ZetaChain Solana Gateway

Security Assessment (Summary Report)

January 28, 2025

Prepared for:
Charlie McCowan
ZetaChain

Prepared by: **Samuel Moelius**

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 consultant was associated with this project:

Samuel Moelius, Consultant
samuel.moelius@trailofbits.com

Project Timeline

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

Date	Event
November 14, 2024	Pre-project kickoff call
November 25, 2024	Delivery of report draft
November 25, 2024	Report readout meeting
January 17, 2025	Added appendix D: Fix Review Results
January 24, 2025	Delivery of final summary report
January 28, 2025	Updated appendix D per ZetaChain's request

Project Targets

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

Solana Protocol Contracts

Repository	https://github.com/zeta-chain/protocol-contracts-solana
Version	`2c186833030720cde8f01761dda6245b93504a5f`
Type	Rust/Anchor
Platform	Solana

Executive Summary

Engagement Overview

ZetaChain engaged Trail of Bits to review the security of its Solana gateway (`2c18683`). The gateway provides Solana programs access to ZetaChain and vice versa. The gateway also holds assets that have been deposited from the Solana blockchain onto ZetaChain.

One consultant conducted the review from November 18 to November 22, 2024, for a total of one engineer-week of effort. With full access to source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes.

Observations and Impact

The goals of the engagement were to answer questions such as the following, which were raised during the kickoff call:

- Is it possible to steal all of the gateway's tokens?
- Can the gateway be used to mint assets on ZetaChain?
- Can a deposit or withdrawal be attributed to the wrong person?
- Can one create a fake transaction and have it mistakenly observed by the gateway?
- Is it possible to perform a deposit and have it counted more than once, or to withdraw the same funds more than once?

We identified one high-severity issue that could allow an attacker to drain the rent account (TOB-ZETASOLANA-1). The rent account is used to reimburse signers for rent needed to create associated token accounts. However, the recipient for which the account is created can close the account and claim the rent. To the best of our knowledge, nothing stops a recipient from doing this repeatedly.

Additionally, the code does not seem to apply a formalized hashing strategy (TOB-ZETASOLANA-2). For example, for two instructions that employ hashing, a nonce is the third element of a message that is hashed. For another instruction, the nonce is the fourth element. Such inconsistencies increase the likelihood of the wrong data being hashed.

Finally, the code would benefit from adhering to a style guide, possibly one developed by ZetaChain. Right now, adherence to a style guide is not apparent. For example, the position of `Programs` within structs that derive the `Accounts` trait varies. Ensuring that `Programs` take the same position within all such structs would make it easier to determine the programs with which an instruction interacts. Additional style recommendations appear in appendix C.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Solana Labs take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or any refactor that may occur when addressing other recommendations.
- **Prepare a formalized hashing strategy and apply it in each of ZetaChain's gateways.** Doing so will reduce the likelihood that signatures prepared by other protocols, or by ZetaChain on other blockchains, could be reused with ZetaChain's Solana gateway.
- **Adhere to a style guide and apply it throughout the codebase.** Doing so will make it easier to review and spot bugs in code.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	Arithmetic does not feature prominently in the codebase.	**Satisfactory**
Auditing	Of the gateway's 13 instructions, only five emit log messages. Some of the log messages that are emitted are of questionable utility.	**Moderate**
Authentication / Access Controls	We found no problems related to authentication or access controls.	**Satisfactory**
Complexity Management	The code would benefit from adhering to a style guide, possibly one developed by ZetaChain. At present, adherence to a style guide is not apparent.	**Moderate**
Cryptography and Key Management	There is no specification describing how messages to be hashed are constructed. Fields are not arranged in a consistent order. Variable-length fields are not delimited, increasing the likelihood of a collision. The `chain_id` field, which should reduce the likelihood of a cross-chain collision, appears after a variable-length field, inhibiting `chain_id`'s usefulness.	**Weak**
Decentralization	The contracts have a designated authority with privileges beyond those of a normal user. It is unclear how the participants in the TSS are determined.	**Further Investigation Required**
Documentation	The README is adequate, but inline comments are lacking. The code features exactly three lines of doc comments.	**Moderate**
Low-Level Manipulation	Low-level manipulation does not feature prominently in the codebase.	**Satisfactory**

Testing and Verification	Some important conditions are not tested. Some tests employ a pattern that could allow them to pass when they should not.	**Moderate**
Transaction Ordering	We found no problems related to transaction ordering.	**Satisfactory**

Summary of Findings

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

ID	Title	Type	Severity
1	Rent payer account can be drained	Undefined Behavior	**High**
2	Hash collision risks	Cryptography	**Undetermined**
3	update_authority does not use a two-step transfer process	Data Validation	**Informational**
4	Requirement that recipients be System-owned is unjustified	Undefined Behavior	**Informational**
5	Receivers lack null address checks	Data Validation	**Informational**
6	Ineffective use of log messages	Auditing and Logging	**Informational**
7	Bump seeds not stored in PDAs	Denial of Service	**Informational**
8	Untested code	Testing	**Informational**
9	Tests may not fail as intended	Testing	**Informational**

Detailed Findings

1. Rent payer account can be drained	
Severity: **High**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-ZETASOLANA-1
Target: `withdraw_spl_token`	

Description

The associated token account to which the `withdraw_spl_token` instruction must transfer funds might not exist when the instruction is called. In such a case, `withdraw_spl_token` creates the associated token account (figure 1.1). Furthermore, `withdraw_spl_token` reimburses the signer for the lamports (i.e., rent) used to create the account (figure 1.2). However, the funds recipient can close the account and reclaim the rent. The recipient can do this repeatedly to siphon funds from ZetaChain.

```
invoke(
    &create_associated_token_account(
        ctx.accounts.signer.to_account_info().key,
        ctx.accounts.recipient.to_account_info().key,
        ctx.accounts.mint_account.to_account_info().key,
        ctx.accounts.token_program.key,
    ),
    &[
        ctx.accounts.mint_account.to_account_info().clone(),
        ctx.accounts.recipient_ata.clone(),
        ctx.accounts.recipient.to_account_info().clone(),
        ctx.accounts.signer.to_account_info().clone(),
        ctx.accounts.system_program.to_account_info().clone(),
        ctx.accounts.token_program.to_account_info().clone(),
        ctx.accounts
            .associated_token_program
            .to_account_info()
            .clone(),
    ],
)?;
```

Figure 1.1: Cross-program invocation of the token program to create a new associated token account
(`protocol-contracts-solana/programs/protocol-contracts-solana/src/lib.rs#383–402`)

```
411    let rent_payer_info = ctx.accounts.rent_payer_pda.to_account_info();
412    let cost = bal_before - bal_after;
413    rent_payer_info.sub_lamports(cost)?;
414    signer_info.add_lamports(cost)?;
```

Figure 1.2: Code to reimburse the signer from the rent-payer account (`protocol-contracts-solana/programs/protocol-contracts-solana/src/lib.rs#411–414`)

We developed a proof of concept by modifying the "`withdraw SPL token to a non-existent account...`" test in `protocol-contracts-solana.ts`. The crucial steps are shown in figure 2.3.

```
// Burn `to`'s token balance so `to` can be closed.
{
    let tx = new anchor.web3.Transaction().add(
        spl.createBurnInstruction(
            to,
            mint.publicKey,
            wallet2.publicKey,
            500_000
        )
    );
    await anchor.web3.sendAndConfirmTransaction(conn, tx, [wallet2]);
}

// Transfer `to`'s lamports to `wallet2` (the attacker's wallet).
{
    let tx = new anchor.web3.Transaction().add(
        spl.createCloseAccountInstruction(
            to,
            wallet2.publicKey,
            wallet2.publicKey
        )
    );
    await anchor.web3.sendAndConfirmTransaction(conn, tx, [wallet2]);
}
```

Figure 2.3: Crucial steps for claiming the associated token account's rent

In our experiments, the rent claimed in figure 2.3 was 2,039,280 lamports. As of this writing, that is slightly less than 1 USD. If an attacker could perform this attack once per block, at 0.4 seconds per block, the attacker would siphon about 216,000 USD per day.

Exploit Scenario

Mallory performs the attack described above and siphons funds from ZetaChain.

Recommendations

Short term, adopt a “pull” rather than a “push” withdrawal model. That is, rather than create associate token accounts for withdrawal recipients, require the recipients to create the accounts and to claim the withdrawals themselves. Note that this model would likely not allow withdrawals to be processed in strictly increasing order as the program currently requires. Thus, the program will likely have to be modified to require a nonce to be within a range, as opposed to requiring a nonce to have a specific value.

Long term, investigate the feasibility of using an intrusion detection system such as Range. If funds are siphoned from ZetaChain, it will help to know as soon as possible.

2. Hash collision risks

Severity: **Undetermined**	Difficulty: **Undetermined**
Type: Cryptography	Finding ID: TOB-ZETASOLANA-2
Target: `withdraw`, `withdraw_spl_token`, `validate_whitelist_tss_signature`	

Description

The instructions that perform hashing (`withdraw`, `withdraw_spl_token`, and `validate_whitelist_tss_signature`) do not follow an apparent specification. By not following a specification, the instructions increase the risk of a collision with a hash prepared by another protocol, or by ZetaChain on another chain.

Specific concerns include the following:

- None of the named functions precede the data they hash with a ZetaChain-specific identifier. This increases the risk of a hash prepared by another protocol colliding with one prepared by ZetaChain.
- Each of the functions hashes the `chain_id` field after the instruction name. Since the instruction name can vary in length, the `chain_id`'s position can vary within a message (yellow in figure 2.1). This reduces the likelihood that two different `chain_ids` will prevent a collision because they may reside at two different positions with a message.
- Variable length fields are not delimited. For example, in a message hashed by the `withdraw` instruction, the eight-byte `chain_id` field aligns with the characters `_spl_tok` in the instruction name in the message hashed by `withdraw_spl_token`.
- Data is not hashed in a consistent order. For example, for the `withdraw` and `withdraw_spl_token` instructions, the nonce is the third element of the message (red in figure 2.1). For the `validate_whitelist_tss_signature` instruction, the nonce is the fourth element of the message (red in figure 2.2). Such inconsistencies increase the risk of data being assembled in the wrong order before being hashed.

```rust
let mut concatenated_buffer = Vec::new();
concatenated_buffer.extend_from_slice("withdraw".as_bytes());
concatenated_buffer.extend_from_slice(&pda.chain_id.to_be_bytes());
concatenated_buffer.extend_from_slice(&nonce.to_be_bytes());
concatenated_buffer.extend_from_slice(&amount.to_be_bytes());
concatenated_buffer.extend_from_slice(&ctx.accounts.to.key().to_bytes());
```

Figure 2.1: Assembly of a message to be hashed in the `withdraw` instruction (`protocol-contracts-solana/programs/protocol-contracts-solana/src/lib.rs#285–290`)

```
474    let mut concatenated_buffer = Vec::new();
475    concatenated_buffer.extend_from_slice(instruction_name.as_bytes());
476    concatenated_buffer.extend_from_slice(&pda.chain_id.to_be_bytes());
477    concatenated_buffer.extend_from_slice(&whitelist_candidate.key()
           .to_bytes());
478    concatenated_buffer.extend_from_slice(&nonce.to_be_bytes());
```

Figure 2.2: Assembly of a message to be hashed in the `validate_whitelist_tss_signature` instruction (`protocol-contracts-solana/programs/protocol-contracts-solana/src/lib.rs#474–478`)

Note that we have given this finding a severity rating of undetermined because we cannot be certain that the cited hash collision risks are merely theoretical.

Exploit Scenario

Mallory notices that withdrawal messages for another ZetaChain-connected blockchain X have a similar structure to those used on Solana, but that the `chain_id` resides at a different offset. Alice is a ZetaChain user who has deposited funds from both X and Solana onto ZetaChain. Mallory tricks Alice into preparing a withdrawal with a certain structure for X. Mallory replays the signed message on Solana and steals Alice's funds on Solana.

Recommendations

Short term, prepare a formal cryptography specification and follow it on all blockchains where ZetaChain needs to perform hashing. The specification should include at least the following:

- A fixed-length prefix (e.g., the nine bytes `ZETACHAIN`) to begin each message
- A `chain_id` offset that is the same regardless of the chain for which a message is constructed
- A strategy for delimiting variable-length fields (e.g., instruction names)
    - The strategy should ensure that two adjacent variable length fields cannot coalesce into one another.
- A standard order for elements common to messages of a blockchain
    - For example, if all messages include a nonce, the nonce should consistently be the *i*th element for some *i*.

Preparing such a specification will reduce the likelihood of producing a hash that collides with one prepared by another protocol, or by ZetaChain on another blockchain.

Long term, following the development of the specification, seek a cryptographic review. Doing so will help ensure the specification does not contain flaws that render ZetaChain vulnerable.

3. update_authority does not use a two-step transfer process	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ZETASOLANA-3
Target: `update_authority`	

Description

The `update_authority` instruction performs a transfer of the authority role in one step (figure 3.1). The existing authority could accidentally transfer the role to a nonexistent address, making it impossible to transfer the role to another account.

```
pub fn update_authority(
    ctx: Context<UpdateAuthority>,
    new_authority_address: Pubkey,
) -> Result<()> {
    let pda = &mut ctx.accounts.pda;
    require!(
        ctx.accounts.signer.key() == pda.authority,
        Errors::SignerIsNotAuthority
    );
    pda.authority = new_authority_address;
    Ok(())
}
```

Figure 3.1: Implementation of the `update_authority` instruction (`protocol-contracts-solana/programs/protocol-contracts-solana/src/lib.rs#80–91`)

Exploit Scenario

Bob is the ZetaChain Solana gateway authority. Bob tries to transfer control of the gateway to Alice but mistypes her address. A new gateway must be deployed, and all code that uses the old gateway must be migrated to use the new one.

Recommendations

Short term, perform administrative transfers using a two-step process. That is, require the new authority to invoke an instruction to accept the role. This will reduce the likelihood that authoritative control over the gateway is unintentionally lost.

Long term, consider requiring a signature from the TSS to transfer the authority role. Various parts of the gateway suggest that the TSS and the authority have comparable privileges. Moreover, the authority must sign to change the address of the TSS (figure 3.2). Requiring the TSS to sign to change the authority's address might similarly make sense.

```
pub fn update_tss(ctx: Context<UpdateTss>, tss_address: [u8; 20]) ->
Result<()> {
    let pda = &mut ctx.accounts.pda;
    require!(
        ctx.accounts.signer.key() == pda.authority,
        Errors::SignerIsNotAuthority
    );
    pda.tss_address = tss_address;
    Ok(())
}
```

Figure 3.2: Implementation of the `update_tss` instruction (`protocol-contracts-solana/programs/protocol-contracts-solana/src/lib.rs#69–77`)

References

- Helius: A Hitchhiker's Guide to Solana Program Security

4. Requirement that recipients be System-owned is unjustified	
Severity: **Informational**	Difficulty: **Medium**
Type: Undefined Behavior	Finding ID: TOB-ZETASOLANA-4
Target: `withdraw_spl_token`	

Description

The `withdraw_spl_token` instruction requires that the recipient be owned by the `System` program (figure 4.1). There is no obvious reason for this requirement. Moreover, the requirement could limit the gateway's applicability.

```
550    #[derive(Accounts)]
551    pub struct WithdrawSPLToken<'info> {
           ...
563        pub recipient: SystemAccount<'info>,
           ...
574    }
```

Figure 4.1: Excerpt of `WithdrawSPLToken`'s definition (`protocol-contracts-solana/programs/protocol-contracts-solana/src/lib.rs#550–574`)

Note that the `withdraw` instruction used to have a similar requirement. However, the requirement was revoked by issue #28 and PR #37.[1]

[1] Thanks to ZetaChain for sharing this with us.

brewmaster012 commented on Sep 11 • edited Contributor

right now the withdraw context

```
#[derive(Accounts)]
pub struct Withdraw<'info> {
    #[account(mut)]
    pub signer: Signer<'info>,

    #[account(mut)]
    pub pda: Account<'info, Pda>,
    #[account(mut)]
    pub to: SystemAccount<'info>,
}
```

says the `to` address must be `SystemAccount` which precludes PDA.
`SystemAccount` is account that is owned by System (i.e. created by System program).
PDA accounts are created by other programs and owned by that program, so it's not SystemAccount.

Many programs use PDA to store SOL, so it would be important to support PDA here, in addition to SystemAccount.

Desired: change `SystemAccount` to accommodate PDA

Figure 4.2: Issue #28 on the `protocol-contracts-solana` repo

Exploit Scenario

Alice is a ZetaChain user. Alice writes a Solana program with the intent of having it transfer tokens to and from ZetaChain. Because Alice's program is not owned by the `System` program, Alice must build additional infrastructure to withdraw tokens to her program. The infrastructure requires considerable additional cost.

Recommendations

Short term, eliminate the requirement that SPL token recipients be `System`-owned. There is no obvious reason for the requirement. Moreover, eliminating the requirement will increase the gateway's applicability.

Long term, as new features are added to the protocol, consider whether the requirements they impose are justified. This will help ensure that the new features are usable by all who could benefit from them.

5. Receivers lack null address checks	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ZETASOLANA-5
Target: `deposit`, `deposit_and_call`, `deposit_spl_token`, `deposit_spl_token_and_call`	

Description

The `deposit`, `deposit_and_call`, `deposit_spl_token`, and `deposit_spl_token_and_call` functions allow funds to be moved onto ZetaChain for a recipient named in the instruction data. However, the instructions do not verify that the recipient is not null—that is, the zero address (figure 5.1). Since zero is a common default value, checks should be added to ensure that deposited tokens are not lost.

```
173    pub fn deposit(
174        ctx: Context<Deposit>,
175        amount: u64,
176        receiver: [u8; 20], // not used in this program; for directing zetachain protocol only
177    ) -> Result<()> {
```

Figure 5.1: Signature of the `deposit` function (`protocol-contracts-solana/programs/protocol-contracts-solana/src/lib.rs#173–177`)

Exploit Scenario

Alice is a ZetaChain user. Alice deposits funds onto ZetaChain, but an error causes the recipient address to be zeroed out. The funds are transferred successfully and thereby lost.

Recommendations

Short term, add code to `deposit`, `deposit_and_call`, `deposit_spl_token`, and `deposit_spl_token_and_call` to verify that the recipient is not null (i.e., the zero address). This will reduce the likelihood that funds are accidentally transferred to this address and thereby lost.

Long term, if new instructions are added to the gateway, and those new instructions require off-chain addresses, consider whether the addresses should be checked for null. Doing so will reduce the likelihood of the instructions being misused.

6. Ineffective use of log messages

Severity: **Informational**	Difficulty: **Low**
Type: Auditing and Logging	Finding ID: TOB-ZETASOLANA-6
Target: Various instructions	

Description

The gateway has 13 instructions. However, only five of them emit log messages. All instructions should emit log messages to indicate their success and any relevant details of their execution.

The following instructions do not emit any log messages:

- `initialize`
- `update_tss`
- `update_authority`
- `whitelist_spl_mint`
- `unwhitelist_spl_mint`
- `initialize_rent_payer`
- `deposit_and_call`
- `deposit_spl_token_and_call`

Also, some of the log messages that are emitted are of questionable utility (e.g., figure 6.1).

```
pub fn deposit_spl_token(
    ctx: Context<DepositSplToken>,
    amount: u64,
    receiver: [u8; 20], // unused in this program; for directing zetachain protocol only
) -> Result<()> {
    ...
    msg!("deposit spl token successfully");

    Ok(())
}
```

Figure 6.1: Log message emitted by the `deposit_spl_token` function (`protocol-contracts-solana/programs/protocol-contracts-solana/src/lib.rs#219–250`)

Exploit Scenario

Alice is a ZetaChain user. Alice submits an instruction to the gateway that completes successfully. However, Alice has difficulty determining the path that the instruction took. Alice's time would have been saved had the instruction emitted proper log messages.

Recommendations

Short term, ensure that each instruction emits at least one log message. The log message should include relevant details of the instruction's execution. This will help ensure that off-chain code can recover such details without having to simulate the instruction.

Long term, if new instructions are added to the gateway, ensure that they also adhere to this standard of emitting at least one log message.

7. Bump seeds not stored in PDAs	
Severity: **Informational**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-ZETASOLANA-7
Target: Various instructions	

Description

The gateway uses three types of PDAs (figure 7.1). Each use of one of those PDAs recomputes its bump seed. If ZetaChain is unlucky, computing one of the PDAs' bump seeds could take many iterations, making it cost prohibitive to call an instruction with the PDA.

```
654    #[account]
655    pub struct Pda {
656        nonce: u64,            // ensure that each signature can only be used once
657        tss_address: [u8; 20], // 20 bytes address format of ethereum
658        authority: Pubkey,
659        chain_id: u64,
660        deposit_paused: bool,
661    }
662
663    #[account]
664    pub struct WhitelistEntry {}
665
666    #[account]
667    pub struct RentPayerPda {}
```

Figure 7.1: The three types of PDAs used by the Solana gateway (`protocol-contracts-solana/programs/protocol-contracts-solana/src/lib.rs#654–667`)

Anchor's official documentation recommends storing a PDA's bump seed in the PDA. To give a specific example, a field `bump` could be added to the Pda struct, like in figure 7.2. Then, each use of Pda could be updated to use the `bump` field, like in figure 7.3.

```
#[account]
pub struct Pda {
    nonce: u64,            // ensure that each signature can only be used once
    tss_address: [u8; 20], // 20 bytes address format of ethereum
    authority: Pubkey,
    chain_id: u64,
    deposit_paused: bool,
```

```
    bump: u8,
}
```

Figure 7.2: Proposed modification of the Pda struct to store a bump seed

```
#[derive(Accounts)]
pub struct Deposit<'info> {
    #[account(mut)]
    pub signer: Signer<'info>,

    #[account(mut, seeds = [b"meta"], bump = pda.bump)]
    pub pda: Account<'info, Pda>,

    pub system_program: Program<'info, System>,
}
```

Figure 7.3: Proposed modification of the `Deposit` struct to use the Pda's bump seed (`protocol-contracts-solana/programs/protocol-contracts-solana/src/lib.rs#506–515`)

Exploit Scenario

An unlucky deployment causes the ZetaChain gateway to receive a program ID that, when hashed with "meta" and a bump seed, takes many iterations to find a bump seed that is not on the ED25519 curve. Each time the PDA is passed to an instruction, the steps to find the bump seed are repeated. The cost to use the ZetaChain Solana gateway is higher than necessary.

Recommendations

Short term, for the PDAs in figure 7.1, store their bump seeds in the PDA. This will reduce the cost of using the PDAs. Moreover, it will help protect against unlucky deployments where finding one of the PDAs' bump seeds requires many iterations.

Long term, when creating PDAs, keep the following in mind:

- Create PDAs only with canonical bump seeds. Creating PDAs with noncanonical bump seeds could allow an attacker to create multiple PDAs for the same set of non-bump seeds.
- Avoid the following situation:
    - Part of a PDA's seeds are user-controlled.
    - The PDA's bump seed is not stored (i.e., it is recomputed each time the PDA is used).
    - The user that causes a PDA to be created might not be the same user that must pass the PDA in an instruction.

In such a situation, Mallory could try to choose seeds for which it is expensive to compute the PDA's bump seed. If Mallory can then require Alice to pass the PDA in an instruction, Mallory can cause Alice to waste lamports computing the PDA's bump seed.

References

- The Anchor Book v0.29.0: PDAs
- Solana Beta: Necessary to pass bump for PDA in the instruction data when using Anchor?
- Solana Beta: Difference between bump vs account.bump in anchor

8. Untested code	
Severity: **Informational**	Difficulty: **High**
Type: Testing	Finding ID: TOB-ZETASOLANA-8
Target: `protocol-contracts-solana.ts`	

Description

Some parts of the gateway are untested. We recommend 100% test coverage for on-chain code to ensure that the code works as intended.

The following "mismatch nonce" code is untested:

- `programs/protocol-contracts-solana/src/lib.rs#L281–L284`
- `programs/protocol-contracts-solana/src/lib.rs#L469–L472`

The following "ECDSA signature error" code is untested:

- `programs/protocol-contracts-solana/src/lib.rs#L297–L300`
- `programs/protocol-contracts-solana/src/lib.rs#L343–L346`
- `programs/protocol-contracts-solana/src/lib.rs#L485–L488`

Exploit Scenario

Mallory uncovers a bug in a ZetaChain Solana gateway function. The bug likely would have been revealed by more thorough test coverage.

Recommendations

Short term, develop tests for each piece of code in the bulleted list above. Doing so will help ensure that the code works as intended.

Long term, regularly review tests to ensure that all important conditions are tested.

9. Tests may not fail as intended	
Severity: **Informational**	Difficulty: **High**
Type: Testing	Finding ID: TOB-ZETASOLANA-9
Target: `protocol-contracts-solana.ts`	

Description

Several tests employ the coding pattern in figure 9.1. The problem with this pattern is that the test passes if the call on line 420 succeeds. The test should be adjusted so that it passes only if the call fails.

```
414    it("unwhitelist SPL token and deposit should fail", async () => {
415        await gatewayProgram.methods.unwhitelistSplMint([], 0, [], new
anchor.BN(0)).accounts({
416            whitelistCandidate: mint.publicKey,
417        }).rpc();
418
419        try {
420            await depositSplTokens(gatewayProgram, conn, wallet, mint, address)
421        } catch (err) {
422            expect(err).to.be.instanceof(anchor.AnchorError);
423            expect(err.message).to.include("AccountNotInitialized");
424        }
425    });
```

Figure 9.1: Problematic testing pattern. If the call to `depositSplTokens` on line 420 succeeds, the test passes.
(`protocol-contracts-solana/tests/protocol-contracts-solana.ts#414–425`)

A better approach is to record the error in the `catch` block but check the error outside of the `catch` block (figure 9.2). In this way, the test fails if no error is caught.

```
it("unwhitelist SPL token and deposit should fail", async () => {
    await gatewayProgram.methods.unwhitelistSplMint([], 0, [], new
anchor.BN(0)).accounts({
        whitelistCandidate: mint.publicKey,
    }).rpc();

    let err: any = null;
    try {
        await depositSplTokens(gatewayProgram, conn, wallet, mint, address)
    } catch (caught) {
        err = caught;
    }
```

```
    expect(err).to.be.instanceof(anchor.AnchorError);
    expect(err.message).to.include("AccountNotInitialized");
});
```

Figure 9.2: Proposed rewrite of the test in figure 9.1. Note that the error is checked outside of the `catch` block.

Locations where the problematic pattern shown in figure 9.1 is used include the following:

- `tests/protocol-contracts-solana.ts#L272`
- `tests/protocol-contracts-solana.ts#L419`
- `tests/protocol-contracts-solana.ts#L461`
- `tests/protocol-contracts-solana.ts#L508`
- `tests/protocol-contracts-solana.ts#L526`
- `tests/protocol-contracts-solana.ts#L540`

Exploit Scenario

Alice, a ZetaChain developer, introduces a bug into `deposit_spl_token`. The function no longer performs appropriate checks and completes successfully when it should not. The tests pass and do not catch the bug.

Recommendations

Short term, address each of the problematic tests in the above bulleted list by rewriting them (e.g., as suggested in figure 9.2). Rewriting the tests in this way will ensure that they fail if the calls they test unintentionally succeed.

Long term, run Necessist on the code. The problems highlighted in this finding were found by running Necessist and reviewing the results.

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. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage
Transaction Ordering	The system's resistance to transaction-ordering attacks

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.

Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. 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.

- **Ensure that the declarations in figures C.1 through C.4 use the same name.** The fact that they do not appears to cause the `anchor test` to exit with the error in figure C.5, even though all of the tests pass.

```
7     [programs.localnet]
8     protocol_contracts_solana = "ZETAjseVjuFsxdRxo6MmTCvqFwb3ZHUx56Co3vCmGis"
```

Figure C.1: A mention of the project name (`protocol-contracts-solana/Anchor.toml#7–8`)

```
1     [package]
2     name = "protocol-contracts-solana"
```

Figure C.2: A second mention of the project name (`protocol-contracts-solana/programs/protocol-contracts-solana/Cargo.toml#1–2`)

```
7     [lib]
8     crate-type = ["cdylib", "lib"]
9     name = "protocol_contracts_solana"
```

Figure C.3: A third mention of the project name (`protocol-contracts-solana/programs/protocol-contracts-solana/Cargo.toml#7–9`)

```
37    #[program]
38    pub mod gateway {
```

Figure C.4: A fourth mention of the project name, which does not match the previous three (`protocol-contracts-solana/programs/protocol-contracts-solana/src/lib.rs#37–38`)

```
Error: No such file or directory (os error 2)
```

Figure C.5: Error that `anchor test` exits with even though all tests pass

- **Correct the typos in `README.md` shown in figure C.6.**

```
75    The Gateway program derive a PDA (Program Derived Address) with seeds
`b"meta"` and canonical bump.  This PDA account/address actually holds the SOL
balance of the Gateway program.  For SPL tokens, the program stores the SPL token in
```

```
PDA derived ATAs. For each SPL token (different mint account), the program creates
ATA from PDA and the Mint (standard way of deriving ATA in Solana SPL program).
```

Figure C.6: "derive" should be "derives"; "the program creates ATA" should be "the program creates an ATA"; and "from PDA and the Mint" should be "from the PDA and the Mint". (`protocol-contracts-solana/README.md#75`)

- **Use a `package.json` script rather than a `Makefile` file rule (figure C.7) to format Rust source files.**

```
fmt:
	@echo "$(C_GREEN)# Formatting rust code$(C_END)"
	@./scripts/fmt.sh
```

Figure C.7: `Makefile` rule used to format Rust source files (`protocol-contracts-solana/Makefile#8–10`)

- **Rely on the version of `rustfmt` installed by `rustup` rather than by `brew` (figure C.8).**

```
if ! command -v rustfmt &> /dev/null
then
    echo "rustfmt could not be found, installing..."
    brew install rustfmt
fi
```

Figure C.8: Excerpt of a shell script that installs `rustfmt` (`protocol-contracts-solana/scripts/fmt.sh#12–16`)

- **In figure C.9, change `lint` to `fmt` or something similar.** Use of the term `lint` is misleading.

```
"scripts": {
  "lint:fix": "prettier */*.js \"*/**/*{.js,.ts}\" -w",
  "lint": "prettier */*.js \"*/**/*{.js,.ts}\" --check"
},
```

Figure C.9: Misleading use of the term `lint` (`protocol-contracts-solana/package.json#10–13`)

- **Enable `noUnusedLocals` in the project's `tsconfig.json` file.** The project's test file has many unused locals. An example appears in figure C.10.

```
    const account = await spl.getAccount(conn, tokenAccount.address);

     // OK; transfer some USDC SPL token to the gateway PDA
     wallet_ata = await spl.getAssociatedTokenAddress(
         mint.publicKey,
         wallet.publicKey,
     );
```

```
187
188         // create a fake USDC token account
189         await mintSPLToken(conn, wallet, mint_fake);
190     })
```

Figure C.10: `account` is an example of a variable that is unused after it is declared. *(`protocol-contracts-solana/tests/protocol-contracts-solana.ts#180–190`)*

- **Remove the `anchor-syn` dependency from the gateway's `Cargo.toml` file (figure C.11).** The dependency is unused.

```
19  [dependencies]
20  anchor-lang = { version = "=0.30.0" }
21  anchor-spl = { version = "=0.30.0", features = ["idl-build"] }
22  anchor-syn = "=0.30.0"
23  spl-associated-token-account = "3.0.2"
24  solana-program = "=1.18.15"
```

Figure C.11: Unused dependency *(`protocol-contracts-solana/programs/protocol-contracts-solana/Cargo.toml#19–24`)*

- **The errors `InsufficientPoints` (figure C.12) and `MemoLengthTooShort` (figure C.13) are unused; either add comments explaining why they are unused or remove them.**

```
15  #[msg("InsufficientPoints")]
16  InsufficientPoints,
```

Figure C.12: An unused error *(`protocol-contracts-solana/programs/protocol-contracts-solana/src/lib.rs#15–16`)*

```
27  #[msg("MemoLengthTooShort")]
28  MemoLengthTooShort,
```

Figure C.13: Another unused error *(`protocol-contracts-solana/programs/protocol-contracts-solana/src/lib.rs#27–28`)*

- **Either wrap `[u8; 20]` in a new type or declare an alias to it, and use the new type or alias throughout the code.** The type `[u8; 20]` is used in many places. Giving it a name will make the code more readable.

```
41  pub fn initialize(
42      ctx: Context<Initialize>,
43      tss_address: [u8; 20],
44      chain_id: u64,
45  ) -> Result<()> {
```

Figure C.14: Example use of the `[u8; 20]` type (`protocol-contracts-solana/programs/protocol-contracts-solana/src/lib.rs#41–45`)

- **Adjust the code in figure C.15 like in figure C.16 so that it assigns all fields at once.** This will ensure that no fields are unassigned if they are later added to the struct.

```
47    initialized_pda.nonce = 0;
48    initialized_pda.tss_address = tss_address;
49    initialized_pda.authority = ctx.accounts.signer.key();
50    initialized_pda.chain_id = chain_id;
51    initialized_pda.deposit_paused = false;
```

Figure C.15: Excerpt of the `initialize` function (`protocol-contracts-solana/programs/protocol-contracts-solana/src/lib.rs#47–51`)

```
**initialized_pda = Pda {
    nonce: 0,
    tss_address: tss_address,
    authority: ctx.accounts.signer.key(),
    chain_id,
    deposit_paused: false,
};
```

Figure C.16: Proposed rewrite of the code in figure C.15

- **Remove the uses of `&mut` in figures C.17 and C.18, as they are unnecessary.**

```
140    let whitelist_candidate: &mut Account<'_, Mint> = &mut ctx.accounts.whitelist_candidate;
```

Figure C.17: An unnecessary use of `&mut` (`protocol-contracts-solana/programs/protocol-contracts-solana/src/lib.rs#140`)

```
460    fn validate_whitelist_tss_signature(
461        pda: &mut Account<Pda>,
462        whitelist_candidate: &mut Account<Mint>,
463        signature: [u8; 64],
464        recovery_id: u8,
465        message_hash: [u8; 32],
466        nonce: u64,
467        instruction_name: &str,
468    ) -> Result<()> {
```

Figure C.18: A second unnecessary use of `&mut` (`protocol-contracts-solana/programs/protocol-contracts-solana/src/lib.rs#460–468`)

- **Remove the use of `#[allow(unused)]` in figure C.19, as all arguments are used.**

```
257    #[allow(unused)]
258    pub fn deposit_spl_token_and_call(
259        ctx: Context<DepositSplToken>,
260        amount: u64,
261        receiver: [u8; 20],
262        message: Vec<u8>,
263    ) -> Result<()> {
264        require!(message.len() <= 512, Errors::MemoLengthExceeded);
265        deposit_spl_token(ctx, amount, receiver)?;
266        Ok(())
267    }
```

Figure C.19: An unnecessary use of `#[allow(unused)]`
(`protocol-contracts-solana/programs/protocol-contracts-solana/src/lib.rs#257–267`)

- **Consider eliminating `message_hash` as an argument of `whitelist_spl_mint`, `unwhitelist_spl_mint`, `withdraw`, and `withdraw_spl_token`.** (The case of `withdraw` appears in figure C.20.) For each of these functions, `message_hash` can be computed from the arguments and thus is redundant. If knowing `message_hash` is useful for debugging, consider emitting it in a log message.

```
271    pub fn withdraw(
272        ctx: Context<Withdraw>,
273        amount: u64,
274        signature: [u8; 64],
275        recovery_id: u8,
276        message_hash: [u8; 32],
277        nonce: u64,
278    ) -> Result<()> {
279        let pda = &mut ctx.accounts.pda;
280
281        if nonce != pda.nonce {
282            msg!("mismatch nonce");
283            return err!(Errors::NonceMismatch);
284        }
285        let mut concatenated_buffer = Vec::new();
           ...
291        require!(
292            message_hash == hash(&concatenated_buffer[..]).to_bytes(),
293            Errors::MessageHashMismatch
294        );
```

Figure C.20: Example use of `message_hash` as an instruction argument
(`protocol-contracts-solana/programs/protocol-contracts-solana/src/lib.rs#271–294`)

- **Remove the log message in figure C.21.** It is redundant since the same log message is emitted by `recover_eth_address` (figure C.22).

```
341    let address = recover_eth_address(&message_hash, recovery_id, &signature)?;
// ethereum address is the last 20 Bytes of the hashed pubkey
342    msg!("recovered address {:?}", address);
```

Figure C.21: Redundant log message
(protocol-contracts-solana/programs/protocol-contracts-solana/src/lib.rs#341–342)

```
441    fn recover_eth_address(
442        message_hash: &[u8; 32],
443        recovery_id: u8,
444        signature: &[u8; 64],
445    ) -> Result<[u8; 20]> {
           ...
452        msg!("recovered address {:?}", address);
           ...
457    }
```

Figure C.22: The log message highlighted in figure C.21 is emitted by `recover_eth_address`.
(protocol-contracts-solana/programs/protocol-contracts-solana/src/lib.rs#441–457)

- **Consistently put spaces around the = character when used after the seeds keyword.** Spaces are used in all places except those shown in figures C.23 through C.25.

```
525    #[account(seeds=[b"whitelist", mint_account.key().as_ref()], bump)]
526    pub whitelist_entry: Account<'info, WhitelistEntry>, // attach whitelist
entry to show the mint_account is whitelisted
```

Figure C.23: A use of seeds= without spaces
(protocol-contracts-solana/programs/protocol-contracts-solana/src/lib.rs#525–526)

```
602    #[account(
603        init,
604        space=8,
605        payer=authority,
606        seeds=[
607            b"whitelist",
608            whitelist_candidate.key().as_ref()
609        ],
610        bump
611    )]
612    pub whitelist_entry: Account<'info, WhitelistEntry>,
```

Figure C.24: A second use of `seeds=` without spaces (`protocol-contracts-solana/programs/protocol-contracts-solana/src/lib.rs#602–612`)

```
    #[account(
        mut,
        seeds=[
            b"whitelist",
            whitelist_candidate.key().as_ref()
        ],
        bump,
        close = authority,
    )]
    pub whitelist_entry: Account<'info, WhitelistEntry>,
```

Figure C.25: A third use of `seeds=` without spaces (`protocol-contracts-solana/programs/protocol-contracts-solana/src/lib.rs#625–634`)

- **Consistently put a blank line before `#[account]` annotations.** Presently, such annotations are sometimes preceded by a blank line and sometimes not (figure C.26).

```
    pub token_program: Program<'info, Token>,

    #[account(mut)]
    pub from: Account<'info, TokenAccount>, // this must be owned by signer; normally the ATA of signer
    #[account(mut)]
    pub to: Account<'info, TokenAccount>, // this must be ATA of PDA
```

Figure C.26: Inconsistent use of blank lines before `#[account]` annotations (`protocol-contracts-solana/programs/protocol-contracts-solana/src/lib.rs#530–535`)

- **Expand the use of doc comments.** Currently, there are exactly three lines of doc comments (figures C.27 and C.28).

```
    /// CHECK: to account is not read so no need to check its owners; the program neither knows nor cares who the owner is.
```

Figure C.27: One of three lines of doc comments (`protocol-contracts-solana/programs/protocol-contracts-solana/src/lib.rs#545`)

```
    /// CHECK: recipient_ata might not have been created; avoid checking its content.
    /// the validation will be done in the instruction processor.
```

Figure C.28: Two of three lines of doc comments (protocol-contracts-solana/programs/protocol-contracts-solana/src/lib.rs#564–565)

- **Change `AccountInfo` to `UncheckedAccount` in figure C.29.** `UncheckedAccount` is considered the more modern convention and should be preferred.

```
550    #[derive(Accounts)]
551    pub struct WithdrawSPLToken<'info> {
           ...
566        #[account(mut)]
567        pub recipient_ata: AccountInfo<'info>,
           ...
574    }
```

Figure C.29: `AccountInfo` that should be replaced with `UncheckedAccount` (protocol-contracts-solana/programs/protocol-contracts-solana/src/lib.rs#550–574)

- **Establish some consistency in the order in which accounts are passed.** Consider the Pda account as an example. For some instructions, it is the first account (figure C.30); for some, it is the second (figure C.31); and for others, it is the third (figure C.32).

```
576    #[derive(Accounts)]
577    pub struct UpdateTss<'info> {
578        #[account(mut, seeds = [b"meta"], bump)]
579        pub pda: Account<'info, Pda>,
580        #[account(mut)]
581        pub signer: Signer<'info>,
582    }
```

Figure C.30: Pda passed as the first account (protocol-contracts-solana/programs/protocol-contracts-solana/src/lib.rs#576–582)

```
495    #[derive(Accounts)]
496    pub struct Initialize<'info> {
497        #[account(mut)]
498        pub signer: Signer<'info>,
499
500        #[account(init, payer = signer, space = size_of::< Pda > () + 8, seeds = [b"meta"], bump)]
501        pub pda: Account<'info, Pda>,
502
503        pub system_program: Program<'info, System>,
504    }
```

Figure C.31: Pda passed as the second account
(protocol-contracts-solana/programs/protocol-contracts-solana/src/lib.rs#495–504)

```
#[derive(Accounts)]
pub struct Whitelist<'info> {
    #[account(
       init,
       space=8,
       payer=authority,
       seeds=[
           b"whitelist",
           whitelist_candidate.key().as_ref()
       ],
       bump
   )]
    pub whitelist_entry: Account<'info, WhitelistEntry>,
    pub whitelist_candidate: Account<'info, Mint>,

    #[account(mut, seeds = [b"meta"], bump)]
    pub pda: Account<'info, Pda>,
    #[account(mut)]
    pub authority: Signer<'info>,

    pub system_program: Program<'info, System>,
}
```

Figure C.32: Pda passed as the third account
(protocol-contracts-solana/programs/protocol-contracts-solana/src/lib.rs#600–621)

- **Remove `system_program` from `Unwhitelist` (figure C.33).** The program is not used by the `unwhitelist_spl_mint` instruction.

```
#[derive(Accounts)]
pub struct Unwhitelist<'info> {
    ...
    pub system_program: Program<'info, System>,
}
```

Figure C.33: Excerpt of the definition of `Unwhitelist`
(protocol-contracts-solana/programs/protocol-contracts-solana/src/lib.rs#623–643)

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 January 17, 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 nine issues described in this report, ZetaChain has resolved six issues and has not resolved two issues. One issue's status could not be determined because it involves changes to code outside of what we reviewed. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Rent payer account can be drained	High	Undetermined
2	Hash collision risks	Undetermined	Resolved
3	update_authority does not use a two-step transfer process	Informational	Unresolved
4	Requirement that recipients be System-owned is unjustified	Informational	Resolved
5	Receivers lack null address checks	Informational	Resolved
6	Ineffective use of log messages	Informational	Resolved
7	Bump seeds not stored in PDAs	Informational	Unresolved
8	Untested code	Informational	Resolved
9	Tests may not fail as intended	Informational	Resolved

Detailed Fix Review Results

TOB-ZETASOLANA-1: Rent payer account can be drained
Undetermined in PR #67. The rent payer account was eliminated, and reimbursements are now made from the Pda account. Furthermore, ZetaChain explained they have mitigated the attack by enforcing a minimum withdrawal fee to cover the rent. When a withdrawal is made to Solana, the withdrawer must pay the equivalent of the cost to create an associated token account upfront. In this way, repeatedly creating and closing an associated token account cannot be profitable.

Because aspects of the fix are in code outside of what was reviewed, we have labeled the fix status as undetermined.

TOB-ZETASOLANA-2: Hash collision risks
Resolved in PR #72. The PR includes the following changes:

- All hashed data now begins with the nine bytes `ZETACHAIN`.
- The use of variable-length fields was eliminated. In particular, instruction names were replaced with one-byte constants. As a result, all hashed fields are now of fixed length.
- The order of the nonce and whitelist candidate fields were swapped in the `validate_whitelist_tss_signature` function. As a result, nonces are now consistently hashed after the chain IDs.

We would like to emphasize the following:

- Chain IDs now appear at an offset of 10 bytes within the hashed data. To be effective, chain IDs should be at this offset in all hashed data on all ZetaChain-supported blockchains.
- We recommend seeking a cryptographic review of the current code.

TOB-ZETASOLANA-3: update_authority does not use a two-step transfer process
Unresolved. ZetaChain provided the following context for this finding's fix status:

> *Will be fixed in future version. Tracked as Issue #3324*

TOB-ZETASOLANA-4: Requirement that recipients be System-owned is unjustified
Resolved in PR #71. Recipients are no longer `SystemAccounts`, which require them to be `System` owned. Recipients are now `UncheckedAccounts`.

TOB-ZETASOLANA-5: Receivers lack null address checks
Resolved in PR #66. The following check was added to both the `deposit` and `deposit_spl_token` functions:

```
require!(receiver != [0u8; 20], Errors::EmptyReceiver);
```

Because the `deposit_and_call` and `deposit_spl_token_and_call` functions call these functions (respectively), they perform the check as well.

TOB-ZETASOLANA-6: Ineffective use of log messages
Resolved in PR #71. The finding named eight instructions that do not emit log messages. One of those instructions (`initialize_rent_payer`) was eliminated as part of the fix for TOB-ZETASOLANA-1. The remaining seven instructions now emit log messages with relevant data.

TOB-ZETASOLANA-7: Bump seeds not stored in PDAs
Unresolved. ZetaChain provided the following context for this finding's fix status:

> *Will be fixed in future version. Tracked as Issue #3324*

TOB-ZETASOLANA-8: Untested code
Resolved in PR #71 and PR #73. A test was added for each line range named in TOB-ZETASOLANA-8. All of the line ranges are now tested.

TOB-ZETASOLANA-9: Tests may not fail as intended
Resolved in PR #71 and PR #73. Each of the locations mentioned in TOB-ZETASOLANA-9 was adjusted so that an error is thrown if the call in the `try` block succeeds.

Fixes for some other findings (e.g., TOB-ZETASOLANA-8) introduced additional calls in `try` blocks that should fail. Those calls were similarly augmented so that an error is thrown if the call succeeds.

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

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.