# **TURBIN3 Q4 CAPSTONE**

# **Name- Saatvik Madan**

# **Address- 9ARoNxwJa2x8KQrcH4Cut2V89cz8nhsMkcegNvUFq9wy**

## **TapShield - Cross-Faucet Abuse Prevention**

# **Part A: Final User Stories & On-Chain Requirements Document**

## **1) Core User Personas (Prioritized for POC)**

## **Primary Users:**

**1. Faucet Operator**

* **Who:** Developers/organizations running Solana testnet/devnet faucets
* **Why Critical for POC:** They are both the integrators and primary beneficiaries. Without faucet operators adopting **TapShield**, there is no network to coordinate.
* **POC Priority:** Essential - must demonstrate value immediately

**2. Integration Developer**

* **Who:** Technical team members implementing **TapShield** into existing faucet codebases
* **Why Critical for POC:** Ease of integration determines adoption success. If integration is painful, operators won't use it.
* **POC Priority:** Essential - SDK must be frictionless

**3. Claiming User (End User)**

* **Who:** Developers requesting testnet tokens from faucets
* **Why Critical for POC:** Their behavior (legitimate vs. abusive) is what the system tracks. User experience must not degrade.
* **POC Priority:** Important - must prove system doesn't harm legitimate users

## **Secondary Users (Post-POC):**

**4. Registry Administrator**

* **Who:** Maintainer of **TapShield** protocol (initially you, potentially transitioning to DAO)
* **Why Deferred:** POC can function without governance layer initially
* **POC Priority:** Low - can be hardcoded/permissionless for v1

**5. Ecosystem Observer**

* **Who:** Researchers, analysts, or third parties viewing claim statistics
* **Why Deferred:** Analytics are value-add, not core functionality
* **POC Priority:** Low - read-only views can be added post-POC

## **2) Function Map - User to Core Actions**

## **Faucet Operator Functions:**

|  |  |  |
| --- | --- | --- |
| **Function ID** | **Function Description** | **User Goal** |
| FO-1 | Register faucet in TapShield registry | Establish faucet identity on-chain |
| FO-2 | Check if wallet has claimed recently | Verify eligibility before distribution |
| FO-3 | Record completed claim to registry | Update network after successful distribution |
| FO-4 | View faucet's total claim statistics | Monitor own faucet usage |

## **Integration Developer Functions:**

|  |  |  |
| --- | --- | --- |
| **Function ID** | **Function Description** | **User Goal** |
| ID-1 | Install TapShield TypeScript SDK | Add dependency to project |
| ID-2 | Initialize TapShield client | Connect to on-chain registry |
| ID-3 | Call eligibility check function | Integrate verification into claim flow |
| ID-4 | Call record claim function | Integrate post-distribution logging |
| ID-5 | Handle error responses | Gracefully manage failures |

## **Claiming User Functions:**

|  |  |  |
| --- | --- | --- |
| **Function ID** | **Function Description** | **User Goal** |
| CU-1 | Request tokens from faucet | Obtain testnet SOL/tokens |
| CU-2 | See eligibility status | Understand if/why claim was rejected |
| CU-3 | Wait for cooldown period | Know when they can claim again |

## **3) Core POC User Stories**

## **Epic 1: Faucet Registration**

**US-1:** As a faucet operator, I want to register my faucet in TapShield, so that I can participate in the abuse prevention network.

**US-2:** As a faucet operator, I want my faucet registration to include my faucet name and admin wallet, so that my faucet is identifiable on-chain.

**US-3:** As a faucet operator, I want registration to be permissionless, so that I don't need approval to join the network.

## **Epic 2: Claim Eligibility Verification**

**US-4:** As a faucet operator, I want to check if a wallet address has claimed recently from ANY participating faucet, so that I can prevent multi-faucet abuse.

**US-5:** As a faucet operator, I want to set my own cooldown period, so that I can control claim frequency for my faucet.

**US-6:** As a faucet operator, I want the eligibility check to return clear yes/no status, so that I can easily integrate it into my claim logic.

**US-7:** As a faucet operator, I want the eligibility check to include timestamp of last claim, so that I can show users when they can claim again.

## **Epic 3: Claim Recording**

**US-8:** As a faucet operator, I want to record a claim after I successfully distribute tokens, so that other faucets know this wallet claimed recently.

**US-9:** As a faucet operator, I want the claim record to include the wallet address, my faucet ID, timestamp, and amount distributed, so that the record is complete and verifiable.

**US-10:** As a faucet operator, I want claim recording to be required for participation, so that all network members contribute data fairly.

## **Epic 4: Developer Integration**

**US-11:** As an integration developer, I want to install TapShield SDK via npm, so that I can quickly add it to my project.

**US-12:** As an integration developer, I want clear documentation with code examples, so that I can integrate without deep blockchain knowledge.

**US-13:** As an integration developer, I want initialization to require only my faucet's keypair and RPC URL, so that setup is simple.

**US-14:** As an integration developer, I want all functions to return typed results with error handling, so that I can build robust integrations.

## **Epic 5: End User Experience**

**US-15:** As a claiming user, I want to receive a clear message when rejected, so that I understand why I can't claim.

**US-16:** As a claiming user, I want to see which faucet I last claimed from, so that I know where my cooldown originated.

**US-17:** As a claiming user, I want the process to feel no different from before **TapShield**, so that legitimate usage isn't disrupted.

## **4) Potential On-Chain Requirements**

## **For US-1, US-2, US-3: Faucet Registration**

**On-Chain Requirements:**

* Need an on-chain account (PDA) to store faucet metadata
* The PDA must be derived from faucet operator's wallet address for uniqueness
* Account must store: faucet name (String), operator address (Pubkey), registration timestamp (i64), total claims recorded (u64)
* Need an initialize\_faucet instruction that anyone can call (permissionless)
* The instruction must validate that the signer is the operator
* Must handle re-initialization attempts gracefully (either error or allow updates)
* Account must pay rent-exemption at creation

## **For US-4, US-5, US-6, US-7: Eligibility Verification**

**On-Chain Requirements:**

* Need a check\_eligibility instruction that queries claim history
* Must accept: wallet address to check, cooldown period duration
* Must search through claim records for matching wallet address
* Must compare timestamps to determine if cooldown has passed
* Must return: boolean eligible status, timestamp of last claim (if exists), faucet ID of last claim
* Function must be read-only (view function) with no state changes
* Must handle case where wallet has never claimed (return eligible=true)
* Must be gas-efficient since called frequently

## **For US-8, US-9, US-10: Claim Recording**

**On-Chain Requirements:**

* Need a record\_claim instruction to create claim records
* Must create a new account (PDA) for each claim
* The claim PDA must be derived from: claimer wallet + faucet ID + timestamp (for uniqueness)
* Claim account must store: claimer address (Pubkey), faucet ID (Pubkey), timestamp (i64), amount distributed (u64)
* Instruction must validate that signer is a registered faucet operator
* Must emit an event/log for off-chain indexing
* Claim accounts must be rent-exempt
* Need to handle storage limits - potentially archive old claims or implement pruning
* Must prevent duplicate claims within same second (PDA derivation handles this)

## **For US-11, US-12, US-13, US-14: SDK Integration**

**On-Chain Requirements:**

* SDK needs to interact with program ID (static address after deployment)
* SDK must construct instruction data correctly for all three instructions
* SDK must derive PDAs correctly (faucet PDA, claim PDAs)
* SDK needs to handle transaction signing and sending
* SDK must parse return data from check\_eligibility view function
* SDK needs proper TypeScript types for all inputs/outputs
* Must handle common errors: insufficient rent, invalid signer, PDA already exists
* SDK should provide helper to calculate cooldown expiry times client-side

## **For US-15, US-16, US-17: User Experience**

**On-Chain Requirements:**

* check\_eligibility must return detailed error messages (not just fail)
* Error types needed: ClaimTooRecent, CooldownNotElapsed, InvalidFaucet
* Successful claim record must emit event with all claim details
* Faucet UIs must be able to query last claim details from on-chain
* Need to ensure transaction times don't add noticeable latency
* Errors must include both error code and human-readable message

## **5) Core On-Chain Architecture Overview**

## **Program Accounts:**

1. **FaucetRegistry Account (PDA)**
   1. Seeds: [b"faucet", operator\_pubkey]
   2. Stores: name, operator, timestamp, claim\_count
   3. Mutable by operator only
2. **ClaimRecord Account (PDA)**
   1. Seeds: [b"claim", claimer\_pubkey, faucet\_pubkey, timestamp]
   2. Stores: claimer, faucet, timestamp, amount
   3. Immutable after creation

## **Program Instructions:**

1. **initialize\_faucet** - Create faucet registry entry
2. **check\_eligibility** - View function to verify claim eligibility
3. **record\_claim** - Create new claim record

## **Key Program Logic:**

* All PDAs for deterministic address derivation
* Rent-exempt accounts to avoid reclaim issues
* Event emission for off-chain indexing
* View functions for gas-efficient reads
* Proper access control (only registered faucets can record claims)

# **Part B: Process Appendix**

## **Part A: Initial User & Function Mapping**

## **Step 1: Manual User Brainstorming**

**Brainstorming Process:**  
 Starting from **TapShield's** value proposition (cross-faucet abuse coordination), I identified all potential touchpoints:

**Initial Comprehensive User List:**

1. Faucet operators (running the service)
2. Integration developers (implementing TapShield)
3. Claiming users (requesting tokens)
4. Protocol administrators (managing TapShield)
5. Malicious actors (trying to abuse system)
6. Ecosystem observers (viewing statistics)
7. Wallet providers (potentially integrating checks)
8. Solana Foundation (potential oversight)
9. Hackathon organizers (bulk distribution use case)
10. Token project teams (promotional faucets)
11. Educational platform admins (student distributions)
12. Bot operators (automated claiming)
13. Researchers (studying abuse patterns)
14. Future DAO members (governance)

**Rationale:** Cast wide net initially - better to trim than miss critical users.

## **Step 2: AI-Assisted User Prioritization**

**AI Prompt Used:**

"My project's value proposition is: **TapShield** is a decentralized on-chain registry that enables Solana faucet operators to coordinate abuse prevention by tracking wallet claim history across multiple faucets. Here is a brainstormed list of all potential user types:

1. Faucet operators
2. Integration developers
3. Claiming users
4. Protocol administrators
5. Malicious actors
6. Ecosystem observers
7. Wallet providers
8. Solana Foundation
9. Hackathon organizers
10. Token project teams
11. Educational platform admins
12. Bot operators
13. Researchers
14. Future DAO members

Based on the value proposition, which 2-5 of these user types are the most critical to focus on for an initial Proof-of-Concept? For each user you recommend, provide a brief rationale explaining why they are essential for proving the project's core value."

**AI Response:**

"For your POC, focus on these 3 critical user types:

1. **Faucet Operators** - They are the adopters. Without them, there's no network. POC must prove value to them immediately.
2. **Integration Developers** - They execute the adoption. If integration is complex, operators won't use it. POC must demonstrate ease of integration.
3. **Claiming Users** - They are the reason the system exists. POC must show the system works without disrupting legitimate users.

**Defer:**

* Administrators (can be permissionless initially)
* Observers (analytics are nice-to-have)
* Hackathon organizers (specific use case, test after core POC)
* Wallet providers (future integration, not POC-critical)"

**My Analysis & Final Decision:**

**Agreed with AI:** Faucet operators, integration developers, and claiming users are indeed the MVP trio.

**My Additions:** None - the AI correctly identified the minimal viable user set.

**Final Prioritized List:**

1. Faucet Operator (PRIMARY)
2. Integration Developer (PRIMARY)
3. Claiming User (PRIMARY)
4. Registry Administrator (SECONDARY - deferred)
5. Ecosystem Observer (SECONDARY - deferred)

## **Step 3: Core Function Mapping**

**AI Prompt Used:**

"For a project with this value proposition [**TapShield** cross-faucet coordination] and focusing on these prioritized user types [Faucet Operator, Integration Developer, Claiming User], help map out the key functions or interactions each user would need to perform."

**AI Response:**

"**Faucet Operator Functions:**

* Register their faucet in the system
* Check if a wallet is eligible to claim
* Record when a claim is completed
* View their faucet's statistics

**Integration Developer Functions:**

* Install the SDK/library
* Initialize connection to **TapShield**
* Call eligibility check in their code
* Call record claim in their code
* Handle errors gracefully

**Claiming User Functions:**

* Request tokens from a faucet
* See why they were rejected (if applicable)
* Understand cooldown periods"

**My Manual Additions:**

Added function ID system for tracking (FO-1, ID-1, CU-1, etc.)

Added "Set custom cooldown period" for faucet operators (flexibility needed)

Added "View which faucet blocked me" for claiming users (transparency)

**Final Function Map:** (See Part A, Section 2 above)

## **Step 4: Deriving Core POC Requirements**

**Critical Interaction Path Identified:**

1. Faucet operator registers faucet
2. User requests tokens
3. Faucet checks eligibility via **TapShield**
4. If eligible, faucet distributes tokens
5. Faucet records claim to **TapShield**
6. Next faucet user visits can see this claim

**AI Prompt Used:**

"Based on this critical user interaction path [described above], what are the key technical requirements needed to build a proof-of-concept for **TapShield**?"

**AI Response:**

"Key technical requirements:

1. **Smart Contract (Anchor Program):**
   1. Faucet registration storage
   2. Claim record storage
   3. Eligibility verification logic
2. **SDK (TypeScript):**
   1. Functions to call each program instruction
   2. PDA derivation helpers
   3. Error handling
3. **On-Chain State:**
   1. Faucet metadata (name, operator, stats)
   2. Individual claim records (who, when, from which faucet)
4. **Key Functions:**
   1. registerFaucet()
   2. checkEligibility(wallet, cooldownPeriod)
   3. recordClaim(wallet, amount)"

**My Analysis:**

AI correctly identified core components. I added:

* Specific data fields for each account
* PDA derivation schemes
* Rent-exemption considerations
* Event emission for indexing

**Final Initial Requirements:** (Documented in Part A, Section 4)

## **Part B: Adversarial Analysis & Granularity Check**

## **Critique & Refine User Stories**

**AI Prompt Used:**

"Review my core user functions/stories [Epic 1-5 from Part A] and requirements [on-chain requirements from Part A]. Considering my project's refined value proposition [**TapShield** cross-faucet coordination], do these stories truly hit the mark? Are the requirements granular enough to map to specific technical components (e.g., Anchor account structs, instruction handlers, SDK methods)? What's missing or unclear?"

**AI Critique Response:**

"**What's Good:**

* User stories are action-oriented and clear
* Requirements identify key data structures
* Separation of faucet operator vs. developer is smart

**Gaps & Issues:**

1. **US-4 is too broad:** 'Check if wallet claimed from ANY faucet' - how does this work technically? Need to specify: does the program iterate through all claim records? That doesn't scale. Missing: indexing strategy or account structure that makes this lookup efficient.
2. **US-10 assumes enforcement:** 'Recording must be required' - how? The program can't force operators to call record\_claim. Need to clarify: is this social contract or technical enforcement?
3. **Missing error handling stories:** What if registration fails? What if claim recording fails mid-distribution? Need user stories for failure cases.
4. **Unclear: Who pays gas?** Every user story that writes on-chain has a cost. Who signs? Who pays? This isn't specified.
5. **Granularity issue in US-2:** 'Register my faucet including name and admin wallet' - this is actually ONE action (initialize\_faucet instruction), so granularity is fine. But need to clarify: can operator update these fields later?
6. **Requirements don't specify access control:** Who can call record\_claim? Only the registered faucet? Need explicit access control requirements.
7. **US-15, US-16, US-17 are UI/UX, not on-chain:** These belong in frontend scope, not smart contract requirements. Either remove or clarify they're SDK/integration requirements, not program requirements."

**My Analysis of AI Critique:**

**Valid Points:**

1. ✅ **Scalability of eligibility check** - AI is right, iterating all claims doesn't scale. Solution: Use PDA derivation with claimer address as seed, enabling direct lookup.
2. ✅ **Enforcement of recording** - Valid concern. Refinement: It's a social contract (opt-in network). Add user story about reputation/visibility.
3. ✅ **Error handling gaps** - Need failure case stories.
4. ✅ **Gas payment clarity** - Must specify: