Multi-Verifiers as a Hedge Against Validating Bridge Implementation Vulnerabilities

Thanks to Justin Drake and Bartek Kiepuszewski for their comments.

Last year, Vitalik Buterin has presented the idea of Mutli-Provers. The idea of Multi-Provers boils down to using two or more state transition correctness enforcement mechanisms (STCEM from hereon) as a hedge against vulnaribilies that could potentially be exploited in a STCEM. For example, the validating bridge may require assertions from two distinct STCEMs (e.g. a fraud proof-based STCEM and a zero-knowledge proof-based STCEM) before finalizing the corresponding state root. Additionally, Justin Drake proposed a concrete instantiation of a 2-of-2 Multi-Prover that utilizes a combination of a zero-knowledge proof-based STCEM and a TEE-based STCEM.

While Multi-Provers could potentially play a significant role in hardening the security properties of rollups, they are not the only cause for concern for the developers and users of validating bridge-based protocols. As is evident from the Rekt dashboard, the vast majority of hacks to date happened due to the exploitation of smart contract vulnerabilities. While the complexity of the established STCEM's means that the surface area for non-smart contract exploits is quite large, the increased complexity of the validating bridge implementations means that potential smart contract exploits remain a major concern.

In this post, I outline the design of a redundant validating bridge, otherwise known as a Multi-Verifier

.

High-Level Overview

The Multi-Verifier is comprised of four distinct components:

1. a publicly-accessible API through which the users/rollup operators are capable of interacting with a bridge, called EntryPoint

,

1. a Solidity implementation of the validating bridge, called VerifierSolidity

(note that VerifierSolidity

may be instantiated as a suite of smart contracts rather than one),

1. a Vyper implementation of the validating bridge, called VerifierVyper

(note that VerifierVyper

may be instantiated as a suite of smart contracts rather than one) and

1. a shared storage, creatively titled SharedStorage

.

The Multi-Verifier is designed to preserve the safety of the rollup as long as one of the verifiers is correctly implemented. The design does not protect against vulnerabilities resultant from the incorrect protocol specification.

Detailed Overview

NOTE: the Multi-Verifier design outlined below is based on a zero-knowledge rollup validating bridge design, through the implementation can be trivially optimized to work with an optimistic rollup validating bridge instead.

EntryPoint

EntryPoint

is a publicly-accessible API though which users/rollups operators are capable of interacting with the bridge. The EntryPoint

implementation is on purpose kept as minimal as possible to avoid unnecessary complexities that may result in a greater surface area for potential vulnerabilities. The concrete instantation of EntryPoint

can either be implemented in Solidity or Vyper.

EntryPoint

has two responsibilities:

1. call the corresponding functions of the two concrete instantiations of the validating bridge,
2. verify that the values returned by the corresponding functions of the respective concrete instantiations of the validating bridge are equal and have been indeed written to SharedStorage

.

Below is an abstract specification of EntryPoint

written in pseudo-code:

function appendBatch(batch) { (flagSolidity, storedDataSolidity) = VerifierSolidity.appendBatch(batch) require(flagSolidity == true)

(flagVyper, storedDataVyper) = VerifierVyper.appendBatch()
require(flagVyper == true)

require(ShareStorage.checkStorage(APPENDBATCH, storedDataSolidity, storedDataVyper) == true)

}

function finalizeBatch(validityProof, publicInputs) { (flagSolidity, storedDataSolidity) = VerifierSolidity.finalizeBatch(validityProof, publicInputs) require(flagSolidity == true)

(flagVyper, storedDataVyper) = VerifierVyper.finalizeBatch(validityProof, publicInputs)
require(flagVyper == true)

require(ShareStorage.checkStorage(FINALIZEBATCH, storedDataSolidity, storedDataVyper) == true)

}

function deposit(message) { (flagSolidity, storedDataSolidity) = VerifierSolidity.deposit(message) require(flagSolidity == true)

(flagVyper, storedDataVyper) = VerifierVyper.deposit(message)
require(flagVyper == true)

require(ShareStorage.checkStorage(DEPOSIT, storedDataSolidity, storedDataVyper) == true)

}

function withdraw(message) { (flagSolidity, storedDataSolidity) = VerifierSolidity.withdraw(message) require(flagSolidity == true)

(flagVyper, storedDataVyper) = VerifierVyper.withdraw(message)
require(flagVyper == true)

require(ShareStorage.checkStorage(WITHDRAW, storedDataSolidity, storedDataVyper) == true)

}

VerifierSolidity

VerifierSolidity

is a concrete instantiation of the rollup validating bridge implemented in Solidity. Since VerifierSolidity

is the first validating bridge instantiation called by EntryPoint

, the implementation is permitted to mutate the state of SharedStorage

. VerifierSolidity

is expected to return the values written to SharedStorage

.

VerifierVyper

VerifierVyper

is a concrete instantiation of the rollup validating bridge implemented in Vyper. Since VerifierVyper

is the second validating bridge instantiation called by EntryPoint

, the implementation is permitted to read from the state of SharedStorage

. VerifierVyper

is expected to return the values read from SharedStorage

that would have otherwise written if VerifierVyper

was the first validating bridge instantiation to be called.

SharedStorage

SharedStorage

, as the name suggests, is the shared storage for two concrete instantiations of the validating bridge. The write access is permitted to VerifierSolidity

only, whereas the read access is permissionless. The SharedStorage

implementation is on purpose kept as minimal as possible to avoid unnecessary complexities that may result in a greater surface area for potential vulnerabilities. The concrete instantation of SharedStorage

can either be implemented in Solidity or Vyper.

Below is an abstract specification of SharedStorage

written in pseudo-code:

function read(type){ if(type == APPENDBATCH) { ... } else if(type == FINALIZEBATCH) { ... } else if(type == DEPOSIT) { ... } else if(type == WITHDRAW) { ... } }

function write(type, data){ require(msg.sender == addressVerifierSolidity)

```
if(type == APPENDBATCH) {
    ...
} else if(type == FINALIZEBATCH) {
    ...
} else if(type == DEPOSIT) {
    ...
} else if(type == WITHDRAW) {
    ...
}

}
```

function checkStorage(type, dataSolidity, dataVyper){ if(type == APPENDBATCH) { ... } else if(type == FINALIZEBATCH) { ... } else if(type == DEPOSIT) { ... } else if(type == WITHDRAW) { ... } }

Conclusion

While Multi-Verifiers increase the cost of bridging, they increase the redundancy of the protocol, enabling developers and users to interact with validating bridge-based protocols in a more secure manner. For maximal resilience, a Multi-Verifier should be paired with a Multi-Prover.