マルチシグ方式のPeg-inを作る

マルチシグ方式ブリッジの試作

2025/10/17

Bitcoin – Ethereum のブリッジを作ってみることになった。

全体の流れはおおよそわかっているつもりだが、ユーザーが使用できるレベルとなると考えることが多すぎる。

まずは何も考えない、ド正常系だけ考えよう。



ブリッジの検討をしたが、Peg-in と Peg-out は別で検討してもよいだろう。BitVM2 bridge では Peg-in 額ごとにインスタンス(スクリプトや検証計算などのセット)を持つので別にはできないかもしれないが、Citrea の Clementine v1 はマルチシグアドレスを使っているので UTXO の額さえ固定にできれば良いのだろう。

Peg-in手順

Ethereum のトークンと交換する。

交換比率は Bitcoin と一対一だが手数料を取る。仮で 1000 satsにしておく。

- Bitcoinマルチシグウォレットからアドレスを割り振る。
  - 1依頼1アドレスとしておくが足りなくなるかな？
- そのアドレスに手数料を含めて入金する
  - Ethereum側のアドレスをどうやって決めようか。
    - `OP_RETURN` に入れても良いが誰からも見られるし、それなら DBの方がまだましか
- Ethereum側署名者に署名対象のデータを配布
  - 送金先EOA、送金額
  - それ以外の情報は nonce だけでスマコンから取得する
    - nonce はマルチシグで操作する mint, burn, マルチシグからの transfer を行うと更新する。mint と burn はあまり行わないだろうと思うが transfer は集中するかもしれない。順番に処理しないと nonce がずれることになってひどい目にあいそうだ。そう

でなくてもボトルネックにはなる。

署名集約集約の担当者が署名を集め、マルチシグからのtransferスマコンを呼び出す

DB 更新: ブリッジ処理の完了

第三者機関での検証

ScaleBit などがやっているようだ。

ブリッジは攻撃に遭いやすい

トランザクションの構造

Bitlayer の Peg-in と思われるトランザクションを見ると、 `OP_RETURN` に EVM アカウントのアドレス文字列が埋め込まれていた(`0x` も含め)。

Bitlayer - Bitcoin L2#f6839c

```
2c6a2a3078313831426437304131426331373331393633306137613138326634386238363962323165623
33039
```

→ `<全データ長> OP_RETURN <データ長> 0x181Bd70A1Bc17319630a7a182f48b869b21eb309`

他のデータは何もない。

このトランザクションを見たら、どのチェーンかはともかく EVM アドレスに関する何かをしているというのはすぐ分かるな。

Testnet だからだろうか。

Bitcoin側

入金のためのマルチシグウォレットとその入金の監視がいる。

Peg-in に関してはアドレスさえ生成できれば良い。

xpubを使ったマルチシグアドレス

Grok

ChatGPT

P2WSH までは BIP-48 で HD wallet を 6階層で使うフォーマットがある。 `script_type` が増えた階層で、P2WSH なら `2H` になる。

bip-48

```
m / purpose' / coin_type' / account' / script_type' / change / address_index
```

P2TR は BIP-389 の提案があるもののまだ整備されていないようだ。

調べてみたものの、マルチシグをどうやって1つの書式にしているのだろうか。

公開鍵をソートする仕様は無いので、ユーザの順番を決めて署名を並べることになる。

miniscript ポリシーを `thresh(3,pk({A}),pk({B}),pk({C}))` の形にすると `tr(UNSPENDABLE_KEY,multi_a(3,A,B,C))` になる。

DeepWiki

policyからスクリプトを作ると、鍵が3つだと定足数が3なので 3-of-3 になり `OP_CHECKSIG_ADD` は使わないようになる。

policy

```
tr(
  50929b74c1a04954b78b4b6035e97a5e078a5a0f28ec96d547bfee9ace803ac0,
  and_v(
    v:and_v(
        v:pk([<FINGERPRINT-A>/86'/1'/0']tpub<KEY-A>/0/*),
          pk([FINGERPRINT-B>/86'/1'/0']tpub<KEY-B>/0/*)),
      pk([FINGERPRINT-C>/86'/1'/0']tpub<KEY-C>/0/*)
  )
)#<CHECKSUM>
```

script

```
OP_PUSHBYTES_32 5992bbb40798837f894fe09dce3f8378ad0d13c2c0475545d0d1ec2a3fecb878
OP_CHECKSIGVERIFY OP_PUSHBYTES_32
8fbb7bdb202585fc9d2471d904341098a517ac4f01189bae31aa36fdb4b7912a OP_CHECKSIGVERIFY
OP_PUSHBYTES_32 3bd20984ce6d3b0732bf05c28ff28cabc99999347ccc02e951cdd4c70b23be2b
OP_CHECKSIG
```

これが 2-of-3 になる構成だと P2TR でよく見る `OP_CHECKSIGADD` を使ったマルチシグの構成になる。

policy

```
thresh(2,
  pk([<FINGERPRINT-A>/86'/1'/0']tpub<KEY-A>/0/*),
  pk([<FINGERPRINT-B>/86'/1'/0']tpub<KEY-B>/0/*),
  pk([<FINGERPRINT-C>/86'/1'/0']tpub<KEY-C>/0/*)
)
```

3-of-3の実例

```
thresh(3,pk([856222ca/86'/1'/0']tpubDCi9NEEiajSYVQ6EEdEXhnNnX1YdkYVdz1BarkPq9hbyExi
XNDDoP1NPea7jM3LAuByUg2eUcUhouebokdVgL6btTqVAXAHySQ7AG2ifkyt/0/
*),pk([872e48b1/86'/1'/0']tpubDCL5ULimhpgdEXTgacLek2xLxZcVxuz3cUq1XF4Mi8vs1Xedhcs4x
```

```
izVXPL28WSQJPP41Yrqn8ShDEqbVg6LtqeQtGkZKday2aFmgWkoauG/0/
*),pk([ca1f2ae0/86'/1'/0']tpubDCXxJpe6HnAJA1fFj6tgUMYCvcu36vv9JnhcRZRMcDiujwkTMNXDS
Dggy1SjKQnM1ev99PJqwphCVhw36XdwgrSgd1NguoGbjW9rVH82iHf/0/*))
```

```
script
    OP_PUSHBYTES_32 bc514d6335c18ac6b2f022eb3d6fbf792c15787623d3f124f55016768e734f78
    OP_CHECKSIG
    OP_PUSHBYTES_32 ebda6c81242817724b9b67f486f6b7ca2fbc6b40187832f27b67f536de2db649
    OP_CHECKSIGADD
    OP_PUSHBYTES_32 61f3dbfb24bcb56c5bdef83412d3554c47179ca8d3875ed27413f714f4af9cee
    OP_CHECKSIGADD
    OP_PUSHNUM_2 OP_NUMEQUAL
```

Ethereum側

ETHを払うのではなくトークンを発行する。新規発行ではなくマルチウォレット管理している持ち分から転送する。
スマートコントラクトを実行するため各アカウントにいくらかのETHは必要だが、分配するためのETHではないので

マルチシグコントラクト

Solicityでマルチシグ相当のコードを書くことになる。
一応 Grok で例を作ってもらったが、OpenZeppelin でも提供しているようなのでそちらを使った方が安全だろう。

> ERC-7913: 署名者表現の概念をEVMアドレスを持たない鍵に対して拡張
>
> `MultiSignerERC7913`

OpenZeppelin - Multisig Account

https://docs.berachain.com/developers/guides/create-erc20-contract-using-foundry

```
$ forge install OpenZeppelin/openzeppelin-contracts@v5.4.0
$ forge install OpenZeppelin/openzeppelin-contracts-upgradeable@v5.4.0
```

ETH をマルチシグで管理するのは良いとして、トークンをマルチシグで管理するのはどうすると良いのか。
ERC-20 は `address` に対して `uint256` の残高を割り当てているだけで、`transfer(to, value)` の送金元は `msg.sender` なので何とかなりそう？

"Account Abstraction" という考えがあり、それを一部実現したのが ERC-4337 だそうだ。

MetaMaskでも使えそうな感じがしなくもないし、方向性としては悪くないだろう。

https://docs.openzeppelin.com/contracts/5.x/account-abstraction

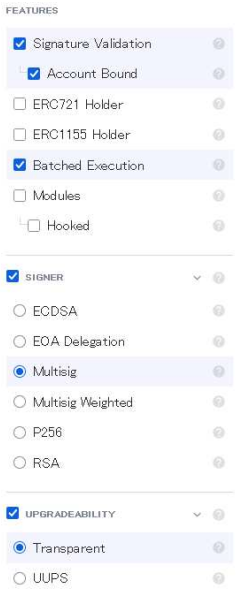
【ERC4337（1）】EntryPointのhandleOps()の処理を理解する、Paymasterを指定してガス代について検証する - レコチョクのエンジニアブログ

Account Abstraction(ERC4337)を、具体的な処理を追ってしっかりと理解してみましょう。

各signer が署名して coodinator に投げて集約するのが、各signer が署名データが入った UserOeration を作って "Blender" である coodinator に投げて EntryPoint を実行して集約するということかな？

https://docs.openzeppelin.com/contracts/5.x/multisig

https://wizard.openzeppelin.com/embed?tab=Account



Developer’s Guide to ERC-4337 #1 - Developing Simple Account - by Nikhil - Block Magnates

おそらく。。。

ERC-4337 の Abstract Account が `Account`

https://docs.openzeppelin.com/contracts/5.x/account-abstraction

`PackedUserOperation` や `EntryPoint`

ERC-7913 は Selecting a signer

これは `_rawSignatureValidation(hash, signature)` を提供する

`hash` は keccak256

`hash` はスマコンで計算するようにしないと

`signature` は `abi.encode(signers, signatures)` で計算(例)

`MultiSingerERC7913`

これは `is` に入れておけばあまり考えなくて良い？

デプロイのために account factory を用意するのが良い？

MultiSignerERC7913._rawSignatureValidation()

署名の再利用を防ぐハッシュ値計算の検討

Ethereum では定足数以上の署名を集めるとトークンをEOAに割り当てるようにしている。
同じ署名があれば同じことができてしまう、では困る。
鍵は変更しないので署名が意味をなさないようにするためにはハッシュ値計算を考える必要がある。

アプリ側はどうとでもできてしまうのでブロックチェーン側で拒否できるようにする必要がある。
チェーンに使用したことがある署名を全部記録して毎回チェックすれば可能かもしれないが、Ethereum L1で保存するにはコストが掛かりすぎるだろうしチェックするコストもかかる。

1回目検討

スマートコントラクトに `nonce` を持つ
`nonce` は署名検証が必要な処理が成功するとインクリメントされる
mint, burn, transfer
ハッシュ計算は `nonce` 、金額、キーワードを連結した値を keccak256する
キーワードは transfer の場合 送金先アドレス
この方式の場合、1つの検証が終わるまで次の検証が行えない。
問題ない？
各ノードに署名要求 → 各ノードが署名して返信 → 検証、がセットなので、この間は mint, burn, transfer できない。
リクエストをブロックするか、シーケンシャルに処理するようにするかだが、P2Pで中央DB を持たないシステムにした場合は難しいと思う。

リクエストが大量に来たときにどうやって順番を付けて1つずつさばくか。
そもそもリクエストをどうやって受け付けるのか。

BitlayerではPeg-Inごとに管理する単位を作ってPeg-Outまで使うらしい。「インスタンス」と呼んでいるがVMインスタンスではなく、ある意味ユーザアカウントのようなものだと思うがBitcoin L2で扱う単位と思えば良いだろう。
Bitlayer address は EVM互換で ERC-20トークンとして扱うということだからイメージは同じだろう。

Bitlayer - Bitcoin L2#11d5e0

1回目の結果

署名側も検証側もスマコンから `nonce` を取得してハッシュ計算するようにした。署名を使った処理が成功したら `nonce` をインクリメントする。

実際にこの方式で複数のPeg-Inを同時に行ったところ問題があった。

> スマコンで取得できる `nonce` は最後に同意が確定した値なので、ブロックが進むまでは同じ値が取得されるし、ブロックが進むとそれまでにインクリメントが呼ばれた分だけ `nonce` も進む
>
> 同時にPeg-Inを行ってもスロット制のため処理をするのはリーダーになったノードだけなので時間がかかる。その間にブロックが進んでしまうと署名したときの `nonce` と検証するときの `nonce` に違いが発生して失敗した。

2回目検討

pendingしたトランザクションを数えることで `nonce` がいくつになりそうか推測はできるが、その間にpendingが増えてしまえば意味がない。

そこでリーダーが署名を求めるP2Pメッセージに `nonce` を載せ、ブロックが変わって `nonce` が更新されてもその値を使って署名と検証を行えば失敗しない。

この方法の問題は、ブロックが進んで `nonce` がブロックチェーンでの値と不一致になったとしてどのくらいの範囲まで許容するかが決めきれないということだ。元々 `nonce` を使った理由は値のインクリメントのためには定足数の署名が必要になるため、再利用しようとしてもそのときには `nonce` が進んでいるため検証に失敗するというメリットがあるからだ。EOAが持つ `nonce` と同じような意味合いである。許容する範囲を広げるとそれだけ署名の再利用ができてしまうリスクが高まる。

ブロックごとに変わる値という意味ではブロック高も同じである。 `nonce` が呼ばれた分だけインクリメントされるのに対してブロックは1つずつしか進まないのでハッシュに含める値をブロック高にすることで範囲が狭くできるのではないかと考えた。なおスマートコントラクトで使用できる `block.number` は次に承認される予定のブロック高で RPC APIで取得できる `blockNumber` は最後に承認されたブロック高なのでブロックが進む前でも `block.number` のほうが1つ進んでいる。

`nonce` の代わりにブロック高にしたところ、結局こちらもどのくらい前のブロック高まで許容するのかという同じ問題が発生した。

3回目検討

ようやく AI に質問することを思いついた。

ChatGPT

- `nonce` はアプリでランダム生成
  - UUID程度でよいようだ(128bit)
- `bool usedNonce[UUID]` で使用済み `nonce` を管理する
- スマコン実行時に `nonce` をチェックし、使用済みであれば処理しない
  - ブロック未確定で同じ `nonce` をチェックをしても通ってしまうが、それはPengingトランザクションとして展開されるだけ。ブロックの承認時にはトランザクションがシーケンシャルに処理されるので同じ `nonce` に対するトランザクションが複数あっても最初に処理するトランザクションが `usedNonce[UUID]` を使用済みにすると次のトランザクションではチェックがNGになってrevertする。
- 当然ながらハッシュ計算にも `nonce` を含める

今のところこれで問題は起きていないが、ブリッジ数1，ユーザ1の場合である。基本的な考え方として問題はないと思うので、あとは運用とかハッシュ計算に見直しがあるかもしれないというところか？

分散合意

Peg-Inはこういう流れになる。

1. Bitcoinマルチシグアドレスをもらう
2. それに対してEOAを登録
3. ユーザはBitcoinマルチシグアドレスに送金
4. 十分confirmしたらEthereumマルチシグコントラクトからトークン送金する署名を集めて登録してあるEOAアドレスに転送

1回のユーザ要求で他のノードから何度もトークンを転送できてはいけない。

署名は使い捨てになっていないといけない。そうするためマルチシグウォレットの操作に必要となる署名には `nonce` を含めるようにし、操作した場合は `nonce` を必ず増やす。 `nonce` はブロックチェーンで管理するので合意が無いと変更できない。

同じユーザ要求に対して何度も署名を集められないようにする。ノードAが署名を集めるリーダーだったとして、同じことをノードBも行えるとよろしくない。これは `nonce` が操作ごとに変わって署名が使い捨てになっていることによって防ぐことができる。

署名するハッシュ値はリーダーからメッセージでもらう。これをそのまま信用すると `nonce` が異なっていたり、ユーザ要求とは異なるEOAになっているかもしれない。今のところPeg-Inの登録はどれかのノードが受け取ったらブロードキャストして全ノードが同じ

データを持つようにしている。Bitcoinマルチシグアドレスは各要求のたびに生成して、そのアドレスへの送金を確認することはブロックチェーンを見ると確認できる。

危険なのは、悪意あるノードがユーザが登録しようとしたEOAではなく自分のアドレスをブロードキャストするという行為。これを検出するのは難しい。登録要求を分散するしかないのか。

考え直して、P2P相手からの情報をなるべく使わないようにする。

Peg-In署名要求は、送られてきたUUIDでDB検索。 `out_point` のトランザクションを取得して送金額とDBの手数料からトークン量を計算。それに対してスマコン用のハッシュ計算と署名を行う。

_P2pRequest

```
struct _P2pRequest {
    uuid: Uuid,
    out_point: String,
}
```

時間はかかるようになるが、チェックできるのはこのくらいだろう。