Supplementary Information for "Security Analysis of Smart Contract Migration from Ethereum to Arbitrum"

1 Arbitrum Cross-Chain Messaging

(1) Issue: Outdated off-chain data was obtained by the inactive sequencer

Case:

When obtaining off-chain data from an oracle, real-time requirements are crucial. If the sequencer goes down and transactions cannot be immediately executed, the contract will not be able to return accurate and real-time off-chain data.

Taking the GLPOracle.sol[1] contract as an example, we first introduce the main functionality of this contract. The getPrice() function is used to calculate the GLP/USD price. The constructor of the contract accepts two parameters: _manager and _ethFeed, representing the addresses of the GLPManager contract and the Ethereum price data source, respectively. The contract includes two functions: getPrice() and getEthPrice().

The getEthPrice() function retrieves the latest Ethereum price by calling ethUsdPriceFeed.latestRoundData() and performs validations for data expiration and negativity. ethUsdPriceFeed is an instance of an oracle contract used to provide external data services, and latestRoundData() is a public function of this contract that returns the latest ETH/USD price.

The getPrice() function first calls manager.getPrice(false) to obtain the current price of the GLP token, and then calls getEthPrice() to retrieve the latest ETH/USD price. The GLP/USD price is calculated as

$$\frac{\text{manager.getPrice(false)}}{\text{getEthPrice()} \times 10^4}$$

where the GLP token price is divided by the ETH/USD price and then multiplied by 10^4. This is done to convert the price unit of ETH to USD in order to obtain the price unit of GLP.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

import {Errors} from "../utils/Errors.sol";
import {IOracle} from "../core/IOracle.sol";
import {IGLPManager} from "./IGLPManager.sol";
import {AggregatorV3Interface} from "../chainlink/AggregatorV3Interface.sol";

contract GLPOracle is IOracle {
/// @notice address of gmx manager
IGLPManager public immutable manager;
/// @notice ETH USD Chainlink price feed
AggregatorV3Interface immutable ethUsdPriceFeed;
/**
@notice Contract constructor
@param _manager address of gmx vault
@param _ethFeed address of eth usdc chainlink feed
*/

constructor(IGLPManager _manager, AggregatorV3Interface _ethFeed) {
manager = _manager;
ethUsdPriceFeed = _ethFeed;
}

```



```
/// @inheritdoc IOracle
function getPrice(address) external view returns (uint) {
return manager.getPrice(false) / (getEthPrice() * 1e4);
}

function getEthPrice() internal view returns (uint) {
(, int answer,, uint updatedAt,) = ethUsdPriceFeed.latestRoundData();
if (block.timestamp - updatedAt >= 86400)
revert Errors.StalePrice(address(0), address(ethUsdPriceFeed));
```

```
if (answer <= 0)
revert Errors.NegativePrice(address(0), address(ethUsdPriceFeed)); return uint(answer);
}
}
```

Security Risks:

There is a potential vulnerability when executing this contract on Arbitrum. When the contract reaches the statement ethUsdPriceFeed.latestRoundData(), it needs to access the oracle through the sequencer to obtain off-chain price data. If the sequencer goes down at this moment, the statement cannot be executed immediately. When the sequencer reconnects and executes this statement, it will retrieve outdated price data, which may be higher or lower than the actual price. Attackers can exploit the price difference between the actual price and the outdated price for profit.

Exploitation of this vulnerability: Assuming a user borrows using GLP tokens as collateral, if the sequencer goes down and the reconnected sequencer retrieves an outdated price higher than the actual price, the user can obtain better borrowing conditions. If the outdated price is lower than the actual price, the user can avoid liquidation.

The above vulnerability has been tested and confirmed to exist. The test code can be found at[2].

Risk Avoidance Methods:

To address the aforementioned vulnerability, we suggest querying the Chainlink L2 Sequencer Uptime Feeds to determine the operational status of the sequencer[3].

```

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;
import {Errors} from "../utils/Errors.sol";
import {IOracle} from "../core/IOracle.sol";
import {IGLPManager} from "./IGLPManager.sol";
import {AggregatorV3Interface} from "../chainlink/AggregatorV3Interface.sol";

contract GLPOracle is IOracle {
/// @notice address of gmx manager
IGLPManager public immutable manager;
/// @notice ETH USD Chainlink price feed
AggregatorV3Interface immutable ethUsdPriceFeed;
/// @notice L2 Sequencer feed
AggregatorV3Interface immutable sequencer;
/// @notice L2 Sequencer grace period
uint256 private constant GRACE_PERIOD_TIME = 3600;

constructor(IGLPManager _manager, AggregatorV3Interface _ethFeed, AggregatorV3Interface _sequencer) {
manager = _manager;
ethUsdPriceFeed = _ethFeed;
sequencer = _sequencer;
}

function getPrice(address) external view returns (uint256) {
if (!isSequencerActive()) revert Errors.L2SequencerUnavailable();
return manager.getPrice(false) / (getEthPrice() * 1e4);
}

function getEthPrice() internal view returns (uint256) {
(, int256 answer,, uint256 updatedAt,) = ethUsdPriceFeed.latestRoundData(); if (block.timestamp - updatedAt >=
86400) {
revert Errors.StalePrice(address(0), address(ethUsdPriceFeed));
}
```



```
if (answer <= 0) {
revert Errors.NegativePrice(address(0), address(ethUsdPriceFeed));
}
return uint256(answer);
}

function isSequencerActive() internal view returns (bool) {
(, int256 answer, uint256 startedAt,,) = sequencer.latestRoundData();
if (block.timestamp - startedAt <= GRACE_PERIOD_TIME || answer == 1) {
return false;
}
return true;
}
}
```

In the getPrice() function, the first step is to call the isSequencerActive() function to check if the sequencer is in a normal operational state. If it is not normal, the function will revert. Only when the sequencer is running normally, the code to retrieve off-chain price data and calculate the GLP price will be executed. This logic aligns with the main functionality of the GLPOracle.sol[1]contract mentioned above.

The isSequencerActive() function retrieves the status of the sequencer by calling sequencer.latestRoundData(). The sequencer is an instance of an oracle contract used to provide external data services, and it primarily uses the sequencer uptime feed proxy address for configuration.latestRoundData() is a public function in that contract, and it returns the status of the sequencer. If it is running normally, the function returns true; otherwise, it returns false.

2 Block Properties

2.1 Issue: Logic errors based on time

Case 1:

The smart contract project for financial derivatives trading relies on block.number to calculate time intervals. Specifically, the _checkDelay()[4]function implements a locking mechanism to check if there is enough time between opening and closing positions. This is done to prevent profiting from opening and closing positions with two different prices in the same transaction within the valid signature pool.

```
function _checkDelay(uint _id, bool _type) internal {
unchecked {
Delay memory _delay = blockDelayPassed[_id];
if (_delay.actionType == _type) {
blockDelayPassed[_id].delay = block.number + blockDelay;
} else {
if (block.number < _delay.delay) revert("0");
blockDelayPassed[_id].delay = block.number + blockDelay;
blockDelayPassed[_id].actionType = _type;
}
}
}
```

Security Risks:

While this structure works fine on Ethereum, it has issues when used on Arbitrum. The sequencer returns the most recently synchronized L1 block number based on block.number every minute. This one-minute time interval can be exploited. Users can open a position before the synchronization occurs (e.g., at 12:00:45 am, L2 obtains L1 block number 1000), and then close it in the next block (e.g., at 12:01 am, L2 obtains number 1004). It may seem like there have been 5 L1 blocks (60/12) since the last transaction, but in reality, there haven't been enough L1 blocks delayed to bypass the locking protection.

Malicious traders can exploit this by continuously updating the block delay in _checkDelay() and increasing the stop-loss threshold, enabling risk-free trading. This is a problem inherent to L1 itself, but if it occurs on Arbitrum, the impact will be amplified as malicious traders can modify the time delay for closing positions without going through the blockDelay.

Risk Avoidance Methods:

Replace 'block.number' with 'block.timestamp'. This way, the code can ensure the correct time delay on both Ethereum and Arbitrum, since 'block.timestamp' always increases according to real time, not based on the block production rate. This can prevent abuse on Arbitrum.

Case 2:

In an Arbitrum smart contract, there are check statements to restrict users from performing multiple operations within a single block. For example, the time check statements in the openPosition() and closePosition() functions[5]are as follows:

```
// Restrict users from performing multiple operations within a single block
traderLatestOperation[trader] != block.number, "ONE_BLOCK_TWICE_OPERATION"
);
```

Security Risks:

Arbitrum updates 'block.number' every minute. However, in reality, several blocks may have passed on L1, but the obtained L1 block number on Arbitrum remains unchanged. This causes a delay of up to one minute in the operations performed in the example code.

Risk Avoidance Methods:

Use ‘block.timestamp‘ in place of ‘block.number‘. The ‘block.timestamp‘ refers to the timestamp of the current block, which is set by the miner producing the block and is generally close to the actual time the block was produced. This method can avoid issues caused by delays in the update of ‘block.number‘.

However, it is important to note that ‘block.timestamp‘ is not entirely reliable, as miners have a degree of freedom in setting this value (usually within certain limits). This could lead to potential security risks. In most cases, these risks are acceptable, but in scenarios requiring very high security, ‘block.timestamp‘ should be used with caution.

3 Contract Address Alias

3.1 Issue: The permission check failed

Case: Uniswap Labs did not consider Address Alias when deploying to Arbitrum.

```
function setOwner(address _owner) external override {
require(msg.sender == owner);
emit OwnerChanged(owner, _owner);
owner = _owner;
}
```

Security Risks:

When deploying the Uniswap v3 Factory[6]to Arbitrum, the owner of the Factory contract is set to the original address of the Timelock contract on Ethereum using the setOwner() function. However, during the L1-to-L2 message call, the msg.sender obtained is the alias address of the Timelock contract on Arbitrum. As a result, the permission check cannot be passed. Additionally, as this alias address is an externally owned account (EOA) and no one possesses the private key for this address, the owner cannot be modified, and functions that require owner permissions cannot be executed on Arbitrum for the Factory contract.

Risk Avoidance Methods:

To address this issue, Arbitrum temporarily disabled address alias for the Timelock contract. In the Inbox contract, there is a specific method created for Uni called uniswapCreateRetryableTicket. In the absence of address alias, the Uniswap Factory contract sends a cross-chain message from the Ethereum Timelock contract to invoke the setOwner() function on Ar bitrum Uniswap Factory, setting the owner to the alias address 0x2BAD8182C09F50c8318d769245beA52C32Be46CD of the Ethereum Timelock contract. This ensures compliance with the permission check in the Factory contract.

4 Gas Fees

4.1 Issue: DOS attack

Case 1:

Each block has an upper limit on the amount of gas that can be consumed. If the gas spent exceeds this limit, the transaction will fail. In a contract, there may be arrays without explicit size restrictions or constraints. An attacker may intentionally add a large amount of data to an array in a function or contract that processes the array, causing the gas to be exhausted during the loop, rendering the function unable to execute successfully.

This situation is prone to occur in crowdfunding projects, auction projects, or other contract projects that may involve batch refunds or other batch operations. For example, an attacker may add a large number of addresses to the contract, each requiring a very small refund. When the project contract attempts to refund by iterating through the array, the loop count becomes

enormous due to the attacker’s addition of a large number of addresses, and the gas cost for this transaction may ultimately exceed the gas limit, resulting in the inability to refund. Since gas prices are lower on Arbitrum, the attacker’s cost is reduced, making such attacks more likely to occur.

In the code snippet below[7], a for loop is used to iterate through each address in the Addresses array.

```
address[] private Addresses;
mapping (address => uint) public fundAmount;

function refundAll() public {
for(uint x; x < Addresses.length; x++) {
require(token.transfer(address(this), Addresses[x], fundAmount[Addresses[x]])); }
}
```

Security Risks:

In each iteration, it uses a require statement to invoke the transfer() function of the Token contract, transferring the funds stored in the contract to the corresponding addresses. If the attacker adds a large number of addresses to the Addresses array, and each address requires a very small refund amount, the loop in the refundAll() function will execute a significant number of iterations. When the loop count exceeds the gas limit of the block, the transaction will fail, resulting in the inability to successfully execute the refunds. In this scenario, the attacker can prevent the refund operation by depleting the gas, thereby affecting the normal operation of the contract.

Risk Avoidance Method 1:

To address this issue, several approaches can be considered:

1.Limiting the array size: Adding a function or modifier to the contract that limits the size of the Addresses array can prevent attackers from adding an excessive number of addresses, thereby reducing the number of iterations in the loop.

2.Batch processing: Dividing the refund operation into multiple loops, with each loop processing a certain number of addresses, can prevent excessive data processing in a single iteration and reduce the number of loops.

3.Using mappings instead of arrays: Consider using mappings instead of arrays to store addresses and refund amounts. Mappings have no size limitation, and refund operations can be handled by iterating through the keys of the mapping without the need for a loop.

4. Using optimized algorithms: Employ more efficient algorithms to handle refund operations, reducing the number of iterations and gas consumption. For example, binary search or other efficient search algorithms can be used to locate specific addresses for refund operations.

Below is an example code demonstrating how to use batch processing to avoid gas limitations. We introduce a new parameter, batchSize, in the refundAll() function. This parameter determines the number of addresses processed in each iteration. The loop pauses after processing the specified number of addresses and then continues with the next batch of addresses. This approach prevents excessive data processing in a single iteration.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Token {
mapping (address => uint) public balances;

function transfer(address from, address to, uint amount) public returns (bool) { require(balances[from] >= amount,
"Insufficient balance");
balances[from] -= amount;
balances[to] += amount;
return true;
}
}

contract RefundContract {
address[] private Addresses;
mapping (address => uint) public fundAmount;
Token private token;

constructor(address _token) {
token = Token(_token);
```



```
}

function addAddress(address _address, uint _amount) public {
Addresses.push(_address);
fundAmount[_address] = _amount;
}

function refundAll(uint batchSize) public {
uint totalAddresses = Addresses.length;
uint processedAddresses;

while (processedAddresses < totalAddresses) {
uint end = processedAddresses + batchSize;
if (end > totalAddresses) {
end = totalAddresses;
}
for (uint i = processedAddresses; i < end; i++) {
require(token.transfer(address(this), Addresses[i], fundAmount[Addresses[i]])); }
processedAddresses = end;
}
```

```
}
}
```

Case 2:

The following example[8]illustrates the security issue caused by frequent small transactions on Arbitrum. In a mortgage lending contract, the calculation statement for updating the collateral ratio is as follows:

```
uint change = timeDelta * _maxCollateralRatioMantissa / _collateralRatioRecoveryDuration;
```

Security Risks:

By rapidly refreshing or setting the _collateralRatioRecoveryDuration greater than _maxCollateralRatioMantissa, it is possible to prevent the update of the collateral ratio. Specifically, let's consider a loan pool where the loan token is WBTC and the collateral token is DAI, with each DAI allowing borrowing of only 1/10000 BTC (with a maximum interest rate of $10,000 per BTC). The _maxCollateralRatioMantissa is set to 1e14, and the _collateralRatioRecoveryDuration is set to 1e15. If an attacker makes frequent deposits of 1 wei WBTC within every 10 seconds, the value of (timeDelta * _maxCollateralRatioMantissa) will always be less than _collateralRatioRecoveryDuration, preventing the update of the collateral ratio. This disrupts the protocol's adaptive pricing mechanism and forces users to borrow at the current interest rate. As the pool's collateral ratio and pool exchange rate will no longer be updated, depositors may experience loss of funds.

Risk Avoidance Methods 2:

To address the vulnerability mentioned above, which involves frequent deposits of 1 wei WBTC preventing the update of the collateral ratio, the following measures can be implemented to mitigate the issue:

1.Adding a minimum deposit amount restriction: Set a minimum deposit amount to prevent frequent small deposits. By establishing a reasonable minimum deposit amount, any deposits below this threshold will be rejected.

2. Adding a deposit time interval restriction: Limit the time interval between deposits to prevent frequent deposits. Set a reasonable time interval, such as allowing only one deposit per hour.

3. Implementing a deposit threshold: Introduce a deposit threshold that requires the deposit amount to reach a certain threshold in order to update the collateral ratio.

4. Introducing a cooldown period: After a deposit is made, introduce a cooldown period. The cooldown period is calculated independently for each user, meaning that each user will have their own cooldown period after making a deposit. Prevent further deposits within the cooldown period, and the length of the cooldown period can be set based on specific circumstances.

Here is an example code that checks for the minimum deposit amount, deposit threshold, and cooldown period when a user makes a deposit. It also updates the collateral ratio and the depositor's last deposit time if all conditions are met.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract CollateralRatioProtection {
```



```
uint256 private constant MIN_DEPOSIT_AMOUNT = 1e8; // 1 BTC
uint256 private constant DEPOSIT_THRESHOLD = 1e6; // 0.01 BTC
uint256 private constant DEPOSIT_COOLDOWN = 1 hours;

mapping(address => uint256) private lastDepositTime;

function deposit(uint256 amount) external {
require(amount >= MIN_DEPOSIT_AMOUNT, "Deposit amount too small");
require(amount >= DEPOSIT_THRESHOLD, "Deposit amount below threshold"); require(block.timestamp >= lastDepositTime[msg.sender] +
DEPOSIT_COOLDOWN, "Deposit cooldown period not elapsed");

// Update collateral ratio and other necessary actions
// ...

lastDepositTime[msg.sender] = block.timestamp;
}
}
```

References

1. Evert0x. Glporacle (2023). https://github.com/sherlock-audit/2023-01-sentiment/blob/main/oracle/src/gmx/GLPOracle.sol.

2. r0ohafza. Glporacle.t.sol (2023). https://github.com/sentimentxyz/oracle/blob/815233add2d23a7e2a2c5136504537b234a65c47/src/tests/GLPOracle.t.sol.

3. r0ohafza. Glporacleupdate.sol (2023). https://github.com/sentimentxyz/oracle/blob/main/src/gmx/GLPOracle.sol.

4. TriHaz. Trading.sol (2022). https://github.com/code-423n4/2022-12-tigris/blob/588c84b7bb354d20cbca6034544c4faa46e 6a80e/contracts/Trading.sol#L857-L868.

5. tintinweb. Marginfactory.sol (2022). https://github.com/tintinweb/smart-contract-sanctuary-arbitrum/blob/662d22a0f98c6a 0c8ef23e43ac6d6a3eac5968da/contracts.

6. Uniswap. Uniswapv3factory (2021). https://arbiscan.io/address/0x1f98431c8ad98523631ae4a59f267346ea31f984#code.

7. DavidBDiligence. Dos (2022). https://github.com/Consensys/smart-contract-best-practices/blob/master/docs/attacks/denial -of-service.md.

8. sherlock admin. Pool (2023). https://github.com/sherlock-audit/2023-02-surge/blob/main/surge-protocol-v1/src/Pool.sol#L 216-L263.