Stellar Smart Wallet (SW) Standard

2024/06/13 – DRAFT
2024/07/09 – DRAFT

Discussion should now happen on the GitHub discussion

[UPDATE 2024/07/09]
Official SEP discussion has been posted
https://github.com/stellar/stellar-protocol/discussions/1499

—

I've spent a lot of time over the past month playing with SWs on Stellar. With the incoming protocol 21 release adding the secp256r1 curve to Soroban we'll be ready to use SWs on mainnet save for one massive gapping hole. What should the standard interface be? A SW is a smart contract so it could be a lot of things. The goal of this doc is to take a stab at what I think the initial v1 interface should permit along with some logic and then some future v2+ features to prioritize.

The hope is we can relatively quickly arrive at some sort of consensus on the interface and then a contract implementation we can send off to The Auditors™ such that we can have a contract interface we're all comfortable and excited to use on mainnet.

V1

In progress reference implementation
(implements all of V1 except for recovery signers and the session signers are currently persistent not temporary)

Resources

- https://github.com/kalepail/passkey-kit
- https://github.com/kalepail/superpeach
- https://github.com/kalepail/launchtube

Admin signers

- During the initial deployment of a new SW you elect a key as an initial "admin signer". Admin signers have special permissions as the only signers on the SW which can perform protected actions such as adding and removing additional signers or upgrading the contract code.

- There should be some discussion around the difference between admin signers and recovery signers. Should there be allowance for more than one sudo signer? Can recovery signers also upgrade the underlying wasm?

Temporary session signers

- The third and final signer type for v1 SWs are session signers. These are signers which are added to the SW as temporary entries which can then be used to sign transactions for the SW “inline” without needing to load in the domain of the sudo signer for every interaction.
- There will be an initial connection to the sudo signer domain in order to sign for adding the temporary signer but after that has been signed and submitted the original wallet requestor page never needs to visit the sudo domain again.
- Given the relatively long lifespan of temporary entries there should be an option to restrict the validity of a session signer even if the entry still exists on ledger.
- Temporary signers remove a centralized domain dependency which exists in the current SW ecosystem. However without this centralized point of contact for all transaction signing there are a few notable concerns:
    - Transaction review
        - Session signers will not be required to confirm or display any transaction information with a user prior to signing. I for one am fine with this as I don’t believe transaction review to be an actual security mechanic anyway. If I was nefarious I wouldn’t hide me schemes in a simple transaction and complex transactions are essentially impossible to review for the majority, which is who SW are aiming to service.
        - Another reason I’m not terribly concerned about this is we already operate in a world of domain based security and passkeys solves for many of the inherent weaknesses in domain based security. Passkeys are domain bound. So you couldn’t use a session signer generated on `goodnft.com` over on `gotchanft.com`. Now I suppose you could load up `gotchanft.com` and spin up a brand new session signer and then start signing things you shouldn’t but at the point of a new session signer creation you have to interact with the sudo domain which would hopefully be able to intercept bad actors. At the very least you’re in no more dangerous of a spot than the traditional single source model.
    - No central hub for deriving the SW’s contract address
        - See the Reverse signer lookup section for a mitigation design
    - ???

Reverse signer lookup

- The final necessary component in my estimation at this time for creating a viable v1 SW is the ability to go from a passkey id to a SW address. This may seem trivial enough but it’s actually not. When creating a passkey you are given an id and a public key. The

public key you can NEVER retrieve again. The id you can get again in the future when you request signatures as a sort of “sign in” method but the public key is locked down never to be seen again.

- This requires that you save both the id and the public key.
- The public key is easy, you just save that as the signer in the SW whenever you deploy a new SW or add an additional recovery or temporary signer.
- The id is the tricky bit as far as reverse lookup is concerned. What most of us think about when we think of a SW is a contract `Address` as we should. However what we have when we’re “signing into” a SW is passkey id. We cannot store this id in the SW then as this would break reverse lookup. I could find all my signer ids but I need to be able to take a signer id and derive a SW contract address. For at least the initial sudo signer this can be solved by using the passkey id as the salt for the SW’s contract address. Then whenever you’ve got the id you can compute the `Address`. This requires knowing the sudo signer passkey id though which, with the inclusion of temporary session signers we won’t have, and given we can both swap out the sudo signer and remove signers altogether we may never be able to retrieve.
- My proposal then is to store a reverse lookup table on the SW factory contract which will allow anyone to lookup a SW contract address for any given passkey id.
    - This requires a cross contract call from the SW to the factory contract during any add or remove signature invocation. Given that most signers will be temporary entries I believe this is an acceptable cost
    - Adds some complexity to keeping track of keys to signers especially given some signers are persistent entries while others are temporary. Not convinced the complexity is significant but I suppose that depends on how we solve for resudo events and future transfer events. My guess is in all cases however it won’t be terribly difficult to track.
    - Recovery signers likely won’t be added to this map
        - They aren’t for general usage and thus this utility wouldn’t be necessary
    - Sudo signers *do* need to be added to this list
        - Only the initial sudo signer could be omitted if it were used as the salt of the SW contract address.
    - If a wallet upgrades its wasm the entire mapping would be broken so either you’d need to:
        - Track the history of factory addresses. Might be reasonable as the likelihood of many wallet upgrades is small imo.
            - I personally don’t love this. In some cases I think folks will want to entirely abandon old factory contracts and hanging onto them just for the reverse lookup convenience is probably the wrong move.
        - Just be okay losing it and starting from scratch. In the end this is a utility feature to aid in a cross device, onchain, sign in process. Temporary entries would eventually expire and the sudo signer could be transferred as part of the upgrade process. (recovery signers likely aren’t affected as they aren’t tracked by this feature in this current version)

Upgradability

- SWs should have the ability to upgrade their underlying wasm in order to upgrade or switch their functionality
- This should be a protected action not just any site can request

V2

G to C compatibility considerations

- How can we easily get funds from G addresses forwarded into C SWs?
- This is likely not a SW interface question as much as it is a utility function external to this interface. There may be protocol changes needed here but more work needs to be done
- ???

C to G compatibility considerations

- This is a bigger meatball as it requires more thought and experimentation to uncover where the real and legitimate pain points are. Should C addresses have access to the DEX? AMMs?
- ???

Transfer signers from one wallet to another

- Not convinced this is a must have feature or just a nice to have feature. Given that passkey public keys are irrecoverable from the passkey/webauthn standard they become a somewhat precious onchain resource. It would be nice to have some mechanics for moving them around to different wallets.
- I think it would make sense for it to even support adding them vs just transferring. Not sure what the implications would be, especially for the reverse signer lookup from the v1 standard but it's probably worth exploring.

Recovery signers

- In order to maintain the resilience of SWs we should be able to add additional signers for the purpose of recovery. The idea is that these could be any number of `Addresses` or maybe even other more arbitrary signer types like hashes (e.g. sha256 of some secret phrase which you could input as an arg) which could be utilized in the case of account recovery. Specifically instances where the sudo signer has been lost.
- The idea is these signers could probably only perform a singular action and maybe when they executed that action they were burned. (at least in the case of a hash signer).
- This action would be something like swapping the wasm (probably not that helpful) or forcing a new sudo signer (likely the most obvious).

- We could add some multisig options here like allowing a m-of-n scheme for recovery signers to be able to perform the recovery action. E.g. you add a hash signer an ed25519 signer and another passkey all as your recovery accounts and configure them in a 2-of-3 setup where you'll need at least 2 of them in order to execute the recovery action.
- I think for v1 we should only support ed25519 and secp256r1 signers

V+

Complex policies

- I would like to see standards arise for supporting complex policies when connecting to a SW. E.g. "Only interact with this/these `Addresses`", "Require multiple signatures for this request", "Spend limits on various assets in a specific time frame", etc.

Robust multisig solutions

- V1 has recovery signers and session signers but we should consider adding more signatory support to allow for a wider range of use cases. Specifically the additional signers and signer scenarios that native Stellar supports.