Parallel Finance

Security Assessment

March 15, 2022

Prepared for:
Yubo Ruan
Parallel Finance

Prepared by:
Artur Cygan
Will Song
Fredrik Dahlgren

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 80+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

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

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

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

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

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

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

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

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

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

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

Table of Contents

Executive Summary

Engagement Overview

Parallel Finance engaged Trail of Bits to review the security of its Substrate parachain. From February 22 to March 11, 2022, a team of three consultants conducted a security review of the client-provided source code, with six person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the target system, including access to the source code and documentation. We performed static and dynamic testing of the target system and its codebase, using both automated and manual processes.

Summary of Findings

The audit did not uncover any significant flaws or defects. A summary of the findings and details on notable findings are provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	2
Low	3
Informational	4
Undetermined	2

CATEGORY BREAKDOWN

Category	*Count*
Auditing and Logging	1
Data Validation	7
Patching	2
Undefined Behavior	1

Notable Findings

The following is a significant flaw that impacts system confidentiality, integrity, or availability.

- **TOB-PLF-2**
  The accrued interest is reset whenever a loan is updated by the `loans` pallet. This allows malicious users to avoid paying interest on loans by repaying a zero amount whenever interest is accrued by the system.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Mary O'Brien, Project Manager
mary.obrien@trailofbits.com

The following engineers were associated with this project:

Artur Cygan, Consultant
artur.cygan@trailofbits.com

Will Song, Consultant
will.song@trailofbits.com

Fredrik Dahlgren, Consultant
fredrik.dahlgren@trailofbits.com

Project Timeline

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

Date	Event
February 22, 2022	Pre-project kickoff call
February 28, 2022	Status update meeting #1
March 7, 2022	Status update meeting #2
March 15, 2022	Delivery of final report draft and final readout
March 31, 2022	Delivery of final report

Project Goals

The engagement was scoped to provide a security assessment of the Parallel Finance parachain pallets. Specifically, we sought to answer the following non-exhaustive list of questions:

- Do all pallets validate extrinsics correctly?
- Are authorization controls for privileged extrinsics implemented properly?
- Are extrinsic weights calculated properly, based on worst-case running times?
- Is it possible to steal funds by overflowing or underflowing balance calculations?
- Is it possible for a malicious relayer to exercise undue influence on the Ethereum bridging protocol?
- Is it possible to track state machine transitions using events emitted by the system?
- Does the state stay consistent after extrinsics are called in any possible sequence?
- Is it possible for the system to use incorrect market prices delivered by oracles?
- Is it possible to crash the system with malicious messages?

Project Targets

The engagement involved a review and testing of the targets listed below.

Parallel

Repository	https://github.com/parallel-finance/parallel
Versions	`5ca8e13b7b4312855ae2ef1d39f14b38088dfdbd`, `5ae9e860f7ba9737f760d24accec6b8ed417faea`
Type	Rust
Platform	Native/Unix

Parallel Token Bridge

Repository	https://github.com/parallel-finance/parallel-token-bridge
Version	`16cff0ca277f2af0cce91fcb01c8d021c782a1b2`
Types	TypeScript, Solidity
Platforms	Node.js, Ethereum

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches and their results include the following:

- **Parallel repository.** We ran Clippy and Dylint to identify code quality issues and unidiomatic Rust code. We also ran `cargo-audit` to identify known vulnerable project dependencies. Finally, we ran Tarpaulin to assess the unit test coverage of the project.

- **Loans pallet.** We manually reviewed all extrinsics to ensure that they perform the necessary validation, including the relevant bounds and signature checks. We also ensured that privileged extrinsics are not exposed to ordinary users of the system. We reviewed the arithmetic related to amount conversions and interest calculations to ensure that the calculations are correctly implemented and cannot overflow or underflow. We also ensured that the correct amounts are transferred to the correct accounts when funds are minted/redeemed and borrowed/repaid. Finally, we reviewed the pallet implementation for arbitrage opportunities.

- **Bridge pallet.** We manually reviewed the extrinsic implementations to ensure that each extrinsic performs the necessary input and signature validation. We also made sure that privileged extrinsics are not exposed to ordinary users of the system. We reviewed the `materialize` voting mechanism implemented by the pallet. In particular, we ensured that vote resolution is implemented correctly and that a single relayer cannot vote multiple times. We ensured that the correct amounts are transferred to the correct accounts for both internal and external transfers, and we reviewed the arithmetic to ensure that it is not possible to overflow or underflow the amounts transferred by the bridge. Finally, we ensured that the assumptions on the voting threshold cannot be invalidated by adding relay nodes to the bridge.

- **Parallel token bridge repository.** We ran `eslint` and Semgrep to identify code quality issues and known vulnerable code patterns. We also performed a high-level manual review of the Ethereum and Parallel services in order to understand how the relayer interacts with the bridge pallet.

- **Crowdloans pallet.** We manually reviewed the extrinsic implementations to ensure that each extrinsic performs the necessary input and signature validation. We also made sure that privileged extrinsics are not exposed to ordinary users of the system. Finally, we ensured that the vault lifecycle is properly managed, and we reviewed the calculations and state consistency.

- **Liquid staking pallet.** We manually reviewed the extrinsic implementations to ensure that each extrinsic performs the necessary input and signature validation.

We reviewed all privileged extrinsics to ensure that they are not exposed to ordinary users of the system. We also reviewed all arithmetic and all interactions with the relay chain.

- **Automatic market maker pallet.** We manually reviewed the extrinsics defined by the pallet to ensure that they perform the necessary input and signature validation. We also ensured that privileged extrinsics are exposed only to privileged origins. Finally, we reviewed all arithmetic calculations in the pallet for logic errors and overflows and underflows.

- **XCM helper pallet.** We manually reviewed the cross-consensus messaging (XCM) helpers implemented by the XCM helper pallet. Here, we focused mainly on implementation correctness, but we also looked for issues related to race conditions and out-of-order messages.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following describes the coverage limitations that we encountered during this audit:

- We performed only a high-level review of the relayer implementation in the Parallel token bridge repository. In particular, we did not review the Ethereum smart contract used by the bridge.

- While we reviewed the liquid staking pallet, its logic is complex and changed throughout the audit. This component requires further review, especially around XCM.

- We did not review other off-chain components, such as oracles that feed the market prices to the pallets.

- Although we ran a number of static analysis tools on the whole Parallel repository, we did not have time to manually review the router and farming pallets.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We selected the following tools for use in the automated testing phase of this project:

Tool	Description	Policy
`cargo-audit`	An open-source static analysis tool used to audit `Cargo.lock` files for crates with security vulnerabilities reported to the RustSec Advisory Database	Appendix D
Clippy	An open-source Rust linter used to catch common mistakes and unidiomatic Rust code	Appendix D
`eslint`	An open-source, pluggable JavaScript and TypeScript linter that analyzes the codebase to find potential problems	Appendix D
`yarn audit`	A `yarn` command that submits a description of the dependencies configured in a project to the default registry and requests a report of known vulnerabilities	Appendix D
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time	Appendix D
Tarpaulin	An open-source code coverage reporting tool that can be integrated with the Cargo build system on x86 Linux architectures	Appendix D

Areas of Focus

Our automated testing and verification focused on the following:

- Identification of general code quality issues and unidiomatic code patterns
- Identification of known vulnerable dependencies
- Identification of poor unit and integration test coverage

Test Results

The results of this focused testing are detailed below.

Parallel repository. The Substrate node codebase implements the Parallel Finance parachain node.

Property	Tool	Result
The project does not import vulnerable dependencies.	`cargo-audit`	TOB-PLN-1
The project adheres to Rust best practices by fixing code quality issues reported by linters like Clippy.	Clippy	**Passed**
All components of the codebase have sufficient test coverage.	Tarpaulin	Appendix D

Parallel token bridge repository. The token bridge repository implements the off-chain components of the Parallel-Ethereum token bridge.

Property	Tool	Result
The project does not import vulnerable dependencies.	`yarn audit`	**Passed**
The project adheres to TypeScript best practices by fixing code quality issues reported by linters like `eslint`.	`eslint`	**Passed**
The project does not contain common vulnerable code patterns.	Semgrep	**Passed**

Codebase Maturity Evaluation

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

Category	Summary	Result
Arithmetic	The system exclusively relies on integer arithmetic, preventing issues related to rounding errors. Overall, arithmetic expressions are defensively written, and all reviewed pallets use checked or saturating arithmetic to handle integer overflows and underflows gracefully. During our review, we identified only one location (TOB-PLF-4) in which integer arithmetic could be strengthened to prevent associated issues.	Strong
Auditing	All calls to extrinsics emit appropriate events that make it easier to track all on-chain occurrences. However, we found a minor issue regarding an unnecessary event (TOB-PLF-8).	Satisfactory
Authentication / Access Controls	We did not find any access control issues. All extrinsics validate signatures correctly, and all privileged extrinsics ensure that the origin represents a privileged account.	Strong
Complexity Management	The code is well structured and adheres to the conventions of the Substrate framework. However, although common functionality is usually extracted into separate functions, we found some places with duplicate logic (TOB-PLF-11).	Satisfactory
Cryptography and Key Management	The Substrate framework makes it generally easy to generate and validate signatures for all transactions, and we did not identify any issues related to signature validation in particular or to cryptography in general.	Strong

Data Handling	Most data is validated and is of the appropriate type; however, the majority of the findings resulting from this audit concern data validation issues, one of which is of medium severity (TOB-PLF-2).	**Moderate**
Documentation	The code is documented sparingly, and the main README file contains limited information. The pallets do not have README files, which would be useful, given the system's domain complexity.	**Moderate**
Memory Safety and Error Handling	The project does not use unsafe Rust and does not interface with C code. The system's error handling is robust and follows standard Rust conventions. We did not find any locations in which the code could panic, either explicitly or implicitly.	**Strong**
Testing and Verification	The unit test coverage across the entire Parallel repository is 46% (as reported by Tarpaulin), which is insufficient for a codebase of this size and complexity. There are no property tests or fuzz tests implemented, and tests for unexpected or malicious behavior are insufficient.	**Moderate**

Summary of Findings

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

ID	Title	Type	Severity
1	Substrate parachain has vulnerable dependencies	Patching	**Medium**
2	Repaying a zero amount allows users to avoid accruing interest	Data Validation	**Medium**
3	Missing validation in Pallet::force_update_market	Data Validation	**Informational**
4	Missing validation in multiple StakingLedger methods	Data Validation	**Undetermined**
5	Failed XCM requests are left in storage	Data Validation	**Low**
6	Risk of using stale oracle prices in loans pallet	Data Validation	**Low**
7	Missing calculations in crowdloans extrinsics	Undefined behavior	**Undetermined**
8	Event emitted when update_vault and set_vrf calls do not make updates	Auditing and Logging	**Informational**
9	The referral code is a sequence of arbitrary bytes	Data Validation	**Informational**
10	The referral code size is not validated	Data Validation	**Low**
11	Code duplication in crowdloans pallet	Patching	**Informational**

Detailed Findings

1. Vulnerable dependencies in the Substrate parachain	
Severity: **Medium**	Difficulty: **High**
Type: Patching	Finding ID: TOB-PLF-1
Target: `parallel` repository	

Description
The Parallel Finance parachain node uses the following dependencies with known vulnerabilities. (All of the dependencies listed are inherited from the Substrate framework.)

Dependency	Version	ID	Description
`chrono`	0.4.19	RUSTSEC-2020-0159	Potential segfault in `localtime_r` invocations
`lru`	0.6.6	RUSTSEC-2021-0130	Use after free in `lru` crate
`time`	0.1.44	RUSTSEC-2020-0071	Potential segfault in the `time` crate
`net2`	0.2.37	RUSTSEC-2020-0016	`net2` crate has been deprecated; use `socket2` instead

Other than `chrono`, all the dependencies can simply be updated to their newest versions to fix the vulnerabilities. The `chrono` crate issue has not been mitigated and remains problematic. A specific sequence of calls must occur to trigger the vulnerability, which is discussed in this GitHub thread in the `chrono` repository.

Exploit Scenario
An attacker exploits a known vulnerability in the Parallel Finance node and performs a denial-of-service attack on the network by taking down all nodes in the network.

Recommendations
Short term, update all dependencies to their newest versions. Monitor the referenced GitHub thread regarding the `chrono` crate segfault issue.

Long term, run `cargo-audit` as part of the CI/CD pipeline and ensure that the team is alerted to any vulnerable dependencies that are detected.

2. Users can avoid accruing interest by repaying a zero amount

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-PLF-2
Target: `pallets/loans/src/lib.rs`	

Description

To repay borrowed funds, users call the `repay_borrow` extrinsic. The extrinsic implementation calls the `Pallet::repay_borrow_internal` method to recompute the loan balance. `Pallet::repay_borrow_internal` updates the loan balance for the account and resets the borrow index as part of the calculation.

```
    fn repay_borrow_internal(
        borrower: &T::AccountId,
        asset_id: AssetIdOf<T>,
        account_borrows: BalanceOf<T>,
        repay_amount: BalanceOf<T>,
    ) -> DispatchResult {
        // ... <redacted>

        AccountBorrows::<T>::insert(
            asset_id,
            borrower,
            BorrowSnapshot {
                principal: account_borrows_new,
                borrow_index: Self::borrow_index(asset_id),
            },
        );

        TotalBorrows::<T>::insert(asset_id, total_borrows_new);

        Ok(())
    }
```

Figure 2.1: `pallets/loans/src/lib.rs:1057-1087`

The borrow index is used in the calculation of the accumulated interest for the loan in `Pallet::current_balance_from_snapshot`. Specifically, the outstanding balance, `snapshot.principal`, is multiplied by the quotient of `borrow_index` divided by `snapshot.borrow_index`.

```
    pub fn current_balance_from_snapshot(
        asset_id: AssetIdOf<T>,
        snapshot: BorrowSnapshot<BalanceOf<T>>,
    ) -> Result<BalanceOf<T>, DispatchError> {
        if snapshot.principal.is_zero() || snapshot.borrow_index.is_zero() {
            return Ok(Zero::zero());
        }
        // Calculate new borrow balance using the interest index:
        // recent_borrow_balance = snapshot.principal * borrow_index /
        // snapshot.borrow_index
        let recent_borrow_balance = Self::borrow_index(asset_id)
            .checked_div(&snapshot.borrow_index)
            .and_then(|r| r.checked_mul_int(snapshot.principal))
            .ok_or(ArithmeticError::Overflow)?;

        Ok(recent_borrow_balance)
    }
```

Figure 2.2: `pallets/loans/src/lib.rs:1106-1121`

Therefore, if the snapshot borrow index is updated to `Self::borrow_index(asset_id)`, the resulting `recent_borrow_balance` in `Pallet::current_balance_from_snapshot` will always be equal to `snapshot.principal`. That is, no interest will be applied to the loan. It follows that the accrued interest is lost whenever part of the loan is repaid. In an extreme case, if the repaid amount passed to `repay_borrow` is `0`, users could reset the borrow index without repaying anything.

The same issue is present in the implementations of the `liquidated_transfer` and `borrow` extrinsics as well.

Exploit Scenario

A malicious user borrows assets from Parallel Finance and calls `repay_borrow` with a `repay_amount` of zero. This allows her to avoid paying interest on the loan.

Recommendations

Short term, modify the code so that the accrued interest is added to the snapshot principal when the snapshot is updated.

Long term, add unit tests for edge cases (like repaying a zero amount) to increase the chances of discovering unexpected system behavior.

3. Missing validation in Pallet::force_update_market

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PLF-3
Target: `pallets/loans/src/lib.rs`	

Description

The `Pallet::force_update_market` method can be used to replace the stored market instance for a given asset. Other methods used to update market parameters perform extensive validation of the market parameters, but `force_update_market` checks only the rate model.

```
    pub fn force_update_market(
        origin: OriginFor<T>,
        asset_id: AssetIdOf<T>,
        market: Market<BalanceOf<T>>,
    ) -> DispatchResultWithPostInfo {
        T::UpdateOrigin::ensure_origin(origin)?;
        ensure!(
            market.rate_model.check_model(),
            Error::<T>::InvalidRateModelParam
        );
        let updated_market = Self::mutate_market(asset_id, |stored_market| {
            *stored_market = market;
            stored_market.clone()
        })?;

        Self::deposit_event(Event::<T>::UpdatedMarket(updated_market));
        Ok(().into())
    }
```

Figure 3.1: `pallets/loans/src/lib.rs:539-556`

This means that the caller (who is either the root account or half of the general council) could inadvertently change immutable market parameters like `ptoken_id` by mistake.

Exploit Scenario

The root account calls `force_update_market` to update a set of market parameters. By mistake, the `ptoken_id` market parameter is updated, which means that `Pallet::ptoken_id` and `Pallet::underlying_id` are no longer inverses.

Recommendations

Short term, consider adding more input validation to the `force_update_market` extrinsic. In particular, it may make sense to ensure that the `ptoken_id` market parameter has not changed. Alternatively, add validation to check whether the `ptoken_id` market parameter is updated and to update the `UnderlyingAssetId` map to ensure that the value matches the `Markets` storage map.

4. Missing validation in multiple StakingLedger methods

Severity: **Undetermined**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PLF-4
Target: `pallets/liquid-staking/src/types.rs`	

Description

The staking ledger is used to keep track of the total amount of staked funds in the system. It is updated in response to cross-consensus messaging (XCM) requests to the parent chain (either Polkadot or Kusama). A number of the `StakingLedger` methods lack sufficient input validation before they update the staking ledger's internal state. Even though the input is validated as part of the original XCM call, there could still be issues due to implementation errors or overlooked corner cases.

First, the `StakingLedger::rebond` method does not use checked arithmetic to update the active balance. The method should also check that the computed `unlocking_balance` is equal to the input value at the end of the loop to ensure that the system remains consistent.

```
    pub fn rebond(&mut self, value: Balance) {
        let mut unlocking_balance: Balance = Zero::zero();

        while let Some(last) = self.unlocking.last_mut() {
            if unlocking_balance + last.value <= value {
                unlocking_balance += last.value;
                self.active += last.value;
                self.unlocking.pop();
            } else {
                let diff = value - unlocking_balance;

                unlocking_balance += diff;
                self.active += diff;
                last.value -= diff;
            }

            if unlocking_balance >= value {
                break;
            }
        }
    }
```

Figure 4.1: `pallets/liquid-staking/src/types.rs:199-219`

Second, the `StakingLedger::bond_extra` method does not use checked arithmetic to update the total and active balances.

```
    pub fn bond_extra(&mut self, value: Balance) {
        self.total += value;
        self.active += value;
    }
```

Figure 4.2: `pallets/liquid-staking/src/types.rs:223-226`

Finally, the `StakingLedger::unbond` method does not use checked arithmetic when updating the active balance.

```
    pub fn unbond(&mut self, value: Balance, target_era: EraIndex) {
        if let Some(mut chunk) = self
            .unlocking
            .last_mut()
            .filter(|chunk| chunk.era == target_era)
        {
            // To keep the chunk count down, we only keep one chunk per era. Since
            // `unlocking` is a FIFO queue, if a chunk exists for `era` we know that
            // it will be the last one.
            chunk.value = chunk.value.saturating_add(value);
        } else {
            self.unlocking.push(UnlockChunk {
                value,
                era: target_era,
            });
        };

        // Skipped the minimum balance check because the platform will
        // bond `MinNominatorBond` to make sure:
        // 1. No chill call is needed
        // 2. No minimum balance check
        self.active -= value;
    }
```

Figure 4.3: `pallets/liquid-staking/src/types.rs:230-253`

Since the staking ledger is updated by a number of the XCM response handlers, and XCM responses may return out of order, it is important to ensure that input to the staking ledger methods is validated to prevent issues due to race conditions and corner cases.

We could not find a way to exploit this issue, but we cannot rule out the risk that it could be used to cause a denial-of-service condition in the system.

Exploit Scenario

The staking ledger's state is updated as part of a `WithdrawUnbonded` request, leaving the `unlocking` vector in the staking ledger empty. Later, when the response to a previous call to `rebond` is handled, the ledger is updated again, which leaves it in an inconsistent state.

Recommendations

Short term, ensure that the balance represented by the staking ledger's `unlocking` vector is enough to cover the input balance passed to `StakingLedger::rebond`. Use checked arithmetic in all staking ledger methods that update the ledger's internal state to ensure that issues due to data races are detected and handled correctly.

5. Failed XCM requests left in storage

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PLF-5
Target: `pallets/liquid-staking/src/lib.rs`	

Description

When the `liquid-staking` pallet generates an XCM request for the parent chain, the corresponding XCM response triggers a call to `Pallet::notification_received`. If the response is of the `Response::ExecutionResult` type, this method calls `Pallet::do_notification_received` to handle the result.

The `Pallet::do_notification_received` method checks whether the request was successful and then updates the local state according to the corresponding XCM request, which is obtained from the `XcmRequests` storage map.

```
    fn do_notification_received(
        query_id: QueryId,
        request: XcmRequest<T>,
        res: Option<(u32, XcmError)>,
    ) -> DispatchResult {
        use ArithmeticKind::*;
        use XcmRequest::*;

        let executed = res.is_none();
        if !executed {
            return Ok(());
        }

        match request {
            Bond {
                index: derivative_index,
                amount,
            } => {
                ensure!(
                    !StakingLedgers::<T>::contains_key(&derivative_index),
                    Error::<T>::AlreadyBonded
                );
                let staking_ledger =
                    <StakingLedger<T::AccountId, BalanceOf<T>>::new(
                        Self::derivative_sovereign_account_id(derivative_index),
                        amount,
                    );
                StakingLedgers::<T>::insert(derivative_index, staking_ledger);
                MatchingPool::<T>::try_mutate(|p| -> DispatchResult {
                    p.update_total_stake_amount(amount, Subtraction)
                })?;
                T::Assets::burn_from(
                    Self::staking_currency()?,
                    &Self::account_id(),
                    Amount
                )?;
            }
            // ... <redacted>
        }
        XcmRequests::<T>::remove(&query_id);
        Ok(())
    }
```

Figure 5.1: `pallets/liquid-staking/src/lib.rs:1071-1159`

If the method completes without errors, the XCM request is removed from storage via a call to `XcmRequests<T>::remove(query_id)`. However, if any of the following conditions are true, the corresponding XCM request is left in storage indefinitely:

1. The request fails and `Pallet::do_notification_received` exits early.
2. `Pallet::do_notification_received` fails.
3. The response type is not `Response::ExecutionResult`.

These three cases are currently unhandled by the codebase. The same issue is present in the `crowdloans` pallet implementation of `Pallet::do_notification_received`.

Recommendations

Short term, ensure that failed XCM requests are handled correctly by the `crowdloans` and `liquid-staking` pallets.

6. Risk of using stale oracle prices in loans pallet

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PLF-6
Target: `pallets/loans/src/lib.rs`	

Description

The `loans` pallet uses oracle prices to find a USD value of assets using the `get_price` function (figure 6.1). The `get_price` function internally uses the `T::PriceFeeder::get_price` function, which returns a timestamp and the price. However, the returned timestamp is ignored.

```
    pub fn get_price(asset_id: AssetIdOf<T>) -> Result<Price, DispatchError> {
        let (price, _) =
            T::PriceFeeder::get_price(&asset_id)
            .ok_or(Error::<T>::PriceOracleNotReady)?;
        if price.is_zero() {
            return Err(Error::<T>::PriceIsZero.into());
        }
        log::trace!(
            target: "loans::get_price", "price: {:?}", price.into_inner()
        );

        Ok(price)
    }
```

Figure 6.1: `pallets/loans/src/lib.rs`:1430-1441

Exploit Scenario

The price feeding oracles fail to deliver prices for an extended period of time. The `get_price` function returns stale prices, causing the `get_asset_value` function to return a non-market asset value.

Recommendations

Short term, modify the code so that it compares the returned timestamp from the `T::PriceFeeder::get_price` function with the current timestamp, returns an error if the price is too old, and handles the emergency price, which currently has a timestamp of zero. This will stop the market if stale prices are returned and allow the governance process to intervene with an emergency price.

7. Missing calculations in crowdloans extrinsics

Severity: **Undetermined**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-PLF-7
Target: `pallets/crowdloans/src/lib.rs`	

Description

The `claim` extrinsic in the `crowdloans` pallet is missing code to subtract the claimed amount from `vault.contributed` to update the total contribution amount (figure 7.1). A similar bug exists in the `refund` extrinsic: there is no subtraction from `vault.contributed` after the `Self::contribution_kill` call.

```
    pub fn claim(
        origin: OriginFor<T>,
        crowdloan: ParaId,
        lease_start: LeasePeriod,
        lease_end: LeasePeriod,
    ) -> DispatchResult {
        // ... <redacted>

        Self::contribution_kill(
            vault.trie_index,
            &who,
            ChildStorageKind::Contributed
        );

        Self::deposit_event(Event::<T>::VaultClaimed(
            crowdloan,
            (lease_start, lease_end),
            ctoken,
            who,
            amount,
            VaultPhase::Succeeded,
        ));

        Ok(())
    }
```

Figure 7.1: `pallets/crowdloans/src/lib.rs:718-765`

Exploit Scenario
The `claim` extrinsic is called, but the total amount in `vault.contributed` is not updated, leading to incorrect calculations in other places.

Recommendations
Short term, update the `claim` and `refund` extrinsics so that they subtract the amount from `vault.contributed`.

Long term, add a test suite to ensure that the vault state stays consistent after the `claim` and `refund` extrinsics are called.

8. Event emitted when update_vault and set_vrf calls do not make updates

Severity: **Informational**	Difficulty: **High**
Type: Auditing and Logging	Finding ID: TOB-PLF-8
Target: `pallets/crowdloans/src/lib.rs`	

Description

The `update_vault` extrinsic in the `crowdloans` pallet is responsible for updating the three values shown in figure 8.1. It is possible to call `update_vault` in such a way that no update is performed, but the function emits an event regardless of whether an update occurred. The same situation occurs in the `set_vrfs` extrinsic (figure 8.2).

```
    pub fn update_vault(
        origin: OriginFor<T>,
        crowdloan: ParaId,
        cap: Option<BalanceOf<T>>,
        end_block: Option<BlockNumberFor<T>>,
        contribution_strategy: Option<ContributionStrategy>,
    ) -> DispatchResult {
        T::UpdateVaultOrigin::ensure_origin(origin)?;

        let mut vault = Self::current_vault(crowdloan)
            .ok_or(Error::<T>::VaultDoesNotExist)?;

        if let Some(cap) = cap {
            // ... <redacted>
        }

        if let Some(end_block) = end_block {
            // ... <redacted>
        }

        if let Some(contribution_strategy) = contribution_strategy {
            // ... <redacted>
        }

        // ... <redacted>

        Self::deposit_event(Event::<T>::VaultUpdated(
            crowdloan,
            (lease_start, lease_end),
            contribution_strategy,
            cap,
            end_block,
        ));

        Ok(())
    }
```

Figure 8.1: `pallets/crowdloans/src/lib.rs:424-472`

```
    pub fn set_vrfs(origin: OriginFor<T>, vrfs: Vec<ParaId>) -> DispatchResult {
        T::VrfOrigin::ensure_origin(origin)?;

        log::trace!(
            target: "crowdloans::set_vrfs",
            "pre-toggle. vrfs: {:?}",
            vrfs
        );

        Vrfs::<T>::try_mutate(|b| -> Result<(), DispatchError> {
            *b = vrfs.try_into().map_err(|_| Error::<T>::MaxVrfsExceeded)?;
            Ok(())
        })?;

        Self::deposit_event(Event::<T>::VrfsUpdated(Self::vrfs()));

        Ok(())
    }
```

Figure 8.2: `pallets/crowdloans/src/lib.rs:599-616`

Exploit Scenario

A system observes that the `VaultUpdate` event was emitted even though the vault state did not actually change. Based on this observation, it performs logic that should be executed only when the state has been updated.

Recommendations

Short term, modify the `VaultUpdate` event so that it is emitted only when the `update_vault` extrinsic makes an actual update. Optionally, have the `update_vault` extrinsic return an error to the caller when calling it results in no updates.

9. The referral code is a sequence of arbitrary bytes

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PLF-9
Target: `pallets/crowdloans/src/lib.rs`	

Description

The referral code is used in a number of extrinsic calls in the `crowdloans` pallet. Because the referral code is never validated, it can be a sequence of arbitrary bytes. The referral code is logged by a number of extrinsics. However, it is currently impossible to perform log injection because the referral code is printed as a hexidecimal string rather than raw bytes (using the debug representation).

```
    pub fn contribute(
        origin: OriginFor<T>,
        crowdloan: ParaId,
       #[pallet::compact] amount: BalanceOf<T>,
        referral_code: Vec<u8>,
    ) -> DispatchResultWithPostInfo {
        // ... <redacted>

        log::trace!(
            target: "crowdloans::contribute",
            "who: {:?}, para_id: {:?}, amount: {:?}, referral_code: {:?}",
            &who,
            &crowdloan,
            &amount,
            &referral_code
        );

        Ok(().into())
}
```

Figure 9.1: `pallets/crowdloans/src/lib.rs`:502-594

Exploit Scenario

The referral code is rendered as raw bytes in a vulnerable environment, introducing an opportunity to perform a log injection attack.

Recommendations

Short term, choose and implement a data type that models the referral code semantics as closely as possible.

10. Missing validation of referral code size

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-PLF-10
Target: `pallets/crowdloans/src/lib.rs`	

Description

The length of the referral code is not validated by the `contribute` extrinsic defined by the `crowdloans` pallet. Since the referral code is stored by the node, a malicious user could call `contribute` multiple times with a very large referral code. This would increase the memory pressure on the node, potentially leading to memory exhaustion.

```
    fn do_contribute(
         who: &AccountIdOf<T>,
         crowdloan: ParaId,
         vault_id: VaultId,
         amount: BalanceOf<T>,
         referral_code: Vec<u8>,
     ) -> Result<(), DispatchError> {
         // ... <redacted>

         XcmRequests::<T>::insert(
             query_id,
             XcmRequest::Contribute {
                 crowdloan,
                 vault_id,
                 who: who.clone(),
                 amount,
                 referral_code: referral_code.clone(),
             },
         );
         // ... <redacted>


         Ok(())
     }
```

Figure 10.1: `pallets/crowdloans/src/lib.rs`*:1429-1464*

Exploit Scenario

A malicious user calls the `contribute` extrinsic multiple times with a very large referral code. This increases the memory pressure on the validator nodes and eventually causes all parachain nodes to run out of memory and crash.

Recommendations

Short term, add validation that limits the size of the referral code argument to the `contribute` extrinsic.

11. Code duplication in crowdloans pallet

Severity: **Informational**	Difficulty: **High**
Type: Patching	Finding ID: TOB-PLF-11
Target: `pallets/crowdloans/src/lib.rs`	

Description
A number of extrinsics in the `crowdloans` pallet have duplicate code. The `close`, `reopen`, and `auction_succeeded` extrinsics have virtually identical logic. The `migrate_pending` and `refund` extrinsics are also fairly similar.

Exploit Scenario
A vulnerability is found in the duplicate code, but it is patched in only one place.

Recommendations
Short term, refactor the `close`, `reopen`, and `auction_succeeded` extrinsics into one function, to be called with values specific to the extrinsics. Refactor common pieces of logic in the `migrate_pending` and `refund` extrinsics.

Long term, avoid code duplication, as it makes the system harder to review and update. Perform regular code reviews and track any logic that is duplicated.

Summary of Recommendations

The Parallel Finance parachain is a work in progress with multiple planned iterations. Trail of Bits recommends that Parallel Finance address the findings detailed in this report and take the following additional steps prior to deployment:

- Expand the test suite to exercise the issues found in this assessment. Implement property testing using a framework like `quickcheck` or `proptest` to get better test coverage for less exercised code paths.
- Review the use of events throughout the platform.
- Expand the code comments and external public documentation. Add a README document for each pallet. If available, insert links to the external documentation in relevant parts of the code.
- Reduce the project's code duplication to decrease the size and complexity of the system.

A. Vulnerability Categories

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

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

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

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

B. Code Maturity Categories

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

Code Maturity Categories

Category	Description
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Configuration	The configuration of system components in accordance with best practices
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Data Handling	The safe handling of user inputs and data processed by the system
Documentation	The presence of comprehensive and readable codebase documentation
Maintenance	The timely maintenance of system components to mitigate risk
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Non-Security-Related Findings

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

Code repetition in the amm pallet. The `calculate_reserves_to_remove` function is called twice in a row, as shown in figure C.1. Because the `do_remove_liquidity` function returns the removed amounts, the call to `calculate_reserves_to_remove` in `remove_liquidity` can be removed.

```
pub fn remove_liquidity(
    origin: OriginFor<T>,
    pair: (AssetIdOf<T, I>, AssetIdOf<T, I>),
    #[pallet::compact] liquidity: BalanceOf<T, I>,
) -> DispatchResult {
    ...
        let (base_amount_removed, quote_amount_removed) =
            Self::calculate_reserves_to_remove(pool, liquidity)?;
        Self::do_remove_liquidity(&who, pool, liquidity,
            (base_asset, quote_asset))?;
    ...
}

#[require_transactional]
fn do_remove_liquidity(
    who: &T::AccountId,
    pool: &mut Pool<AssetIdOf<T, I>, BalanceOf<T, I>, T::BlockNumber>,
    liquidity: BalanceOf<T, I>,
    (base_asset, quote_asset): (AssetIdOf<T, I>, AssetIdOf<T, I>),
) -> Result<(BalanceOf<T, I>, BalanceOf<T, I>), DispatchError> {
    let (base_amount, quote_amount) =
        Self::calculate_reserves_to_remove(pool, liquidity)?;
```

Figure C.1: `pallets/amm/src/lib.rs`

Missing curve model validation. The jump rate model defined by the `loans` pallet enforces a number of invariants that can be checked by calling the `JumpModel::check_model` method. In particular, this method ensures that the maximum rate is always below 50%. The curve rate model enforces a minimum rate (less than 10%) but not a maximum rate. In particular, if the utilization rate is close to 100%, the resulting borrow rate may exceed 100%.

```
    pub fn get_borrow_rate(&self, utilization: Ratio) -> Option<Rate> {
        const NINE: usize = 9;
        let utilization_rate: Rate = utilization.into();
        utilization_rate
            .saturating_pow(NINE)
            .checked_add(&self.base_rate)
    }
```

Figure C.2: `pallets/loans/src/rate_model.rs:181-187`

Unnecessary mutable reference. The `Pallet::dissolve_vault` method, defined in the `crowdloans` pallet, passes a mutable reference to `Pallet::total_contribution`, but this reference does not need to be mutable.

Stale comment. The comment for `liquidate_borrow_internal`, defined by the `loans` pallet, refers to nonexistent arguments.

Unnecessary call in `liquidate_borrow_internal`. The code in figure C.3 can be simplified to the code in figure C.4.

```
Self::ensure_active_market(liquidate_asset_id)?;
Self::ensure_active_market(collateral_asset_id)?;

let market = Self::market(liquidate_asset_id)?;
```

Figure C.3: `pallets/loans/src/lib.rs`

```
let market = Self::ensure_active_market(liquidate_asset_id)?;
Self::ensure_active_market(collateral_asset_id)?;
```

Figure C.4: Simplified code from figure C.3

Naming inconsistency in the `bridge` pallet. The argument for `register_chain` is named `chain_id`, but the analogous argument for `unregister_chain` is named `id`. We recommend using consistent naming conventions for these arguments.

Complex control flow. The `merge_overlapping_intervals` function contains control flow code that could be simplified to a `match` construct on both variants: `None` and `Some(last_merged)` (figures C.5 and C.6).

```
if merged.is_empty() {
    merged.push(r);
} else if let Some(last_merged) = merged.last_mut() {
    if r.0 > last_merged.1 {
        merged.push(r);
    } else {
        (*last_merged).1 = r.1.max(last_merged.1);
    }
}
```

Figure C.5: `pallets/bridge/src/lib.rs:597-605`

```
match merged.last_mut() {
    None => merged.push(r),
    Some(last_merged) =>
        if r.0 > last_merged.1 {
            merged.push(r);
        } else {
            (*last_merged).1 = r.1.max(last_merged.1);
        }
}
```

Figure C.6: Simplified code from figure C.5

Unnecessary call to `unwrap` in the `bridge` pallet. The code in figure C.7 could be reorganized to remove the call to `unwrap`, as shown in figure C.8.

```
if BridgeRegistry::<T>::get(&id).is_none() {
    return;
}
let mut registry = BridgeRegistry::<T>::get(&id).unwrap();
```

Figure C.7: `pallets/bridge/src/lib.rs:619-622`

```
match BridgeRegistry::<T>::get(&id) {
    None => return,
    Some(mut registry) => { … }
}
```

Figure C.8: Refactored code from figure C.7, removing the unnecessary call to `unwrap`

Code duplication in the `bridge` pallet. The `commit_vote` function contains a repeated fragment (`let MaterializeCall { … } = call.clone();`) that could be extracted to the top of the function.

A typo in a name in the `liquid-staking` pallet. In the `ensure_market_cap` function, the `new_issurance` variable should be named `new_issuance`.

D. Automated Testing

This section describes the setup for the various automated analysis tools used during this audit.

cargo-audit

The Cargo plugin `cargo-audit` can be installed using the command `cargo install cargo-audit`. Invoking `cargo audit` in the root directory of the project runs the tool.

We ran `cargo-audit` on the Parallel repository. This run identified a number of dependencies with known vulnerabilities and resulted in one issue (TOB-PLF-1).

Clippy

The Rust linter Clippy can be installed using `rustup` by running the command `rustup component add clippy`. Invoking `cargo clippy` in the root directory of the project runs the tool.

We ran Clippy on the Parallel repository. This run did not identify any serious issues, but it did result in a few code quality recommendations, which are detailed in appendix C.

eslint

The JavaScript linter `eslint` can be installed using either `npm` or `yarn`. To install `eslint` using `npm`, run `npm install eslint --save-dev`.

We ran `eslint` on the Parallel token bridge repository using the recommended `eslint vscode` extension. This run did not uncover any significant issues.

yarn audit

We ran `yarn audit` on the Parallel token bridge repository. This run did not detect any vulnerable dependencies.

Semgrep

Semgrep can be installed using `pip` by running `python3 -m pip install semgrep`. To run Semgrep on a codebase, simply run `semgrep --config "<CONFIGURATION>" --exclude static` in the root directory of the project. (This excludes all files under the `static` directory.)

We ran Semgrep on the Parallel token bridge repository with the following configurations:

- `p/ci`
- `p/javascript`

- `p/clientside-js`
- `p/security-audit`
- `p/owasp-top-ten`

This run did not identify any issues in the codebase.

Tarpaulin

Tarpaulin can be installed using Cargo by running `cargo install cargo-tarpaulin`. To execute Tarpaulin under Linux, simply run the following command in the project root directory:

```
cargo tarpaulin --release
```

(The `--release` flag builds the project in release mode.)

The test coverage for the Parallel repository is 46% according to Tarpaulin (which corresponds to 3,158 of 6,796 lines covered).