SCI

Secure Contract Interactions

Whitepaper

v2.0 Dec 2024

SCI Website

Table of contents

Abstract

The Secure Contract Interaction (SCI) Protocol is an open-source initiative aimed at enhancing security within the web3 ecosystem. Recognizing the increasing risk of interacting with malicious smart contracts, SCI introduces a decentralized verification system allowing domain owners to authorize specific smart contracts to interact with their domains.

Its flexibility and simple architecture supports easy integration for users, wallets, security apps, and on-chain contracts. SCI is open-source, free to use, and committed to providing a vital layer of protection for users dealing with smart contracts in the web3 ecosystem.

Introduction

Secure Contract Interaction (SCI) is an open-source protocol designed as a public good to improve security within web3.

We aim to bolster security measures within the web3 ecosystem by establishing an on-chain registry that allows owners to verify which smart contracts should be allowed to interact with their web domains.

SCI's purpose is to verify that websites are interacting with validated and authorized smart contracts minimizing user risks and exploits.

Why do we exist?

In the realm of web3, ensuring robust security is a paramount challenge that must be addressed to facilitate the onboarding of the next billion users. Numerous incidents, such as those involving Balancer, Badger, and KyberSwap, highlight the vulnerability of users interacting with malicious contracts unknowingly.

Quote from **KyberSwap**'s exploit:

"Using the injected malicious script via GTM, the hackers made users approve their funds and sent them to the hacker's address"

In response to these threats, the SCI protocol introduces a verification system. Here, domain owners (for example the owner of kyberswap.com) can add all contracts permitted to interact within their domains and app users can get insights on whether the contract they are engaging with is verified (within the domain) or not.

The SCI protocol stands as a public good, offering a completely open-source, permission-less, and free-to-use solution. By providing a unified system that exposes on-chain data, SCI serves as an essential layer of protection for users interacting with smart contracts across the entire ecosystem.

How it works

Key components

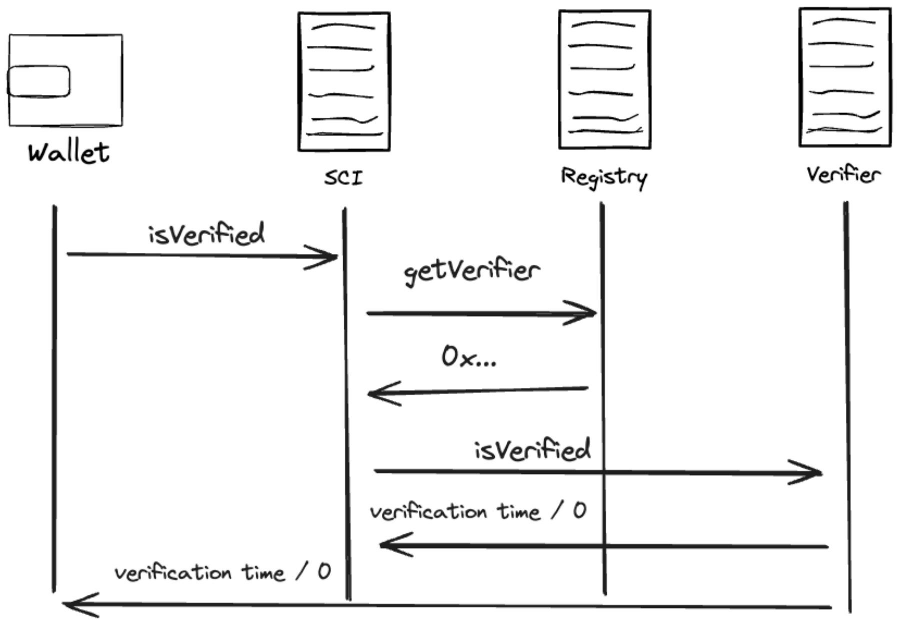
There are four key smart contract:

- **SCI**: It is used to interact with the protocol
- **Registry:** It keeps track of domains, owners and Verifiers
- **Verifiers**: It resolves if a contract is allowed on the domain
- **Registrars**: It is designed to verify if an Ethereum address is the legitimate owner of a domain, even outside the protocol's boundaries. These contracts are also responsible for registering domains within the Registry.

Verifying that a contract can interact with a domain

When a user initiates a contract interaction within a web application hosted on a specific domain, the wallet verifies if the contract is allowed on that domain by consulting the SCI contract.

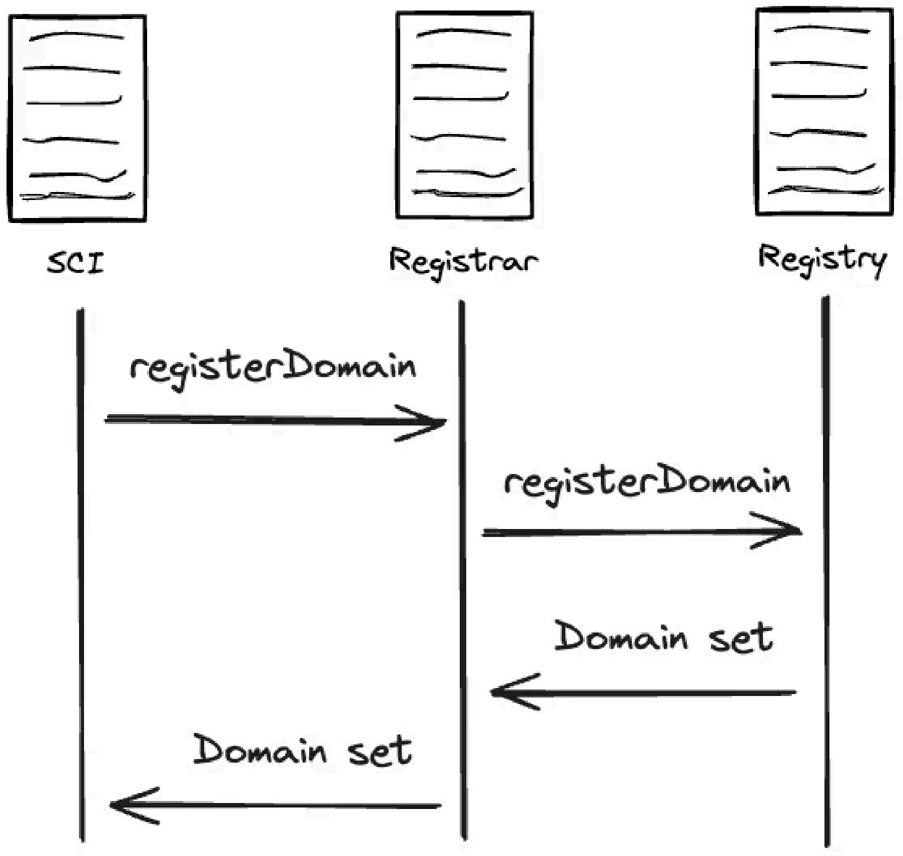
The SCI contract, in turn, queries the Registry to obtain the Verifier, and subsequently, it queries the Verifier to determine the validity of an address on that domain.



Registering a domain

When a domain owner seeks to register a domain for the purpose of verifying contract addresses, they use a **Registrar**. Upon successful execution, the **Registrar** will record the domain URI (in the form of a name hash) along with the address that owns it and the associated **Verifier** in the **Registry**.

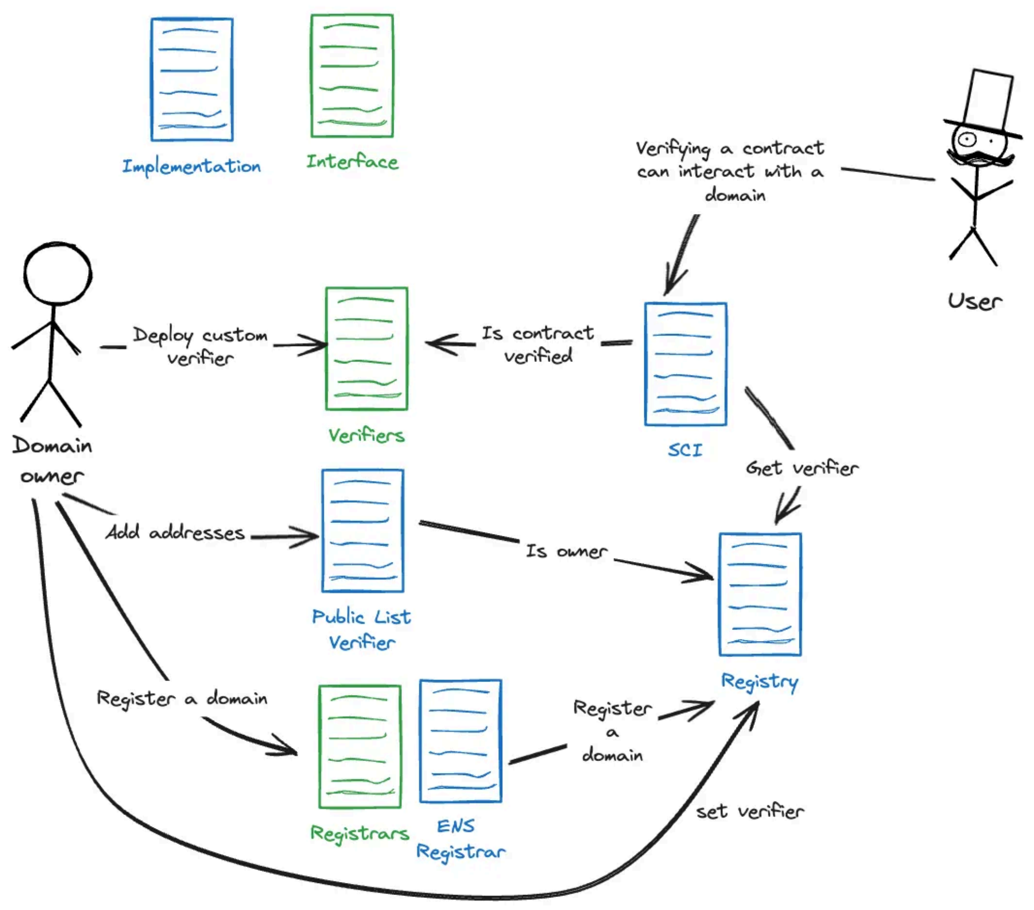
To register a domain, the owner of the domain sends a transaction to the **Registrar** contract, providing the desired domain and a **Verifier**. The parameters to register a domain may change depending on each Registrar implementation as they may have different requirements to validate the domain ownership.



Architecture

General Overview

The protocol is structured in different smart contract components:



SCI Contract

The **SCI contract** serves as an upgradable smart contract designed to streamline interactions within the protocol. It will continuously evolve by incorporating new features introduced in future updates.

Users of the protocol can implement their own logic independently and can interact with the protocol without using the **SCI Contract**. However, the contract is designed to simplify the integration process and enhance the user experience.

For instance, the verification process to determine if a contract is permitted to interact with a domain involves two separate calls—one to the **Registry** and another to the **Verifier**. The **SCI Contract** abstracts these complexities, providing a single function that enables users to easily verify if a contract is allowed to interact with a domain.

Registars

To ascertain domain ownership, the protocol can employ different strategies which include relying on established systems such as the Ethereum Name Service (ENS) or validating Domain Name System Security Extensions (DNSSEC) records on the blockchain. By implementing these strategies, SCI identifies, in a robust and decentralized way, that an address is the owner of a domain.

The Registrars don't require a specific implementation method since the arguments may vary for each registration. We call registrars to the Smart Contracts that have the REGISTRAR_ROLE in the **Registry**

As part of the initial rollout of the project, registrars are created and added to the registry by SCI's core team. The initial implementation introduces the ENS Registrar. However, future registrars can employ diverse strategies and the protocol won't rely solely on ENS for domain verifications.

ENS Registrar

The **ENS Registrar** takes the provided domain hash and subsequently retrieves the owner information from the ENS registry. Following this, it conducts a comparison with the sender parameter. If the owner of the node matches the sender parameter, the registrar registers the domain in the **Registry.**

More on ENS DNS registrar: https://docs.ens.domains/dns-registrar-guide

Verifiers

Verifiers are smart contracts that resolve which contracts are allowed on the domain.

To be a verifier, a smart contract needs to support the following interface:

```
/**
 * @title IVerifier
 * @dev Required interface of a Verifier compliant contract for the SCI Registry.
 * @custom:security-contact security@sci.domains
 */
interface IVerifier {
    /**
     * @dev Verifies if a contract in a specific chain is authorized
     * to interact within a domain.
     * @param domainHash The domain's namehash.
     * @param contractAddress The address of the contract trying to be verified.
     * @param chainId The chain where the contract is deployed.
     * @return a uint256 representing the time when the contract was verified.
     * If the contract is not verified, it returns 0.
     *
     * Note: The return timestamp is a best effor approach to provide the time when the contract
     * was verified. For verifiers that can't know when the contract was verified they could
     * return when the verifier was deployed.
     */
    function isVerified(
        bytes32 domainHash,
        address contractAddress,
        uint256 chainId
    ) external view returns (uint256);
}
```

They are only required to implement the `isVerified` function, so that the registry can determine whether the contract is verified for that domain or not. The result indicates the time when the contract was verified or returns 0 if it was not verified.

Public List Verifier

The protocol provides a **Public List Verifier** which is an address-to-domain mapping. It consists of a simple contract that stores the list of addresses permitted to interact with a specific domain. The contract looks up the list when determining if the contract is verified or not.

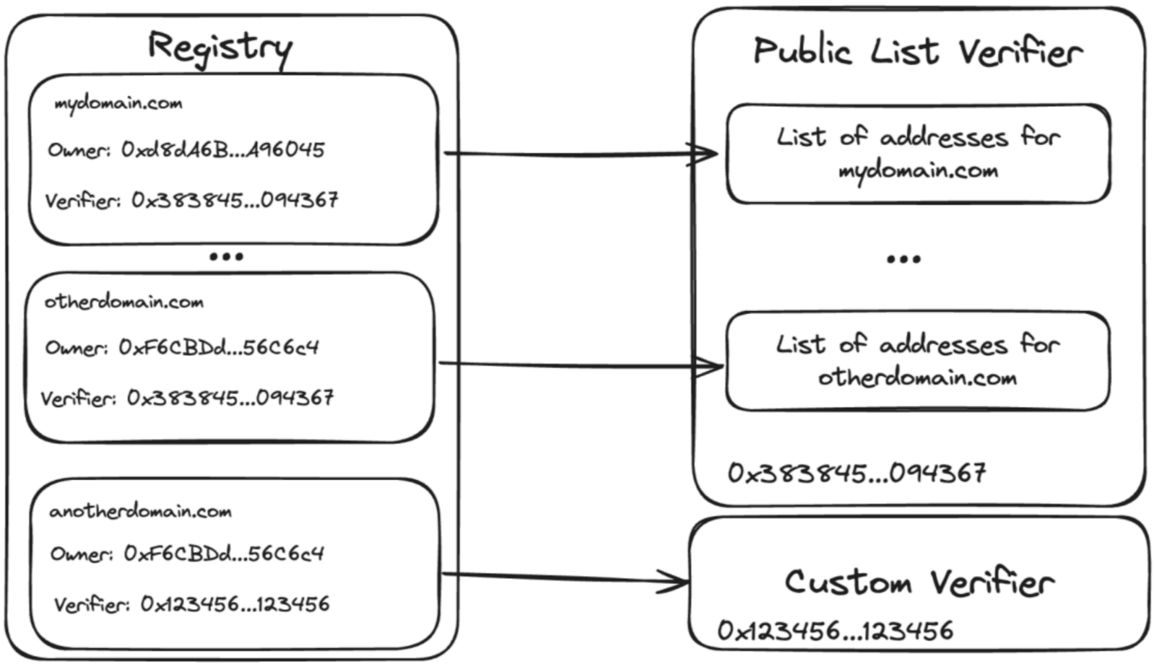
When adding an address, the **Public List Verifier** will confirm ownership of the domain through the **Registry**. The chain id needs to be provided. It can be set to max uint256 to signal that the contract is available on all chains.

Custom Verifiers

More complex verifications could be developed in the future. Any domain owner could create their own verification system. If necessary, verifiers could also implement ERC-3668 to perform off-chain verification logic.

Any custom verifier has the flexibility to inherit or combine functionality from various verifiers. This is why the protocol supports only one verifier per domain.

The following diagram gives a simple representation on how the **Registry** interacts with the **Public List Verifier** or a **Custom Verifier**.



Registry

The **Registry** smart contract records the domain owner's address and the associated **Verifier** for a specific domain.

Smart contract interface:

```
// SPDX-License-Identifier: AGPL-3.0
pragma solidity 0.8.28;

import {IVerifier} from '../Verifiers/IVerifier.sol';

/**
 * @title ISciRegistry
 * @dev This contract manages domain registration and verifiers. It uses role-based access control to allow
 * only authorized accounts to register domains and update verifiers.
 * The contract stores domain ownership and verifier information and allows domain owners to modify verifiers
 * @custom:security-contact security@sci.domains
 */
interface ISciRegistry {
    /**
     * @dev Emitted when a new `domain` with the `domainHash` is
     * registered by the `owner`.
     */
    event DomainRegistered(
        address indexed registrar,
        address indexed owner,
        bytes32 indexed domainHash
    );

    /**
     * @dev Emitted when the `owner` of the `domainHash` adds a `verifier`.
     *
     * Note: This will also be emitted when the verifier is changed.
     */
    event VerifierSet(
        address msgSender,
        bytes32 indexed domainHash,
        IVerifier indexed oldVerifier,
        IVerifier indexed newVerifie
    );

    /**
     * @dev Emitted when the owner of a `domainHash` is set.
     *
     */
    event OwnerSet(
        address msgSender,
        bytes32 indexed domainHash,
        address indexed oldOwner,
        address indexed newOwner
    );

    /**
     * @dev Returns the owner, the IVerifier, lastOwnerSetTime and lastIVerifierSetTime
     * for a given domainHash.
     * @param domainHash The namehash of the domain.
     */
    function domainHashToRecord(
        bytes32 domainHash
    )
        external
        view
        returns (
            address owner,
            IVerifier verifier,
            uint256 lastOwnerSetTime,
            uint256 lastIVerifierSetTime
        );
```

```
/**
 * @dev Register a domain.
 *
 * @param owner The owner of the domain.
 * @param domainHash The namehash of the domain being registered.
 *
 * Requirements:
 *
 * - Only valid Registrars must be able to call this function.
 *
 * May emit a {DomainRegistered} event.
 */
function registerDomain(address owner, bytes32 domainHash) external;

/**
 * @dev Same as registerDomain but it also adds a IVerifier.
 *
 * @param owner The owner of the domain being registered.
 * @param domainHash The namehash of the domain being registered.
 * @param verifier The verifier that is being set for the domain.
 *
 * Requirements:
 *
 * - Only valid Registrars must be able to call this function.
 *
 * Note: Most of registrars should implement this function by sending
 * the message sender as the owner to avoid other addresses changing or setting
 * a malicous verifier.
 *
 * May emit a {DomainRegistered} and a {IVerifierAdded} events.
 */
function registerDomainWithVerifier(
    address owner,
    bytes32 domainHash,
    IVerifier verifier
) external;

/**
 * @dev Returns true if the account is the owner of the domainHash.
 */
function isDomainOwner(bytes32 domainHash, address account) external view returns (bool);

/**
 * @dev Returns the owner of the domainHash.
 * @param domainHash The namehash of the domain.
 * @return The address of the owner or the ZERO_ADDRESS if the domain is not registered.
 */
function domainOwner(bytes32 domainHash) external view returns (address);

/**
 * @dev Returns the IVerifier of the domainHash.
 * @param domainHash The namehash of the domain.
 * @return The address of the IVerifier or the ZERO_ADDRESS if the domain or
 * the IVerifier are not registered.
 */
function domainVerifier(bytes32 domainHash) external view returns (IVerifier);
```

```
    /**
     * @dev Returns the timestamp of the block where the IVerifier was set.
     * @param domainHash The namehash of the domain.
     * @return The timestamp of the block where the IVerifier was set or
     * 0 if it wasn't.
     */
    function domainVerifierSetTime(bytes32 domainHash) external view returns (uint256);

    /**
     * @dev Sets a IVerifier to the domain hash.
     * @param domainHash The namehash of the domain.
     * @param verifier The address of the IVerifier contract.
     *
     * Requirements:
     *
     * - the caller must be the owner of the domain.
     *
     * May emit a {IVerifierAdded} event.
     *
     * Note: If you want to remove a IVerifier you can set it to the ZERO_ADDRESS.
     */
    function setVerifier(bytes32 domainHash, IVerifier verifier) external;
}
```

Namehash

Instead of using human readable domains, the Registry contract uses the namehash algorithm to transform them into 256-bit cryptographic hashes. This algorithm is the same used by ENS to represent domains in their architecture.

We call the namehash of a domain a domain hash

More on ENS namehash:
https://docs.ens.domains/contract-api-reference/name-processing

Applications

We can see multiple products that could leverage the protocol to enhance the security of their users. In this section we will list a few examples.

Wallets

Due to the nature of the protocol we propose, it is fairly simple to integrate this protection/prevention layer into any wallet.

An example of this can be seen using a Metamask snap that verifies whether the interaction with the contract is validated in the domain it is happening.

Furthermore, any wallet that can verify the domain where the call is taking place can do a security layer check pulling SCI information and providing the user with a safe and intuitive experience.

Standalone Security Applications

Applications can be built to easily verify contracts allowed to interact with specific domains, serving as a valuable additional security measure alongside what the wallet provides.

Any back-end service can use SCI before submitting any transaction and prevent any malicious or fraudulent transactions. SCI enables you to protect your own protocols, services and the infrastructure you offer to other businesses.

Examples of this would be our main SCI web page, where you can search the domains and the addresses that are verified, browser plugins or transaction relayers.

On-chain applications

Smart contracts can utilize our protocol before transferring funds to ensure the recipient address is verified, preventing losses.

Currently, several applications manage their funds, requiring the protocol team to handle asset collection or transfers regularly. Mistakenly sending funds to the wrong address, causing permanent loss can be avoided by using the SCI Protocol.
https://consensys.io/open-roles/5645608

Other Implementations

During our research phase for designing the protocol, we encountered some existing proposals such as ERC 6897 and ERC 7529, aiming to address similar issues. While these proposals offer valuable insights, we found certain limitations that could hinder their widespread adoption.

ERC-6897

Although resembling our design, this EIP lacks support for passing a domain during verification calls, which poses challenges for reusing Verifiers (or DRC as they are called in the EIP) across multiple domains. Consequently, deploying a separate Verifier for each domain significantly increases costs for domain owners. Employing a JSON file in the /.well-known root for double verification presents an intriguing idea that we might consider incorporating into our protocol in the future.

ERC-7529

This EIP proposes embedding contracts into a TXT record within a DNS domain and retrieving them via DoH (DNS over HTTP). However, this approach relies on a centralized provider for DNS over HTTP, introducing potential vulnerabilities such as data withholding or service downtime. In contrast, our on-chain registry leverages existing node infrastructure, enabling users to access information from any node provider including their own node. Additionally, ERC 7529's reliance on a simple list of addresses on-chain overlooks the composability benefits offered by smart contracts. Our Verifiers, being complex smart contracts, can provide additional security checks beyond mere address listings.

While the idea of smart contracts managing a list of allowed domains is intriguing, it fails to address the fundamental issue of potential vulnerabilities when frontends interact with malicious smart contracts. Malicious actors could manipulate the domain allowance within their contracts, undermining the security of the system.