hexens × SOCKET

MAR.24

SECURITY REVIEW REPORT FOR SOCKET

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.

EXECUTIVE SUMMARY

OVERVIEW

This audit covered Socket's new Supermodular smart contracts which makes use of the Socket plugs to facilitate bridging of tokens between blockchains.

This report of this audit only covers the findings for components of the Supermodular smart contracts that are not part of YieldHook. The findings for the YieldHook component are covered in a separate report.

Our security assessment was a full review of the new smart contracts, spanning a total of 2 weeks.

During our audit we have identified several major severity vulnerabilities, minor severity 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/socket-plugs/tree/3c289f3e2278f1f9439652bb4cac69c17f909a1f

The issues described in this report were fixed in the following commit:

https://github.com/SocketDotTech/socket-plugs/tree/60ba635c7e4999f7884cc3d21e465a94d5005ad7

AUDITING DETAILS

STARTED
15.03.2024

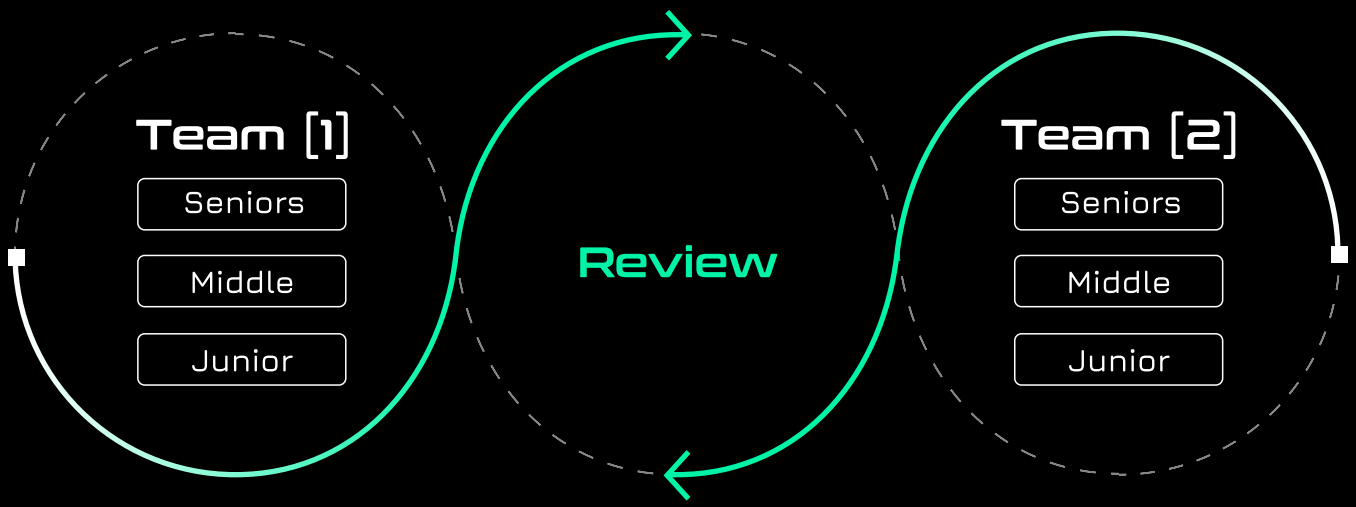
DELIVERED
22.03.2024

Review Led by

KASPER ZWIJSEN
Head of Smart Contract Audits | 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.

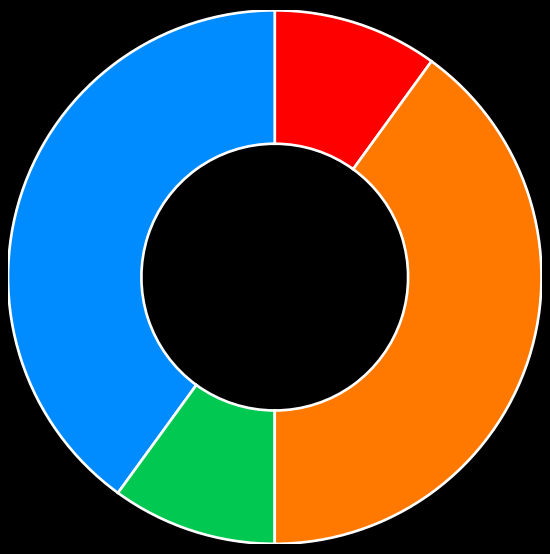
ISSUE SYMBOLIC CODES

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.

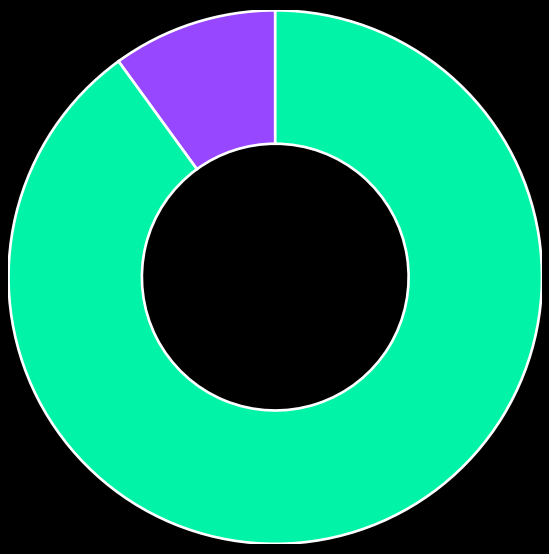
FINDINGS SUMMARY

Severity	Number of Findings
Critical	0
High	1
Medium	4
Low	1
Informational	4

Total: 10



- High
- Medium
- Low
- Informational



- Fixed
- Acknowledged

WEAKNESSES

This section contains the list of discovered weaknesses.

SCKMA-1

INCORRECT PERMIT LOGIC IN YIELDTOKENBASE

SEVERITY: High

PATH:

YieldTokenBase.sol:permit:L161-207

REMEDIATION:

Consider changing this line of code in the permit() function:

allowance[recoveredAddress][spender] = convertToShares(value);

STATUS: Fixed

DESCRIPTION:

The contract **YieldToken** has a **permit** function, allowing token approvals to be made via signatures. The signature incorporates a token amount equivalent to **convertToShares(value)**, meaning that **value** is supposed to be in underlying assets, yet it effectively provides approval for the specified **value**, and **allowance** is expressed in shares.

This means that the effective approval value will always be higher than the user meant and signed for. It may lead to user fund loss, when the user plans to approve X amount, but will approve more.

```
    function permit(
        address owner,
        address spender,
        uint256 value,
        uint256 deadline,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) public virtual {
        require(deadline >= block.timestamp, "PERMIT_DEADLINE_EXPIRED");
        // Unchecked because the only math done is incrementing
        // the owner's nonce which cannot realistically overflow.
        unchecked {
            address recoveredAddress = ecrecover(
                keccak256(
                    abi.encodePacked(
                        "\x19\x01",
                        DOMAIN_SEPARATOR(),
                        keccak256(
                            abi.encode(
                                keccak256(
                                    "Permit(address owner,address spender,uint256 value,uint256 nonce,uint256 deadline)"
                                ),
                                owner,
                                spender,
                                convertToShares(value),
                                nonces[owner]++,
                                deadline
                            )
                        )
                    )
                ),
                v,
                r,
                s
            );
```

```
            require(
                recoveredAddress != address(0) && recoveredAddress == owner,
                "INVALID_SIGNER"
            );

            allowance[recoveredAddress][spender] = value;
        }

        emit Approval(owner, spender, value);
    }
```

INCONSISTENT ASSET CONVERSION LOGIC IN YIELDTOKEN WITHDRAWAL AND REDEMPTION FUNCTIONS

SEVERITY: Medium

PATH:

YieldToken.sol:maxWithdraw, maxRedeem:L68-70, L72-74

REMEDIATION:

See description

STATUS: Fixed

DESCRIPTION:

In the **maxWithdraw** function within the **YieldToken** contract, where **convertToAssets** is called twice on **_balanceOf[owner]**, leading to an incorrect calculation of the maximum amount of underlying assets that can be withdrawn. Similarly, the **maxRedeem** function currently does not align with typical expectations for such a function in an investment or yield token contract, as it also incorrectly calculates the redeemable amount due to improper use of **convertToAssets**.

```
    function maxWithdraw(address owner) public view virtual returns
(uint256) {
        return convertToAssets(convertToAssets(_balanceOf[owner]));
    }

    function maxRedeem(address owner) public view virtual returns (uint256)
{
        return convertToAssets(_balanceOf[owner]);
    }
```

If the described logic is correct change the maxWithdraw and maxRedeem functions as follows:

```
    function maxWithdraw(address owner) public view virtual returns (uint256) {
// @audit-issue
        return convertToAssets(_balanceOf[owner]);
    }

    function maxRedeem(address owner) public view virtual returns (uint256) {
        return _balanceOf[owner];
    }
```

INCORRECT USE OF ROUNDING-UP WHEN CALCULATING THE SHARESTOMINT

SEVERITY: Medium

PATH:

contracts/token/yield-token/YieldToken.sol:L24-L35

REMEDIATION:

Consider using the round-down calculation to calculate the minted shares.

STATUS: Fixed

DESCRIPTION:

In the function **YieldToken::calculateMintAmount()**, the minted shares are determined using a round-up calculation. However, this rounding method introduces a vulnerability, allowing an attacker to mint shares with a minimal amount of tokens and subsequently steal tokens from other users.

For instance, consider the following scenario:

- **totalUnderlyingAssets = 100**
- **supply = 10**

If an attacker uses 1 token, they will obtain a number of shares calculated as follows:

- **calculateMintAmount() = ceil(1 * 10 / 100) = 1**

Upon burning 1 share of tokens, the attacker can acquire:

- **convertToAssets() = floor(1 * (100 + 1) / (10 + 1)) = 9**

By employing this strategy, the attacker gains a profit of 9 - 1 = 8 tokens.

```
function calculateMintAmount(
    uint256 underlyingAssets_
) external view returns (uint256) {
    // total supply -> total shares
    // total yield -> total underlying from all chains
    // yield sent from src chain includes new amount hence subtracted here
    uint256 supply = _totalSupply; // Saves an extra SLOAD if _totalSupply is
non-zero.
    return
        supply == 0
            ? underlyingAssets_
            : underlyingAssets_.mulDivUp(supply, totalUnderlyingAssets);
}
```

SCKMA-18

TOKENS CAN BECOME FROZEN IN CASE OF PERSISTENCE EXECUTION FAILURE

SEVERITY: Medium

PATH:

contracts/hub/Vault.sol:L34-L59

REMEDIATION:

We would recommend to consider implement a rescue or cancellation mechanism for stuck transfers.

STATUS: Acknowledged

DESCRIPTION:

The **Vault::bridge()** function facilitates the transfer of tokens from the current chain to another chain. Initially, users must transfer their tokens to the contract. Subsequently, the **connector.outbound()** function is triggered to relay the message to the destination chain. However, successful delivery of the message from the socket to the connector on the destination chain is not guaranteed due to factors such as **gasLimit** or fees: Message Failure & Retry | Socket Data Layer .

If the message delivery fails on the destination chain, there may be a need to resend the message from the source chain. The issue arises when the message cannot be delivered on the destination chain, resulting in users' tokens transferred to the vault on the source chain not being refunded. Consequently, users' funds become locked within the contract.

```
function bridge(
    address receiver_,
    uint256 amount_,
    uint256 msgGasLimit_,
    address connector_,
    bytes calldata execPayload_,
    bytes calldata options_
) external payable nonReentrant {
    (
        TransferInfo memory transferInfo,
        bytes memory postHookData
    ) = _beforeBridge(
            connector_,
            TransferInfo(receiver_, amount_, execPayload_)
        );


    _receiveTokens(transferInfo.amount);


    _afterBridge(
        msgGasLimit_,
        connector_,
        options_,
        postHookData,
        transferInfo
    );
}
```

SCKMA-19

LIMITHOOK RESTRICTIONS CAN BE BYPASSED

SEVERITY: Medium

PATH:

contracts/hooks/LimitHook.sol

REMEDIATION:

Incorporate the connector value into identifierCache and verify within preRetryHook() whether params_.connector matches the connector decoded from the identifierCache value.

STATUS: Fixed

DESCRIPTION:

The limit imposed by **LimitHook** could be circumvented. Imagine a scenario where a **Controller** or **Vault** has connectors for various chains (e.g., Optimism and Arbitrum) and utilizes the **LimitHook** contract to enforce daily limits.

The issue arises from the fact that **identifierCache** does not take into account the **connector** value, only incorporating the receiver and pending amount:

```
identifierCache = abi.encode(updatedReceiver, pendingAmount)
```

When a user receives tokens from Optimism and reaches the daily limit, an entry is made in the **identifierCache** mapping for the message id with the value **abi.encode(params_.transferInfo.receiver, pendingAmount)**. Consequently, the user can later retry the message id once the daily limit resets for Optimism.

Instead, the user could invoke **Controller.retry()** with a connector for a different chain (Arbitrum) where the limit has not been reached yet, specifying the message id received from Optimism.

This action will trigger **_limitDstHook()** for a different connector (Arbitrum) but with the pendingMint received from Optimism:

```
(uint256 consumedAmount, uint256 pendingAmount) = _limitDstHook(
    params_.connector,
    pendingMint
);
```

```
function preRetryHook(
    PreRetryHookCallParams memory params_
)
    external
    nonReentrant
    isVaultOrController
    returns (
        bytes memory postRetryHookData,
        TransferInfo memory transferInfo
    )
{
    (address updatedReceiver, uint256 pendingMint) = abi.decode(
        params_.cacheData.identifierCache,
        (address, uint256)
    );
    (uint256 consumedAmount, uint256 pendingAmount) = _limitDstHook(
        params_.connector,
        pendingMint
    );

    postRetryHookData = abi.encode(
        updatedReceiver,
        consumedAmount,
        pendingAmount
    );
    transferInfo = TransferInfo(updatedReceiver, consumedAmount, bytes(""));
}
```

UNUSED TIMESTAMP VARIABLE IN YIELDTOKEN

SEVERITY: Low

PATH:

YieldTokenBase.sol:lastSyncTime:L47

REMEDIATION:

Unused variables should be removed, this will reduce gas costs for storage and computation.

STATUS: Fixed

DESCRIPTION:

The **lastSyncTimestamp** variable in the **YieldToken** contract is only set but never used for any aspect of the code.

As of right now, the variable serves no purpose and should therefore not be stored on the blockchain. If the timestamp is required for front-end purposes, the timestamp could be stored off-chain or emitted in an event and tracked off-chain.

Removing the variable would reduce gas costs by one SSTORE (5.000 gas) for each call to **updateTotalUnderlyingAssets**. At an ETH price of $4000 and gas price of 80 Gwei that equals and extra ~$1.60 per call.

```
function updateTotalUnderlyingAssets(
        uint256 amount_
    ) external onlyRole(HOOK_ROLE) {
        _updateTotalUnderlyingAssets(amount_);
    }

    function _updateTotalUnderlyingAssets(uint256 amount_) internal {
        lastSyncTimestamp = block.timestamp;
        totalUnderlyingAssets = amount_;
    }
```

USE CUSTOM ERRORS

SEVERITY: Informational

PATH:

YieldTokenBase.sol:permit:L161-207

REMEDIATION:

Replace the statements with custom errors to reduce the deployment gas and make the error-handling technique more consistent.

STATUS: Fixed

DESCRIPTION:

YieldTokenBase.permit() uses **require** statements instead of custom errors.

```
    function permit(
        address owner,
        address spender,
        uint256 value,
        uint256 deadline,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) public virtual {
        require(deadline >= block.timestamp, "PERMIT_DEADLINE_EXPIRED");
        // Unchecked because the only math done is incrementing
        // the owner's nonce which cannot realistically overflow.
        unchecked {
            address recoveredAddress = ecrecover(
                keccak256(
                    abi.encodePacked(
                        "\x19\x01",
                        DOMAIN_SEPARATOR(),
                        keccak256(
                            abi.encode(
                                keccak256(
                                    "Permit(address owner,address spender,uint256
value,uint256 nonce,uint256 deadline)"
                                ),
                                owner,
                                spender,
                                convertToShares(value),
                                nonces[owner]++,
                                deadline
                            )
                        )
                    )
                ),
                v,
                r,
                s
            );
```

```
        require(
            recoveredAddress != address(0) && recoveredAddress == owner,
            "INVALID_SIGNER"
        );

        allowance[recoveredAddress][spender] = value;
    }

    emit Approval(owner, spender, value);
}
```

SCKMA-10

BAD UX NATIVE VALUE CHECK DURING VAULT BRIDGE

SEVERITY: Informational

PATH:

Base.sol:_afterBridge:L117-162

REMEDIATION:

See description

STATUS: Fixed

DESCRIPTION:

The function **Base._afterBridge()** uses an implicit approach to check whether the ETH amount received is bigger than the amount bridged: **msg.value - transferInfo.amount** during the fee calculation will revert with an underflow error otherwise. It is recommended that an explicit check with a custom error be added to improve the UX.

```
function _afterBridge(
    uint256 msgGasLimit_,
    address connector_,
    bytes memory options_,
    bytes memory postSrcHookData_,
    TransferInfo memory transferInfo_
) internal {
    TransferInfo memory transferInfo = transferInfo_;

    ...

    uint256 fees = address(token) == ETH_ADDRESS
        ? msg.value - transferInfo.amount
        : msg.value;

    ...
}
```

contracts/hub/Base.sol

```
    function _afterBridge(
        uint256 msgGasLimit_,
        address connector_,
        bytes memory options_,
        bytes memory postSrcHookData_,
        TransferInfo memory transferInfo_
    ) internal {
        TransferInfo memory transferInfo = transferInfo_;

+       if (msg.value < transferInfo.amount) revert NotEnoughMsgValue();

        if (address(hook__) != address(0)) {
            transferInfo = hook__.srcPostHookCall(
                SrcPostHookCallParams(
                    connector_,
                    options_,
                    postSrcHookData_,
                    transferInfo_
                )
            );
        }

        uint256 fees = address(token) == ETH_ADDRESS
            ? msg.value - transferInfo.amount
            : msg.value;
```

contracts/common/Errors.sol

```
  error NotEnoughMsgValue();
```

THE ALLOWANCE FOR THE OLD HOOK SHOULD BE REVOKED WHEN THE NEW HOOK IS SET

SEVERITY: Informational

PATH:

contracts/hub/Base.sol:L55-L63

REMEDIATION:

Consider revoking the allowance associated with the old hook in the updateHook function.

STATUS: Fixed

DESCRIPTION:

The function **Base::updateHook()** is utilized to update the new hook address for the hub. This function grants the new hook_ address an unlimited allowance of tokens. However, it does not revoke the allowance of the contract with the old hook yet. This failure to revoke can lead to a scenario where the old hook, if controlled by malicious actors or hacked, can still steal assets from hub contracts through the allowance, even if the new hook is set for the hub.

```
function updateHook(
    address hook_,
    bool approve_
) external virtual onlyOwner {
    hook__ = IHook(hook_);
    if (approve_)
        SafeTransferLib.safeApprove(ERC20(token), hook_, type(uint256).max);
    emit HookUpdated(hook_);
}
```

REDUNDANT USE OF MODIFIER NOTSHUTDOWN IN POST-HOOK CALL

SEVERITY: Informational

REMEDIATION:

Remove the notShutDown modifier in the post-hook call function.

STATUS: Fixed

DESCRIPTION:

In each pre-hook call function, the **notShutDown** modifier is used to check the shutdown status of the hub. However, the post-hook call function also contains the same modifier, making it redundant for these functions, assuming that the flow will always be that these are called in pairs from the Vault/Controller.

The only exception is **srcPreHookCall** in **Controller_YieldLimitExecHook** because it doesn't have the **notShutdown** modifier (**srcPostHookCall** does).

```
function srcPreHookCall(
    SrcPreHookCallParams calldata params_
) public override notShutdown returns (TransferInfo memory, bytes memory) {
```

```
  function srcPostHookCall(
      SrcPostHookCallParams memory srcPostHookCallParams_
  )
      public
      override
      notShutdown
      isVaultOrController
      returns (TransferInfo memory transferInfo)
  {
```


hexens
×
SOCKET