# ZetaChain, Bitcoin Inscriptions

## Security Assessment (Summary Report)

**January 13, 2025**

*Prepared for:*
**Lucas Bertrand**
ZetaChain

*Prepared by:* **Simone Monica and Jaime Iglesias**

# Table of Contents

# Project Summary

## Contact Information

The following project manager was associated with this project:

**Jeff Braswell**, Project Manager
jeff.braswell@trailofbits.com

The following engineering director was associated with this project:

**Josselin Feist**, Engineering Director, Blockchain
josselin.feist@trailofbits.com

The following consultants were associated with this project:

**Simone Monica**, Consultant
simone.monica@trailofbits.com

**Jaime Iglesias**, Consultant
jaime.iglesias@trailofbits.com

## Project Timeline

The significant events and milestones of the project are listed below.

| Date | Event |
|---|---|
| **October 15, 2024** | Pre-project kickoff call |
| **October 22, 2024** | Delivery of report draft |
| **October 22, 2024** | Report readout meeting |
| **November 5, 2024, January 2–3, 2025** | Completion of fix review |
| **January 13, 2025** | Delivery of final summary report |

# Project Targets

The engagement involved a review and testing of the following target.

**Node**

| | |
|---|---|
| Repository | https://github.com/zeta-chain/node |
| Version | `09e8c3a120c1e1de53e3bbf00d1c028f0a45c99d` |
| Type | Go |
| Platform | Cosmos |

# Executive Summary

## Engagement Overview

ZetaChain engaged Trail of Bits to review the security of a number of pull requests (PRs) that introduce support for Bitcoin deposits on ZetaChain using inscriptions: PR #2727 (replaced by PR #2957), PR #2524, and PR #2533.

A team of two consultants conducted the review from October 15 to October 21, 2024, for a total of two engineer-weeks of effort. With full access to source code and documentation, we performed static and dynamic testing of the target, using automated and manual processes.

Our testing efforts focused on making sure the inscription support was correctly implemented (e.g., memos are decoded correctly from the inscriptions) and that its addition did not introduce unwanted behavior or affect the legacy `OP_RETURN` support. We also looked for any opportunities to carry out DoS attacks such as by creating transactions with a huge number of inputs or outputs.

Finally, we created a fuzzing harness for the `DecodeScript` function, which is tasked with getting the transaction memo out of the inscription, to find any panics in the function that could result in a node crash.

Overall, we found the code in scope to be relatively easy to understand and well structured.

## Observations and Impact

During the engagement, we identified three findings—two of low/informational severity and one whose severity is undetermined. Two of the issues are related to possible edge cases (TOB-ZETA-2, TOB-ZETA-3) and one is related to discrepancies between the legacy `OP_RETURN` implementation and the new inscription functionality (TOB-ZETA-1).

## Recommendations

We recommend that the ZetaChain team address the issues presented in this report.

# Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

| ID | Title | Type | Severity |
|---|---|---|---|
| 1 | Donations are not possible when using a witness | Undefined Behavior | Low |
| 2 | Ability to create fake donations | Data Validation | Informational |
| 3 | tryExtractInscription iterates over all the inputs | Data Validation | Undetermined |

# Detailed Findings

| 1. Donations are not possible when using a witness | |
|---|---|
| Severity: **Low** | Difficulty: **Low** |
| Type: Undefined Behavior | Finding ID: TOB-ZETA-1 |
| Target: `zetaclient/chains/bitcoin/tx_script.go` | |

### Description

The new mode to execute cross-chain deposits from Bitcoin to ZetaChain through the use of a witness does not check whether the message sent is a donation. Therefore, it is not possible to execute a donation.

Figure 1 shows that the `DecodeOpReturnMemo` function (the old mode) checks whether the `memoBytes` value corresponds to the donation message, and in that case, it returns as if there were no deposit.

```
func DecodeOpReturnMemo(scriptHex string, txid string) ([]byte, bool, error) {
		...
		memoBytes, err := hex.DecodeString(scriptHex[4:])
		if err != nil {
			return nil, false, errors.Wrapf(err, "error hex decoding memo:
%s", scriptHex)
		}
		if bytes.Equal(memoBytes, []byte(constant.DonationMessage)) {
			return nil, false, fmt.Errorf("donation tx: %s", txid)
		}
		return memoBytes, true, nil
	}


	return nil, false, nil
}
```

*Figure 1.1: Snippet of the `DecodeOpReturnMemo` function (`zetaclient/chains/bitcoin/tx_script.go#L172–L193`)*

However, figure 2 shows that the `DecodeScript` function (the new mode), which is used to decode the tapscript in the witness, does not check whether the `memoBytes` value corresponds to the donation message.

```
func DecodeScript(script []byte) ([]byte, bool, error) {
```

```
    ...
    memoBytes, err := decodeInscriptionPayload(&t)
    if err != nil {
        return nil, false, errors.Wrap(err, "decodeInscriptionPayload: unable
to decode the payload")
    }


    return memoBytes, true, nil
}
```

*Figure 1.2: Snippet of the `DecodeScript` function*
*(`zetaclient/chains/bitcoin/tx_script.go#L210–L223`)*

This creates a discrepancy between the implementations.

### Exploit Scenario

Alice sends a donation. The `memoBytes` value contains the donation message but not valid calldata. When ZetaChain tries to execute the call, the transaction reverts and the funds are refunded to Alice.

### Recommendations

Short term, have the `DecodeScript` function check whether the decoded `memoBytes` value corresponds to the donation message and, in that case, return as if there were no deposit.

Long term, when a new mode is added to an existing feature, make sure that the same functionality is supported in both modes; otherwise, make it clear (through documentation) that the discrepancy is intentional. For example, in this case, the donation message is not specific to the legacy mode and therefore should be supported by the new mode as well.

| 2. Ability to create fake donations | |
|---|---|
| Severity: **Informational** | Difficulty: **Low** |
| Type: Data Validation | Finding ID: TOB-ZETA-2 |
| Target: `zetaclient/chains/bitcoin/observer/witness.go` | |

### Description

It is currently possible to create a fake donation by constructing a transaction with an `OP_RETURN` output containing the donation message and a witness input containing a deposit message. Such a transaction would cause a donation message to be logged.

As shown in figure 2.1, the `GetBtcEventWithWitness` function will try to parse the memo by first trying to extract it from an `OP_RETURN` output through a call to `tryExtractOpRet`; otherwise, it will try to get the memo from the inscription in the witness field. However, there is an edge case in this implementation: if a donation message is used in an `OP_RETURN` output, then the call to `tryExtractOpRet` will return `nil`, leading to the execution of the inscription case; however, the call to `tryExtractOpRet` will log the donation, but the function will continue as if a normal deposit message were found inside the witness.

This basically allows the creation of fake donations, as the `OP_RETURN` output will be ignored while the witness will be processed.

```
func GetBtcEventWithWitness(
	client interfaces.BTCRPCClient,
	tx btcjson.TxRawResult,
	tssAddress string,
	blockNumber uint64,
	logger zerolog.Logger,
	netParams *chaincfg.Params,
	depositorFee float64,
) (*BTCInboundEvent, error) {
	...
	var memo []byte
	if candidate := tryExtractOpRet(tx, logger); candidate != nil {
		memo = candidate
		logger.Debug().
			Msgf("GetBtcEventWithWitness: found OP_RETURN memo %s in tx %s",
hex.EncodeToString(memo), tx.Txid)
	} else if candidate = tryExtractInscription(tx, logger); candidate != nil {
		memo = candidate
```

```
            logger.Debug().Msgf("GetBtcEventWithWitness: found inscription memo %s
in tx %s", hex.EncodeToString(memo), tx.Txid)
        } else {
            return nil, nil
        }
        ...
```

*Figure 2.1: Snippet of the `GetBtcEventWithWitness` function (`zetaclient/chains/bitcoin/observer/witness.go#L52–L62`)*

This issue is of informational severity because, currently, no action is taken after the donation message is logged (figure 2.2); however, future code changes could implement an automatic action when a donation message is detected and could have a higher impact.

```
func tryExtractOpRet(tx btcjson.TxRawResult, logger zerolog.Logger) []byte {
       ...

        memo, found, err := bitcoin.DecodeOpReturnMemo(tx.Vout[1].ScriptPubKey.Hex,
tx.Txid)
        if err != nil {
            logger.Error().Err(err).Msgf("tryExtractOpRet: error decoding OP_RETURN
memo: %s", tx.Vout[1].ScriptPubKey.Hex)
            return nil
        }

        ...
}
```

*Figure 2.2: Snippet of the `tryExtractOpRet` function (`zetaclient/chains/bitcoin/observer/witness.go#L127–L143`)*

### Exploit Scenario

Eve sends a transaction with an input containing the actual message encoded as a tapscript in the witness field and two outputs, the first of which locks the amount deposited with a P2WPKH script unlockable by the `tssAddress`, and the second of which is an `OP_RETURN` output with a donation message. The system logs this transaction as a donation, but it is executed as a normal deposit.

### Recommendations

Short term, consider having the system always treat the transaction as a donation if a donation message is present and a standard deposit is attached.

Long term, when adding a new mode to make cross-chain deposits, consider whether undefined behavior could be possible when the new mode is used with another mode.

| 3. tryExtractInscription iterates over all the inputs | |
|---|---|
| Severity: **Undetermined** | Difficulty: **Low** |
| Type: Data Validation | Finding ID: TOB-ZETA-3 |
| Target: `zetaclient/chains/bitcoin/observer/witness.go` | |

## Description

The `tryExtractInscription` function (figure 3.1) tries to extract the inscription encoded as a tapscript by iterating over all the inputs until it finds a valid inscription.

For every input, the `DecodeScript` function tries to decode a script following the expected rules by parsing each opcode.

```
func tryExtractInscription(tx btcjson.TxRawResult, logger zerolog.Logger) []byte {
	for i, input := range tx.Vin {
		script := ParseScriptFromWitness(input.Witness, logger)
		if script == nil {
			continue
		}

		logger.Debug().Msgf("potential witness script, tx %s, input idx %d",
tx.Txid, i)

		memo, found, err := bitcoin.DecodeScript(script)
		if err != nil || !found {
			logger.Debug().Msgf("invalid witness script, tx %s, input idx
%d", tx.Txid, i)
			continue
		}

		logger.Debug().Msgf("found memo in inscription, tx %s, input idx %d",
tx.Txid, i)
		return memo
	}

	return nil
}
```

*Figure 3.1: The `tryExtractInscription` function*
*(`zetaclient/chains/bitcoin/observer/witness.go#L146–L166`)*

A transaction can have hundreds or thousands of inputs which, if well crafted, may make the `tryExtractInscription` function take a long time to execute. Note, however, that an attacker submitting such a transaction would have to pay a lot in fees.

We did not have time to fully investigate whether this issue can be exploited, so the severity of this issue is undetermined.

### Exploit Scenario

Eve sends a transaction with thousands of inputs, all of which have a tapscript in the witness field; however, she crafts the scripts in such a way that all but one end with an opcode that the ZetaChain node would consider invalid or incorrect. This forces the `DecodeScript` function inside `tryExtractInscription` to parse all of them while only a single one is valid.

### Recommendations

Short term, consider imposing a limit on the number of inputs that can be parsed. For example, in the old mode (i.e., the `tryExtractOpRet` function), it is assumed that the `OP_RETURN` output will be in the second position.

Long term, in the code that parses data from users, consider implementing code that is the most defensive possible and reduces the attack surface for an attacker (e.g., by imposing limits on the inputs).

# A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

| Vulnerability Categories | |
|---|---|
| **Category** | **Description** |
| **Access Controls** | Insufficient authorization or assessment of rights |
| **Auditing and Logging** | Insufficient auditing of actions or logging of problems |
| **Authentication** | Improper identification of users |
| **Configuration** | Misconfigured servers, devices, or software components |
| **Cryptography** | A breach of system confidentiality or integrity |
| **Data Exposure** | Exposure of sensitive information |
| **Data Validation** | Improper reliance on the structure or values of data |
| **Denial of Service** | A system failure with an availability impact |
| **Error Reporting** | Insecure or insufficient reporting of error conditions |
| **Patching** | Use of an outdated software package or library |
| **Session Management** | Improper identification of authenticated users |
| **Testing** | Insufficient test methodology or test coverage |
| **Timing** | Race conditions or other order-of-operations flaws |
| **Undefined Behavior** | Undefined behavior triggered within the system |

| Severity Levels | |
|---|---|
| **Severity** | **Description** |
| Informational | The issue does not pose an immediate risk but is relevant to security best practices. |
| Undetermined | The extent of the risk was not determined during this engagement. |
| Low | The risk is small or is not one the client has indicated is important. |
| Medium | User information is at risk; exploitation could pose reputational, legal, or moderate financial risks. |
| High | The flaw could affect numerous users and have serious reputational, legal, or financial implications. |

| Difficulty Levels | |
|---|---|
| **Difficulty** | **Description** |
| Undetermined | The difficulty of exploitation was not determined during this engagement. |
| Low | The flaw is well known; public tools for its exploitation exist or can be scripted. |
| Medium | An attacker must write an exploit or will need in-depth knowledge of the system. |
| High | An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue. |

# B. Non-Security-Related Recommendations

The following recommendations are not associated with specific vulnerabilities. However, implementing them may enhance code readability and prevent the introduction of vulnerabilities in the future.

- **Update the comment for the `FilterAndParseIncomingTx` function to describe the new witness mode.**

```
// FilterAndParseIncomingTx given txs list returned by the "getblock 2" RPC command,
return the txs that are relevant to us
// relevant tx must have the following vouts as the first two vouts:
// vout0: p2wpkh to the TSS address (targetAddress)
// vout1: OP_RETURN memo, base64 encoded
func FilterAndParseIncomingTx(
	rpcClient interfaces.BTCRPCClient,
	txs []btcjson.TxRawResult,
	blockNumber uint64,
	tssAddress string,
	logger zerolog.Logger,
	netParams *chaincfg.Params,
) ([]*BTCInboundEvent, error) {
```

*Figure B.1: Comment for the `FilterAndParseIncomingTx` function (`zetaclient/chains/bitcoin/observer/inbound.go#L348–L359`)*

# C. Fuzzing Harness for the DecodeScript Function

We developed a fuzzing harness for the `DecodeScript` function to run with the native Go fuzzer, particularly to find any possible panics that would make the node crash.

The harness is provided in figure C.1. Note that we used the `f.Add()` function to add the current test data as seeds to improve the fuzzer's coverage; however, for readability purposes, we did not include all the data in this figure. It can be added in the `tx_script_test.go` file, and to start fuzzing, the following command should be run from the `node\zetaclient\chains\bitcoin` directory:

```
go test -fuzz=FuzzDecodeScript
```

```go
func FuzzDecodeScript(f *testing.F) {
	f.Add([]byte("2001a7bae79bd61c2368fe41a565061d6cf22b4f5...68"))
	f.Add([]byte("20d6f59371037bf30115d9fd6016...68"))
	f.Add([]byte("20cabd6ecc0245c40f27ca6299dcd3...68"))
	f.Add([]byte("2001a7bae79bd61c2368fe41a565...d0"))
	f.Add([]byte("2001a7bae79bd61c236...06d98b8fd0c7ab"))


	f.Fuzz(func(t *testing.T, in []byte) {
		bitcoin.DecodeScript(in)
	})
}
```

*Figure C.1: Fuzz testing of the `DecodeScript` function*

# D. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

On November 5, 2024, and from January 2 to January 3, 2025, Trail of Bits reviewed the fixes and mitigations implemented by the ZetaChain team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the issues described in this report, ZetaChain has resolved two issues and has not resolved the remaining issue. For additional information, please see the Detailed Fix Review Results below. Additionally, one issue was identified after the original audit that could have made the Bitcoin inscription parsing panic due to a null dereference. It was fixed in PR #3155.

| ID | Title | Status |
|---|---|---|
| 1 | Donations are not possible when using a witness | Resolved |
| 2 | Ability to create fake donations | Resolved |
| 3 | tryExtractInscription iterates over all the inputs | Unresolved |

## Detailed Fix Review Results

**TOB-ZETA-1: Donations are not possible when using a witness**
Resolved in PR #2654. The `AuthorizationList` is now validated in the `GenesisState` type's `Validate` function.

**TOB-ZETA-2: Ability to create fake donations**
Resolved in PR #2674. The new logic allows voters to vote on older keygen ballots or ballots already finalized and disallows ballots from changing their status in these cases. This allows voters to get observer rewards for voting on a ballot.

**TOB-ZETA-3: tryExtractInscription iterates over all the inputs**
Unresolved. The ZetaChain team gave the following response:

> *The ZetaChain team tested this and found zetaclient is able to process thousands of inputs extremely quickly. It would be extremely expensive to attempt an attack here and would barely slow down zetaclient.*

# E. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

| Fix Status | |
|---|---|
| **Status** | **Description** |
| **Undetermined** | The status of the issue was not determined during this engagement. |
| **Unresolved** | The issue persists and has not been resolved. |
| **Partially Resolved** | The issue persists but has been partially resolved. |
| **Resolved** | The issue has been sufficiently resolved. |

# About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

**Trail of Bits, Inc.**
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

# Notices and Remarks

## Copyright and Distribution



The sole canonical source for Trail of Bits publications is the Trail of Bits Publications page. Reports accessed through sources other than that page may have been modified and should not be considered authentic.

## Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.