SolLightning

Security Audit

by Ackee Blockchain

12.1.2024

Contents

1. Document Revisions

1.0	Final report	20.12.2023
2.0	Re-audit final report	12.1.2024

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain

Ackee Blockchain is an auditing company based in Prague, Czech Republic, specializing in audits and security assessments. Our mission is to build a stronger blockchain community by sharing knowledge – we run free certification courses School of Solana, Summer School of Solidity and teach at the Czech Technical University in Prague. Ackee Blockchain is backed by the largest VC fund focused on blockchain and DeFi in Europe, RockawayX.

2.2. Audit Methodology

The Ackee Blockchain auditing process follows a routine series of steps:

1. **Code review**
    a. High-level review of the specifications, sources, and instructions provided to us to make sure we understand the project's size, scope, and functionality.
    b. Detailed manual code review, which is the process of reading the source code line-by-line to identify potential vulnerabilities. We focus mainly on common classes of Solana program vulnerabilities, such as:

       missing ownership checks, missing signer authorization, signed CPI of unverified programs, cosplay of Solana accounts, missing rent exemption assertion, bump seed canonicalization, incorrect accounts closing, casting truncation, numerical precision errors, arithmetic overflows or underflows, ...

    c. Comparison of the code and given specifications, ensuring that the program logic correctly implements everything intended.

   d. Review of best practices to improve efficiency, clarity, and maintainability.

2. **Testing and automated analysis**

   a. Run client's tests to ensure that the system works as expected, potentially write missing unit or fuzzy tests using our testing framework Trdelnik.

3. **Local deployment + hacking**

   a. The programs are deployed locally, and we try to attack the system and break it. There is no specific strategy here, and each project's attack attempts are characteristic of each program audited. However, when trying to attack, we rely on the information gained from previous steps and our rich experience.

2.3. Finding classification

A *Severity* rating of each finding is determined as a synthesis of two sub-ratings: *Impact* and *Likelihood*. It ranges from *Informational* to *Critical*.

If we have found a scenario in which an issue is exploitable, it will be assigned an impact rating of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of *Warning* or *Info*.

Low to *High* impact issues also have a *Likelihood*, which measures the probability of exploitability during runtime.

The full definitions are as follows:

Severity

		Likelihood			
		High	**Medium**	**Low**	**-**
Impact	**High**	Critical	High	Medium	-
	Medium	High	Medium	Low	-
	Low	Medium	Low	Low	-
	Warning	-	-	-	Warning
	Info	-	-	-	Info

Table 1. Severity of findings

Impact

- **High** - Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- **Medium** - Code that activates the issue will result in consequences of serious substance.
- **Low** - Code that activates the issue will have outcomes on the system that are either recoverable or don’t jeopardize its regular functioning.
- **Warning** - The issue cannot be exploited given the current code and/or configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.), but could be a security vulnerability if these were to change slightly. If we haven’t found a way to exploit the issue given the time constraints, it might be marked as a "Warning" or higher, based on our best estimate of whether it is currently exploitable.
- **Info** - The issue is on the borderline between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or configuration (see above) was to change.

Likelihood

- **High** - The issue is exploitable by virtually anyone under virtually any circumstance.
- **Medium** - Exploiting the issue currently requires non-trivial preconditions.
- **Low** - Exploiting the issue requires strict preconditions.

2.4. Review team

Member's Name	Position
Adam Hrazdira	Lead Auditor
Jan Kalivoda	Auditor
Andrej Lukačovič	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.5. Disclaimer

We've put our best effort to find all vulnerabilities in the system, however our findings shouldn't be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

3. Executive Summary

Revision 1.0

SolLightning engaged Ackee Blockchain to perform a security review of the SolLightning protocol with a total time donation of 15 engineering days in a period between December 04 and December 18, 2023, with Adam Hrazdira as the lead auditor.

The scope included two programs in the following repositories and with the given commits:

- **Repository:** https://github.com/adambor/SolLightning-program
- **Commit:** 144010785eeb6616e70255d9146395fd63187abd

and

- **Repository:** https://github.com/adambor/BTCRelay-Sol
- **Commit:** 81f2ab91ec74713c17d7293f0c070504d825a34a

The beginning of the review was dedicated to understanding the high-level goals and architecture of the project. We then took a deep dive into the logic of both programs. During the review, we paid special attention to:

- ensuring the program is correctly implemented according to the documentation and matching project goals,
- ensuring there are no hazards for future refactoring and development,
- ensuring the program correctly uses dependencies or other programs it relies on (e.g. SPL dependencies),
- ensuring no critical information is missing from the state of any party,
- ensuring that the relevant data is signed properly and that the signatures

are verified properly,

- ensuring that the transactions cannot be modified in a malicious way before broadcasting to the network,
- ensuring it is impossible to perform a replay attack,
- ensuring the program is not vulnerable to economic attacks,
- ensuring the arithmetic of the system is correct,
- looking for common issues such as data validation, and owner and address checks.

Our review resulted in 20 findings, ranging from Info to Critical severity. Three of these issues were assessed as of critical severity. The issue C1 is related to a vulnerability that leads to a loss of the refundable fee of the escrow contract initializer. The issue C2 exposes a vulnerability, that allows fraudulent claiming of locked funds from an escrow contract using a replay attack. Finally, the issue C3 allows anyone to lock assets of the intermediary node without the possibility of claiming it back.

Regarding the general code quality, the codebase uses the Anchor framework version 0.26.0. Both programs are well-structured and modularized, however, the SolLightning program reuses instructions in multiple situations (ie. there is only one claim/refund instruction for all 6 transfer possibilities). This makes the program flow much harder to follow and the programmer must deal with multiple combinations of instruction parameters and corresponding sets of input accounts. It also requires the user of the protocol to have a deeper insight into the backbone of the protocol, as the API is vague. The code style is mostly consistent with sufficient explanatory comments, however, the advantages of Rust language are not always fully utilized as described in W1, W2, W4 and W5. Both programs have integration tests in Typescript (SolLightning-program: 11 passing, 6 failing tests; BTCRelay-Sol: 2 passing, 3 disabled tests). However, the tests contain only basic test scenarios, commented code, disabled tests and overall look incomplete for a project of this size and complexity. Based on Git commit history, both programs are entirely maintained (developed and tested) by one developer. Both projects do not have any CI/CD pipelines or automated tests set up on GitHub.

Ackee Blockchain recommends SolLightning:

- Address all reported issues.
- Write a more extensive test suite and include ”unhappy path” scenarios with unexpected accounts, input parameters and program flows.
- Automate the tests using GitHub workflows.
- Avoid complicated program flows and keep the instructions as straightforward and easy to understand as possible.
- Write a more detailed protocol documentation.
- Consider upgrading the Anchor version. (Although the currently used version does not contain any known security issues, it is not possible to compile the code with higher versions of the Solana suite than 1.14 and the latest Anchor version also brings additional benefits such as smaller binary sizes.)

The project is not ready for production deployment. Based on the number and severity of the findings and an extensive list of recommendations, we ask the client to address these issues and perform another full-scope audit revision once implemented.

See Revision 1.0 for the system overview of the codebase.

Revision 2.0

SolLightning engaged Ackee Blockchain to perform a security review of the SolLightning protocol with a total time donation of 3 engineering days in a period between January 08 and January 11, 2024, with Adam Hrazdira as the lead auditor.

The scope included two programs in the following repositories and with the given commits:

- **Repository:** https://github.com/adambor/SolLightning-program
- **Commit:** 45594fa3039a284546dbb5d0ccde4bacc76de91f

and

- **Repository:** https://github.com/adambor/BTCRelay-Sol
- **Commit:** e393a6a5820da6d57f64f6d076ab26bd92802ce1

For revision 2.0, we had the following goals:

- Review the fixes implemented in response to our previous audit.
- Review the refactored code to ensure that the new structure does not introduce new vulnerabilities.

We began our review by analyzing the new codebase and the introduced changes. The analysis showed that most of the issues identified in the previous scope were effectively addressed, improving both programs' overall functionality and security. However, two medium-severity issues were acknowledged but not explicitly fixed in the on-chain programs (see issues the M1 and M4 for more details and the client's responses). Additionally, we identified one new low-severity issue L5.

The codebase refactoring introduced new instructions that make the API simpler and easier to understand. These additional changes increase the overall code quality and readability, indicating a more mature state of the project. The SolLightning-program now contains over 600 test cases, providing good test coverage. However, the BTCRelay-Sol program still only contains 5 test cases.

Ackee Blockchain recommends SolLightning:

- Fix the L5 issue.
- Write a more extensive test suite for the BTCRelay-Sol program.
- Automate the tests using GitHub workflows.
- Write more detailed protocol documentation including a detailed description of the API parameters.

Based on the fixes introduced in revision 2.0 we don't see any further blockers for production deployment.

4. Summary of Findings

The following table summarizes the findings we identified during our review. Unless overridden for purposes of readability, each finding contains:

- a *Description*,
- an *Exploit scenario*,
- a *Recommendation* and if applicable
- a *Fix*.

There might often be multiple ways to solve or alleviate the issue, with varying requirements regarding the necessary changes to the codebase. In that case, we will try to enumerate them all, clarifying which solves the underlying issue better (albeit possibly only with architectural changes) than others.

	Severity	Reported	Status
C1: Initializer can lose refundable fee	Critical	1.0	Fixed
C2: Transaction hash uniqueness not guaranteed	Critical	1.0	Fixed
C3: Possibility to lock funds	Critical	1.0	Fixed
M1: Artificial reputation increase	Medium	1.0	Acknowledged
M2: Number of confirmations not limited	Medium	1.0	Fixed
M3: Unchecked token account	Medium	1.0	Fixed
M4: Multiple lock-times	Medium	1.0	Acknowledged

	Severity	Reported	Status
M5: Reputation can overflow	Medium	1.0	Partially fixed
L1: Passing Rent SysVar	Low	1.0	Fixed
L2: Unused instruction parameters	Low	1.0	Fixed
L3: Bump calculation	Low	1.0	Fixed
L4: Account does not need to be mutable	Low	1.0	Fixed
W1: Manual code modifications for testnet	Warning	1.0	Fixed
W2: Clippy warnings	Warning	1.0	Fixed
W3: Irrelevant CHECK comments	Warning	1.0	Fixed
W4: Use of Rust type system	Warning	1.0	Fixed
W5: Use appropriate error handling	Warning	1.0	Fixed
I1: Code style	Info	1.0	Fixed
L5: Unused stored bump	Low	2.0	Reported
I2: Difficulty calculation	Info	2.0	Reported

Table 2. Table of Findings

5. Report revision 1.0

5.1. System Overview

This section contains an outline of the audited contracts. Note that this is meant for understandability purposes and does not replace project documentation.

Actors

This part describes the actors of the system, their roles, and permissions.

Payer

An individual or entity that wishes to pay in Bitcoin but owns SOL or an arbitrary SPL token. An individual or entity that wishes to pay in SOL or an arbitrary SPL token but owns Bitcoin.

Receiver

An individual or entity that receives a payment from the payer (in Bitcoin, SOL or an arbitrary SPL token).

Intermediary node

A third party that provides liquidity for the transfer. Either SOL or an arbitrary SPL token when transferring from Bitcoin/Lightning Network to Solana or Bitcoin when transferring from Solana to Bitcoin/Lightning Network.

Initializer

Either the payer or the receiver on the Solana side initializing an escrow contract.

Offerer

The provider of the SOL/SPL tokens to lock into the escrow contract. When transferring from Solana to Bitcoin, the offerer is the payer and initializer of the contract. When transferring from Bitcoin to Solana, the offerer is the intermediary node.

Claimer

The recipient of the SOL/SPL token from the escrow contract. When transferring from Solana to Bitcoin, the claimer is the intermediary node. When transferring from Bitcoin to Solana, the claimer is the receiver and initializer of the contract.

Relayer

Submits Bitcoin headers to the BTCRelay-Sol program to enable Bitcoin transaction verifications.

Watchtower

Claims the tokens to the claimer's account, on the claimer's behalf.

Programs

SolLightning-program

The SolLightning-program provides the escrow contract functionality. There are three main instructions available - initialize an escrow contract, claim locked funds and refund locked funds. When transferring assets between Solana and Lightning Network, the program allows the creation of a hash-time locked contract (HTLC) that locks the assets of the offerer and allows the claimer to claim it once he proves the corresponding invoice was paid (by knowing the secret provided by the receiver). When transferring assets between Solana and Bitcoin, the program allows the creation of a proof-time locked contract (PTLC) that locks the assets of the offerer and allows the claimer to claim it once he proves the corresponding Bitcoin transaction was processed and confirmed by the Bitcoin blockchain using the BTCRelay-Sol program.

The program allows two types of refunds (or canceling of the escrow account):

1. Cooperative refund where the offerer and the claimer mutually agree to cancel the payment and close the escrow account before the account expiry.
2. Un-cooperative refund where the offerer can close the escrow account once the lock period has expired even without the cooperation with the claimer.

Finally, the program enables the intermediary nodes to deposit or withdraw (unlocked) assets from the program vault.

BTCRelay-Sol

On-chain program used to verify and store Bitcoin block headers on Solana. The program allows anyone to submit new block headers and new chain forks while automatically verifying the validity of the headers by checking the Bitcoin consensus rules. The purpose of this program is to provide a way to verify that a particular Bitcoin transaction was processed and confirmed by the main Bitcoin blockchain.

Process

A client (initializer) that wishes to pay in SOL/SPL or wishes to receive a payment in SOL/SPL initializes a Solana escrow account in collaboration with a particular intermediary node where both parties (the initializer and the intermediate node) must sign the transaction.

There are 6 different transfer possibilities: Transfer between Solana and Lightning Network, between Solana and Bitcoin using nonced transactions and finally between Solana and Bitcoin using single-use-seals transactions where each type of transfer has two directions (ie. Solana→Bitcoin and Bitcoin→Solana).

When transferring from Solana, the payer initializes the escrow account, the intermediary node observes the Solana log events and proceeds to the payment of the Lightning Network invoice or sends the corresponding Bitcoin transaction. The intermediary node proves that he performed the required payment (to the receiver) by providing either a secret provided by the receiver or verifying the Bitcoin transaction using the BTCRelay-Sol program, claims the assets from the escrow contract and the transfer is finished.

When transferring from Bitcoin, the receiver initializes the escrow account and locks the corresponding funds of an intermediate node in the contract. The payer must then proceed with the Lightning Network invoice payment or broadcast the corresponding Bitcoin transaction. The receiver then proves that the payment (to the intermediate node) was processed by providing the corresponding secret of the Lightning Network payment or by verifying the Bitcoin transaction using the BTCRelay-Sol program, claims the assets from the escrow contract and the transfer is finished.

5.2. Trust Model

The protocol is designed to be fully trustless. However, it requires an additional effort from the initializer of the contract. Especially in the case of transfers between Bitcoin and Solana where the correct function relies on the fact, that correct Bitcoin headers must be submitted to the BTCRelay-Sol program and that the transfer will be claimed on time. If these two conditions are not satisfied, the claimer or the offerer may lose their funds.

In theory, a user of the protocol can submit correct headers to the BTCRelay-Sol program (as anyone can do it) and also make sure transfers are claimed on time, however, it is highly impractical for a regular user. Therefore, the protocol relies on independent relayers (that will submit the headers to the BTCRealy-Sol program) and also watchtowers, that will claim the transfers on behalf of the claimer.

The intermediary nodes as well as the watchtowers are incentivized by receiving a fee from a transfer or a claim. As there is no financial motivation for the relayers, it is assumed that a relayer and watchtower will be one entity.

C1: Initializer can lose refundable fee

Critical severity issue

Impact:	High	Likelihood:	High
Target:	SolLightning-program/programs/test-anchor/src/lib.rs, SolLightning-program/programs/test-anchor/src/instructions.rs	Type:	Missing address check

Description

The instruction `offerer_refund` does not verify the correct address of the initializer. The refundable fee in case of a transfer from Bitcoin to Solana can thus be sent to any account resulting in a loss of funds for the initializer.

Exploit scenario

1. A transfer from Bitcoin to Solana is initialized using the `offerer_initialize` instruction and by creating the PTLC (proof-time locked contract) Solana contract. The initializer is the receiver on the Solana side.
2. The `claimer_bounty` is set to a greater value than the `security_deposit` during the initialization. The difference between the two is the refundable fee that the initializer should receive in case of failed payment.
3. In case no Bitcoin payment arrives and the PTLC expires, the intermediary node can use the `offerer_refund` instruction to refund his locked funds and close the contract.
4. After the PTLC expiration, the intermediary node does not need the cooperation of the receiver (initializer) to send the `offerer_refund`

instruction and can submit his own account to claim the refundable part of the fee thus stealing the funds of the receiver.

The correct address of the initializer is also not verified in the case of the cooperative PTLC cancelation. However in this case the receiver (initializer) must sign and approve the transaction while reducing the likelihood of setting an incorrect initializer account.

Recommendation

Verify that the submitted initializer account is equal to the claimer account set during the PTLC initialization.

Solution (Revision 2.0)

The original instruction `offerer_initialize` was split into two separate instructions: `offerer_refund` and `offerer_refund_pay_in`. Both instructions now verify that the address of the claimer (initializer) corresponds to the address saved in EscrowAccount: `escrow_state.claimer == *claimer.key`.

Go back to Findings Summary

C2: Transaction hash uniqueness not guaranteed

Critical severity issue

Impact:	High	Likelihood:	High
Target:	SolLightning-program/programs/test-anchor/src/lib.rs, SolLightning-program/programs/test-anchor/src/instructions.rs	Type:	Replay attack

Description

The uniqueness of the transaction hash is not verified and gives the sender the possibility to fraudulently claim locked funds from an escrow contract using a replay attack.

Exploit scenario

1. A transfer from Solana to Bitcoin is initialized using the `offerer_initialize` instruction and by creating the PTLC (proof-time locked contract) Solana contract. The initializer is the sender (offerer) on the Solana side.
2. The initializer is responsible for submitting a transaction with a unique hash and eventually unique nonce that is passed to the PTLC during the initialization.
3. The initializer and the intermediary node agree on the cooperative cancelation of the PTLC and the intermediary node sends a signed specific refund message to the initializer allowing him to claim his locked funds and close the contract.
4. The initializer claims back his locked funds and closes the contract.

5. The initializer initializes a new Solana to Bitcoin transfer with the same intermediary node using the same PTLC settings as in step 1. The uniqueness of the transaction hash is not verified allowing the initializer to submit the same initialize instruction.
6. The intermediary node observes the creation of the PTLC contract and sends the appropriate amount to the receiver's address.
7. The intermediary node must wait for a given number of BTC block confirmations before he can claim the locked funds from the PTLC.
8. The initializer however already has the signed specific refund message from the previous canceled payment from step 3 and uses this message to refund the locked funds from the PTLC thus stealing the funds belonging to the intermediary node.

Recommendation

Make sure that both the initializer and the intermediary node verify that a specific transaction hash has not been processed previously.

Solution (Revision 2.0)

A unique sequence number identifying the specific swap was added to the cooperative pre-signed refund message. This restricts the use of the pre-signed refund message to the specific swap without the possibility of a replay attack. It is important to note that the responsibility to generate a unique sequence number is on the side of the intermediary node.

Go back to Findings Summary

C3: Possibility to lock funds

Critical severity issue

Impact:	High	Likelihood:	High
Target:	SolLightning-program/programs/test-anchor/src/lib.rs, SolLightning-program/programs/test-anchor/src/instructions.rs	Type:	Missing account check

Description

In case of a transfer from BTC to Solana, the receiver on Solana can initialize a PTLC contract in a way that the intermediary node will be unable to refund his locked funds.

Exploit scenario

The receiver on Solana can initialize a PTLC contract using the `offerer_initialize` instruction with the pay_out option deactivated (pay_out=false). In case of failed payment, the intermediary invokes the refund instruction that requires passing a PDA UserAccount of the claimer (in this case of the receiver on Solana) in the remaining accounts.

Listing 1. Excerpt from SolLightning-program.offerer_refund

```
        if !ctx.accounts.escrow_state.pay_out {
            //Check the remainingAccounts
            let user_data_acc = &ctx.remaining_accounts[0];
            let (user_data_address, _user_data_bump) =
                Pubkey::find_program_address(&[USER_DATA_SEED,
ctx.accounts.escrow_state.claimer.as_ref(),
ctx.accounts.escrow_state.mint.as_ref()], ctx.program_id);
```

```

        require!(
            user_data_address==*user_data_acc.key,
            SwapErrorCode::InvalidUserData
        );

        require!(
            user_data_acc.is_writable,
            SwapErrorCode::InvalidUserData
        );

        let mut data = user_data_acc.try_borrow_mut_data()?;
        let mut user_data = UserAccount::try_deserialize(&mut &**data)?;
```

If this PDA account does not exist, the intermediary node has no way to create it and will be unable to refund his locked assets. As the existence of the claimer (receiver) UserAccount PDA is not verified during the initialization, the receiver can therefore lock an arbitrary amount of assets of the intermediary node for the cost of a single transaction.

Recommendation

Verify, that the UserAccount PDA of the receiver exists during the escrow contract initialization.

Solution (Revision 2.0)

The instruction `offerer_initialize` now requires submission of the correct user data account of the claimer in case the `pay_out` option is set to false.

Go back to Findings Summary

M1: Artificial reputation increase

Medium severity issue

Impact:	Low	Likelihood:	High
Target:	SolLightning-program/programs/test-anchor/src/lib.rs	Type:	Code logic

Description

An intermediary node can artificially increase its reputation (number and volume of successful payments) by playing simultaneously the role of intermediary node and either a sender or a receiver. The reputation is updated in the `claimer_claim` instruction where the party invoking this instruction must know the secret in case of HTLC (hash-time locked contract) or must prove a Bitcoin transaction was finalized in case of PTLC (proof-time locked contract). If an intermediary node initializes a payment while knowing the secret or reusing an already finalized Bitcoin transaction, it allows it to directly claim the funds from the escrow account and increase the count and volume of successful transfers without actually sending the funds to a third party. The intermediary must only pay transaction fees that are very cheap on Solana and can thus at a low price falsify and artificially increase its reputation.

Solution (Revision 2.0)

The finding was acknowledged by the client with the following comment:

> Agreed, the success reputation can be easily gamed for almost zero cost, this will be taken into consideration when using the reputation data by the off-chain intermediary node discovery

> process.
>
> — SolLightning Team

Go back to Findings Summary

M2: Number of confirmations not limited

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	SolLightning-program/programs/test-anchor/src/lib.rs	Type:	Denial of Service

Description

To claim funds locked in a PTLC (proof-time locked contract), the claimer must prove a BTC transaction was confirmed by at least a given number of mined blocks in the main chain using the BTCrelay program. However, the BTCrelay program supports only a limited number of confirmations defined by the pruning factor (currently 250). If a PTLC is initialized with more confirmations required than the currently set pruning factor, the claimer won't be able to prove the BTC transfer and claim his funds.

Recommendation

Limit the number of confirmations during the PTLC initialization to the pruning factor reduced by a safety margin.

Solution (Revision 2.0)

The number of confirmations has been limited to the pruning factor reduced by a safety margin, resulting currently in 200 confirmations at most.

Go back to Findings Summary

M3: Unchecked token account

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	SolLightning-program/programs/test-anchor/src/instructions.rs	Type:	Missing account check

Description

When initializing a new escrow contract using `offerer_initialize_pay_in` or `offerer_initialize` instruction with the pay_out option activated, the account `claimer_token_account` is not verified as a Token account with the correct mint. If accidentally a wrong account is passed, the claimer won't be able to claim his tokens.

Recommendation

Verify that the `claimer_token_account` is a token account with the correct mint.

Solution (Revision 2.0)

Both instructions `offerer_initialize_pay_in` and `offerer_initialize` now correctly verify that the `claimer_token_account` is indeed a TokenAccount with the correct mint when the option pay_out is set to true.

Go back to Findings Summary

M4: Multiple lock-times

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	SolLightning-program/programs/test-anchor/src/lib.rs	Type:	Protocol design

Description

Both HTLC and PTLC escrow contracts use an expiry time after which the initializer can refund his locked assets. If the expiry time is set incorrectly, the claimer might lose his funds.

Exploit scenario

When transferring from Solana to Bitcoin, the intermediate node must verify that the time necessary to prove the Bitcoin transaction was confirmed is less than the expiry time of the PTLC contract. Otherwise, the payer can refund his locked assets before the intermediary node can claim it resulting in a loss of funds of the intermediary.

Conversely, when transferring from Bitcoin to Solana, the receiver must set the PTLC expiry time further in the future than the minimal time needed to confirm the Bitcoin payment transaction. Otherwise, the intermediary node can refund his locked assets before the receiver can claim it resulting in a loss of funds of the receiver.

Finally, when transferring from Solana to Lightning Network, the intermediary node must set the lock-time of its Lightning payment less in the future than the HTLC expiry time. Otherwise, in case the receiver does not settle the payment, the payer can refund his locked assets before the Lightning payment expires. If the receiver finally settles the payment, the intermediate node won't be able to claim his assets from the HTLC anymore resulting in loss of funds of the intermediary node.

Recommendation

When initializing a new PTLC, verify that the time needed for the Bitcoin transaction verification is less than the time needed for the contract expiry. Add a safety margin to account for unknown delays.

Solution (Revision 2.0)

The finding was acknowledged by the client with the following comment:

> This will be handled off-chain during negotiation between intermediary node & client as it can be dynamic and every intermediary node operator should be able to set a safety margin desired by them (this can also be based on the current fee situation on bitcoin, block time differences due to hash rate spikes, etc.), this is already how the Lightning network works where every node can set its own CLTV delta when forwarding lightning payments, CSV delay for possible force closes and min_final_cltv_expiry_delta for bolt11 invoices.
>
> — SolLightning Team

Go back to Findings Summary

M5: Reputation can overflow

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	SolLightning-program/programs/test-anchor/src/lib.rs	Type:	Denial of Service

Description

The intermediate node reputation (count and volume of successful or failed transfers) is updated using unchecked arithmetics that can overflow. In such a case, it will be impossible to claim or refund assets from the escrow contract, because the program will panic on overflow and the funds will be stuck.

Listing 2. Excerpt from SolLightning-program.claimer_claim

```
            //Pay out to internal wallet
            let user_data = ctx.accounts.user_data.as_mut().unwrap();
            user_data.amount += ctx.accounts.escrow_state.initializer_amount;
            user_data.success_volume[usize::from(ctx.accounts.escrow_state.kind)] += ctx.accounts.escrow_state.initializer_amount;
            user_data.success_count[usize::from(ctx.accounts.escrow_state.kind)] += 1;
```

Listing 3. Excerpt from SolLightning-program.offerer_refund

```
            if auth_expiry>0 {
                user_data.coop_close_volume[usize::from(ctx.accounts.escrow_state.kind)] += ctx.accounts.escrow_state.initializer_amount;
                user_data.coop_close_count[usize::from(ctx.accounts.escrow_state.kind)] += 1;
            } else {
                user_data.fail_volume[
    usize::from(ctx.accounts.escrow_state.kind)] +=
    ctx.accounts.escrow_state.initializer_amount;
                user_data.fail_count[
    usize::from(ctx.accounts.escrow_state.kind)] += 1;
            }
```

Recommendation

Use the `saturating_add()` method to update the values.

Solution (Revision 2.0)

The method `saturating_add()` to avoid overflow is now used to update the values related to the volume of successful or failed transfers. The values related to the number of successful or failed transfers are still updated using the unchecked arithmetic. However, the overflow will only happen after 2^64 transfers (successful or failed separately), and the likelihood of this happening is extremely low in a real use case.

Go back to Findings Summary

L1: Passing Rent SysVar

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	SolLightning-program/programs/test-anchor/src/instructions.rs	Type:	Transaction cost

Description

Passing the Rent SysVar is not necessary.

Recommendation

Use `Rent::get()?` and avoid passing Rent SysVar to reduce transaction size.

Solution (Revision 2.0)

The Rent SysVar account was removed from all instructions.

Go back to Findings Summary

L2: Unused instruction parameters

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	SolLightning-program/programs/test-anchor/src/instructions.rs, BTCRelay-Sol/programs/btc-relay/src/instruction.rs	Type:	Transaction cost

Description

The SolLightning-program instruction parameters:

- `kind: u8`
- `confirmations: u16`
- `auth_expiry: u64`
- `escrow_nonce: u64`
- `pay_out: bool`
- `txo_hash: [u8; 32]`

are listed in the `#[instruction]` macro of both `InitializePayIn<'info>` and `Initialize<'info>` context structures, however, these parameters are not used during the context struct validation and are therefore not needed.

The BTCRelay-Sol instruction parameters:

- `block_height: u32,`
- `chain_work: [u8; 32],`
- `last_diff_adjustment: u32,`

- `prev_block_timestamps: [u32; 10]`

are listed in the `#[instruction]` macro of the `Initialize<'info>` context structure, however, these parameters are not used during the context struct validation and are therefore not needed.

The BTCRelay-Sol instruction parameter:

- `init: bool`

is listed in the `#[instruction]` macro of the `SubmitForkHeaders<'info>` context structure, however, the parameter is not used during the context struct validation and is therefore not needed.

Recommendation

Remove the unused instruction parameters.

Solution (Revision 2.0)

The unused instruction parameters were removed.

Go back to Findings Summary

L3: Bump calculation

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	SolLightning-program/programs/test-anchor/src/instructions.rs	Type:	Transaction cost

Description

When passing an existing PDA to an instruction, the bump macro attribute is not assigned to a value. For example in (but not limited to) this code snippet:

Listing 4. Excerpt from [SolLightning-program.](SolLightning-program)

```
    #[account(
        mut,
        seeds = [USER_DATA_SEED.as_ref(), claimer.key.as_ref(),
mint.to_account_info().key.as_ref()],
        bump
    )]
    pub user_data: Account<'info, UserAccount>,
```

This has for effect that the program has to calculate the canonical bump every time the instruction is invoked.

Also, it is not necessary to again call the `find_program_address` function within the instruction to get the bump value.

Recommendation

Save the on-chain calculated canonical bump in the given PDA state and compare the passed bump with this trusted bump value (such as `bump = state.bump`) to save the compute budget. To get the bump from within an instruction, use `let bump = ctx.bumps.get("<ACCOUNT_NAME>").ok_or(<ERROR>);` (for Anchor 0.28.0 and lower, see Anchor documentation for higher versions).

Solution (Revision 2.0)

A new `bump` field was added to the `UserAccount` structure to store the bump value, and this value is correctly used to avoid unnecessary bump re-calculations.

For other account types, this design pattern was not used. However, we evaluated it as a design choice as it would not bring substantial benefits.

Go back to Findings Summary

L4: Account does not need to be mutable

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	BTCRelay-Sol/programs/btc-relay/src/instructions.rs	Type:	Runtime optimization

Description

The `main_state` account in the context structures `BlockHeight<'info>` and `VerifyTransaction<'info>` does not need to be mutable.

Recommendation

Remove the `mut` macro attribute.

Solution (Revision 2.0)

The unnecessary `mut` attributes were removed.

Go back to Findings Summary

W1: Manual code modifications for testnet

Impact:	Warning	Likelihood:	N/A
Target:	BTCRelay-Sol/programs/btc-relay/src/utils.rs	Type:	Code smell

Description

Part of the BTCRelay code has to be manually commented out for testnet.

Listing 5. Excerpt from [BTCRelay-Sol.verify_header]()

```rust
pub fn verify_header(header: &BlockHeader, last_commited_header: &mut
CommittedBlockHeader, remaining_account: &AccountInfo, _signer: &Signer,
program_id: &Pubkey) -> Result<[u8; 32]> {

    //Correct difficulty target
    //
    //Should be disabled for testnet, since if no valid block is
    // found on testnet in 20 minutes, the difficulty drops to 1
    //Implementing this functionality is beyond scope of this
implementation,
    // so nBits checking is disabled for TESTNET,
    // comment the following code block to make this compatible with TESTNET
    //
    // !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    // !!!!!!!!!!!!!!  DISABLE FOR TESTNET  !!!!!!!!!!!!!!
    // !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    {
        require!(
            has_correct_difficulty_target(*last_commited_header,
header.nbits),
            RelayErrorCode::ErrDiffTarget
        );
    }
```

Recommendation

Any manual code modifications should be avoided and should be done programmatically. Use a compile-time feature flag to deactivate code for testnet.

Solution (Revision 2.0)

A new cargo feature, `bitcoin_testnet`, was added to deactivate parts of code not relevant to testnet.

Go back to Findings Summary

W2: Clippy warnings

Impact:	Warning	Likelihood:	N/A
Target:	`**/*`	Type:	Code smell

Description

The `SolLightning-program` contains 67 clippy warnings (with default settings). The `BTCRelay-Sol` contains 118 clippy warnings (with default settings).

Clippy helps the program to be efficient and readable for other developers.

Recommendation

Correct the warnings or explicitly ignore individual warnings where reasonable.

Go back to Findings Summary

W3: Irrelevant CHECK comments

Impact:	Warning	Likelihood:	N/A
Target:	`**/*`	Type:	Code smell

Description

When using an unchecked account with Anchor, the programmer is required to provide a documentation comment. The comment is supposed to explain why it is not dangerous to use an unchecked account to prevent missing account verification.

The codebase uses however the same comment for all unchecked accounts (and in some cases also for checked accounts) with an irrelevant or incomplete explanation:

```
/// CHECK: This is not dangerous because we don’t read or write from this
account
```

Recommendation

Comment all the unchecked accounts appropriately and remove the `CHECK` comments where not required.

Go back to Findings Summary

W4: Use of Rust type system

Impact:	Warning	Likelihood:	N/A
Target:	`**/*`	Type:	Code smell

Description

An important strength of Rust programming language is its strong type checking system. The codebase does not fully utilize this powerfull and security oriented feature that also (in most cases) increases the maintainability and readability of the program. Some examples are listed below.

Recommendation

Listing 6. Excerpt from SolLightning-program.

```
pub static KIND_LN: u8 = 0;
pub static KIND_CHAIN: u8 = 1;
pub static KIND_CHAIN_NONCED: u8 = 2;
pub static KIND_CHAIN_TXHASH: u8 = 3;
```

Use an enumeration for the chain kind instead of static constants.

Listing 7. Excerpt from SolLightning-program.

```
    pub fn verify_tx_ix(ix: &Instruction, reversed_tx_id: &[u8; 32],
confirmations: u32) -> Result<u8> {
        let btc_relay_id: Pubkey =
Pubkey::from_str(BTC_RELAY_ID_BASE58).unwrap();

        if  ix.program_id       != btc_relay_id
        {
            return Ok(10);
        }

        return Ok(check_tx_data(&ix.data, reversed_tx_id, confirmations));
    }

    // Verify serialized BtcRelay instruction data
    pub fn check_tx_data(data: &[u8], reversed_tx_id: &[u8; 32],
confirmations: u32) -> u8 {
        for i in 0..8 {
            if data[i] != TX_VERIFY_IX_PREFIX[i] {
                return 1;
            }
        }
        for i in 8..40 {
            if data[i] != reversed_tx_id[i-8] {
                return 2;
            }
        }

        let _confirmations = u32::from_le_bytes(data[40..
44].try_into().unwrap());
        if confirmations != _confirmations {
            return 3;
        }

        return 0;
    }
```

The function `verify_tx_ix` return type is `Result<u8>`, however the function always returns only the `Ok(u8)` variant of the result where the `u8` number actualy represents an error code which defies the purpose of the `Result<u8>` type. Define a new error enumeration (to replace the hard coded error codes), change the `check_tx_data` function signature to return a Result. Always return the `Err() variant to express an error and the Ok()` variant to express success. Use the `?` syntax to propagate the error results. Finally handling the result using a match expression will guarantee that all possible errors are handled.

The same is relevant also for functions `verify_blockheight_ix` and `check_blockheight_data`.

Solution (Revision 2.0)

More Rust idiomatic and type-oriented approach was adopted throughout both programs.

Consider also adding `#[repr(u8)]` before the definitions of fieldless enums to guarantee the expected representation.

Go back to Findings Summary

W5: Use appropriate error handling

Impact:	Warning	Likelihood:	N/A
Target:	`**/*`	Type:	Code smell

Description

There are multiple situations when the input parameters are not checked properly and the program will panic instead of returning a proper error result. For example, instructions expecting additional remaining accounts do not handle the situation, when none or not enough remaining accounts are supplied. The program will then panic with an "array out of bounds" message.

Listing 8. Excerpt from BTCRelay-Sol.submit_fork_headers

```
                //Has to use new fork_id from the fork_counter
                require!(
                    main_state.fork_counter == fork_id,
                    RelayErrorCode::NoHeaders
                );
```

The error `RelayErrorCode::NoHeaders` above is not relevant.

Recommendation

Avoid program panicking and explicitly handle edge case situations to return an appropriate error result. This improves code quality, debugging and user experience.

Solution (Revision 2.0)

Numerous new error variants were added, and both programs use Result type variables appropriately. Both programs still use the `unwrap()` method, which might panic, however, we consider the current implementation acceptable in terms of readability balance.

Go back to Findings Summary

I1: Code style

Impact:	Info	Likelihood:	N/A
Target:	SolLightning-program/programs/test-anchor/src/*.rs	Type:	Code smell

Description

The program contains commented parts of code such as the complete `claimer_refund_payer` function or parts of code in the `check_claim` function. The name of the program is `anchor-test` instead of `SolLightning`.

Recommendation

Delete the commented code and rename the program.

Solution (Revision 2.0)

The program was renamed to `swap-program`, and most of the commented code was removed.

Go back to Findings Summary

6. Report revision 2.0

6.1. System Overview

Revision 2.0 did not introduce any new features or changes to the trust model. All changes are related to the implementation details of the protocol. You can, therefore, refer to the system overview and trust model description in Revision 1.0 section.

L5: Unused stored bump

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	SolLightning-program/programs/swap-program/src/instructions.rs	Type:	Code smell

Description

A new `bump` field in the `EscrowState` structure was introduced as part of the fix of issue L3. However, this field is never used.

Recommendation

Remove the `bump` field from the `EscrowState` structure.

Go back to Findings Summary

I2: Difficulty calculation

Impact:	Info	Likelihood:	N/A
Target:	BTCRelay-Sol/programs/btc-relay/src/utils.rs	Type:	Optimization

Description

The cumulative work spent on finding a hash equal to or less than the target value is approximated as cumulative difficulty where difficulty is based on the following formula:

`difficulty = max_difficulty_target / target`

where the `max_difficulty_target` is equal to `(2^16 -1) * 2^208`.

This formula is used for scaling and representation reasons. The downside is that to achieve the minimal difficulty 1 that will result in a provable work (difficulty must be greater than 0) requires the target to be equal to the `max_difficulty_target`, meaning it will be required to calculate approximately `2^(256-16-208) = 2^32` hashes.

We did not find any way, how this fact could be maliciously exploited, however, the used approximation is impractical for testing purposes, as the target value cannot be increased to reduce the necessary number of calculated hashes.

Recommendation

Use the same formula as in Bitcoin core implementation to calculate the work:

```
arith_uint256 GetBlockProof(const CBlockIndex& block)
{
    arith_uint256 bnTarget;
    bool fNegative;
    bool fOverflow;
    bnTarget.SetCompact(block.nBits, &fNegative, &fOverflow);
    if (fNegative || fOverflow || bnTarget == 0)
        return 0;
    // We need to compute 2**256 / (bnTarget+1), but we can't represent 2**256
    // as it's too large for an arith_uint256. However, as 2**256 is at least as
large
    // as bnTarget+1, it is equal to ((2**256 - bnTarget - 1) / (bnTarget+1)) +
1,
    // or ~bnTarget / (bnTarget+1) + 1.
    return (~bnTarget / (bnTarget + 1)) + 1;
}
```

Take into account possible corner cases to prevent overflow or invalid results and write specific unit tests.

Go back to Findings Summary

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain, SolLightning: Security Audit, 12.1.2024.

Thank You

Ackee Blockchain a.s.

Prague, Czech Republic

hello@ackeeblockchain.com

https://twitter.com/AckeeBlockchain