hexens × SOCKET

Aug.23

SECURITY REVIEW REPORT FOR SOCKET

CONTENTS

CONTENTS

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.

AUDIT LED BY

KASPER ZWIJSEN

Head of Smart Contract Audits | Hexens

Audit Starting Date
02.08.2023

Audit Completion Date
04.08.2023

---

hexens × SOCKET

METHODOLOGY

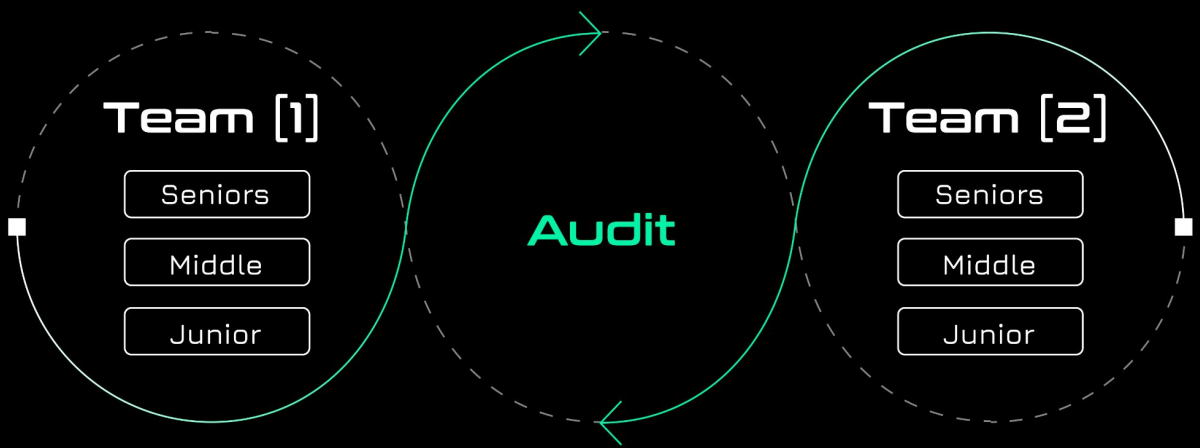
COMMON AUDIT PROCESS

Companies often assign just one engineer to one security assessment with no specified level. Despite the possible impeccable skills of the assigned engineer, it carries risks of the human factor that can affect the product's lifecycle.

Auditor* Audit

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

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 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 optimisations 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.

It's important to consider all types of vulnerabilities, including informational ones, when assessing the security of the project. A comprehensive security audit should consider all types of vulnerabilities to ensure the highest level of security and reliability.

EXECUTIVE SUMMARY

OVERVIEW

This audit covered Socket's new smart contracts for bridgeable tokens on their App Chain. It included a Controller for minting/burning tokens on the App Chain and a Vault per chain to lock these tokens. These were connected using Connectors and Socket DL.

Our security assessment was a full review of these new contracts, spanning a total of 3 days.

During our audit, we have identified 3 medium severity vulnerabilities. We have also identified various minor vulnerabilities and code optimisations.

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 have increased after completion of our audit.

SCOPE

The analyzed resources are located on:
https://github.com/SocketDotTech/app-chain-token/commit/10f9ad503e7af936a32e614ad08141689efe300c

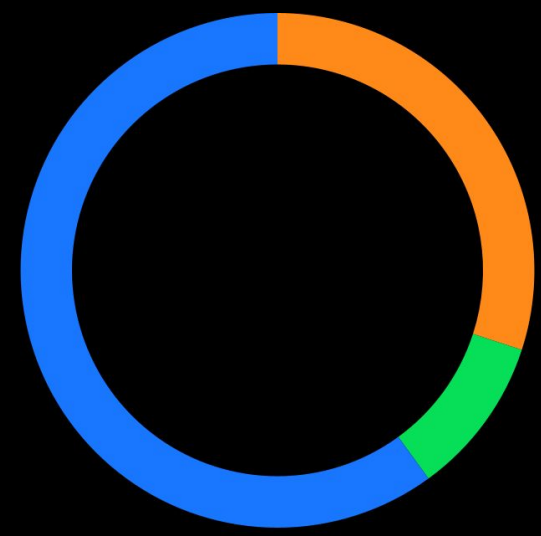
The issues described in this report were fixed. Corresponding commits are mentioned in the description.

SUMMARY

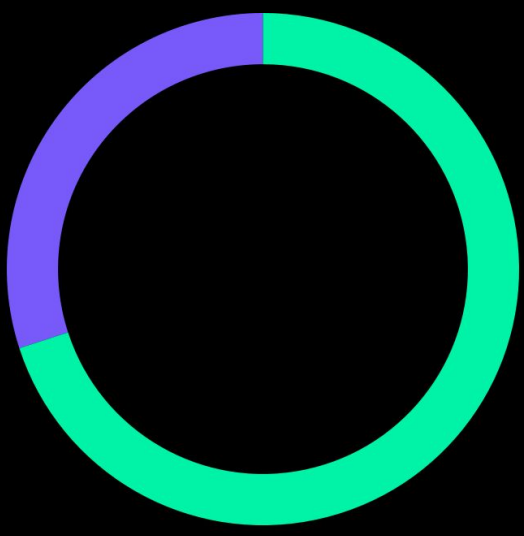
SEVERITY	NUMBER OF FINDINGS
CRITICAL	0
HIGH	0
MEDIUM	3
LOW	1
INFORMATIONAL	6

TOTAL: 10

SEVERITY



STATUS



WEAKNESSES

This section contains the list of discovered weaknesses.

SOC-7. VAULT AND CONTROLLER LACK FUNCTIONS TO ESTIMATE FEES

SEVERITY: Medium

PATH: Vault.sol:depositToAppChain:L84-103

Controller.sol:withdrawFromAppChain:L95-121

REMEDIATION: Vault.sol and Controller.sol should implement a function allowing to estimate fees for the connector used

STATUS: fixed

DESCRIPTION:

When transferring a message cross-chain from Vault on chain A to Vault on chain B, a user should pay for gas fees. As such, the functions are payable and fees should be submitted in **msg.value** and the **gasLimit parameter** when calling the function.

However, currently a user has to blindly determine the correct amount of fees to submit, which can be prone to mistakes.

The ISocket | Socket Data Layer interface has a function **getMinFees()** which would allow to determine correct fee amount and it can be checked inside of the function before sending the cross-chain message.

```
function withdrawFromAppChain(
    address receiver_,
    uint256 burnAmount_,
    uint256 gasLimit_,
    address connector_
) external payable {
    if (_burnLimitParams[connector_].maxLimit == 0)
        revert ConnectorUnavailable();

    _consumeFullLimit(burnAmount_, _burnLimitParams[connector_]); // reverts on limit hit

    totalMinted -= burnAmount_;
    token__.burn(msg.sender, burnAmount_);

    uint256 unlockAmount = exchangeRate__.getUnlockAmount(
        burnAmount_,
        connectorLockedAmounts[connector_]
    );
    connectorLockedAmounts[connector_] -= unlockAmount; // underflow revert expected

    IConnector(connector_).outbound{value: msg.value}(
        gasLimit_,
        abi.encode(receiver_, unlockAmount)
    );

    emit TokensWithdrawn(connector_, msg.sender, receiver_, burnAmount_);
}

function depositToAppChain(
    address receiver_,
    uint256 amount_,
    uint256 gasLimit_,
    address connector_
) external payable {
    if (_lockLimitParams[connector_].maxLimit == 0)
        revert ConnectorUnavailable();

    _consumeFullLimit(amount_, _lockLimitParams[connector_]); // reverts on limit hit

    token__.safeTransferFrom(msg.sender, address(this), amount_);

    IConnector(connector_).outbound{value: msg.value}(
        gasLimit_,
        abi.encode(receiver_, amount_)
    );
```

```
        emit TokensDeposited(connector_, msg.sender, receiver_, amount_);
    }
```

SOC-12. UPDATING LIMIT PARAMS DOES NOT TAKE LAST UPDATE TIMESTAMP INTO ACCOUNT

SEVERITY: Medium

PATH: Controller.sol:updateLimitParams:L74-92

Vault.sol:updateLimitParams:L64-79

REMEDIATION: the function should either call _consumePartLimit with a 0 amount first, or directly set the lastUpdateTimestamp and lastUpdateLimit to 0 for a full reset, whichever is desired

STATUS: fixed

DESCRIPTION:

The function to update the limit params in Controller and Vault directly updates the **maxLimit** and **ratePerSecond** variables in the connector's LimitParams struct.

However, this function does not take the **lastUpdateTimestamp** into account and so it will be applied to the past as well.

The struct would previously already have the **maxLimit** and **ratePerSecond** defined and if they are directly updated, the already passed period of time will be counted using the new **ratePerSecond**, which may be higher or lower.

```
function updateLimitParams(
    UpdateLimitParams[] calldata updates_
) external onlyOwner {
    for (uint256 i; i < updates_.length; i++) {
        if (updates_[i].isLock) {
            _lockLimitParams[updates_[i].connector].maxLimit = updates_[i]
                .maxLimit;
            _lockLimitParams[updates_[i].connector]
                .ratePerSecond = updates_[i].ratePerSecond;
        } else {
            _unlockLimitParams[updates_[i].connector].maxLimit = updates_[i]
                .maxLimit;
            _unlockLimitParams[updates_[i].connector]
                .ratePerSecond = updates_[i].ratePerSecond;
        }
    }
```

SOC2-5. A PLUG HAS NO ABILITY TO DISCONNECT FROM A SOCKET BY SPECIFYING ADDRESS(0) AS SWITCHBOARD

SEVERITY: Medium

PATH: SocketConfig.sol

REMEDIATION: fix the disconnect() function in DataLayer usage example

STATUS: fixed

DESCRIPTION:

There is no ability for a Plug to disconnect from a socket by calling function **Socket.sol:connect()** with **address(0)** as the **inboundSwitchboard_** and **outboundSwitchboard_** parameters.

Because of the checks inside **Socket.sol:connect()** (L155-L160) that revert on inbound/outbound switchboard with **address(0)**. Furthermore, **registerSwitchboardForSibling()** doesn't allow to register a switchboard with **address(0)**.

As a result, the **disconnect()** function in the DataLayer usage example reverts.

```
function connect(
    uint32 siblingChainSlug_,
    address siblingPlug_,
    address inboundSwitchboard_,
    address outboundSwitchboard_
) external override {
    // only capacitor checked, decapacitor assumed will exist if capacitor does
    // as they both are deployed together always
   if (
        address(capacitors__[inboundSwitchboard_][siblingChainSlug_]) ==
        address(0) ||
        address(capacitors__[outboundSwitchboard_][siblingChainSlug_]) ==
        address(0)
    ) revert InvalidConnection();
    ...
}
```

SOC-6. MALICIOUS CONNECTORPLUG DEPLOYER COULD DRAIN VAULT/CONTROLLER

SEVERITY: Low

PATH: Vault.sol:receiveInbound:L130-155

Controller.sol:receiveInbound:L149-181

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

The function **receiveInbound** for both the Vault and Controller contract will try to mint or transfer the token from the limits. In case of limits, the remaining amount is added to the pending amount for the receiver. Both functions also do not check that the Connector (**msg.sender**) is registered.

It could be the case that the deployer of a connector contract **ConnectorPlug.sol** on some chain turns malicious. He/she knows that his/her connector will be used by some **Vault.sol** or **Controller.sol** soon. Instead of deploying the original **ConnectorPlug.sol** on address **0x13...37.** he/she can deploy a malicious contract first using the Smart Contract Audits - Composable Security **CREATE/CREATE2 trick**.

Malicious contract calls **Vault.sol:receiveInbound()**, for the **Vault.sol** the attacker wants to drain. In the call arguments **unlockAmount** equals to the vault balance and the **receiver** address is under the attacker's control.

Consequently, **pendingUnlocks[msg.sender][receiver]** will be set to a controlled **unlockAmount** value.

After that the malicious deployer destroys the malicious contract and deploys normal **ConnectorPlug.sol** on address **0x13...37**.

When the target **Vault.sol** whitelists the connector by calling **Vault.sol:updateLimitParams()**. Malicious deployer drains the Vault by calling **Vault.sol:unlockPendingFor()**.

```
function receiveInbound(bytes memory payload_) external override {
    (address receiver, uint256 unlockAmount) = abi.decode(
        payload_,
        (address, uint256)
    );
    ...
    if (pendingAmount > 0) {
        // add instead of overwrite to handle case where already pending amount is left
        pendingUnlocks[msg.sender][receiver] += pendingAmount;
        connectorPendingUnlocks[msg.sender] += pendingAmount;
        emit TokensPending(
            msg.sender,
            receiver,
            pendingAmount,
            pendingUnlocks[msg.sender][receiver]
        );
    }
    ...
}
```

Vault.sol and **Controller.sol** inside **receiveInbound()** should check that a caller is a registered connector similarly to the **depositToAppChain()** and **unlockPendingFor()** functions:

```
    if (_lockLimitParams[msg.sender].maxLimit == 0)
        revert ConnectorUnavailable();
```

SOC-3. UNUSED FUNCTIONALITY OF OWNABLE2STEP

SEVERITY: Informational

PATH: ExchangeRate.sol:L19

REMEDIATION: if the Ownable2Step functionality is not needed, remove the import statement and the inheritance from Ownable2Step in the ExchangeRate contract declaration

STATUS: fixed

DESCRIPTION:

The **ExchangeRate** contract imports and implements **Ownable2Step** from the OpenZeppelin library but does not utilize any functionality or properties provided by the **Ownable2Step** contract.

Inclusion of this contract is therefore redundant and unnecessarily increases the bytecode size of the contract.

```
contract ExchangeRate is IExchangeRate, Ownable2Step {
    // chainId input needed? what else? slippage?
    function getMintAmount(
        uint256 lockAmount,
        uint256 /* totalLockedAmount */
    ) external pure returns (uint256 mintAmount) {
        return lockAmount;
    }

    function getUnlockAmount(
        uint256 burnAmount,
        uint256 /* totalLockedAmount */
    ) external pure returns (uint256 unlockAmount) {
        return burnAmount;
    }
}
```

SOC-10. STORAGE VARIABLES COULD BE MARKED AS IMMUTABLE

SEVERITY: Informational

PATH: ConnectorPlug.sol:hub__, socket__ (L21, L22)

Controller.sol:token__ (L12)

Vault.sol:token__ (L11)

REMEDIATION: change the variables to immutable in favour of gas savings on each call to these external contracts

STATUS: fixed

DESCRIPTION:

Some public storage variables are only set in the constructor and can then never be set again.

```
IHub public hub__;
ISocket public socket__;
ERC20 public token__;
```

SOC-2. UNUSED ERRORS

SEVERITY: Informational

PATH: Controller.sol:LengthMismatch:L40

Vault.sol:LengthMismatch:L33

REMEDIATION: remove the unused error

STATUS: fixed

DESCRIPTION:

In the **Controller.sol** and **Vault.sol** contracts the **error LengthMismatch** is declared on line 33 and 40 respectively, but is is never used.

```
error LengthMismatch();
```

SOC-8. CONNECTORPLUG DISCONNECT FUNCTION IS REDUNDANT

SEVERITY: Informational

PATH: ConnectorPlug.sol:disconnect:L71-77

REMEDIATION: remove ConnectorPlug.sol:disconnect() function in favour of smaller bytecode size and deployment gas costs

STATUS: acknowledged

DESCRIPTION:

The ConnectorPlug contract has functions to both connect and disconnect from a socket.

However the **ConnectorPlug.sol:disconnect()** becomes redundant because the same effect can be achieved when calling **ConnectorPlug.sol:connect()** with **address(0)** as the **switchboard_** parameter.

ConnectorPlug.sol should be deployed many times, therefore it's crucial it has the smallest size possible.

```
function disconnect(address siblingPlug_) external onlyOwner {
    socket__.connect(
        siblingChainSlug,
        siblingPlug_,
        address(0),
        address(0)
    );

    emit ConnectorPlugDisconnected(siblingPlug_);
}
```

SOC-11. STRUCTS COULD BE PACKED

SEVERITY: Informational

PATH: Gauge.sol:LimitParams:L4-9

REMEDIATION: see description

STATUS: acknowledged

DESCRIPTION:

The Gauge contract defines a struct **LimitParams** that is used in both Controller and Vault for connector specific limits. However, the struct is not optimally packed.

The field **lastUpdateTimestamp** is always set to **block.timestamp**, which could fit in a **uint32**. In that case, **ratePerSecond** could be reduced to at least **uint224** or less such that both fields would fit into one slot. Both variables are retrieved on a call to **_getCurrentLimit** , so that would save one **SLOAD** on each inbound message.

Similarly, depending on specification of maximum values, both **maxLimit** and **lastUpdateLimit** could be stored as **uint128** so they would fit into one slot and save another **SLOAD** on each inbound message.

```
struct LimitParams {
    uint256 lastUpdateTimestamp;
    uint256 ratePerSecond;
    uint256 maxLimit;
    uint256 lastUpdateLimit;
}
```

SOC-13. CONNECTORS COULD BE FORCEFULLY CONGESTED DUE TO SHARED LIMITS

SEVERITY: Informational

PATH: Controller.sol

Vault.sol

REMEDIATION: set up monitoring for these types of congestion attacks and set the Connector limits accordingly

STATUS: acknowledged

DESCRIPTION:

Both the Controller and Vault employ a limit to minting and unlocking transferred funds through a Connector.

These limits are global and thus shared by all users. As such, a malicious user could constantly keep sending the same tokens back and forth using the Connectors and forcefully congest the limit queue.

This would lower and completely block the bridge for other users as they would constantly get rate limited.

```
function _consumeFullLimit(
    uint256 amount_,
    LimitParams storage _params
) internal {
    uint256 currentLimit = _getCurrentLimit(_params);
    if (currentLimit >= amount_) {
        _params.lastUpdateTimestamp = block.timestamp;
        _params.lastUpdateLimit = currentLimit - amount_;
    } else {
        revert AmountOutsideLimit();
    }
}
```