Aug.23

SECURITY REVIEW REPORT FOR SOCKET

CONFIDENTIAL

CONFIDENTIAL

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.

CONFIDENTIAL

AUDIT LED BY

VAHE KARAPETYAN

Co-founder / CTO | Hexens

Audit Starting Date
07.08.2023

Audit Completion Date
16.08.2023

hexens × SOCKET

CONFIDENTIAL

LIMITATIONS ON DISCLOSURE AND USAGE OF THIS REPORT

METHODOLOGY

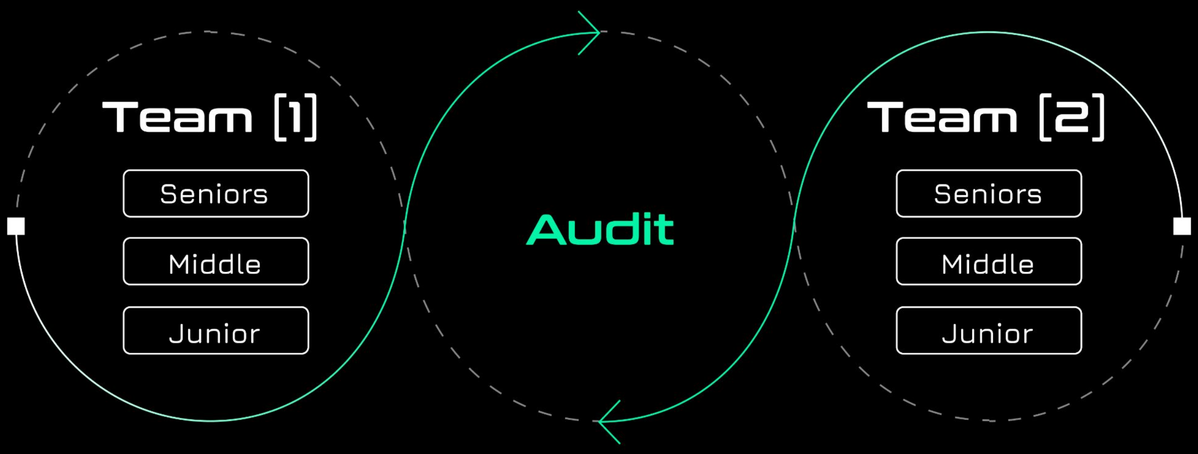
COMMON AUDIT PROCESS

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 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.

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.

SCOPE

The analyzed resources are located on:
https://github.com/SocketDotTech/socket-DL/tree/master/contracts/socket/Socket.sol
https://github.com/SocketDotTech/socket-DL/tree/master/contracts/socket/SocketBase.sol
https://github.com/SocketDotTech/socket-DL/tree/master/contracts/socket/SocketConfig.sol
https://github.com/SocketDotTech/socket-DL/tree/master/contracts/socket/SocketDst.sol
https://github.com/SocketDotTech/socket-DL/tree/master/contracts/socket/SocketSrc.sol

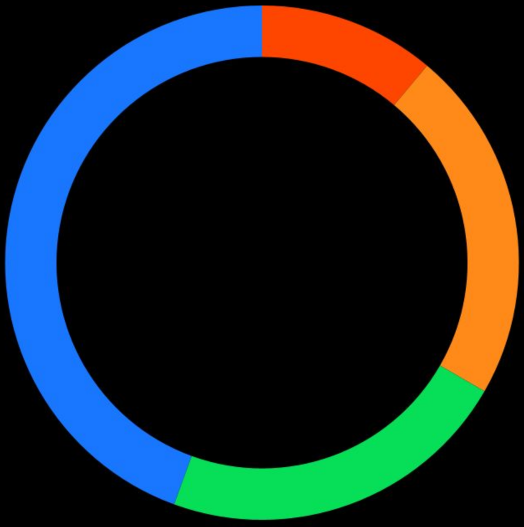
The issues described in this report were fixed. Corresponding commits are mentioned in the description.

SUMMARY

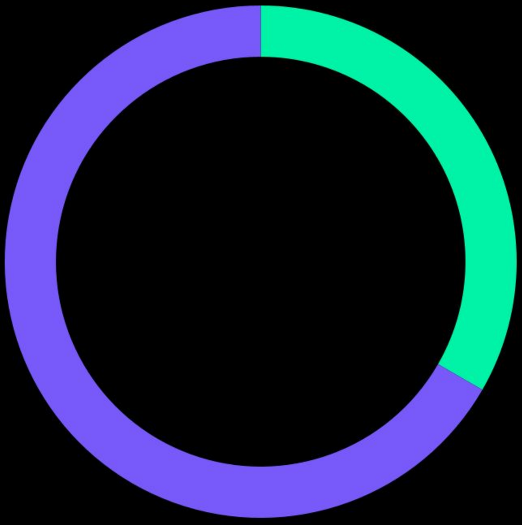
SEVERITY	NUMBER OF FINDINGS
CRITICAL	0
HIGH	1
MEDIUM	2
LOW	2
INFORMATIONAL	4

TOTAL: 9

SEVERITY



STATUS



WEAKNESSES

This section contains the list of discovered weaknesses.

SOC2-18. CENTRALIZATION ISSUE IN GOVERNANCE ROLE MANAGEMENT

SEVERITY: High

PATH: SocketBase.sol

REMEDIATION: override/rewrite the revokeRole and grantRole implemenations to set GOVERNANCE_ROLE only once (grantRole) and revert in case its being revoked (revokeRole)

STATUS: acknowledged, see commentary

DESCRIPTION:

SocketBase contract uses AccessControlExtended as a base class to manage authorization roles. Some of the crucial SocketBase functions, such as setCapacitorFactory, setHasher, setExecutionManager, and setTransmitManager are designed to be called only by GOVERNANCE_ROLE holder, which is supposed to be the governance contract address.

Although the AccessControlExtended and AccessControl contracts' main functionality, and specifically the revokeRole and grantRole functions use onlyOwner modifier and the contract deployer is automatically set as the owner in the constructor.

Thus whenever the GOVERNANCE_ROLE will be given to any address, the owner still has full access to that role management and can add other GOVERNANCE_ROLE holders or revoke the existing ones; this brings to a situation where governance management is, in fact, fictitious, whereas the owner still holds all the power over roles and authorization roles.

```
/**
 * @dev Set the capacitor factory contract
 * @dev Only governance can call this function
 * @param capacitorFactory_ The address of the capacitor factory contract
 */
function setCapacitorFactory(
    address capacitorFactory_
) external onlyRole(GOVERNANCE_ROLE) {
    capacitorFactory__ = ICapacitorFactory(capacitorFactory_);
    emit CapacitorFactorySet(capacitorFactory_);
}

/**
 * @notice updates hasher__
 * @dev Only governance can call this function
 * @param hasher_ address of hasher
 */
function setHasher(address hasher_) external onlyRole(GOVERNANCE_ROLE) {
    hasher__ = IHasher(hasher_);
    emit HasherSet(hasher_);
}

/**
 * @notice updates executionManager__
 * @dev Only governance can call this function
 * @param executionManager_ address of Execution Manager
 */
function setExecutionManager(
    address executionManager_
) external onlyRole(GOVERNANCE_ROLE) {
    executionManager__ = IExecutionManager(executionManager_);
    emit ExecutionManagerSet(executionManager_);
}

/**
 * @notice updates transmitManager__
 * @param transmitManager_ address of Transmit Manager
 * @dev Only governance can call this function
 * @dev This function sets the transmitManager address. If it is ever upgraded,
 * remove the fees from executionManager first, and then upgrade address at socket.
 */
function setTransmitManager(
    address transmitManager_
) external onlyRole(GOVERNANCE_ROLE) {
    transmitManager__ = ITransmitManager(transmitManager_);
    emit TransmitManagerSet(transmitManager_);
}
```

Commentary from the client:

“ - The contract owner is also expected to be handed over to governance after system is stable and tested. We acknowledge it but this is the intended design. We will add a function to explicitly handover the owner role to governance.”

SOC2-13. THE OUTBOUND() FUNCTION LACKS AN INPUT PARAMETER TO LIMIT THE MAXIMUM AMOUNT OF FEES PAID

SEVERITY: Medium

PATH: SocketSrc.sol

REMEDIATION: as an additional safeguard introduce an input parameter maxUserLimitForFees_ for the outbound() function to limit the maximum amount of fees a user is willing to pay

if (msg.value > maxUserLimitForFees_) revert ExceededUserLimitForFees();

STATUS: acknowledged, see commentary

DESCRIPTION:

There is currently no protection for users against components taking too much of the provided ETH as fees. This is similar to no slippage protection in swap protocols.

Function **outbound()** lacks an input parameter that limits the maximum amount of fees for message transfers the user is willing to pay. Currently, the user transfers the required amount of fees inside **msg.value**, including switchboard fees, execution fees, and transmission fees.

Switchboards | Socket Data Layer is deemed a less-trusted component in the protocol, any developer can craft and propose a new one. Switchboard fees are controlled solely by a switchboard (L237-L248 of **SocketSrc.sol**).

In a normal workflow, the user through the plug should call **getMinFees()** to get the minimum amount of fees (the sum of switchboard/execution/transmission fees) before calling the **outbound()** function. There might be a situation when a misbehaving switchboard requested too many fees during the call to the **getMinFees()** function. If a user is a kind-of automated agent that trusts the return value from the call, it might result in a loss.

An additional parameter could give such a user additional control over fees paid and misbehaving switchboard. This ultimately protects the users by specifying how much they are willing to pay in fees.

Commentary from the client:

“ - This seems like a user input sanitisation issue on plugs and not a socket protocol issue. Even if we take max amount as input, it doesn’t mitigate the issue since this param is also an input that would need to be sanitised by plugs.”

SOC2-20. POTENTIAL DATA LOSS IN PLUG RECONNECTION

SEVERITY: Medium

PATH: SocketConfig.sol

REMEDIATION: To handle this, consider the following steps:

1. Add a validation check to see if a connection already exists for the given plug and siblingChainSlug_. If it does, either:
   - Prevent the reconnection to preserve data, or
   - Provide a migration mechanism to move the data from the old capacitor to the new one.
2. If data preservation isn't a priority, at least provide explicit warnings in function comments and documentation so users are aware of the potential data loss when reconnecting.
3. Consider introducing a separate function for updating connections to make the behavior explicit, segregating the logic of initial connection and reconnection.

STATUS: acknowledged, see commentary

DESCRIPTION:

The **connect** function allows plugs to establish a connection with the socket, setting up configurations specific to a **siblingChainSlug_** and **siblingPlug_**. If a plug decides to reconnect with a new inbound/outbound switchboard address for the same **siblingChainSlug_** and **siblingPlug_**, the existing capacitor/decapacitor addresses linked with the plug would be overwritten. This would mean that any state stored in the old capacitor/decapacitor is lost.

The function, as currently structured, does not account for a scenario where a plug needs to update an existing connection without data loss. This poses a risk in scenarios where updates are required, as plugs might lose access to crucial data in their previous capacitor/decapacitor.

```
/**
 * @notice connects Plug to Socket and sets the config for given `siblingChainSlug_`
 * @notice msg.sender is stored as plug address against given configuration
 * @param siblingChainSlug_ the sibling chain slug
 * @param siblingPlug_ address of plug present at siblingChainSlug_ to call at inbound
 * @param inboundSwitchboard_ the address of switchboard to use for verifying messages at inbound
 * @param outboundSwitchboard_ the address of switchboard to use for sending messages
 */
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();

    PlugConfig storage _plugConfig = _plugConfigs[msg.sender][
        siblingChainSlug_
    ];

    _plugConfig.siblingPlug = siblingPlug_;
    _plugConfig.capacitor__ = capacitors__[outboundSwitchboard_][
        siblingChainSlug_
    ];
    _plugConfig.decapacitor__ = decapacitors__[inboundSwitchboard_][
        siblingChainSlug_
    ];
    _plugConfig.inboundSwitchboard__ = ISwitchboard(inboundSwitchboard_);
    _plugConfig.outboundSwitchboard__ = ISwitchboard(outboundSwitchboard_);
```

```
        emit PlugConnected(
            msg.sender,
            siblingChainSlug_,
            siblingPlug_,
            inboundSwitchboard_,
            outboundSwitchboard_,
            address(_plugConfig.capacitor__),
            address(_plugConfig.decapacitor__)
        );
    }
```

Commentary from the client:

“ - To do this migration, contract admin should first update the outbound switchboard to redirect all the new messages from this path. To not block already sent messages, it is recommended that admin manually or programmatically verifies that all the messages are executed and then proceed to switch inbound switchboard. Note that, this will cause delay for messages being sent in this duration. To avoid this, plugs can pause the execution for sometime.”

SOC2-3. SIGNATURE REPLAY IN PROPOSEFORSWITCHBOARD

SEVERITY: Low

PATH: SocketDst.sol

REMEDIATION: use nonce protection for transmitters' signatures

STATUS: acknowledged, see commentary

DESCRIPTION:

Function **proposeForSwitchboard()** of **SocketDst.sol** allows anyone to re-use the valid signature of a transmitter to propose a packet indefinite number of times. As a result it's possible to increase **proposalCount[packetId_]** to any value (L129 of **SocketDst.sol**).

Currently, the **SwitchboardBase** contract allows watchers to block a certain proposal by calling the **tripProposal()** with **packetId** and **proposalCount**. **FastSwitchBoard** and **OptimisticSwitchboard** have a check inside **allowPacket()** to disallow a tripped proposal.

```
function allowPacket(
    bytes32 root_,
    bytes32 packetId_,
    uint256 proposalCount_,
    uint32 srcChainSlug_,
    uint256 proposeTime_
) external view override returns (bool) {
    ...

    // any relevant trips triggered or invalid packet count.
    if (
        isGlobalTipped ||
        isPathTripped[srcChainSlug_] ||
        isProposalTripped[packetId_][proposalCount_] ||
        packetCount < initialPacketCount[srcChainSlug_]
    ) return false;
    ...
}
```

However, signature replay allows to circumvent this mechanism easily, anyone can call proposeForSwitchboard() with an old and valid transmitter's signature thus untripping the proposal.

Furthermore, this might also lead to a grief attack for a transmitter. In such a scenario a malicious plug on chain A can send a message with extremely high value of **minMsgGasLimit_**, resulting in a revert during packet execution (L166 of **SocketDst.sol:execute()**).

Innocuous transmitter proposes a packet by calling **proposeForSwitchboard()** with a valid signature. Watchers might want to trip the proposal with a message that's impossible to process. But because of the signature replay issue, a bad actor can propose the packet in a loop with the signature from the innocuous transmitter. As a result, watchers might wrongly decide to kick out the transmitter.

```
function proposeForSwitchboard(
    bytes32 packetId_,
    bytes32 root_,
    address switchboard_,
    bytes calldata signature_
) external payable override {
    if (packetId_ == bytes32(0)) revert InvalidPacketId();

    (address transmitter, bool isTransmitter) = transmitManager__
        .checkTransmitter(
            uint32(_decodeChainSlug(packetId_)),
            keccak256(abi.encode(version, chainSlug, packetId_, root_)),
            signature_
        );

    if (!isTransmitter) revert InvalidTransmitter();

    packetIdRoots[packetId_][proposalCount[packetId_]][
        switchboard_
    ] = root_;
    rootProposedAt[packetId_][proposalCount[packetId_]][
        switchboard_
    ] = block.timestamp;

    emit PacketProposed(
        transmitter,
        packetId_,
        proposalCount[packetId_]++,
        root_,
        switchboard_
    );
}
```

Commentary from the client:

“ - In case of bad messages that cannot be executed, the root proposed would still be valid since it matches the root sealed on source. Even if others reuse the same sig to propose multiple times, they would all match the root sealed on source and still be valid. Transmitter would not be kicked out due to this activity. It just costs fee to the party doing duplicate proposes. Again, this does not enable fraud or kicking of any form.

By design, proposeForSwitchboard needs to have the same digest and signature as seal function on SocketSrc. Due to this we cannot include a nonce field. This makes sure anyone can propose a valid root that was seal on source.

Furthermore, transmitters are not expected to broadcast wrong signatures, so the probability of this is rare. In case they do it, protocol is supposed to kick them out, making the invalid sig unusable. We think this mitigates the issue. This should be marked as a low severity issue since it doesnt enable fraud, just allows for rare watcher griefing as they need to trip all the invalid proposals till a transmitter is kicked.”

SOC2-6. MINMSGGASLIMIT PARAMETER ISN'T CHECKED INSIDE OUTBOUND FUNCTION

SEVERITY: Low

PATH: SocketSrc.sol

REMEDIATION: check inside the outbound() function that the minMsgGasLimitparameter lies in a sensible range and it's not possible to specify an arbitrary large value

STATUS: acknowledged, see commentary

DESCRIPTION:

There is no check for the **minMsgGasLimit** parameter value inside the **outbound()** function. If the value is too high or it's not consistent with the value of **executionDetails_.executionGasLimit** parameter passed during message execution to the function **SocketDst.sol:execute()**, a revert will happen (L166 of **SocketDst.sol:execute()**).

Introducing guardrails and first checking the **minMsgGasLimit** parameter on the sender's side is more effective and requires less burden.

Inside the **outbound()** function **minMsgGasLimit_** is passed to the **_validateAndSendFees()** function (L108) and further to the **executionManager__ .payAndCheckFees()** (L181).

Next, inside **ExecutionManager.sol** the value of **minMsgGasLimit_** is passed to the **_getMinFees()** (L237) which ignores that parameter.

```
function _getMinFees(
    uint256,
    uint256 payloadSize_,
    bytes32 executionParams_,
    uint32 siblingChainSlug_
) internal view returns (uint128) {
...
}
```

```
function outbound(
    uint32 siblingChainSlug_,
    uint256 minMsgGasLimit_,
    bytes32 executionParams_,
    bytes32 transmissionParams_,
    bytes calldata payload_
) external payable override returns (bytes32 msgId) {
    PlugConfig memory plugConfig;

    // looks up the sibling plug address using the msg.sender as the local plug address
    plugConfig.siblingPlug = _plugConfigs[msg.sender][siblingChainSlug_]
        .siblingPlug;

    // if no sibling plug is found for the given chain slug, revert
    if (plugConfig.siblingPlug == address(0)) revert PlugDisconnected();

    // fetches auxillary details for the message from the plug config
    plugConfig.capacitor__ = _plugConfigs[msg.sender][siblingChainSlug_]
        .capacitor__;
    plugConfig.outboundSwitchboard__ = _plugConfigs[msg.sender][
        siblingChainSlug_
    ].outboundSwitchboard__;

    // creates a unique ID for the message
    msgId = _encodeMsgId(plugConfig.siblingPlug);

    // validate if caller has send enough fees, if yes, send fees to execution manager
    // for parties to claim later
    ISocket.Fees memory fees = _validateAndSendFees(
        minMsgGasLimit_,
        uint256(payload_.length),
        executionParams_,
        transmissionParams_,
        plugConfig.outboundSwitchboard__,
        plugConfig.capacitor__.getMaxPacketLength(),
        siblingChainSlug_
    );
```

```
        ISocket.MessageDetails memory messageDetails = ISocket.MessageDetails({
            msgId: msgId,
            minMsgGasLimit: minMsgGasLimit_,
            executionParams: executionParams_,
            payload: payload_,
            executionFee: fees.executionFee
        });

        // create a compressed data-struct called PackedMessage
        // which has the message payload and some configuration details
        bytes32 packedMessage = hasher__.packMessage(
            chainSlug,
            msg.sender,
            siblingChainSlug_,
            plugConfig.siblingPlug,
            messageDetails
        );

        // finally add packedMessage to the capacitor to generate new root
        plugConfig.capacitor__.addPackedMessage(packedMessage);

        emit MessageOutbound(
            chainSlug,
            msg.sender,
            siblingChainSlug_,
            plugConfig.siblingPlug,
            msgId,
            minMsgGasLimit_,
            executionParams_,
            transmissionParams_,
            payload_,
            fees
        );
    }
```

Commentary from the client:

“ - We believe its sender's responsibility to estimate the gas properly and this issue only affects the messages that have provided wrong gas limit hence adding one more check is not a good trade off considering gas.

Moreover, EM is updateable component hence we will monitor the cases and update EM if needed.”

SOC2-8. PARAMETER MESSAGEDETAILS_.EXECUTIONPARAMS SHOULD BE CHECKED EARLIER

SEVERITY: Informational

PATH: SocketDst.sol

REMEDIATION: check the messageDetails_.executionParams parameter at the beginning of execute() to save the executor’s gas

STATUS: fixed

DESCRIPTION:

Input parameter **messageDetails_.executionParams** is checked almost at the end of the execute() function (L268 of **SocketDst.sol**).

If this parameter is set incorrectly **execute()** function reverts (L366 of **ExecutionManager.sol**) wasting the executor’s gas. Therefore it's important to check the **messageDetails_.executionParams** parameter at the very beginning of **execute()**.

```
function execute(
    ISocket.ExecutionDetails calldata executionDetails_,
    ISocket.MessageDetails calldata messageDetails_
) external payable override {
    // make sure message is not executed already
    if (messageExecuted[messageDetails_.msgId])
        revert MessageAlreadyExecuted();
    ...
    _verify(
        executionDetails_.packetId,
        executionDetails_.proposalCount,
        remoteSlug,
        packedMessage,
        packetRoot,
        plugConfig,
        executionDetails_.decapacitorProof,
        messageDetails_.executionParams
    );

    // execute message
    _execute(
        executor,
        localPlug,
        remoteSlug,
        executionDetails_.executionGasLimit,
        messageDetails_
    );
}
```

```
function _verify(
    bytes32 packetId_,
    uint256 proposalCount_,
    uint32 remoteChainSlug_,
    bytes32 packedMessage_,
    bytes32 packetRoot_,
    PlugConfig memory plugConfig_,
    bytes memory decapacitorProof_,
    bytes32 executionParams_
) internal {
    // NOTE: is the the first un-trusted call in the system, another one is Plug.inbound
    if (
        !ISwitchboard(plugConfig_.inboundSwitchboard__).allowPacket(
            packetRoot_,
            packetId_,
            proposalCount_,
            uint32(remoteChainSlug_),
            rootProposedAt[packetId_][proposalCount_][
                address(plugConfig_.inboundSwitchboard__)
            ]
        )
    ) revert VerificationFailed();

    if (
        !plugConfig_.decapacitor__.verifyMessageInclusion(
            packetRoot_,
            packedMessage_,
            decapacitorProof_
        )
    ) revert InvalidProof();

    // finally make sure executor params were respected by the executor
    executionManager__.verifyParams(executionParams_, msg.value);
}
```

SOC2-2. REDUNDANT PAYABLE MODIFIER

SEVERITY: Informational

PATH: SocketSrc.sol:seal:L301-324

REMEDIATION: remove the payable modifier

STATUS: acknowledged, see commentary

DESCRIPTION:

The function **seal** doesn't handle **msg.value** nor is it ever called with a value. The modifier is therefore redundant.

```
function seal(
    uint256 batchSize_,
    address capacitorAddress_,
    bytes calldata signature_
) external payable override {
    [..]
}
```

Commentary from the client:

“ - Added this modifier to optimise gas”

SOC2-7. REDUNDANT CASTING TO UINT32

SEVERITY: Informational

PATH: SocketDst.sol

REMEDIATION: see description

STATUS: fixed

DESCRIPTION:

Parameter **remoteChainSlug_** of type **uint32** is redundantly casted to uint32 inside the internal function **_verify()** (L252).

The result of **_decodeChainSlug(packetId_)** function is redundantly casted to **uint32** inside the **proposeForSwitchboard()** function (L122).

```
function _verify(
    bytes32 packetId_,
    uint256 proposalCount_,
    uint32 remoteChainSlug_,
    bytes32 packedMessage_,
    bytes32 packetRoot_,
    PlugConfig memory plugConfig_,
    bytes memory decapacitorProof_,
    bytes32 executionParams_
) internal {
    // NOTE: is the the first un-trusted call in the system, another one is Plug.inbound
    if (
        !ISwitchboard(plugConfig_.inboundSwitchboard__).allowPacket(
            packetRoot_,
            packetId_,
            proposalCount_,
            uint32(remoteChainSlug_),
            rootProposedAt[packetId_][proposalCount_][
                address(plugConfig_.inboundSwitchboard__)
            ]
        )
    ) revert VerificationFailed();
    ...
}
```

```
function proposeForSwitchboard(
    bytes32 packetId_,
    bytes32 root_,
    address switchboard_,
    bytes calldata signature_
) external payable override {
    ...

    (address transmitter, bool isTransmitter) = transmitManager__
        .checkTransmitter(
            uint32(_decodeChainSlug(packetId_)),
            keccak256(abi.encode(version, chainSlug, packetId_, root_)),
            signature_
        );
    ...
}
```

SOC2-16. DEFAULT VALUE INITIALISATION

SEVERITY: Informational

PATH: see description

REMEDIATION: We would recommend to remove the initialisation to a default value in favour of saving gas.

For example:

uint64 public globalMessageCount;

STATUS: fixed

DESCRIPTION:

We found that at the following location in the code there is a variable initialised to its default value. This is already the default behaviour of Solidity and it becomes therefore redundant and a waste of gas, especially for storage variables.

1. **SocketBase.sol:globalMessageCount** (L22)

```
uint64 public globalMessageCount = 0;
```

hexens