ADR-015: Blind Ballot Token Anonymity

Submitters

- Luke Doyle (D00255656)
- Hafsa Moin (D00256764)

Change Log

- approved 2026-02-16

Referenced Use Case(s)

- CVP-ADR-015: Blind Ballot Token Anonymity

Context

An election system must satisfy two requirements that are in direct tension. Accountability requires proof that each voter voted at most once, which means linking a voter's identity to the act of voting. Anonymity requires that nobody, including the server operator, can determine how any specific voter voted, which means severing any link between a voter's identity and their ballot choice. The system must resolve this tension through architectural design rather than policy.

The project research paper identifies three categories of identity fraud, and the anonymity mechanism must protect against the most serious of these: insider abuse. Even an administrator with full database access should not be able to determine which candidate any specific voter chose.

Traditional online voting systems often fail this test. Systems that store `voter_id` alongside the vote choice provide no anonymity at all. Systems that use session tokens linked to voter accounts can be correlated through timing analysis. The goal is to create a cryptographic gap between identity verification and ballot casting that cannot be bridged after the fact.

Proposed Design

The blind ballot token protocol separates the identity verification process from the ballot casting process using a cryptographic intermediary. The protocol runs in three phases.

Phase 1: Identity verification (identity-linked)

The voter clicks their email link, which carries a `voting_token`. The auth-service validates the token, which is linked to the voter's record, and the voter completes identity verification via date-of-birth MFA.

Phase 2: The anonymity bridge

The auth-service marks `voter.has_voted = TRUE` and marks the voting token as used. It then generates a fresh `ballot_token` using `secrets.token_urlsafe(32)` — pure cryptographic randomness with no derivation from any voter attribute — and stores it in the `blind_tokens` table with only an `election_id`. No record is written linking the voter to this ballot token. The ballot token is returned to the voter's browser.

After this transaction commits, the link between voter and ballot is permanently severed. The auth-service knows that voter X received a ballot token, but not which one. The voting-service will know that ballot token Y was used to cast a vote, but not which voter held it.

Phase 3: Anonymous ballot casting (identity-separated)

The voter submits their vote using the ballot token only — no `voting_token`, no `voter_id`. The voting-service validates the ballot token, encrypts the vote choice using `pgp_sym_encrypt` before storage, writes the encrypted ballot with no `voter_id` and no reference to the ballot token, and returns a `receipt_token` for verification.

Schema enforcing anonymity

The anonymity is enforced at the schema level. The columns do not exist, so no application bug can accidentally store the voter-ballot link.

```sql
-- blind_tokens: NO voter_id, NO voting_token reference
CREATE TABLE blind_tokens (
    id           SERIAL PRIMARY KEY,
    ballot_token VARCHAR(255) UNIQUE NOT NULL,
    election_id  INTEGER REFERENCES elections(id),
    is_used      BOOLEAN DEFAULT FALSE,
    issued_at    TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    used_at      TIMESTAMP
    -- NO voter_id
);

-- encrypted_ballots: NO voter_id, NO user_id, NO ballot_token_id
CREATE TABLE encrypted_ballots (
    id             SERIAL PRIMARY KEY,
    election_id    INTEGER REFERENCES elections(id),
    encrypted_vote BYTEA NOT NULL,        -- pgp_sym_encrypt
    ballot_hash    VARCHAR(255) NOT NULL, -- SHA-256 auto-generated
    previous_hash  VARCHAR(255),          -- hash chain
    receipt_token  VARCHAR(255) UNIQUE NOT NULL,
    cast_at        TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    -- NO voter_id, NO user_id, NO ballot_token_id
);

```

Cryptographic parameters

Ballot tokens are 256 bits of entropy (`secrets.token_urlsafe(32)`). Receipt tokens are 192 bits (`secrets.token_urlsafe(24)`). Vote encryption uses the `pgp_sym_encrypt` function from the PostgreSQL `pgcrypto` extension. The encryption key is per-election, generated with `secrets.token_urlsafe(32)`, and is stored in the `elections` table for use during tallying. Ballot immutability is enforced by a `BEFORE UPDATE/DELETE` trigger that raises an exception.

The receipt token allows a voter to confirm their ballot was recorded. It contains a `ballot_hash` but not the vote choice, providing verifiability without revealing the vote.

The anonymity bridge code in the auth-service is documented extensively in `auth-service/app.py` (lines 246-342) and should be reviewed carefully during any changes to ballot token issuance logic.

Considerations

Option 1: Voter-linked ballots

Store `voter_id` as a foreign key in the votes table alongside the candidate choice. This was rejected outright as it provides no anonymity. Anyone with database read access can see every voter's choice. It is not a credible option for any election system.

Option 2: Blind ballot tokens (chosen)

The three-phase protocol described in the Proposed Design section. The main limitation is that the anonymity guarantee is architectural rather than mathematical: it depends on the auth-service code not logging the voter-to-ballot mapping. A compromised auth-service running modified code at the exact moment of token issuance could record that mapping. This is mitigated by code review, deployment automation that prevents ad-hoc code changes, and audit logging of the issuance event.

Option 3: Homomorphic encryption

Encrypt votes such that tallying can be performed on ciphertext without decrypting individual records. This is the academic gold standard but was rejected as impractical. There are no mature Python libraries for practical homomorphic encryption, operations on encrypted data carry extreme performance overhead, and implementation would consume the entire project timeline. It also fails the requirement that the implementation be auditable without deep cryptographic expertise.

Option 4: Mixnets (mix networks)

Route votes through multiple independent servers that shuffle and re-encrypt them, making it impossible to trace which input corresponds to which output. This is used in some national e-voting systems but requires multiple independent servers with distributed trust, complex re-encryption and zero-knowledge proof protocols, and significant additional infrastructure. It is designed for national-scale elections with millions of voters and multiple election authorities, which makes it entirely disproportionate for a student council election. It also fails the constraint against external infrastructure.

Future upgrade path

A subsequent iteration could upgrade to RSA blind signatures as described by Chaum (1983), where the voter blinds a nonce, the server signs it, and the voter unblinds to obtain a signature the server has never seen in the clear. This would provide mathematical rather than purely architectural anonymity. This is scheduled for evaluation at the end of Stage 2 (April 2026).

Trade-off: trust in code vs trust in mathematics

The current anonymity guarantee depends on the code not recording the voter-to-ballot mapping. This is a weaker guarantee than what homomorphic encryption or mixnets would provide, but it is auditable, implementable within the project constraints, and practical. The mechanism is approximately 50 lines of Python rather than a black-box cryptographic library.

Decision

Blind ballot tokens (Option 2) were chosen. The protocol achieves strong anonymity without external dependencies and within the constraints of a two-semester project. Even a database administrator with full SELECT access across all tables cannot link a voter to their ballot choice, because the columns required to make that link do not exist in the schema.

Accountability is maintained separately through the `voters.has_voted` flag, which records that a voter participated without recording how they voted. Double-voting is prevented by the combination of this flag and the single-use ballot token.

Any future changes to the ballot token issuance logic in the auth-service must be reviewed against the anonymity contract documented in the code comments. The issuance transaction must not, under any circumstances, write a record linking `voter_id` to `ballot_token`.

Other Related ADRs

- ADR-002: PostgreSQL Database - The `pgcrypto` extension used for `pgp_sym_encrypt` is a PostgreSQL feature
- ADR-005: Token-Based Voter Authentication - The voting token initiates Phase 1 of this protocol
- ADR-007: SHA-256 Hash-Chain Audit Logs - Audit logs record ballot issuance events without linking voter to ballot

- ADR-013: Domain-Driven Service Separation - The separation of auth-service (identity) and voting-service (ballots) is what makes this protocol structurally sound
- ADR-014: Per-Service Database Users - The voting-service database user has no access to the `voters` table, enforcing the separation at the permission level

References

- Chaum, D. (1983). "Blind Signatures for Untraceable Payments." Advances in Cryptology.
- PostgreSQL pgcrypto Documentation
- Python secrets module
- Investigation Log, section 5.3: Full analysis of anonymity mechanisms