Femto Security

Smart Contract Auditing

Smart Contract GIFT-V2

Ophir Ubuntu International

2026-07-08-UTribe

DOCUMENT PROPERTIES:

Title:	**Ophir Ubuntu International Smart Contract GIFT-V2 Report**
Version:	**1.0**
Document ID:	**2026-07-08-UTribe**
Status:	**Final version**
Prepared for:	**Ophir Ubuntu International**
Prepared by:	**Femto Security**
Prepared at:	**21-06-2026**
Release date:	**08-07-2026**
Document classification:	**Confidential**

Table of Contents:

Timeline of the Assessment

Type of the Assessment:	Smart Contract Auditing
Status:	Completed
Start Date:	17-06-2026
End Date:	20-06-2026
Review and Delivery of Final Report:	21-06-2026

Scope of the Assessment:

URL / IP
1. d0de7a14f5984286222b327987f5f398d7b236a8

Findings Overview:

Severity	Open	In Progress / Retest	Closed
Critical	0	0	0
High	0	0	1
Medium	0	0	5
Low	0	0	2
None	0	0	0
Total	0	0	8

Vulnerability Details and Mitigation

01. execute()/confiscate() bypass the pause switch; funds still move while the EU agent is paused

High Fixed

Last update: 08-07-2026

Description:

Only registerTicket carries whenNotPaused (:89). The four state-mutating functions that actually move GIFT or change ticket state execute (:131), freeze (:139), unfreeze (:146), confiscate (:153), have no pause guard. A pause is the emergency stop for a compliance relay, yet the entire backlog of Pending tickets can still be drained while paused, and confiscation can still push funds to regulatorWallet . Pause therefore only blocks registration of new tickets, not disbursement of funds,the opposite of what an emergency halt is meant to protect.

Impact:

An emergency pause (compromised COMPLIANCE key, detected exploit, regulator order) does not stop GIFT from leaving the contract via execute() or confiscate() . All previously-registered Pending tickets remain spendable, and JUDICIAL_ROLE can still push the full ticket amount to regulatorWallet , defeating the kill switch.

Proof of Concept:

Exploit scenario.

(1) A COMPLIANCE key is suspected compromised; PAUSE_ROLE calls pause() .

(2) registerTicket() is blocked, so operators believe funds are safe.

(3) The attacker holding the compromised COMPLIANCE key calls execute(ticketId) repeatedly; none revert under pause, draining every Pending ticket's GIFT to the to addresses recorded at registration time.

Reachability caveat. Exploitation is privilege-gated (COMPLIANCE/JUDICIAL role), so the impact materializes when those roles are compromised/misused, which is exactly the threat model a pause exists to contain. EUTransferAgent is not yet deployed; this is a pre-deployment fix.

Remediation:

Add whenNotPaused to execute , freeze , unfreeze , and confiscate (at minimum the two fund-moving paths, execute and confiscate). The disbursement path is exactly what a pause must be able to stop.

```
<code class="language-solidity">89:       whenNotPaused   // only on registerTicket
131:    function execute(bytes32 ticketId) external onlyRole(COMPLIANCE_ROLE) {
132:        Ticket storage t = tickets[ticketId];
133:        require(t.state == State.Pending, "not pending");
134:        t.state = State.Executed;
135:        gift.transfer(t.to, t.amount);
...
153:    function confiscate(bytes32 ticketId) external onlyRole(JUDICIAL_ROLE) {
160:        gift.transfer(regulatorWallet, t.amount);
```

02. inflateSupply() is an unbounded, unbacked mint bypassing PoR/MintControllerV2 and the pause guard

Medium Fixed

Last update: 08-07-2026

Description:

The V2 architecture routes legitimate mints through `supplyController` (MintControllerV2) so every GIFT minted is backed by a Proof-of-Reserve proof. `inflateSupply(uint256 _value)` is a separate second mint path that calls `_mint(supplyManager, _value)` with (a) no amount bound, (b) no PoR/reserve check (`reserveFeed` is never read on-chain), and (c) no `whenNotPaused` (unlike `increaseSupply`, which has it). Authorization is only `onlySupplyManager`. `supplyManager` is set by `setSupplyManager` (`onlyOwner`, `:335, which,` unlike `supplyController` is **not**locked: there is `supplyControllerLocked` /`lockSupplyController`, but no equivalent for `supplyManager`. `initializeV2` never sets `supplyManager`, so on a fresh proxy it defaults to `address(0)` and the function is dormant, but once the owner Safe (or a compromised owner key) calls `setSupplyManager(X)`, X can mint unlimited unbacked GIFT, even while paused.

Impact:

Direct, unbounded inflation of a gold-backed token with zero reserve backing, ignoring the emergency pause. A single call can mint arbitrary GIFT to `supplyManager`, destroying the 1:1 peg and diluting every holder.

Proof of Concept:

Exploit scenario.

(1) Owner calls `setSupplyManager(attacker)` (or owner key compromised).

(2) `inflateSupply(1e30)`.

(3) 1e30 unbacked GIFT minted with no PoR proof, no reserve check, ignoring pause.

(4) Attacker dumps on liquidity pools.

Remediation:

Remove `inflateSupply` entirely (the supported path is `increaseSupply` via MintControllerV2/PoR). If a non-PoR mint must exist, gate it behind the same reserve/PoR accounting, add `whenNotPaused`, add a hard per-call and cumulative cap, and lock `supplyManager` analogously to `supplyControllerLocked`. At minimum confirm `supplyManager` is permanently `address(0)` on the deployed proxy and add a test asserting `inflateSupply` always reverts.

```
<code class="language-solidity">361:    function inflateSupply(uint256 _value) external onlySupplyManager returns (bool) {
362:        _mint(supplyManager, _value);   // no PoR, no cap, no whenNotPaused
363:        return true;
364:    }
// Contrast — backed path HAS whenNotPaused and flows through supplyController/MC V2
366:    function increaseSupply(address _userAddress, uint256 _value)
369:        external onlySupplyController whenNotPaused returns (bool)
// supplyManager freely set by owner, no lock parity with supplyControllerLocked
335:    function setSupplyManager(address _newSupplyManager) external onlyOwner { ... }
```

03. ax-on-transfer breaks bridge backing on deposit: full amount minted on Solana, net-of-tax locked on Polygon

Medium Fixed

Last update: 08-07-2026

Description:

This is the deposit-side, configuration-gated facet of the supply-conservation problem (GIFT-01 is the broader backing/insolvency framing rated High). `depositToSolana` pulls GIFT via `transferFrom` (`:105`) then emits `DepositedToSolana(..., amount, nonce)` with the full pre-tax `amount` (`:110-115`). `GIFT.transferFrom` routes through `_transferGIFT`, which deducts outbound tax and delivers only `amount - totalTax` to the bridge (`GIFT.sol:874`). Unless both the depositing sender and the bridge are fee-excluded / liquidity-pool in `GIFTTaxManager`, the pool receives strictly less than `amount`. The relayer mints `amount` GIFT_SOL keyed off the emitted event value, not the realized balance delta; `mint_from_polygon` blindly mints the relayer-supplied amount.

Impact:

`GIFT_SOL_supply` can exceed locked GIFT by the cumulative tax taken on every non-excluded deposit, violating the 1:1 invariant. Late withdrawers drain the pool and eventually cannot be paid, undercollateralization on a gold-backed asset.

Proof of Concept:

(1) Bridge/depositor not added to fee-exclusion (the `GiftPolygonBridge.sol:104` comment literally says "this is where tax exemption matters", acknowledging but not enforcing it).

(2) `depositToSolana(1000e18, solRecipient)`.

(3) 2% tax routes to the beneficiary; pool rises by only 980e18.

(4) Event emits `amount=1000e18`.

(5) Relayer mints 1000 GIFT_SOL.

(6) Over time: GIFT_SOL outstanding = Σ full amounts, locked = Σ (amount − tax); the tail of holders is unredeemable.

Reachability caveat. Gated on the bridge/depositor not being exclusion-configured. The Solana program ID is still a placeholder, so the Solana side is not yet live, lowering currently-demonstrable impact (hence Medium vs GIFT-01's High framing).

Remediation:

Do not trust the emitted/parameter amount. Measure the realized lock via `balanceOf` delta, emit `lockedDelta` as the bridge credit, and have the relayer mint exactly `lockedDelta`. Additionally enforce bridge fee-exclusion on-chain (revert in `depositToSolana` if realized delta ≠ amount) and add a global invariant so `completeWithdrawalFromSolana` can never release more than net locked.

```
// GiftPolygonBridge.sol:104-115
// Pull GIFT from sender to bridge (this is where tax exemption matters)
bool ok = gift.transferFrom(msg.sender, address(this), amount);
require(ok, "Bridge: transferFrom failed");
uint256 currentNonce = ++depositNonce;
emit DepositedToSolana(msg.sender, solanaRecipient, amount, currentNonce);
// GIFT.sol:868-874 — tax skimmed before recipient credit
_transfer(sender, recipient, amount - totalTax);
```

04. GIFT_reserve accumulator is never decremented; PoR totals permanently overstate backing

Medium Fixed

Last update: 08-07-2026

Description:

`GIFT_reserve` is the contract's headline total-reserve figure exposed by `retrieveReserve()` and `getTotalReserves()`. The only statement that writes it is the increment in `updateVault` (`:253 GIFT_reserve += _amountAdded;`). It is **never**decremented, not in `updateReserveAfterMint` (which debits `vault.amount` only, `:524`), not in `RedeemGold` (debits physical only), not in `moveSupply`, not on burn. Grepping all writers confirms a single `+=` and zero `-=`. As gold is minted out or physically redeemed, `GIFT_reserve` stays at its all-time-high cumulative-inflow value.

Impact:

Any consumer (exchange, integrator, auditor dashboard, redemption escrow, public) reading `retrieveReserve()` /`getTotalReserves()` as the proof-of-reserve total gets a monotonically-increasing figure that diverges upward from actual remaining reserves over the system's life. For a proof-of-reserve contract this is a correctness/transparency failure that can mislead reserve-ratio attestations (overstating backing)

Proof of Concept:

Exploit scenario

`updateVault(1, 100, ...)` '`GIFT_reserve = 100`. Mint 100 via MintControllerV2 => `updateReserveAfterMint(1,100)` sets `vault.amount` to 0 but `GIFT_reserve` stays 100. `RedeemGold` reduces physical, `GIFT_reserve` still 100. `retrieveReserve()` reports 100 against zero digital reserve and reduced physical gold. Repeated cycles make the reported total an ever-growing fiction.

Remediation:

Decrement `GIFT_reserve` in `updateReserveAfterMint` by `_amount` (and on any physical redemption that reduces the backed total), or derive the total on-read by summing `vaultsById` balances instead of maintaining a write-only-up accumulator. Add a unit test asserting `GIFT_reserve == sum(vault.amount)` after a mint cycle.

```
<code class="language-solidity">// Only writer (increment) — legacy/GIFTPoR.sol:253
GIFT_reserve += _amountAdded;
// Readers — legacy/GIFTPoR.sol:423-431
totalAmount = GIFT_reserve;
function retrieveReserve() public view returns (uint256) { return GIFT_reserve; }
// Mint debit does NOT touch GIFT_reserve — legacy/GIFTPoR.sol:524
vault.amount -= _amount;
```

05. completeRedemption is permanently bricked; both burn primitives unconditionally revert

Medium Fixed

Last update: 08-07-2026

Description:

`completeRedemption()` is the only success path that consumes a redemption: it calls `minting.burnEscrowBalance(amount)` to burn escrowed GIFT, then burns the NFT. But `burnEscrowBalance()` does nothing except `revert LegacyBurnPathDisabled()` (`MintControllerV2.sol:584-587`). The call is a plain external call (not wrapped in try/catch, only the subsequent NFT `burn()` is, at `:331`), so the revert propagates and the entire `completeRedemption()` reverts every time. Even if `burnEscrowBalance` were re-enabled, its downstream `GIFT.redeemGold(escrow, amount)` is also disabled (`revert LegacyRedeemGoldDisabled()`), so the burn chain is doubly dead. The escrow imports root `contracts/polygon/GIFT.sol`, whose `redeemGold` likewise reverts, the conclusion holds regardless of which GIFT copy is referenced.

Impact:

Redemption can never complete on-chain. GIFT locked against NFTs that have been physically redeemed/shipped is never burned, so token supply permanently overstates backing gold (supply-conservation break). Operationally the only exits are `adminRefundRedemption` / `adminConfiscateRedemption`, neither of which burns GIFT;the redeem-and-burn product flow is entirely inoperable. The repo's own `test_CompleteRedemption_FullFlow` asserts the burn succeeds, which is impossible against current source, independent corroboration showsthe live burn primitive is broken.

Proof of Concept:

Exploit scenario.

(1) Marketplace calls `lockGiftForNFT`; escrow holds GIFT for `tokenId`.

(2) User transfers the NFT into escrow (`inRedemption=true`).

(3) Gold is shipped; admin calls `completeRedemption(nft, tokenId)`.

(4) `minting.burnEscrowBalance(amount)` hits `revert LegacyBurnPathDisabled()` The whole tx reverts. The record stays `inRedemption` forever; GIFT is never burned.

Remediation:

Re-implement `burnEscrowBalance()` and `GIFT.redeemGold()` to actually burn the escrow's GIFT, or repoint `completeRedemption()` at the live burn primitive. Until a working burn path exists, do not advertise the function as functional. Add an integration test asserting `completeRedemption` succeeds end-to-end.

```
<code class="language-solidity">// Escrow completeRedemption — burn NOT wrapped in try/catch
324:        minting.burnEscrowBalance(amount);
331:        try GIFTBarNFTDeferred(nftContract).burn(tokenId) { } catch { }
// MintControllerV2.sol
584:    function burnEscrowBalance(uint256 amount) external nonReentrant whenNotPaused {
586:        revert LegacyBurnPathDisabled();
587:    }
// GIFT.sol redeemGold likewise: revert LegacyRedeemGoldDisabled();
```

06. NFT bar minting (default LEAF_MAX) is decoupled from ERC-20 registry consumption; same gold can back both

Medium Fixed

Last update: 08-07-2026

Description:

In the default `capacityMode == LEAF_MAX`, `_capacityUnits` returns `fineWeightMg / unitMg` computed purely from the Merkle leaf, and `mintBarsFromLeaf` only calls `registry.verifyLeaf` a view itnever calls `registry.consume`, tracking units solely in its own local `unitsMinted[(batchId,leafHash)]` map. The ERC-20 supply path (MintControllerV2 => `registry.consume`) maintains a completely separate `_consumed` counter. Nothing links the two: the same leaf representing `fineWeightMg` of gold can be fully consumed to mint ERC-20 GIFT **and**, in `LEAF_MAX`, fully minted as NFT bars up to `fineWeightMg/unitMg`, double-counting the same physical gold across both product lines. Only the non-default `STRICT_CONSUMED` mode caps NFT capacity by `registry.leafConsumed`. `deployNewCore.ts` never calls `setCapacityMode`, so production runs in the vulnerable default.

Impact:

If `LEAF_MAX` is active, total claims on a leaf's gold (ERC-20 GIFT + NFT bars) can exceed `fineWeightMg`, overstating backing. No on-chain guard prevents it; correctness depends entirely on the owner choosing `STRICT_CONSUMED` and on operational discipline. Both lines are real, redeemable claims on the same physical gold (the escrow redeems NFT bars), so the double-counting has genuine economic meaning.

Proof of Concept:

Exploit scenario. Owner registers a leaf with `fineWeightMg = 1,000,000`. MintController consumes the full leaf to mint 1,000,000 GIFT. Separately, with `LEAF_MAX` default, owner calls `mintBarsFromLeaf` for 1 bar (`unitMg=1,000,000`) allowed because `cap = 1,000,000/1,000,000 = 1` and registry consumption is ignored. The same kilo now backs both 1,000,000 GIFT and one redeemable NFT bar.

Remediation:

Make `STRICT_CONSUMED` the only/default mode, or have `mintBarsFromLeaf` call `registry.consume` (or debit a shared leaf budget) so NFT bars and ERC-20 GIFT draw from the same counter. At minimum, document that `LEAF_MAX` must never be used for batches also backing ERC-20 supply, and add an invariant test.

```
<code class="language-solidity">// mintBarsFromLeaf — only verifyLeaf, no consume
120:       (bytes32 leafHash, bool ok) = registry.verifyLeaf(leaf, proof);
123:       uint256 cap = _capacityUnits(leaf.batchId, leafHash, leaf.quantity, leaf.fineWeightMg);
126:       require(minted + units <= cap, "exceeds capacity");
// _capacityUnits LEAF_MAX ignores registry
155:       if (capacityMode == CapacityMode.LEAF_MAX) {
156:           return fineWeightMg / unitMg;
157:       }
```

07. Tax beneficiary transfers silently bypass the recipient-side frozen/blocklist compliance check

Low Fixed

Last update: 08-07-2026

Description:

New in V2: `_beforeTokenTransfer` computes `toIsTaxBeneficiary = _isTaxBeneficiary(to)` and, when true, skips the `frozen[to]` and `blocklisted[to]` checks (to let tax allocations reach the beneficiary even if flagged). But this gate is evaluated for **all** transfers, not just tax-allocation legs, so any user can move tokens **into** the tax-beneficiary address even after compliance has frozen or blocklisted it. Impact is bounded: `from` is always checked, so a frozen/blocklisted user still cannot send, and the beneficiary only gains the ability to **receive** while flagged (it cannot send out). The beneficiary address is whatever `taxManager.beneficiary()` returns (set via `setBeneficiary`, `onlyOwner` a trusted role, not attacker-chosen).

Impact:

A frozen or blocklisted address that is the configured tax beneficiary can still receive arbitrary GIFT from any sender, partially defeating the freeze/blocklist control on that specific address. The compliance team cannot fully quarantine inbound to the flagged beneficiary without changing the TaxManager beneficiary.

Proof of Concept:

Exploit scenario. The tax beneficiary becomes subject to freeze/blocklist (compromised or sanctioned). Despite the freeze, any user calls `transfer(beneficiary, amount)` and it succeeds because the `to`-side check is skipped for the beneficiary.

Remediation:

Restrict the beneficiary frozen/blocklist exemption to only the tax-allocation legs inside `_transferGIFT` (e.g. via a transient flag set only while paying allocs), rather than exempting every transfer whose `to` equals the beneficiary in the global `_beforeTokenTransfer` hook.

```
<code class="language-solidity">944:       if (!moduleSourcesOfTruth) {
945:           bool toIsTaxBeneficiary = _isTaxBeneficiary(to);
946:           if (frozen[from] || (!toIsTaxBeneficiary && frozen[to])) {
947:               revert ComplianceBlocked("FROZEN", from, to);
948:           }
949:           if (blocklisted[from] || (!toIsTaxBeneficiary && blocklisted[to])) {
950:               revert ComplianceBlocked("BLOCKLISTED", from, to);
951:           }
952:       }
```

08. Legacy delegateTransfer() hardcodes deadline = type(uint256).max, producing never-expiring signed authorizations

Low Fixed

Last update: 08-07-2026

Description:

`delegateTransfer` (the backward-compatible entry) calls `delegateTransferV2` with `deadline = type(uint256).max`. Because the EIP-712 digest includes `deadline`, a delegator using this legacy path signs a transfer authorization that never expires. Replay is still prevented by the per-delegator monotonic nonce (`:782`) and the signature binds the relayer to `msg.sender`; the domain separator includes `block.chainid` (`:980`), so this is not a replay or cross-chain issue. The residual is the classic never-expiring meta-transaction signature: if such a signature leaks or is captured before relay (e.g. from an off-chain relayer queue), it remains valid indefinitely until the delegator's nonce advances. The signed `amount`/`networkFee` are fixed, bounding loss to that one authorization.

Impact:

A leaked/intercepted legacy delegate-transfer signature can be relayed at any future time (no expiry), moving the signed `amount` (+ `networkFee`) from the delegator. Bounded to a single authorization and consumed by the nonce.

Proof of Concept:

Exploit scenario. A delegator signs a legacy (no-deadline) `delegateTransfer` and the signature is exposed before relay. An attacker who is also a MANAGER (`onlyManager` gate), or who can get a manager to relay, submits it later; with `deadline=max` there is no time-window protection.

Remediation:

Deprecate/disable the no-deadline `delegateTransfer` path (as was done for `delegateTransferProof`/`redeemGold`) and require an explicit, bounded deadline on all delegated transfers; or have `delegateTransfer` inject a short default (`block.timestamp + N`) instead of `type(uint256).max`.

```
<code class="language-solidity">736:    function delegateTransfer(...) external whenNotPaused onlyManager returns (bool) {
737:        return delegateTransferV2(signature, delegator, recipient, amount, networkFee, type(uint256).max);
738:    }
762:    if (block.timestamp > deadline) revert SignatureExpired(deadline, block.timestamp); // always false for max
782:    nonces[delegator] = nonce + 1;
```

Methodology

FemtoSec's skilled cybersecurity experts follow a precise methodology to make sure they cover every aspect of your systems during the assessment and detect vulnerabilities that may be overlooked by other parties or automated scanner. All of our Vulnerability Assessments and Penetration Testing Engagements go through the following phases

1. **Reconnaissance:**
   The process of collecting as much information as we can about the system to have better mapping of the attack surface before conducting any real attacks.
2. **Enumeration:**
   The process of identifying the likely entry points into the target system.
3. **Vulnerability Analysis:**
   The process in which our team tries to identify and locate security weaknesses within the attack scope.
4. **Exploitation:**
   The process in which we use the discovered vulnerabilities to achieve an acceptable/agreed-on security impact on the vulnerable system.
5. **Post-Exploitation:**
   The process of escalation of the privileges gained from exploitation phase and trying to perform lateral movement.
6. **Reporting:**
   The process of documenting the details of the discovered vulnerabilities and all the steps that lead to a successful attack.

Penetration Testing Methodology

The severity of the findings is classified based on the Common Vulnerability Scoring System (CVSS) Version 3.