hexens x train

Security Review Report for Train Protocol

May 2025

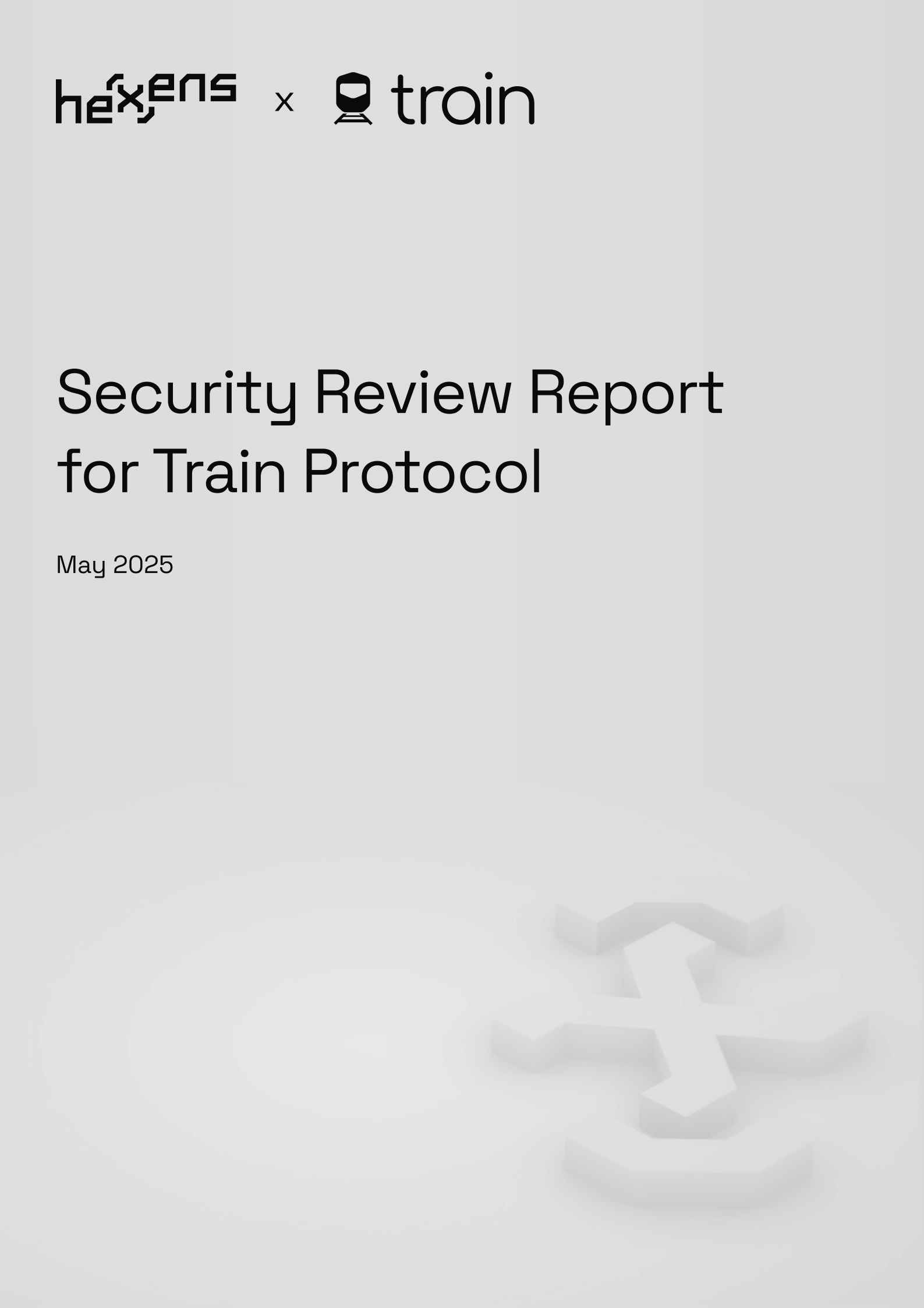


Table of Contents

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 covered the Solana and Fuel contracts of the Train HTLC cross-chain functionality. This the Hashed Time Lock Contract and allows for users and liquidity providers to perform atomic swaps across chains using commitments and time locks.

Our security assessment was a full review of the smart contracts in scope, spanning a total of 1 week.

During our audit, we did identified 2 high severity vulnerabilities. The first one would allow an LP to game the time lock and potentially steal a user's assets, while the second one was an incorrect comparison of the time lock value.

We also identified several minor severity vulnerabilities and code optimisations.

All of our reported issues were fixed or acknowledged 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/TrainProtocol/contracts/tree/7ee13734cb8c800badeda7f51783db29152e64a7/chains/solana
- https://github.com/TrainProtocol/contracts/tree/6462ba7237a8b8e2501d2d6a8ff9d89cb4f550de/chains/fuel

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

- https://github.com/TrainProtocol/contracts/tree/dbf3d6a9d6362c2f58b6d32c639c2f6b556b4d5f/chains/solana
- https://github.com/TrainProtocol/contracts/tree/45730db8d003cc89578fc044a1714c058d07b1c5/chains/fuel

Changelog

19 May 2025	Audit start
27 May 2025	Initial report
29 May 2025	Revision received
02 June 2025	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.

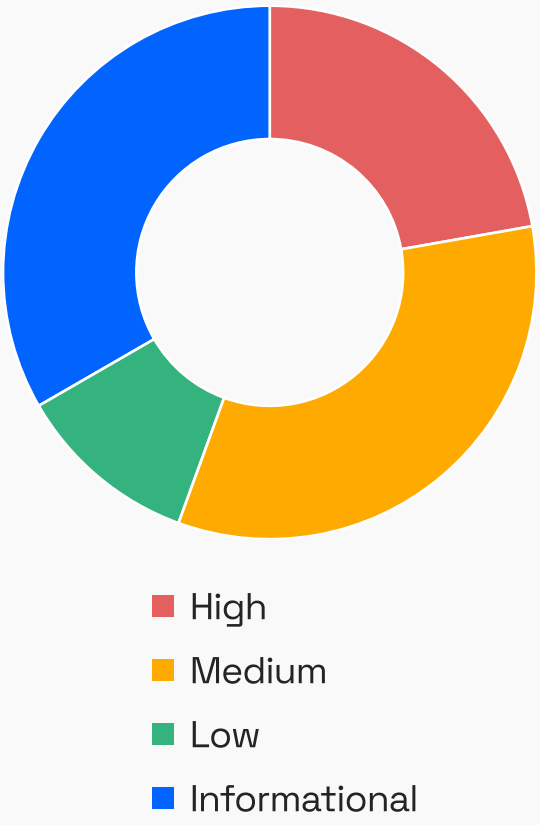
- **Issue Symbolic Codes**

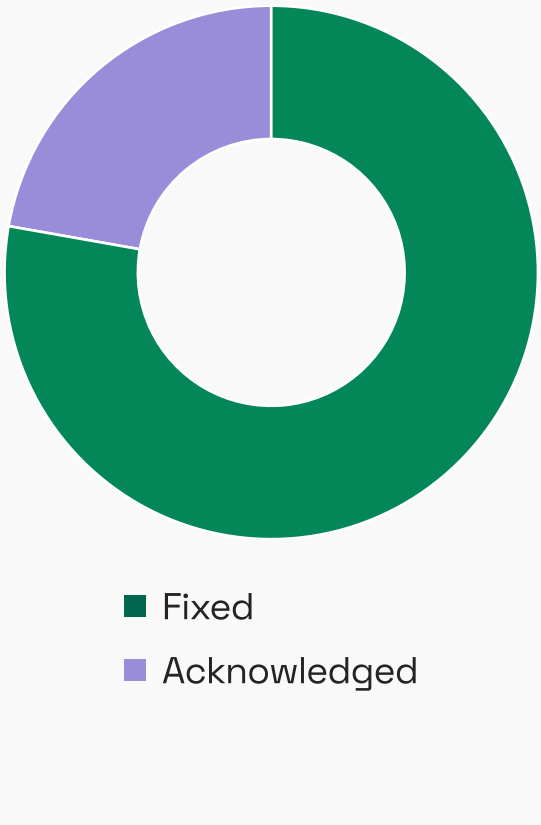
Each identified and validated issue is assigned a unique symbolic code during the security research stage.

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

5. Findings Summary

Severity	Number of findings
Critical	0
High	2
Medium	3
Low	1
Informational	3
Total:	**9**





6. Weaknesses

This section contains the list of discovered weaknesses.

TRNP-3 | Solana HTLC lock timestamp delta can be abused by LPs

Fixed

Severity:	Probability:	Impact:
High	Unlikely	Critical

Path:

contracts/chains/solana/sol/programs/sol/src/lib.rs:lock#L106-L156

Description:

The function **lock** accepts a lock timestamp of any value greater than the current timestamp, similar to the **commit** function.

Because of this, an LP could lock the assets with a timestamp that would allow them both call **refund** on the destination chain and **redeem** on the origin chain, effectively stealing the assets from the user.

This is the same issue as **LYSWP2-7** that was found in the Train EVM contract.

This issue also exists in the Cairo HTLC contract: contracts/chains/starknet/src/TrainERC20.cairo at dev · TrainProtocol/contracts

```
pub fn lock(
    [..]
) -> Result<[u8; 32]> {
    let clock = Clock::get().unwrap();
    require!(
        timelock > clock.unix_timestamp.try_into().unwrap(),
        HTLCError::NotFutureTimeLock
    );
    [..]
}
```

Remediation:

We recommend to enforce a delay on the lock timestamp, similar to the Train EVM contracts where there is a minimum of 30 minutes delay required.

TRNP-9 | Fuel HTLC redeem function uses incorrect time condition to distribute rewards

Fixed

Severity:	Probability:	Impact:
High	Likely	High

Path:

contracts/blob/dev/chains/fuel/src/main.sw#L418-L424

Description:

In the **redeem** function of Fuel's HTLC contract, if the lock contains rewards, it distributes the rewards to the caller (msg.sender) when the current timestamp is less than **reward.timelock**. When **reward.timelock** < **timestamp()**, the rewards are refunded to the HTLC's sender.

```
let reward: Reward = storage.rewards.get(Id).try_read().unwrap();
// Check if reward timelock has passed and distribute funds accordingly
if reward.timelock < timestamp() {
    transfer(
        Identity::Address(htlc.srcReceiver),
        htlc.assetId,
        htlc.amount,
    );
    transfer(Identity::Address(htlc.sender), htlc.assetId, reward.amount);
} else {
    let sender = match msg_sender().unwrap() {
        Identity::Address(addr) => addr,
        _ => {
            require(false, "Not an Address");

Address::from(0x0000000000000000000000000000000000000000000000000000000000000000)
        }
    };
    if htlc.srcReceiver == sender {
        transfer(
            Identity::Address(htlc.srcReceiver),
            htlc.assetId,
            reward
                .amount + htlc.amount,
        );
```

```
    } else {
        transfer(Identity::Address(sender), htlc.assetId, reward.amount);
        transfer(
            Identity::Address(htlc.srcReceiver),
            htlc.assetId,
            htlc.amount,
        );
    }
}
```

However, the rewards of the lock were intended to incentivize late redemption when the current time has passed **reward.timelock**. Therefore, the redeem function of Fuel's HTLC incorrectly distributes the reward to the caller at the wrong time. As evidence, the **redeem** functions in the EVM's HTLC contract, Solana's HTLC contract, and other networks distribute rewards to the caller (msg.sender) when the current timestamp is greater than **reward.timelock**.
contracts/chains/solana/sol/programs/sol/src/lib.rs at dev · TrainProtocol/contracts
contracts/chains/evm/solidity/contracts/TrainERC20.sol at dev · TrainProtocol/contracts

Remediation:

The condition for reward distribution should be reversed.

TRNP-4 | Solana HTLC lock_reward overwrites the previous reward value

Fixed

Severity:	Probability:	Impact:
Medium	Rare	High

Path:

contracts/chains/solana/sol/programs/sol/src/lib.rs:lock_reward#L157-L190

Description:

The function **lock_reward** allows the creator of an HTLC to add a reward amount and reward timestamp to the HTLC.

However, the function does not check whether the reward amount was not already zero. As a result, any subsequent calls would overwrite the value, which would lead to the previous reward to become lost.

```
    pub fn lock_reward(
        ctx: Context<LockReward>,
        Id: [u8; 32],
        reward_timelock: u64,
        reward: u64,
        lock_bump: u8,
    ) -> Result<bool> {
        let clock = Clock::get().unwrap();
        let htlc = &mut ctx.accounts.htlc;

        require!(
            reward_timelock < htlc.timelock
                && reward_timelock > clock.unix_timestamp.try_into().unwrap(),
            HTLCError::InvalidRewardTimeLock
        );

        htlc.reward_timelock = reward_timelock;
        htlc.reward = reward;

        let bump_vector = lock_bump.to_le_bytes();
        let inner = vec![Id.as_ref(), bump_vector.as_ref()];
        let outer = vec![inner.as_slice()];
        let transfer_context = CpiContext::new_with_signer(
```

```
            ctx.accounts.system_program.to_account_info(),
            system_program::Transfer {
                from: ctx.accounts.sender.to_account_info(),
                to: htlc.to_account_info(),
            },
            outer.as_slice(),
        );
        system_program::transfer(transfer_context, reward)?;

        Ok(true)
    }
```

Remediation:

We recommend to add a require statement that checks whether a reward was already added to the HTLC.

For example:

```
require!(htlc.reward == 0, HTLCError::RewardAlreadyExists);
```

TRNP-5 | Solana HTLC commit does not enforce a timelock delta

Fixed

Severity:	Probability:	Impact:
Medium	Likely	Medium

Path:

contracts/chains/solana/sol/programs/sol/src/lib.rs:commit#L42-L97

Description:

The **commit** function in the Solana contract does not enforce a delta on the lock timestamp and instead allows it to be 1 second in the future. This is in contradiction with the protocol and could allow for exploitable race conditions.

```
pub fn commit(
    [..]
) -> Result<[u8; 32]> {
    let clock = Clock::get().unwrap();
    require!(
        timelock > clock.unix_timestamp.try_into().unwrap(),
        HTLCError::NotFutureTimeLock
    );
    [..]
}
```

Remediation:

The **commit** function should enforce a delta timestamp equal to the Train EVM contract, which is 900 seconds.

TRNP-6 | Solana HTLC add_lock does not enforce a timelock delta

Fixed

Severity:	Probability:	Impact:
Medium	Likely	Medium

Path:

contracts/chains/solana/sol/programs/sol/src/lib.rs:add_lock#L196-L213

Description:

The **add_lock** function in the Solana contract does not enforce a delta on the lock timestamp and instead allows it to be 1 second in the future. This is in contradiction with the protocol and could allow for exploitable race conditions.

```
pub fn add_lock(
    ctx: Context<AddLock>,
    Id: [u8; 32],
    hashlock: [u8; 32],
    timelock: u64,
) -> Result<[u8; 32]> {
    let clock = Clock::get().unwrap();
    require!(
        timelock > clock.unix_timestamp.try_into().unwrap(),
        HTLCError::NotFutureTimeLock
    );

    [..]
}
```

Remediation:

The **commit** function should enforce a delta timestamp equal to the Train EVM contract, which is 900 seconds.

TRNP-2 | EVM addLockSig gas griefing against solver

Acknowledged

Severity:	Probability:	Impact:
Low	Likely	Low

Path:

contracts/chains/evm/solidity/contracts/{Train,TrainERC20}.sol:addLockSig#L260-L286, L289-L315

Description:

When a user deposits funds via the **commit** function and provides a signature, the relayer calls the **addLockSig** function to verify the signature and then initializes the **hashlock** and **timelock**.

If **htlc.sender** is a contract, the function calls **isValidSignature** to verify the signature using the ERC-1271 standard.

```
function addLockSig(
  addLockMsg calldata message,
  bytes32 r,
  bytes32 s,
  uint8 v
) external _exists(message.Id) _validTimelock(message.timelock) nonReentrant
returns (bytes32) {
  HTLC storage htlc = contracts[message.Id];
  bool verified = false;
  if (htlc.sender.code.length == 0) {
    verified = verifyMessage(message, r, s, v);
  } else {
    bytes memory signature = abi.encodePacked(r, s, v);
    bytes32 digest = keccak256(abi.encodePacked('\\x19\\x01',
_domainSeparatorV4(), hashMessage(message)));
    verified = SignatureChecker.isValidERC1271SignatureNow(htlc.sender, digest,
signature);
  }
 ...
}
```

```
/**
 * @dev Checks if a signature is valid for a given signer and data hash. The
signature is validated
 * against the signer smart contract using ERC-1271.
 *
 * NOTE: Unlike ECDSA signatures, contract signatures are revocable, and the
outcome of this function can thus
 * change through time. It could return true at block N and false at block N+1
(or the opposite).
 */
function isValidERC1271SignatureNow(
    address signer,
    bytes32 hash,
    bytes memory signature
) internal view returns (bool) {
    (bool success, bytes memory result) = signer.staticcall(
        abi.encodeCall(IERC1271.isValidSignature, (hash, signature))
    );
    return (success &&
        result.length >= 32 &&
        abi.decode(result, (bytes32)) ==
bytes32(IERC1271.isValidSignature.selector));
}
```

Additionally, with the recent **EIP-7702: Set Code for EOAs** update, externally owned accounts (EOAs) can now temporarily act as smart contracts.

An attacker could exploit this logic by triggering a call to **isValidSignature** and deploying a contract with meaningless or gas-intensive logic. This causes the relayer to unnecessarily consume gas executing the attacker's contract code.

```
var childWorkflowTask = ExecuteTransactionAsync(new TransactionRequest()
{
    PrepareArgs = JsonSerializer.Serialize(new
AddLockSigTransactionPrepareRequest
    {
        Id = _htlcCommitMessage.Id,
        Hashlock = hashlock.Hash,
        Signature = _htlcAddLockSigMessage.Signature,
        SignatureArray = _htlcAddLockSigMessage.SignatureArray,
        Timelock = _htlcAddLockSigMessage.Timelock,
        R = _htlcAddLockSigMessage.R,
        S = _htlcAddLockSigMessage.S,
```

```
            V = _htlcAddLockSigMessage.V,
            Asset = _htlcCommitMessage.SourceAsset,
            SignerAddress = _htlcAddLockSigMessage.SignerAddress,
        }),
        Type = TransactionType.HTLCAddLockSig,
        NetworkName = _htlcCommitMessage.SourceNetwork,
        NetworkType = _htlcCommitMessage.SourceNetworkType,
        FromAddress = _solverManagedAccountInSource!,
        SwapId = _swapId
    });
```

The above code executes the solver's **addLockSig** function, and since there are no restrictions on gas usage, there is a potential risk that the relayer's gas could be completely exhausted.

GitHub - TrainProtocol/solver: Solver Implementation for TRAIN protocol

Remediation:

Consider to employ some limit to the amount of gas for the transaction in the solver code, as automatic estimation might lead to gas griefing.

TRNP-7 | Solana HTLC refund timestamp check is off-by-one

Fixed

Severity:	Probability:	Impact:
Informational	Likely	Informational

Path:

contracts/chains/solana/sol/programs/sol/src/lib.rs:Refund#L410-L431

Description:

The Refund context enforces a constraint on the **htlc.timelock** to be greater than or equal to the current time (**>=**). This in contradiction with the specification and the Train EVM contract, where the current time should have passed the lock timestamp (**>**).

This creates a second window where the logic of the protocol on different chains diverges.

```
pub struct Refund<'info> {
    #[account(mut)]
    user_signing: Signer<'info>,

    #[account(mut,
    seeds = [
        Id.as_ref()
    ],
    bump,
    has_one = sender @HTLCError::NotSender,
    constraint = htlc.claimed == 1 @ HTLCError::AlreadyClaimed,
    constraint = Clock::get().unwrap().unix_timestamp >=
htlc.timelock.try_into().unwrap() @ HTLCError::NotPastTimeLock,
    )]
    pub htlc: Box<Account<'info, HTLC>>,

    ///CHECK: The sender
    #[account(mut)]
    sender: UncheckedAccount<'info>,

    system_program: Program<'info, System>,
    rent: Sysvar<'info, Rent>,
}
```

Remediation:

Use strictly greater than (**>**) for the check on the timestamp.

TRNP-8 | Solana HTLC add_lock does not check for a zero hashlock

Acknowledged

Severity:	Probability:	Impact:
Informational	Likely	Informational

Path:

contracts/chains/solana/sol/programs/sol/src/lib.rs:add_lock#L196-L213

Description:

The function **add_lock** allows the user to add a hashlock to an HTLC. The context contains a constraint to check whether the existing hashlock is the default value (zero bytes) and only allows to add a hashlock in that case.

However, the function does not check whether the parameter **hashlock** isn't also the default value, after which you could set the **timelock** but not the **hashlock**.

```
    pub fn add_lock(
        ctx: Context<AddLock>,
        Id: [u8; 32],
        hashlock: [u8; 32],
        timelock: u64,
    ) -> Result<[u8; 32]> {
        let clock = Clock::get().unwrap();
        require!(
            timelock > clock.unix_timestamp.try_into().unwrap(),
            HTLCError::NotFutureTimeLock
        );

        let htlc = &mut ctx.accounts.htlc;
        htlc.hashlock = hashlock;
        htlc.timelock = timelock;

        Ok(Id)
    }
```

Remediation:

We recommend to add a require check to **add_lock** to enforce the setting of the **hashlock** to an actual value and just zero bytes.

TRNP-10 | Unvalidated Bump Parameters in PDA Operations

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

contracts/chains/solana/sol/programs/sol/src/lib.rs

Description:

The **commit** and **lock** functions accept bump parameters (**commit_bump** and **lock_bump** respectively) without validating that these values match the actual canonical bump used to derive the PDA address.

In both functions, the provided bump values are used to generate seeds for CPI contexts:

```
let bump_vector = commit_bump.to_le_bytes();
let inner = vec![Id.as_ref(), bump_vector.as_ref()];
let outer = vec![inner.as_slice()];

let transfer_context = CpiContext::new_with_signer(
    ctx.accounts.system_program.to_account_info(),
    system_program::Transfer { ... },
    outer.as_slice(),
);
```

While Anchor correctly validates the PDA address using **#[account(seeds=[Id.as_ref()], bump)]**, it does not enforce that the provided bump matches this derived value. This creates a potential discrepancy between address validation and other operations.

The bump parameters are also unnecessary for the specific transfers being performed, as these transfers are from user-owned accounts (which are already signers) to PDA accounts.

Remediation:

We recommend to use Anchor's public bump field of the HTLC account directly for the seeds of the SOL transfer.


hexens
x
train