Capsules are an attractive primitive in that they push on-chain complexity to off-chain components. However, there have been valid concerns as to how this affects composability. In this document we'll first identify some use cases on Ethereum that could be served by capsules, and then attempt to identify different composability scenarios, both at the contract and the dapp level, and propose solutions for each of them.

This is a long doc. Short on time? Jump ahead to theconclusion.

This doc was written in collaboration with@nventuro. Read also "Using capsules for private storage-less open-oracles" and "DeFi Liquidity Mining and Capsules".

Ethereum sample use cases

In general, capsules are a good fit for scenarios where a contract needs to adhere to a standardized interface, but needs additional contextual data to execute. Some examples from Ethereum are:

- Token approvals

: A token transfer on behalf of a different user (ie msg-sender not being the owner of the the token) requires approval

from the owner, while exposing a simple transfer(what, from, to)

interface to comply with ERC20 or 721 standards.

- Oracles

: An oracle, such as a price oracle, is expected to expose a query(asset1, asset2)

interface that returns the data being requested, but it requires up-to-date data

on the prices.

- Claiming rewards

: Rewards (eg from centralized liquidity mining programs) are often dispersed via a merkle tree, where each participant needs to provide a sibling path

to claim their reward. However, most DeFi automation protocols (such as yearn) [expect a simple claim()

interface for claiming rewards](https://forum.aztec.network/t/defi-liquidity-mining-and-capsules/5518).

- Swap hints

: Advanced DEXes may rely on additional hints

computed off-chain to efficiently process a swap. However, swap aggregators (such as 1inch) expect a vanilla swap(asset1, asset2, amountIn, amountOut)

interface for integrating a DEX.

- Minimizing storage use

: Contracts that depend on large blobs of data may opt to not store them in storage to save gas, and rely on users re-submitting them when needed. A good example are Governor or Timelock contracts, where users first broadcast the timelocked action as an event, and then are required to provide it again when executing it, instead of having an execute(action_id)

interface.

These scenarios in Ethereum are solved in a few ways:

1. "Priming" the contract by sending an initial call with the additional data needed for execution (eg setting a token approval via an approve

or a permit

call, or seeding the oracle before it's queried). This additional initial call may be part of the same tx if it's possible to issue a multicall, or must be run in a prior tx if it needs to be sent from an EOA.

1. Breaking the standard interface and requiring all clients to develop custom "adapters" to interact with them, which rarely works.

Note that priming is such a common pattern in Ethereum that a new set of opcodes,TSTORE/TLOAD, were recently

introduced to reduce its cost:

Running a transaction in Ethereum can generate multiple nested frames of execution, each created by CALL (or similar) instructions. Contracts can be re-entered during the same transaction, in which case there are more than one frame belonging to one contract. Currently, these frames can communicate in two ways: via inputs/outputs passed via CALL instructions, and via storage updates. If there is an intermediate frame belonging to another untrusted contract, communication via inputs/outputs is not secure

. Notable example is a reentrancy lock which cannot rely on the intermediate frame to pass through the state of the lock. Communication via storage (SSTORE/SLOAD) is costly. Transient storage is a dedicated and gas efficient solution to the problem of inter frame communication.

EIP1153 lists its own set of use cases also worth reading.

Still, priming, and in particular token approvals, are known to be a flawed pattern. Aside from the UX hurdle of requiring an additional transaction, token pre-approvals are too coarse-grained: it is not possible to approve tokens for a specific action

- instead, a blanket approval is given for anything coming from a specific address.

Scenarios

Composability

is a broad term, and can be applied both to contract and dapp composability. We'll categorize different composability scenarios from the perspective of capsules, and what solutions we could build to address each of them.

In all scenarios, we'll assume we have a chain of contract calls that at some point hits a contract that requires additional contextual data to run, such as an oracle, a reward contract, a swapper, or any from the list above.

graph LR; User((User)) --> Account[Account Contract] --> ContractA --> ContractB --> Contextual[[Contextual]] style Account fill:lightgray

We'll name this contract Contextual

, and the out-of-band data it requires, context

. We'll also assume that Contextual

has the data needed to verify if context

is valid (eg if context

is a sibling path, it can be checked against a stored root).

Single dapp single protocol

In the simplest scenario we have a single dapp that interacts with a single protocol that requires a call to Contextual

.

The solution here is trivial: if Contextual

needs additional context

args to run, we can just modify the interfaces of contracts A, B, and C to pass context

along until it reaches Contextual

, without the need for capsules.

graph LR; User((User)) --> Account[Account Contract] --args,context--> ContractA --args,context--> ContractB --args,context--> Contextual[[Contextual]] style Account fill:lightgray

Nevertheless, capsules here still provide a benefit: if the context

needed is large, passing a large blob of data across multiple calls is costly, since each contract needs to hash all of its arguments to verify them. This translates to more proving time in private-land, and more compute fees in public-land.

graph LR; User((User)) --> Account[Account Contract] --args--> ContractA --args--> ContractB --args--> Contextual Capsule[(Capsule)] --context--> Contextual[[Contextual]] style Account fill:lightgray

A mitigation to this, that doesn't involve capsules, could be allowing the user to decorate an argument as unconstrained

, so it is skipped from the check performed by the application circuit against the injected args hash. However, this only removes the performance hit of hashing: large arguments are still propagated through each app circuit as they are passed along to the next contract.

Learnings from this scenario

- Capsules do not enable new things.
- Capsules offer a performance improvement over args passing if contextual data is large.

Single dapp multiple protocol

Now things start to get interesting. What if our dapp issues a call that needs to go through a 3rd party protocol that relies on a standard interface, meaning we cannot tweak Contextual

’s methods to receive context

?

graph LR; User((User)) --> Account[Account Contract] --> ContractA --> ThirdParty --> Contextual[[Contextual]] style ThirdParty fill:lightgreen style Account fill:lightgray

Here we cannot modify ThirdParty

to include context

args in the call to Contextual

. This is the usual scenario seen on Ethereum, where the solution is to “prime” Contextual

by providing the context

data in advance.

How to inject context

into Contextual

?

We explore three options:

1. A prior call for priming context

requires Contextual

to expose an additional method set_context

. In the example above, we could modify ContractA

to receive the context

as additional arguments, and issue a call to set_context

before passing control onto ThirdParty

.

graph LR; ContractA --context--> Contextual User((User)) --args,context--> Account[Account Contract] -- args, context --> ContractA --args--> ThirdParty --args--> Contextual[[Contextual]] style ThirdParty fill:lightgreen style Account fill:lightgray

contract Contextual { storage context: Context;

```
fn set_context(context) {
   storage.context.write(context);
}

fn foo(args) {
   let context = storage.context.read();
   work(args, context);
}

}
```

On the dapp side of things, the code for issuing this call would be familiar to most devs:

const context = await getContext(); await contractA.methods.foo(args, context).send();

1. Alternatively, we could also prime the context from the account contract directly

leveraging its multicall. This is useful if the initial call is made to ThirdParty

:

graph LR; Account --context--> Contextual User((User)) --args,context--> Account[Account Contract] --args--> ThirdParty --args--> Contextual[[Contextual]] style ThirdParty fill:lightgreen style Account fill:lightgray

const context = await getContext(); await wallet.batch([contextual.method.set_context(context), thirdParty.methods.foo(args)]).send()

1. A capsule

would allow us to remove the need for an additional set_context

method, and instead load the context

data directly from the PXE. This, however, requires additional primitives both on and off chain.

graph LR; User((User)) --args--> Account[Account Contract] --args--> ThirdParty --args--> Contextual[[Contextual]] Capsule[(Capsule)] --context--> Contextual style ThirdParty fill:lightgreen style Account fill:lightgray

contract Contextual { fn foo(args) { let context = oracle.get_context(); work(args, context); } }

const context = await getContext(); await thirdParty.methods.foo(args).send({ capsules: { [contextual.address]: context } });

The first two approaches are similar in that they require no additional primitives. Approach (1) has the added benefit that the requirement for context

data is made explicit

: since ContractA

acts as a facade to the system, it enforces the passing of context

. In approaches (2) and (3) nothing about the function being called hints at a hidden context requirement.

But this comes at the cost of the performance hit of adding an extra function call to ContractA

. It also means that we now need a ContractA

-like facade for every ThirdParty

contract we want to integrate with. In other words, we're just moving the burden of priming for each use case from client-side code to contract-code.

If we remove ContractA

from the picture, both the arguments-based (2) and capsule-based (3) approaches suffer from the same issue on client-side: the client code needs to explicitly prime the contract to be called. Capsules have the added benefit that they do not require an additional external function in the contract (less contract code is usually better).

Transient storage

Storage-based approaches (1 and 2) potentially have the additional issue of extra DA costs. However, we can make use of note squashing in the kernels, where a note that's created and nullified in the same transaction is not added to the tree. We could design a data type in aztec-nr that can be just initialized once and read once, which if done on the same tx, emulates private transient storage. This still incurs in extra proving cost for processing the transient note, even if it ends up not being emitted.

Another approach would be generalizing packedArgs

to a generic key-value transaction-scoped data store (which is almost what it is today), and using that for private transient storage.

Multiple pieces of context

In this scenario we assumed that Contextual

required a single piece of context

data for its execution, but it's possible it requires more. An example could be an oracle that's queried for multiple asset prices, or a transaction that involves multiple transfers of the same token from different owners.

This requires identifying each context

somehow. If using storage as in (1) or (2), using a mapping works simply enough:

contract Contextual { storage contexts: Map;

```
fn set_context(key, context) {
    storage.context.at(key).write(context);
}

fn foo(args) {
    let key = derive_key(args);
    let context = storage.context.at(key).read();
    work(args, context);
}

}
```

const context1 = await getContext(key1); const context2 = await getContext(key2);

await wallet.batch([contextual.method.set_context(key1, context1), contextual.method.set_context(key2, context2), thirdParty.methods.foo(args)]).send()

This means that capsules may also need an identifier, to emulate the API above:

contract Contextual { fn foo(args) { let key = derive_key(args); let context = oracle.get_context(key) work(args, context); } }

const context1 = await getContext(key1); const context2 = await getContext(key2);

await thirdParty.methods.foo(args).send({ capsules: { [contextual.address]: { [key1]: context1, [key2]: context2 } });

Up until this point, capsules don't seem to offer too much of an advantage over priming a contract using an external method and transient-like storage, aside from the performance gained from not requiring the additional priming function calls.

Capsules in public-land

Note that capsules, as defined above, can also be implemented in public-land. The tx object broadcasted to the network would require capsule data to be included, and the AVM would need a new opcode to load a specific capsule, or to prove it was not present in the tx payload.

On the other hand, storage-based priming works out of the box, but the lack of transient storage in public would make it expensive to use.

Learnings from this scenario

- Capsules do not enable new things over priming.
- Capsules offer a performance improvement due to less function calls.
- Capsules require less contract code and reduces their surface.

Single dapp with dynamic context

All approaches on the previous scenario have something in common: the dapp needs to know exactly what context will be required to compute it in advance. This is not always trivial.

Let's pretend we have a private DEX that requires oracle prices for exchanges (we may need to suspend the disbelief that we can implement a DEX like this privately, but bear with us for the sake of the example). The DEX has logic written in Noir for routing the trade across different pairs. This means that the price oracle will be hit with queries that are only known as the routing logic is executed, and not known in advance to the dapp

.

graph LR; User((User)) --args--> Account[Account Contract] --args--> DEX --arg1--> Contextual[[Price Oracle]] DEX --arg2--> Contextual DEX --arg3--> Contextual Capsule[(Context)] --???--> Contextual style Account fill:lightgray

This breaks all three approaches explored in the previous scenario. Since the dapp does not know in advance the context, it cannot provide it initially as arguments to a facade contract, nor it can prime them via a prior call, nor it can supply them in advance as a capsule.

A makeshift solution here is trying to replicate the contract's logic in client-code, so the dapp can emulate the contract and collect the required pieces of context, and supply them in advance hoping its simulation will match the execution path of the contract.

Alternatively, apps may brute-force their way into discovering missing data by triggering simulations, catching errors, and use the errors as guidelines on which data pieces are missing - assuming errors are descriptive enough.

However, we can rethink capsules as callbacks that reach all the way back to a dapp. This would allow a contract to use the dapp as an oracle, and not just the PXE. From the contract's perspective, this is no different than the previous capsules approach. We can even have generic versions of get_context

so the contract can pass additional in the callback:

// Library code with generic callback oracle call // We define a "topic" to support multiple request types through the same callback interface

[oracle(callback)]

fn callback(topic: Field, request: [Field; T]) -> [Field; RESPONSE_SIZE] {}

// Library code that provides strong typing around the generic callback oracle interface unconstrained fn get_context(key) -> Context { let topic_id = hash("get_context"); Context::deserialize(callback(hash("get_context"), [key])) }

// Contract code (pretty much same as before) contract Contextual { fn foo(args) { let key = derive_key(args); let context = get_context(key) work(args, context); } }

await dex.methods.swap(assetIn, assetOut, amount).send({ callbacks: { get_context: (key) => getContext(key) } })

This enables a new development pattern for private functions, where contracts can operate dynamically in full collaboration with the dapp. Needless to say, this approach only works for private-land.

API design and security implications

This breaks the API in a very annoying way: sending a transaction from the dapp to the wallet is no longer a single request-response operation, but rather may require several back-and-forths between dapp and wallet. Supporting this would require changing the return type of a simulateTransaction

operation of the PXE JSON-RPC API, so it returns either a simulated transaction or a request for more data.

We can hide this complexity in aztec-js as shown above, but the fact remains that the low-level API is now more complex, and wallets need to keep simulations on hold as they wait for the dapp to provide continuation data.

Alternatively, wallets could remain stateless, and just re-simulate the entire tx until the point in which they require new data. This is similar to the approach of having the dapp just brute-force one data request after another, but surprisingly this is not too bad an idea.

Security-wise, this does not seem to open new security issues. Dapps were already able to access user private data via queries, which they could exfiltrate if they were malicious. A dapp should not be allowed to initiate a tx simulation unless it has been given permission by the user in advance, same as it should not be allowed to query private user data in a contract. However, it could be argued that sensitive data could be used throughout a transaction (such as a secret hash preimage) and callbacks increase the risk of it being exfiltrated.

Authwits as callbacks

AuthWits fall very well into this "dynamic context" scenario, since a dapp initiating a transaction does not known a priori which authorizations will be required throughout its execution, and needs a way to collect these requests to present them to the user.

It should be possible to implement AuthWiths as callbacks, with a distinguished topic that's always intercepted by the PXE instead of sent back to the dapp. Also, AuthWits as they stand today suffer from a limitation: the user is presented with a hash to approve, with no knowledge of the preimage. A more flexible callbak interface may allow a contract to pass additional information to the PXE on what is being signed.

Learnings from this scenario

- Capsules can be a lot more fun than a key-value store and allow dapps to act as oracles that provide responses to requests computed on the spot. This simplifies client-side developer experience (vs storage-based priming or key-value capsules), as dapp devs no longer need to guess the contextual data requirements for a transaction in advance.

Multiple dapps

While the previous scenario opens up new patterns, it also comes with the risk that it tightly couples a contract to a dapp. In order to be able to interact with a contract, either directly or indirectly, we now also need off-chain code to support it, which is an apparent risk to composability.

However, this is the case in every single approach where we need specific data to operate a contract. In the example for requiring a specific sibling path for executing a claim

, it doesn't matter whether we pass it directly as an argument, we prime it via a prior call, we supply it as a capsule, or we return it on the fly from a callback. The fact remains that we need the dapp triggering the tx to know how to construct this sibling path. This is important because it highlights that we are not compromising composability by introducing a capsule-like pattern.

Still, let's analyze what happens in the event where the dapp triggering the transaction has no knowledge of the context

required to run Contextual

. In other words, what happens if now Contextual

is the contract from a third party dapp.

graph LR; User((User)) --args--> Account[Account Contract] --args--> ContractA --args--> Contextual[[Contextual]] Capsule[(???)] --context--> Contextual style Contextual fill:lightgreen style Capsule fill:lightgreen style Account fill:lightgray

This scenario is not solvable today in Ethereum nor Aztec. It depends on the user to manually priming the Contextual

contract in a transaction from a dapp that "knows" how to do it, and then navigating to the dapp they were interacting with and running a second transaction.

graph LR; User1((User)) --context--> Account1[Account Contract] --context--> Contextual[[Contextual]] User2((User)) --args--> Account2[Account Contract] --args--> ContractA --args--> Contextual[[Contextual]]

style User1 fill:lightgreen style Contextual fill:lightgreen style Account1 fill:lightgray style Account2 fill:lightgray

The ideal scenario here would be for the PXE to know that the request for context

coming from Contextual

should be served by a dapp different

than the one in context. We propose to handle this via plugins that register themselves for handling specific topics across specific contracts.

Plugins

We define a plugin as a gadget that can be installed or registered in a PXE, with metadata that lets the PXE filter which plugin should serve each callback request.

plugin_manifest: handles: topics: - get_sibling_path contracts: - 0xabcd - 0x1234

Plugins would be installed by a dapp when the user visits the dapp. In our example above, we'd expect that, if the dapp is interacting with Contextual

because of some specific settings of the user, then the user has at least once visited a dapp (the green one!) that allowed them to interact with Contextual

. Upon visiting, the dapp would issue a register_plugin

call to the PXE's API, that installed a handler for get_context

calls from Contextual

- with the user's authorization, of course.

HTTP plugins

The simplest and most flexible form of plugin we came up with is an http endpoint. Apps would expose an endpoint that the PXE would query to fulfill callback requests based on plugin registrations. This means that a transaction triggered from a dapp could result in the PXE calling a different dapp (the green one) to gather specific context

data needed to power a (green) contract. All without the initial dapp requiring any knowledge of this. This allows multiple dapps to collaborate transparently in assembling a transaction.

This approach does feel riskier from a security standpoint, since we're loading data from arbitrary sources during an execution. Not just that, but we're also potentially exposing sensitive data (the callback request) to these API endpoints, and losing privacy by revealing steps in the private execution. It's also unclear how much users would actually pay attention to the authorization dialog, potentially shooting themselves on the foot by allowing a malicious dapp to register itself as the handler for critical topics.

Discarded: isolated plugins

A more secure form of plugin we discussed is a piece of code that would run sandboxed on the wallet. But this is potentially more dangerous from a security standpoint if sandboxing is not done carefully, and requires siginificant engineering effort for managing plugins and keeping them up to date.

Furthermore, isolated pieces of code are severely limited as they cannot access external data, such as real-time price oracles. All in all, it's unclear if this option has any benefits over an unconstrained Noir function with an efficient ACVM.

Discarded: re-routing callbacks from the dapp

An alternative approach we explored was having the PXE always call back to the current dapp, and having the dapp call out to other dapps based on the topic of the callback if they didn't know how to handle it.

However, this requires that each dapp keeps an up-to-date registry of topics to apps, or worse, that there is a centralized one that all dapps delegate to. It also allows the current dapp to intercept and tweak requests meant for other dapps.

Learnings from this scenario

- Capsules can be really fun by allowing multiple dapps to interact transparently when assembling a transaction, providing data on-the-fly as needed, enabling composability both on and off chain. However, this involves security and privacy tradeoffs.

Dealing with unconstrained data

Another critique of capsules is that they inject unconstrained data into a circuit, which could lead to security issues on apps that use them. This is a common problem to all oracle data accessed directly from a circuit. We see two ways around that:

1. Packaging specific capsule use-cases in a constraining function. We should encourage devs to never expose the capsule unconstrained call directly to a contract, but rather wrap it in a function that asserts its correctness. AuthWits are the easiest example, where the same function that requests the signed message is responsible for checking the signature. We could further enforce this with a linter in the future, but this relies mostly on the patterns we push for building in Noir. As another example, capsules that return sibling paths can also easily be verified on the spot by providing the root to check it against:

// General oracle interface from aztec-nr

[oracle(callback)]

fn callback(topic: Field, request: [Field; T]) -> [Field; RESPONSE_SIZE] {}

// Internal method of a sibling-path capsule library unconstrained fn oracle_get_sibling_path(index) -> [Field; N] { callback(hash("get_sibling_path"), [index]) }

// External interface for a sibling-path capsule library pub fn get_sibling_path(root, index) -> Context { let sibling_path = oracle_get_sibling_path(index); assert(verify_sibling_path(sibling_path, root)); sibling_path }

1. Leveraging tainted data types or unsafe blocks. Having unconstrained data find its way silently into constrained code is common in Noir, and not exclusive to capsules. There are ongoing discussions on using unsafe

-like blocks to identify where unconstrained data is being used, or wrapping data in an Unsafe

type that forces consumers to explicitly assert its correctness to unwrap it. Again, this discussion is more general and exceeds capsules, as the underlying problem is more general as well.

End to end example of HTTP plugins

Let's go over an example to make this all a bit less abstract.

Say we have a DeFi aggregation protocol that collects fees accrued in other DeFi protocols, and swaps those tokens in some DEX for one which the end user has selected as their withdrawal token (not terribly unlike how Yearn works).

The aggregator likely expects some form of standard interface for claiming yield (such asEIP 4626), since it interacts with so many protocols, and may know the details of one or two DEXes that it uses to conduct the swaps. DEXes are typically composed of multiple liquidity pools, each with their own internal pricing logic.

Now, what happens if claiming the tokens in one of these protocolsrequires external data, such as a Merkle sibling path? Or if the liquidity pool that offers the best price requires a trusted CEX to act as a price oracle by signing a message for a pair price? This is similar to themultiple dapps scenario: for the transaction to be processed using pre-seeded capsules, the aggregation dapp would have to:

1. know where to fetch the sibling paths from
2. know which liquidity pool will be used and how to get the signed pair price
3. know that these two protocols will be interacted with, and prepare the data in advance (by e.g. simulating the transaction)

By instead using HTTP plugins, the PXE would halt execution as the oracles for querying the sibling path and pair price are called, and request the required data from the corresponding servers registered by the plugins. The circuits then verify that the data is correct (the path resolves to the root, the price is correctly signed) and carry on execution as if nothing had happened. The developers of the dapp are unaware of this happening behind the scenes, and no extra engineering effort is required on their side to integrate with these two protocols (other than instructing the user to install the plugins).

graph LR; DeFiCapsule[(DeFi backend)] --sibling path--> DeFi User((User)) --> Account[Account Contract] --> Aggregation[Aggregation Protocol] --claim() --> DeFi[Defi Protocol] Aggregation --swap() --> Dex[DEX] --swap() --> DexPool[Liquidity Pool] DexCapsule[(CEX backend)] --signed pair price--> DexPool style Aggregation fill:lightgreen style DeFi fill:cyan style DeFiCapsule fill:lightblue style Dex fill:orange style DexPool fill:orange style DexCapsule fill:pink

Code snippets

What does the code look like for some of the components above?

Aggregation Protocol

The dapp of the aggregation protocol simply fires a transaction to its contract, which in turn calls to the DeFi protocol and DEX contracts without requiring to pass any contextual information.

await aggregationProtocolContract.methods.aggregate(defiProtocolAddress, dexAddress, assetIn, assetOut,).send().wait();

contract AggregationProtocol { fn aggregate(defi_protocol: AztecAddress, dex: AztecAddress, asset_in: AztecAddress, asset_out: AztecAddress) { let claimed = DefiProtocol::at(defi_protocol).claim(asset_in).call(); let swapped = DEX::at(dex).swap(asset_in, asset_out, claimed).call(); Token::at(asset_out).transfer(context.msg_sender(), swapped); } }

Conclusion

We recommend including capsules to private execution

in one of two flavors: either a contract-scoped key-value pre-seeded store per transaction (option 3), or implementing full callbacks to the dapp to support dynamic context data. The former is similar to what we have today, but replacing a stack with a key-value mapping. The latter would be interesting when it comes to see what developers build with it, but it requires a more thorough security analysis before enabling it, as well as additional engineering effort.

A good full-fledged example

that shows the need for capsules and expores callbackscan be found here.

We defined context

data as arguments to an execution that are not part of its standard interface, but additional data that is specific to its implementation. We then analyzed three main approaches for passing context

data into a contract:

- Changing the interface of the contract on-chain to add the context

on every call is not acceptable, as it breaks standardisation and composability.

- Adding an external function to prime the contract using some form of transient storage is viable, but increases proving time, and still requires the dapp developer to be aware of the additional context

that needs to be primed.

- Capsules have the same cognitive overhead as priming, but does not pollute on-chain code, and makes it feasible to inject data dynamically as requested during the simulation. Capsules are also more efficient as they reduce the amount of data hashed and passed from one circuit to another.

All in all, capsules move the context

from the contract's on-chain interface to an off-chain one. Reducing on-chain complexity is always preferrable, both for performance and security reasons.