Independent Security Audit Report

NFT42

Prepared by: Nick Bailo

Date: September 22, 2025

This report provides an independent security assessment of the NFT42 and MintGuard smart contracts. It is based on manual code review, test analysis, and business logic evaluation. No vulnerabilities were identified.

Scope

This audit covers the following components:

- `NFT42` — ERC721 token with immutable `MAX_TOKENS` cap and restricted minting via MintGuard.
- `MintGuard` — upgradeable sale manager handling voucher-based minting, fee collection, withdrawals, and sale start.
- Deployment scripts: `DeployNFT42`, `DeployMintGuard`.

Repository: https://github.com/ltvprotocol/42/tree/codefreeze
Commit hash: `a1c8d4310a73d6c0a1b147a7e410071e7ef288a9`

Compiler targets: `pragma solidity ^0.8.28` (contracts), `pragma solidity ^0.8.24` (scripts).

Methodology

Manual review focused on correctness and clarity, avoiding tool-driven false positives:

- Architecture & responsibilities, access control (`onlyOwner`, `onlyMintGuard`).
- Mint/supply constraints: `MAX_TOKENS`, one-mint-per-address invariants.
- Payments/withdrawals: exact-fee checks, ETH handling, `nonReentrant`.
- Signatures: voucher verification and replay context.
- Reentrancy and CEI (Check–Effects–Interactions) considerations.
- Upgradeability: initializer safety, storage layout, proxy wiring.
- Events: coverage and semantics for off-chain analytics.
- Deployment & ops sequencing (linking NFT, sale start, admin batches).

Test coverage. The repository includes a comprehensive suite of tests, covering boundary conditions, fuzzing, event emission, signature paths, signer updates, and withdrawals.

Executive Summary

The contracts are concise and secure:

- No vulnerabilities identified that compromise funds or NFTs.
- Strict enforcement of supply cap by `NFT42`.
- Safe handling of payments, withdrawals, and access control; reentrancy protections present.
- Upgradeability pattern is correct and properly initialized.
- The test suite provides strong coverage, including fuzzing and edge cases.

The primary design consideration is `start()` in `MintGuard`, which enables public minting and can optionally perform admin minting. This is intentional and safe, but it ties directly to the business policy for unsold NFTs (treasury mint to reach cap vs. demand-driven final supply). The policy should be documented.

1 Findings

1.1 F1 — `start()` behavior and unsold supply policy

```
function start(address to, uint256 amount) external onlyOwner nonReentrant {
    mintStarted = true;
    emit MintStarted();

    if (to != address(0) && amount > 0) {
        require(address(nft) != address(0), ZeroAddress());

        for (uint256 i = 0; i < amount; i++) {
            uint256 tokenId = nft.mint(to);
            emit Minted(to, tokenId);
        }
    }
}
```

Listing 1: MintGuard.start()

Observation. The `start()` function enables public minting by setting `mintStarted = true` and can additionally mint a batch of NFTs to a specified address. It may be called multiple times, which allows several admin mint batches over time. All mints (including admin batches) go through `NFT42.mint`, which enforces the `MAX_TOKENS` cap.

Why this is might be considered as a bug or a product choice. The function intentionally merges two responsibilities: (1) toggling public sale and (2) performing admin minting. Whether this is desired depends on how the project handles *unsold supply*. If the sale closes with items remaining below the cap, the team must decide: mint the remainder into a controlled wallet (treasury) to reach the advertised fixed supply, or leave them unminted so the final supply reflects demand. Both approaches are valid and safe with the current code; they simply imply different collection semantics and expectations.

Unsold supply policy — two valid models and when to choose each:

- **Fixed-supply (treasury mint of unsold).** *Why:* keeps the collection at exactly `MAX_TOKENS`; preserves pre-computed rarity math and trait distributions; gives the project/DAO inventory for future utility (rewards, partnerships, quests, community grants). *Trade-offs:* creates a visible inventory overhang and requires strong transparency (treasury address, vesting/lock, usage rules) to avoid perception of arbitrary supply release. *Choose this if:* your brand/story relies on a fixed number (e.g., "1024 forever"); rarity tiers were curated for the full set; the roadmap needs inventory under governance control.
- **Demand-driven (do not mint the unsold).** *Why:* final supply becomes strictly market-driven; scarcity increases if demand is lower than the cap; optics are simple—no retained stock. *Trade-offs:* trait/rarity distributions may differ from the originally intended full set; some traits may never appear on-chain; messaging must be clear that the cap is a hard upper bound, not a promise to fill. *Choose this if:* you prefer lean supply and "no overhang", do not need inventory for future programs, and are comfortable with rarity emerging from demand rather than a fixed full set.

Operational notes. If you retain the dual role of `start()`, document that it can be invoked multiple times to mint reserves (and record where those reserves go). If you prefer stricter semantics, split responsibilities later into `startSale()` (toggle only) and `adminMint()` (reserve), but this is an optional clarity change—not a security requirement.

Recommendation. Publish a clear, immutable sale policy before launch/end-of-sale:

1. Declare whether unsold NFTs will be treasury-minted to reach the cap or left unminted;
2. If treasury-minted, disclose the destination address, any lock/vesting, and permissible uses;
3. Reflect the choice in events/announcements so analytics and marketplaces can track the final supply decision.

Code reference: https://github.com/ltvprotocol/42/blob/codefreeze/src/MintGuard.sol#L103

1.2 F2 — Event semantics: payer vs. recipient

```
tokenId = nft.mint(voucher.minter)
emit Minted(msg.sender, tokenId); // public mint (payer logged)
```

Listing 2: Minted event and usage

Observation. In public mint, the event logs the payer (`msg.sender`), while the NFT is minted to `voucher.minter` (recipient). In admin mint, payer=recipient.

Analysis. Off-chain analytics might assume buyer equals recipient and mislabel public mints.

Recommendation. Document current semantics, or in future upgrades emit an event containing both payer and recipient.

Code reference: https://github.com/ltvprotocol/42/blob/codefreeze/src/MintGuard.sol#L92

1.3 F3 — Signature replay context

```
function verifyVoucher(Voucher calldata voucher) private view returns (bool
    ) {
    address signer = voucherSigner;
    require(signer != address(0), ZeroAddress());

    // Hash the voucher payload
    // forge-lint: disable-next-line
    bytes32 digest = keccak256(abi.encodePacked(voucher.minter));

    // Verify signature
    address recovered = ECDSA.recover(digest, voucher.v, voucher.r, voucher
        .s);
    require(recovered == signer, InvalidSignature());
    return true;
}
```

Listing 3: MintGuard.verifyVoucher()

Observation. The voucher digest only encodes the minter address.

Analysis. Replay within this contract is blocked by the one-mint-per-address rule. Replay across other contracts does not increase risk, as an attacker's clone could remove checks entirely. Therefore, replay does not create a practical threat here.

Recommendation. Acceptable as-is for EOAs with one-per-minter logic. If stricter provenance is required later, migrate to EIP-712 with domains, nonces, and deadlines.

Code reference: `https://github.com/ltvprotocol/42/blob/codefreeze/src/MintGuard.sol#L135`

1.4 F4 — State update order and CEI

```
function mint(address to) external onlyMintGuard returns (uint256 tokenId) {
    tokenId = ++totalSupply;
    require(totalSupply <= MAX_TOKENS, MaxTokensReached(MAX_TOKENS));
    _safeMint(to, tokenId);
}
```

Listing 4: NFT42.mint()

Observation. `totalSupply` is incremented before the cap check.

Analysis. Reverts roll back state, so safety holds. CEI is a style guideline; no issue here.

Recommendation. Optionally reorder: compute `next`, check, then assign.

Code reference: `https://github.com/ltvprotocol/42/blob/codefreeze/src/42.sol#L55`

1.5 F5 — `setVoucherSigner(0)` as pause mechanism

```
function setVoucherSigner(address _newSigner) external onlyOwner nonReentrant {
    address old = voucherSigner;
    voucherSigner = _newSigner;
    emit VoucherSignerUpdated(old, _newSigner);
}
```

Listing 5: MintGuard.setVoucherSigner()

Observation. Setting signer to zero makes future voucher checks revert.

Analysis. This acts as an implicit pause for public minting.

Recommendation. Document it as an operational control.

Code reference: `https://github.com/ltvprotocol/42/blob/codefreeze/src/MintGuard.sol#L154`

1.6 F6 — Sale start before NFT link

Observation. `start()` can be called before `setNft`.

Analysis. Public mint then fails until the NFT address is set. Operational concern, not a security issue.

Recommendation. Document sequence: link NFT before starting the sale.

Code reference: `https://github.com/ltvprotocol/42/blob/codefreeze/src/MintGuard.sol#L122`

1.7 F7 — Admin mint not recorded in `mintAddress`

Observation. Admin recipients are not marked in `mintAddress`.

```
103 function start(address to, uint256 amount) external onlyOwner nonReentrant
    {
104     mintStarted = true;
105     emit MintStarted();
106
107     if (to != address(0) && amount > 0) {
108         require(address(nft) != address(0), ZeroAddress());
109
110         for (uint256 i = 0; i < amount; i++) {
111             uint256 tokenId = nft.mint(to);
112             emit Minted(to, tokenId);
113         }
114     }
115 }
```

Listing 6: MintGuard.start()

Analysis. They can still perform a public mint; affects distribution policy only.

Recommendation. Keep as-is if intended. If not, mark admin recipients as redeemed.

Code reference: https://github.com/ltvprotocol/42/blob/codefreeze/src/MintGuard.sol#L111

1.8 F8 — Empty base metadata URI

```
39 constructor(string memory _baseMetadataUri, address _mintGuard, uint256
       _maxTokens)
40     ERC721("42", "LT42")
41 {
42     baseMetadataUri = _baseMetadataUri;
43     mintGuard = _mintGuard;
44     MAX_TOKENS = _maxTokens;
45 }
```

Listing 7: NFT42 constructor excerpt

Observation. The constructor accepts an empty base URI.

Analysis. This is a business decision: either enforce non-empty URIs at deployment or allow flexibility.

Recommendation. Define a clear policy for `baseMetadataUri`.

Code reference: https://github.com/ltvprotocol/42/blob/codefreeze/src/42.sol#L39

1.9 F9 — Gas limits on large admin batches

```
110 for (uint256 i = 0; i < amount; i++) {
111     uint256 tokenId = nft.mint(to);
112     emit Minted(to, tokenId);
113 }
```

Listing 8: Admin mint loop

Observation. Large `amount` can exceed block gas limits.
Analysis. Admin-only; a failing tx has no side effects.
Recommendation. Batch large reserves into multiple calls.
Code reference: https://github.com/ltvprotocol/42/blob/codefreeze/src/MintGuard.sol#L110

Checklist

The following aspects were explicitly reviewed and confirmed:

- **Architecture:** clear separation of roles. `NFT42` handles token logic and supply cap; `MintGuard` manages sale, vouchers, and fees.
- **Upgradeability:** `MintGuard` uses proxy pattern with initializer functions and `_disableInitializers()` in the constructor. Storage layout is stable. `NFT42` is non-upgradeable.
- **Storage gap:** not required in this design. `MintGuard` is final and not intended for inheritance, and `NFT42` is non-upgradeable.
- **Initialization safety:** Proper use of `initializer`, `__Ownable_init`, and `__ReentrancyGuard_init`; prevents accidental double initialization.
- **Access control:** `onlyOwner` consistently restricts administrative functions; `onlyMintGuard` restricts minting on `NFT42`. No privileged path to bypass caps or fees.
- **Mint/supply constraints:** `MAX_TOKENS` enforced in every mint; `totalSupply` tracked correctly. One-mint-per-address rule applied for vouchers.
- **Events:** Adequate coverage. Semantics noted in F2 (payer vs. recipient).
- **Error handling:** Uses custom errors with clear revert reasons. No silent failures.
- **Payments:** Exact fee checks enforced (`require(msg.value == fee)`). No over/underpayment acceptance.
- **Withdrawals:** ETH withdrawn via low-level `call`, success flag checked, protected by `nonReentrant`. Emits `Withdrawn` event.
- **ETH handling:** `receive()` and `fallback()` enabled to accept ETH. No unintended side effects.
- **Reentrancy:** `nonReentrant` applied on mint, start, setFee, setVoucherSigner, withdraw. ERC721 minting path is safe (OpenZeppelin).
- **Check–Effects–Interactions (CEI):** State updates precede external calls; exception is noted in F4 but safe.
- **Gas usage:** Public functions have bounded loops. Admin batch mint noted in F9 (safe, but large loops may exceed gas).
- **Signature verification:** Vouchers verified with ECDSA over `keccak256(minter)`. Replay context analyzed in F3. Safe under current one-mint-per-address invariant.
- **Telemetry / analytics:** All significant state changes emit events. Admin mint vs. public mint semantics documented.
- **Deployment:** Deployment scripts set up contracts correctly. Proper proxy initialization sequence. No unlinked state.
- **Tests:** Comprehensive suite present: boundary supply, events, fuzzing, invalid input, mismatches, update signer, withdrawal, zero-minter, etc. Covers positive and negative paths.

Final Table of Findings

ID	Observation	Recommendation
F1	`start()` mixes sale toggle and admin mint; policy decides intent.	Document policy; optionally split into `startSale()` and `adminMint()`.
F2	Event logs payer, not recipient, in public mint.	Document semantics or emit payer+recipient in future.
F3	Minimal voucher digest; replay irrelevant in practice.	Accept current; consider EIP-712 for stricter provenance.
F4	`totalSupply` increment precedes cap check.	Safe; reorder for readability if desired.
F5	`setVoucherSigner(0)` effectively pauses minting.	Document as control mechanism.
F6	Sale can start before NFT is linked.	Document operational sequence.
F7	Admin recipients not tracked in `mintAddress`.	Keep or mark per distribution policy.
F8	Constructor allows empty base URI.	Define a policy for URIs.
F9	Admin batch may hit gas limits.	Batch large reserves.

Conclusion

No vulnerabilities were identified. The contracts are secure and production-ready. The key open item is the documented policy for unsold NFTs; all other findings concern semantics, telemetry, or operations and do not impact security. Strong automated tests further support reliability.