Zellic

Socket Data Layer

Smart Contract Security Assessment

July 27, 2023

Prepared for:

Arth Patel

Socket Technology

Prepared by:

Syed Faraz Abrar and Aaron Esau

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for Socket Technology from July 12th to July 27th, 2023. During this engagement, Zellic reviewed Socket Data Layer's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Is it possible to replay messages using the same signature?
- Can a signature be replayed for the incorrect slug?
- Can a transmitter sign a `msgId` to trigger the replay protection to censor a message?
- Are denial-of-service attacks that permanently block execution of messages possible to execute without any given permission?
- Can an on-chain attacker drain the fees stored in contracts?
- Can a malicious transaction trigger a lockup of fees?
- Can a malicious plug bypass fee paying?

1.2 Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody or upgradable contract attacks
- Governance implementations
- Those specifically requested to not be assessed by Socket Technology, including the following:
  - Missing `address(0)` checks on setup and deployment
  - Unchecked receiver address while withdrawing fees or rescuing funds
  - Open nature of switchboard registration
  - Possibility of plugs not being connected properly
  - Message execution order not being enforced — this is expected to be

enforced in apps if desired

- Apps not handling failures — collected fees are not paid out and no state is updated when message execution reverts
- Possibility of desynchronized chain slug IDs — it is assumed that chain slugs are consistent across chains
- Unknown chain slugs — they are not supported without verification
- Native switchboards allowing the same packet to be relayed multiple times — the only downside is the cost to the initiator
- Watchers' ability to trip path and stop execution permanently on fast or optimistic switchboards
- Unused variable warnings, variable naming, and other nonsecurity code-quality–related issues

Note that due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

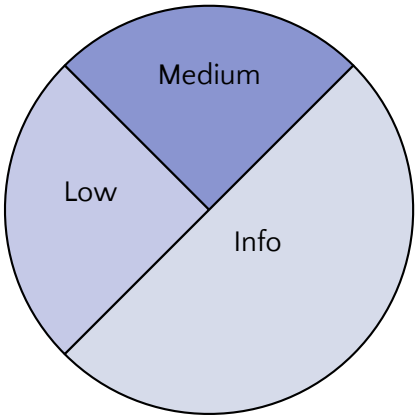
1.3 Results

During our assessment on the scoped Socket Data Layer contracts, we discovered eight findings. No critical issues were found. Two were of medium impact, two were of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Socket Technology's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	2
Low	2
Informational	4



2 Introduction

2.1 About Socket Data Layer

Socket Data Layer is a protocol for generic message passing between chains. It has been designed to be highly configurable so that dApps (plugs) can choose the best trade-offs for specific use cases.

2.2 Glossary of Terms

The following glossary is published on Socket Technology's documentation website here. Throughout this report, we will reference these terms but may not necessarily define them where we use them.

1. **Socket**: Socket is the core contract deployed on all supported networks. All core modules and functions exist within Socket.

2. **Plug/Plugs**: Plugs are smart contract applications that connect with Socket to send and receive cross-chain messages via the IPlug interface. Plugs act as adapter contracts that connect your main Smart contract to the messaging infrastructure.

3. **Message**: Message is the payload you want to transmit, along with relevant metadata like destination ChainSlug, etc.

4. **Packet**: Packets are collections of messages sent from one chain to another. The validity of messages in a packet is verified on the destination chain based on the logic prescribed in the configured switchboard. For more details, refer to the documentation on Packets and their contents here.

5. **SealedPacket**: As soon as the transmitter seals the packet on the source-chain with their signature, it is called "SealedPacket". More information on how transmitters work can be found here.

6. **Capacitor**: Capacitor is responsible for storing messages in the form of a Packet. The packet is released when the transmitter pokes the capacitor to seal the packet. Capacitors allow for native batching of payloads for better gas performance. Learn more about the capacitor here.

7. **Switchboard**: Switchboards are authentication/verification modules that allow developers to have custom verification for their payloads/messages. They can be permissionlessly built and deployed by the community. Additional details can be found here.

8. **Transmitter**: Transmitters are the entities responsible for the transmission of Packets across layers. Their activity is entirely on-chain and managed by TransmitManager. To delve deeper into Transmitters, visit the documentation here.

9. **ChainSlug**: ChainSlug is a unique identifier for a particular network or Socket deployment. It differs from Chain ID or Network ID used on EVM networks. The assigned ChainSlugs can be found in the deployments section.

2.3 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the code base in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations —– found in the Discussion (4) section of the document —– may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.4 Scope

The engagement involved a review of the following targets:

Socket Data Layer Contracts

Repository	https://github.com/SocketDotTech/socket-DL
Version	socket-DL: `96534e6aad318b26627dacc50f4118fbe32a94fc`

Programs	CapacitorFactory ExecutionManager OpenExecutionManager TransmitManager BaseCapacitor SingleCapacitor SingleDecapacitor AddressAliasHelper RescueFundsLib Socket SocketBase SocketConfig SocketDst SocketSrc FastSwitchboard OptimisticSwitchboard SwitchboardBase ArbitrumL1Switchboard ArbitrumL2Switchboard NativeSwitchboardBase OptimismSwitchboard PolygonL1Switchboard PolygonL2Switchboard AccessControl AccessControlExtended AccessRoles Hasher Ownable SigIdentifiers SignatureVerifier
Type	Solidity
Platform	EVM-compatible

2.5 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of three and a half person-weeks. The assessment was conducted over the course of two calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Syed Faraz Abrar, Engineer
faith@zellic.io

Aaron Esau, Engineer
aaron@zellic.io

2.6 Project Timeline

The key dates of the engagement are detailed below.

July 12, 2023	Kick-off call
July 12, 2023	Start of primary review period
July 27, 2023	End of primary review period

3 Detailed Findings

3.1 Switchboard can steal extra execution fees

- **Target**: ExecutionManager
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: High
- **Impact**: Medium

Description

The `payAndCheckFees` function considers any fees left over after transmission fees, switchboard fees, and the minimum execution fees — the verification overhead fees added to `msg.value` transfer fees) — to be extra execution fees which can be optionally provided to encourage priority execution:

```
if (msg.value ≥ type(uint128).max) revert InvalidMsgValue();
uint128 msgValue = uint128(msg.value);

// transmission fees are per packet, so need to divide by number of
    messages per packet
transmissionFees =
    transmissionMinFees[transmitManager_][siblingChainSlug_] /
    uint128(maxPacketLength_);

uint128 minMsgExecutionFees = _getMinFees(
    minMsgGasLimit_,
    payloadSize_,
    executionParams_,
    siblingChainSlug_
);

uint128 minExecutionFees = minMsgExecutionFees +
    verificationOverheadFees_;
if (msgValue < transmissionFees + switchboardFees_ + minExecutionFees)
    revert InsufficientFees();

// any extra fee is considered as executionFee
executionFee = msgValue - transmissionFees - switchboardFees_;
```

The `switchboardFees_` and `verificationOverheadFees_` both come from the switchboard when `ISwitchboard.getMinFees` is called to fetch the fees in SocketSrc (these values are passed into `payAndCheckFees` as arguments):

```
/**
 * @notice Retrieves the minimum fees required for switchboard.
 * @param siblingChainSlug_ The slug of the destination chain for the
    message.
 * @param switchboard__ The switchboard address for which fees is
    retrieved.
 * @return switchboardFees fees required for message verification
 */
function _getSwitchboardMinFees(
    uint32 siblingChainSlug_,
    ISwitchboard switchboard__
)
    internal
    view
    returns (uint128 switchboardFees, uint128 verificationOverheadFees)
{
    (switchboardFees, verificationOverheadFees)
    = switchboard__.getMinFees(
        siblingChainSlug_
    );
}
```

Impact

A switchboard can return values from `ISwitchboard.getMinFees` such that the `payAndCheckFees` call does not revert with `InsufficientFees` but has no extra execution fee, thereby stealing from the executor and/or the user. The values can be configured in the on-chain switchboard by front-running the `SocketSrc.outbound` call.

The following steps represent the simplest exploitation of this issue, where each top-level, numbered item represents a separate transaction:

1. **The frontrunning transaction:** Switchboard fees are increased.
2. **The victim transaction:** Fees were calculated off-chain in advance (including an extra fee). On-chain, the outbound call is made, and the switchboard steals the extra fee.

Per the Socket Data Layer documentation, the `getMinFees` call should be done in the same transaction as the `outbound` call, preventing exploitation using those steps.

However, if this issue were to be exploited, it would require the switchboard to perform malicious behavior: increasing fees unexpectedly during the outbound call. Thus, in the event that the issue is exploited, it is reasonable to expect that the switchboard may perform other unexpected behavior aside from simply increasing fees such as configuring the switchboard to return one set of values for the first `_getMinFees` call and another for the second:

1. **The frontrunning transaction:** The switchboard contract is upgraded to change the behavior of `_getMinFees` such that its return value is variable and determined based on call order.
2. **The victim transaction:**

   a. The first call to `getMinFees` calls `_getMinFees` on the switchboard. The expected switchboard and overhead fees are returned such that the plug passes in the value it expected (i.e. the original minimum fees with an extra fee added on for priority execution purposes).

   b. The second call to the switchboard's `_getMinFees` returns a value for the switchboard fee that includes the extra fees passed into the outbound call such that no extra fees remain, and all fees go to the switchboard.

So, we believe calculating the minimum fees in the same transaction as making the outbound call does not effectively mitigate the risk. The protocol cannot enforce an implementation of a switchboard, so the threat model should include the switchboard behaving in any manner it chooses. While the user ultimately needs to assess the risk, it is important to acknowledge that the risk does exist.

Recommendations

We understand the extra fees are to encourage prioritized execution and that refunding them to the user to discourage sending extra fees would defeat this goal.

Either of the following solutions ensure the switchboard cannot increase fees (but can decrease — which would increase message priority incentive) while still allowing the user to pass extra execution fees:

- To ensure fees cannot be stolen, we recommend adding a `SocketSrc.outbound` caller-specified argument for the maximum value for `transmissionFees + switchboardFees_ + minExecutionFees` (i.e. the minimum fees required to send the message — without any extra fees).

- Alternatively, simply require the `SocketSrc.outbound` caller to specify an argument for the amount of extra fees — if any — and add this value to the `InsufficientFees` check.

The protocol is not directly at risk from this issue; the purpose of mitigating this issue would be to reduce risk and prevent potential harm to a user who does not sufficiently vet the plug's configured switchboard — or, since a plug implementation may allow a frontrunning attack to change the switchboard before the transaction, a user who does not sufficiently vet the plug.

Remediation

Socket Technology acknowledged the finding, noting that the system relies on reputation and that if a switchboard were to act maliciously, users would lose trust in the switchboard and/or the plug configured to use it:

> The Plugs are expected to only select Switchboards they trust after thoroughly vetting its fee mechanism.

We agree that malicious behavior would cause users to lose trust in the switchboard and/or the plug. However, we believe risk is still presented to the system if the issue is exploitable even once, which is a possibility presently determined by the plug implementation and configured switchboard. Mitigating the issue prevents the plug and switchboard from creating the opportunity to exploit the user in the first place.

3.2 Unconstrained `minMsgGasLimit` unaccounted for in fees

- **Target**: ExecutionManager
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Medium
- **Impact**: Medium

Description

The `minMsgGasLimit_` passed into the `SocketSrc.outbound` function specifies the minimum gas that the `SocketDst.inbound` executor must pass.

The setting is passed into `outbound`, then follows this chain:

1. `_validateAndSendFees(minMsgGasLimit_, ...)`
2. `_executionManager.payAndCheckFees(minMsgGasLimit_, ...)`
3. `_getMinFees(minMsgGasLimit_, ...)`

Finally, `_getMinFees` drops this value; the first parameter is not named:

```
function payAndCheckFees(
    uint256 minMsgGasLimit_,
    uint256 payloadSize_,
    bytes32 executionParams_,
    bytes32, // Zellic: this is `_getMinFees`
    uint32 siblingChainSlug_,
    uint128 switchboardFees_,
    uint128 verificationOverheadFees_,
    address transmitManager_,
    address switchboard_,
    uint256 maxPacketLength_
)
    external
    payable
    override
    returns (uint128 executionFee, uint128 transmissionFees)
{
    // [...]
}
```

Nowhere along this chain are limits enforced on `minMsgGasLimit`, and the value is not used when calculating fees.

Impact

The executor may take a loss if gas fees are high because of `minMsgGasLimit`. Additionally, messages are not guaranteed to be deliverable on the data layer of Socket if the gas limit were too high.

Note that this does not affect plugs; only executors are potentially negatively impacted.

Recommendations

Account for the `minMsgGasLimit` in fees.

Remediation

Socket Technology acknowledged this finding, noting that the code is simply incomplete in the assessment version and that fee accounting will be implemented in the future:

> For now `minMsgGasLimit` is part of `packetMessage` and it is used on destination side to check if provided `executionGasLimit` is enough. We plan to introduce detailed `_getMinFees` that would use both `minMsgGasLimit` and `payloadSize`.
>
> When we do it, the `executionFees[siblingChainSlug_]` that is present currently would break into parts, which would be multiplied with `minMsgGasLimit` and `payloadSize`.

3.3 Arbitraging against Socket Data Layer

- **Target**: ExecutionManager
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: Low
- **Impact**: Low

Description

A feature of Socket Data Layer is the ability to send `msg.value` with messages cross-chain. This essentially acts as a swap from the source chain's native coin to that of the destination chain.

The price of the source chain's native coin in terms of the destination chain's native coin is determined by a ratio set by the `FEES_UPDATER_ROLE` role. When paying fees for the message, the `_getMinFees` debits native tokens based on that ratio and the requested `msgValue`:

```
// decodes and validates the msg value if it is under given transfer
    limits and calculates
// the total fees needed for execution for given payload size and msg
    value.
function _getMinFees(
    uint256,
    uint256 payloadSize_,
    bytes32 executionParams_,
    uint32 siblingChainSlug_
) internal view returns (uint128) {
    // [...]

    uint256 params = uint256(executionParams_);
    uint8 paramType = uint8(params >> 248);

    if (paramType == 0) return executionFees[siblingChainSlug_];
    uint256 msgValue = uint256(uint248(params));

    if (msgValue < msgValueMinThreshold[siblingChainSlug_])
        revert MsgValueTooLow();
    if (msgValue > msgValueMaxThreshold[siblingChainSlug_])
        revert MsgValueTooHigh();

    uint256 msgValueRequiredOnSrcChain = (relativeNativeTokenPrice[
        siblingChainSlug_
```

```
        ] * msgValue) / 1e18;

        // [...]
    }

    // [...]

    function setRelativeNativeTokenPrice(
        uint256 nonce_,
        uint32 siblingChainSlug_,
        uint256 relativeNativeTokenPrice_,
        bytes calldata signature_
    ) external override {
        // [...]
        _checkRoleWithSlug(FEES_UPDATER_ROLE, siblingChainSlug_,
        feesUpdater);
        // [...]
        relativeNativeTokenPrice[siblingChainSlug_]
        = relativeNativeTokenPrice_;
        // [...]
    }
```

There may be a delay between the fee updater's submission of the relative native token prices and the actual relative price.

Arbitrage happens between at least two exchanges.[1] Socket Data Layer acts as one exchange, and any exchange (e.g., Uniswap, Curve, or even another Socket Data Layer path) may be used as the second.

The core of the issue is that there is an arbitrage opportunity anytime the relative native token price difference between Socket Data Layer and another exchange is exploitable for profit (i.e., after fees), which is especially likely to happen in a volatile market.

There are a number of protections implemented that may make an arbitrage opportunity with Socket Data Layer less trivial to exploit:

- Socket Technology noted that the fee updater will quickly submit signatures to try to keep the price as up-to-date as possible.
- There is the existence of the maximum `msgValue` threshold:

[1] In arbitrage, there may be one or more intermediate exchange(s) used in a chain to maximize profitability. But the minimum number of exchanges required is two.

```
if (msgValue > msgValueMaxThreshold[siblingChainSlug_])
    revert MsgValueTooHigh();
```

- Cross-chain message transfers do not occur immediately. Any delay leaves room for more price disparities between the involved exchanges, potentially ending the opportunity and causing a loss to the arbitrageur.

However, none of these eliminates the possibility of an arbitrage opportunity; while these measures may mitigate the ease of exploitation, if the price ratio is not updated atomically (i.e., within the same transaction) before sending a packet, the potential for a price difference exists.

Additionally, the maximum `msgValue` threshold can be bypassed by sending many messages in the same packet, splitting transmission fees.

The cross-chain message transfer may not be instant, but the Socket Data Layer exchange occurs immediately on the sending chain. Only, the output native coin is essentially redeemed once the message arrives on the destination chain. So, if the other exchange is on the source chain, the arbitrage attack can be atomically executed.

Impact

For many exchanges, arbitrage is generally beneficial because of its role in promoting market efficiency and price convergence; that is, arbitrageurs facilitate the alignment of asset values across decentralized exchanges, reducing spreads.

However, Socket Data Layer does not operate an order book and swaps do not impact pricing, so arbitrage against it does not resolve price inefficiencies and potentially has negative impact on the executors who provide the `msg.value` liquidity on the destination chain.

When successfully executed, the arbitrageur "wins" the price difference between the trade. This value must come from somewhere —– and the "losers" are the liquidity providers who lost their liquidity to a bad trade. In Socket Data Layer's case, the executors ultimately provide the native coin to the destination chain, so they are the entity negatively impacted by the fee updater's slow response.

Assuming no limit to the number of messages that can be sent during a price inefficiency, the profits are only limited by how many native coins the executor is able to provide on the destination chain. Once the executors run out of destination chain native coins, the arbitrage opportunity closes.

Recommendations

Ensure executors evaluate whether the fees paid for the `msgValue` transfer are satisfactory before executing the message on chain.

Set up monitoring to ensure other on-chain oracles' prices do not vary too much fron the fee updater oracle's prices.

Remediation

Socket Technology acknowledged this finding, noting:

> `msgValue` checks are already in [the execution client] to check it before execution and we are working on the monitoring system.

3.4 Random address recovered from ECDSA's signature recovery may be used for executor fee accounting

- **Target**: OpenExecutionManager
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

In OpenExecutionManager, the NatSpec for `isExecutor` states the following:

```
/**
 * @notice This function allows all executors
 * @notice The executor recovered here can be a random address hence
    should not be used for fee accounting
 * @param packedMessage Packed message to be executed
 * @param sig Signature of the message
 * @return executor Address of the executor
 * @return isValidExecutor Boolean value indicating whether the executor
    is valid or not
 */
function isExecutor(
    bytes32 packedMessage,
    bytes memory sig
) external view override returns (address executor, bool isValidExecutor)
    {
    executor = signatureVerifier__.recoverSigner(packedMessage, sig);
    isValidExecutor = true;
}
```

Specifically, the notice `The executor recovered here can be a random address henc e should not be used for fee accounting` is important.

Impact

The address returned by this function is used within `_execute()` when updating the executor's fee accounting:

```
executionManager__.updateExecutionFees(
    executor_, // Zellic: this address is from isExecutor()
```

```
    uint128(messageDetails_.executionFee),
    messageDetails_.msgId
);
```

If the address recovered is random, the accounting would be incorrect.

Recommendations

Document prominently above the `isExecutor()` function, or alternatively above the call to `updateExecutionFees()` in `_execute()`, that executors must provide a valid signature that recovers to their address, as otherwise the executor fee accounting will be done incorrectly.

Remediation

This issue has been acknowledged by Socket Technology, and a fix was implemented in commit 00688523. They have also stated that executor's nodes will go through rigorous testing that should catch any issues like this.

3.5 ExecutionManager should assert function requirements

- **Target**: ExecutionManager
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The `payAndCheckFees` function uses the `maxPacketLength_` argument to reduce the transmission fees to split the fee between messages in the packet:

```
function payAndCheckFees(
    uint256 minMsgGasLimit_,
    uint256 payloadSize_,
    bytes32 executionParams_,
    bytes32,
    uint32 siblingChainSlug_,
    uint128 switchboardFees_,
    uint128 verificationOverheadFees_,
    address transmitManager_,
    address switchboard_,
    uint256 maxPacketLength_
)
    external
    payable
    override
    returns (uint128 executionFee, uint128 transmissionFees)
{
    if (msg.value ≥ type(uint128).max) revert InvalidMsgValue();
    uint128 msgValue = uint128(msg.value);

    // transmission fees are per packet, so need to divide by number of
    messages per packet
    transmissionFees =
        transmissionMinFees[transmitManager_][siblingChainSlug_] /
        uint128(maxPacketLength_);
```

This value is ultimately passed from the `SocketSrc.outbound` function, where it is fetched from the capacitor:

```
function outbound(
    uint32 siblingChainSlug_,
    uint256 minMsgGasLimit_,
    bytes32 executionParams_,
    bytes32 transmissionParams_,
    bytes calldata payload_
) external payable override returns (bytes32 msgId) {
    // [...]

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

    // [...]

    ISocket.Fees memory fees = _validateAndSendFees(
        minMsgGasLimit_,
        uint256(payload_.length),
        executionParams_,
        transmissionParams_,
        plugConfig.outboundSwitchboard__,
        plugConfig.capacitor__.getMaxPacketLength(), // Zellic: this is
    `maxPacketLength_`
        siblingChainSlug_
    );
```

During our assessment, there were only two capacitor/decapacitor pairs available to deploy through CapacitorFactory:

- SingleCapacitor — hardcoded maximum packet length of 1.
- HashChainCapacitor — variable maximum packet length between 0[2] and `HashChainCapacitor.MAX_LEN`:

```
constructor(
    address socket_,
    address owner_,
    uint256 maxPacketLength_
```

[2] Though HashChainCapacitor is out of scope, we wanted to document the possibility of a division by zero in the transmission fee splitting because the maximum packet length can be zero.

```
) BaseCapacitor(socket_, owner_) {
    if (maxPacketLength > MAX_LEN) revert InvalidPacketLength();
    maxPacketLength = maxPacketLength_;
}
```

There are currently limits on the `maxPacketLength`. However, there is risk of a future capacitor/decapacitor pair being written that does not enforce a maximum packet length because the variable is checked at the capacitor level.

Impact

If a `maxPacketLength` were greater than the transmission fees, no transmission fees would be paid. Additionally, a tiny amount of transmission fees are regularly lost in precision from the division.

As the transmission fees decrease or `maxPacketLength` increases, the division loses precision, and fees are lost because the division rounds down.

Recommendations

We recommend the following:

- Enforce a maximum value for `maxPacketLength` as a `payAndCheckFees` function requirement or in CapacitorFactory when deploying the capacitor/decapacitor contract pair.
- Enforce a minimum value transmission fee, though keep it relatively insignificant. This ensures parties are properly compensated even in cases of high `maxPacketLength` values.
- Round the division up to prevent transmission fee losses due to precision.

Remediation

Socket Technology acknowledged this finding, adding that they will add the `maxPacketLength` check in CapacitorFactory. This change was implemented in commit 83d6d0af.

3.6 Risk of proof type confusion

- **Target**: SocketDst
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The Socket Data Layer protocol supports capacitors and decapacitors. Capacitors are used on the sending chain to pack packets into messages. These messages can then be sealed by transmitters and subsequently transmitted to a remote chain. A decapacitor on the remote chain would then be able to unpack this message so that it may be executed.

There are currently two types of capacitors:

- SingleCapacitor
- HashChainCapacitor

A plug can choose to change the capacitors it uses by simply changing the switchboards that it is using (which is what the capacitors and decapacitors are connected to). This can be done without any restrictions at any point in time.

Impact

Currently, the two capacitor and decapacitor implementations are mutually exclusive. That is to say that a packed message packed by the SingleCapacitor would fail to be unpacked by the HashChainDecapacitor. This is what the ideal scenario is.

However, it is possible that with future implementations of new capacitor types, a message packed by one capacitor may actually be able to be unpacked by a completely different decapacitor. In this case, the unpacked message would very likely not match the original message that was sent, and therefore an arbitrary message may get executed.

Recommendations

There is no immediate risk presented by the capacitors and decapacitors in scope, but we recommend that Socket Data Layer be very careful when introducing new types of capacitors. Ensure that all implemented capacitors are mutually exclusive (as they are now), or consider adding restrictions on when a plug can change its switchboard implementations.

Additionally, consider adding type information to packets that specifies which capaci-

tor generated the proof and thus which decapacitor should be used to verify message inclusion using the proof. This would eliminate this class of threat entirely because type confusion would no longer be possible.

Remediation

Socket Technology acknowledged this finding, noting that users should only use vetted and reputable plugs, and such plugs should pause functionality and take care of all in-flight messages prior to changing switchboards. They also noted that changing switchboards should be a fairly rare occurrence:

> Switchboard change is expected to be rare.
>
> Plugs would have to pause new messages and finish all in flight ones before they change it for graceful migration.

3.7 Gas optimization for switchboard registration

- **Target**: SwitchboardBase
- **Category**: Gas Optimization
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

Within the `registerSiblingSlug()` function of NativeSwitchboardBase, there is an already initialized check:

```
function registerSiblingSlug(/* ... */)
    external override onlyRole(GOVERNANCE_ROLE) {
    if (isInitialized) revert AlreadyInitialized();

    initialPacketCount = initialPacketCount_;
    (address capacitor, ) = socket__.registerSwitchboardForSibling(/* ...
    */);

    isInitialized = true;
    capacitor__ = ICapacitor(capacitor);
    remoteNativeSwitchboard = remoteNativeSwitchboard_;
}
```

This check is here because the `registerSwitchboardForSibling()` function that is called on the `socket__` can only be called once. This initialization check prevents gas from being wasted on an unnecessary call if the switchboard has already been registered.

Impact

The above check is nonexistent in the corresponding SwitchboardBase contract, which can lead to a waste of a small amount of gas if the switchboard owner calls `registerSiblingSlug()` after the switchboard has already been initialized.

Recommendations

Consider adding an initialization check in the `registerSiblingSlug()` function within the SwitchboardBase contract.

Remediation

This issue has been acknowledged by Socket Technology.

3.8 Ambiguous return value

- **Target**: BaseCapacitor
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

Within BaseCapacitor, the `getLastFilledPacket()` is written as follows:

```
/**
 * @dev Returns the count of the latest packet that finished filling.
 * @dev Returns 0 in case no packets are filled.
 * @return lastFilledPacket count of the latest packet.
 */
function getLastFilledPacket()
    external
    view
    returns (uint256 lastFilledPacket)
{
    return _nextPacketCount == 0 ? 0 : _nextPacketCount - 1;
}
```

Impact

The NatSpec for the function states that the function should return 0 in case no packets are filled.

However, looking at the code, the function also returns 0 when one packet is filled. The storage variable `_nextPacketCount` starts at 0 and is incremented by 1 by the `addPackedMessage()` function.

This function is currently unused and thus does not pose any issues. However, if a Plug or any user of the capacitors decide to rely on this function for any critical functionality, the ambiguous return value may result in a security vulnerability.

Recommendations

Consider clarifying this ambiguity within the NatSpec of the function.

Remediation

This issue has been acknowledged by Socket Technology, and a fix was implemented in commit ac4f5fcd.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1 Disclaimer about governance implementation

Note that the future security of Socket Data Layer relies on the proper behavior of the governance implementation; that is, malicious governance behavior can potentially compromise any protocols built on Socket Data Layer.

For example, if the governance contract were to change `capacitorFactory`, a malicious decapacitor could allow any message through on the destination chain.

So, it is critical that the governance implementation —– which was out of scope of this assessment — is secure and does not behave maliciously.

4.2 Cautions to cross-chain app developers

We wanted to warn cross-chain app developers who build on Socket Data Layer that certain security checks are left up to the app to implement, including (but not necessarily limited to)

- requiring that the `inbound` function caller is the socket — as in,

```
require(msg.sender == socket);
```

- and ensuring the remote slug and plug are trusted; otherwise, any remote plug may be able to send messages to the app.

4.3 Trust assumptions

Within the Socket Data Layer protocol, there exists multiple roles and contracts that have the authority to perform their own set of actions. In this section, we document the trust assumptions for each of these roles.

Plugs

Protocols build on top of the Socket Data Layer by independently developing and deploying a Plug as part of their set of contracts. These contracts would allow users of those protocols to send packets (messages) across chains. A corresponding Plug contract on the remote chain would then receive the packet and execute it. Users should ensure they trust any protocol and its smart contracts.

Socket Data Layer has made the assumption that users should only use plugs that are a part of vetted and reputable protocols. This would include plugs that have likely been independently audited and are verified on Etherscan (or an equivalent block scanner for other chains). This is because plugs are externally called by the Socket contracts at a critical point, where plugs may be able to abuse bugs within the Socket Data Layer protocol for their own gain.

Plugs are also trusted to not censor packets based on arbitrary criteria.

Aside from that, plugs do not have any special permissions. They call into a Socket's `outbound()` function to send a cross-chain packet, and a receiving socket would call the Plug's `inbound()` function in order to execute a cross-chain packet.

As with any new protocol, vet a protocol's smart contracts before using them.

Switchboards

Switchboards are contracts that act as a permission layer that cross-chain packets need to go through before being allowed to be executed.

The sender plug and the receiving plug are both connected to their own switchboards (in this case, the outbound and inbound switchboards). Prior to a packet being executed on the receiving chain, the Socket checks to ensure that the inbound switchboard for the receiving plug can verify that this packet was sent to it by the corresponding outbound switchboard (which are connected across chains).

Therefore, when a packet is sent, the outbound switchboard must perform any necessary checks and then send the packet's hash to the remote inbound switchboard.

Socket Data Layer has made the assumption that plugs should only use vetted and reputable switchboards. This would include switchboards that have likely been independently audited and are verified on Etherscan (or an equivalent block scanner for other chains). This is because switchboards are externally called by the Socket contracts at a critical point, where they may be able to abuse bugs within the Socket Data Layer protocol for their own gain.

The above trust assumptions are also important because switchboards also control their own switchboard fee amounts. They are able to change these fee amounts at

any time, which can be abused to steal excess execution fees that are provided by the packet sender for the executor. See finding 3.1 for more details.

Switchboards are also trusted to not censor packets based on arbitrary criteria.

Aside from that, switchboards do not have any other special permissions.

Transmitters

Transmitters are a special role within the Socket Data Layer protocol. They are responsible for sealing packets that have been sent on a source chain by signing a digest. Anyone can permissionlessly propose a packet using the signature on the destination chain.

Only transmitters have the permission to seal. They cannot censor packets, as sealing can only be done for sequential packets, and once a packet is sealed and the digest is signed, anyone is able to propose it on the destination chain.

Transmitters are crucial for the liveliness of the system. Socket Data Layer must ensure that they have adequate transmitters at all times to ensure packets are sent without long delays. If there are not any transmitters, the protocol would come to a halt.

Transmitters are trusted to not propose malicious packets on the receiving chain, but on the off chance that this does occur, the malicious packets would not pass verification on the inbound switchboard.

Aside from that, transmitters do not have any other special permissions.

Executors

Executors are a special role within the Socket Data Layer protocol. They are responsible for executing messages on the receiving chain once a transmitter has proposed them. They cannot censor packets, as any executor can execute any packets that have been proposed.

Executors are crucial for the liveliness of the system. Socket Data Layer must ensure that they have adequate executors at all times to ensure that packets are executed without long delays. If there are no executors, the protocol would come to a halt.

Aside from that, executors do not have any other special permissions.

5 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1 Module: ArbitrumL1Switchboard.sol

Function: `initiateNativeConfirmation(byte[32] packetId_, uint256 maxSubmissionCost_, uint256 maxGas_, uint256 gasPriceBid_, address callValueRefundAddress_, address remoteRefundAddress_)`

This function is used to send a packet ID to the remote switchboard, essentially allowing it on the remote switchboard.

Inputs

- `packetId_`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the packet ID that is sent to be received by the remote switchboard.
- `maxSubmissionCost_`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the maximum cost to submit a retryable ticket for this message. See the Arbitrum Docs.
- `maxGas_`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the maximum gas used to cover L2 execution of this retryable ticket. See the Arbitrum Docs.
- `gasPriceBid_`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.

  - **Impact**: This is the gas price bid for L2 execution of this ticket. See the Arbitrum Docs.
- `callValueRefundAddress_`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the L2 address where any `msg.value` is refunded if this ticket times out or gets cancelled. See the Arbitrum Docs.
- `remoteRefundAddress_`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the L2 address where any excess submission cost is refunded. See the Arbitrum Docs.

Branches and code coverage (including function calls)

Intended branches

- Cross-chain native confirmation is sent successfully.
  ☑ Test coverage

5.2 Module: ArbitrumL2Switchboard.sol

Function: `initiateNativeConfirmation(byte[32] packetId_)`

This function is used to send a packet ID to the remote switchboard, essentially allowing it on the remote switchboard.

Inputs

- `packetId_`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the packet ID that is sent to be received by the remote switchboard.

Branches and code coverage (including function calls)

Intended branches

- Cross-chain native confirmation is sent successfully.
  ☑ Test coverage

5.3 Module: CapacitorFactory.sol

Function: `deploy(uint256 capacitorType_, uint32 None, uint256 maxPacketLength_)`

Deploys a capacitor and decapacitor. Configuration settings include `capacitorType` and `maxPacketLength_`. It is intended to be called by `SocketConfig.registerSwitchboardForSibling`, but it can be called directly; however, this would have no security impact as the deployed contracts would not be configured in the socket.

Inputs

- `capacitorType_`
  - **Control**: Full.
  - **Constraints**: Must be equal to `SINGLE_CAPACITOR` or `HASH_CHAIN_CAPACITOR`.
  - **Impact**: Determines which type of contract is deployed.
- `siblingChainSlug_`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: None — this value is dropped.
- `maxPacketLength_`
  - **Control**: Full.
  - **Constraints**: None, though it should be enforced that it is nonzero since could otherwise lead to a division-by-zero scenario.[1] informally.]
  - **Impact**: Controls the maximum number of messages allowed to be included in a packet for the HashChainCapacitor and HashChainDecapacitor. This value is ignored for SingleCapacitor and SingleDecapacitor.

Branches and code coverage (including function calls)

Intended branches

- Deploys a SingleCapacitor and SingleDecapacitor successfully.
  ☑ Test coverage
- Deploys a HashChainCapacitor and HashChainDecapacitor successfully.
  ☑ Test coverage

[1] This is not documented in a finding because HashChainCapacitor is out of scope and there is little or no security impact to this. The observation was communicated to [^CustomerName

Negative behavior

- Invalid `capacitorType_` is passed in.
  ☑ Negative test

5.4 Module: ExecutionManager.sol

Function: `payAndCheckFees(uint256 minMsgGasLimit_, uint256 payloadSize_, byte[32] executionParams_, byte[32] None, uint32 siblingChainSlug_, uint128 switchboardFees_, uint128 verificationOverheadFees_, address transmitManager_, address switchboard_, uint256 maxPacketLength_)`

This is the function called by `SocketDst.outbound` to store the fees for an outgoing message. It performs accounting to ensure at least the minimum fees are paid and that fees are paid to the proper users.

Additionally, it is responsible for enforcing the maximum message payload size of 3,000.

The function can be called by anyone with arbitrary values; however, it only costs the caller and can have no negative impact. If an invalid slug gets fees stored, they can be rescued thanks to the `rescueFunds` function.

Inputs

- `minMsgGasLimit_`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: This value is dropped.
- `payloadSize_`
  - **Control**: Full.
  - **Constraints**: This value must be less than or equal to 3,000.
  - **Impact**: Used for enforcing the payload length limit.
- `executionParams_`
  - **Control**: Full.
  - **Constraints**: Must be a properly formatted execution params value.
  - **Impact**: Used for optionally specifying a `msg.value` to send to the destination chain.
- `transmissionParams`
  - **Control**: Full.
  - **Constraints**: None.

  - **Impact**: This value is dropped.
- `siblingChainSlug_`
  - **Control**: Full.
  - **Constraints**: None, unless the execution params specify a nonzero `msg.value` to transfer, in which case invalid values will lead to a reversion because the minimum and maximum thresholds for `msg.value` will both be zero exclusive.
  - **Impact**: Used for accounting the fees.
- `switchboardFees_`
  - **Control**: Full.
  - **Constraints**: Cannot be too high of a value or it will revert with `InsufficientFees`.
  - **Impact**: The switchboard fees to charge.
- `verificationOverheadFees_`
  - **Control**: Full.
  - **Constraints**: Cannot be too high of a value or it will revert with `InsufficientFees`.
  - **Impact**: The verification overhead fees to charge as part of the execution fees.
- `transmitManager_`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Used to calculate the transmission fees. Passing an invalid address is not possible via `SocketDst.outbound`, and manually calling with an invalid address has no negative effect on the contract (only on the caller).
- `switchboard_`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Used for accounting where the switchboard fees should go.
- `maxPacketLength_`
  - **Control**: Full.
  - **Constraints**: Must be nonzero.
  - **Impact**: Divides off of the transmission fees.

Branches and code coverage (including function calls)

Intended branches

- Fees are calculated and accounted for.

☑ Test coverage

Negative behavior

- Fees paid are insufficient.
  ☑ Negative test
- `msg.value` is greater than the maximum uint128 value.
  ☑ Negative test
- Payload size is too large.
  ☑ Negative test
- `msg.value` is less than the minimum threshold for the sibling slug.
  ☑ Negative test
- `msg.value` is more than the maximum threshold for the sibling slug.
  ☑ Negative test

Function: `verifyParams(byte[32] executionParams_, uint256 msgValue_)`

Called by `SocketDst._verify` to ensure the execution parameters are properly followed.

Inputs

- `executionParams_`
  - **Control**: Full.
  - **Constraints**: Must be a valid execution parameters value.
  - **Impact**: The rules for execution that must be followed.
- `msgValue_`
  - **Control**: Full.
  - **Constraints**: If the eight bits following the 248th bit of `executionParams_` is nonzero, the first 248 bits of `executionParams` must be greater or equal to this value.
  - **Impact**: Used when enforcing a minimum `msg.value` of the inbound destination socket call.

Branches and code coverage (including function calls)

Intended branches

- The `msg.value` passed in is sufficient given the `executionParams_`.
  ☑ Test coverage

Negative behavior

- The `msg.value` passed in is insufficient.

☑ Negative test

Function: `withdrawTransmissionFees(uint32 siblingChainSlug_, uint128 amount_)`

Withdraws `amount_` of transmission and execution fees to `siblingChainSlug_` chain, sending the funds to the transmit manager.

Inputs

- `siblingChainSlug_`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The chain slug to look up in `totalExecutionAndTransmissionFees`.
- `amount_`
  - **Control**: Full.
  - **Constraints**: Must be <= the amount stored in `totalExecutionAndTransmissionFees[siblingChainSlug_]`.
  - **Impact**: The amount to withdraw.

Branches and code coverage (including function calls)

Intended branches

- Withdraws fees successfully.
  ☑ Test coverage

Negative behavior

- The `amount_` is too high.
  ☑ Negative test
- The call to `receiveFees` reverts.
  ☐ Negative test

Function: `_getMinFees(uint256 None, uint256 payloadSize_, byte[32] executionParams_, uint32 siblingChainSlug_)`

Calculates the minimum execution fees required for a given message.

Inputs

- `minMsgGasLimit_`
  - **Control**: Full.

  - **Constraints**: None.
  - **Impact**: None — value is dropped.
- `payloadSize_`
  - **Control**: Full.
  - **Constraints**: Must be less than or equal to 3000 bytes.
  - **Impact**: Used for enforcing the maximum message payload size limit.
- `executionParams_`
  - **Control**: Full.
  - **Constraints**: Must be a properly formatted execution parameters value.
  - **Impact**: If the bits from 248 to 256 are equal to zero, then there is no `msg.value` requirement. Otherwise, it uses the first 248 bits to calculate the fees required on the source chain to cover the `msg.value` requested on the destination chain using a mapping of native token prices. Note that this may enable arbitrage opportunities.
- `siblingChainSlug_`
  - **Control**: Full.
  - **Constraints**: Must be a key in `msgValueMinThreshold` if the 248th to 256th bits of `executionParams_` and the lower 248 bits are both nonzero; otherwise, the maximum `msgValue` check will fail.
  - **Impact**: Used for calculating the fee required to cover the `msg.value` transfer.

Branches and code coverage (including function calls)

Intended branches

- Calculates fees given a `msg.value` to the sibling chain.
  ☑ Test coverage
- Calculates fees given no `msg.value` transfer.
  ☑ Test coverage

Negative behavior

- Sibling chain is not configured in `msgValueMaxThreshold`.
  ☐ Negative test
- Payload size is greater than 3,000.
  ☑ Negative test
- The `totalNativeValue` is greater than the uint128 maximum value.
  ☐ Negative test
- The `msgValue` is outside of the min and max settings.
  ☑ Negative test

5.5 Module: FastSwitchboard.sol

Function: `allowPacket(byte[32] root_, byte[32] packetId_, uint256 proposalCount_, uint32 srcChainSlug_, uint256 proposeTime_)`

This function is used by sockets to determine if the specified packet is allowed by this switchboard.

Inputs

- `root_`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the packet root that is being verified.
- `packetId_`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the packet ID, used to check if this packet plus proposal was disallowed specifically.
- `proposalCount_`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the proposal count for this packet, used to check if this packet plus proposal was disallowed specifically.
- `srcChainSlug_`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the chain slug of the source chain, used to check if packets are specifically disallowed for this chain and to determine if the packet ID is valid.
- `proposeTime_`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the time that packet was proposed to the switchboard, used to allow packets that have been waiting for attestations for too long.

Branches and code coverage (including function calls)

Intended branches

- Should return true for a valid, allowed packet.

☑ Test coverage

- Packets are allowed if enough time has passed without enough attestations.
  ☑ Test coverage

Negative behavior

- Should return false when the switchboard is globally tripped.
  ☐ Negative test
- Should return false if the path to the source chain is tripped.
  ☐ Negative test
- Should return false if the specific packet proposal is tripped.
  ☐ Negative test
- Should return false if the `packetCount` is invalid for the source chain.
  ☐ Negative test
- Should return false if there are not enough attestations and the timeout has not been reached.
  ☐ Negative test

Function: `attest(byte[32] packetId_, uint256 proposalCount_, byte[32] root_, byte[] signature_)`

This function is used by watchers to attest packets. Packets with enough attestations are then allowed to be executed.

Inputs

- `packetId_`
  - **Control**: Fully controlled.
  - **Constraints**: Must be a valid packet ID.
  - **Impact**: The packet root is fetched using this.
- `proposalCount_`
  - **Control**: Fully controlled.
  - **Constraints**: Must be a valid proposal count.
  - **Impact**: The packet root is fetched using this.
- `root_`
  - **Control**: Fully controlled.
  - **Constraints**: This must match the packet root that is fetched using the two previous arguments.
  - **Impact**: This is the expected packet root.
- `signature_`
  - **Control**: Fully controlled.

  - **Constraints**: This must recover to the signature of a watcher.
  - **Impact**: This is the signature that is used to ensure that the packet is attested by a watcher.

Branches and code coverage (including function calls)

Intended branches

- Attestation works with all valid arguments.
  ☑ Test coverage

Negative behavior

- Attestation fails if the root fetched from the socket is 0.
  ☑ Negative test
- Attestation fails if the expected root does not match the root fetched.
  ☑ Negative test
- Attestation fails if the same watcher attempts to attest the same packet more than once.
  ☑ Negative test
- Attestation fails if the signature does not recover to a watcher address.
  ☑ Negative test

5.6 Module: NativeSwitchboardBase.sol

Function: `allowPacket(byte[32] root_, byte[32] packetId_, uint256 None, uint32 None, uint256 None)`

This function is used by sockets to determine if the specified packet is allowed by this switchboard.

Inputs

- `root_`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the packet root that is being verified.
- `packetId_`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the packet ID, used to check if this packet has been received by this switchboard.

Branches and code coverage (including function calls)

Intended branches

- Should return true for a valid, allowed packet.
  ☑ Test coverage

Negative behavior

- Should return false when the switchboard is globally tripped.
  ☐ Negative test
- Should return false if this specific packet has not been received by this switchboard.
  ☐ Negative test
- Should return false if the `packetCount` is invalid for the source chain.
  ☐ Negative test

Function: `receivePacket(byte[32] packetId_, byte[32] root_)`

This function is used to receive a packet ID and root that are to be allowed by this switchboard.

Inputs

- `packetId_`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the packet ID. The root (next argument) is mapped to this ID.
- `root_`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the packet root that will now be allowed by this switchboard.

Branches and code coverage (including function calls)

Intended branches

- The packet ID and root provided are stored correctly within this contract.
  ☑ Test coverage

Negative behavior

- Should only be callable by the remote switchboard (tested for all implemented

native switchboards).
☑ Negative test

5.7 Module: OptimismSwitchboard.sol

Function: `initiateNativeConfirmation(byte[32] packetId_)`

This function is used to send a packet ID to the remote switchboard, essentially allowing it on the remote switchboard.

Inputs

- `packetId_`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the packet ID that is sent to be received by the remote switchboard.

Branches and code coverage (including function calls)

Intended branches

- Cross-chain native confirmation is sent successfully.
  ☑ Test coverage

5.8 Module: OptimisticSwitchboard.sol

Function: `allowPacket(byte[32] None, byte[32] packetId_, uint256 proposalCount_, uint32 srcChainSlug_, uint256 proposeTime_)`

This function is used by sockets to determine if the specified packet is allowed by this switchboard.

Inputs

- `packetId_`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the packet ID, used to check if this packet plus proposal was disallowed specifically.
- `proposalCount_`

  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the proposal count for this packet, used to check if this packet plus proposal was disallowed specifically.
- `srcChainSlug_`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the chain slug of the source chain, used to check if packets are specifically disallowed for this chain, and to determine if the packet ID is valid.
- `proposeTime_`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the time that packet was proposed to the switchboard, used to allow packets that have been waiting for attestations for too long.

Branches and code coverage (including function calls)

Intended branches

- Should return true for a valid, allowed packet.
  ☑ Test coverage
- Packets are allowed if enough time has passed without enough attestations.
  ☑ Test coverage

Negative behavior

- Should return false when the switchboard is globally tripped.
  ☐ Negative test
- Should return false if the path to the source chain is tripped.
  ☑ Negative test
- Should return false if the specific packet proposal is tripped.
  ☐ Negative test
- Should return false if the `packetCount` is invalid for the source chain.
  ☐ Negative test

5.9 Module: PolygonL1Switchboard.sol

Function: `initiateNativeConfirmation(byte[32] packetId_)`

This function is used to send a packet ID to the remote switchboard, essentially allowing it on the remote switchboard.

Inputs

- `packetId_`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the packet ID that is sent to be received by the remote switchboard.

Branches and code coverage (including function calls)

Intended branches

- Cross-chain native confirmation is sent successfully.
  ☑ Test coverage

Function: `_processMessageFromChild(byte[] message_)`

This function is used to process a message sent by the remote L2 switchboard.

Inputs

- `message_`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This contains the packet ID and root that are now going to be allowed by this switchboard.

Branches and code coverage (including function calls)

Intended branches

- Received packets are stored correctly.
  ☐ Test coverage

5.10 Module: PolygonL2Switchboard.sol

Function: `initiateNativeConfirmation(byte[32] packetId_)`

This function is used to send a packet ID to the remote switchboard, essentially allowing it on the remote switchboard.

Inputs

- `packetId_`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the packet ID that is sent to be received by the remote switchboard.

Branches and code coverage (including function calls)

Intended branches

- Cross-chain native confirmation is sent successfully.
  ☑ Test coverage

Function: `_processMessageFromRoot(uint256 None, address rootMessageSender_, byte[] data_)`

This function is used to process a message sent by the remote L1 switchboard.

Inputs

- `rootMessageSender_`
  - **Control**: Not controlled.
  - **Constraints**: This is the address of the sender of this message from the remote L1 chain.
  - **Impact**: Used to ensure that this function is only called by the L1 switchboard.
- `data_`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This contains the packet ID and root that are now going to be allowed by this switchboard.

Branches and code coverage (including function calls)

Intended branches

- Received packets are stored correctly.
  - ☐ Test coverage

Negative behavior

- Should revert if the message received is not from the L1 switchboard.
  - ☐ Negative test

5.11 Module: SingleCapacitor.sol

Function: `addPackedMessage(byte[32] packedMessage_)`

Takes the provided `packedMessage_` hash and stores it in storage, mapping it to the next available packet count.

Inputs

- `packedMessage_`
  - **Control**: Not controlled.
  - **Constraints**: Constrained to be a `Keccak256` hash.
  - **Impact**: This hash is stored into this contract's storage within the `_roots` mapping.

Branches and code coverage (including function calls)

Intended branches

- The `packedMessage_` is added to the `_roots` mapping correctly.
  - ☑ Test coverage

Negative behavior

- Should revert if not called by the socket.
  - ☑ Negative test

Function: `sealPacket(uint256 None)`

Fetches the next packed message hash that must be sealed and returns the hash along with the packet count.

Branches and code coverage (including function calls)

Intended branches

- Should seal a packet and return the hash and packet count correctly.

☑ Test coverage

Negative behavior

- Should revert if not called by the socket.
  ☑ Negative test
- Should revert if there are no packets to be sealed.
  ☑ Negative test

5.12 Module: SingleDecapacitor.sol

Function: `verifyMessageInclusion(byte[32] root_, byte[32] packedMessage_, byte[] None)`

Verifies that the provided `packedMessage_` should be part of the provided packet root.

Inputs

- `root_`
  - **Control**: Completely controlled by the transmitter who proposed this packet.
  - **Constraints**: N/A.
  - **Impact**: The `packedMessage_` is verified to be a part of this packet.
- `packedMessage_`
  - **Control**: The inputs that determine this hash are fully controlled, except the `chainSlug`.
  - **Constraints**: The `chainSlug` input to this is preset to the chain the calling Socket (and by extension, this capacitor) is deployed in.
  - **Impact**: This is verified to be a part of the packet root that is passed in.

Branches and code coverage (including function calls)

Intended branches

- The `packedMessage_` is verified correctly.
  ☑ Test coverage

5.13 Module: SocketConfig.sol

Function: `connect(uint32 siblingChainSlug_, address siblingPlug_, address inboundSwitchboard_, address outboundSwitchboard_)`

This function configures a Plug and stores the sibling plug, switchboard, and capacitor/decapacitor addresses within the config. The `msg.sender` is the Plug that is being configured.

Inputs

- `siblingChainSlug_`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: Used to fetch capacitor and decapacitor addresses.
- `siblingPlug_`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: Stored in the Plug's configuration. This is the sibling plug on the sibling chain.
- `inboundSwitchboard_`
  - **Control**: Fully controlled.
  - **Constraints**: Must have been registered previously using `registerSwitchboardForSibling()`.
  - **Impact**: This is the switchboard used for inbound packets.
- `outboundSwitchboard_`
  - **Control**: Fully controlled.
  - **Constraints**: Must have been registered previously using `registerSwitchboardForSibling()`.
  - **Impact**: This is the switchboard used for outbound packets.

Branches and code coverage (including function calls)

Intended branches

- The plug (`msg.sender`) is configured correctly after execution.
  - ☐ Test coverage

Negative behavior

- Should revert if the inbound and outbound switchboards are not registered.
  - ☐ Negative test

Function: `registerSwitchboardForSibling(uint32 siblingChainSlug_, uint256 maxPacketLength_, uint256 capacitorType_, address siblingSwitchboard_)`

This function deploys a capacitor and a decapacitor using the `CapacitorFactory` for a switchboard. The `msg.sender` is the switchboard.

Inputs

- `siblingChainSlug_`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: Capacitors and decapacitors are mapped to the switchboard address and this sibling chain slug.
- `maxPacketLength_`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the maximum packet length of the deployed capacitor and decapacitor.
- `capacitorType_`
  - **Control**: Fully controlled.
  - **Constraints**: Must be one of the valid capacitor types.
  - **Impact**: This is the type of capacitor and decapacitor that will be deployed.
- `siblingSwitchboard_`
  - **Control**: Fully controlled.
  - **Constraints**: None.
  - **Impact**: The switchboard address deployed on `siblingChainSlug_`. This value is emitted in the `SiblingSwitchboardUpdated` event.

Branches and code coverage (including function calls)

Intended branches

- Should deploy the capacitor and decapacitor correctly.
  - ☐ Test coverage
- Should emit the events correctly.
  - ☐ Test coverage

Negative behavior

- Should revert if the switchboard already exists and has capacitors and

decapacitors deployed for it.
  ☑ Negative test

5.14 Module: SocketDst.sol

Function: `execute(ISocket.ExecutionDetails executionDetails_, ISocket.MessageDetails messageDetails_)`

This function executes a message after it has been authenticated by a switchboard.

Inputs

- `executionDetails_`
  - **Control**: Fully controlled.
  - **Constraints**: Too many constraints to list here. See constraints in `_execute()` and `_verify()` below.
  - **Impact**: This structure contains details regarding the execution of this message.
- `messageDetails_`
  - **Control**: Fully controlled.
  - **Constraints**: Too many constraints to list here. See constraints in `_execute()` and `_verify()` below.
  - **Impact**: This structure contains details regarding the message that is to be executed.

Branches and code coverage (including function calls)

Intended branches

- Packet execution is handled successfully with valid message and execution details and without any `msg.value`.
  ☑ Test coverage
- Packet execution is handled successfully with valid message and execution details and with `msg.value`.
  ☑ Test coverage

Negative behavior

- Should revert if called by an invalid executor.
  ☑ Negative test
- Should revert if the message has already been executed.
  ☑ Negative test

- Should revert if the packet ID is 0.
  - ☐ Negative test
- Should revert if the packet and message are not from the same chain.
  - ☑ Negative test
- Should revert if the execution gas limit is less than the minimum gas limit required by the message.
  - ☐ Negative test
- Should revert if the packet has not been proposed.
  - ☐ Negative test

Function: `proposeForSwitchboard(byte[32] packetId_, byte[32] root_, address switchboard_, byte[] signature_)`

This function is used by transmitters to propose packet root hashes to specific switchboards.

Inputs

- `packetId_`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: The ID of the packet.
- `root_`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the root hash of the packet being proposed.
- `switchboard_`
  - **Control**: Fully controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the switchboard that the packet is being proposed for.
- `signature_`
  - **Control**: Fully controlled.
  - **Constraints**: Must recover to an address with the transmitter role.
  - **Impact**: Used to verify that a transmitter has signed the arguments to this function and for this socket on this chain.

Branches and code coverage (including function calls)

Intended branches

- Packet proposal works correctly.

☑ Test coverage

Negative behavior

- Should revert if `packetId_` is 0.
  ☑ Negative test
- Should revert if the signature validation fails.
  ☑ Negative test

Function: `_execute(address executor_, address localPlug_, uint32 remoteChainSlug_, uint256 executionGasLimit_, ISocket.MessageDetails messageDetails_)`

This function calls the Plug's `inbound()` function with the message payloads, which will effectively execute the message. It then updates the executor's fees.

Inputs

- `executor_`
  - **Control**: Not controlled.
  - **Constraints**: This must be the executor's address retrieved from the signature.
  - **Impact**: This is the executor's address.
- `localPlug_`
  - **Control**: Fully controlled.
  - **Constraints**: Must be a valid and configured local plug.
  - **Impact**: This is the plug that the message payload is sent to for execution.
- `remoteChainSlug_`
  - **Control**: Not controlled.
  - **Constraints**: This must be the chain slug of the chain the message came from.
  - **Impact**: This is the chain slug of the source chain.
- `executionGasLimit_`
  - **Control**: Fully controlled.
  - **Constraints**: Must be set to enough gas to execute `inbound()` on the plug.
  - **Impact**: This is the gas limit needed to execute the Plug's `inbound()` function.
- `messageDetails_`
  - **Control**: Fully controlled.
  - **Constraints**: Too many constraints to list here. See constraints above and in `_verify()`.

- **Impact**: This structure contains details about the message sent by the user for execution.

Branches and code coverage (including function calls)

Intended branches

- Execution completes successfully with valid arguments.
  ☑ Test coverage

Function: `_verify(byte[32] packetId_, uint256 proposalCount_, uint32 remoteChainSlug_, byte[32] packedMessage_, byte[32] packetRoot_, PlugConfig plugConfig_, byte[] decapacitorProof_, byte[32] executionParams_)`

This function verifies the message that is being executed. It checks that the switchboard allows it, that the decapacitor acknowledges that this message should be included in this packet, and that the correct amount of `msg.value` is passed in.

Inputs

- `packetId_`
  - **Control**: Fully controlled.
  - **Constraints**: Must be a valid packet ID for a proposed packet.
  - **Impact**: Used for verification.
- `proposalCount_`
  - **Control**: Fully controlled.
  - **Constraints**: Must be a valid proposal count for a proposed packet.
  - **Impact**: Used for verification.
- `remoteChainSlug_`
  - **Control**: Fully controlled.
  - **Constraints**: Must be the chain slug for the source chain.
  - **Impact**: The chain slug for the remote chain that the packet is coming from.
- `packedMessage_`
  - **Control**: Partially controlled.
  - **Constraints**: Must be a valid message to pass verification.
  - **Impact**: This is the message that is being verified.
- `packetRoot_`
  - **Control**: Not controlled.
  - **Constraints**: N/A.

  - **Impact**: This is the packet that the message is being verified against.
- `plugConfig_`
  - **Control**: Not controlled.
  - **Constraints**: N/A.
  - **Impact**: This is the configuration of the plug that this message is intended for.
- `decapacitorProof_`
  - **Control**: Fully controlled.
  - **Constraints**: Must be a valid proof to pass verification.
  - **Impact**: This is the proof for the inclusion of the message within this packet.
- `executionParams_`
  - **Control**: Fully controlled.
  - **Constraints**: Must match the execution parameters from the original message to pass verification.
  - **Impact**: This contains extra parameters for message execution.

Branches and code coverage (including function calls)

Intended branches

- Execution completes successfully with valid arguments.
  ☑ Test coverage

Negative behavior

- Should revert if the message does not pass switchboard verification.
  ☐ Negative test
- Should revert if the message does not pass decapacitor verification.
  ☐ Negative test

5.15 Module: SocketSrc.sol

Function: `getMinFees(uint256 minMsgGasLimit_, uint256 payloadSize_, byte[32] executionParams_, byte[32] transmissionParams_, uint32 siblingChainSlug_, address plug_)`

Calculates the minimum amount of fees required to pass into `outbound` to send a message.

Inputs

- `minMsgGasLimit_`
  - **Control**: Full control.
  - **Constraints**: None.
  - **Impact**: The minimum amount of gas that the message must be executed with on the destination chain.
- `payloadSize_`
  - **Control**: Full control.
  - **Constraints**: Must be <= 3,000.
  - **Impact**: The size of the payload to calculate fees for.
- `executionParams_`
  - **Control**: Full control.
  - **Constraints**: None.
  - **Impact**: Used for calculating the transmission fees.
- `transmissionParams_`
  - **Control**: Full control.
  - **Constraints**: Must be valid `extraParams`.
  - **Impact**: Passed to the execution manager for use when calculating fees.
- `siblingChainSlug_`
  - **Control**: Full control.
  - **Constraints**: Must be a key in `_plugConfigs[plug_]`.
  - **Impact**: Used for determining which capacitor to use.
- `plug_`
  - **Control**: Full control.
  - **Constraints**: Must be a key in `_plugConfigs`.
  - **Impact**: Used for determining which capacitor to use.

Branches and code coverage

Intended branches

- Correctly returns minimum fees.
  ☑ Test coverage

Negative behavior

- Invalid inputs cause a reversion.
  ☐ Negative test
- Invalid `plug_` or `siblingChain_` (pointing to a capacitor that is not configured).
  ☐ Negative test

Function: `outbound(uint32 siblingChainSlug_, uint256 minMsgGasLimit_, byte[32] executionParams_, byte[32] transmissionParams_, byte[] payload_)`

Sends a cross-chain message. Fees are inputted via `msg.value`.

Inputs

- `siblingChainSlug_`
  - **Control**: Full control.
  - **Constraints**: Must be configured in `_plugConfigs[msg.sender]`.
  - **Impact**: The destination chain to send to. It is also used to look up the destination plug (node on the receiving chain) that determines where the message will go.
- `minMsgGasLimit_`
  - **Control**: Full control.
  - **Constraints**: None.
  - **Impact**: Setting for the minimum gas limit for the receiving chain. This is used to determine the fees that must be inputted to the function.
- `executionParams_`
  - **Control**: Full control.
  - **Constraints**: Must be formatted as valid `extraParams` per the ExecutionManager's `verifyParams` function.
  - **Impact**: Execution parameters bundled with the request.
- `transmissionParams_`
  - **Control**: Full control.
  - **Constraints**: None.
  - **Impact**: N/A.
- `payload_`
  - **Control**: Full.
  - **Constraints**: Must be <= 3,000 bytes long.
  - **Impact**: The payload to send cross-chain.

Branches and code coverage

Intended branches

- The `MessageOutbound` event emitted, indicating successful sending of the message.
  ☑ Test coverage

Negative behavior

- Revert if no sibling plug is found for the given chain.
  - ☐ Negative test
- Revert if `msg.value` is greater than the maximum `uint128` value.
  - ☐ Negative test
- Revert if `msg.value` is not greater than or equal to the minimum required fees.
  - ☐ Negative test

6 Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped Socket Data Layer contracts, we discovered eight findings. No critical issues were found. Two were of medium impact, two were of low impact, and the remaining findings were informational in nature. Socket Technology acknowledged all findings and implemented fixes for some.

6.1 Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.