VLT Security Audit

Claude Opus 4.7 with Solidity Audit Skill

April 22, 2026

VLT (Bankroll Vault) — Security Audit

Contract: `VaultToken` (VLT — Bankroll Vault Token) **Address:** `0x6b785a0322126826d8226d77e173d75dafb84d11` **Network:** Ethereum **Compiler pragma:** `>=0.6.2` **Performed by:** Claude Opus 4.7 with Solidity Audit Skill **Date:** April 22, 2026

Executive Summary

`VaultToken` is a small, single-purpose, fixed-supply ERC20 with a one-shot `bootstrap()` function that seeds a Uniswap V2 pool and effectively locks the LP tokens by minting them to the contract's own address. There is no mint function, no pause, no blocklist, no upgrade path, no `selfdestruct`, and no `delegatecall`. The deployer (the `owner`) has exactly one privilege — calling `bootstrap()` — and loses even that after the first invocation.

For a 2020-era contract, the code is clean. The identified weaknesses are stylistic (open compiler pragma, storage variables not declared `immutable`, missing zero-address guards) rather than exploitable. No function in the contract grants any party the ability to mint, pause, censor, or drain liquidity.

Findings Summary

Severity	Count
Critical	0
High	0
Medium	0
Low	3
Informational	8

Scope

File	Contract	Lines (approx.)
`VaultToken.sol`	`VaultToken`, `SafeMath` library	~240

Interfaces (`IUniswapV2Router01/02`, `IERC20`, `ApproveAndCallFallBack`) are external definitions and were reviewed only to the extent they affect the contract under audit. This audit covers source code analysis only; it does not include dynamic testing, fork simulation, or bytecode-to-source verification against any deployed contract.

Architecture Overview

Contracts

- `VaultToken` — ERC20 implementation with bootstrap mechanism and burn functionality.
- `SafeMath` **(library)** — Standard overflow-checked arithmetic, required because the pragma allows compilers below 0.8.

External Dependencies

- **Uniswap V2 Router** — Called exactly once during `bootstrap()` to create the initial liquidity pool and mint LP tokens to the contract address.

Actors

Actor	Capabilities	Notes
Deployer (`owner`)	Call `bootstrap()` exactly once	Loses all privileges after bootstrap
Token holders	Standard ERC20 operations, `burn()`, `multiTransfer()`, `approveAndCall()`	No special privileges
Uniswap Router	Receives tokens and ETH during bootstrap to create pair	Called once

State Machine

The contract has two states:

1. **Uninitialized** — after deployment, before `bootstrap()` runs.
2. **Bootstrapped** — after `bootstrap()` completes (flag `isBootStrapped` flips to `true`).

The transition is one-way and gated by `require(isBootStrapped == false)`.

Core Invariants

1. `totalSupply` is monotonically non-increasing (only `burn()` modifies it, and only by subtraction).
2. Sum of all balances equals `totalSupply`.
3. The deployer can act privileged only via `bootstrap()`, which can run at most once.
4. LP tokens held at the contract address are unrecoverable — no function transfers them out.
5. No function can increase `_totalSupply` above its initial value of 1,800,000 × 10^18.

Function-Level Analysis

`constructor(address routerAddr)`

Stores the Uniswap V2 router address. Implicitly sets `owner = msg.sender` via the state variable initializer. Performs no validation on `routerAddr`.

`bootstrap() external payable`

One-shot initializer. Guards: `!isBootStrapped`, `msg.sender == owner`, `msg.value >= 1 ether`. Distributes 89% of supply to the deployer (for OTC and market-making distribution per the in-code comment) and 11% to the contract itself, which is then paired with the sent ETH via `addLiquidityETH`. LP recipient is set to the contract address, rendering the LP permanently locked. Sets `isBootStrapped = true` at the end.

ERC20 surface

`transfer`, `transferFrom`, `approve`, `increaseAllowance`, `decreaseAllowance`, `allowance`, `balanceOf`,

`totalSupply` — all standard implementations using SafeMath for balance arithmetic. `transfer` and `transferFrom` validate `to != address(0)`.

`multiTransfer(address[], uint256[])`

Batches `transfer` calls in a loop. Relies on underlying `transfer` for all validation; performs no length check on the two arrays.

`approveAndCall(address, uint256, bytes)`

Sets allowance and then calls `receiveApproval` on the spender. External call happens after state is updated.

`burn(uint256 amount)`

Allows a holder to burn their own balance. Decrements both `_totalSupply` and the caller's balance, and emits a `Transfer` to `address(0)`. No one can burn another holder's tokens.

Findings

L-01 — `owner` not declared `immutable`

Severity: Low

Location: `VaultToken` state variable declaration

```
address public owner = msg.sender;
```

This is a mutable storage variable initialized at deployment. There is no setter, so in practice the value cannot change after construction.

Impact: No exploit path exists, because no function writes to `owner` and the variable is read only inside `bootstrap()` — which can run at most once. However, the storage-variable pattern is functionally equivalent to, but not bytecode-equivalent to, a properly `immutable` declaration.

Recommendation: In future contracts, declare `address public immutable owner;` and assign in the constructor. This provides stronger guarantees at both source and bytecode levels.

L-02 — `router` is also a mutable storage variable

Severity: Low

```
IUniswapV2Router02 public router;
```

Same pattern as `owner` — no setter, but not declared `immutable`. Used exactly once in `bootstrap()`. No exploit path exists in the deployed contract, but the same recommendation as L-01 applies for future work.

L-03 — No zero-address check on `routerAddr` in constructor

Severity: Low (pre-deployment only)

If deployed with `address(0)`, `bootstrap()` would fail on the external call, bricking the token pre-launch. This is a deploy-time concern only; a successfully bootstrapped contract has, by definition, a valid router.

Recommendation: Add `require(routerAddr != address(0))` in the constructor of future deployments.

I-01 — `approveAndCall` missing zero-address check on `spender`

Severity: Informational

`approve()` checks `spender != address(0)` but `approveAndCall()` does not. The call would fail on `receiveApproval` if `spender == 0`, so the missing check is self-correcting rather than exploitable — but the inconsistency is worth noting.

I-02 — `multiTransfer` does not validate array lengths

Severity: Informational

```
for (uint256 i = 0; i < receivers.length; i++) {
    transfer(receivers[i], amounts[i]);
}
```

If `amounts.length < receivers.length`, the loop reverts out-of-bounds mid-iteration. Because the whole transaction reverts, no partial state change persists — this is fail-safe. An explicit `require(receivers.length == amounts.length)` at the top would be cleaner and cheaper to fail.

I-03 — Slippage parameters hardcoded to 1 in `bootstrap()`

Severity: Informational (accepted by design)

```
router.addLiquidityETH.value(msg.value)(
  token, balances[token], 1, 1, token, now + 1 hours
);
```

`amountAMin = 1, amountBMin = 1` offers no slippage protection. This is acceptable here because the pair does not yet exist — `bootstrap()` creates the initial LP, so there is no prior price to front-run.

I-04 — `bootstrap()` math partially bypasses SafeMath

Severity: Informational

```
balances[owner] = _totalSupply * 89 / 100;
balances[token] = _totalSupply.sub(balances[owner]);
```

The `.sub()` call uses SafeMath; the `* 89 / 100` expression does not. With `_totalSupply = 1.8e24`, the intermediate product `1.602e26` is well below `type(uint256).max` (~1.16e77) — no overflow is possible. Inconsistent style, but not a bug.

I-05 — Standard ERC20 approve race condition

Severity: Informational (widely accepted)

`approve()` allows non-zero → non-zero updates without requiring a zero reset first. This is the well-known ERC20 front-running vector. Mitigated by the presence of `increaseAllowance` and `decreaseAllowance`. Widely accepted as standard ERC20 behavior.

I-06 — `bootstrap()` sets `isBootStrapped = true` after external call

Severity: Informational

The external call to `router.addLiquidityETH` happens before `isBootStrapped = true`. In isolation this would be a Checks-Effects-Interactions violation. Reentrancy is infeasible here: the only callback path would be Uniswap calling `transferFrom` on VLT itself, and `transferFrom` does not call out to any external contract. No exploit exists, but reordering the assignment to precede the external call would be more defensive.

I-07 — Use of deprecated `now` alias

Severity: Informational

`now` is an alias for `block.timestamp` (deprecated in Solidity 0.7+). Fine for this pragma; cosmetic.

I-08 — Open-ended compiler pragma `>=0.6.2`

Severity: Informational

Open-ended pragmas are discouraged as a general practice. The code would also compile under 0.8.x, which changes arithmetic semantics (built-in overflow checks would layer on top of SafeMath without conflict). Since the contract is already deployed, the recommendation applies to future deployments: pin an exact compiler version.

Security Posture Checklist

Item	Status
No mint function	Pass
No `selfdestruct`	Pass
No `delegatecall`	Pass
No proxy / upgradeability	Pass
No pausability	Pass
No blocklist / denylist	Pass
No fee-on-transfer behavior	Pass
No rebasing or dynamic balance logic	Pass
LP tokens locked (sent to contract address)	Pass
Deployer privileges limited to one-shot bootstrap	Pass
Bootstrap flag-guarded against repeat calls	Pass
Burn decrements `totalSupply` correctly	Pass
SafeMath used for all balance arithmetic	Pass
`owner` declared `immutable`	Fail (cosmetic)
`router` declared `immutable`	Fail (cosmetic)
Zero-address checks on all constructor parameters	Fail (low)
Pinned compiler version	Fail (info)

Conclusion

VLT is a minimal, non-upgradeable, fixed-supply ERC20 with a one-shot initializer and permanently locked liquidity. No function present in the contract grants any party the ability to mint, pause, censor, or drain liquidity.

No critical, high, or medium severity findings were identified. The three low-severity findings are cosmetic or pre-deployment-only and do not affect the security posture of the deployed contract. The eight informational findings are stylistic and are noted primarily for reference in future deployments.

This audit was performed by Claude Opus 4.7 using the Solidity Audit Skill, based on the Trail of Bits Security Skills Marketplace methodology and industry best practices. It reflects analysis of the provided source code only. Readers should independently verify that the source audited matches the verified source on Etherscan at the deployment address.