hexens x Everclear

Security Review Report for Everclear

December 2025

Table of Contents

1. About Hexens

Hexens is a pioneering cybersecurity firm dedicated to establishing robust security standards for Web3 infrastructure, driving secure mass adoption through innovative protection technology and frameworks. As an industry elite experts in blockchain security, we deliver comprehensive audit solutions across specialized domains, including infrastructure security, Zero Knowledge Proof, novel cryptography, DeFi protocols, and NFTs.

Our methodology combines industry-standard security practices combined with unique methodology of two teams per audit, continuously advancing the field of Web3 security. This innovative approach has earned us recognition from industry leaders.

Since our founding in 2021, we have built an exceptional portfolio of enterprise clients, including major blockchain ecosystems and Web3 platforms.

2. Executive Summary

This report covers the security review for Everclear. Everclear is a cross-chain intent protocol facilitating permissionless near-instant swaps across blockchains. This audit covered the Solana smart contracts.

Our security assessment was a full review of the code, spanning a total of 1 week.

During our review, we did identified 1 Critical severity vulnerability, which could have resulted in permanently frozen user assets.

We also identified several minor severity vulnerabilities and code optimisations.

All of our reported issues were fixed by the development team and consequently validated by us.

We can confidently say that the overall security and code quality have increased after completion of our audit.

3. Security Review Details

▪ Review Led by

Kasper Zwijsen, Head of Audits

▪ Scope

The analyzed resources are located on:

🔗 https://github.com/everclearorg/monorepo/tree/91ac02d711245802c990a2c8517dc622e69b7a30/packages/contracts/solana-spoke

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

🔗 https://github.com/everclearorg/monorepo/tree/7ed53e8c639a31ed6aee7b9074e42a45c25aa5a7

▪ Changelog

1 December 2025	Audit start
10 December 2025	Initial report
5 January 2026	Revision received
7 January 2026	Final report

4. Severity Structure

The vulnerability severity is calculated based on two components:

1. Impact of the vulnerability
2. Probability of the vulnerability

Impact	Probability			
	Rare	Unlikely	Likely	Very likely
Low	Low	Low	Medium	Medium
Medium	Low	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

- **Severity Characteristics**

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities that are highly likely to be exploited and can lead to catastrophic outcomes, such as total loss of protocol funds, unauthorized governance control, or permanent disruption of contract functionality.

High

Vulnerabilities that are likely to be exploited and can cause significant financial losses or severe operational disruptions, such as partial fund theft or temporary asset freezing.

Medium	Vulnerabilities that may be exploited under specific conditions and result in moderate harm, such as operational disruptions or limited financial impact without direct profit to the attacker.
Low	Vulnerabilities with low exploitation likelihood or minimal impact, affecting usability or efficiency but posing no significant security risk.
Informational	Issues that do not pose an immediate security risk but are relevant to best practices, code quality, or potential optimizations.

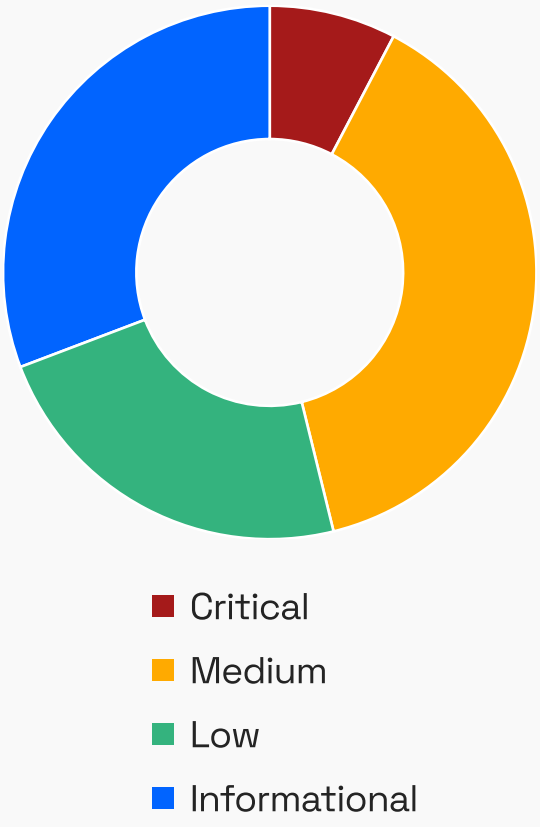
- **Issue Symbolic Codes**

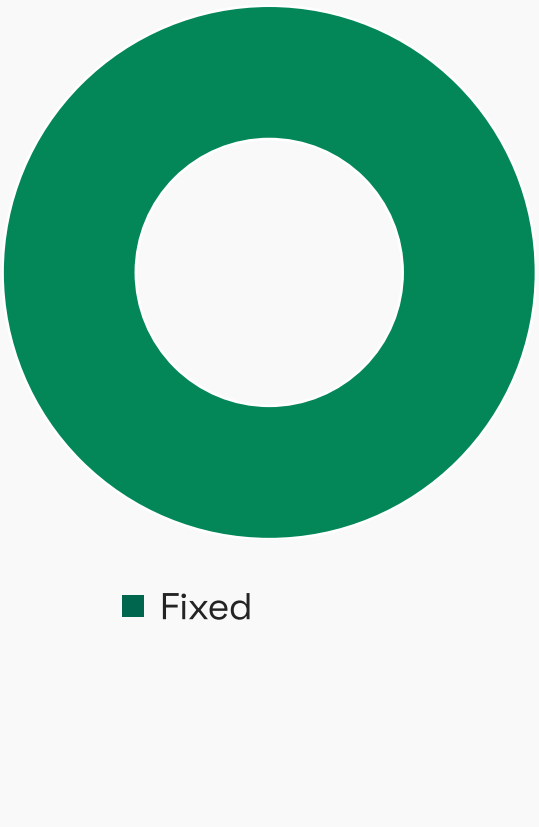
Each identified and validated issue is assigned a unique symbolic code during the security research stage.

Due to the structure of the vulnerability reporting flow, some rejected issues may be missing.

5. Findings Summary

Severity	Number of findings
Critical	1
High	0
Medium	5
Low	3
Informational	4
Total:	**13**





6. Weaknesses

This section contains the list of discovered weaknesses.

EVER1-6 | Attacker can block account creation causing user funds to become locked

Fixed

Severity:	Probability:	Impact:
Critical	Very likely	Critical

Path:

packages/contracts/solana-spoke/programs/everclear_spoke/src/instructions/intent/fill_intent.rs::handle_fill_intent#L134-L327

Description:

In the handle_fill_intent function, a previously created intent is fulfilled. The intent was created by locking the creator's assets.

In the function, it uses the create_account system instruction to create the intent_status_pda. This reverts if the account already exists.

If an attacker predicts the address that will be created be pre-calculating it, and pre-initializes it by sending dust to that address, then the transaction will fail and cannot be successfully executed in the future.

As a result, the user's funds become permanently locked.

```
// try to create intent status pda. Same logic as in mark_settlement_as_delivered
let data = IntentStatusAccount::try_deserialize(&mut &intent_status_pda.data.borrow()[..]);
if data.is_err() {
  // TODO: we create the same size intent status account as in settlement here for simplicity.
  // We can probably optimize this to only create 9 bytes status account here; need to think about
  // security implications tho.
  let space = 8
     + std::mem::size_of::<IntentStatusAccount>()
     + 12 * std::mem::size_of::<SerializableAccountMeta>();

  let __anchor_rent = Rent::get()?;
  let lamports = __anchor_rent.minimum_balance(space);
  let inst = anchor_lang::solana_program::system_instruction::create_account(
```

```
    &accounts.pda_payer.key(),
    &intent_status_pda.key(),
    lamports,
    space as u64,
    &program_id,
);

let payer_seed = &[
    "everclear_spoke".as_bytes(),
    "-".as_bytes(),
    "pda_payer".as_bytes(),
];
let (_payer_pda, payer_pda_bump) = Pubkey::find_program_address(payer_seed, &program_id);

msg!("{:?}", inst);
msg!(
    "{:?}",
    Pubkey::create_program_address(
        &[b"everclear_spoke", b"-", b"pda_payer", &[payer_pda_bump]],
        &program_id
    )
);

invoke_signed(
    &inst,
    &[
        accounts.pda_payer.clone(),
        intent_status_pda.to_account_info(),
    ],
    &[
        &[b"everclear_spoke", b"-", b"pda_payer", &[payer_pda_bump]],
        intent_status_pda_seeds!(intent_id, intent_status_bump),
    ],
)?;
```

Remediation:

We recommend to make the flow such that it cannot be DoSed by an attacker's actions.

Replace the manual **system_instruction::create_account** call with Anchor's init constraint for the **intent_status_pda**. Anchor's init logic automatically handles pre-funded accounts by using a combination of transfer, allocate, and assign instructions, which is immune to this DoS attack.

Alternatively, if manual initialization is required, check if the account already has lamports and use **system_instruction::allocate** and **system_instruction::assign** instead of **create_account** to safely claim ownership of the pre-funded account.

EVER1-3 | Fee signature not bound to intent data allows signature reuse

Fixed

Severity:	Probability:	Impact:
Medium	Unlikely	Medium

Path:

programs/everclear_spoke/src/instructions/intent/new_intent.rs

Description:

In **programs/everclear_spoke/src/instructions/fee_adapter/fees.rs,** the fee signature verification logic only covers **FeeData** (fees, asset, deadline) and omits critical intent parameters and unique nonces. Unlike the Solidity **FeeAdapterV2**, where signatures bind strictly to the full intent payload and are single-use, the Solana implementation allows a single valid signature to be replayed for any intent.

This discrepancy breaks the intended fee-gating model, allowing an attacker to reuse a single signature (potentially one with low or zero fees) to repeatedly call **new_intent** with arbitrary intent details. While this does not directly allow theft of user funds - since the caller must still provide the principal assets - it undermines the protocol's access control, enables fee evasion, and exposes the system to spam or denial-of-service (DoS) vectors via queue congestion. Additionally, the current logic bypasses verification entirely when the contract is paused, further weakening the security model.

```
pub fn new_intent(
    ctx: Context<NewIntent>,
    receiver: Pubkey,
    output_asset: Pubkey,
    amount: u64,
    amount_out_min: u128,
    ttl: u64,
    destinations: Vec<u32>,
    data: Vec<u8>,
    message_gas_limit: u64,
    fee_param: FeeParams,
) -> Result<()> {
    ...
    let fee_data = FeeData {
        token_fee: fee_param.token_fee,
        native_fee: fee_param.native_fee,
```

```
        input_asset: ctx.accounts.mint.key(),
        deadline: fee_param.deadline,
    };
    let fee_accounts = HandleFeeAccounts {
        signature_accounts: SignatureAccounts {
            signer: ctx.accounts.fee_signer.to_account_info(),
            instruction_sysvar: ctx.accounts.instruction_sysvar.to_account_info(),
        },
        user_account: ctx.accounts.authority.to_account_info(),
        user_token_account: ctx.accounts.user_token_account.to_account_info(),
        user_authority_account: ctx.accounts.authority.to_account_info(),
        fee_reciever_account: ctx.accounts.fee_recipient.to_account_info(),
        fee_reciever_token_account: ctx.accounts.fee_recipient_token_account.to_account_info(),
        token_program: ctx.accounts.token_program.to_account_info(),
        system_program: ctx.accounts.system_program.to_account_info(),
    };

    if !ctx.accounts.fee_adapter_state.paused {
        handle_fees(fee_data, fee_param.signature, fee_accounts)?;
    }
```

Remediation:

Bind the fee signature to the intent (e.g., include an intent hash or all intent fields in the signed data) and add replay protection (nonce or used-hash tracking). Align pause handling so fee validation is not bypassed - either revert during pause or enforce a strict allowlist.

EVER1-5 | Settle delivered intent can be executed while paused

Fixed

Severity:	Probability:	Impact:
Medium	Likely	High

Path:

packages/contracts/solana-spoke/programs/everclear_spoke/src/instructions/receive_message/settle.rs::settle_delivered_intent#L21-L145

Description:

The function **settle_delivered_intent** does not check the spoke state's paused state in **spoke_state.paused**, so it can be executed even when **paused** is **true**.

This is in contradiction with other instructions that do revert whenever the state is paused.

```
pub fn settle_delivered_intent(
    ctx: Context<SettleDeliveredIntentContext>,
    _ix: SettleDeliveredIntentInstruction,
) -> Result<()> {
    // assert settlement exists and the status is delivered
    require!(
        ctx.accounts.intent_status_pda.settlement.is_some()
            && ctx.accounts.intent_status_pda.status == IntentStatus::Delivered,
        SpokeError::InvalidIntentStatus
    );
    // verify all account here matches the one in the intent status pda (except the event authority as they have seperate checks)
    require!(
        ctx.accounts.spoke_state.key() == ctx.accounts.intent_status_pda.accounts[0].pubkey,
        SpokeError::IncorrectSettlementAccounts
    );
    require!(
        ctx.accounts.intent_status_pda.key() == ctx.accounts.intent_status_pda.accounts[1].pubkey,
        SpokeError::IncorrectSettlementAccounts
    );
    require!(
        ctx.accounts.vault_authority.key() == ctx.accounts.intent_status_pda.accounts[2].pubkey,
        SpokeError::IncorrectSettlementAccounts
    );
    require!(
        ctx.accounts.token_program.key() == ctx.accounts.intent_status_pda.accounts[3].pubkey,
```

```
        SpokeError::IncorrectSettlementAccounts
    );
    require!(
        ctx.accounts.system_program.key() == ctx.accounts.intent_status_pda.accounts[4].pubkey,
        SpokeError::IncorrectSettlementAccounts
    );
    require!(
        ctx.accounts.mint_account.key() == ctx.accounts.intent_status_pda.accounts[5].pubkey,
        SpokeError::IncorrectSettlementAccounts
    );
    require!(
        ctx.accounts.associated_token_program.key()
            == ctx.accounts.intent_status_pda.accounts[6].pubkey,
        SpokeError::IncorrectSettlementAccounts
    );
    require!(
        ctx.accounts.recipient.key() == ctx.accounts.intent_status_pda.accounts[7].pubkey,
        SpokeError::IncorrectSettlementAccounts
    );
    require!(
        ctx.accounts.recipient_token_account.key()
            == ctx.accounts.intent_status_pda.accounts[8].pubkey,
        SpokeError::IncorrectSettlementAccounts
    );
    require!(
        ctx.accounts.vault_token_account.key() == ctx.accounts.intent_status_pda.accounts[9].pubkey,
        SpokeError::IncorrectSettlementAccounts
    );

    // SAFE: settlement existence is checked
    let settlement = ctx.accounts.intent_status_pda.settlement.clone().unwrap();

    // 2) Mark as settled in storage
    ctx.accounts.intent_status_pda.status = IntentStatus::Settled;

    let mut buf = [0u8; 32];
    settlement.amount.to_little_endian(&mut buf);
    let normalized_amount = u128::from_be_bytes(buf[16..32].try_into().unwrap());

    // 3) Normalise the settlement amount
    let minted_decimals = ctx.accounts.mint_account.decimals;
    let amount = normalize_decimals(
        normalized_amount,
```

```
    DEFAULT_NORMALIZED_DECIMALS,
    minted_decimals,
)?;

require!(amount < u64::MAX.into(), SpokeError::InvalidAmount);

if amount == 0 {
    return Ok(());
}

// Create ATA idempotently
let create_idempotent_inst = create_associated_token_account_idempotent(
    ctx.accounts.authority.key,
    ctx.accounts.recipient.key,
    &ctx.accounts.mint_account.key(),
    ctx.accounts.token_program.key,
);
msg!("{:?}", create_idempotent_inst);
anchor_lang::solana_program::program::invoke(
    &create_idempotent_inst,
    &[
        ctx.accounts.authority.to_account_info(),
        ctx.accounts.recipient_token_account.to_account_info(),
        ctx.accounts.recipient.to_account_info(),
        ctx.accounts.mint_account.to_account_info(),
        ctx.accounts.system_program.to_account_info(),
        ctx.accounts.token_program.to_account_info(),
    ],
)?;

let signer_seeds: &[&[u8]] =
    vault_authority_pda_seeds!(ctx.accounts.spoke_state.vault_authority_bump);
let signer = &[signer_seeds];

let cpi_accounts = anchor_spl::token::Transfer {
    from: ctx.accounts.vault_token_account.to_account_info(),
    to: ctx.accounts.recipient_token_account.to_account_info(),
    authority: ctx.accounts.vault_authority.to_account_info(),
};
let cpi_ctx = CpiContext::new_with_signer(
    ctx.accounts.token_program.to_account_info(),
    cpi_accounts,
    signer,
```

```
    );

    // NOTE: Removed the virtual balance logic
    token::transfer(cpi_ctx, amount as u64)?;

    emit_cpi!(SettledEvent {
        intent_id: settlement.intent_id,
        recipient: settlement.recipient,
        asset: settlement.asset,
        amount: amount as u64,
        domain: ctx.accounts.spoke_state.domain,
    });
    Ok(())
}
```

Remediation:

To prevent usage while the contract is paused, we recommend adding a check like the following inside the function:

```
let state = &mut accounts.spoke_state;
require!(!state.paused, SpokeError::ContractPaused);
```

This ensures the function cannot be used when the contract is in a paused state.

EVER1-7 | Missing type hashes in signature digests

Fixed

Severity:	Probability:	Impact:
Medium	Rare	High

Path:

packages/contracts/solana-spoke/programs/everclear_spoke/src/instructions/fee_adapter/signature.rs::verify_signature#L21-L107

Description:

In the Solana spoke program there are multiple uses of ed25519 signatures by the special **fee_signer** account from the **fee_adapter_state** account, for example in the **new_intent** instruction the signature is verified against **FeeData**, containing the fixed token fee amounts.

However, none of the signature digests contain any domain-specific or function-specific type hashes. The signature digest is a simple Borsh serialization of the raw data structure:

```
pub fn verify_signature<T>(data: &T, signature: Vec<u8>, accounts: SignatureAccounts) -> Result<()>
where
    T: AnchorSerialize,
{
    let mut encoded_message = vec![];
    data.serialize(&mut encoded_message)?;

    [..]
}
```

Here the **data** is an arbitrary struct derived from **AnchorSerialize**.

This means that the data fields in the struct are simply packed together and used as input for the **ed25519_program**, where the signature will be checked against the digest.

This use of off-chain signatures is dangerous as there is no guarantee:

1. For which domain the signature was intended (domain type hash)
2. For which function signature was intended (functional type hash)

As a result, if the signing authority signs a message for a completely different protocol/purpose, then the signature can still be used as input for this program, as the digest is just raw bytes of data fields. Similarly, if there are multiple function taking signatures, there is also no discriminator to distinguish the different struct digests from each other.

In the EVM ecosystem, there is the widely adopted EIP-712, which introduces domain type hashes containing information such as address, chain ID, etc. to separate signatures from intended destination. On top of this, functional type hashes should also be added so that signatures for one function cannot be substituted to be used in another function.

For Solana, there is an official accepted proposal here: Off-chain message signing | Solana Validator

```
pub fn verify_signature<T>(data: &T, signature: Vec<u8>, accounts: SignatureAccounts) -> Result<()>
where
    T: AnchorSerialize,
{
    let mut encoded_message = vec![];
    data.serialize(&mut encoded_message)?;

    // NOTE: signature programs are native in nodes but they requires lamports to run, thus this have
    // to be done in preinstructions.

    let current_instruction_idx = load_current_index_checked(&accounts.instruction_sysvar)?;

    require!(
        current_instruction_idx != 0,
        SpokeError::MissingEd25519Instruction
    );

    // NOTE: this will not underflow as current_instruction_idx != 0
    let preinstruction_idx = current_instruction_idx - 1;

    let preinstruction =
        load_instruction_at_checked(preinstruction_idx.into(), &accounts.instruction_sysvar)?;

    // check size
    require!(
        preinstruction.program_id == ed25519_program::ID,
        SpokeError::MissingEd25519Instruction
    );
    require!(
        preinstruction.accounts.is_empty(),
        SpokeError::InvalidFeeSignature
    );
    // NOTE: the data struct: 16 byte header (with all the offsets), 32 bytes (pubkey), 64 bytes (signature), msg
    require!(
        preinstruction.data.len()
```

```
        == DATA_START
            + PUBKEY_SERIALIZED_SIZE
            + SIGNATURE_SERIALIZED_SIZE
            + encoded_message.len(),
        SpokeError::InvalidFeeSignature
    );

    // check data
    // Byte 0: num of signatures
    require!(preinstruction.data[0] == 1, SpokeError::InvalidFeeSignature);
    // Byte 1: padding byte
    require!(preinstruction.data[1] == 0, SpokeError::InvalidFeeSignature);
    // Byte 2-16: offsets

    let offsets = Ed25519SignatureOffsets {
        signature_offset: SIGNATURE_OFFSET as u16,
        signature_instruction_index: u16::MAX,
        public_key_offset: PUBLIC_KEY_OFFSET as u16,
        public_key_instruction_index: u16::MAX,
        message_data_offset: MESSAGE_DATA_OFFSET as u16,
        message_data_size: encoded_message.len() as u16,
        message_instruction_index: u16::MAX,
    };
    require!(
        preinstruction.data[SIGNATURE_OFFSETS_START..DATA_START] == *bytes_of(&offsets),
        SpokeError::InvalidFeeSignature
    );

    // signing key used in verify call
    require!(
        preinstruction.data[PUBLIC_KEY_OFFSET..PUBLIC_KEY_OFFSET + PUBKEY_SERIALIZED_SIZE]
            == *accounts.signer.key().as_array(),
        SpokeError::InvalidFeeSignature
    );

    // signature used in verify call
    require!(
        preinstruction.data[SIGNATURE_OFFSET..SIGNATURE_OFFSET + SIGNATURE_SERIALIZED_SIZE]
            == signature,
        SpokeError::InvalidFeeSignature
    );

    // message used in verify call
```

```
    require!(
        preinstruction.data[MESSAGE_DATA_OFFSET..MESSAGE_DATA_OFFSET + encoded_message.len()]
            == encoded_message,
        SpokeError::InvalidFeeSignature
    );

    // NOTE: if all preinstruction calldata is verify, the preinstruction must succeed, or else it will revert the whole
tx.
    Ok(())
}
```

Remediation:

We highly recommend implementing a form of domain separation and function separation, such as type hashes, in the signature digests.

EVER1-8 | Fee handling is skipped when fee adapter is paused

Fixed

Severity:	Probability:	Impact:
Medium	Unlikely	Medium

Path:

packages/contracts/solana-spoke/programs/everclear_spoke/src/instructions/intent/new_intent.rs::new_intent

Description:

The function **new_intent** is an entry point into the program and callable by a user. The function will create a new intent according to the parameters.

Inside of the function, it constructs a **FeeData** struct and passes this along with a signature to **handle_fees** to handle any required fees for the intent.

However, the call to the function is conditional on the paused state of the fee adapter:

```
if !ctx.accounts.fee_adapter_state.paused {
    handle_fees(fee_data, fee_param.signature, fee_accounts)?;
}
```

This means that if the fee adapter were to be paused, no fees would be charged at all since **handle_fees** performs the actual token transfers. In such a case, the user could create intents with arbitrarily high fee amounts.

This could lead to off-chain confusion about whether fees were actually paid for an intent whenever the fee adapter becomes unpaused. Furthermore, this is also contradictory to the other function **create_order**, which also creates intents but instead reverts when the fee adapter is paused.

```
pub fn new_intent(
    ctx: Context<NewIntent>,
    receiver: Pubkey,
    output_asset: Pubkey,
    amount: u64,
    amount_out_min: u128,
```

```
    ttl: u64,
    destinations: Vec<u32>,
    data: Vec<u8>,
    message_gas_limit: u64,
    fee_param: FeeParams,
) -> Result<()> {
    let mut accounts = NewIntentAccounts {
        spoke_state: ctx.accounts.spoke_state.clone().as_ref().clone(),
        mint: ctx.accounts.mint.clone(),
        token_program: ctx.accounts.token_program.clone(),
        program_vault_account: ctx.accounts.program_vault_account.clone(),
        user_token_account: ctx.accounts.user_token_account.clone(),
        authority: ctx.accounts.authority.clone(),
        system_program: ctx.accounts.system_program.clone(),
        spl_noop_program: ctx.accounts.spl_noop_program.clone(),
        hyperlane_mailbox: ctx.accounts.hyperlane_mailbox.clone(),
        mailbox_outbox: ctx.accounts.mailbox_outbox.clone(),
        dispatch_authority: ctx.accounts.dispatch_authority.clone(),
        unique_message_account: ctx.accounts.unique_message_account.clone(),
        dispatched_message_pda: ctx.accounts.dispatched_message_pda.clone(),
        igp_program: ctx.accounts.igp_program.clone(),
        igp_program_data: ctx.accounts.igp_program_data.clone(),
        igp_payment_pda: ctx.accounts.igp_payment_pda.clone(),
        configured_igp_account: ctx.accounts.configured_igp_account.clone(),
        inner_igp_account: ctx.accounts.inner_igp_account.clone(),
    };
    let program_id = *ctx.program_id;

    let fee_data = FeeData {
        token_fee: fee_param.token_fee,
        native_fee: fee_param.native_fee,
        input_asset: ctx.accounts.mint.key(),
        deadline: fee_param.deadline,
    };
    let fee_accounts = HandleFeeAccounts {
        signature_accounts: SignatureAccounts {
            signer: ctx.accounts.fee_signer.to_account_info(),
            instruction_sysvar: ctx.accounts.instruction_sysvar.to_account_info(),
        },
```

```
        user_account: ctx.accounts.authority.to_account_info(),
        user_token_account: ctx.accounts.user_token_account.to_account_info(),
        user_authority_account: ctx.accounts.authority.to_account_info(),
        fee_reciever_account: ctx.accounts.fee_recipient.to_account_info(),
        fee_reciever_token_account: ctx.accounts.fee_recipient_token_account.to_account_info(),
        token_program: ctx.accounts.token_program.to_account_info(),
        system_program: ctx.accounts.system_program.to_account_info(),
    };

    if !ctx.accounts.fee_adapter_state.paused {
        handle_fees(fee_data, fee_param.signature, fee_accounts)?;
    }

    let event = handle_new_intent(
        &mut accounts,
        program_id,
        receiver,
        output_asset,
        amount,
        amount_out_min,
        ttl,
        destinations,
        data,
        message_gas_limit,
    )
    .unwrap();

    emit_cpi!(event);

    Ok(())
}
```

Remediation:

Consider either reverting on a paused state in **new_intent** or allowing for 0 fee orders in **new_order**.

EVER1-12 | Batch intent creation in new order fails due to reused message accounts

Fixed

Severity:	Probability:	Impact:
Medium	Unlikely	Medium

Path:

packages/contracts/solana-spoke/programs/everclear_spoke/src/instructions/fee_adapter/new_order.rs::new_order#L12-L103

Description:

The **new_order** function is designed to create multiple intents in a single transaction by iterating over a vector of **OrderParameters** and calling **handle_new_intent** for each entry. However, this batch functionality does not work as intended.

Each call to **handle_new_intent** invokes Hyperlane's **outbox_dispatch** to send a cross-chain message. The Mailbox program requires a unique unique_message_account and its corresponding **dispatched_message_pda** for every dispatch call. The **dispatched_message_pda** is derived from the **unique_message_account** public key and must be uninitialized at the time of dispatch.

In the current implementation, the same **unique_message_account** and **dispatched_message_pda** from **ctx.accounts** are cloned into the **NewIntentAccounts** struct and reused across all loop iterations. When the first intent is processed, the Mailbox initializes the **dispatched_message_pda**. On subsequent iterations, the Mailbox's **verify_account_uninitialized** check fails because the PDA has already been created, causing the entire transaction to revert.

As a result, **new_order** cannot process more than one intent per transaction. While users can still create intents individually through the **new_intent** function, the batch feature remains non-functional.

```
pub fn new_order(
    ctx: Context<NewIntent>,
    params: Vec<OrderParameters>,
    fee_param: FeeParams,
) -> Result<()> {
    ...
    let mut accounts = NewIntentAccounts {
        spoke_state: ctx.accounts.spoke_state.as_ref().clone(),
        mint: ctx.accounts.mint.clone(),
```

```
    token_program: ctx.accounts.token_program.clone(),
    program_vault_account: ctx.accounts.program_vault_account.clone(),
    user_token_account: ctx.accounts.user_token_account.clone(),
    authority: ctx.accounts.authority.clone(),
    system_program: ctx.accounts.system_program.clone(),
    spl_noop_program: ctx.accounts.spl_noop_program.clone(),
    hyperlane_mailbox: ctx.accounts.hyperlane_mailbox.clone(),
    mailbox_outbox: ctx.accounts.mailbox_outbox.clone(),
    dispatch_authority: ctx.accounts.dispatch_authority.clone(),
    unique_message_account: ctx.accounts.unique_message_account.clone(),
    dispatched_message_pda: ctx.accounts.dispatched_message_pda.clone(),
    igp_program: ctx.accounts.igp_program.clone(),
    igp_program_data: ctx.accounts.igp_program_data.clone(),
    igp_payment_pda: ctx.accounts.igp_payment_pda.clone(),
    configured_igp_account: ctx.accounts.configured_igp_account.clone(),
    inner_igp_account: ctx.accounts.inner_igp_account.clone(),
};
...
for p in &params {
    let event_data = handle_new_intent(
        &mut accounts,
        program_id,
        p.receiver,
        p.output_asset,
        p.amount,
        p.amount_out_min,
        p.ttl,
        p.destinations.clone(),
        p.data.clone(),
        p.message_gas_limit,
    )?;

    emit_cpi!(event_data);

    intent_ids.push(event_data.intent_id);
}
...
```

```
fn outbox_dispatch(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    dispatch: OutboxDispatch,
) -> ProgramResult {
    ...
    // Account 5: Unique message account.
    // Uniqueness is enforced by making sure the message storage PDA based on
    // this unique message account is empty, which is done next.
    let unique_message_account_info = next_account_info(accounts_iter)?;
    if !unique_message_account_info.is_signer {
        return Err(ProgramError::MissingRequiredSignature);
    }

    // Account 6: Dispatched message PDA.
    let dispatched_message_account_info = next_account_info(accounts_iter)?;
    let (dispatched_message_key, dispatched_message_bump) = Pubkey::find_program_address(
        mailbox_dispatched_message_pda_seeds!(unique_message_account_info.key),
        program_id,
    );
    if dispatched_message_key != *dispatched_message_account_info.key {
        return Err(ProgramError::InvalidArgument);
    }
    // Make sure an account can't be written to that already exists.
    verify_account_uninitialized(dispatched_message_account_info)?;

    if accounts_iter.next().is_some() {
        return Err(ProgramError::from(Error::ExtraneousAccount));
    }
```

Remediation:

Require a distinct **unique_message_account** and derived **dispatched_message_pda** for each intent in the batch (e.g., pass per-intent accounts via remaining accounts or a structured array) and ensure the loop supplies a fresh pair to every **handle_new_intent** call.

EVER1-1 | Confusing endian conversion for settlement amount

Fixed

Severity:	Probability:	Impact:
Low	Rare	Medium

Path:

packages/contracts/solana-spoke/programs/everclear_spoke/src/instructions/receive_message/settle.rs::settle_delivered_intent#L21-L145

Description:

In the function **settle_delivered_intent** there is a conversion between byte formats on line 82:

```
let mut buf = [0u8; 32];
settlement.amount.to_little_endian(&mut buf);
let normalized_amount = u128::from_be_bytes(buf[16..32].try_into().unwrap());
let amount = normalize_decimals(
  normalized_amount,
  DEFAULT_NORMALIZED_DECIMALS,
  minted_decimals,
)?;
```

The **settlement.amount** (a **U256**) is converted into little-endian format before being stored in the 32-byte buffer **buf**. Afterwards, the right-most 16 bytes are parsed back into a **u128** but from big-endian format.

At first this seems incorrect and confusing, but it works because the **Settlement** struct inherits the default **AnchorSerialize** while overwriting the **AnchorDeserialize** to use big-endian parsing on the **amount** bytes.

What happens is that while the initial **Settlement.amount** has the correct amount, it gets written as bytes to the PDA using little-endian and later when the PDA is read again, the raw bytes get interpreted as big-endian because of the **AnchorDeserialize** trait implementation.

As a result, at line 82 in the code, the **settlement.amount** is flipped. For example, an amount of 500000000000000000000 becomes 22327863389048795604574193111063820799333501522095898215586715100250112O:

```
>>> int.from_bytes((500000000000000000000).to_bytes(32, 'little'), 'big')
22327863389048795604574193111063820799333501522095898215586715100250112O
```

So another conversion to little-endian is required to flip it back into big-endian.

The **settlement.amount** is only used on line 82, so currently there is no impact, but this is a confusing style of conversion. For example, the **settlement.amount** cannot be trusted in the function, so if any future upgrades use this, it could lead to serious critical issues down the line.

```
pub fn settle_delivered_intent(
    ctx: Context<SettleDeliveredIntentContext>,
    _ix: SettleDeliveredIntentInstruction,
) -> Result<()> {
    // assert settlement exists and the status is delivered
    require!(
        ctx.accounts.intent_status_pda.settlement.is_some()
            && ctx.accounts.intent_status_pda.status == IntentStatus::Delivered,
        SpokeError::InvalidIntentStatus
    );
    // verify all account here matches the one in the intent status pda (except the event authority as they have
seperate checks)
    require!(
        ctx.accounts.spoke_state.key() == ctx.accounts.intent_status_pda.accounts[0].pubkey,
        SpokeError::IncorrectSettlementAccounts
    );
    require!(
        ctx.accounts.intent_status_pda.key() == ctx.accounts.intent_status_pda.accounts[1].pubkey,
        SpokeError::IncorrectSettlementAccounts
    );
    require!(
        ctx.accounts.vault_authority.key() == ctx.accounts.intent_status_pda.accounts[2].pubkey,
        SpokeError::IncorrectSettlementAccounts
    );
    require!(
        ctx.accounts.token_program.key() == ctx.accounts.intent_status_pda.accounts[3].pubkey,
        SpokeError::IncorrectSettlementAccounts
    );
    require!(
        ctx.accounts.system_program.key() == ctx.accounts.intent_status_pda.accounts[4].pubkey,
        SpokeError::IncorrectSettlementAccounts
    );
    require!(
        ctx.accounts.mint_account.key() == ctx.accounts.intent_status_pda.accounts[5].pubkey,
        SpokeError::IncorrectSettlementAccounts
    );
    require!(
        ctx.accounts.associated_token_program.key()
            == ctx.accounts.intent_status_pda.accounts[6].pubkey,
        SpokeError::IncorrectSettlementAccounts
```

```
);
require!(
    ctx.accounts.recipient.key() == ctx.accounts.intent_status_pda.accounts[7].pubkey,
    SpokeError::IncorrectSettlementAccounts
);
require!(
    ctx.accounts.recipient_token_account.key()
        == ctx.accounts.intent_status_pda.accounts[8].pubkey,
    SpokeError::IncorrectSettlementAccounts
);
require!(
    ctx.accounts.vault_token_account.key() == ctx.accounts.intent_status_pda.accounts[9].pubkey,
    SpokeError::IncorrectSettlementAccounts
);

// SAFE: settlement existence is checked
let settlement = ctx.accounts.intent_status_pda.settlement.clone().unwrap();

// 2) Mark as settled in storage
ctx.accounts.intent_status_pda.status = IntentStatus::Settled;

let mut buf = [0u8; 32];
settlement.amount.to_little_endian(&mut buf);
let normalized_amount = u128::from_be_bytes(buf[16..32].try_into().unwrap());

// 3) Normalise the settlement amount
let minted_decimals = ctx.accounts.mint_account.decimals;
let amount = normalize_decimals(
    normalized_amount,
    DEFAULT_NORMALIZED_DECIMALS,
    minted_decimals,
)?;

require!(amount < u64::MAX.into(), SpokeError::InvalidAmount);

if amount == 0 {
    return Ok(());
}

// Create ATA idempotently
let create_idempotent_inst = create_associated_token_account_idempotent(
    ctx.accounts.authority.key,
    ctx.accounts.recipient.key,
    &ctx.accounts.mint_account.key(),
    ctx.accounts.token_program.key,
);
```

```
    msg!("{:?}", create_idempotent_inst);
    anchor_lang::solana_program::program::invoke(
        &create_idempotent_inst,
        &[
            ctx.accounts.authority.to_account_info(),
            ctx.accounts.recipient_token_account.to_account_info(),
            ctx.accounts.recipient.to_account_info(),
            ctx.accounts.mint_account.to_account_info(),
            ctx.accounts.system_program.to_account_info(),
            ctx.accounts.token_program.to_account_info(),
        ],
    )?;

    let signer_seeds: &[&[u8]] =
        vault_authority_pda_seeds!(ctx.accounts.spoke_state.vault_authority_bump);
    let signer = &[signer_seeds];

    let cpi_accounts = anchor_spl::token::Transfer {
        from: ctx.accounts.vault_token_account.to_account_info(),
        to: ctx.accounts.recipient_token_account.to_account_info(),
        authority: ctx.accounts.vault_authority.to_account_info(),
    };
    let cpi_ctx = CpiContext::new_with_signer(
        ctx.accounts.token_program.to_account_info(),
        cpi_accounts,
        signer,
    );

    // NOTE: Removed the virtual balance logic
    token::transfer(cpi_ctx, amount as u64)?;

    emit_cpi!(SettledEvent {
        intent_id: settlement.intent_id,
        recipient: settlement.recipient,
        asset: settlement.asset,
        amount: amount as u64,
        domain: ctx.accounts.spoke_state.domain,
    });
    Ok(())
}
```

Remediation:

We highly recommend refactoring these traits such that these conversions are no longer necessary. The statement **deserialize(serialize(T)) == T** should always hold, which is not the case currently.

One way would be to separate the decoding from the initial EVM data, such that you don't need to overwrite **AnchorDeserialize**.

EVER1-2 | Precision loss for high-decimal tokens during intent creation and settlement

Fixed

Severity:	Probability:	Impact:
Low	Unlikely	Low

Path:

programs/everclear_spoke/src/instructions/utils.rs
programs/everclear_spoke/src/instructions/intent/new_intent.rs

Description:

In both **handle_new_intent** and **settle_delivered_intent** amounts are normalized to **DEFAULT_NORMALIZED_DECIMALS** (currently set to 18) for hashing and messaging while transfers use differently scaled values.

Since the cross-chain intent only records the truncated (normalized) amount, the **settle_delivered_intent** function later settles using this lower value. As a result, the difference - representing the truncated precision or "dust" - is permanently locked in the spoke contract's vault, causing a minor loss of funds for the user.

While tokens with decimals higher than **DEFAULT_NORMALIZED_DECIMALS** are rare in the Solana ecosystem, this logic inconsistency could lead to small amounts of unclaimable tokens accumulating in the contract over time.

```
pub(crate) fn normalize_decimals(
    amount: u128,
    minted_decimals: u8,
    target_decimals: u8,
) -> Result<u128> {
    match minted_decimals.cmp(&target_decimals) {
        // No scaling needed
        std::cmp::Ordering::Equal => Ok(amount),
        // e.g. minted_decimals=9, target_decimals=6 => downscale
        std::cmp::Ordering::Greater => {
            let shift = minted_decimals - target_decimals;
            // prevent potential divide-by-zero or overshoot
            if shift > 12 {
                // you might fail or just saturate for large differences
                return err!(SpokeError::DecimalConversionOverflow);
            };
            Ok(amount / u128::from(10u64.pow(shift as u32)))
```

```
        }
        // minted_decimals < target_decimals => upscale
        std::cmp::Ordering::Less => {
            ...
        }
    }
}
```

```
pub fn handle_new_intent<'info>(
    accounts: &mut NewIntentAccounts<'info>,
    program_id: Pubkey, // for ctx.programId
    receiver: Pubkey,
    output_asset: Pubkey,
    amount: u64,
    amount_out_min: u128,
    ttl: u64,
    destinations: Vec<u32>,
    data: Vec<u8>,
    message_gas_limit: u64,
) -> Result<IntentAddedEvent> {
    ...
    let minted_decimals = accounts.mint.decimals;
    let normalized_amount =
        normalize_decimals(amount as u128, minted_decimals, DEFAULT_NORMALIZED_DECIMALS)?;
    require!(normalized_amount > 0, SpokeError::ZeroAmount); // Add zero amount check like Solidity

    ...
    // Transfer from user's token account -> program's vault
    let cpi_accounts: Transfer<'_> = Transfer {
        from: accounts.user_token_account.to_account_info(),
        to: accounts.program_vault_account.to_account_info(),
        authority: accounts.authority.to_account_info(),
    };
    let cpi_ctx = CpiContext::new(accounts.token_program.to_account_info(), cpi_accounts);
    token::transfer(cpi_ctx, amount)?;
```

Remediation:

Reject or explicitly handle mints whose decimals exceed DEFAULT_NORMALIZED_DECIMALS by enforcing a maximum or requiring divisibility by the downscale factor. Alternatively align normalization and transfer so that the amount locked hashed and later settled remain consistent without truncation.

EVER1-11 | Same signer account for fees and intent fills violates PoLP

Fixed

Severity:	Probability:	Impact:
Low	Rare	Medium

Path:

packages/contracts/solana-spoke/programs/everclear_spoke/src/instructions/intent/fill_intent.rs::fill_intent#L37-L132

Description:

The function **fill_intent** is an instruction that allows for the fulfillment of an intent. It requires a signature on the **FillSignParams** struct and its corresponding data fields. The signer is an account in the provided context:

```
#[account(address = fee_adapter_state.fee_signer)]
pub signer: AccountInfo<'info>,
```

The restriction on the account is that it's the same as the **fee_adapter_state.fee_signer,** which is the same signer for the **FeeData** in both **new_order** and **new_intent**.

This violates the principle of least privilege (PoLP) and we advise to separate these two signing authorities: one for fee signing and one for fulfillment signing.

By adhering to the principle of least privilege, it limits the potential impact of breaches, such as a compromise of the private kay of the fee signer account.

```
pub fn fill_intent(
    ctx: Context<FillIntent>,
    // origin intent, flattened
    origin_initiator: [u8; 32],
    // NOTE: origin_receiver is put in ctx for space saving using LUT
    origin_input_asset: [u8; 32],
    // NOTE: we do not need output_asset here as this woule be `ctx.mint`. This is removed for space saving using LUT.
    intent_origin: u32,
    origin_nonce: u64,
    origin_timestamp: u64,          // actually uint48 in Solidity
    origin_ttl: u64,                // actually uint48 in Solidity
    origin_amount: [u8; 32],        // big-endian, matching typical EVM usage
```

```
    origin_amount_out_min: [u8; 32], // uint256
    origin_destinations: Vec<u32>,
    origin_data: Vec<u8>,

    // data for fill intent
    amount_out: u64,
    receiver: Pubkey,
    destinations: Vec<u32>,

    // hyperlane params
    message_gas_limit: u64,
    signature: Vec<u8>,
) -> Result<()> {
    let evm_intent = EVMIntent {
        initiator: origin_initiator,
        receiver: ctx.accounts.origin_receiver.key().to_bytes(),
        input_asset: origin_input_asset,
        output_asset: ctx.accounts.mint.key().to_bytes(),
        origin: intent_origin,
        nonce: origin_nonce,
        timestamp: origin_timestamp,
        ttl: origin_ttl,
        amount: origin_amount,
        amount_out_min: origin_amount_out_min,
        destinations: origin_destinations,
        data: origin_data,
    };

    let mut accounts = FillIntentAccounts {
        spoke_state: ctx.accounts.spoke_state.clone().as_ref().clone(),
        mint: ctx.accounts.mint.clone(),
        token_program: ctx.accounts.token_program.clone(),
        origin_receiver: ctx.accounts.origin_receiver.clone(),
        solver_token_account: ctx.accounts.solver_token_account.clone(),
        origin_receiver_token_account: ctx.accounts.origin_receiver_token_account.clone(),
        authority: ctx.accounts.authority.clone(),
        intent_status_pda: ctx.accounts.intent_status_pda.clone(),
        pda_payer: ctx.accounts.pda_payer.clone(),
        system_program: ctx.accounts.system_program.clone(),
        spl_noop_program: ctx.accounts.spl_noop_program.clone(),
        hyperlane_mailbox: ctx.accounts.hyperlane_mailbox.clone(),
        mailbox_outbox: ctx.accounts.mailbox_outbox.clone(),
        dispatch_authority: ctx.accounts.dispatch_authority.clone(),
```

```
        unique_message_account: ctx.accounts.unique_message_account.clone(),
        dispatched_message_pda: ctx.accounts.dispatched_message_pda.clone(),
        igp_program: ctx.accounts.igp_program.clone(),
        igp_program_data: ctx.accounts.igp_program_data.clone(),
        igp_payment_pda: ctx.accounts.igp_payment_pda.clone(),
        configured_igp_account: ctx.accounts.configured_igp_account.clone(),
        inner_igp_account: ctx.accounts.inner_igp_account.clone(),
    };
    let program_id = *ctx.program_id;

    // verify signatures
    let intent_id = compute_intent_hash(&evm_intent);
    let sign_params = FillSignParams {
        intent_id,
        domain: THIS_DOMAIN,
        filler: ctx.accounts.authority.key(),
        amount_out,
        receiver: receiver.to_bytes(),
        destinations: destinations.clone(),
    };
    let signature_accounts = SignatureAccounts {
        signer: ctx.accounts.signer.clone(),
        instruction_sysvar: ctx.accounts.instruction_sysvar.clone(),
    };
    verify_signature(&sign_params, signature, signature_accounts)?;

    let event_data: IntentFilledEvent = handle_fill_intent(
        &mut accounts,
        program_id,
        evm_intent,
        amount_out,
        receiver,
        destinations,
        message_gas_limit,
    )
    .unwrap();

    emit_cpi!(event_data);

    Ok(())
}
```

Remediation:

See description.

EVER1-4 | Debug logging in production code

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Description:

The program retains msg! debug logging in production paths (e.g., settle.rs logs the idempotent ATA instruction). These logs add compute cost and clutter transaction logs on mainnet without providing operational value, slightly increasing fees and reducing log clarity for monitoring/ indexing.

For example, in **fill_intent.rs** on lines 214-221:

```
msg!("{:?}", inst);
msg!(
    "{:?}",
    Pubkey::create_program_address(
        &[b"everclear_spoke", b"-", b"pda_payer", &[payer_pda_bump]],
        &program_id
    )
);
```

Remediation:

Remove non-essential msg! calls from production code, keeping only logs required for operational monitoring. If visibility is needed, prefer structured events over ad-hoc debug messages.

EVER1-9 | Typographical error in HandleFeeAccounts

Fixed

Severity:	Probability:	Impact:
Informational	Very likely	Informational

Path:

packages/contracts/solana-spoke/programs/everclear_spoke/src/instructions/fee_adapter/fees.rs::HandleFeeAccounts#L30-L39

Description:

In the **HandleFeeAccounts** struct, the **fee_reciever_account** and **fee_reciever_token_account** have a typographical error.

It should be corrected to **fee_receiver_account** and **fee_receiver_token_account** instead.

```
pub struct HandleFeeAccounts<'info> {
    pub signature_accounts: SignatureAccounts<'info>,
    pub user_account: AccountInfo<'info>,
    pub user_token_account: AccountInfo<'info>,
    pub user_authority_account: AccountInfo<'info>,
    pub fee_reciever_account: AccountInfo<'info>,
    pub fee_reciever_token_account: AccountInfo<'info>,
    pub token_program: AccountInfo<'info>,
    pub system_program: AccountInfo<'info>,
}
```

Remediation:

See description.

EVER1-10 | Redundant state pause check in new order instruction

Fixed

Severity:	Probability:	Impact:
Informational	Very likely	Informational

Path:

packages/contracts/solana-spoke/programs/everclear_spoke/src/instructions/fee_adapter/new_order.rs::new_order#L12-L103

Description:

The instruction **new_order** contains a check for the spoke state paused field on line 19:

```
require!(!state.paused, SpokeError::ContractPaused);
```

However, this function later flows into **handle_new_intent** for each **OrderParameters** in the **params** parameter field. Given the other check on the params parameter being non-empty, it will always flow into **handle_new_intent**.

The function **handle_new_intent** already contains a check on the **SpokeState.paused** and so the first check in **new_order** is redundant.

```
pub fn new_order(
    ctx: Context<NewIntent>,
    params: Vec<OrderParameters>,
    fee_param: FeeParams,
) -> Result<()> {
    let state = &mut ctx.accounts.spoke_state;

    require!(!state.paused, SpokeError::ContractPaused);
    require!(!params.is_empty(), SpokeError::EmptyParams);
    require!(
        !ctx.accounts.fee_adapter_state.paused,
        SpokeError::FeeAdapterPaused
    );

    // NOTE: asset is required to be the same from accounts.mint.
    let mut accounts = NewIntentAccounts {
        spoke_state: ctx.accounts.spoke_state.as_ref().clone(),
        mint: ctx.accounts.mint.clone(),
```

```
    token_program: ctx.accounts.token_program.clone(),
    program_vault_account: ctx.accounts.program_vault_account.clone(),
    user_token_account: ctx.accounts.user_token_account.clone(),
    authority: ctx.accounts.authority.clone(),
    system_program: ctx.accounts.system_program.clone(),
    spl_noop_program: ctx.accounts.spl_noop_program.clone(),
    hyperlane_mailbox: ctx.accounts.hyperlane_mailbox.clone(),
    mailbox_outbox: ctx.accounts.mailbox_outbox.clone(),
    dispatch_authority: ctx.accounts.dispatch_authority.clone(),
    unique_message_account: ctx.accounts.unique_message_account.clone(),
    dispatched_message_pda: ctx.accounts.dispatched_message_pda.clone(),
    igp_program: ctx.accounts.igp_program.clone(),
    igp_program_data: ctx.accounts.igp_program_data.clone(),
    igp_payment_pda: ctx.accounts.igp_payment_pda.clone(),
    configured_igp_account: ctx.accounts.configured_igp_account.clone(),
    inner_igp_account: ctx.accounts.inner_igp_account.clone(),
};

let program_id = *ctx.program_id;

let fee_data = FeeData {
    token_fee: fee_param.token_fee,
    native_fee: fee_param.native_fee,
    input_asset: ctx.accounts.mint.key(),
    deadline: fee_param.deadline,
};
let fee_accounts = HandleFeeAccounts {
    signature_accounts: SignatureAccounts {
        signer: ctx.accounts.fee_signer.to_account_info(),
        instruction_sysvar: ctx.accounts.instruction_sysvar.to_account_info(),
    },
    user_account: ctx.accounts.authority.to_account_info(),
    user_token_account: ctx.accounts.user_token_account.to_account_info(),
    user_authority_account: ctx.accounts.authority.to_account_info(),
    fee_reciever_account: ctx.accounts.fee_recipient.to_account_info(),
    fee_reciever_token_account: ctx.accounts.fee_recipient_token_account.to_account_info(),
    token_program: ctx.accounts.token_program.to_account_info(),
    system_program: ctx.accounts.system_program.to_account_info(),
};

handle_fees(fee_data, fee_param.signature, fee_accounts)?;

let mut intent_ids: Vec<[u8; 32]> = Vec::with_capacity(params.len());
```

```
    for p in &params {
        let event_data = handle_new_intent(
            &mut accounts,
            program_id,
            p.receiver,
            p.output_asset,
            p.amount,
            p.amount_out_min,
            p.ttl,
            p.destinations.clone(),
            p.data.clone(),
            p.message_gas_limit,
        )?;

        emit_cpi!(event_data);

        intent_ids.push(event_data.intent_id);
    }

    let order_id = hash_intent_id_array(&intent_ids);

    emit_cpi!(OrderCreated {
        order_id,
        user: ctx.accounts.authority.key(),
        intent_ids: intent_ids.clone(),
        fee: fee_param.token_fee,
        native_value: fee_param.native_fee,
    });

    Ok(())
}
```

Remediation:

Remove the redundant check in **new_order** on line 19.

EVER1-13 | TTL/output validation diverges between Solana and EVM newIntent

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

packages/contracts/solana-spoke/programs/everclear_spoke/src/instructions/intent/new_intent.rs
packages/contracts/src/contracts/intent/EverclearSpokeV5.sol

Description:

The validation logic for creating new intents differs between the Solidity and Solana implementations. In the Solidity contract, single-destination intents are allowed to have a null **outputAsset** if the **ttl** is zero. However, the Solana program strictly enforces that the **output_asset** must be a non-default value for all single-destination intents, ignoring the **ttl** parameter. This mismatch permits creating immediately expired single-destination intents on Solana, causing behavior to differ from the EVM version.

```
pub fn handle_new_intent<'info>(
    accounts: &mut NewIntentAccounts<'info>,
    program_id: Pubkey, // for ctx.programId
    receiver: Pubkey,
    output_asset: Pubkey,
    amount: u64,
    amount_out_min: u128,
    ttl: u64,
    destinations: Vec<u32>,
    data: Vec<u8>,
    message_gas_limit: u64,
) -> Result<IntentAddedEvent> {
    ...
    if destinations.len() == 1 {
        require!(output_asset != Pubkey::default(), SpokeError::InvalidIntent);
    } else {
        require!(
            ttl == 0 && output_asset == Pubkey::default(),
            SpokeError::InvalidIntent
        );
    }
    ...
```

```
function _newIntent(
  uint32[] memory _destinations,
  bytes32 _receiver,
  address _inputAsset,
  bytes32 _outputAsset,
  uint256 _amount,
  uint256 _amountOutMin,
  uint48 _ttl,
  bytes calldata _data,
  bool _usesPermit2
) internal returns (bytes32 _intentId, Intent memory _intent) {
  if (_destinations.length == 1) {
    // output asset should not be null if the intent has a single destination and ttl != 0
    if (_ttl != 0 && _outputAsset == 0) revert EverclearSpoke_NewIntent_OutputAssetNull();
  } else {
    // output asset should be null if the intent has multiple destinations
    // ttl should be 0 if the intent has multiple destinations
    if (_ttl != 0 || _outputAsset != 0) revert EverclearSpoke_NewIntent_OutputAssetNotNull();
  }
  ...
```

Remediation:

Align the Solana **new_intent** validation with the EVM rules: apply the same **ttl/output_asset** checks for single- and multi-destination intents so intents cannot be created with **ttl = 0** unless explicitly intended.


hexens
x
Everclear