Introduction

We propose a protocol for composable block building on SUAVE. In this protocol for composable block construction, both builders and searchers can bid on sections of blocks, which can be included into a block with some pre-defined fees. Presently, there exists a private trust relationship between searchers and builders, where bundles are sent exclusively to specific builders. However, with the introduction of modular block construction, we anticipate achieving more efficient block building processes.

Design

Meta Bundles

[

Untitled

1266×927 110 KB

](https://collective.flashbots.net/uploads/default/original/2X/9/923c4de539146532411c38bbafc0eca760d64dbf.png)

In order to compose a block from multiple parties, we introduce meta bundles. It is an aggregation of bundles built from a builder. Meta bundles can conceptually be understood as partial block.

In order to ‘unlock’ this bundle to include them into the block state, builders need to additionally send a signed transaction on a destination chain that pays to the fee recipient of the meta bundle. The payment transaction is verified on suave by simulating the transaction through the simulateBundle

precompile. Upon successful verification of payment to the meta bundle author (fee recipient), the meta bundle merged with a valid payment transaction is saved in the confidential store. Block builder can later access the meta bundle by its id, and merge the bundles and meta bundles into block by passing list of DataId

.

Sequence Diagram

We outline block building scenario in following sequence diagram:

[

image

4479×2081 321 KB

](https://collective.flashbots.net/uploads/default/original/2X/3/324d52822628922a8c635f6143a233f9c7b6a0dd.png)

- User uploads a bundle to the bundle contract, which then stores the bundle information in a secure storage.
- The bundle contract generates a HintEvent that includes a data identifier for locating the bundle within the secure storage.
- Builder creates and submits a meta bundle containing a series of data identifiers linked to the bundles. This submission includes setting the value

and feeRecipient

to indicate the cost for another builder to access and use the bundle in their block.

- The meta bundle contract announces the creation of the meta bundle through an event, using the data id of the meta bundle.
- Builder intending to integrate the meta bundle into his block executes a signed transaction to transfer the specified value

to the feeRecipient

. Upon successful verification of the payment, the meta bundle’s content is marked as “matched,” making it accessible through the builder contract.

- The builder invokes the build()

function on the builder contract, supplying the sequence of data ids. These ids are used by the buildEthBlock()

precompile function to assemble the block, which is then forwarded to a relay for processing.

Basic Bundle Contract

[

image

2118×1096 110 KB

](https://collective.flashbots.net/uploads/default/original/2X/a/a7ddd878d05e95b8ad250ea9be5e30e99a4e72ad.png)

The basic bundle contract is any contract that produces bundle in the confidential store. To clearly state what protocol it is using, it is saved in mevcompose:v0:ethBundles

namespace.

Mev-Compose Contract

[

image

2118×1580 222 KB

](https://collective.flashbots.net/uploads/default/original/2X/9/9a51ac8269357a107b85b1d2147b3aa6155d9224.png)

Upon submission of a metabundle, it is initially stored within the mevcompose:v0:unmatchedMetaBundles

version. This process involves specifying a feeRecipient

and a value

, which are used to determine the price of the metabundle, should the builder opt to incorporate it into a block. Note that the meta bundles consist of the DataId

s of the bundle in confidential store, not the actual contents of bundle. It is to ensure that nobody in the bundle supply chain has access to bundle until it is actually included to block.

[

image

2265×1635 276 KB

](https://collective.flashbots.net/uploads/default/original/2X/2/29876e5d32a9ddcba364c99a519b55e52a7dedc2.png)

If the payment transaction is verified from the mev compose contract, the metabundle is merged with the payment transaction. Now the paid

meta bundle is saved in mevcompose:v0:matchMetaBundles

.

Block Builder Contract

[

image

2841×1101 265 KB

](https://collective.flashbots.net/uploads/default/original/2X/2/2e4d9c27f34958fa05ae38e60e51f095352dd731.png)

To build the contract, builder issues build()

transaction to his builder contract. Builder sends the list of bundle ids and meta bundle ids, and buildEthBlock

precompile processes them by the Version

attribute of the data record.

var mergedBundles []types.SBundle for _, record := range recordsToMerge { switch record.Version { case "mevcompose:v0:matchMetaBundles": log.Info("Started processing matchMetaBundles.", "record", record) matchedBundleIdsBytes, err := b.suaveContext.Backend.ConfidentialStore.Retrieve(record.Id, buildEthBlockAddr, "mevcompose:v0:mergedDataRecords") if err != nil { return nil, nil, fmt.Errorf("could not retrieve record ids data for record %v, from cdas: %w", record, err) } unpackeddataIDs, err := dataIDsAbi.Inputs.Unpack(matchedBundleIdsBytes) matchdataIDs := unpackeddataIDs[0].([][16]byte)

```
    log.Info("Retrieved dataids", "matchdataIDs", matchdataIDs)

    for _, matchdataID := range matchdataIDs {
        bundleBytes, err := b.suaveContext.Backend.ConfidentialStore.Retrieve(
            matchdataID, buildEthBlockAddr, "mevcompose:v0:ethBundles")
        if err != nil {
            return nil, nil, fmt.Errorf("could not retrieve bundle data for dataID %v, from cdas: %w", matchdataID, err)
        }

        var bundle types.SBundle
        if err := json.Unmarshal(bundleBytes, &bundle); err != nil {
            return nil, nil, fmt.Errorf("could not unmarshal bundle data for dataID %v, from cdas: %w", matchdataID, err)
        }
        mergedBundles = append(mergedBundles, bundle)
    }

    paymentBundleBytes, err := b.suaveContext.Backend.ConfidentialStore.Retrieve(record.Id, buildEthBlockAddr,
        "mevcompose:v0:paymentBundles")
    if err != nil {
        return nil, nil, fmt.Errorf("could not retrieve payment bundle data for dataID %v, from cdas: %w", record.Id, err)
    }

    var paymentBundle types.SBundle
    if err := json.Unmarshal(paymentBundleBytes, &paymentBundle); err != nil {
        return nil, nil, fmt.Errorf("could not unmarshal payment bundle data for dataID %v, from cdas: %w", record.Id, err)
    }

    mergedBundles = append(mergedBundles, paymentBundle)

case "default:v0:ethBundles":
    bundleBytes, err := b.suaveContext.Backend.ConfidentialStore.Retrieve(record.Id, buildEthBlockAddr, "default:v0:ethBundles")
    if err != nil {
        return nil, nil, fmt.Errorf("could not retrieve bundle data for dataID %v, from cdas: %w", record.Id, err)
    }

    var bundle types.SBundle
    if err := json.Unmarshal(bundleBytes, &bundle); err != nil {
        return nil, nil, fmt.Errorf("could not unmarshal bundle data for dataID %v, from cdas: %w", record.Id, err)
    }
    mergedBundles = append(mergedBundles, bundle)
default:
    return nil, nil, fmt.Errorf("unknown record version %s", record.Version)
}

}
```

This allows the builder to treat basic bundle types and meta bundle types in same manner as buildEthBlock

will handle them differently. If the given DataId

belongs to a metabundle, the list of data ids within the meta bundle is used to access the actual bundle data in the confidential store. Payment transaction is handled after including all the bundles in a meta bundle.

Discussion topics

allowedPeekers - can it have only one precompile address?

It seems like the allowedPeekers requires at least one contract address to check whether the data record is being accessed in a contract that user allows. However in my design, the bundles are referenced entirely based on their DataId

s such that bundles are never revealed until inclusion to the block. So I set the allowedPeekers to [address(this), Suave.buildEthBlock]

as I am not going to call confidentialRetrieve

at all. The advantage of this scheme is that Bundle contracts don’t need to care about to what contracts it should give access to. The suapps that want to use this bundle should reference the bundles by DataId

, and block builders use DataId

to include them to the block. Otherwise, it becomes bundle contract owner’s responsibility to set the correct allowedPeekers

by checking if builder contract or meta bundle contract isn’t doing something malicious (such as leaking the confidential store). Have less trust assumptions and responsibilities promises greater composability and larger search space, and since block building is possible without revealing the actual bundle data, it should be possible to set allowedPeekers

of bundles to be buildEthBlock

precompile only.

We need fully confidential version of M.O.S.S

As IBundle.apply()

returns the raw transactions in a bundle, it automatically requires full trust on the peeker. How about IBundle.apply(DataId block)

, which applies the bundles to the block in the confidential store

? The IBundle.apply()

would simply append the bundle’s DataId to a list in the confidential store, and IBlock.build()

triggers actual block building by the list of DataId. As it doesn’t require trust assumption and careful access control, this offers greater flexibility to the composability.

Recursive meta bundles

It is theoretically possible to have recursive meta bundles - a meta bundle nesting a list of meta bundles. Following pseudo code (python) should flatten the nested meta bundles into a flat list of bundles

def flatten_bundle(bundle_ids): ret = [] for data_id in bundle_ids: record = Suave.confidentialRetrieve(data_id) if record.Version == "mevcompose:v0:metaBundles": ret.extend(flatten(data_id)) elif record.Version == "mevcompose:v0:ethBundles" ret.append(data_id) else: fatal("unknown bundle type") return ret

I decided to leave it as a future TODO because writing a integration test for this feature over-complicates the project.

Conclusion / Thoughts

The objective was to design a secure protocol that keeps the bundle hidden until buildEthBlock

is triggered. The handling of bundles is conducted in the confidential store without retrieving the actual bundle data in any of the smart contracts - that was until I realized setting allowedPeekers

is not possible without specifying a contract address. My goal was to reduce the need for trust as much as possible, thereby providing more flexibility to suapp developers, and I achieved some level of success in this proof of concept. I understand there’s a plan to retire certain precompiles and transfer protocol-specific functions to smart contracts. However, preserving the protocol’s operations within precompiles (like in buildEthBlock

) helps us minimize the burden of managing access control to bundles in confidential stores.