



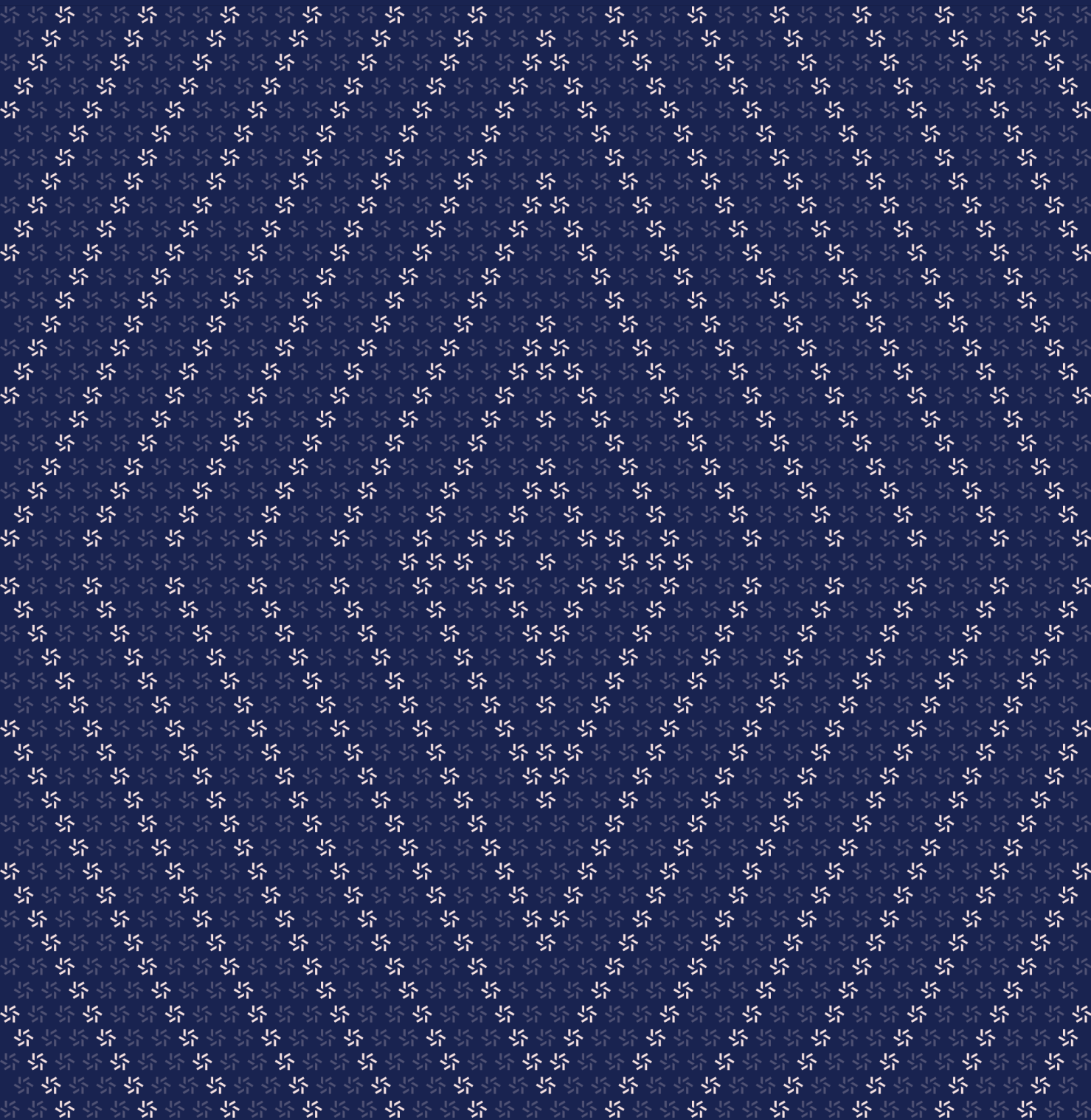
Prepared for
Arshan Khanifar
Kartik Chopra
Spencer Solit
Ritual

Prepared by
Filipe Alves
Katerina Belotskaia
Zellic

August 27, 2026

Ritual (Smart Contracts)

Smart Contract Security Assessment



Contents

About Zellic	5
1. Overview	6
1.1. Executive Summary	6
1.2. Goals of the Assessment	6
1.3. Non-goals and Limitations	6
1.4. Results	7
2. Introduction	8
2.1. About Ritual (Smart Contracts)	8
2.2. Methodology	8
2.3. Scope	10
2.4. Project Overview	10
2.5. Project Timeline	11
3. Detailed Findings	12
3.1. Any account can mint unlimited RITUAL for 0.01 RITUAL by scheduling a call with <code>gas = 0</code>	12
3.2. A <code>cancel</code> in the same block as a due execution makes <code>Scheduler.execute</code> revert, which halts block production	18
3.3. Far-future start blocks evade the residency cap and force an unbounded array scan on every scheduled execution	22
3.4. Expired jobs are designed to be recommittable, but the escrow life cycle makes recommitment impossible	25
3.5. Unbounded stored blobs make user-initiated cleanup exceed block gas	28

3.6.	A registered executor can claim any agentId and make its owner pay for revivals it never approved	30
3.7.	Measured callback gas is not capped by the callback gas limit	32
3.8.	Neither AsyncDelivery.settle or AsyncDelivery.deliver validate the job's expiryBlock	35
3.9.	Revoked executors retain heartbeat authority	38
3.10.	Final TTL skip leaves schedule stuck	41
3.11.	Insufficient test coverage	43
3.12.	A zero delivery callback gas limit is admitted end to end, guaranteeing the Phase 2 callback fails	45
3.13.	Dead balance restoration before revert in the scheduled fee-deduction path	47
3.14.	Redundant isValid check in _validateExecutor and _scheduledExecutorInvalidReason is unreachable	49
3.15.	Scheduled executions and Phase 2 deliveries collect gas payments but never disburse them	51
4.	Discussion	54
4.1.	The return value of _remove is silently ignored	54
4.2.	EIP-7702 delegated accounts bypass the contract-only check on scheduling	55
5.	System Design	56
5.1.	Component: RitualWallet	56
5.2.	Component: Scheduler	58
5.3.	Component: AsyncJobTracker	61
5.4.	Component: AsyncDelivery	63
5.5.	Component: AgentHeartbeat	64

6.	Assessment Results	67
6.1.	Observations and recommendations	67
6.2.	Recommended follow-up assessment	67
6.3.	Disclaimer	67

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the [#1 CTF \(competitive hacking\) team](#) [worldwide](#) in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io [and](#) follow [@zellic_io](#) [on Twitter](#). If you are interested in partnering with Zellic, contact us at hello@zellic.io [.](#)



1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Ritual from July 13th to July 27th, 2026. During this engagement, Zellic reviewed Ritual's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Is there any way that a system transaction can be jailbroken out of the system contracts?
 - Are there any griefing vectors whereby a user would be able to create many jobs in an unbounded way?
 - Are there any paths that could overpay, underpay, or permanently lock user funds in RitualWallet?
 - Can malicious callbacks or predicates cause reentrancy, corrupt bookkeeping, or create a chain-liveness issue?
 - Given that these contracts form part of the chain's core protocol logic, are their configurability, privileged administration, and upgradability mechanisms appropriately constrained, and what governance and operational processes should be followed when making changes?
 - Can any async job or scheduled execution be committed, settled, delivered, expired, or cleaned up more than once or in the wrong order?
-

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- The node-side (reth) construction and validation of system transactions, including the independent verification each validator performs on the sender, target, calldata, ordering, and on-chain preconditions
- The off-chain executor, TEE attestation, and precompile infrastructure (for example, the TEE service registry, capability policies, and agent-launch precompiles) that these contracts rely on and trust
- Front-end components
- Infrastructure relating to the project
- Key custody and the operational security of the reserved system accounts

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped Ritual (Smart Contracts) contracts, we discovered 15 findings. Two critical issues were found. One was of high impact, four were of medium impact, five were of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Ritual in the Discussion section (4. 7).

Breakdown of Finding Impacts

Impact Level	Count
Critical	2
High	1
Medium	4
Low	5
Informational	3



2. Introduction

2.1. About Ritual (Smart Contracts)

Ritual contributed the following description of Ritual's smart contracts:

Core set of smart contracts that allow for net-new functionality for EVM chains including asynchronous execution for computationally intensive & non-deterministic operations and enshrined scheduling of txs without needing external invocation.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect

its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion ([4.7](#)) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Ritual (Smart Contracts)

Type	Solidity
Platform	EVM-compatible
Target	ritual-sc-core
Repository	https://github.com/ritual-net/ritual-sc-core
Version	80bfa10bc123a9b3dd05f14e92ebb9c7ab17780b
Programs	async/AsyncDelivery.sol async/AsyncJobTracker.sol heartbeat/AgentHeartbeat.sol scheduler/Scheduler.sol wallet/RitualWallet.sol

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 3.2 person-weeks. The assessment was conducted by two consultants over the course of 11 calendar days.

Contact Information

The following project manager was associated with the engagement:

Jacob Goreski
✂ Engagement Manager
jacob@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Filipe Alves
✂ Engineer
filipe@zellic.io ↗

Katerina Belotskaia
✂ Engineer
kate@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

July 13, 2026 Start of primary review period

July 14, 2026 Kick-off call

July 27, 2026 End of primary review period

3. Detailed Findings

- 3.1. Any account can mint unlimited RITUAL for 0.01 RITUAL by scheduling a call with gas = 0

Target	Scheduler, ritual-revm (revm-handler)		
Category	Business Logic	Severity	Critical
Likelihood	High	Impact	Critical

Description

The `_schedule` function validates `maxFeePerGas` in two places, but both validations can be satisfied while the price remains unbounded, because neither is expressed in terms of the price alone.

```
// Scheduler.sol, _schedule
// EIP-1559 fee fields are u128 in the execution layer; reject values that
// would overflow
if (maxFeePerGas > type(uint128).max || maxPriorityFeePerGas >
    type(uint128).max) {
    revert FeeExceedsU128();
}

if (maxFeePerGas < MIN_SCHEDULED_MAX_FEE_PER_GAS) {
    revert MaxFeePerGasBelowFloor();
}

// Scheduled calls are intentionally funded just-in-time: creation only
// proves the first execution can be paid, and every later execution
// re-checks/consumes balance and allowance. A missed payment evicts the
// remaining schedule instead of requiring lifetime escrow up front.
uint256 executionFee = SCHEDULED_EXECUTION_FEE();
uint256 maxExecutionCost = (maxFeePerGas * gas) + value + executionFee;
if (sponsor != msg.sender && RITUAL_WALLET.allowance(sponsor, msg.sender) <
    maxExecutionCost) {
    revert UnauthorizedSponsor();
}

// Validate sponsor has sufficient funds in system wallet
if (RITUAL_WALLET.availableBalance(sponsor) < maxExecutionCost) {
    revert InsufficientWalletFunds();
}
```

The first two checks bound `maxFeePerGas` to a 128-bit range (`[7, type(uint128).max]`) and nothing tighter — `type(uint128).max` wei per gas is many orders of magnitude beyond any plausible fee and is accepted outright. The only check standing between naming that price and having it admitted is the balance gate, `availableBalance(sponsor) >= maxExecutionCost`. But `maxExecutionCost` prices the sponsor on `maxFeePerGas * gas`, a product, and `gas` — the `uint32` callback gas limit — has no lower bound anywhere in `_schedule`. Setting `gas = 0` zeroes the product regardless of what `maxFeePerGas` is, collapsing the entire balance requirement to the fixed `SCHEDULED_EXECUTION_FEE` (`0.01 RITUAL`, a value that does not depend on the schedule at all).

So the balance gate does not actually gate price. It checks `availableBalance(sponsor) >= 0.01 RITUAL` for every schedule with `gas = 0`, whatever `maxFeePerGas` says. As a result, a sponsor holding the fixed fee and nothing else can schedule a call priced at `type(uint128).max` wei per gas exactly as easily as one priced at the 7-wei floor.

The `maxFeePerGas` value reaches the block beneficiary directly. When the scheduled call executes, the node builds the corresponding transaction with fee fields copied verbatim from this same `Call`:

```
// ritual-evm/src/tx.rs
impl FromRecoveredTx<TxScheduled> for TxEnv {
    fn from_recovered_tx(tx: &TxScheduled, caller: Address) -> Self {
        Self {
            gas_price:
                u128::try_from(tx.max_fee_per_gas).unwrap_or(u128::MAX),
            gas_priority_fee: Some(
                u128::try_from(tx.max_priority_fee_per_gas).unwrap_or(u128::MAX),
            ),
            // ...
        }
    }
}
```

The `gas_price` and `gas_priority_fee` fields on `TxEnv` are exactly `maxFeePerGas` and `maxPriorityFeePerGas`, unchanged. The `reward_beneficiary` function (`crates/handler/src/post_execution.rs`) turns those two fields into the actual credit in two steps:

```
// context/interface/src/transaction.rs - effective_gas_price()
let max_price = self.gas_price(); // =
    maxFeePerGas
let max_priority_fee = self.max_priority_fee_per_gas().unwrap(); // =
    maxPriorityFeePerGas
min(max_price, base_fee.saturating_add(max_priority_fee))
```

```
// handler/src/post_execution.rs - reward_beneficiary()
```

```

let effective_gas_price = tx.effective_gas_price(basefee);
let coinbase_gas_price = effective_gas_price.saturating_sub(basefee);
journal.load_account_mut(block.beneficiary())?
    .incr_balance(U256::from(coinbase_gas_price * gas.used() as u128));

```

So `maxFeePerGas` reaches the mint through a straight line: `effective_gas_price` is the smaller of `maxFeePerGas` and `basefee + maxPriorityFeePerGas`. Naming `maxPriorityFeePerGas == maxFeePerGas` makes the second term the larger one whenever `basefee > 0`, so `effective_gas_price` resolves to `maxFeePerGas` itself. Subtracting `basefee` (negligible next to `maxFeePerGas`) barely changes it, and the result is multiplied by `gas.used()` and credited to the beneficiary.

Nothing debits the account this credit is drawn from — the sponsor's `_schedule`-time balance check only ever verified 0.01 RITUAL was available, and `execute()`'s own deduction (`deductScheduledExecutionFeesFrom`) is charged on the same `maxFeePerGas * gas` product, which is also zero. The price the balance gate should have constrained is the same price that ends up sizing the mint directly, and it lands as a real native-token credit to whoever proposes the block.

Confirmed against the real compiled contract

The balance-gate bypass is directly reproducible against the compiled `Scheduler.sol` bytecode, deployed and executed inside Ritual revm's system-transaction handler:

```

#[test]
fn schedule_u128_max_price() {
    let mut h = Harness::setup(EXECUTION_FEE);
    println!("sponsor wallet funded with: {EXECUTION_FEE} wei (0.01 RITUAL)");

    // `gas = 0`, price = the literal ceiling `_schedule` allows
    // (`type(uint128).max`).
    h.schedule(0, START_BLOCK_U32, 1, 1, 10, u128::MAX, u128::MAX);
    println!("schedule() at maxFeePerGas = maxPriorityFeePerGas =
type(uint128).max: ADMITTED (no revert)");

    h.evm.ctx.block.number
    = revm_handler::primitives::U256::from(START_BLOCK + 1);

    let beneficiary_before = h.balance_of(BENEFICIARY);
    let res = h.execute_scheduled(0, u128::MAX, u128::MAX);
    assert!(res.is_success(), "execute failed: {res:?}");
    let gas_used = res.gas().tx_gas_used();
    let beneficiary_after = h.balance_of(BENEFICIARY);

    println!("gas consumed by execute(): {gas_used}");
    println!("beneficiary balance before: {beneficiary_before}");

```

```
println!("beneficiary balance after: {beneficiary_after}");
println!("u128::MAX for reference: {}", u128::MAX);

assert!(beneficiary_after > beneficiary_before, "no credit was minted");
}
```

```
running 1 test
sponsor wallet funded with: 10000000000000000 wei (0.01 RITUAL)
schedule() at maxFeePerGas = maxPriorityFeePerGas = type(uint128).max:
  ADMITTED (no revert)
gas consumed by execute(): 101032
beneficiary balance before: 0
beneficiary balance after: 340282366920938463463374506399768110424
u128::MAX for reference: 340282366920938463463374607431768211455
```

That test funds a sponsor's MockRitualWallet contract with exactly EXECUTION_FEE (0.01 RITUAL) and then calls the Scheduler.schedule(..., gas = 0, maxFeePerGas = u128::MAX, maxPriorityFeePerGas = u128::MAX, ...). The _schedule function admits it against the 0.01 RITUAL balance, execute() runs to completion, and the beneficiary is credited 340282366920938463463374506399768110424 wei.

Impact

Any unprivileged account can mint RITUAL without limit, for the price of a single 0.01 RITUAL schedule() call.

The balance gate is the only check tying a schedule's price to real, verifiable backing, and gas = 0 reduces it to availableBalance >= 0.01 RITUAL no matter what price is named. That price is not inert; it is the same price copied verbatim into the credit the block beneficiary receives (see above), with no debit anywhere to fund it. Because a failed gas = 0 callback is treated as a normal outcome rather than a malformed schedule, none of this needs special handling to succeed — it is admitted, executed, and paid out on chain in the same transaction, every time.

Recommendations

Require gas to be nonzero and large enough to cover the accounting work execute() performs on the call's behalf:

```
if (gas < MIN_SCHEDULED_CALLBACK_GAS) {
    revert CallbackGasBelowFloor();
}

if (maxFeePerGas > type(uint128).max || maxPriorityFeePerGas >
    type(uint128).max) {
```

```
    revert FeeExceedsU128();  
}
```

That restores the balance gate's intended meaning, since `maxExecutionCost` can then no longer collapse to a constant regardless of `maxFeePerGas`. It should be treated as necessary but not sufficient; a `MIN_SCHEDULED_CALLBACK_GAS` set too low still lets `maxFeePerGas` be named far above what the sponsor's balance would support at any gas figure resembling real usage, since the gate would then require only `maxFeePerGas × MIN_SCHEDULED_CALLBACK_GAS`, a number an attacker can still hold while naming an extreme price.

Bound `maxFeePerGas` against something economically real, independent of the balance check, rather than against the width of its own storage type. A ceiling expressed as a multiple of the prevailing base fee, or an absolute cap set by the owner alongside `SCHEDULED_EXECUTION_FEE`, keeps the field's range inside values a real wallet could plausibly back, so that no single choice of gas can put the field's range and the balance check at odds with each other.

None of the above removes the mint, however; it only bounds its size. The `reward_beneficiary` function is the only gas path in the handler carrying no system-transaction guard: `should_skip_gas_deduction`, `should_skip_balance_check`, and `should_skip_gas_reimbursement` all return true for tx type `0x10`, so no caller is ever debited, yet the beneficiary is credited regardless — making that credit new supply rather than a fee someone paid. The missing guard completes the family,

```
pub fn reward_beneficiary<CTX: ContextTr>(  
    context: &mut CTX,  
    gas: &Gas,  
) -> Result<(), <CTX::Db as Database>::Error> {  
    if should_skip_beneficiary_reward_for_ritual_system_tx(context.tx()) {  
        return Ok(());  
    }  
    let (block, tx, cfg, journal, _, _) = context.all_mut();
```

where the local wrapper mirrors `should_skip_gas_reimbursement_for_ritual_system_tx` directly below it in the same file, forwarding to a new `should_skip_beneficiary_reward(tx) = is_system_transaction(tx)` beside its five siblings in `ritual-revm-system-tx`. Defined that way, it also covers tx types `0x11`, `0x12`, and `0x13`, which skip the same debit and therefore credit the beneficiary from nothing for the same reason.

The block producer should still get paid, and the funds for it are already sitting in the contract, unused. So the producer's share can be paid straight out of that funds:

```
uint256 gasUsed = startGas - gasleft();  
  
uint256 minerFee = priorityFee * gasUsed;  
if (minerFee != 0) {
```

```
RITUAL_WALLET.depositFor{value: minerFee}(block.coinbase);  
}
```

Remediation

Note: As this remediation was implemented partly in out-of-scope code, we were only able to review it on a best effort basis. See [6.2](#) ↗ for more information.

This issue has been acknowledged by Ritual, and fixes were implemented in the following commits:

- [a1a4bd60](#) ↗
- [9bdfdbc0](#) ↗
- [31e719c0](#) ↗

Ritual provided the following comment:

For this solution, we decided to completely disable the default execution-layer proposer payments for system transactions. We viewed this as a stronger way to prevent implicit tip payments in the node code; proposer payments are instead distributed explicitly inside the smart contracts from sponsor-funded budgets. Part of the fix is in REVM, where we disable the default beneficiary reward for system transactions. The same class of attack would also have been possible with a two-phase async transaction. The REVM fix covers those system transactions as well. The other part of the fix is in the contracts, where we explicitly pay `block.coinbase`. Validator payment failures revert the system transaction, causing Reth to reject the containing block because mandatory system transactions must execute successfully.

- 3.2. A `cancel` in the same block as a due execution makes `Scheduler.execute` revert, which halts block production

Target	Scheduler		
Category	Business Logic	Severity	Critical
Likelihood	High	Impact	Critical

Description

The node decides which scheduled executions are due by reading `Scheduler` storage at the **parent** state, fabricates a `Scheduler.execute` system transaction for each of them, and then runs those alongside ordinary user transactions from a single queue ordered by effective tip. A `cancel` paying more than the schedule's `maxPriorityFeePerGas` therefore executes before the execution the node has already committed to including, and the two see different views of the same storage: the node selected the call while it existed, the contract resolves it after it is gone.

The `cancel` function removes every trace the resolution path depends on:

```
_clearBitmap(callId);

// Free the slot back to the available pool
uint256 slotIndex = callToSlot[callId];
_freeSlot(slotIndex);

// Remove the call ID mapping and auxiliary storage
delete callToSlot[callId];
```

The `_execute` function resolves the target through `_getCall`, which reverts on exactly that state:

```
function _getCall(uint256 callId) internal view returns (Call storage call) {
    uint256 slotIndex = callToSlot[callId];
    if (slotIndex >= slots.length) {
        revert CallNotFound();
    }
    call = slots[slotIndex];
    if (call.callId != callId || call.to == address(0) || call.state ==
    CallState.EXPIRED) {
        revert CallNotFound();
    }
}
```

A revert in the top frame of a scheduled system transaction is not an ordinary failed transaction. Because system transactions bypass EVM gas charging, their contribution to the block gas limit is reconstructed from the terminal event the Scheduler emits, and a reverted execution emits none. The node cannot account for the transaction, cannot include it, and — since the obligation was derived from parent state, where the cancellation is invisible — cannot drop it either. Payload construction is aborted and the proposer publishes nothing for that slot.

The `cancel` function requires only `msg.sender == call.to`, so the party able to invalidate a due execution mid-block is the schedule's own target — which means anyone willing to deploy a contract and schedule against it.

Additionally, three other branches already have the opposite problem: they return successfully, but emit an event the matcher does not know. Overall expiry emits `CallExpired`, a per-execution TTL lapse emits `CallSkippedTTLExpired`, and a base-fee spike above `maxFeePerGas` emits `CallSkippedInsufficientFunds` and returns without evicting. Reaching any of the three produces `MissingScheduledTerminalEvent` and the same aborted build.

`CallNotFound` is likewise not the only revert reachable in the top frame. The predicate branch reverts on both of its outcomes, `PredicateCheckFailed` and `PredicateReturnedFalse`, even though a predicate returning false is an ordinary, expected result rather than an error, and a predicate that flips between the node's evaluation and execution reproduces this finding exactly.

Impact

Block production stops, and it does not restart on its own. While no block is produced the `cancel` cannot land, so it stays pending in every pool that received it; the due window is keyed on block number, so with the height frozen the same obligation is due again on every subsequent attempt. Each build attempt reproduces the same tip ordering, hits the same revert, and fails identically.

Triggering this takes one schedule and one cancellation. The attacker deploys a contract, schedules a call through it, and cancels that call in the block where an execution is due. Since `cancel` requires `msg.sender == call.to`, the attacker controls both sides of the ordering — the schedule's `maxPriorityFeePerGas` and the tip paid on the cancellation — so the cancellation sorts ahead of the execution in the tip-ordered queue. The same sequence also occurs by accident whenever a contract cancels its own schedule in a block where one of its executions is due. The bug also has a mild accidental form: a contract that cancels its own schedule in a block where one of its executions happens to be due triggers the revert without intending to. That costs a slot or two rather than the chain, because nothing keeps the cancellation ahead of the system transaction across attempts.

Recommendations

Treat an unresolvable call the way the neighbouring branches already treat expiry and TTL: complete successfully and record the outcome in an event, so the execution has a terminal state the node can account for.

Declare the event alongside the other terminal events. Only the identifiers are still available, since

the slot is gone, and callId must remain indexed because the node matches on the first topic:

```
/// @notice Emitted when a due execution can no longer be resolved, e.g. the
///         call was cancelled
///         earlier in the same block. Terminal outcome for this system
///         transaction: no callback
///         ran and no fee was charged.
event CallSkippedNotFound(uint256 indexed callId, uint256 executionIndex);
```

Split the existence check out of _getCall rather than changing it, since cancel, getCallState, calls, and the owner recovery path all rely on it reverting:

```
function _findCallSlot(uint256 callId) internal view returns (bool found,
    uint256 slotIndex) {
    slotIndex = callToSlot[callId];
    if (slotIndex >= slots.length) {
        return (false, 0);
    }
    Call storage call = slots[slotIndex];
    if (call.callId != callId || call.to == address(0) || call.state ==
    CallState.EXPIRED) {
        return (false, 0);
    }
    return (true, slotIndex);
}
```

Then resolve through it in _execute:

```
(bool found, uint256 slotIndex) = _findCallSlot(callId);
if (!found) {
    emit CallSkippedNotFound(callId, executionIndex);
    return;
}
Call storage call = slots[slotIndex];
```

The contract change is inert on its own — the transaction would succeed and the node would still fail to account for it, only with a different error — so the new event has to be added to the node's set of accepted terminal events in the same change.

Remediation

Note: As this remediation was implemented partly in out-of-scope code, we were only able to review it on a best effort basis. See [6.2](#), ↗ for more information.

This issue has been acknowledged by Ritual, and a fix was implemented in commit [0d312b6a](#) ↗.

Ritual provided the following comment:

Scheduled transaction reverts remain invalid for block verification. We are not catching that in a specific event and raising it.

The block builder executes the scheduled transaction, realizes that the error is a call not found, and simply skips over that transaction. Therefore, the block builder does not include it.

Given that we have obligations and censorship-resistant protection for scheduled transactions, the block verifier needs to be convinced that the block builder did not censor. Which is why we collect all of the events raised in case of cancellations from the scheduler contract. In case there exists an event that canceled that call ID, then we do not attribute fault due to censorship resistance to that block builder.

In other words: if `Set(call id of all missing scheduled transactions) - set (call id of cancellations observed) = {}` then the block is valid & is not censoring.

3.3. Far-future start blocks evade the residency cap and force an unbounded array scan on every scheduled execution

Target	Scheduler		
Category	Business Logic	Severity	High
Likelihood	High	Impact	High

Description

A contract caller can create arbitrarily many schedules whose `startBlock` is near `type(uint32).max`, because the lifespan check measures only `expiryBlock - startBlock` rather than residency from the current block. A one-call schedule with a zero time to live (TTL) therefore passes with a nominal lifespan of zero while occupying a slot for decades, and just-in-time funding lets the same single-execution wallet balance validate every such schedule without reserving it.

Once `slots.length` passes `CLEANUP_THRESHOLD` (1,000), every `execute` invocation runs `_cleanupExpiredCallsScan` before it even loads the call it was asked to run. That scan caps the number of expired calls removed, not the number of slots visited:

```
// Scheduler._cleanupExpiredCallsScan
for (uint256 i = 0; i < slots.length && cleaned < MAX_CLEANUP_PER_EXECUTE;) {
```

When the attacker's far-future calls are not expired, `cleaned` never advances, the early-exit condition is never reached, and the loop walks the entire attacker-inflated array performing a cold read and an expiry computation per entry.

That scan cannot be stopped by running out of gas, however. The payload builder constructs these transactions with `gas_limit = u64::MAX(SCHEDULED_GAS_LIMIT_SENTINEL)`, and `verify_scheduled_transactions_full` rejects any scheduled transaction whose limit is not exactly that sentinel, so the value is consensus-enforced rather than incidental. The revm fork skips the caller gas-limit comparison for system transaction types, and the figure the payload builder checks against remaining block gas is not the execution cost but the transaction's calldata footprint, roughly 21,000 gas for `execute`'s 68-byte input, as `effective_gas_limit_for_block` states directly: "System transactions use unlimited EVM execution gas, but block accounting is bounded by their deterministic input footprint".

Impact

The consequence is resource exhaustion rather than gas exhaustion, and it lands on the node rather than on the transaction. Each scheduled execution carries a full traversal of the inflated array as real EVM work with no gas bound, dominated by cold storage reads, which the payload

builder performs while building and every validating node repeats while importing. Since the sweep runs at the top of `execute`, before the requested call is even loaded, the cost is paid on every scheduled execution in every block for as long as the records persist, and it grows linearly with the number of records the attacker has added.

The requested callback still executes, so this is degradation of block production rather than failure of the scheduled path — but degradation with no ceiling, which is the more damaging shape of the two. An out-of-gas failure would at least be contained: the transaction would revert, the block would proceed, and the cost would be bounded by a gas limit. Here, nothing bounds it and nothing pays for it, so the attacker converts a priced write into unpriced work repeated by the whole network on every subsequent block.

Combined with Finding [3.1](#), that unpriced work also becomes minted revenue. The credit a scheduled transaction pays the block beneficiary is computed over `gasUsed()` of the whole system transaction, which includes this sweep, while the sponsor is charged only over the callback's own gas.

Recommendations

Bound a schedule's lifetime to the window between the current block and its final expiry, so schedules that start too far in the future are rejected:

```
uint256 lifespan = expiryBlock - uint256(startBlock);
uint256 lifespan = expiryBlock - block.number;
```

The subtraction cannot underflow, since `_schedule` already rejects `startBlock <= block.number` and `expiryBlock >= startBlock`.

Separately, cap the automatic cleanup by the number of slots it visits rather than by the number of expirations it finds, so `execute` cannot be forced to walk an array an attacker has inflated.

Remediation

Note: As this remediation was implemented partly in out-of-scope code, we were only able to review it on a best effort basis. See [6.2](#) for more information.

This issue has been acknowledged by Ritual, and fixes were implemented in the following commits:

- [cd431783](#)
- [e8101d58](#)
- [9bfdabc0](#)

Ritual provided the following comment:

We moved expired-call discovery into Reth, where it can operate efficiently over canonical scheduler state without imposing an unbounded EVM scan on every execution. Reth includes an explicit, deterministic batch of at most 25 expired call IDs in the scheduled transaction. The Scheduler caps each batch at 25 IDs and independently validates each supplied ID, removing it only if it is actually expired. This eliminates the full-array contract scan while keeping cleanup bounded and contract-enforced. We also changed Scheduler pricing to account for how long a job occupies contract storage. At scheduling time, the sponsor pays a non-refundable residency fee calculated as $\text{SCHEDULED_RESIDENCY_FEE_PER_BLOCK} \times (\text{expiry-Block} - \text{block.number})$. Consequently, jobs scheduled far into the future remain supported, but their cost increases linearly with their residency duration, preventing inexpensive long-term storage griefing. The residency fee is paid directly to `block.coinbase` rather than accruing in the Scheduler contract.

3.4. Expired jobs are designed to be recommittable, but the escrow life cycle makes recommitment impossible

Target	AsyncJobTracker		
Category	Business Logic	Severity	Medium
Likelihood	Medium	Impact	Medium

Description

When a commitment expires without ever being settled, the natural recovery path is to commit the same work again, and any such retry necessarily reproduces the same `job.id`. The tracker is written to support exactly this. When a new commitment arrives for an identifier that already exists, `_prepareJob` accepts it as long as the previous instance has expired, clearing the stale tracker state so the job can be stored anew.

```
function _prepareJob(AddJobInput calldata job) internal {
    if (jobExecutor[job.id] != address(0)) {
        require(expiryBlock[job.id] < block.number, "active job exists");
        _remove(job.id, false);
    }
    // [...]
}
```

The recommitment process, however, can never complete, because the second half of every commitment is a wallet reservation, and the wallet does not share the tracker's view of identifier life cycles.

The commitment functions reserve funds using `job.id` as the `escrowId`, so a recommitted job needs a fresh escrow under an identifier the wallet has already seen. The wallet treats such an identifier as permanently consumed. An escrow transitions from `NONE` to `ACTIVE` and then to a terminal `EXPIRED` or `SETTLED` state, never back to `NONE`, and the reservation path rejects any identifier whose escrow is not in the `NONE` state.

```
function _reserveFrom(/* [...] */) internal {
    _sweepExpired(sponsor, expiredEscrows);

    if (_escrows[escrowId].status != EscrowStatus.NONE) {
        revert EscrowAlreadyExists();
    }
    // [...]
}
```

Both `addJobAndReserve` and `addScheduledJobAndReserve` go through this recommitment path. In the case of `addScheduledJobAndReserve`, however, the failure does not even surface as a revert. The scheduled variant attempts the reservation with `tryReserveFrom`, which returns a failure flag instead of reverting, and routes the failure into an eviction.

```
(bool ok, IRitualWallet.ReserveFailure reason)
    = ritualWallet.tryReserveFrom(/* [...] */);

if (ok) {
    // [...]
} else {
    ISchedulerAsyncBridge(scheduler)
        .evictScheduledAsync(schedule.callId, schedule.executionIndex, job.id,
            bytes32(uint256(reason)));
    // [...]
}
```

Because `_prepareJob` has already deleted the previous tracker entry and the transaction does not revert, the deletion sticks. A recommitment that was supposed to revive an expired scheduled execution instead returns `ESCROW_ALREADY_EXISTS`, takes the eviction branch, and leaves the system strictly worse off than before the retry: the earlier job is gone from the tracker, no new reservation exists, the scheduled execution has been evicted, and the wallet still holds the original escrow under an identifier that no longer corresponds to any tracked job.

Impact

The recommitment path is effectively unreachable through `addJobAndReserve`, since any attempt to re-add a job under a previously used identifier reverts inside the wallet.

Through `addScheduledJobAndReserve`, the outcome is a silent state inconsistency rather than a revert. A scheduled execution whose identifier collides with a prior expired job is evicted instead of committed, the tracker loses the earlier entry, and the wallet retains a terminal escrow that no longer corresponds to any tracked job. No funds are stolen, but scheduled work can be dropped without a clear signal, and the two contracts hold divergent views of the same identifier.

Recommendations

We recommend reconciling the escrow life cycle in the wallet with the identifier reuse that the tracker permits. One approach is to allow the wallet to open a fresh reservation over an identifier whose previous escrow is `EXPIRED` while continuing to reject reservations against an escrow that is `ACTIVE` or `SETTLED`.

```
function _reserveFrom(/* [...] */) internal {
    _sweepExpired(sponsor, expiredEscrows);
```

```

if (_escrows[escrowId].status != EscrowStatus.NONE) {
    revert EscrowAlreadyExists();
}
EscrowStatus existingStatus = _escrows[escrowId].status;

if (existingStatus == EscrowStatus.ACTIVE || existingStatus == EscrowStatus
.SETTLED) {
    revert EscrowAlreadyExists();
}
// [...]
}

```

The tracker should also finalize the stale escrow when it removes the expired job in `_prepareJob` by calling the `ritualWallet.expire()` function so that accounting stays consistent.

```

function _prepareJob(AddJobInput calldata job) internal {
    if (jobExecutor[job.id] != address(0)) {
        require(expiryBlock[job.id] < block.number, "active job exists");
        _remove(job.id, false);
        ritualWallet.expire(job.id);
    }
    // [...]
}

```

Remediation

This issue has been acknowledged by Ritual, and a fix was implemented in commit [217dfa86](#).

Ritual provided the following comment:

Escrow records are deleted on final settlement and on expiry, so a job ID can be recommitted after expiry ("no record" means free). Escrow deadlines are derived from the same TTL the tracker uses (expiryBlock), so the wallet and tracker clocks stop drifting.

3.5. Unbounded stored blobs make user-initiated cleanup exceed block gas

Target	Scheduler, AgentHeartbeat		
Category	Business Logic	Severity	Medium
Likelihood	High	Impact	Medium

Description

The `_schedule` function validates `numCalls`, `ttl`, `startBlock`, the schedule horizon, the lifespan, and both fee fields but never the length of the data payload it persists. The writer pays once for storing it — but every later slot move that happens to relocate this record as the array's tail repays a cost sized to that same payload, and that cost lands on whoever triggers the move, not on whoever chose the payload's length:

```
// Scheduler._freeSlot
if (slotIndex != lastIndex) {
    _wipeBytes(slots[slotIndex].data);
    uint256 movedCallId = slots[lastIndex].callId;
    slots[slotIndex] = slots[lastIndex];
    callToSlot[movedCallId] = slotIndex;
}
_wipeBytes(slots[lastIndex].data);
slots.pop();
```

Only some callers are exposed, depending on how the transaction is metered. Four of the six paths reaching `_freeSlot` — `_handlePostExecute`, `_evictInsufficientFunds`, `_expireCall`, and the cleanup sweep — run inside `execute`, which is dispatched as a type 0x10 system transaction with `gas_limit = u64::MAX` and `should_skip_gas_limit_check`, so completion, eviction, and expiry cannot run out of gas, however large the payload is. The exposure is confined to `cancel` and `forceCancel`, which are ordinary transactions bound by the block gas limit.

`AgentHeartbeat` has the same shape: `claim` stores an unbounded encrypted `DAConfig` and `heartbeat` that a `manifestCID` checked only for being nonempty, while clearing them is deferred to `_clearPreviousAssignmentForClaim` inside the *next* account's `claim`, and to `adminRemoveAgent`. Its liveness sweep is likewise unaffected, since `heartbeat` maintenance is also dispatched as a type 0x10 transaction.

Impact

An attacker who holds the tail slot with an oversized payload makes `cancel` prohibitively expensive, and past a large enough payload simply impossible, for every other schedule in the

array. The victim is not merely inconvenienced: a schedule that cannot be canceled keeps executing on its original terms, consuming the sponsor's wallet balance on every execution until the schedule completes on its own or the funds run out. The same holds for `forceCancel`, so the owner's recovery path is blocked by the same record. In `AgentHeartbeat`, a departing holder can leave an agent slot that the next claimant cannot afford to claim.

Recommendations

We recommend validating an explicit maximum length at the point of storage for every caller-controlled payload — `Call.data`, `encryptedDAConfig`, and `manifestCID` — chosen so the worst-case cleanup stays well inside a block. As a result, no slot move can then be made unaffordable for the account that triggers it.

Remediation

This issue has been acknowledged by Ritual, and a fix was implemented in commit [45eafc9d](#).

3.6. A registered executor can claim any agentId and make its owner pay for revivals it never approved

Target	AgentHeartbeat		
Category	Business Logic	Severity	Medium
Likelihood	Low	Impact	Medium

Description

The `AgentHeartbeat.claim` function assigns a holder (`agentAddress`) and a bound executor to an `agentId`. It reads the identity owner but uses it only for the emitted event; it never checks that the owner authorized the assignment.

The consequence surfaces on the revival path, where the async job that revives a failed agent is sponsored by the identity owner. The node derives the revival precompile input canonically from on-chain state, placing the identity owner in the sponsor field.

The resulting async commitment calls `AsyncJobTracker.addJobAndReserve`, which reserves from the owner's `RitualWallet` via `reserveFrom(reserve.sponsor, job.caller, ...)`. Because `sponsor` equals `caller`, the wallet skips the allowance branch and debits the owner's available balance directly. The owner therefore backs the cost of a holder/executor binding it never approved on chain.

Impact

A holder whose claim is signed by any registered executor can bind an `agentId` owned by a third party and cause that owner's `RitualWallet` balance to be consumed by revival jobs the owner never authorized.

The path is gated by the signature of a registered executor attested inside a trusted execution environment (TEE), so under a strict honest-executor assumption the likelihood is low: it requires a malicious executor to sign the claim. Within that trust model, however, a single such executor can force any funded agent owner to pay for revivals, because owner consent is never checked and there is no per-agent spending authorization or budget.

Recommendations

Move the revival payer off the owner's general wallet balance and onto a bounded budget the owner provisions for the agent at launch. Additionally, consider accepting an owner signature over (`agentId`, `holderKey`, `executor`, `agentAddress`, `nonce`) alongside the executor signature, so the owner co-authorizes the specific agent/executor pair.

Remediation

Note: As this remediation was implemented in out-of-scope code, we were only able to review it on a best effort basis. See [6.2](#), ↗ for more information.

This issue has been acknowledged by Ritual.

Ritual provided the following comment:

We refactored the revival payment flow so the agent approves AgentHeartbeat to spend from the agent's RitualWallet and pays for its own revival. This directly addresses the primary concern: a revival can no longer charge the identity owner's wallet without that owner's approval. The agent is responsible for claiming its own heartbeat slot and paying the transaction gas. The claim does not debit the identity owner's RitualWallet. Launch is a separate operation. Only the owner of the agentId, or an approved operator, can authorize the precompile to launch the agent. Registered executors operate within attested TEEs and are assumed not to be Byzantine. Under this trust model, a registered executor cannot arbitrarily claim a slot for another agent to drain somebody else's wallet.

3.7. Measured callback gas is not capped by the callback gas limit

Target	Scheduler		
Category	Business Logic	Severity	Medium
Likelihood	Medium	Impact	Medium

Description

The `_execute` function measures the callback by sampling `gasleft()` around the `call` and uses the result for the refund, the emitted event, and the fee distribution without clamping it to the funded limit:

```
uint256 startGas = gasleft();
(bool success,) = callSnapshot.to.call{gas: callSnapshot.gas, value:
    callSnapshot.value}(callbackData);
uint256 gasUsed = startGas - gasleft();
```

`gasUsed` is measured around the `CALL` itself, so it also includes the cold-account access and the value transfer. Neither is charged to the forwarded budget. Unspent callback gas is credited back, so this overhead stays hidden only while the callback leaves enough of its budget unused. Otherwise `gasUsed` exceeds the provided `callSnapshot.gas`. Call that excess `delta`.

Impact

Any callback that uses most of its gas budget makes the measurement exceed the funded limit, and from that point every number derived from it is wrong. Nothing reverts, so none of this is visible on chain.

If the call also sends value and the callback reverts, the refund is computed on the inflated `gasUsed`, so it consumes `effectiveFeePerGas * delta` of the sponsor's value. The refund is short by that much, and once `effectiveFeePerGas * gasUsed` reaches `feesPaid` the guard below fails and the sponsor gets nothing back at all:

```
uint256 actualFailedCallFee = effectiveFeePerGas * gasUsed;

// Refund only the value portion and any excess gas fees. A refund failure is a
// protocol accounting invariant violation for scheduled system transactions,
// so
// it must revert execute() and make the scheduled transaction block-invalid.
if (feesPaid > actualFailedCallFee) {
```

```
uint256 refundAmount = feesPaid - actualFailedCallFee;
RITUAL_WALLET.refundGas{value: refundAmount}(callSnapshot.sponsor,
refundAmount, callId);
}
```

On a successful callback the funds are not lost, but they are split wrongly. The burn address takes `block.basefee * gasUsed` rather than the base-fee share of the gas the sponsor funded, and because the tip is paid out of what is left under the same cap, it absorbs the whole difference:

```
function _distributeGasFees(uint256 priorityFee, uint256 gasUsed,
uint256 gasFeeBudget) private {
uint256 baseFee = block.basefee * gasUsed;
if (baseFee > gasFeeBudget) {
baseFee = gasFeeBudget;
}
payable(BURN_ADDRESS).transfer(baseFee);
uint256 tip = priorityFee * gasUsed;
if (tip > gasFeeBudget - baseFee) {
tip = gasFeeBudget - baseFee;
}
_payBlockProposer(tip);
}
```

The same inflated `gasUsed` is emitted in `CallExecuted/CallExecutionFailed`, and the node reads it back as the scheduled transaction's whole block gas contribution, so the block is accounted above the funded limit too.

Recommendations

Clamp the measurement to the funded limit, as `AsyncDelivery.deliver` already does:

```
uint256 gasUsed = startGas - gasleft();
if (gasUsed > callSnapshot.gas) {
gasUsed = callSnapshot.gas;
}
```

Remediation

This issue has been acknowledged by Ritual, and a fix was implemented in commit [0285cb85](#).

Ritual provided the following comment:

The fix clamps measured Scheduler callback gas to the configured callback limit before calculating refunds, emitted gas usage, and fee distribution. Regression coverage includes reverting callbacks that exhaust the forwarded gas while returning both ordinary and very small revert payloads.

3.8. Neither AsyncDelivery.settle or AsyncDelivery.deliver validate the job's expiryBlock

Target	AsyncDelivery, AsyncJobTracker		
Category	Business Logic	Severity	Low
Likelihood	Low	Impact	Low

Description

Neither settlement entry point in AsyncDelivery reads the tracker's own expiryBlock[jobId] before acting on a job.

```
function settle(/* [...] */)
    external onlySettlementSystem returns (bool success) {
        // [...]
        if (isLongRunning) {
            uint64 phase2Deadline = ASYNC_JOB_TRACKER.computePhase2Deadline(jobId,
                precompileInput);
            RITUAL_WALLET.settle(jobId, payments, 0, phase2Deadline, false);
            ASYNC_JOB_TRACKER.markPhase1Settled(jobId, precompileInput);
        } else {
            RITUAL_WALLET.settle(jobId, payments, 0, 0, true);
            ASYNC_JOB_TRACKER.removeJob(jobId, precompileInput, true);
        }
        // [...]
    }
}
```

The getSettlementContext, computePhase2Deadline, removeJob, and markPhase1Settled functions check job existence, the input hash, and (for the two-phase functions) that phase 1 has not already been settled, but none of them checks whether the job's own TTL has elapsed. The deliver function has the same gap: _requireDeliverable and markDelivered check only the input hash and phase-1 settlement status.

The only expiry check on either path belongs to the wallet escrow, inside RitualWallet._settle, which both settle and deliver call before touching the tracker.

```
if (block.number > escrow.expiryBlock) {
    revert InvalidEscrowExpiry();
}
// [...]
if (newExpiry != 0) {
```

```
if (newExpiry <= block.number) revert InvalidEscrowExpiry();
escrow.expiryBlock = newExpiry;
}
```

Critically, this check runs against the escrow's *current* expiryBlock, before newExpiry overwrites it. Whether that check happens to also validate the job's own deadline depends entirely on what the escrow's current expiry is at the time:

- For settle's regular (non-long-running) branch and for deliver, the escrow's current expiry does equal the job's own deadline: reserveExpiryBlock is set to commitBlock + ttl off chain (no phase-2 offset for a regular job), matching the tracker's expiryBlock[jobId], and by the time deliver runs, markPhase1Settled has already moved both the escrow's expiry and the tracker's expiryBlock[jobId] to the identical phase-2 deadline. In both cases, the escrow check incidentally validates the job's deadline.
- For settle's long-running branch, it does not. At phase-1 settlement, the escrow's expiryBlock is still the original reserveExpiryBlock set at commitment time, which off chain is computed as commitBlock + ttl + phase2offset, up to MAX_PHASE2_DEADLINE_OFFSET (70,000 blocks) beyond the job's actual phase-1 deadline (capped at MAX_TTL_BLOCKS, 500 blocks). The escrow check therefore accepts phase-1 settlement for the entire 70,000-block window after the job's own TTL has elapsed.

Impact

On chain, phase-1 settlement of a long-running job can proceed after the job's own TTL has passed, for as long as the job's phase-2 offset allows, up to MAX_PHASE2_DEADLINE_OFFSET, because the escrow's expiry describes that longer window by design and no function on the settle path compares the current block against the tracker's expiryBlock. The window closes only incidentally, when lazy cleanup happens to remove the expired job first, at which point settlement reverts. The only component that rejects such a settlement on expiry grounds is the off-chain node, which reads expiryBlock from tracker storage and both skips building and rejects verifying a settlement submitted after it.

Recommendations

Both settle and deliver call getSettlementContext first, before doing anything else, so adding the check there covers both entry points, including the long-running branch that currently has no incidental protection, with a single change.

```
function getSettlementContext(bytes32 id)
    external
    view
    returns (address executorPaymentAddress, address commitmentValidator)
```

```
{  
    require(jobExecutor[id] != address(0), "not found");  
    require(block.number <= expiryBlock[id], "job expired");  
    Job storage job = jobs[id];  
    // [...]  
}
```

The same check can equally be added directly to `removeJob`, `markPhase1Settled`, and `markDelivered` functions.

Remediation

This issue has been acknowledged by Ritual, and a fix was implemented in commit [217dfa86](#).

3.9. Revoked executors retain heartbeat authority

Target	AgentHeartbeat		
Category	Business Logic	Severity	Low
Likelihood	Low	Impact	Low

Description

The `claim` function binds executor to the slot only after checking that it is currently valid.

```
require(_isRegisteredExecutor(executor), "executor not registered");
// [...]
slot.boundExecutor = executor;
```

```
function _isRegisteredExecutor(address addr) internal view returns (bool) {
    try TEE_REGISTRY.getService(addr)
    returns (ITEEServiceRegistry.TEEServiceContext memory ctx) {
        return ctx.isValid;
    } catch {
        return false;
    }
}
```

The heartbeat function, by contrast, only checks that the submitted signature recovers to the address already stored in `slot.boundExecutor`. It never calls `_isRegisteredExecutor` or otherwise consults `TEE_REGISTRY` again.

```
function heartbeat(uint256 agentId, string calldata manifestCID,
    bytes calldata executorSignature) external {
    // [...]
    address signer
    = digest.toEthSignedMessageHash().recover(executorSignature);
    require(signer == slot.boundExecutor, "executor mismatch");

    slotNonce[agentId] = nonce + 1;
    slot.lastHeartbeat = uint64(block.number);
    // [...]
}
```

The `ctx.isValid` field is a live property of `TEE_REGISTRY`, not a fact fixed at claim time; an executor's attestation can later expire, be revoked, or otherwise be marked invalid. Because `heartbeat` never rereads it, a `boundExecutor` that was valid at claim time keeps full signing authority over the slot for as long as it holds the key, regardless of what the registry says about it afterward.

The revival path has the same gap: it writes `slot.boundExecutor` without consulting the registry at all.

```
function _attemptRevival(uint256 agentId, SlotAssignment storage slot,
    bytes calldata precompileInput) internal {
    revivalAttempts[agentId] += 1;

    address executor = _extractExecutorFromInput(precompileInput);
    bytes memory precompilePayload = precompileInput;
    _applyRevivalPrecompileCall(agentId, slot, executor, precompilePayload);
}
```

```
function _applyRevivalPrecompileCall(/* [...] */) internal {
    address previousExecutor = slot.boundExecutor;
    slot.boundExecutor = executor;
    // [...]
}
```

Impact

The `heartbeat` only checks that the signature matches `slot.boundExecutor` and never checks whether the registry still says that executor is valid. So once an executor is revoked, its signature still matches, and `heartbeat` still accepts it as proof of a valid TEE, even though it no longer is one. Since every accepted heartbeat resets `slot.lastHeartbeat`, the slot never times out and the agent never enters the revival path.

Additionally, revival can bind an executor that the registry would reject, and `heartbeat` will then accept that executor's signatures indefinitely.

Recommendations

We recommend rechecking `_isRegisteredExecutor(slot.boundExecutor)` inside `heartbeat`, so a heartbeat signed by an executor the registry no longer considers valid is rejected and the slot is left to expire into the existing revival path instead of being kept alive indefinitely.

```
address signer = digest.toEthSignedMessageHash().recover(executorSignature);
require(signer == slot.boundExecutor, "executor mismatch");
```

```
require(!_isRegisteredExecutor(slot.boundExecutor), "executor no longer valid"
);
```

We also recommend validating the executor on the revival path before binding it:

```
address executor = _extractExecutorFromInput(precompileInput);
require(!_isRegisteredExecutor(executor), "executor not registered");
bytes memory precompilePayload = precompileInput;
_applyRevivalPrecompileCall(agentId, slot, executor, precompilePayload);
```

Remediation

This issue has been acknowledged by Ritual, and fixes were implemented in the following commits:

- [48b285b0](#) ↗
- [579ab813](#) ↗

3.10. Final TTL skip leaves schedule stuck

Target	Scheduler		
Category	Coding Mistakes	Severity	Low
Likelihood	Low	Impact	Low

Description

Per-index TTL expiry marks the index consumed and returns immediately, without invoking the common postexecution transition that decides whether the call is finished:

```
// Scheduler.execute
uint256 executionExpiryBlock = expectedBlock + uint256(callSnapshot.ttl);
if (block.number > executionExpiryBlock) {
    _markExecuted(callId, executionIndex);
    emit CallSkippedTTLExpired(...);
    return;
}
```

This matters because the scheduler explicitly permits out-of-order processing and overlapping windows, so later indexes can already be consumed when an earlier index is finally skipped. If that expired earlier index is the final unset bit, `_allExecutionIndicesConsumed` becomes true while the call stays SCHEDULED, because the completion check inside `_handlePostExecution` is never reached. Every index then rejects as `AlreadyExecuted`, so the call can no longer complete normally or emit `CallCompleted`.

Impact

A recurring schedule whose executions have all either run or expired keeps reporting SCHEDULED and holds its slot, instead of reaching its documented terminal state.

Recommendations

When a TTL skip consumes the last outstanding index, consider applying the same cleanup the execution path performs: clear the bitmap, free the slot, and delete the auxiliary storage, so the call ends instead of staying SCHEDULED.

Remediation

This issue has been acknowledged by Ritual, and fixes were implemented in the following commits:

- [ffa19f6e ↗](#)
- [1c39a85b ↗](#)

3.11. Insufficient test coverage

Target	Project-wide		
Category	Protocol Risks	Severity	Low
Likelihood	N/A	Impact	Low

Description

The contracts have a substantive test suite that passes in full and whose breadth is good: every core contract has a dedicated suite, feature coverage is broad, and testing goes beyond the happy path in places, with property tests over the bounded-cleanup logic and adversarial cases such as reentrancy during scheduler execution and signature replay on the heartbeat slot.

However, the test suite is not deep. Most lines of code are covered, but most branches are not. In practice, this means a function gets tested when it succeeds far more often than when it rejects or fails, so those cases are largely unverified. The problem is not which features are tested but how many of each function's possible outcomes are.

Specifically, the areas requiring additional coverage fall into the following categories.

- **Entry points that no test calls.** In `RitualWallet`, these are `deposit`, `increaseAllowance`, `decreaseAllowance`, and `expireMany`. In `Scheduler`, these are `cleanupInvalidScheduledAsync` and three of the four `schedule` overloads. In `AgentHeartbeat`, this is the pause mechanism. Notably, `_authorizeUpgrade` is uncovered on all contracts in scope, so no test demonstrates that a non-owner cannot upgrade an implementation.
- **Guard rejection paths.** Across the reviewed contracts, roughly half of all `require` and `revert` sites are never asserted by any test, so a large share of the input validation, authorization, and state checks in the codebase are unverified.
- **Economic failure paths.** Since `RitualWallet._tryReserveFrom` is entirely uncovered, the soft-failure reservation path it implements is not tested. The `addScheduledJobAndReserve` function does have tests that drive a reservation failure and assert the tracker evicts the schedule without storing the job. However, those tests reach the mock rather than the production wallet code, so the tracker's behavior is verified while the wallet logic is not. Branches for `_deductScheduledExecutionFees`, `_settle`, and `_reserveFrom` also are not fully covered, and on the tracker side, neither are `addScheduledJobAndReserve` and `_prepareJob`.

The pattern is that the contracts are tested for what happens when a sponsor can pay — and much less for what happens when a sponsor cannot.

- **Escrow life cycle and the mock boundary.** `RitualWallet` is substituted by `MockRitualWallet` in the `Scheduler`, `AsyncJobTracker`, and `AsyncDelivery` tests. Even where the mock's `_tryReserveFrom` reimplements the production control flow closely,

a change to the production version would not fail a single test, because the tests would still be executing the mock's unchanged copy. The same substitution leaves the `reserveFrom` → `settle` → `expire` sequence without a single end-to-end test against the production contract.

Impact

Insufficient coverage of failure paths increases the likelihood of the following:

- Defects in payment and escrow accounting reach production, since those are precisely the branches with the least coverage and the highest value at risk.
- Refactors silently break soft-failure semantics. A change that converts a soft failure into a revert, or the reverse, would not be caught in functions such as `tryReserveFrom` and `deductScheduledExecutionFeesFrom`.
- An upgrade authorization regression goes unnoticed, as no test asserts that a non-owner is rejected on any of the five proxies.

Recommendations

We recommend expanding the suite in the following order, which is roughly the order of value at risk.

1. Extend the wallet's failure-path tests.
2. Add integration tests against the real wallet.
3. Cover the scheduled-async handoff end to end.
4. Test the heartbeat guards' rejection paths.
5. Test authorization and admin rejection paths.
6. Add invariant or property tests over the accounting, such as balances reconciling with the contract's RITUAL balance and no settled or expired job retaining escrow.

Remediation

This issue has been acknowledged by Ritual, and a fix was implemented in commit [b2c25074](#).

3.12. A zero delivery callback gas limit is admitted end to end, guaranteeing the Phase 2 callback fails

Target	AsyncJobTracker, AsyncDelivery		
Category	Business Logic	Severity	Low
Likelihood	Low	Impact	Low

Description

`_prepareJob` is the admission point every async commitment passes through, and for two-phase precompiles it is the only place the contract inspects the delivery configuration the caller supplied. It calls `_validateDeliveryFees` to validate input data, which dispatches to `_requireFeesU128` at the family-specific field offset. That function reads the delivery gas limit and bounds it from above only:

```
function _requireFeesU128(bytes calldata input, uint256 feeOffset)
    private pure {
    // [...]
    assembly {
        gasLimit := calldataload(add(input.offset, gasLimitOffset))
        maxFee := calldataload(add(input.offset, feeOffset))
        maxPriorityFee := calldataload(add(input.offset,
            maxPriorityFeeOffset))
    }

    require(gasLimit <= type(uint64).max, "delivery gasLimit exceeds u64");
    // [...]
}
```

`AsyncDelivery.deliver` forwards the value directly to the callback, specifying the gas as a `callbackGasLimit` equal to the delivery gas limit:

```
uint256 gasBefore = gasleft();
(success,) =
    target.call{gas: callbackGasLimit, value:
        callbackValue}(abi.encodeWithSelector(selector, jobId, result));
uint256 gasConsumed = gasBefore - gasleft();

if (gasConsumed > callbackGasLimit) {
    gasConsumed = callbackGasLimit;
}
```

With `callbackGasLimit == 0` the call fails, but `deliver` does not treat that as an error condition. It proceeds to `markDelivered(jobId, precompileInput, false)`, which emits `ResultDelivered(jobId, target, false)` and calls `_remove`, deleting the job, its input hashes, its expiry-bucket entry and its executor index entry.

Impact

A sponsor who commits a two-phase job with a zero delivery gas limit pays the Phase 1 executor, commitment and inclusion fees for work the TEE genuinely performed, and the result can never be delivered on chain.

Recommendations

Add the missing floor beside the existing ceiling, so the field is bounded on both sides in the same place:

```
require(gasLimit >= MIN_DELIVERY_CALLBACK_GAS, "delivery gasLimit below  
floor");  
require(gasLimit <= type(uint64).max, "delivery gasLimit exceeds u64");  
require(maxFee <= type(uint128).max, "maxFeePerGas exceeds u128");  
require(maxPriorityFee <= type(uint128).max, "maxPriorityFeePerGas exceeds  
u128");
```

Remediation

This issue has been acknowledged by Ritual, and a fix was implemented in commit [7fab426e](#).

3.13. Dead balance restoration before revert in the scheduled fee-deduction path

Target	RitualWallet		
Category	Code Maturity	Severity	Informational
Likelihood	N/A	Impact	Informational

Description

In `_deductScheduledExecutionFees`, `RitualWallet` deducts the fee from the sponsor's internal balance, forwards the corresponding `RITUAL` to the Scheduler, and manually recredits the balance if the transfer fails, immediately before reverting.

```
// Deduct from sponsor's internal balance
_balances[sponsor] -= actualFee;

// Transfer actual ETH to the Scheduler contract
(bool success,) = payable(msg.sender).call{value: actualFee}("");
if (!success) {
    // Restore balance if transfer fails
    _balances[sponsor] += actualFee;
    revert TransferFailed();
}
```

The recredit is dead code. A `revert` already undoes every state change made in the current call, including the deduction above, so the manual restoration has no effect on any reachable state.

Impact

There is no security impact. The redundant write is discarded by the `revert`, so there is no accounting error. What the code does carry is a false statement about its own failure semantics: the restoration implies that the deduction survives the branch and must be undone by hand, which is not the case. Should that branch ever be changed to something other than a `revert` — returning a failure flag, or recording the failure and continuing — the write would become live code whose correctness nobody had reason to check.

Recommendations

We recommend deleting the redundant write and its comment so that the failure path consists of the `revert` alone.

```
// Transfer actual ETH to the Scheduler contract
(bool success,) = payable(msg.sender).call{value: actualFee}("");
if (!success) {
    // Restore balance if transfer fails
    _balances[sponsor] += actualFee;
    revert TransferFailed();
}
```

Remediation

This issue has been acknowledged by Ritual, and a fix was implemented in commit [8ee4c413](#).

3.14. Redundant isValid check in _validateExecutor and _scheduledExecutorInvalidReason is unreachable

Target	AsyncJobTracker		
Category	Code Maturity	Severity	Informational
Likelihood	N/A	Impact	Informational

Description

Both `_validateExecutor` and `_scheduledExecutorInvalidReason` call `TEE_SERVICE_REGISTRY.getService(executor)` inside a try-catch and then separately check the returned context's `isValid` field.

```
ITEEServiceRegistry.TEEServiceContext memory context;
try TEE_SERVICE_REGISTRY.getService(executor)
    returns (ITEEServiceRegistry.TEEServiceContext memory result) {
    context = result;
} catch {
    return SCHEDULED_INVALID_EXECUTOR_REASON;
}

if (!context.isValid) {
    return SCHEDULED_INVALID_EXECUTOR_REASON;
}
```

The `_validateExecutor` function follows the identical pattern, reverting with `InvalidExecutor` in both the catch block and the `isValid` check. The one-argument `getService` overload, however, is documented as checking validity by default and reverting whenever the attestation is no longer valid, guaranteeing that any value it does return has a valid attestation.

```
/// @notice Get service context for an address (checks validity by default)
/// @dev Reverts with AttestationInvalid if the service's attestation is no
///      longer valid
/// @return context The service context (guaranteed valid attestation)
function getService(address addr)
    external view returns (TEEServiceContext memory context);
```

The catch block therefore already handles every case in which the executor is unregistered or its attestation is invalid, mapping both to the same outcome as the `if (!context.isValid)` check that follows.

By the time execution reaches that check, `context` can only be the result of a successful call, which the registry guarantees has `isValid == true`. The check can never evaluate to true in either function; it is dead code.

Impact

There is no functional impact, since the redundant branch can never execute and the `catch` block already produces the correct rejection. The cost is misleading code; both functions read as though a successfully returned context can still be invalid, which misrepresents the one-argument `getService`'s documented contract and could lead a future maintainer to believe the explicit check, rather than the try-catch, is what is doing the validation.

Recommendations

We recommend removing the dead check in both functions, since the try-catch already covers it.

```
ITEEServiceRegistry.TEEServiceContext memory context;
try TEE_SERVICE_REGISTRY.getService(executor)
returns (ITEEServiceRegistry.TEEServiceContext memory result) {
    context = result;
} catch {
    revert InvalidExecutor(executor);
}

if (!context.isValid) {
    revert InvalidExecutor(executor);
}

if (!_capabilityMatchesPrecompile(precompile, context.node.capability)) {
```

Remediation

This issue has been acknowledged by Ritual, and a fix was implemented in commit [a47c0fa5](#).

3.15. Scheduled executions and Phase 2 deliveries collect gas payments but never disburse them

Target	Scheduler, AsyncDelivery, RitualWallet		
Category	Business Logic	Severity	Informational
Likelihood	N/A	Impact	Informational

Description

Because system transactions bypass EVM gas charging, the gas economics of scheduled calls and Phase 2 deliveries are reimplemented in Solidity. The collection half of that reimplementation exists; the disbursement half does not.

The Scheduler.execute function computes an EIP-1559 effective price and pulls the full budget from the wallet as real ETH:

```
uint256 effectiveFeePerGas = baseFee + priorityFee;
// Deduct execution fees from sponsor's account in system wallet.
uint256 feesPaid = RITUAL_WALLET.deductScheduledExecutionFeesFrom(
    callSnapshot.sponsor,
    callSnapshot.to,
    effectiveFeePerGas,
    callSnapshot.gas,
    callSnapshot.value,
    executionFee,
    callId
);
```

The deductScheduledExecutionFeesFrom function debits the sponsor's tracked balance and transfers the corresponding native ETH to the Scheduler:

```
// RitualWallet.sol
actualFee = (effectiveFeePerGas * gasLimit) + value + fixedFee;
// ...
_balances[sponsor] -= actualFee;

// Transfer actual ETH to the Scheduler contract
(bool success,) = payable(msg.sender).call{value: actualFee}("");
```

After the callback runs, only the unused portion returns:

```
// Scheduler.sol
uint256 actualCallFee = (effectiveFeePerGas * gasUsed) + callSnapshot.value
    + executionFee;

if (feesPaid > actualCallFee) {
    uint256 refundAmount = feesPaid - actualCallFee;
    RITUAL_WALLET.refundGas{value: refundAmount}(callSnapshot.sponsor,
        refundAmount, callId);
}
```

What stays behind is `actualCallFee` minus the value already forwarded to the target: `effectiveFeePerGas * gasUsed + executionFee`. Nothing in `Scheduler.sol` moves it.

The `AsyncDelivery.deliver` function has the same shape. It credits the callback budget to itself in escrow settlement, then withdraws it as native ETH

```
payments[1] = IRitualWallet.Payment({to: payable(address(this)), amount:
    callbackBudget});
RITUAL_WALLET.settle(jobId, payments, 0, 0, true);
if (callbackBudget > 0) {
    // Escrow settlement credits wallet balances; delivery pulls its own
    budget to fund the callback.
    RITUAL_WALLET.withdraw(callbackBudget);
}
```

and refunds only unused gas plus, on revert, the returned value. The product `gasConsumed * effectiveFeePerGas` remains on the contract.

Note the asymmetry with Phase 1. The `AsyncDelivery.settle` function does pay the validator that included the transaction, through an escrow payment to `block.coinbase` that credits a wallet balance the coinbase can later withdraw. The scheduled path has no equivalent, so the pattern for paying a validator from escrow exists in the codebase and is simply not applied here.

Impact

Since the gas component was always meant to leave circulation, nobody was ever due to receive it, and the sponsor pays the same amount whether it is burned or stranded. These are not lost user funds.

The team confirmed that the intended behavior is to transfer the extra amount to a burn address, but it is not clear whether the same is meant to apply to `SCHEDULED_EXECUTION_FEE`, since that is a flat protocol charge with no EIP-1559 counterpart.

Recommendations

The extra funds should be transferred to a burn address, including SCHEDULED_EXECUTION_FEE if that is intended as well.

Remediation

This issue has been acknowledged by Ritual, and fixes were implemented in the following commits:

- [78a13db9](#) ↗
- [9bdfdbc0](#) ↗
- [840e481d](#) ↗
- [9c20d4ec](#) ↗

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. The return value of `_remove` is silently ignored

The internal function `_remove` in `AsyncJobTracker` returns a boolean indicating whether the job existed and was removed. When the job identifier has no registered executor, the function returns `false` without performing any state change.

```
function _remove(bytes32 id, bool done) internal returns (bool) {
    address exec = jobExecutor[id];
    if (exec == address(0)) {
        return false;
    }
    // [...]
}
```

None of the six call sites inspect this return value. The `removeJob` function forwards the boolean to its caller, but `AsyncDelivery` discards it in `settle`. The remaining callers, `_prepareJob`, `markDelivered`, `_cleanup`, `cleanupBatch`, and `cleanupExpired`, ignore it entirely.

Under the contract's present invariants, the `false` branch is unreachable at every call site, so ignoring it has no effect. The risk is what happens if a future change ever leaves a stale identifier in an expiry bucket without a matching executor. The cleanup loops in `_cleanup` and `cleanupBatch` pop entries from a bucket only inside `_remove`; a `false` return would leave the entry in place while the loop counter still advances, so the loop would retry the same tail element until it exits with the bucket still nonempty. Because `lastCleanup` only advances past fully drained buckets, every subsequent cleanup would restart at the same poisoned bucket and remove nothing, a silent livelock rather than a revert.

There is no security impact in the code as written. The pattern does, however, mean that a hypothetical future invariant violation would degrade the cleanup mechanism silently rather than being reported anywhere, delaying detection of the underlying bug. As one possible solution, we recommend that `_cleanup` and `cleanupBatch` act on the return value rather than discard it, dropping the entry from the bucket when `_remove` reports `false` so that the sweep always makes progress, and emitting an event naming the stale identifier so it can be investigated.

4.2. EIP-7702 delegated accounts bypass the contract-only check on scheduling

Scheduler requires the scheduling account to be a contract, so that the callback it later receives reaches code implementing `IScheduledCallback`. The restriction is a code-length test, applied once in `_schedule`.

```
function _isEOA(address target) internal view returns (bool) {  
    return target.code.length == 0;  
}
```

Under EIP-7702, an externally owned account that has signed an authorization carries a delegation indicator as its code, so `code.length` is nonzero while the delegation is set and the account passes the check. It then becomes the call's `to` address, which is both the callback target and the only address permitted to cancel it.

Nothing requires the account to stay delegated, and the check is never repeated: `execute` dispatches to the recorded target without reexamining its code. An account that revokes its authorization after scheduling is a plain EOA by the time the call is made, and a call to an account with no code succeeds unconditionally, so the calldata is discarded, no logic runs, and the attached value is transferred. Because the EVM reports success, `execute` takes the successful branch, where the callback value is accounted for as consumed rather than refunded. A delegation left in place, pointing at code with no matching function, would instead cause the call to revert and the value to be returned.

There is no security impact in the code as written, since the value is delivered to `call.to`, the account that created the schedule. The outcome is that the account pays the execution fee and the gas consumed in order to send itself value at a delay, and a third-party sponsor must already have approved that address for a maximum-cost execution. The check does, however, illustrate an inconsistency in validation: the contract-only restriction is enforced when a call is scheduled but not when it is executed, and because an account's delegation status can change in any block, the check in `_schedule` cannot establish what the target will be by the time the callback is made.

5. System Design

This provides a description of the high-level components of the system and how they interact, including details like a function's externally controllable inputs and how an attacker could leverage each input to cause harm or which invariants or constraints of the system are critical and must always be upheld.

Not all components in the audit scope may have been modeled. The absence of a component in this section does not necessarily suggest that it is safe.

Four of the five in-scope contracts form a single settlement pipeline. RitualWallet custodies all native RITUAL and is the only contract that moves funds; Scheduler and AsyncJobTracker are the admission points that decide when a recurring callback and when an async precompile job may draw on a sponsor; and AsyncDelivery is the sole contract permitted to pay out an async job's escrow and run its Phase 2 callback. AgentHeartbeat sits outside it, holding no funds and calling none of the other four; it tracks persistent-agent liveness, and a revival it initiates reaches the pipeline only indirectly as a fresh async job the node then funds and settles like any other.

Two assumptions recur below. The reserved system addresses — 0x...fa7e (scheduled execution and heartbeat maintenance), 0x...Fa8e (async commitment), and 0x...fA9E (async settlement) — are fully trusted, and every argument they pass is accepted as canonical protocol context. Their node-side construction and validation was out of scope, so wherever a contract documents that a check "is performed in reth", this section records that assumption rather than verifying it. Each contract is also a UUPS proxy whose upgrade authority is a single owner.

5.1. Component: RitualWallet

Description

RitualWallet is the custody and accounting layer for every payment the protocol makes. All native RITUAL spent on scheduled executions and async precompile jobs passes through it, and it is the only in-scope contract that holds user funds.

Each account has two balances: an available one it may withdraw at any time, and a reserved one backed by per-job escrow records keyed by job identifier. Reserving moves value between them and opens an escrow recording the sponsor, the amount, an expiry block, and a status. Funding is permissionless — anyone may deposit for any account, or send RITUAL directly — while withdrawal is strictly self-service. Third-party sponsorship uses an ERC-20-style allowance that the sponsor grants and the wallet decrements when the acting caller differs from the sponsor. When caller and sponsor are the same address, the allowance check is skipped entirely.

Four operations are reachable only from Scheduler, AsyncDelivery, and AsyncJobTracker, fixed as immutables at construction:

1. **Reserve** (reserveFrom, tryReserveFrom) opens an escrow against the sponsor's available balance. The try variant returns a typed failure rather than reverting, so one unfundable scheduled job cannot revert an entire system transaction.
2. **Settle** debits an escrow, credits each recipient, and then either extends the expiry — the Phase 1 to Phase 2 handoff — or closes the escrow and returns the remainder to the

sponsor. Recipients are credited internally rather than paid by transfer.

3. **Deduct** (Scheduler only) is the just-in-time debit for one scheduled execution, transferring real RITUAL directly to the Scheduler so it can fund the callback's gas and value. It returns zero rather than reverting when the sponsor is short. AsyncDelivery funds its own Phase 2 callback the same way in substance, though through a different mechanism: `deliver` settles the callback budget to its own internal balance and then calls the ordinary self-service `withdraw` to pull it out before forwarding it to the callback target.
4. **Refund** returns unused execution value, requiring the attached value to match the declared amount.

Escrow expiry is permissionless: anyone may expire a lapsed escrow back to its sponsor, and withdrawing or reserving accounts may sweep their own expired escrows in the same transaction. Reentrancy guards cover withdrawal, settlement, and the scheduled debit.

Invariants

- The contract's RITUAL balance is always at least the sum of every account's available and reserved balance. Value leaves only through a withdrawal or the scheduled debit, each of which decrements the accounting by the amount it moves.
- For each account, its reserved balance equals the total remaining reserve across its active escrows. Reservation, settlement, closure, and expiry must all maintain this pairing.
- For each escrow, the amount originally reserved equals its remaining reserve, plus the total paid out of it, plus the total returned to its sponsor.
- An escrow's status only advances, from NONE to ACTIVE to either SETTLED or EXPIRED, so an identifier is single-use, and a terminal escrow always has a remaining reserve of zero.
- An account's balance decreases only through its own withdrawal, or a spend it authorized as the sponsor or through an allowance, which is then reduced by exactly the amount spent.

Test coverage

Cases covered

- The escrow life cycle — reservation moving available balance into reserved, settlement crediting and refunding out of reserve, and an expired escrow swept during withdrawal
- Allowance handling for a delegated reserve, including an allowance set to the maximum left unchanged
- Guard rejections — unauthorized reserve callers, identifier reuse while an escrow is active, and a refund value mismatch
- Adversarial settlement — a recipient that rejects RITUAL credited without reverting

Cases not covered

- `deposit`, `increaseAllowance`, `decreaseAllowance`, and `expireMany`, which no test calls
- `tryReserveFrom` against the production contract — only the mock's copy is exercised, leaving its typed failures and allowance restoration unverified
- The `reserveFrom` to `settle` to `expire` sequence end to end against the production contract
- Settlement guards — exceeding the reserve, an escrow already settled or expired, and nonfuture expiry values
- Reserving an identifier whose escrow already settled or expired (Finding [3.4](#), [7](#))
- The scheduled debit returning zero rather than reverting on a short balance or allowance
- Withdrawing more than the available balance and a non-owner upgrade
- An accounting invariant over randomized operation sequences

Attack surface

The permissionless entry points are self-limiting; deposits, allowance updates, and the caller-supplied escrow arrays only credit or bind the caller's own funds, and expiry is a no-op until an escrow actually lapses. Expiry can still race a settlement into a revert, which matters because `AsyncDelivery` does not check the job's own expiry (Finding [3.8](#), [7](#)), leaving the wallet's status check as the only one standing.

The principal surface is the `sponsor` and `caller` pair. The wallet authenticates neither, enforcing only the balance check plus, when they differ, the allowance. Any defect in how `Scheduler`, `AsyncJobTracker`, or `reth` binds the two becomes a direct debit against that sponsor — the shape of Finding [3.6](#), [7](#), where both resolve to the agent's identity owner and the allowance branch is skipped entirely.

Elsewhere, the design limits exposure; settlement credits recipients internally, so a hostile recipient can neither reenter nor revert a system transaction, and the paths that push value out are constrained to the immutable `Scheduler` and `AsyncDelivery`. `AsyncDelivery`'s share of that surface is the wider of the two, since the callback target and calldata it forwards value to are decoded from off-chain precompile input rather than fixed at construction. The residual risk is the owner, who can upgrade the implementation and reach every deposit.

5.2. Component: Scheduler

Description

`Scheduler` lets an on-chain contract register a callback that the node will invoke at a future block, optionally repeating on a fixed cadence.

Only a contract may schedule, and it becomes the call's target — so the callback always returns to the contract that requested it, and only that contract may cancel it. The sponsor is either that

contract or a third party who granted it a wallet allowance. Creation bounds the schedule on every axis that later arithmetic depends on: cadence, a start block in the future, at most 256 executions, a TTL, a total lifespan, and the EIP-1559 fee fields. Solvency, however, is checked only for a single maximum-cost execution — schedules are funded just in time rather than escrowed for their lifetime.

The `execute` function is the system-only path. It validates that the execution is due, unconsumed, within its TTL window, and permitted by the call's optional predicate, then pulls the maximum cost from `RitualWallet`, marks the index consumed, invokes the target with the configured gas and value, and refunds the difference between the maximum and the actual cost. `Reth` remains the authority on which scheduled transactions are canonical; the contract mirrors the same checks as a backstop.

Calls are held in a compacting array, so a target that reenters and cancels another call can move the record currently being executed — which is why the execution state is snapshotted and reacquired around the callback.

Three further entry points bridge to the async pipeline, reachable only by the configured `AsyncJobTracker` or the async commitment account. They consume or evict a scheduled execution whose callback reached an async precompile, each bound to an identifier derived from the schedule's origin hash. Owner powers are `forceCancel` — a recovery path that skips the caller check `cancel` enforces and accepts live states `cancel` rejects — the per-execution fee setter, the tracker address, and the UUPS upgrade.

Invariants

- A call executes at most once per execution index, and never more times than its execution count. The index is marked before the target is called, so a reentrant target cannot replay it.
- Execution index i never runs before `block startBlock + i × frequency` nor after that block plus the call's TTL.
- A call reaches the completed state exactly when all of its execution indexes have been consumed.
- A canceled, completed, expired, or evicted call never executes again, and no call is left midexecution between transactions.
- A call's target, sponsor, cadence, and execution count are fixed at creation and never change.
- The cost charged to a sponsor for one execution never exceeds `maxFeePerGas × gas + value + fixed fee`. After the refund, the net charge is `effectiveFeePerGas × gasUsed + value + fixed fee`, and the `value` term drops out when the callback fails — the sponsor pays for the gas actually consumed rather than the gas budgeted, and gets the value back if the call did not go through.
- Every live call becomes removable within a bounded number of blocks of its creation.

Test coverage

Cases covered

- The execution life cycle, including multiexecution schedules, out-of-order execution across overlapping windows, and every terminal path releasing its storage
- Payment behavior — fee accounting and its boundaries, base-fee skips versus insolvency eviction, and refund failure reverting the execution
- Predicates returning false and predicates reverting
- Reentrancy — a target that compacts the slot array midexecution, and fragmentation under stress
- Every creation bound, at and past its limit
- `forceCancel` by the owner and its rejection for a non-owner, including recovery of a stale executing call

Cases not covered

- Three of the four schedule overloads, and `cleanupInvalidScheduledAsync`
- `execute` from a nonsystem account, and a non-owner upgrade
- Replaying a consumed index or executing one beyond the schedule's execution count
- The terminal state after the *final* index is skipped for TTL expiry (Finding [3.10](#), ↗)
- A far-future start block against a populated slot array (Finding [3.3](#), ↗)
- A predicate destroyed since scheduling, or returning fewer than 32 bytes
- A sponsor revoking an allowance midschedule

Attack surface

Every field of a schedule is attacker controlled, and the interesting one is the start block: the lifespan check measures only the window from it, so a schedule beginning decades from now passes validation while occupying a slot indefinitely, and the cleanup sweep — capping expirations found rather than slots examined — then walks the whole inflated array on every execution. This is Finding [3.3](#), ↗, severe because it degrades a chain-level liveness path rather than one user's funds.

The callback target and predicate are arbitrary attacker code. A target may revert, exhaust its gas, or reenter to compact the slot array, the hazard the snapshot-and-reacquire pattern neutralizes; a predicate cannot mutate state, but a revert or short return makes `execute` revert rather than skip.

Three asymmetries remain: one execution's allowance admits up to 256 executions; a single failed async reserve evicts the whole schedule, unlike the transient skip granted to a base-fee spike; and the per-execution fee applies retroactively.

5.3. Component: AsyncJobTracker

Description

AsyncJobTracker is the registry of in-flight async precompile jobs and the admission point where a commitment becomes a funded obligation. When a transaction reaches one of the async precompiles, the node holds the original back and commits the job by calling this contract, recording the job and reserving the sponsor's worst-case escrow in one atomic call.

Which precompile was reached decides everything downstream. Single-phase families settle once, in-line; two-phase families settle a first time to return a handle and later deliver the real result through a callback, so their escrow must stay funded past the initial job deadline.

Two admission paths exist. The user-origin path reverts if anything fails. The scheduled-origin path instead requires the job identifier to match the scheduler's derived identifier, reserves through the nonreverting variant, and reports the outcome back to the Scheduler, which confirms the execution index is still pending and either emits an in-flight event or evicts it with a typed reason — so that an unfundable or badly targeted scheduled job cannot revert the commitment transaction. The index itself is only consumed later, at settlement.

The tracker stores only hashes of the precompile input and the origin transaction; the full bytes are resupplied at settlement and checked against them. Jobs are capped in count, input size, TTL, and Phase 2 offset, and expired ones are removed in gas-bounded batches, opportunistically at admission and permissionlessly otherwise.

Life-cycle transitions are restricted to AsyncDelivery, and each reverifies the input hash. Single-phase jobs are removed outright; two-phase jobs are marked Phase 1 settled — which derives the Phase 2 deadline from a field of the input, relative to the settlement block — and are closed out at delivery. The tracker also checks, on the node's behalf, that the executor is registered with a capability matching the requested precompile and that the delivery fee fields fit the integer widths reth assumes.

Invariants

- A job exists if and only if an escrow was reserved under the same identifier, since both are created in one commitment that reverts as a unit.
- A job's committed input never changes: every later transition requires bytes matching the hash recorded at admission.
- A job is settled at most once per phase — Phase 1 only for a two-phase precompile and delivery only after Phase 1 — and delivery removes it, so nothing is delivered twice.
- The number of live jobs never exceeds the cap and always matches the tracker's own pending count.
- Every live job's TTL, input size, and Phase 2 offset stay within the bounds checked at admission.
- Cleanup removes only jobs past their expiry block and never records progress past a bucket it did not fully drain.

- An identifier whose job has expired is available for recommitment.

Test coverage

Cases covered

- Admission and retrieval, with duplicate identifiers rejected while a job is active
- The scheduled-origin path end to end against a real Scheduler: success, reserve failure, and malformed or unauthorized references
- Executor and capability validation across precompile families, including settlement rejecting an executor invalidated after admission
- Phase 2 deadlines for every two-phase family, with rebucketing, offset rejection, and the delivery guards
- Delivery fee validation at and beyond every integer bound
- Bounded cleanup, as property and fuzz tests

Cases not covered

- Admission with an oversized input or an out-of-range TTL
- Admission at exactly the pending-job cap
- Re-admitting an identifier whose previous instance has expired (Finding [3.4](#), ↗)
- Malformed inputs at each decoder's length boundary, confirming the silent-return, revert, and zeroed-config behaviors are each intended
- Settlement context for an executor whose registry entry resolves to a zero payment address
- A non-owner upgrade, and setScheduler by a non-owner

Attack surface

Every field of a commitment originates with the user and is passed through by the node. The tracker rechecks sizes, TTL, precompile membership, executor validity and capability, and capacity, but not that the caller and sponsor correspond to the actual transaction sender — that binding lives in reth, and the wallet declines to rederive it. This is the seam Finding [3.6](#), ↗ travels through.

Three decoders read raw ABI words at fixed per-family offsets out of that input and handle short inputs inconsistently; the deadline extractor reverts, the delivery decoder returns a zeroed configuration, and the fee validator silently returns. The last means an input crafted to sit just below the fee offset skips the integer-width check entirely, leaving the node-side check as the only gate rather than a redundant one.

Capacity is the other pressure point. Cleanup is gas bounded and never advances past a bucket it did not drain, so expirations concentrated in one block are cleared slowly while newer buckets wait behind them, and sustained pressure could hold the count near its cap and make fresh commitments revert — though every admitted job requires a funded escrow, so the attack is

priced rather than free. Settlement-time revalidation is deliberately conservative in the same vein: an executor that loses registry validity midflight makes settlement revert and leaves the sponsor's escrow locked until expiry, and the redundant check noted in Finding [3.14](#), [↗](#) is unreachable for that same reason.

5.4. Component: AsyncDelivery

Description

AsyncDelivery is the settlement coordinator for async jobs and the only contract permitted to advance a job's life cycle in AsyncJobTracker or settle its escrow in RitualWallet — both are leaves that accept those calls from nobody else. It in turn accepts calls only from the reserved settlement account, so every entry point runs inside a node-built system transaction.

The `settle` function performs Phase 1: it pays the executor, the commitment validator, and the block's coinbase out of the job's escrow, resolving each destination from protocol state. A single-phase job is then closed, with the remainder returned to the sponsor, and removed; a long-running job stays open, its escrow expiry extended to the computed Phase 2 deadline. The `deliver` function performs Phase 2: it pays the executor, reserves and withdraws a callback budget — the gas allowance priced at the effective fee, plus the callback value — invokes the target, and refunds the sponsor the unused gas, plus the callback value if the call reverted.

The fee, gas, target, selector, and result fields all arrive as calldata from the settlement transaction; they are validated in reth's block verification rather than in this contract.

Invariants

This contract holds no state beyond its owner, so its invariants are per-transaction accounting properties rather than predicates over stored state.

- Within a delivery, the budget withdrawn from the escrow equals the callback value sent, plus the gas cost retained, plus the amount refunded; a refund that cannot be paid reverts the settlement rather than completing it partially.
- The total moved out of an escrow by a settlement never exceeds its remaining reserve.
- The escrow settles before the tracker's life-cycle state advances, so a failed settlement cannot leave the tracker believing a job was paid.
- Every settlement path verifies that the supplied input hashes to the value recorded at commitment.
- A job past its deadline is intended to be rejected, but neither entry point checks the tracker's own deadline; only the wallet's escrow expiry stands in for it (see Finding [3.8](#), [↗](#)).

Test coverage

Cases covered

- Phase 1 paying the derived recipients out of the sponsor's reserve
- Long-running settlement extending the escrow through the Phase 2 deadline instead of closing it
- Delivery refunding unused gas, and the callback value when the callback reverts
- Refund failure reverting on both callback outcomes and the delivery guards each rejecting before the callback runs

Cases not covered

- Either entry point from a nonsettlement account, and a non-owner upgrade
- `settle` called twice for the same job, on both the single-phase and long-running paths
- A callback consuming its entire gas allowance, or a zero callback budget
- Settlement of a job whose escrow was expired in an earlier block (Finding [3.8. 7](#))
- Settlement whose fees exceed the escrow's remaining reserve
- A callback that reenters `RitualWallet` or `AsyncDelivery`, and an end-to-end accounting assertion over a full two-phase job

Attack surface

Because both entry points are restricted to the settlement account, the realistic adversary is the user who shaped the job and controls the callback target. That target is arbitrary code invoked with a fixed gas stipend and value, but all escrow accounting happens before the call, and a reentrant callback cannot reenter `deliver`. The protection is structural rather than an explicit guard — worth remembering for any entry point added later.

The unvalidated fee fields are the largest assumption in the async pipeline: the only ceiling here is the escrow's remaining reserve, so a node-side defect could pay a job's entire escrow out in attacker-chosen amounts. Gas provisioning is a quieter form of the same assumption — if `deliver` runs with too little, the callback is silently underprovisioned while the sponsor is still charged for the envelope.

Neither entry point consults the job's own expiry (Finding [3.8. 7](#)), leaving the wallet's escrow check as the only backstop; because escrow expiry is permissionless, anyone can turn a settlement that would have succeeded into a reverting system transaction.

5.5. Component: AgentHeartbeat

Description

`AgentHeartbeat` tracks the liveness of persistent agents and drives their recovery. A persistent agent is a long-lived workload running inside a TEE executor, launched through the persistent-agent precompile as a two-phase async job sponsored by its owner. Each agent has at most one assignment, recording its address, the bound executor, the last heartbeat, a time-out, the latest manifest, and an encrypted data-availability configuration.

The agent claims its own assignment, so the caller is the address being bound. A claim is

accepted against a vacant assignment, or against an active one whose time-out has elapsed, and requires a minimum balance plus a signature from a registered executor attesting that the agent runs inside it. The bound agent then heartbeats on a cadence derived from the time-out, supplying a manifest and a fresh executor signature over an incrementing nonce, which together prove it is alive and still inside its TEE. The identity owner's only gated action is `recall`.

Maintenance runs from a reserved system account, walking the tracked agents in bounded batches to remove those that can no longer pay, vacate those that have timed out, and fail revivals whose cooldown has elapsed, until an attempt limit drops the agent altogether. Revival takes a new bound executor from the supplied precompile input and fires the precompile, with success proven only by a later fresh claim. It becomes a fresh async job whose sponsor the node derives from the registry owner, so the owner, not the agent, pays for it.

Invariants

- Each agent has at most one assignment. An active one is replaced only after its heartbeat time-out has elapsed, and is otherwise removed by recall, by administration, or by exhausting its revival attempts.
- An agent's nonce never decreases, and it increases on every claim, heartbeat, and vacation, so each executor signature is single-use and cannot be reused across bindings.
- Maintenance visits a bounded number of agents per call.
- The bound executor is registered at all times it holds authority.
- An owner's exposure to revival cost is bounded by something they approved.

Test coverage

Cases covered

- The claim, heartbeat, recall, time-out, and revival life cycle
- Signature binding, including stale-signature rejection after a lazy reclaim and replay rejection
- The deliberate absence of a registry-to-agent binding at claim
- Revival through cooldown and attempt-limit eviction, including synchronous precompile failure
- Maintenance cursor wrapping, a removal not skipping the swapped agent, and administrative removal

Cases not covered

- The pause mechanism entirely
- A claim from a caller below the minimum balance, or naming an unregistered executor
- A heartbeat signed by an executor revoked since the claim (Finding [3.9](#), ↗)
- A claim by a party the identity owner never authorized and the revival cost it imposes (Finding [3.6](#), ↗)

- Revival input shorter than one word, or decoding to an unregistered or zero executor
- Boundary and administrative rejections: an empty manifest, a nonactive assignment, the tracked-agent cap, the parameter setters at their bounds, and a non-owner upgrade

Attack surface

Every `claim` argument is attacker chosen except the executor signature, and any registered executor may sign for any agent. The cost of an unauthorized binding falls on the identity owner: the resulting revival is sponsored by them and reaches the wallet with sponsor and caller equal, skipping the allowance branch and debiting their general balance with no per-agent cap (Finding [3.6](#), ↗).

The unbounded byte fields are the sharpest griefing surface. The data-availability configuration at `claim` and the manifest at every heartbeat are stored without a length bound, and the cost is not paid only at write time: the deep wipe loops over every word on replacement or removal, and the replacement event reemits both. The replacement cost falls on the next claimant, the removal cost on maintenance — a system transaction processing up to 100 agents.

Revival is restricted to the system account, but its input is opaque, assembled off chain, forwarded verbatim to the precompile from this contract's address, and has its first word reinterpreted as the new bound executor with no registry validation. Whatever authority the precompile grants on the basis of the caller being `AgentHeartbeat` is therefore exercised on data this contract never inspects — the component's principal trust boundary, resting on node-side construction that was out of scope.

6. Assessment Results

During our assessment on the scoped Ritual (Smart Contracts) contracts, we discovered 15 findings. Two critical issues were found. One was of high impact, four were of medium impact, five were of low impact, and the remaining findings were informational in nature.

6.1. Observations and recommendations

The contracts depend on node and off-chain code that fell outside this engagement's scope

This engagement covered only the Solidity contracts listed in section 2.3. Those contracts depend on the rest of the Ritual stack — the `ritual-reth`, `ritual-reth-nodebuilder`, `ritual-revm`, `ritual-revm-inspectors`, `ritual-alloy` and `ritual-alloy-evm` forks, which implement the chain's consensus rules, block building and EVM, together with the off-chain Go executor and agent-sidecar components. These ritual stack components fell outside the scope of our review. Ritual provided view access to those repositories, and parts of them were read where a contract-level issue could not be validated without the node behavior it rests on. Findings [3.1](#), [3.2](#), [3.3](#) and [3.6](#) all depend on the node in this way, whether because a contract-level guarantee is actually enforced by the node, or because the defect and its remediation span both codebases.

No assurance can be offered about the out of scope code in general, and further issues may exist in it. The same limit applies to the remediations. The fixes for Findings [3.1](#), [3.2](#), [3.3](#) and [3.6](#) were implemented partly outside the contracts — in the node, and for [3.6](#) in the Go executor and sidecar as well — as such these remediations were reviewed on a best-effort basis.

6.2. Recommended follow-up assessment

Several of the issues found in this engagement — and several of their fixes — reach beyond the scoped contracts and into the node and off-chain components. Zellic recommends a separate audit of Ritual's `ritual-reth`, `ritual-alloy`, `ritual-revm`, `ritual-reth-nodebuilder`, `executor-go-internal` and `claw-spawner` components, which enforce security properties the contracts rely on but cannot verify themselves. Following this, for greater assurance we recommend a targeted re-review of parts of the smart contracts that are directly related to the offchain code or that will be affected by new updates from this review.

6.3. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zel-

lic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Where Zellic states that a finding has been addressed or fixed in one or more specific commits, this indicates only that those commits introduce changes that, in our assessment, address the finding relative to the codebase version originally reviewed. This does not imply that Zellic reviewed other changes contained in those commits, the overall state of the codebase at those commits, or the interplay between remediation measures for different findings.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.