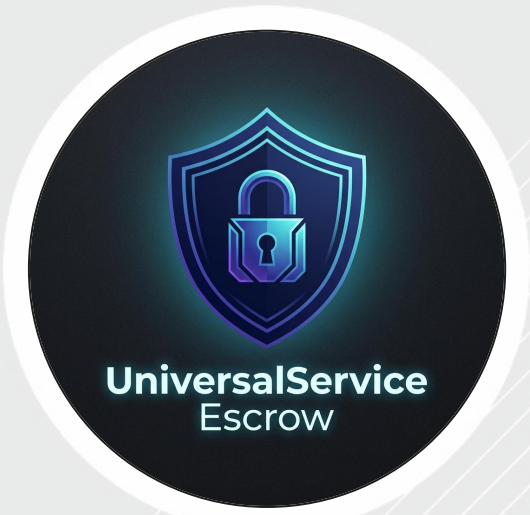
SPYWOLF

Security Audit Report



Audit prepared for

Universal Service Escrow (V4)

Completed on

June 24, 2026

TABLE OF CONTENTS

Project Information 01

Audit Methodology 02

Findings 03

Conclusion 04

About SPYWOLF 05

Disclaimer 06

PROJECT INFORMATION

- **Project Name:** Universal Service Escrow (V4)
- **Contract:** UniversalServiceEscrow
- **Category:** On-chain service escrow engine for B2B/B2C engagements (primarily KOL/influencer marketing). A sponsor's funds are locked on creation and released to the provider on delivery, refunded on cancellation, or split by the platform on dispute.
- **Network:** BNB Smart Chain (BSC Mainnet, chainId 56), with native auto-whitelisting of BUSD, USDT, USDC, and WBNB on that chain. Test configuration targets BSC Testnet (chainId 97) ahead of mainnet deployment.
- **Deployed Contracts:** Live on BSC Mainnet across five instances, all verified on BscScan:

  0xD5B180580D183A7A9278118312207bc8a9C9f89E
  0xa45f887b938a08B295A5b96b6559600632F09Ab0
  0x56c2227E06dBC16062179Be397839b101a8e58c7
  0x3EEEA456daCF2247CB0023a70923E60C3E13D6C3
  0x7986Bd37C4DA6d1822958fCB97E7a284b40DD7Cc

- **Token Standard:** BEP-20 / ERC-20.
- **Language:** Solidity 0.8.20.
- **Compiler:** solc 0.8.20, optimizer enabled at 200 runs.
- **Framework:** Hardhat, with a parallel Foundry test suite.
- **Libraries:** OpenZeppelin Contracts v5 — ReentrancyGuard, Ownable2Step, IERC20, and SafeERC20.
- **Files in Scope:** src/UniversalServiceEscrow.sol (the UniversalServiceEscrow contract), totalling 569 lines of code.
  Out of Scope: the six test-only mocks under src/mocks/ (MockUSDT, MockFeeToken, MockFlakyToken, MockGasGriefingToken, MockExoticToken, and MaliciousReceiver), which exist solely to exercise adversarial token behaviour in tests.
- **Core Functionality:** A six-state escrow lifecycle (FUNDED → RELEASED / DISPUTED / RESOLVED / CANCELLED) governs each engagement. Funds are credited via a pull-over-push model and withdrawn by beneficiaries; disputes resolve to a proportional split with fees applied only to the provider's portion; a stale-dispute fallback prevents permanent locking; per-token min/max limits, fee-on-transfer-safe deposit accounting, timelocked administrative parameters, and a balance-bounded stuck-token recovery round out the design.
- **Codebase:** Delivered as EscrowX_Audit_Package_V4.zip, containing the contract source, Hardhat and Foundry test suites, configuration, package manifest, and an architecture/audit guide.
  Audit Type: Full manual security review supported by proof-of-concept exploit development on a local EVM.

AUDIT METHODOLOGY(1)

The audit evaluated the UniversalServiceEscrow.sol smart contract on BNB Smart Chain, focusing on security, correctness, and alignment with the intended business logic. Since BSC is EVM-compatible, standard Ethereum auditing techniques were applied. Special attention was given to the escrow state machine and to the contract's custody model: funds move through a six-state lifecycle and are settled via a pull-over-push accounting system backed by totalLocked and totalWithdrawable invariants, so the review concentrated on whether every transition preserves those invariants and whether any party can extract value they are not owed.

Steps

Project Familiarization

- Reviewed the architecture guide and the documented V4 hardening claims (pull-over-push, bounded recovery, non-binary dispute resolution, evidence anchoring, stale-dispute fallback, timelocked admin, fee-on-transfer support).
- Mapped the intended engagement flow (fund → accept → release / timeout / cancel / dispute → resolve → withdraw) to the implemented state machine, noting which transitions are permitted from each state and which actors can trigger them.

Automated Analysis

- Static analysis (Slither, Mythril) to surface reentrancy, integer overflow/underflow, uninitialized storage, gas inefficiencies, and unchecked external-call patterns.
- Compilation under the project's own settings (solc 0.8.20, optimizer at 200 runs) to confirm the audited bytecode matches the deployment configuration.

Manual Code Review

- Line-by-line review of UniversalServiceEscrow.sol.
- Verified state-transition guards, access control on each lifecycle action, fee and dispute-split arithmetic, the totalLocked / totalWithdrawable accounting invariant, fee-on-transfer deposit measurement, the bounded recoverStuckTokens logic, and the pull-based withdraw / batchWithdraw paths.
- Checked checks-effects-interactions ordering, nonReentrant coverage, rounding behaviour in split settlements, the timelock queue/execute mechanism, and — critically — whether the dispute and stale-dispute outcomes can be steered by one party to the detriment of the other.

AUDIT METHODOLOGY(2)

Scenario Simulation

- Compiled the contract and exercised it against an in-memory EVM with proof-of-concept scripts driving real fund flows:
- Standard funding, acceptance, and happy-path release with fee deduction.
- Provider timeout claims and pre-acceptance client cancellations.
- Dispute opening, proportional resolution, and the stale-dispute fallback.
- Adversarial recipients and tokens (reverting, blacklisting, fee-on-transfer, gas-griefing) against the pull-payment and batch-withdraw paths.
- Cross-escrow and cross-token accounting to confirm one engagement cannot draw on another's locked funds or pending withdrawals.

Risk Assessment

- Classified findings into Critical, High, Medium, Low, and Informational based on exploitability and protocol impact.
- Special focus on fund-custody and settlement-fairness risks — whether disputes or timeouts can be manipulated to misallocate funds, and whether the recovery and withdrawal paths can ever exceed a party's true entitlement.

Tools Used

- Static analyzers: Slither, Mythril.
- Compilation: solc 0.8.20 (optimizer 200 runs) via Hardhat; Foundry for supplementary test execution.
- Scenario simulation: in-memory EVM (ganache) with ethers.js-driven PoC exploit scripts run against the compiled bytecode.
- Manual reasoning: Solidity 0.8.x checked-arithmetic guarantees and OpenZeppelin ReentrancyGuard, Ownable2Step, and SafeERC20 patterns.

Severity	Definition
Critical	Direct loss of user funds or protocol-level fund control is achievable by an external actor under realistic deployment conditions, with low cost and high reliability.
High	Significant loss of funds is possible but requires specific preconditions, or causes systemic protocol malfunction without immediate fund loss.
Medium	Limited loss of value, exploitable griefing, or economic logic deviations that meaningfully affect protocol behavior without direct fund theft.
Low	Minor deviations from best practice, dead code, gas inefficiencies, or edge cases with negligible practical impact.
Informational	Code quality observations, documentation gaps, or stylistic notes with no security or economic implication.

FINDINGS

🟨 Medium Risk ✅ RESOLVED

~~Stale-dispute resolution is client-biased — a client can reclaim funds for delivered work~~

resolveStaleDispute always refunds **100% to the client**, regardless of merit, and can be triggered by **either party** once the stale window passes. Two enabling facts turn this into an exploit: openDispute only checks for FUNDED status — it has no timeout bar, so it can be called even after timeoutDate — and entering DISPUTED blocks both releaseFunds and claimTimeout. A client can therefore let the provider accept and complete the work, open a dispute instead of releasing, wait out staleDisputeTimeout, and call resolveStaleDispute to claw back the entire amount. The provider has no symmetric remedy: if the provider calls the stale function, it still pays the client. Absent owner resolution, the stale path returns the provider's earned funds to the client in every case.

Impact:
A provider can lose 100% of legitimately earned funds, and a client can obtain services for free. PoC result: client funds 10,000 USDT, provider accepts and delivers, client opens a dispute, and after the 30-day window the client calls resolveStaleDispute then withdraw — ending with the full 10,000 USDT while the provider's withdrawable balance is 0. The exploit is gated behind 30-day owner inaction, which is why it is rated Medium under an active owner, but the directional bias is a logic flaw.

Code (excerpt):

```
// openDispute — no timeout bar; either party can freeze release/timeout
require(e.status == Status.FUNDED, "Escrow not in FUNDED state");   // L382

// resolveStaleDispute — always 100% to client, callable by either party
require(msg.sender == e.client || msg.sender == e.provider, "Not party to escrow"); // L436
require(block.timestamp > disputeOpenedAt[_escrowId] + staleDisputeTimeout, "...");  // L437
_credit(e.client, e.token, e.amount); // 100% refund to client, no fee  // L444
```

Implemented Fixes:
The team replaced the unconditional 100%-to-client refund with a fixed 50/50 split, applying the fee only to the provider's half, and now records the disputeOpener. Because neither party can obtain more than half through the stale path, the incentive for a client to accept delivered work and then claw back the full amount is removed. SpyWolf re-ran the original proof of concept against the revised contract and confirmed the client now receives exactly 50% and the provider 50% (net of fee), with the accounting invariant intact.

FINDINGS

🟨 Low Risk ✅ RESOLVED

~~A single failing token bricks the entire batchWithdraw~~

batchWithdraw iterates the caller's token list and performs a safeTransfer for each inside one transaction. If any single token reverts on transfer — a blacklisted recipient, a paused token, or one of the flaky/reverting tokens the project itself mocks — the whole batch reverts and **none** of the caller's withdrawals settle, including the healthy tokens. Because state is zeroed before the transfer, the revert simply rolls everything back, so no funds are lost; the caller is just blocked from using the batch path.

Impact:
Degraded availability, not fund loss. A user holding a credited balance in one problematic token cannot use batchWithdraw at all until that token is excluded from the list. The single-token withdraw remains a full workaround, so impact is limited to convenience.

Code (excerpt):

```
function batchWithdraw(address[] calldata _tokens) external nonReentrant {
    for (uint256 i = 0; i < _tokens.length; i++) {
        ...
        IERC20(token).safeTransfer(msg.sender, amount); // one revert kills the whole loop
    }
}
```

Implemented Fixes:
The per-token transfer was changed to a low-level call whose result is checked; on failure the function re-credits the user's withdrawable and totalWithdrawable balances and continues to the next token rather than reverting the entire batch. A single problematic token (blacklist, paused, flaky) can no longer block a user from withdrawing their other balances, and accounting stays consistent on a skipped entry.

FINDINGS

🟨 Low Risk ✅ RESOLVED

~~claimTimeout pays a provider who never accepted the job~~

claimTimeout requires only that the escrow is FUNDED and past timeoutDate; it does **not** check e.accepted. A provider who never accepted — and therefore never committed to the work — can still take the full amount (minus fee) once the timeout elapses. The client's only protections are to cancelEscrow (allowed only while unaccepted) or openDispute before the deadline; if the client does neither in time, an uncommitted provider is paid.

Impact:
A fairness/logic deviation rather than a theft primitive — the client chose the provider and set the timeout, and the timeout exists precisely as a provider-side deadman. But coupling payout to a deadline with no acceptance requirement means funds can flow to a provider who never engaged, which may not match user expectations for a "service delivered" release.

Code (excerpt):

```
function claimTimeout(uint256 _escrowId) external nonReentrant whenNotPaused {
    Escrow storage e = escrows[_escrowId];
    require(msg.sender == e.provider, "Only provider can claim timeout");
    require(e.status == Status.FUNDED, "Escrow not in FUNDED state");
    require(block.timestamp > e.timeoutDate, "Timeout not reached yet");
    // no require(e.accepted)
```

Implemented Fixes:
A require(e.accepted, "Provider never accepted") check was added, so the timeout payout is now available only to a provider who actually committed to the engagement. An uncommitted provider can no longer be paid when a funded escrow lapses.

FINDINGS

Informational

Unbounded loop in getActiveEscrowsByUser

The paginated getter iterates from escrow 1 up to escrowCounter, allocating a temporary array sized to the total escrow count. As an off-chain eth_call this is fine, but it grows linearly with all escrows ever created and is not safe to call from another on-chain contract — at scale it can exceed the block gas limit and revert. Best consumed via event indexing (e.g., The Graph) rather than on-chain iteration.

De-whitelisting a token clears its stored symbol

setTokenAllowed(..., false) deletes the entire TokenConfig, including symbol. Any escrow still active in that token will then return an empty string from getEscrowDetails and tokenSymbol. Funds remain fully withdrawable (settlement reads e.token directly, not the config), so this is purely a cosmetic/UI degradation for in-flight escrows in a token that was later removed.

feeBPS is read from storage at settlement but frozen per escrow at creation

Each escrow correctly snapshots defaultFeeBPS into e.feeBPS at creation, so later fee changes don't affect existing escrows — good. Worth noting only that the fee is applied to the full amount on releaseFunds/claimTimeout but to the provider's portion only on resolveDispute; this is intentional per the design guide, but the asymmetry should be documented for integrators.

Redundant tokenSymbol / tokenSymbols duplication

tokenSymbol and tokenSymbols are byte-for-byte identical, the latter kept only as a V2/V3 ABI alias. Harmless, but it adds a small amount of bytecode and surface area; consider removing once no integrators depend on the old name.

acceptDelaySeconds defaults to zero

The optional anti-front-running delay between escrow creation and provider acceptance is 0 by default, so acceptance can occur in the same block as funding. This is the intended default and not a flaw, but deployments relying on the delay as a safeguard must set it explicitly via the timelocked queueLimits / executeLimits flow.

CONCLUSION

SpyWolf performed a full security review of the UniversalServiceEscrow.sol contract, combining line-by-line manual analysis with proof-of-concept exploit development against the compiled bytecode on a local EVM. The review focused on the six-state escrow lifecycle and on whether every transition preserves the contract's custody invariants and settles funds to the correct party. Following delivery of the initial report, the development team remediated every finding and submitted revised code, which SpyWolf re-reviewed and re-tested.

This is a well-constructed contract, and notably stronger than the team's earlier work. The custody model is sound: funds move through a strict status machine guarded by checks-effects-interactions ordering and nonReentrant on every state-changing entry point, settlement uses a correct pull-over-push pattern, deposits are measured by balance delta to support fee-on-transfer tokens, dispute splits are arithmetically exact with no dust, and the recoverStuckTokens function is provably bounded by balanceOf - totalLocked - totalWithdrawable so it can never touch committed funds. SpyWolf found no path by which one escrow can draw on another's locked funds or pending withdrawals, no reentrancy vector, and no theft primitive available to an external actor. The accounting invariant holds across all lifecycle transitions.

The single Medium finding was not a custody flaw but a settlement-fairness flaw: the stale-dispute fallback originally refunded 100% to the client and could be triggered by either party, which — confirmed by a working proof of concept — allowed a client to accept delivered work, open a dispute, wait out the stale window, and reclaim the full amount, leaving the provider with nothing. In the revised code the team replaced this with a fixed 50/50 split, applying the fee only to the provider's half and recording the dispute opener, so neither party can obtain more than their share through the stale path. SpyWolf re-ran the original exploit against the corrected contract and confirmed the client now receives exactly 50% and the provider 50% net of fee, with the accounting invariant intact. The two Low findings were likewise resolved: batchWithdraw now re-credits and skips a failing token instead of reverting the entire batch, and claimTimeout now requires that the provider actually accepted the engagement before a timeout payout can be claimed. The informational items concerning the unbounded view getter, an ABI-compatibility alias, and documentation semantics were acknowledged and carry no fund-security impact.

Recommendation: All findings from this engagement have been resolved and independently re-verified, and no new issues were introduced by the changes. On the basis of the reviewed and corrected code, SpyWolf considers the contract suitable for mainnet deployment. The development team's adoption of pull-over-push settlement, bounded recovery, timelocked administration, and fee-on-transfer-safe accounting reflects a mature security posture, and the engagement is considered closed.

This report reflects the state of the reviewed code at the time of audit and does not constitute financial advice or a guarantee against future vulnerabilities, particularly in code modified after this review.

SPYWOLF

CRYPTO SECURITY

ABOUT US

We are a growing crypto security agency offering audits, KYCs and consulting services for some of the top names in the crypto industry.

- **OVER 700 SUCCESSFUL CLIENTS**
- **MORE THAN 1000 SCAMS EXPOSED**
- **MILLIONS SAVED IN POTENTIAL FRAUD**
- **PARTNERSHIPS WITH TOP LAUNCHPADS, INFLUENCERS AND CRYPTO PROJECTS**
- **CONSTANTLY BUILDING TOOLS TO HELP INVESTORS DO BETTER RESEARCH**

Audits | KYCs | dApps
Contract Development

To hire us, reach out to
contact@spywolf.co or
t.me/joe_SpyWolf

FIND US ONLINE

SPYWOLF.CO

@SPYWOLFNETWORK

@SPYWOLFNETWORK

Disclaimer

This report shows findings based on our limited project analysis, following good industry practice from the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, overall social media and website presence and team transparency details of which are set out in this report. In order to get a full view of our analysis, it is crucial for you to read the full report.
While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us on the basis of what it says or doesn't say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below – please make sure to read it in full.

DISCLAIMER:
By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any and all copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis, and does not constitute investment advice.
No one shall have any right to rely on the report or its contents, and SpyWolf and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers and other representatives) (SpyWolf) owe no duty of care towards you or any other person, nor does SpyWolf make any warranty or representation to any person on the accuracy or completeness of the report.
The report is provided "as is", without any conditions, warranties or other terms of any kind except as set out in this disclaimer, and SpyWolf hereby excludes all representations, warranties, conditions and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, SpyWolf hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against SpyWolf, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of use of this report, and any reliance on this report. The analysis of the security is purely based on the smart contracts, website, social media and team.
No applications were reviewed for security. No product code has been reviewed.