Scanned Code Report

AUDITAGENT

Powered by NETHERMIND SECURITY

Code Info

Auditor Scan

Scan ID 1	Date January 21, 2026
Organization omarespejel	Repository starknet-stealth-addresses
Branch main	Commit Hash 16c58a67...64ff252b

Contracts in scope

src/contracts/stealth_account.cairo
src/contracts/stealth_account_factory.cairo
src/contracts/stealth_registry.cairo
src/crypto/constants.cairo
src/crypto/view_tag.cairo
src/errors.cairo
src/lib.cairo
src/types/announcement.cairo
src/types/meta_address.cairo

Code Statistics

Findings	Contracts Scanned	Lines of Code
8	9	1402

Findings Summary

Total Findings

🟥 High Risk (0)
🟧 Medium Risk (0)
🟨 Low Risk (2)
🟦 Info (3)
🟩 Best Practices (3)

Code Summary

This protocol implements a stealth address system for Starknet, enabling private, non-interactive transactions. It enhances user privacy by allowing recipients to receive funds at unique, one-time addresses that are cryptographically un-linkable to their primary on-chain identity. The architecture is composed of three main contracts:

- **StealthRegistry**: This contract serves as a public directory where users register a `StealthMetaAddress`, which contains their public spending and viewing keys. It is also responsible for broadcasting `Announcement` events when a stealth payment is made. These announcements contain the necessary cryptographic information for recipients to discover their funds without revealing any identifying information on-chain.
- **StealthAccountFactory**: A factory contract that deploys `StealthAccount` contracts deterministically using a CREATE2-style mechanism. This allows senders to compute a recipient's stealth address off-chain before it exists on-chain. Recipients only need to deploy their account contract when they are ready to spend the funds, saving on gas fees.
- **StealthAccount**: A SNIP-6 compliant account contract representing a single one-time stealth address. Each account is controlled by a unique spending key that only the intended recipient can derive. It supports native account abstraction features, including compatibility with paymasters, which solves the initial gas funding problem for new, empty stealth accounts.

The typical user flow involves a recipient first registering their meta-address. A sender then fetches this meta-address, generates a one-time stealth address, and sends funds to it. The sender then calls the `announce` function on the registry. The recipient scans these announcements, using an efficient `view_tag` to filter and identify payments intended for them. Once a payment is detected, the recipient can derive the corresponding private key and use the factory to deploy the account to control the funds.

Main Entry Points and Actors

- **`StealthRegistry`**
- `register_stealth_meta_address`: A **User/Recipient** registers their public keys to start receiving stealth payments.
- `update_stealth_meta_address`: A **User/Recipient** updates their registered public keys.
- `announce`: A **Sender** broadcasts a notification after sending funds to a stealth address, allowing the recipient to discover the payment.

- `accept_ownership`: The **Pending Owner** finalizes a two-step ownership transfer of the registry contract.

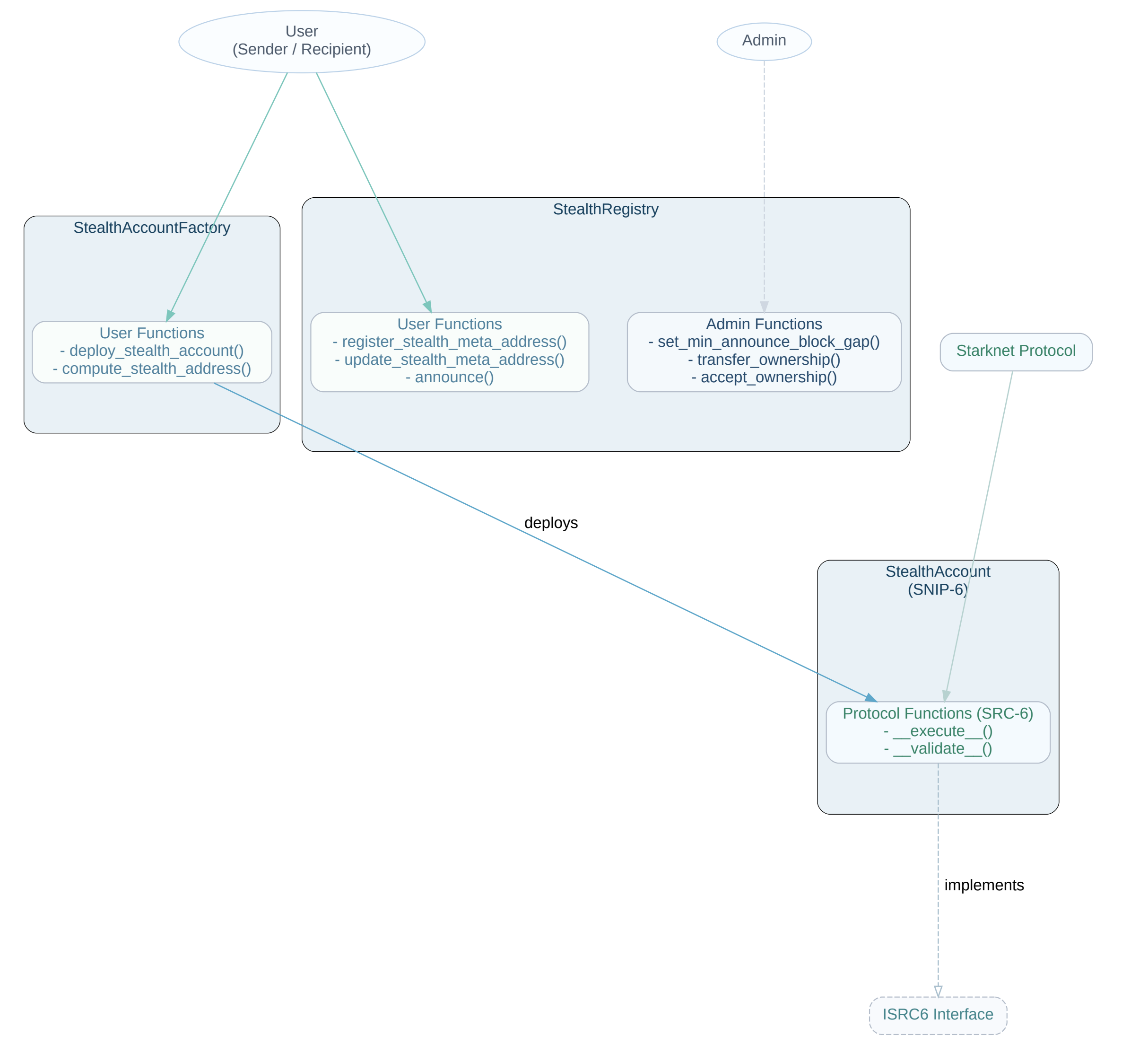
- **`StealthAccountFactory`**

- `deploy_stealth_account`: **Anyone** (typically the recipient) can deploy the account contract for a given stealth address once funds have been sent to it.

- **`StealthAccount`**

- `__execute__`: The **Starknet Protocol** executes transactions initiated and signed by the **Stealth Account Owner** (the recipient).
- `__validate__`: The **Starknet Protocol** validates transactions signed by the **Stealth Account Owner**.

Code Diagram



1 of 8 Findings

src/contracts/stealth_registry.cairo

Announcement event stream can be economically spammed; per-caller rate limiting is sybil-bypassable and announcements are not bound to actual payments

● Low Risk

The `announce()` function is intentionally permissionless and does not attempt to prove that (a) a transfer occurred, (b) the stealth address exists, or (c) the provided (ephemeral key, view_tag, stealth_address) tuple is internally consistent. The only cryptographic constraint is that the ephemeral public key passes `is_valid_public_key()` and the scheme id is in {0,1}:

"fn announce(\n ref self: ContractState,\n scheme_id: u8,\n ephemeral_pubkey_x: felt252,\n ephemeral_pubkey_y: felt252,\n stealth_address: ContractAddress,\n view_tag: u8,\n metadata: felt252\n) {\n assert(\n scheme_id == SchemeId::STARK_CURVE_ECDH\n || scheme_id == SchemeId::STARK_CURVE_DUAL_KEY,\n Errors::INVALID_SCHEME_ID\n);\n\n assert(\n is_valid_public_key(ephemeral_pubkey_x, ephemeral_pubkey_y),\n Errors::INVALID_EPHEMERAL_KEY\n);\n\n ...\n self.emit(Announcement { ... });\n}"

As a result, any attacker willing to pay transaction fees can emit arbitrary announcements that will be indistinguishable at the event level from real payment notifications. While the view-tag mechanism reduces the *cryptographic* work per announcement for a recipient on average, it does not prevent log growth.

Additionally, the implemented rate limit is per caller address (`last_announce_block[caller]`) and can be bypassed by using many sender accounts/addresses. This makes it difficult to enforce a global bound on announcement volume at the contract level.

The harmful impact is primarily on liveness and operational cost: scanners/indexers/RPC consumers may face increasing bandwidth and processing overhead, and recipients may experience degraded scanning performance or increased reliance on third-party indexing infrastructure as the announcement stream grows under adversarial load.

2 of 8 Findings src/contracts/stealth_account_factory.cairo

Factory compute_stealth_address omits Starknet address normalization, so it can disagree with deploy_syscall and/or revert on valid deployments

● Low Risk

The factory aims to let users pre-compute the exact address that `deploy_syscall(...)` will deploy to. However, `compute_stealth_address(...)` computes the Pedersen hash chain and then directly converts the final hash into a `ContractAddress`:

```
let a4 = pedersen(a3, constructor_calldata_hash);
let final_hash = pedersen(a4, 5);  // 5 = number of elements

// Convert to address
final_hash.try_into().expect(Errors::ADDRESS_MISMATCH)
```

In Starknet, the contract address derivation includes an additional normalization step that reduces the raw hash into the L2 address range (commonly described as modulo an upper bound such as `2^251 - 256`). (docs.rs) This normalization is part of the canonical contract address calculation used by the system when deploying. (docs.rs)

Because `compute_stealth_address(...)` does not apply the same normalization, there are raw hash outputs for which:

1) `compute_stealth_address(...)` can **revert** (because `final_hash.try_into()` enforces the `ContractAddress` range), while an actual deployment would still succeed after normalization in the canonical formula, or
2) `compute_stealth_address(...)` can **return an address that differs** from the normalized address actually used by `deploy_syscall`, making the “pre-computed stealth address” incorrect.

Even though these cases are rare, they are correctness-critical in a stealth-address system: a sender can fund an address derived from `compute_stealth_address(...)`, while the recipient later deploys the account at the address derived by the canonical deployment formula, resulting in funds being held by an address that is not controlled by the intended stealth account.

This breaks the intended determinism guarantee and can violate the invariant that the factory’s computed address must match the deployed address for the same `(pubkey_x, pubkey_y, salt)` tuple.

Severity Note:
- Canonical deploy_syscall uses the standard Starknet contract-address derivation and only differs from a raw hash when enforcing/normalizing to the address domain.
- If the computed hash is already within the allowed address range, normalization is identity (no disagreement).
- Pedersen outputs are close to uniformly distributed over the field, making out-of-range hits negligibly likely.

3 of 8 Findings   src/contracts/stealth_registry.cairo

Unexpected Ownership Assignment in StealthRegistry Constructor   ● Info

The constructor of the `StealthRegistry` contract initializes the `owner` using `get_execution_info().unbox().tx_info.unbox().account_contract_address`. This sets the owner to the address of the account contract that pays the transaction fees for the deployment.

This behavior deviates from the common and safer practice of accepting an `owner` address as a constructor argument or using `get_caller_address()`. In complex deployment scenarios, such as those involving deployer contracts, multisigs, or DAOs, the account paying the fee might not be the intended owner of the contract.

For instance, if a DAO contract executes a proposal to deploy the registry, but a specific DAO member's account is used to pay the gas fee, that member's account will become the owner, not the DAO contract. This can lead to the registry being deployed with an incorrect owner, resulting in a permanent loss of administrative control. The owner has the ability to change the announcement rate-limiting policy via `set_min_announce_block_gap`, and losing this ability could impact the protocol's ability to respond to Denial-of-Service or spam attacks.

Code snippet:

```
// File: src/contracts/stealth_registry.cairo

#[constructor]
fn constructor(ref self: ContractState) {
    let exec_info = get_execution_info();
    let tx_info = exec_info.unbox().tx_info.unbox();
    self.owner.write(tx_info.account_contract_address); // Owner is the fee-paying account
    self.pending_owner.write(zero_address());
    self.version.write(1);
    self.announcement_count.write(0);
    self.min_announce_block_gap.write(DEFAULT_MIN_ANNOUNCE_BLOCK_GAP);
}
```

4 of 8 Findings

src/contracts/stealth_registry.cairo src/contracts/stealth_account_factory.cairo

src/contracts/stealth_account.cairo

Contracts lack upgrade mechanisms for post-deployment fixes

● Info

None of the three main contracts (StealthRegistry, StealthAccountFactory, StealthAccount) implement upgrade mechanisms. If a critical vulnerability is discovered post-deployment, there is no way to patch the deployed contracts.

For the StealthAccount, this may be intentional as stealth accounts are one-time-use addresses. However, for the StealthRegistry and StealthAccountFactory which are shared infrastructure contracts, the inability to upgrade could be problematic. Users would need to migrate to new contract deployments, requiring re-registration of meta-addresses and potential loss of announcement history linkage.

This is a design trade-off between immutability/trust and operational flexibility. The lack of upgrade capability does provide stronger guarantees that the protocol logic cannot be changed post-deployment.

5 of 8 Findings

src/contracts/stealth_registry.cairo

Documentation Claims 'No Admin Roles' But Implementation Has Owner With Censorship Powers

● Info

ROOT CAUSE: Documentation claims “no admin roles / permissionless protocol” and “rate limiting disabled by default”, but StealthRegistry implements an owner role (set in constructor to tx_info.account_contract_address) with privileged admin functions and DEFAULT_MIN_ANNOUNCE_BLOCK_GAP=5 (enabled by default). The owner can set min_announce_block_gap to any u64 via set_min_announce_block_gap with immediate effect and no cap.

EXPLOIT PATH: (1) Owner calls set_min_announce_block_gap with a very high value (e.g., 2^64-1). (2) announce reads the current min_gap and applies it against the stored last_announce_block for each caller; since last_announce_block persists across min_gap changes, raising min_gap retroactively rate-limits all callers who have ever announced. (3) This can immediately and effectively DoS announcements for those callers for an extremely long period, degrading protocol availability and payment discovery.

BYPASS ANALYSIS: New caller addresses bypass on first announce (last==0), and relayers can distribute load across multiple relayer addresses, but both impose cost/complexity and can undermine privacy goals by introducing linkable operational patterns.

IMPACT: Trust-model violation and centralization risk: owner can unilaterally and instantly degrade/interrupt announcement availability; default rate limiting is enabled contrary to documentation; unbounded min_gap allows extreme or near-permanent DoS for repeat callers.

MITIGATIONS: Align documentation with implementation and/or remove owner/admin functions for true permissionlessness; otherwise set DEFAULT_MIN_ANNOUNCE_BLOCK_GAP=0 if “disabled by default” is intended, add a strict MAX_ANNOUNCE_BLOCK_GAP cap, and consider governance safeguards (multisig/timelock) and/or non-retroactive application (apply new min_gap only to future announces).

6 of 8 Findings src/contracts/stealth_registry.cairo

Transfer ownership can be initiated to zero address

● Best Practices

The `transfer_ownership` function does not validate that `new_owner` is non-zero:

```
fn transfer_ownership(ref self: ContractState, new_owner: ContractAddress) {
    let caller = get_caller_address();
    let owner = self.owner.read();
    assert(caller == owner, Errors::UNAUTHORIZED);

    self.pending_owner.write(new_owner);  // Can be zero address

    self.emit(OwnershipTransferStarted {
        previous_owner: owner,
        new_owner,
    });
}
```

While the `accept_ownership` function correctly checks `assert(pending != zero_address(), Errors::NO_PENDING_OWNER)`, preventing the transfer from completing, initiating a transfer to zero address wastes gas and emits a misleading `OwnershipTransferStarted` event. This is a minor usability issue rather than a security vulnerability, as the actual invariant that owner cannot be zero is maintained.

7 of 8 Findings src/contracts/stealth_registry.cairo

No mechanism for owner to cancel pending ownership transfer

● Best Practices

Once an ownership transfer is initiated via `transfer_ownership`, there is no explicit function for the current owner to cancel it. The only way to effectively cancel is to initiate a new transfer to a different address:

```
fn transfer_ownership(ref self: ContractState, new_owner: ContractAddress) {
    let caller = get_caller_address();
    let owner = self.owner.read();
    assert(caller == owner, Errors::UNAUTHORIZED);
    self.pending_owner.write(new_owner);
    // ...
}
```

This means if an owner mistakenly initiates a transfer to an address they don't control, they must start a new transfer to cancel the pending one. While this works, a dedicated `cancel_ownership_transfer` function would provide clearer intent and better UX. This is a design consideration rather than a security vulnerability.

8 of 8 Findings src/contracts/stealth_account_factory.cairo

ClassHashUpdated event is defined but never used ● Best Practices

The `StealthAccountFactory` contract defines a `ClassHashUpdated` event but there is no function that emits this event or allows updating the `stealth_account_class_hash` storage variable after construction.

```
/// Emitted when the account class hash is updated
#[derive(Drop, starknet::Event)]
struct ClassHashUpdated {
    old_class_hash: ClassHash,
    new_class_hash: ClassHash,
}
```

The `stealth_account_class_hash` is only set in the constructor and cannot be modified thereafter. While this immutability may be intentional for a permissionless protocol, the presence of the `ClassHashUpdated` event is misleading as it suggests upgrade capability that does not exist. This constitutes dead code that should either be removed or the corresponding update functionality should be implemented if upgradability is desired. If a vulnerability is discovered in the `StealthAccount` implementation, a new factory would need to be deployed rather than updating the existing one.