

LAMPORT: A Self-Protecting Token on Solana

The LAMPORT contributors · lamport.computer

Abstract

A Solana account is controlled by a single ed25519 key: whoever holds it, whether through phishing, malware, a leaked seed phrase or a future cryptanalytic break, can move everything in it. We present \$LAMPORT, a Token-2022 token whose transfer hook refuses any transfer from an *armed* account unless the same transaction carries proof from a second, hash-based key. The everyday proof is a link of a Lamport hash chain, revealed through a commit–reveal step that defeats front-running; rare administrative actions use a stateful hash-based signature (WOTS/XMSS). Holders opt in, pools and unarmed holders are unaffected, and a seven-day recovery that the second key can veto handles lost keys. We describe the protocol, its accounts and instructions, a security argument, and its honest limits.

1 Introduction

Every signature scheme in production use on Solana today rests on the hardness of the elliptic-curve discrete logarithm problem. That assumption is challenged from three directions at once: everyday key theft (phished seed phrases, malicious approvals, wallet drainers), the possibility of a new classical algorithm, and Shor’s algorithm on a large quantum computer [5]. In every case the outcome is the same: an adversary holds a valid signing key and the chain cannot tell them from the owner.

Hash functions offer a different footing. In 1979 Lamport showed how to build digital signatures from any one-way function [1]; in 1981 he proposed authenticating with a chain of hashed passwords, each used once [7]. Both need nothing but preimage resistance, which quantum computers weaken only quadratically [6]. Their limitation is statefulness: each key may be used once, or a fixed number of times.

Token-2022’s `TransferHook` extension lets a mint name a program that Solana invokes on *every* transfer of the token, including transfers initiated by delegates and DEX programs, and whose failure aborts the whole transaction [9]. We use that hook to enforce a second factor at the token level: once a holder *arms* their token account, the hook rejects any outgoing transfer that is not accompanied, in the same transaction, by a fresh link of the holder’s hash chain. A stolen wallet key, of any provenance, is no longer sufficient.

Our contributions are:

1. a commit–reveal protocol that turns a Lamport hash chain into a per-transfer, front-running-resistant proof that the hook can check with one hash (§4);
2. a stateful XMSS-style Master Key for rare actions, staged across transactions to fit Solana’s limits (§5);
3. a time-locked recovery that the second key can veto (§6); and
4. a security argument and an explicit list of limitations (§7–§8).

2 Threat model

Adversary. The adversary $\mathcal{A}$ knows the owner’s ed25519 secret key and can therefore sign any transaction the owner could. $\mathcal{A}$ sees all on-chain state and all transactions as they are broadcast, and can submit its own transactions with arbitrary priority fees. $\mathcal{A}$ does not know the Quick Key secret x_0 or the Master Key seed, which are generated independently of the wallet seed and stored apart from it.

Assumptions. H is SHA-256, modelled as preimage- and second-preimage-resistant; against quantum adversaries we assume the generic Grover bound, 2^{128} work for a preimage. Solana’s consensus and the Token-2022 program behave as specified. The owner’s reveal transaction is included within $\Delta = 4$ slots of being broadcast (see §8 for what happens if it is censored for longer).

Goals. (G1) *Unstealability*: $\mathcal{A}$ cannot reduce the balance of an armed token account by any transfer. (G2) *Liveness*: the owner can always transfer. (G3) *Recoverability*: an owner who loses the second key can disarm with the wallet key alone, after a delay during which the second key can veto.

Out of scope. Burns (which do not invoke transfer hooks), assets other than \$LAMPORT, compromise of the device while it holds both keys in the clear, and implementation bugs (addressed by audit and immutability).

3 Design

\$LAMPORT is a Token-2022 mint with the `TransferHook` (program `lamport_hook`), `MetadataPointer` and `TokenMetadata` extensions, six decimals and a fixed supply. After launch, the mint authority, freeze authority and transfer-hook authority are set to none, there is no permanent delegate and no transfer fee, and both programs are made immutable after audit, so no single key, including the authors’, can change the rules.

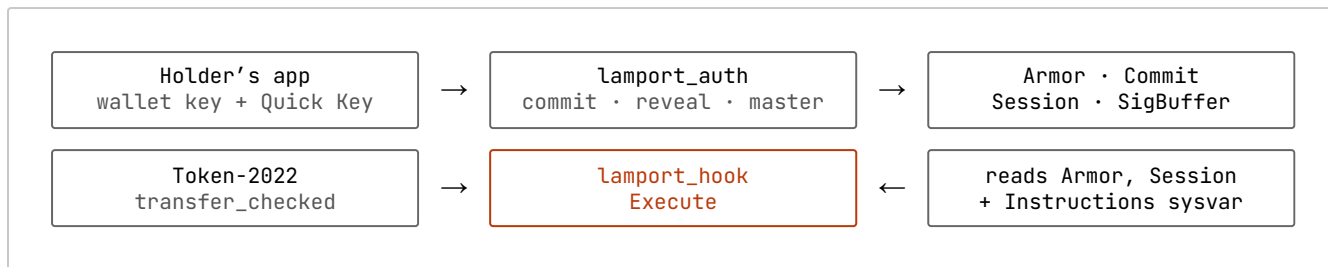


Fig. 1 — Architecture. `lamport_auth` holds per-account state and verifies proofs; `lamport_hook` is called by Token-2022 on every transfer and reads that state.

Table 1 — Program-derived accounts of `lamport_auth`.

Account	Seeds	Fields
Armor	["armor", token_account]	owner, token_account, quick_anchor (32 B), quick_links_remaining, master_root (32 B), master_next_leaf, recovery_delay_slots, recovery_unlock_slot?
Commit	["commit", armor, commitment]	slot
Session	["session", token_account]	slot, remaining, link
SigBuffer	["sig", armor, digest, payer]	staged Master Key signature bytes

The hook's `ExtraAccountMetaList` resolves, for every transfer, the source account's `Armor` PDA, its `Session` PDA (writable) and the `Instructions` sysvar. Its decision procedure is:

Algorithm 1 · `lamport_hook::Execute(source, amount)`

1. if `owner(source)` is off-curve (a PDA: `pool`, `vault`) → allow
2. if `Armor(source)` does not exist → allow (unarmed)
3. require `Session.slot` = current slot
4. require `Session.remaining` ≥ amount
5. require an earlier instruction in this transaction is `lamport_auth::reveal` for source
6. `Session.remaining` ← `Session.remaining` - amount; allow
7. otherwise fail with `LamportArmed`

Delegated transfers carry no special privilege: the hook judges the *source* account, so a delegate (or a drainer holding an approval) faces the same requirement as the owner. Arming requires the token account to carry the `ImmutableOwner` extension, so a stolen key cannot reassign the account's owner to escape the `Armor`.

4 The Quick Key protocol

4.1 Setup

The client draws a 32-byte secret from the system CSPRNG, independent of the wallet seed, and computes the chain

$$\ell_0 = x_0 \xleftarrow{\$} \{0,1\}^{256}, \quad \ell_i = H(\ell_{i-1}), \quad a = \ell_n, \quad n = 10\,000 \quad (1)$$

`arm(quick_anchor = a, quick_len = n, master_root, recovery_delay)`, signed by the owner, stores a in the Armor account. Links are recomputed on demand from x_0 , which is kept encrypted under a passphrase (sCrypt or Argon2id, then AES-256-GCM) in the browser and in a downloadable backup.

4.2 Announce, then reveal

To transfer v units with the next link $\ell = \ell_{k-1}$ (where the current anchor is $a = \ell_k$), the client first posts a commitment binding the link to the exact transfer:

$$c = H(\text{"lamport/commit/v1"} \parallel \ell \parallel v \parallel \nu \parallel s_{\text{exp}} \parallel \text{account}) \quad (2)$$

where ν is a random nonce and s_{exp} an expiry slot. `commit(c)` may be paid by anyone and records the current slot. Once the commitment is at least $\Delta = 4$ slots old, a single transaction carries `reveal( $\ell$ ,  $v$ , nonce,  $s_{\text{exp}}$ )` followed by the Token-2022 transfer. `reveal` checks

$$\exists \text{Commit}(c) : s_{\text{now}} - s_c \geq \Delta, \quad s_{\text{now}} \leq s_{\text{exp}}, \quad H(\ell) = a \quad (3)$$

and then updates state atomically:

$$a \leftarrow \ell, \quad \text{links} \leftarrow \text{links} - 1, \quad \text{Session} \leftarrow (s_{\text{now}}, v, \ell), \quad \text{close Commit}(c) \quad (4)$$

The hook (Algorithm 1) then accepts the transfer only in the same slot, only up to v , and only if the reveal precedes it in the same transaction. The commit's rent is refunded when it is closed.

tx 1 · slot s	lamport_auth::commit(c) → Commit(c).slot = s
wait	≥ 4 slots
tx 2 · slot s'	lamport_auth::reveal(ℓ , v, nonce, s_exp) → anchor ← ℓ , Session(s', v) token_2022::transfer_checked(v) → lamport_hook::Execute ✓

Fig. 2 — The two-step armed transfer. The app runs both steps behind one button; the wait is about 1.6 s.

Why the announcement? A revealed link is public the moment the reveal transaction is broadcast. Without the commitment, an adversary holding the wallet key could copy ℓ into its own transaction and race the owner. With it, any commitment the adversary creates for ℓ is necessarily younger than Δ slots when the owner's reveal lands, and the owner's reveal advances the anchor past ℓ in the same transaction, so ℓ is never accepted twice (Lemma 1).

5 The Master Key

Rare actions (disarming, rotating the Quick Key, cancelling recovery, changing the recovery delay) require a signature from a stateful hash-based scheme in the XMSS family [3]: Winternitz one-time signatures (WOTS) with $w = 16$ over SHA-256 with tweakable-hash domain separation, aggregated by a Merkle tree of height $h = 10$, giving $2^{10} = 1024$ one-time keys under one public root R . The signed digest binds the program, the account, the action and its parameters:

$$d = H(\text{"lamport/master/v1"} \parallel \text{program_id} \parallel \text{armor} \parallel \text{action} \parallel \text{params} \parallel i) \quad (5)$$

With $n = 32$ bytes and $w = 16$, a WOTS signature has $\ell_1 + \ell_2 = 64 + 3 = 67$ chain values (2,144 bytes); with the height-10 authentication path it exceeds one transaction, so it is staged into a SigBuffer with `stage_master_sig(offset, chunk)` and verified by `master_action(action, leaf_index, auth_path)`, which recomputes the WOTS public key, the leaf and the path to R . The program enforces a strictly increasing leaf index,

$$i \geq \text{master_next_leaf}, \quad \text{master_next_leaf} \leftarrow i + 1, \quad (6)$$

so a one-time key is never accepted twice, even if the client's own usage ledger is lost.

6 Recovery

An owner who loses the second key calls `request_recovery()` with the wallet key alone, setting $unlock = s_{\text{now}} + \delta$ with default $\delta = 1\,512\,000$ slots (about seven days). Until then, either second key can cancel: `cancel_recovery_quick` with a valid Quick Key reveal, or the `CancelRecovery` Master action. After the unlock slot, `finalize_recovery()` disarms the account.

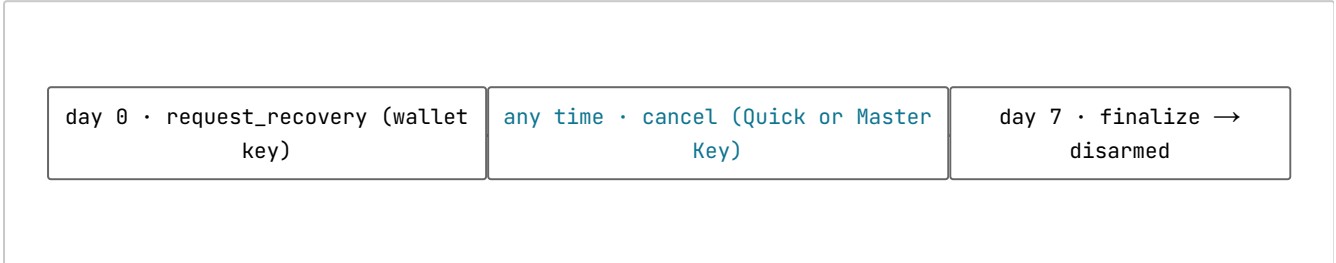


Fig. 3 — Recovery timeline. A thief with only the wallet key can start the clock; the owner stops it with either second key.

7 Security analysis

Theorem 1 (Unstealable). *Under the threat model of §2, no transfer that reduces the balance of an armed token account passes the hook unless the transaction contains a preimage of that account’s current anchor.*

Proof sketch. Algorithm 1 allows a transfer from an armed account only if a `reveal` for that account succeeded earlier in the same transaction, which by (3) requires a preimage of the current anchor under H . By preimage resistance, $\mathcal{A}$ cannot compute one from a ; by Lemma 1 it cannot use one the owner reveals; by Lemma 2 it cannot reuse an old one. ■

Lemma 1 (Front-running). *A link ℓ revealed by the owner cannot be used by $\mathcal{A}$: $\mathcal{A}$ learns ℓ only when the owner’s reveal is broadcast, so any commitment it makes to ℓ is younger than Δ when the owner’s reveal (included within Δ slots, by assumption) advances the anchor to ℓ , after which $H(\ell) \neq a$.*

Lemma 2 (No replay). *A session authorises at most v units, in one slot, in the transaction that created it; the anchor advances on every reveal, so a link is accepted at most once.*

Lemma 3 (No back doors). *Delegated transfers, swaps and drains are judged by the source account (Algorithm 1); owner reassignment is impossible (ImmutableOwner); Master Key leaves are single-use by (6).*

Quantum adversaries. A quantum computer that breaks ed25519 yields exactly the adversary of §2: a party holding the wallet key. Theorem 1 then applies unchanged. The remaining primitive, SHA-256, retains roughly 128-bit preimage security against Grover’s algorithm [6].

Composability. Pools and program vaults are off-curve and pass step 1 of Algorithm 1, so buying \$LAMPORT, providing liquidity and receiving tokens work everywhere. Selling from an armed account requires the venue integration (or the \$LAMPORT app) to place the reveal before the swap instructions, which aggregator “swap-instructions” APIs permit.¹

8 Limitations

- **Unarmed holders are as exposed as with any token.** Protection is opt-in, by design.
- **Burns.** Token-2022 burns do not invoke transfer hooks, so an adversary with the wallet key can burn armed tokens. It can never *take* them; a burn only reduces supply.
- **One extra step.** An armed transfer is two transactions about 1.6 s apart, with one extra fee (the commit’s rent is refunded).
- **Scope.** Only \$LAMPORT is protected. SOL and other tokens in the same wallet are not (see §9).
- **Censorship race.** If the owner’s reveal is broadcast but withheld from inclusion for at least Δ slots, an adversary who also holds the wallet key could commit to the exposed link, wait, and race the owner. Clients mitigate by submitting through several independent paths and retrying immediately; binding commits to a key derived from $\mathfrak{x}_0$ would remove the race and is planned.
- **Both factors lost.** If the wallet key is stolen *and* both second keys are lost, the adversary can request recovery and the owner cannot veto it. Recovery also assumes the owner notices a request within the delay; the app surfaces pending requests prominently.
- **Endpoint compromise.** Malware on a device that holds the wallet key and the decrypted Quick Key at the same moment defeats the separation. Hardware-held second keys are future work.
- **Implementation risk.** Both programs will be audited before mainnet and then made immutable; until then this document describes intended behaviour.

9 Future work

The Lamport Hook standard: any new mint can name the same hook at launch, and a single arming setup can protect every Lamport-hooked token a holder owns. **Armored wrappers:** deposit SOL, USDC or any SPL token and receive a 1:1 hook-protected wrapper (qSOL, qUSDC), with the underlying held by a program account that has no private key. **Platform economics:** fees from hooked launches and wrappers fund \$LAMPORT buybacks.

References

- [1] L. Lamport. Constructing digital signatures from a one way function. Technical Report CSL-98, SRI International, October 1979.
- [2] R. C. Merkle. A certified digital signature. In Advances in Cryptology: CRYPTO '89, LNCS 435, Springer, 1990.
- [3] A. Hülsing, D. Butin, S. Gazdag, J. Rijneveld, A. Mohaisen. XMSS: eXtended Merkle Signature Scheme. RFC 8391, IRTF, 2018.
- [4] NIST. FIPS 205: Stateless Hash-Based Digital Signature Standard (SLH-DSA). 2024.
- [5] P. W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In Proc. 35th FOCS, IEEE, 1994.
- [6] L. K. Grover. A fast quantum mechanical algorithm for database search. In Proc. 28th STOC, ACM, 1996.
- [7] L. Lamport. Password authentication with insecure communication. Communications of the ACM 24(11), 1981.
- [8] N. Haller. The S/KEY One-Time Password System. RFC 1760, IETF, 1995.
- [9] Solana Program Library. Token-2022 and the Transfer Hook Interface. Solana Labs / Anza documentation.
- [10] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, B.-Y. Yang. High-speed high-security signatures. Journal of Cryptographic Engineering 2(2), 2012.

-
1. Liquidity venues must support transfer-hook mints; at the time of writing, Orca Whirlpools does so through token badges, and a minimal constant-product pool with full hook support is planned as a fallback. ↩

\$LAMPORT is not affiliated with or endorsed by Leslie Lamport.