Lesson 12 - Security and MEV

Security and Best Practices

The field is quite new, so there are not many resources or known exploits.
However many of the best practices that apply to solidity contracts also apply to cairo contracts.

Follow the guidelines from Open Zeppelin and Nethermind

Security Guidelines from Open Zeppelin

See Guidelines

Safe Math

If we use the maths libraries such as Uint256 we get checked methods when doing arithmetic.
A safe math library is also available
See the Starket-DAI bridge audit for an example where the leack of a Uint256 check was a vulnerability.

Initializable Library

This mimics the role of a constructor

```
from openzeppelin.security.initializable.library import Initializable

@external
        func foo{
        syscall_ptr : felt*,
        pedersen_ptr : HashBuiltin*,
        range_check_ptr
    }(){
    let (initialized) = Initializable.initialized();
    assert initialized = FALSE;

    Initializable.initialize();
    return ();
        }
```

The recommended pattern with Initializable is to include a check that the Initializable state is `False` and invoke `initialize` in the target function.

Pausable

The Pausable library allows contracts to implement an emergency stop mechanism.
To use the Pausable library, the contract should include `pause` and `unpause` functions (which should be protected).
For methods that should be available only when not paused, insert `assert_not_paused`.
For methods that should be available only when paused, insert `assert_paused`.
For example:

```
from openzeppelin.security.pausable.library import Pausable

@external
func whenNotPaused{
        syscall_ptr: felt*,
        pedersen_ptr: HashBuiltin*,
        range_check_ptr
    }() {
    Pausable.assert_not_paused();

    # function body
    return ();
}

@external
func whenPaused{
        syscall_ptr: felt*,
        pedersen_ptr: HashBuiltin*,
        range_check_ptr
    }(){
    Pausable.assert_paused();

    # function body
    return ();
}
```

Re entrancy

Similar to the fallback function in Solidity there is a default entry point in Cairo
See Docs

```
func __default__{
    syscall_ptr: felt*,
    pedersen_ptr: HashBuiltin*,
    range_check_ptr,
}(selector: felt, calldata_size: felt, calldata: felt*) -> (
    retdata_size: felt, retdata: felt*
) {
    ... function body
    return (retdata_size=retdata_size, retdata=retdata);
}
```

Follow the Check Effects Interactions pattern to avoid re entrancy or use a re entrancy guard

Re entrancy guards

We don't have modifiers in cairo, so the guard is a little different to what we would see in solidity. We have a start and end around the function body.

```
from openzeppelin.security.reentrancyguard.library import ReentrancyGuard

func test_function{
        syscall_ptr : felt*,
        pedersen_ptr : HashBuiltin*,
        range_check_ptr
    }(){
   ReentrancyGuard._start()
   # function body
   ReentrancyGuard._end()
   return ()
}
```

Access Control

Open Zeppelin provide a range of contracts to implement access control, from the simple Ownable, to the finer grained Access Control

Audit Examples

Starknet - DAI bridge audit

General Best Design Practices

1. Split the contract into a logic file and a contract file
2. If using the `with_attr error_message ..` make sure only one error can occur within the code block.
3. Use namespaces to avoid storage collision, see Lesson 9 notes.
4. Since inheritance is not possible we use composability

The role of hints

See background article

If we have hints in our code, we should have asserts to check that the value the hint produces is valid.
If we used hints in our contracts we could have a problem if we didn't adequately constrain the contract, since we would be unaware who is executing the hint, they could insert a malicious value.
In summary if we have hints, there should always be some constraints / asserts to test that the hint output is correct.

Security tools

Amarna Static Analysis tool.
Linter from BibliothecaForAdventurers

MEV

Background Resources

See SoK : Front Running Attacks on Blockchain
Useful Overview - MEV Wiki

"Front-running is a course of action where someone benefits from early access to market information about upcoming transactions and trades"

From https://hackmd.io/@flashbots/quantifying-REV
Maximal (formerly Miner) Extractable Value is the value that can be extracted from a blockchain by any agent without special permissions. Considering this permissionless nature, any agent with transaction ordering rights will be in a privileged position to perform the extraction.

There are features of Ethereum (and other blockchains) that allow front running

1. All transactions are available in a public mempool before they are mined
2. All transaction data is public
3. Transactions can be cloned

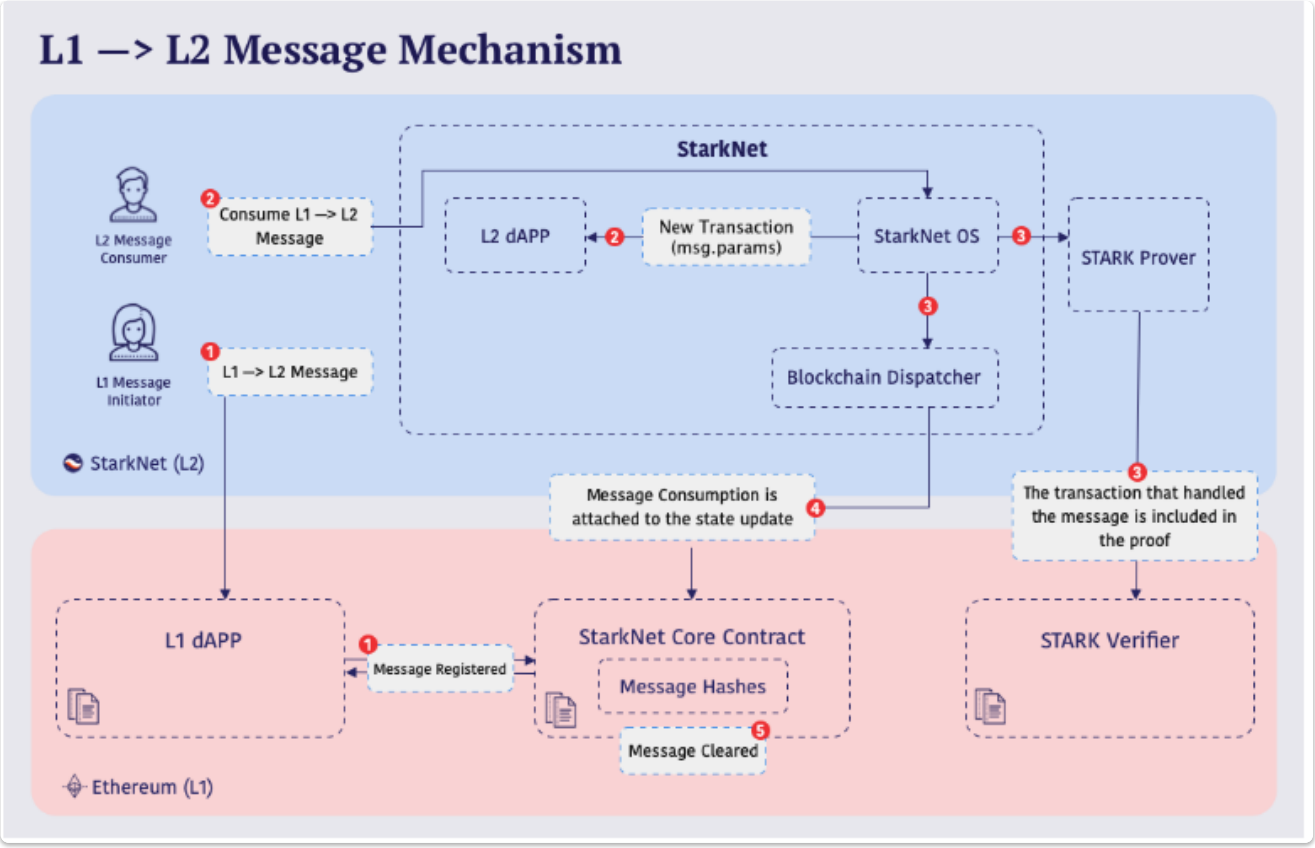
A Taxonomy of Front-running Attacks

- Front Running: Monitoring mempool for profitable trades/TXs then submitting them but with higher gas fees.
- Back Running: monitoring of a mempool to execute a transaction immediately after a pending target transaction.
- Sandwich Attack: Combination of front & back running to sandwich a trade.

MEV on L2s

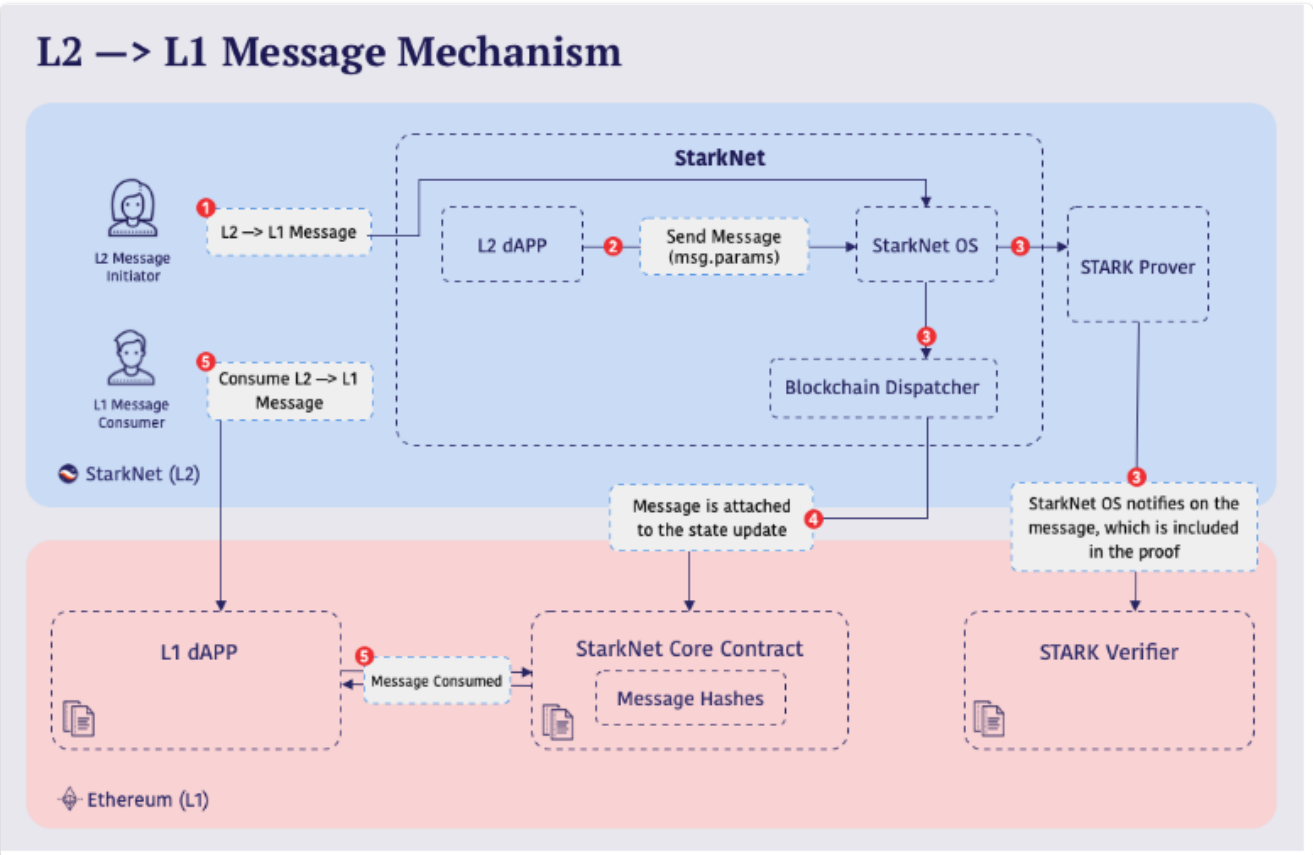
From article from Open Zeppelin

StarkNet's approach to L1<>L2 interoperability is through messages. There's two kind of messages:



A: L1 → L2 messages

1. L1 contract calls `send_message()` on the StarkNet L1 contract, passing target and calldata
2. L2 sequencer consumes the message and invokes the function on the target L2 contract (implemented with the `l1_handler` modifier)



B: L2 → L1 messages

1. L2 contract calls system function `send_message_to_l1()`, registering the message on the sequencer's state
2. L2 sequencer settles state on L1, storing the message on the StarkNet L1 contract waiting to be consumed
3. The L1 target contract can now call `consumeMessageFromL2()` of the StarkNet L1 contract

MEV Possibilities

- L2 sequencers can mev any regular L2 and `l1_handler` transaction
- Nothing currently forces L2 sequencers to invoke `l1_handler` functions let alone in a particular order. This makes `A2` a focal point for mev
- Conversely, this renders `A1` a no-MEV zone since txs don't get orderered until the L2 sequencer picks them up on the other side
- No "immediate" MEV can be extracted from `B2` (L2 → L1 state settlement) since it has no immediate side effects on any layer
- Since the L2 → L1 lifecycle is closed by an async L1 transaction, it would make sense to have some sort of keeper/worker/scheduler network automatically executing them for a profit, creating a mevvable market

In summary

- `A1`: can be mevved by the L1 block proposer
- `A2`: can be mevved by the L2 sequencer
- `B1`: can be mevved by the L2 sequencer
- `B2`: can be mevved by the L1 block proposer
- `B3`: can be mevved by the L1 block proposer

Multiple execution layers make MEV extraction a more complex landscape. Considering how complex single-layer or cross-chain existing MEV strategies are, adding multiple ordering points and asynchrony opens the door for a combinatorial amount of new strategies, markets, and actors.

Some work on front running mitigation strategies have been presented already like this proof of concept of a commit-reveal + timelock system by Yael Doweck and team from StarkWare.