macro

Sapience A-1

Security Audit

February 26, 2026
Version 1.0.0

Presented by 0xMacro

Table of Contents

Introduction

This document includes the results of the security audit for Sapience's smart contract code as found in the section titled ‘Source Code’. The security audit was performed by the Macro security team from February 4 to February 26, 2026.

The purpose of this audit is to review the source code of certain Sapience Solidity contracts, and provide feedback on the design, architecture, and quality of the source code with an emphasis on validating the correctness and security of the software in its entirety.

Disclaimer: While Macro’s review is comprehensive and has surfaced some changes that should be made to the source code, this audit should not solely be relied upon for security, as no single audit is guaranteed to catch all possible bugs.

Overall Assessment

The following is an aggregation of issues found by the Macro Audit team:

Severity	Count	Acknowledged	Won't Do	Addressed
Critical	4	-	-	4
Medium	3	-	-	3
Code Quality	2	1	-	1
Informational	1	-	-	-
Gas Optimization	1	1	-	-

Sapience was quick to respond to these issues.

Specification

Our understanding of the specification was based on the following sources:

- Discussions on Discord with the Sapience team.
- Technical documentation available in the repository.

Trust Model, Assumptions, and Accepted Risks (TMAAR)

End Users (Predictor / Counterparty / LP)

Considered untrusted and potentially malicious. The system is designed to be defensive against users attempting to claim collateral they are not entitled to.

- Users must provide valid EIP-712 signatures (ECDSA, EIP-1271, or session key) to participate in mint and burn operations. The contract verifies signatures against the claimed signer address and a

per-account nonce to prevent replay attacks.

- Users are responsible for providing correct picks, wagers, nonces, and deadlines. The contract validates these inputs on-chain.
- A malicious user's actions are confined to their own positions. They cannot affect the collateral or settlement outcome of other user's predictions.
- Position tokens are freely transferable ERC20s. The protocol is agnostic to secondary market activity, any holder of a position token can redeem it proportionally after settlement, regardless of how they acquired it.
- Users interacting with `burn()` (bilateral exit) are responsible for agreeing to a fair payout split off-chain. The contract only enforces the conservation constraint (`predictorPayout + counterpartyPayout == totalTokensBurned`) and that both parties signed the same parameters.
- Vault users (LPs) deposit assets and receive shares through a manager request flow. They are responsible for submitting requests within the expiration window and accept that the manager determines when conditions are favorable for processing.

Protocol Owner

Highly trusted during the setup phase, can renounce ownership after configuration. The owner has administrative control across multiple contracts and is trusted to:

PredictionMarketEscrow

- `setAccountFactory(address)` — Sets the `ZeroDevKernelAccountFactory` wrapper used to verify smart account ownership in session key flows. The wrapper is a protocol-deployed contract pointing at the ZeroDev Kernel factory and ECDSA validator. The owner is trusted to deploy the wrapper with correct external addresses and to register it here accurately. An incorrect factory or wrapper configuration would allow session keys associated with arbitrary accounts to sign on behalf of users, or silently disable smart account verification entirely.
- `sweepDust(bytes32, address)` — Sweeps rounding-error collateral from fully redeemed pick configurations.

PredictionMarketBridge / PredictionMarketBridgeRemote

- `setBridgeConfig(BridgeConfig)` — Sets the remote endpoint ID and remote bridge address for LayerZero messaging. Misconfiguration would allow messages from unauthorized senders to release escrowed tokens.

- `withdrawETH(address, uint256)` — Withdraws the ETH balance used to fund ACK fees.
- `renounceOwnershipSafe()` — Renounces ownership once all bridge configuration is complete.

PredictionMarketTokenFactory

- `setDeployer(address)` — Sets the authorized deployer (typically the bridge) that can deploy new position token contracts via `CREATE3`. Pointing this at an untrusted address would allow arbitrary token deployments at deterministic addresses.
- `renounceOwnershipSafe()` — Renounces ownership once the deployer is set. After this, no new deployer can be authorized.

ConditionalTokensConditionResolver / LZConditionResolver

- `setBridgeConfig(LZTypes.BridgeConfig)` — Sets the LayerZero remote endpoint ID and trusted bridge address for receiving resolution messages. Misconfiguration would allow this address to inject resolution outcomes into the resolver.

PredictionMarketVault

- `setManager(address)` — Designates the manager EOA. The manager has broad authority over user funds and deposit/withdrawal processing.
- `setMaxUtilizationRate(uint256)` — Sets the maximum fraction of vault assets that can be deployed to external protocols.
- `setDepositInteractionDelay(uint256)` / `setWithdrawalInteractionDelay(uint256)` — Sets the cooldown period between user requests. Setting this to zero or a very large value can break normal user flows.
- `setExpirationTime(uint256)` — Sets how long a request remains valid before it can be cancelled. Setting this to zero makes all requests immediately expired.
- `toggleEmergencyMode()` — Enables or disables emergency withdrawals, bypassing the normal manager-mediated flow.
- `pause()` / `unpause()` — Halts or resumes all user-facing operations.

Vault Manager

Highly trusted operational role. The vault manager controls fund deployment and request processing on behalf of the `PredictionMarketVault`. This entity is trusted to:

- `processDeposit(address)` / `batchProcessDeposit(address[])` — Process pending deposit requests by minting shares to users. There is no on-chain enforcement of the share price at processing time; the manager is trusted to process only at fair rates.
- `processWithdrawal(address)` / `batchProcessWithdrawal(address[])` — Process pending withdrawal requests by burning shares and returning assets. The manager controls the ordering and timing of withdrawals.
- `approveFundsUsage(address, uint256)` — Approves vault assets to an external protocol (PredictionMarketEscrow). A malicious or negligent manager can deploy funds to an unfavorable or incorrect protocol, resulting in permanent loss of LP assets.
- `cleanInactiveProtocols()` — Removes protocols with zero deposits and zero allowance from the active set.
- Signs EIP-1271 messages on behalf of the vault. Any message hash signed by the manager is considered a valid vault signature, granting broad signing authority. A compromised manager key can authorize arbitrary EIP-1271 validations against the vault.

Condition Resolvers

Trusted external contracts. The protocol delegates condition resolution entirely to resolver contracts registered per pick. These are trusted to:

- Return correct and final outcome vectors `[yesWeight, noWeight]` for each `conditionId`. An incorrect outcome directly determines who wins the collateral pool, there is no appeal mechanism once a pick configuration is resolved.
- Behave correctly under the `RESOLVER_GAS_LIMIT` cap. A resolver that reverts, runs out of gas, or returns arrays shorter than expected will cause settlement to fail or revert permanently for affected predictions.
- Non-decisive outcomes (vectors where neither weight is zero, e.g. `[1,1]`) are currently treated as `COUNTERPARTY_WINS` by the escrow's parlay logic. A resolver returning such a vector is therefore equivalent to a counterparty win.
- **PythConditionResolver** — Trusts the Pyth Lazer oracle to provide accurate historical price data. The resolver is permissionless; anyone can call `settleCondition()` with a valid Pyth payload. Price feed manipulation or use of stale data would affect settlement outcomes.
- **ConditionalTokensConditionResolver** — Trusts Polygon `ConditionalTokensReader` to accurately read and relay payout data from the Gnosis ConditionalTokens contract via LayerZero.

OnboardingSponsor / Budget Manager

The budget manager is a semi-trusted operational role, within the `OnboardingSponsor` contract. The budget manager and the owner are trusted to:

- `setBudget(address, uint256)` / `setBudgets(address[], uint256[])` — Set per-user collateral budgets unlocked by invite codes. Setting an excessive budget for a user allows them to place sponsored bets beyond the intended onboarding allocation.
- The owner additionally controls: `setMatchLimit(uint256)` — caps collateral per sponsored mint; `setBudgetManager(address)` — rotates the API signer; `sweepToken(IERC20, address, uint256)` / `sweepNative(address, uint256)` — recovers all funds held by the sponsor.

Constraints enforced on-chain by `OnboardingSponsor` at call time: required counterparty (prevents self-dealing), max entry price cap (prevents risk-free farming on near-certain outcomes), per-user budget limit, and per-mint match limit. A misconfigured or compromised budget manager can exhaust the sponsor's collateral balance by over-allocating budgets.

LayerZero Protocol

Trusted cross-chain infrastructure. LZ peer configuration is set by the owner at deployment and is assumed to be correct before ownership is renounced. Post-renouncement, peer addresses are permanent and any misconfiguration cannot be corrected. The protocol relies on LayerZero V2 for two distinct message flows:

- **Bridge messaging** (`CMD_BRIDGE` / `CMD_ACK`) between Ethereal and Arbitrum for position token bridging. A failed message leaves a bridge in `PENDING` state until retried. A maliciously delivered message from an unconfigured peer would need to pass the LZ endpoint's source verification, the protocol trusts that LZ enforces this correctly.
- **Resolution relay** from Polygon (`ConditionalTokensReader` → `ConditionalTokensConditionResolver`) carrying condition outcomes to Ethereal resolvers. A failed or reordered message blocks settlement for any prediction depending on that condition indefinitely, as the resolver has no fallback.

General Risk Observations

- Changing any bridge or resolver configuration while predictions are active (mid-flight) can permanently break resolution for unresolved conditions, locking user collateral in escrow with no recovery path. The owner is trusted not to modify live configurations until all dependent predictions are settled.

- Changing the `deployer` on `PredictionMarketTokenFactory` after tokens are already deployed breaks the trust assumption that only the legitimate bridge can mint and burn bridged position tokens. Any address set as the new deployer gains the ability to deploy contracts at deterministic `CREATE3` addresses, potentially shadowing existing tokens or minting unbacked supply. The owner is trusted not to rotate the deployer once the bridge is operational.

Source Code

The following source code was reviewed during the audit:

- **Repository:** sapience
- Commit Hash **(initial)**: `f2b6bd9231ae9915f9ba88f6bef02e7a914e6288`
- Commit Hash **(final)**: `94919e2e9420a07234ad92910318465549049084`

We audited the following contracts with **f2b6bd9231ae9915f9ba88f6bef02e7a914e6288** commit hash:

Source Code	SHA256
./src/v2/PredictionMarketEscrow.sol	fe0c18ba03c347cbf3f3f60a33a2a382b921200568f8d2af07fe8b78d417931c
./src/v2/PredictionMarketToken.sol	198d76d66fef56d563d22394155c5c66e38f97d1b77eecf3cb88fcc8e2eba5b4
./src/v2/PredictionMarketTokenFactory.sol	d8a8c43b626b0cd4bbe00a5baea2153013623ed5aeba16e06ecd5beefdb63052
./src/v2/SecondaryMarketEscrow.sol	4c112d0fb3ef8fb465d00abbe295eb08ba66f87ba8b7dcc3b00d4b0deb8bd74b
./src/v2/bridge/PredictionMarketBridge.sol	3fd3cc0564333d8464ff6b0de6deb28a57094a8afe99ff8e9e6905b1d975f48a
./src/v2/bridge/PredictionMarketBridgeBase.sol	3abd34604fd6fa360594393ac0af06e718866b645ea890b130ae17beaf6779a1
./src/v2/bridge/PredictionMarketBridgeRemote.sol	9f7d6a512db41277d6ac9b61b11da50d8c8c35b69b63fd4710f28368ddd7059f
./src/v2/bridge/interfaces/IPredictionMarketBridge.sol	1c6d2a20c0f1bbb907c995051894d392c57a4eaf4bdf0997bc574c7e92a1bb1f
./src/v2/bridge/interfaces/IPredictionMarketBridgeBase.sol	9ae9e9bb85beb7e6b67a78e71e1513c6224982dba69b4b51180bc40535b49967

Source Code	SHA256
./src/v2/bridge/interfaces/IPredictionMarketBridgeRemote.sol	e062eb5a4f9457e680d2cdd6a7dbf2de1c4206ebdf59fee73ff56cd0f8b2ec4a
./src/v2/interfaces/IConditionResolver.sol	6a0020bd755f209c7d165758272cea4a359ed222524ba2f30844b638f86879b9
./src/v2/interfaces/IMintSponsor.sol	d99f15d7a35e817d490b9b5763956688c13e4280efa7fef493018c2344a3f057
./src/v2/interfaces/IPredictionMarketEscrow.sol	a49a4947cef12233a2eb78b6c0208c9c39b9e4ca095f8b91579806910d7d61ba
./src/v2/interfaces/IPredictionMarketToken.sol	9e8d3d501caca427db8269f3fe022b3e81b682a579e100b49071289d99eb5d28
./src/v2/interfaces/IPredictionMarketTokenFactory.sol	64d257ca69f5cea70da1f5783b887ae1bae626c11840c65ad2b0cfd0ee7a73bb
./src/v2/interfaces/ISecondaryMarketEscrow.sol	25b8666dc38cba5f9c6f5077e31c07aa07f76a12ae11fa8291f421238f56ff25
./src/v2/interfaces/IV2Events.sol	1163665b56ea161095652ce67b0568634be368ef50172a210c3f9aa2fe9087d8
./src/v2/interfaces/IV2Types.sol	28c7bde3a8b4970168cb399ff00fb81adce53ba49939174d0961be434ca4a05f
./src/v2/resolvers/conditionalTokens/ConditionalTokensConditionResolver.sol	5aee5a565adeb485a2b81f89b4e3648bfa14faff65701ffadd37d70effd9ad15
./src/v2/resolvers/conditionalTokens/ConditionalTokensReader.sol	038e5d7b44bf2a4b7aa457d0ad6319bf8faeb24b63e4c9f510f64a942c914d24
./src/v2/resolvers/conditionalTokens/interfaces/IConditionalTokensConditionResolver.sol	a999559e2717d9c1c6884026fe819a535e6bd343eb33e40de48ca8e6a894c3a2
./src/v2/resolvers/conditionalTokens/interfaces/IConditionalTokensReader.sol	eaaa006b981a47c647a394882f1056d17549b0dc076d66301c99de395be37132

Source Code	SHA256
./src/v2/resolvers/lz-uma/LZConditionResolver.sol	313c7082bccc33087199f4d9a9ffe3870cdced03ad48d6f4bac76540535c4e7d
./src/v2/resolvers/lz-uma/LZConditionResolverUmaSide.sol	8e124572a8aa57da90cdbb722400679607d56a2db95740c005dbd7a6ea936645
./src/v2/resolvers/lz-uma/LZETHManagement.sol	e5b7ff1ab151592a1567cea87a3c8b3b72e2cb63142f03a43dd1f909973b6c03
./src/v2/resolvers/lz-uma/interfaces/ILZConditionResolver.sol	46b925318e4be22731178af469c940a88592bf823782c39612c5b1503a62fe54
./src/v2/resolvers/lz-uma/interfaces/ILZConditionResolverUmaSide.sol	30bbe87be86b4136e5c025f5c61ab674e027ff5857b293f9c0b734e89054e19b
./src/v2/resolvers/mocks/ManualConditionResolver.sol	3da2c3ada52f8a60dcf603bea373b8273a18f5c84ccb949c09540e8142e08ced
./src/v2/resolvers/pyth/PythConditionResolver.sol	dfd5989421cfa4f29a659274fea9ee0ed26f864203f445d29bfcf0dc9fc4d946
./src/v2/resolvers/pyth/PythLazerLibs/IPythLazer.sol	b134c9fbc1d42e0c0ccd6f2bc6ff5d3f5fa490f16674e90e6ff3b8ef618ebc42
./src/v2/resolvers/pyth/PythLazerLibs/PythLazerLib.sol	49a4945b29758112f5a7408aba31b8527cdc21e394f1db7334d1f78a816f4ce1
./src/v2/resolvers/pyth/PythLazerLibs/PythLazerLibBytes.sol	27ca50a8422e23f50c22a836908934f64cf9959951a5d6704425f5b32df690e6
./src/v2/resolvers/pyth/PythLazerLibs/PythLazerStructs.sol	ddaa9035f7794d7eb56616bc68ea7b8ccd896c863bb35908bc424517e55f8115
./src/v2/resolvers/shared/LZTypes.sol	82d96ecdf30cb074cd68d1d8860ba70c5072ee21a08387914b290a5c735f351d

Source Code	SHA256
./src/v2/sponsors/OnboardingSponsor.sol	13ca9b1889918f3365c8725f86eaa68cadc173f22bf67d67d6963a5a6a22fce6
./src/v2/utils/IAccountFactory.sol	50bd14f9be77309ac0e54e419ae111e78ba535f85d14e5a9f993eddeef4b9b80
./src/v2/utils/SignatureProcessor.sol	10f1c9c9ffa1894182e851924633528e7480f7a5e9d4d82875da269669470c29
./src/v2/utils/SignatureValidator.sol	ebec7ada1ab5cd5e96a9ec21d0ecc6866908a192e790c476a4005d99a54e36b9
./src/v2/utils/ZeroDevKernelAccountFactory.sol	52986141b26e01651a982e8ba003a37126b2ca7015e4ff025ac971c3fdbfc8a1
./src/v2/vault/PredictionMarketVault.sol	050445136d86787b131c7195933e7ac6092337a6babd3254b8f880fd2448db89
./src/v2/vault/interfaces/IPredictionMarketVault.sol	59536d72e5c8bec2f8c09da5e76ab30e46e094be7d058c8facc17056d2e13fb8

Note: This document contains an audit solely of the Solidity contracts listed above. Specifically, the audit pertains only to the contracts themselves, and does not pertain to any other programs or scripts, including deployment scripts.

Issue Descriptions and Recommendations

Click on an issue to jump to it, or scroll down to see them all.

Security Level Reference

We quantify issues in three parts:

1. The high/medium/low/spec-breaking **impact** of the issue:
    - How bad things can get (for a vulnerability)
    - The significance of an improvement (for a code quality issue)
    - The amount of gas saved (for a gas optimization)
2. The high/medium/low **likelihood** of the issue:
    - How likely is the issue to occur (for a vulnerability)
3. The overall critical/high/medium/low **severity** of the issue.

This third part – the severity level – is a summary of how much consideration the client should give to fixing the issue. We assign severity according to the table of guidelines below:

Severity	Description
(C-x) Critical	We recommend the client **must** fix the issue, no matter what, because not fixing would mean **significant funds/assets WILL be lost.**
(H-x) High	We recommend the client **must** address the issue, no matter what, because not fixing would be very bad, *or* some funds/assets will be lost, *or* the code's behavior is against the provided spec.
(M-x) Medium	We recommend the client to **seriously consider** fixing the issue, as the implications of not fixing the issue are severe enough to impact the project significantly, albiet not in an existential manner.
(L-x) Low	The risk is small, unlikely, or may not relevant to the project in a meaningful way. Whether or not the project wants to develop a fix is up to the goals and needs of the project.
(Q-x) Code Quality	The issue identified does not pose any obvious risk, but fixing could improve overall code quality, on-chain composability, developer ergonomics, or even certain aspects of protocol design.
(I-x) Informational	Warnings and things to keep in mind when operating the protocol. No immediate action required.
(G-x) Gas Optimizations	The presented optimization suggestion would save an amount of gas significant enough, in our opinion, to be worth the development cost of implementing it.

Issue Details

C-1 User payout can be stolen with minimal risk

TOPIC	STATUS	IMPACT	LIKELIHOOD
Stolen Rewards	Fixed	Critical	High

Users are allowed to make bets on picks that already have bets on them, and as long as there is a willing counter party, the bet can be of any amount on either side. There is nothing preventing a user from being both party and counter party to a bet, be it using the same address or another they control. Since the way rewards are calculated is based on a percent proportion of the winning bet collaterals, and tokens are minted based on the wager amount for a given side, an outsized bet made at any point would lead to them taking an outside proportion of the winnings from the losing side:

```
// Mint tokens proportional to wager (1:1 ratio)
IPredictionMarketToken(predictorToken)
    .mint(request.predictor, request.predictorWager);
IPredictionMarketToken(counterpartyToken)
    .mint(request.counterparty, request.counterpartyWager);
```

Reference: PredictionMarketEscrow.sol#L283-287

```
// Use ORIGINAL total tokens (= collateral in 1:1 ratio), not current totalSupply
// This ensures consistent payouts even after partial redemptions
uint256 originalTotalTokens = isPredictor
  ? config.totalPredictorCollateral
  : config.totalCounterpartyCollateral;

// Calculate claimable pool based on result
uint256 claimablePool = _calculateClaimablePool(
    config.result,
    config.totalPredictorCollateral,
    config.totalCounterpartyCollateral,
    isPredictor
```

```
);

// Proportional payout: (amount / originalTotalTokens) * claimablePool
payout = (amount * claimablePool) / originalTotalTokens;
```

Reference: PredictionMarketEscrow.sol#L541-556

In cases where there are active bets for a given pick, and the likelihood of one side has shifted to being very likely or a certainty to win, someone could make a disproportionately large bet for the likely outcome and place one wei as the counterparty. This results in them taking winnings owed to honest betters, with little to no risk depending on how certain the pick is. There should be an invariant where if a bet is made, the winner should be rewarded the sum both parties collateral, irrelevant of the actions others make on the same pick.

Remediations to Consider

When minting tokens each party should receive token amounts proportional to the sum of collateral placed for the given bet, rather than equal to each sides wager. This means that a token will always represents a dollar (or collateral token equivalent) if side wins, and zero if not. Then once settled, rather than rewarding based on a percent share of collateral, each token can be redeemed for one collateral token, simplifying the contract, removing the possibility of dust accumulating, and prevents users from stealing rewards.

C-2 Users can mint predictions after settlement

TOPIC	STATUS	IMPACT	LIKELIHOOD
Exploit	Fixed	High	High

In `PredictionMarketEscrow`, the `PredictionMarketEscrow.mint()` function does not validate whether the `pickConfigId` has already been resolved. This allows an attacker to mint new position tokens after the outcome is known, diluting existing winning token holders and extracting value from the pool as mentioned in C-1.

The `mint()` function checks if a token pair exists for a `pickConfigId` but never validates the resolution state:

```
function mint(IV2Types.MintRequest calldata request)
    external
    nonReentrant
    returns (
        bytes32 predictionId,
        address predictorToken,
        address counterpartyToken
    )
{
    // ... validation ...

    bytes32 pickConfigId = _computePickConfigId(request.picks);

    // Create or reuse token pair for this pick configuration
    IV2Types.TokenPair storage tokenPair = _tokenPairs[pickConfigId];
    if (tokenPair.predictorToken == address(0)) {
        // First bet with these picks creates both tokens
        (predictorToken, counterpartyToken) = _createTokenPair(pickConfigId);
        // ...
    } else {
        // Reuse existing tokens but no checks for config.resolved!
        predictorToken = tokenPair.predictorToken;
        counterpartyToken = tokenPair.counterpartyToken;
    }

    _executeMint(...);
}
```

Reference: PredictionMarketEscrow.sol#L150-265

An external attacker could monitor resolution events and mint predictions after they are resolved, diluting the payouts of legitimate winners and extracting value from them. The exploit can be executed atomically, is effectively riskless, since the outcome is known and the losing side can be minimized to only 1 wei, and can be scaled with flash loans since settlement is already public and set in the `pickConfigId` state.

Remediations to Consider

Consider adding a resolution state check in the `mint()` function.

```
IV2Types.PickConfiguration storage config = _pickConfigurations[pickConfigId];
if (config.resolved) {
```

```
        revert PickConfigAlreadyResolved();
    }
```

C-3 All assets in the PredictionMarketVault can be locked

TOPIC	STATUS	IMPACT	LIKELIHOOD
Locked assets	Fixed	Critical	High

In PredictionsMarketVault's requestWithdrawal the user can request `shares` and `expectedAssets`, where shares is constrained to the users balance, but expected assets is not and can be any value. `expectedShares` is used to increment `pendingWithdrawalAssets` which is a simple bookkeeping variable that tracks assets pending to be removed and is not acted on directly in the contract. When making a withdrawRequest, a user can maliciously set this value to be the max uint value which would prevent any further withdrawal requests as expectedAssets has to be non-zero so would result in a overflow and revert. Effectively all assets in the vault could be locked, with a simple griefing attack, and a minimum of 1 wei of shares.

Remediations to Consider

Remove `pendingWithdrawalAssets`, since it is not directly used by the contract, or constrain the value of expected assets to be something reasonable based on shares trying to be withdrawn.

C-4 Fake tokens can be bridged to mint unbacked bridged tokens and permanently corrupt mappings

TOPIC	STATUS	IMPACT	LIKELIHOOD
Exploit	Fixed	Critical	High

When bridging a position token from Ethereal to Arbitrum, the `PredictionMarketBridge.bridge()` function validates position tokens by calling `pickConfigId()` and `isPredictorToken()` on the provided token address ensuring the target address implements these methods:

```
// Validate token has required interface (must be a PositionToken)
try IPredictionMarketToken(token).pickConfigId() returns (bytes32) { }
catch {
    revert InvalidToken(token);
}

try IPredictionMarketToken(token).isPredictorToken() returns (bool) { }
catch {
    revert InvalidToken(token);
}
```

Reference: PredictionMarketBridge.sol#L52-60

However, there is no check that the token was actually deployed by `PredictionMarketEscrow`. Any contract that implements these two view functions with matching return values passes validation. An attacker can deploy a fake ERC20 with arbitrary supply, returning the same `pickConfigId` and `isPredictorToken` as a legitimate token, and bridge it successfully.

This is exploitable because of two additional design choices on the Arbitrum side. First, the `CREATE3` salt used by `PredictionMarketTokenFactory` does not include the source token address:

```
function predictAddress(bytes32 pickConfigId, bool isPredictorToken)
    public
    view
    returns (address)
{
    bytes32 salt = computeSalt(pickConfigId, isPredictorToken);
    return CREATE3.predictDeterministicAddress(salt, address(this));
}
```

Reference: PredictionMarketTokenFactory.sol#L68-75

This means a fake token with matching `pickConfigId` and `isPredictorToken` maps to the exact same bridged token address as the legitimate one.

And, second the `sourceToRemote` and `remoteToSource` mappings in `PredictionMarketBridgeRemote` will be overwritten on every bridge call if the `sourceToken` address

is different, which will be when using fake tokens:

```
// Always update mappings (in case of re-bridging after full release)
if (sourceToRemote[sourceToken] == address(0)) {
    sourceToRemote[sourceToken] = remoteToken;
    remoteToSource[remoteToken] = sourceToken;
}
```

Reference: PredictionMarketBridgeRemote.sol#L260-264

Together, these allow an attacker to:

1. Mint unbacked bridged tokens: Bridge a fake token before any legitimate user. The attacker receives legitimate bridged tokens and can trade them on Arbitrum.
2. Drain escrowed tokens: Corrupt `remoteToSource` to point the bridged token at the real source token, then bridge back. This can be done if a legitimate user bridges canonical tokens after the attacker. The ACK on Ethereal releases from `_escrowedBalances[realToken]`, draining user funds.
3. Permanently lock legitimate funds: Overwrite `remoteToSource` to the fake token address. If the attacker bridges fake tokens after a legitimate bridge, and corrupt the mapping pointers with a fake address. Causing all future bridge-backs release worthless fake tokens, permanently locking real escrowed tokens.

Remediations to Consider

- Validate source token origin on Ethereal. `PredictionMarketBridge.bridge()` should verify the token was deployed by the protocol's `PredictionMarketEscrow`.
- Additionally, include source token address in the CREATE3 salt, changing the factory salt to `keccak256(abi.encode(pickConfigId, isPredictorToken, sourceTokenAddress))`. This isolates each source token into its own bridged token address, so a fake token cannot claim or share the bridged token of a legitimate one.

~~M-1~~ Losing-side position tokens can never be burned permanently blocking dust sweeping

TOPIC	STATUS	IMPACT	LIKELIHOOD
Design Flaw	Fixed	Medium	High

In `PredictionMarketEscrow` contract, when a pick configuration resolves decisively (`PREDICTOR_WINS` or `COUNTERPARTY_WINS`), the losing side's position tokens have a payout of zero. If they call the `redeem()` function it will silently skip both the token burn and the collateral transfer when `payout == 0` .

```
function redeem(address positionToken, uint256 amount, bytes32 refCode)
    external
    nonReentrant
    returns (uint256 payout)
{
    // ... validation and payout calculation ...

    // Payout math
    payout = (amount * claimablePool) / originalTotalTokens;

    if (payout > 0) {
        // Burn the position tokens
        IPredictionMarketToken(positionToken).burn(msg.sender, amount);
        collateralToken.safeTransfer(msg.sender, payout);
        // ...
    }
    // @audit When payout == 0, entire logic is skipped and nothing happens
}
```

Reference: PredictionMarketEscrow.sol#L517-577

The losing-side tokens remain permanently in circulation. There is no alternative mechanism to burn them since `burn()` reverts after resolution due to the `!config.resolved` check.

This permanently blocks the `sweepDust()` function, which requires **both** token supplies to reach zero:

```
function sweepDust(bytes32 pickConfigId, address recipient) external onlyOwner nonReent
    // ...
    uint256 predictorSupply =
        IPredictionMarketToken(tokenPair.predictorToken).totalSupply();
```

```
        uint256 counterpartySupply =
            IPredictionMarketToken(tokenPair.counterpartyToken).totalSupply();

        if (predictorSupply > 0
          || counterpartySupply > 0) {
            revert TokensStillOutstanding(predictorSupply, counterpartySupply);
        }
        // ...
    }
```

Reference: PredictionMarketEscrow.sol#L96-L145

Because `sweepDust` is the only mechanism to recover rounding dust left from payout divisions, any collateral dust from decisive outcomes is permanently locked in the contract.

Additionally, there is no incentive for any losing-side holder to call `redeem()`. They would pay gas to receive nothing. Even if `redeem()` burns tokens on zero-payout, it would still rely on losing-side holders voluntarily spending gas for no benefit.

Remediations to Consider

- Modify `sweepDust()` to only require winning-side tokens to be burned. Also considering the non-decisive case were both sides should burn their positions.
- Also consider burning the losing-side tokens in `redeem()` even when `payout == 0`.

~~M-2~~ Session keys lack on-chain revocation and permission verification

TOPIC	STATUS	IMPACT	LIKELIHOOD
Session Management	Fixed	Medium	Medium

The current session key implementation for the `PredictionMarketEscrow` has two related limitations that stem from its purely off-chain, stateless design.

No revocation before expiry

Once an owner signs a `SessionKeyApproval`, it remains valid until `validUntil` timestamp, there is no on-chain way to cancel it early. There is no mapping of revoked sessions, no nonce on the approval itself, and no registry of active sessions:

```
// Check session key is still valid
if (block.timestamp > sessionApproval.validUntil) {
    return false;
}
```

Reference: SignatureValidator.sol#L480-L483

If a session key is compromised, the attacker can continue signing valid mint and burn approvals until the session expires.

`permissionsHash` is signed but never enforced

The `SessionKeyApproval` struct commits to a `permissionsHash` in the owner's signed message:

```
bytes32 public constant SESSION_KEY_APPROVAL_TYPEHASH = keccak256(
    "SessionKeyApproval(address sessionKey,address smartAccount,uint256 validUntil,byte
);
```

Reference: SignatureValidator.sol#L35-L37

However, after verifying the owner's signature, `permissionsHash` is never checked against the actual operation. A session key approved with any value or argument here can equally sign burns or unlimited wagers.

Remediations to Consider

1. Consider adding on-chain session revocation. Store a mapping of revoked session keys and check revocation status during `_isSessionKeyApprovalValid` before returning true.
2. Additionally, consider enforcing `permissionsHash` or remove it. Either define a concrete `Permissions` struct (e.g. allowed operations, max wager, allowed `pickConfigIds`, mint or burn), hash it to derive `permissionsHash`, and validate the actual operation against decoded permissions on-chain; or remove the field from the typehash entirely to avoid misleading users into thinking it has any effect.

~~M-3~~ Sequential nonces may prevent users from making multiple bets

TOPIC	STATUS	IMPACT	LIKELIHOOD
Nonces	Fixed	Low	High

In `PredictionMarketEscrow` while minting a bet for a prediction, the signature of both parties is verified and the signatures nonce is checked to ensure it is the same as the currently set sequential nonce for each user:

```
// Validate counterparty signature (EOA or session key)
if (!_validatePartySignature(
        predictionHash,
        request.counterparty,
        request.counterpartyWager,
        request.counterpartyNonce,
        request.counterpartyDeadline,
        request.counterpartySignature,
        request.counterpartySessionKeyData
    )) {
    revert InvalidSignature();
}
if (request.counterpartyNonce != _nonces[request.counterparty]) {
    revert InvalidNonce();
}

// Increment nonces
_nonces[request.predictor]++;
_nonces[request.counterparty]++;
```

Reference: PredictionMarketEscrow.sol#L210-228

However, if a user makes multiple bets around the same time, processing these wagers may result in collisions if they use the same nonce, or if they are ordered incorrectly and result in failure. This would limit the throughput of bets and cause friction, especially for users intending to take on lots of counterparty bets, like the `PredictionMarketVault` intends to. Since each party has to sign with their nonce, once the wager is known, which can take variable time for both parties, there is no way to know which of your wagers will be ready sooner and should use the current nonce. Additionally, if either user takes part in a wager using that nonce, as another may have been signed by both parties quicker, it would invalidate the wager and require a new signature as their nonce has increased.

Remediations to Consider

Instead of using sequential nonces, generate a random nonce for each wager, and use a mapping to ensure it has not been used before. This will allow users to take on any number of wagers at the same time with no issues as they would each have a unique nonce.

I-1 Arbitrary contract address can be used as condition resolver

TOPIC	IMPACT
Informational	Informational ✳

By design, the `PredictionMarketEscrow` does not enforce a whitelist or registry of approved condition resolvers. Any contract address can be passed as `conditionResolver` in a pick, and the escrow accepts it if it implements `IConditionResolver` and returns valid results. During mint, the escrow only calls `isValidCondition(conditionId)` during settlement, it calls `getResolution()` or `getResolutions()` and uses the returned outcome vectors to decide the winner. A malicious resolver could always return outcomes in favor of one side, enabling theft of the other side's collateral. It is the user's responsibility to sign only with resolvers they trust. Both predictor and counterparty must verify the resolver address before signing the `MintRequest`. Users who sign picks without checking the resolver accept the associated trust and risk.

RESPONSE BY SAPIENCE

> Acknowledged — by design, documented in TMAAR. User responsibility to verify resolver trust.

~~Q-1~~ `getClaimableAmount` does not check token address

TOPIC	STATUS	QUALITY IMPACT
Authentication	Fixed	Low

In `PredictionMarketEscrow` contract, the `getClaimableAmount()` view function does not check if the `positionToken` address is a valid position token, it only checks if it's a `predictorToken` and defaults to a counterparty. Consider adding a check to ensure the passed argument is in fact a valid position token.

Q-2 `settle()` has no functional flow

TOPIC	STATUS	QUALITY IMPACT
Protocol Design	Acknowledged	Medium

In `PredictionMarketEscrow`, `settle()` function marks individual `prediction.settled = true` and emits events with calculated claimable amounts. But `redeem()` operates entirely on position tokens and the PickConfiguration, it never checks whether any prediction is settled. It only requires `config.resolved == true`.

The resolution of a `PickConfiguration` is done as a side effect of the first `settle()` call on any prediction within that config. After that, any token holder can call `redeem()` directly.

This means:

- The `_calculateClaimableForPrediction` output in `settle()` is purely informational (emitted in events but never stored or used).
- A user can `redeem()` without ever having their specific prediction settled.
- The `settled` flag on individual predictions serves no functional purpose.

While not a direct vulnerability, it creates misleading invariants, off-chain systems relying on `PredictionSettled` events for accounting could diverge from actual on-chain redemptions.

RESPONSE BY SAPIENCE

> Acknowledged — settle() triggers resolution and emits events consumed by indexer/frontend. Redundant prediction.settled flag is cosmetic, not a security risk.

G-1 Bridge acknowledgment pattern is unnecessary and costly

TOPIC	STATUS	GAS SAVINGS
Bridging	Acknowledged	Medium

Currently bridging via either PredictionMarketBridge or PredictionMarketBridgeRemote uses a bridge acknowledgement pattern, where once the bridge message is received from the other chains pair contract and is processed, the contract sends an acknowledgement bridge message back which makes the request from pending to completed. In the case of PredictionMarketBridgeRemote the acknowledgement function also burns the tokens and adjusts escrow accounting:

```
/// @dev Handle ACK from Ethereal (burn escrowed tokens)
function _handleAck(bytes memory data) internal override {
    bytes32 bridgeId = abi.decode(data, (bytes32));
    PendingBridge storage pending = _pendingBridges[bridgeId];

    if (pending.status == BridgeStatus.PENDING) {
        pending.status = BridgeStatus.COMPLETED;

        // Now burn the escrowed tokens
        _escrowedBalances[pending.token] -= pending.amount;
        IPredictionMarketTokenBridged(pending.token)
            .burn(address(this), pending.amount);

        emit BridgeCompleted(bridgeId);
    }
}
```

Reference: PredictionMarketBridgeRemote.sol#L283-297

However, bridging can function without requiring an acknowledgement back to the sending chain since it currently will already prevent processed bridge messages from being executed again. Additionally for the remote bridge, since bridge requests cannot be canceled, burning the tokens on acknowledgement could instead be done on the bridge request. All in all the pattern adds unnecessary complexity and additional cost for users for the extra bridge message.

Remediations to Consider

Remove the acknowledge pattern to reduce complexity and reduce the cost of bridging.

RESPONSE BY SAPIENCE

> Acknowledged — valid optimization, but refactoring bridge state machine pre-launch carries more risk than the gas savings justify. Tracked for future improvement.

Disclaimer

Macro makes no warranties, either express, implied, statutory, or otherwise, with respect to the services or deliverables provided in this report, and Macro specifically disclaims all implied warranties of merchantability, fitness for a particular purpose, noninfringement and those arising from a course of dealing, usage or trade with respect thereto, and all such warranties are hereby excluded to the fullest extent permitted by law.

Macro will not be liable for any lost profits, business, contracts, revenue, goodwill, production, anticipated savings, loss of data, or costs of procurement of substitute goods or services or for any claim or demand by any other party. In no event will Macro be liable for consequential, incidental, special, indirect, or exemplary damages arising out of this agreement or any work statement, however caused and (to the fullest extent permitted by law) under any theory of liability (including negligence), even if Macro has been advised of the possibility of such damages.

The scope of this report and review is limited to a review of only the code presented by the Sapience team and only the source code Macro notes as being within the scope of Macro's review within this report. This report does not include an audit of the deployment scripts used to deploy the Solidity contracts in the repository corresponding to this audit. Specifically, for the avoidance of doubt, this report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. In this report you may through hypertext or other computer links, gain access to websites operated by persons other than Macro. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such websites' owners. You agree that Macro is not responsible for the content or operation of such websites, and that Macro shall have no liability to your or any other person or entity for the use of third party websites. Macro assumes no responsibility for the use of third party software and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.