for Third Parties (VASPs)

12.1.5. Description

Centralized platforms, including custodial exchanges, asset management services, and institutional financial entities, interact with smart contracts on behalf of their users. While these platforms remove the operational burden of private key management—an advantage for less tech-savvy individuals—this also introduces custodial risk. Users' funds are fully dependent on the platform's security, financial stability, and governance. In cases of insolvency or mismanagement, as seen with the collapse of FTX and other centralized platforms, users risk losing access to their assets, particularly in the absence of regulatory protections.

These platforms can, however, serve as a secure gateway to DeFi, offering users access to decentralized financial services through their own interfaces, managing funds mostly off-chain. By conducting internal due diligence, security audits, and risk assessments, they can provide a more regulated and structured way for users to engage with DeFi protocols while mitigating certain risks associated with direct smart contract interactions.

12.1.6. Security Considerations

Custodial Risks and Asset Concentration:

The centralized storage of private keys and digital assets creates a high-value attack vector, exposing platforms to potential breaches, insider threats, or operational failures.

Mitigation: Implementation of secure management of keys, multi-signature authentication, cold storage solutions, and rigorous access control policies. Also, selecting regulated service providers would provide a stronger safety net.

Third-Party Smart Contract Dependencies:

Centralized platforms integrate external smart contracts to provide further services such as in DeFi: lending, staking or other on-chain services which introduces risks of transparency. Since the platform's responsibility is indirectly involved, these risks should be minimized but not entirely eliminated, as centralized platforms do not have full control and may be unable to provide protection if the underlying services are compromised.

Mitigation: Ensure the platform conducts comprehensive security audits of third-party contracts, implements real-time monitoring, and follows secure integration frameworks to mitigate potential threats.

12.1.7. Decentralized Application (dApp) Integrators

12.1.8. Description

Decentralized applications (dApps) enhance the functionality of existing smart contracts by providing user-friendly web interfaces, automated workflows, and additional abstraction layers that simplify user interactions with their smart contracts. These applications include decentralized exchanges (DEXs), lending protocols, and automated market makers (AMMs), all of which rely on base-layer smart contracts to facilitate financial transactions and asset management without intermediaries.

However, it is important to note that ownership of the underlying smart contracts is often delegated to another entity, such as a foundation, while the web application providing services and connectivity to those contracts may be managed by a separate company, splitting responsibilities between different organizational structures.

Therefore, the term Decentralized Application can also refer to the on-chain smart contracts that implement crypto asset services. For example, decentralized exchanges (DEXs) enable permissionless market creation, allowing any user with a crypto wallet to list and trade ERC-20 tokens without requiring approval from a centralized entity.

12.1.9. Security Considerations

For decentralized application that offer their smart contract services via their web app.

- **Web App Security**: Web applications are vulnerable to data manipulation, injections, and API abuse. Attackers may also attempt to modify transaction payloads from the front end or directly target APIs to gain unauthorized access to restricted services.
    - *Mitigation:* ensure compliance with OWASP best practices and standards.

- **Dependency and Supply Chain Attacks**: Malicious modifications of Web3 libraries can introduce vulnerabilities, potentially compromising user transactions or injecting harmful logic into dApps.
    - *Mitigation:* ensure the projects use dependency integrity checks, security scanners, and regular audits to detect and mitigate risks.

As of for the smart contract components of decentralized applications we have:

- External Dependency and Systemic Risks: Smart contracts that depend on third-party services are vulnerable if those services are compromised. For example, an oracle providing price feeds could be manipulated, impacting contract operations.

    - Mitigation: Perform comprehensive audits before integration, enforce strong and standardized operational security, and establish incident response countermeasures and clear communication channels to handle potential threats effectively.

- **Smart Contract Upgradeability Risks**: Unauthorized or malicious upgrades can introduce vulnerabilities.
    - *Mitigation:* Enforce governance-controlled upgrades, time-locked execution, and multi-signature approvals.

- **Access Control and Permission Management**: Compromised admin privileges can lead to fund theft or protocol manipulation.
    - *Mitigation:* Use role-based access control (RBAC), multi-signature authentication, and restricted admin access.

- **Interacting with Unverified Contracts**: Calling external unverified or proxy contracts can expose funds to exploits.
    - *Mitigation:* Require contract verification on online services and control measures to prevent cascading impacts.

13. Description of the technical operating environment

14. Asset to protect

In this analysis, we will focus on OpenZeppelin's ERC-20 implementation. Regarding EVM smart contracts, the analysis can be divided into two parts

14.1.1. Analysing the Specificities of ERC20

14.1.2. Core Smart Contract Assets (Common to All Contracts)

These assets are present in all smart contracts, regardless of the specific use case, application or type of crypto asset they represent or represent

14.1.3. Contract Bytecode

The deployed bytecode represents the executable logic of the contract. Its integrity is fundamental to ensure the contract functions as intended. Any unauthorized modification of the bytecode could lead to unexpected behaviour, vulnerabilities, or loss of funds.

- **Security Objective:** *Integrity:* The bytecode associated with an address must not be modified after deployment, except through authorized upgrade mechanisms such as the proxy architecture.

14.1.4. Contract State Data

The contract's state, stored as variables on the storage database, is a critical asset. This data reflects the current state of the contract and its managed assets. Protected state data includes:

Contract Variables: Variables and datas structures 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.

Contract Logic (Functions): Functions define the logic and behavior of the smart contract, enabling interactions such as token transfers, approvals, and administrative operations.

Security Objectives:

Integrity: Functions must enforce proper access control and validation to prevent unauthorized modifications or unintended behavior.

Reliability: Functions should execute deterministically and handle edge cases to prevent failures or unexpected states.

Efficiency: Functions should be optimized to minimize gas consumption, avoid unnecessary complexity and ensure service availability as the service scales.

14.1.5. Event Logs

Event logs emitted by the smart contract are considered assets. These logs provide an auditable record and direct access to the contract's activity and are essential for tracking transactions and state changes. Their integrity is important for accountability.

Security Objective:

- Integrity*:* Event logs must accurately reflect the actions performed by the contract and must not be forgeable or modifiable.

14.1.6. Gas Token (ETH) Stored in the Contract

Gas Token held by the smart contract is a critical asset, as it can be used for transactions, paying fees, or transferred with other contracts. Proper management and security of ETH balances are essential to prevent unauthorized withdrawals, funds being locked in case the contract lack the mechanisms to manage them.

Security Objectives:

- Integrity: ETH balances must not be modified or accessed by unauthorized actors. Withdrawals and transfers should only occur under explicitly defined conditions.
- Availability: The contract must ensure ETH remains accessible for legitimate operations, preventing unintended locks due to logic errors or external dependencies.
- Auditability: All ETH-related transactions should be logged through event emissions, ensuring transparency and traceability for users and external monitoring services.

14.1.7. 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 the following audits: link to OpenZeppelin audits.

14.1.8. Scope of Evaluation

While OpenZeppelin provides a suite of additional smart contracts for functionalities such as access control, proxies patterns, and governance, this CSPN evaluation is strictly limited to the core ERC-20 implementation and its associated security considerations.

14.1.9. Reference Implementation Requirement

The ERC-20 standard itself is not a deployable contract; it serves as a contract interface that must be implemented within a new ERC20 implementation. Therefore, for the purposes of this evaluation, the following reference implementation is added to the scope of evaluation:

GitHub Repository: IPSProtocol/ERC20-CSPN

This implementation follows OpenZeppelin's industry-standard ERC-20 contract found in the following release:

GitHub Repository: OpenZeppelin/ERC20

14.1.10. Maturity of OpenZeppelin Contracts

OpenZeppelin's smart contract libraries are widely regarded as the industry standard, having undergone extensive auditing and advanced security testing. Their widespread adoption within the Ethereum ecosystem demonstrates a high level of trust in their security.

However, as this is the first application of the CSPN methodology to smart contracts, we want to establish a foundational evaluation on a widely used and well-understood contract before expanding to more complex implementations evaluation. The ERC-20 contract serves as an ideal starting point, providing a reference implementation that supports the understanding, validation, and future adoption of the CSPN methodology in blockchain security assessments.

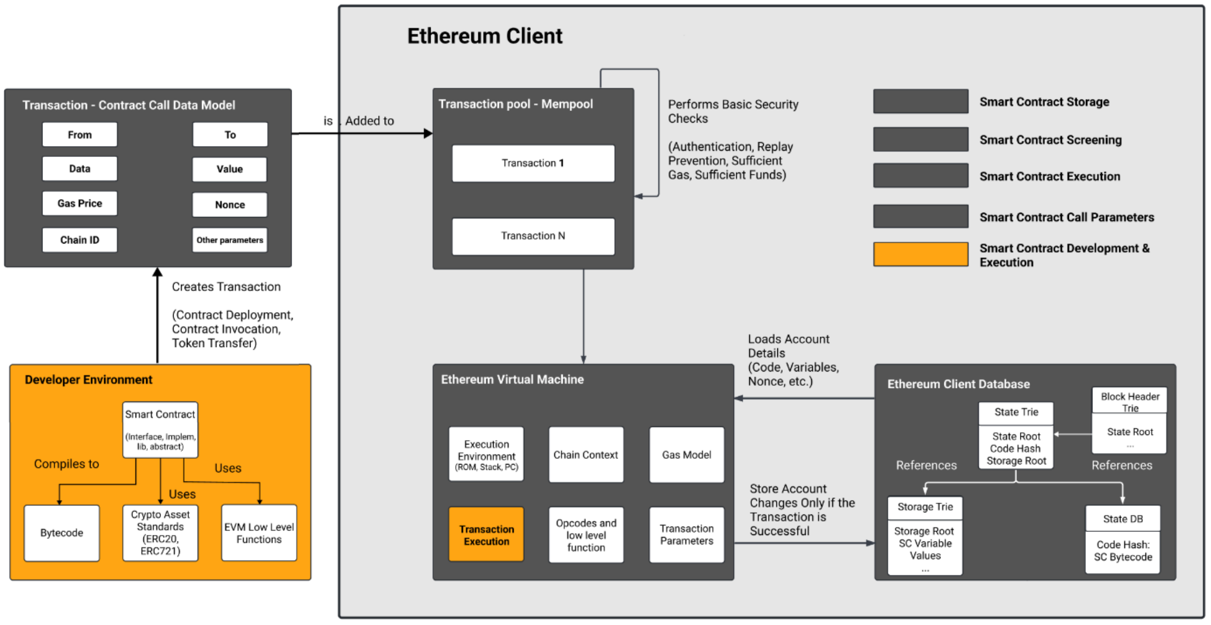
Ethereum Client
Transaction - Contract Call Data Model
From
To
Data
Value
Gas Price
Nonce
Chain ID
Other parameters
is Added to
Transaction pool - Mempool
Transaction 1
Transaction N
Performs Basic Security Checks
(Authentication, Replay Prevention, Sufficient Gas, Sufficient Funds)
Smart Contract Storage
Smart Contract Screening
Smart Contract Execution
Smart Contract Call Parameters
Smart Contract Development & Execution
Creates Transaction
(Contract Deployment, Contract Invocation, Token Transfer)
Developer Environment
Smart Contract
(Interface, Implem, lib, abstract)
Compiles to
Uses
Uses
Bytecode
Crypto Asset Standards (ERC20, ERC721)
EVM Low Level Functions
Ethereum Virtual Machine
Execution Environment (ROM, Stack, PC)
Chain Context
Gas Model
Transaction Execution
Opcodes and low level function
Transaction Parameters
Loads Account Details (Code, Variables, Nonce, etc.)
Store Account Changes Only if the Transaction is Successful
Ethereum Client Database
State Trie
State Root Code Hash Storage Root
Block Header Trie
State Root ...
References
References
Storage Trie
Storage Root SC Variable Values ...
State DB
Code Hash: SC Bytecode

Figure 3: Definition of the Evaluation Scope (in Orange)

Asset ID	Assets	Storage / Access Location	Integrity	Authenticity	Consistency
A1	Contract Bytecode	Account Trie / EVM	Yes	Yes	Yes
A2	Events Logs	Receipt Trie / EVM	Yes	Yes	Yes
A3	User Balances	Account Trie / EVM	Yes	Yes	Yes
A4	User Allowances	Account Trie / EVM	Yes	Yes	Yes
A5	Token Supply	Account Trie / EVM	Yes	Yes	Yes

A6	Token Name	Account trie / EVM	Yes	Yes	Yes
A7	Token Symbol	Account Trie / EVM	Yes	Yes	Yes

15. Description of Environnmental 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 and secure EVM implementation.
H5	The cryptographic algorithms utilized in client' implementation, including those in the Merkle Patricia Trie and the associated Merkle proofs within the Ethereum client, which ensure the integrity of smart contract code and state, are considered out of scope for this analysis. This evaluation focuses exclusively on the implementation of the smart contract itself and its business logic, rather than the underlying cryptographic mechanisms securing Ethereum's data structures.
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 0.8.20 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.
H10	The evaluation will not assess the validity or appropriateness of the ERC-20 standard implementation compared to the EIP20.
H11	The ERC20 Reference implementation will have fixed total supply

16. 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 can could reduce costs for end users. High development and operational costs make this approach is usually not adopted.

Threat ID	Threat	Description	Assurance Level
T1	Balance integrity	Ensures balances remain accurate and cannot be arbitrarily modified due to contract vulnerabilities.	High
T2	Allowance integrity	Ensures the ERC-20 approval system cannot be bypassed or manipulated.	High
T3	Unauthorized Token Transfer in Delegated Token Transfer	Preventing unauthorized transactions due to faulty or missing allowance checks.	High
T4	Ensures only the rightful owner can initiate direct transfers.	Ensures only the rightful owner can initiate transfers.	High
T5	Unauthorized Token Metadata Modification	Ensuring the ERC-20 token metadata remains immutable after deployment.	High
T6	Unauthorized Token Supply Modification	Preventing unauthorized modifications to the total token supply.	High
T7	Unauthorized minting	Ensuring tokens cannot be minted after the deployment.	High
T8	Incorrect Token Minting	Ensuring minted tokens are sent only to valid, pre-determined addresses.	High
T9	Token Burn Integrity	Ensures that when tokens are burned, user balances and total supply are correctly updated, preventing inconsistencies or supply miscalculations.	High
T10	Unauthorized Token Burn	Ensures that only the token owner can burn their own tokens and that no other account can burn tokens belonging to another user.	High
T11	Meaningful and unambiguous logging	Ensures that contract events accurately log all critical changes, maintaining consistent logging, auditability and preventing ambiguous or misleading event logs that could disrupt dApp integrations, or hide underlying issues.	Low
T12	Bytecode Modification	Ensures that the deployed smart contract bytecode remains immutable and cannot be	High

		altered post-deployment. While Ethereum enforces immutability, risks may arise from upgradeable contract mechanisms, proxy contracts, or deployment vulnerabilities allowing unauthorized modifications.	

Possible Exploitation Scenarios for Threats:

T1: Balance Integrity Flaw

- Flaw: Balance is not updated correctly after a transfer.
- Exploitation: A user's overall balance increases when they transfer tokens to themselves.

T2: Allowance Integrity Flaw

- Flaw: Allowance is not updated after a transfer, allowing unauthorized reuse of funds.
- Exploitation: A third-party attacker continues to access and drain the main user's funds after an approval was intended to be used only once.

T3: Unauthorized Token Transfer in Delegated Token Transfer

- Flaw: The transferFrom() function allows transfers even when an allowance check fails due to a poorly structured conditional statement.
- Exploitation: The attacker calls transferFrom() without an actual approval, and due to a misplaced require() condition, the transfer still goes through.

T4: Unauthorized Direct Transfers

- Flaw: The contract lacks a sender validation check, allowing forced transfers.
- Exploitation: An attacker forces transfers from user accounts, emptying their funds.

T5: Unauthorized Token Metadata Modification

- Flaw: The contract includes an unprotected function that allows metadata changes.
- Exploitation: A malicious user changes the token name and symbol, supporting misleading campaigns.

T6: Unauthorized Token Supply Modification

- Flaw: The contract lacks restrictions on modifying totalSupply.
- Exploitation: A malicious user increases or decreases the total supply, impacting tokenomics and token value.

T7: Unauthorized Minting

- Flaw: The minting function is callable by unauthorized users.
- Exploitation: The attacker calls mint() from an external contract, generating an unlimited number of tokens.

T8: Incorrect or Unauthorized Token Minting

- Flaw: The contract does not validate recipient addresses when minting tokens.
- Exploitation: An attacker mints tokens to an address that cannot manage them, permanently locking the supply.

T9: Token Burn Integrity Flaw

- Flaw: The contract does not correctly update totalSupply after burning tokens.
- Exploitation: A user burns their tokens, but since totalSupply is not updated, the burned tokens remain in circulation, impacting token value.

T10: Unauthorized Token Burn

- Flaw: The contract does not verify ownership before burning tokens.
- Exploitation: A hacker calls burn() on another user’s balance, effectively destroying their assets.

T11: Meaningful and Unambiguous Logging Failure

- Flaw: The contract does not correctly emit event logs for transfers and approvals.
- Exploitation: Exchanges and off-chain platforms rely on on-chain events to update their accounts. A hacker exploits the lack of event logging to mislead and steal funds from a centralized exchange as happened in the past.

T12: Bytecode Modification Risk

- Flaw: The contract is upgradeable but lacks access control on proxy modifications.
- Exploitation: A malicious deployer modifies the contract implementation post-deployment, injecting arbitrary logic that allows infinite minting or direct fund transfers.

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

17. Security Functions Description

Security Function ID	Security Function	Description	Category
SF1	No Direct Balance Modification	Token balances cannot be modified directly. They are only updated through ERC-20 functions (transfer(), transferFrom()), ensuring controlled state changes.	State Integrity
SF2	Secure Balance Update Logic in transfer() and transferFrom()	These functions correctly implement sender ownership checks and balance verification, ensuring that transactions execute only if sufficient funds exist and that unauthorized parties cannot alter balances.	State Integrity, Access Control
SF3	Allowance Control	approve(), increaseAllowance(), and decreaseAllowance() ensure only the owner can modify allowances, preventing unauthorized spending.	Access Control
SF4	Allowance Verification in transferFrom()	Before a transfer, transferFrom() verifies that the spender has sufficient allowance, preventing unauthorized transfers and performing the correctly updating the variable values.	State Integrity
SF5	Ownership Verification via Transfer Function	Ensures that only the token holder can initiate transfers when using the Transfer Function	Access Control
SF6	Immutable Token Metadata	Token name, symbol, and decimals are immutable and cannot be modified after deployment.	Immutability
SF7	Fixed Total Supply Post-Deployment	The totalSupply variable cannot be changed after contract deployment	State Integrity
SF8	Minting Restricted to Deployment	_mint() function is not accessible and only used in the constructor, preventing any future token creation.	Access Control
SF9	Minting Limited to Deployment Parameters	Minting capabilities ensures tokens are only minted to a value and sent to an address chosen by the deployer.	State Integrity
SF10	Correct Burn Logic Implementation	Ensures the burn() function correctly updates the user's balance and the total supply,	State Integrity

		preventing inconsistencies in the contract's internal state.	
SF11	Burn Authorization via Transaction Sender	Ensures that only the owner of the tokens (i.e., msg.sender) can initiate a burn operation, preventing unauthorized burning of another user's tokens.	Access Control
SF12	Bytecode immutability	Thanks to Ethereum data structure and consensus regarding the blockchain state, contract bytecode cannot be altered without the majority of the blockchain rejecting the proposed changes	Integrity

18. Coverage Matrix

18.1.1. Threats and Assets

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

	A1	A2	A3	A4	A5	A6	A7
T1			I				
T2				I			
T3		I	I	Au, I			
T4		Au, I	Au, I				
T5						I	I
T6					I		
T7					I		
T8							
T9			I		I		
T10			Au, I				
T11		C					
T12	I						

18.1.2. Confidentiality Considerations in the Evaluation

Confidentiality is not a primary security criterion in the context of this CSPN evaluation because Ethereum is a public blockchain, where all contract data, state, transactions, and interactions are inherently transparent and accessible to all network participants. Unlike traditional web applications or private databases, smart contracts deployed on Ethereum store all state variables and transaction details on a globally shared ledger.

18.1.3. Publicly Accessible Data by Design

Ethereum's transparency ensures that:

- Contract bytecode, state variables, and transaction history are fully public. This allows for trustless verification but inherently removes any expectation of confidentiality.
- Event logs, which are stored in the blockchain's receipt trie, can be queried by any participant, making on-chain interactions fully auditable.
- Balances, allowances, and transfers are visible through standard Ethereum node queries.

18.1.4. Private Variables Are Not Truly Private

While Solidity allows for "private" visibility modifiers on variables, this only restricts direct access from other smart contracts, not from external observers. Since Ethereum nodes store all contract state data in the state trie, any party with access to an Ethereum node can still retrieve private variables by inspecting the contract storage directly.

Thus, confidentiality, in the scope of a ERC-20 implementation is not applicable in this evaluation, as data secrecy cannot be enforced within the Ethereum execution model.

18.1.5. Threats and Security Function Matrix

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

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12
SF1	X											
SF2	X			X								
SF3		X	X									
SF4	X	X	X									
SF5				X								
SF6					X							
SF7						X						
SF8						X	X					
SF9							X	X				
SF10									X			
SF11	X									X		
SF12												X

The Threat-Security Function Matrix establishes a comprehensive security framework by systematically mapping identified threats to their corresponding security functions.

18.1.6. Threat Coverage

The evaluation confirms that:

- While not impacting the usage of the crypto asset, T11 is not addressed by the ERC20 implementation.
- Each identified threat, but T11, is addressed by at least one security function, ensuring that no unmitigated risks remain within the scope of this assessment.
- The security functions effectively enforce state integrity, access control, limited accessibility, which are critical to preserving the contract's expected behaviour and preventing unauthorized modifications.

18.1.7. Event Logging Ambiguity in ERC-20: Implications for dApps and Indexers

18.1.8. Context: Logging in ERC-20

The ERC-20 standard defines two key events for tracking token movements:

- Transfer(address indexed from, address indexed to, uint256 value): Emitted whenever tokens are transferred between addresses.
- Approval(address indexed owner, address indexed spender, uint256 value): Emitted when an allowance is set or changed.

These events enable external dApps, wallets, and blockchain indexers to monitor transactions efficiently without having to query contract storage directly.

18.1.9. Ambiguity in Mint and Burn Events

Unlike explicit *mint*() or *burn*() functions in some other token standards (e.g., ERC-721 for NFTs), the evaluated ERC-20 implementation does not use dedicated *mint* or *burn* events. Instead, token minting and burning are implicitly represented using Transfer events. Given that Mint and Burn have a different impact on the token, such as changing the token supply, it would be a best practice to differentiate them clearly in the logging logic.

- Minting Tokens:
  - When new tokens are created, the contract typically calls *transfer*(0x0, to, amount), simulating an incoming transfer from the null address (0x0).

    - This makes it difficult to distinguish between a genuine mint operation and an airdrop or accidental user transfers from 0x0.
- Burning Tokens:
    - When a user burns tokens, the contract usually calls transfer(from, 0x0, amount), simulating a transfer to the null address.

18.1.10. Security and Indexing Implications

1. Indexing Confusion:
    a. Indexers and dApps parsing ERC-20 logs may misinterpret burn transactions as erroneous transfers to 0x0 instead of intentional burns.
    b. Conversely, minting events might be confused with manual balance increases or airdrops.
2. Logging enabling bug detection
    a. If the condition checks are incorrectly implemented in these functions, logs would enable straightforward detection.
3. Potential Frontend Issues:
    a. Web applications displaying transaction history may fail to accurately label token mints and burns if they rely solely on Transfer events.
4. Performance & Query Optimization:
    a. dApps and indexers that wish to retrieve only minting or burning transactions must scan all Transfer events and apply filtering based on event parameters. This increases the computational burden on querying systems and reduces efficiency.

Finally, it is important to note that the zero address (0x0) is a special address used in mint and burn events. Due to the underlying cryptographic primitives used for generating Ethereum addresses, it is assumed that obtaining the private key associated with the zero address is practically infeasible.

However, in a hypothetical scenario where this becomes possible, an attacker could gain access to all burned tokens, as their sent to the zero address, and move funds freely. A transfer originating from the zero address would generate a Transfer event with the sender set to 0x0.

In the current implementation, and in this hypothetical scenario, a transfer from the zero address would be interpreted as a minting event, effectively concealing the exploit while also leading to inconsistent data for applications that rely on contract-emitted events.

18.1.11. Considerations for Future Implementations

Although this evaluation does not modify the ERC-20 reference implementation, the following best practices are recommended for developers implementing ERC-20-based tokens:

- Overriding the implementation of mint() and burn() function to introduce associated events. to improve clarity and performance for indexers and dApps.
- Ensure frontend applications differentiate burns from accidental transfers to 0x0 using internal logic.
- Consider implementing fail-safes for accidental burns, such as requiring explicit user confirmation before transferring tokens to 0x0.

18.1.12. Conclusion

While this ambiguity does not pose a direct security risk to the smart contract itself, it introduces interpretation challenges for dApps, blockchain indexers, and end-users analyzing transaction histories while also potentially hiding underlying issues. The reliance on a single Transfer event for regular transfers, minting, and burning requires additional processing to differentiate between these operations, increasing complexity and reducing efficiency in querying on-chain data.

Future token standards could enhance clarity and performance by introducing explicit Mint and Burn events. This would:

- Eliminate ambiguity in transaction logs.
- Improve query efficiency, allowing dApps and analytics platforms to retrieve minting and burning events without processing the entire transfer history.
- Ensure consistency across the ecosystem, aligning with other token implementations that already distinguish between these events in their logging mechanisms.

Standardizing explicit event differentiation would contribute to a more secure, structured and efficient on-chain data environment, enhancing the usability and transparency of blockchain applications.

Form references: IPSP.RAL.SC-2023
Version: 1.0

Written by:

Role	Name	Date (DD/MM/YYYY)	Signature
IoT Product Security Evaluator	GEDEON Paul	10/02/2023	
IPSProtocol Lead	MUGNIER Jean-Loïc	20/02/2025	

Approved by:

Fonction	Nom	Date (DD/MM/YYYY)	Signature
RAL Lab Director	KHALIL Ayman	16/06/2021	
IPSProtocol Lead	MUGNIER Jean-Loïc	23/03/2024	

History:

Version	**Redactor**	**Date (DD/MM/YYYY)**	**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 analysing 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	Specify the product, 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.
0.8	MUGNIER Jean-Loïc	23/12/2024	Conclusions and external review
0.9	MUGNIER Jean-Loïc	17/02/2024	Integrating external final reviews
1	MUGNIER Jean-Loïc	13/03/2024	Final version

END OF THE DOCUMENT