Overcoming Reentrancy and Circular Call Challenges in Polkadot Smart Contracts

A Journey with Protofire

By Luca Auet and Gabriel Gonzalez

Resume

At Protofire, we are constantly pushing the boundaries of what's possible with blockchain technology. Recently, we embarked on a journey to develop a multisig smart contract using Rust and the Ink! smart contract language for the Polkadot Contracts Pallet. However, we encountered a couple of blockers that challenged our understanding of reentrancy and circular calls in smart contracts. This article will delve into the problems we faced and how we overcame them.

The Challenges

Reentrancy and Circular Calls in Smart Contracts

Reentrancy is a common issue in smart contracts where a contract calls another contract before it resolves its state. This can lead to unexpected behavior and potential security vulnerabilities. In our case, we needed to allow for reentrancy to enable our contract to change its own state - such as changing an account or the threshold.

We also encountered a problem with circular calls between multiple contracts. In our scenario, we had two contracts, A and B. We could successfully make cross-contract calls from A to B and from B to A. However, we faced a problem when trying to make a circular cross-contract call from A to B and then back to A. This led to a failed transaction, indicating a potential limitation or issue in the ink! framework.

The Solutions

Leveraging the Flush Trait and Understanding Circular Calls

After some research and discussions on the Substrate StackExchange, we found potential solutions to our problems.

For the reentrancy issue, we needed to manually reload the contract state after a reentrant call to ensure that it was up-to-date. This was achieved by using the Flush trait from the OpenBrush library. This code checks if the transaction allows for reentry and if the transaction is being called by the same contract. If both conditions are met, it reloads the contract state using the self.load() function. This ensures that the contract state is up-to-date after a reentrant call.

```
if tx.allow_reentry && tx.address == self.env().account_id() {
    self.load();
}
```

However, this solution came with a trade-off. To use the Flush trait from openbrush, we had to downgrade our ink! version to 4.1.0. This was a necessary step to ensure the security and consistency of our contract.

For the circular call issue, we realized that ink! might have limitations when it comes to circular calls between multiple contracts. Our tests showed that a circular call from A to B and then back to A failed, while a call from A to B and then to a third contract C was successful. This suggests that developers need to be aware of potential limitations when designing their contracts and consider alternative approaches when necessary.

Conclusion

A Step Forward in Our Blockchain Journey

Overcoming these reentrancy and circular call challenges was a significant milestone in our journey at Protofire. It not only helped us progress with our multisig smart contract but also deepened our understanding of the intricacies of smart contract development.

We hope that sharing our experience will help other developers facing similar challenges. As we continue to explore the possibilities of blockchain technology, we remain committed to sharing our learnings and contributing to the growth of this exciting field.

Remember, the path to innovation is often riddled with challenges. But as we've learned at Protofire, these challenges are just opportunities for learning and growth.