SHERLOCK

Security Review For Velar

Public Audit Contest Prepared For:	**Velar**
Lead Security Expert:	**bughuntoor**
Date Audited:	**August 30 - September 9, 2024**
Final Commit:	**3338c2c**

Introduction

Audit The World's First perpetual DEX on Bitcoin

Scope

Repository: Velar-co/gl-sherlock

Audited Commit: 0e291ffee5e1239efbd01a080f393af59772ae47

Final Commit: 3338c2c286e58f4816cd97c472d39c8423f2b425

Files:

- contracts/RedstoneExtractor.sol
- contracts/api.vy
- contracts/core.vy
- contracts/fees.vy
- contracts/math.vy
- contracts/oracle.vy
- contracts/params.vy
- contracts/pools.vy
- contracts/positions.vy
- contracts/tokens/ERC20Plus.vy
- contracts/tokens/IERC20Plus.vy
- contracts/types.vy

Final Commit Hash

3338c2c286e58f4816cd97c472d39c8423f2b425

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues Found

High	Medium
2	10

Issues Not Fixed and Not Acknowledged

High	Medium
0	0

Security experts who found valid issues

0x37
0xbranded
4gontuk
Bauer
Greed
Japy69
KupiaSec
PASCAL
Waydou
aslanbek
bughuntoor
pashap9990
y4y

Issue H-1: Fee Precision Loss Disrupts Liquidations and Causes Loss of Funds

Source: https://github.com/sherlock-audit/2024-08-velar-artha-judging/issues/66

Found by

0xbranded

Summary

Borrowing and funding fees of both longs/shorts suffer from two distinct sources of precision loss. The level of precision loss is large enough to consistently occur at a significant level, and can even result in total omission of fee payment for periods of time. This error is especially disruptive given the sensitive nature of funding fee calculations both in determining liquidations (a core functionality), as well as payments received by LPs and funding recipients (representing a significant loss).

Vulnerability Detail

The first of the aforementioned sources of precision loss is relating to the `DENOM` parameter defined and used in `apply` of math.vy:

This function is in turn referenced (to extract the `fee` parameter in particular) in several locations throughout fees.vy, namely in determining the funding and borrowing payments made by positions open for a duration of time:

The comments for `DENOM` specify that it's a "magic number which depends on the smallest fee one wants to support and the blocktime." In fact, given its current value of 10^{-9}, the smallest representable fee per block is 10^{-7}%. Given the average blocktimes of 2.0 sec on the BOB chain, there are 15_778_800 blocks in a standard calendar year. Combined with the fee per block, this translates to an annual fee of 1.578%.

However, this is also the interval size for annualized fees under the current system. As a result, any fee falling below the next interval will be rounded down. For example, given an annualized funding rate in the neighborhood of 15%, there is potential for a nearly 10% error in the interest rate if rounding occurs just before the next interval. This error is magnified the smaller the funding rates become. An annual fee of 3.1% would round down to 1.578%, representing an error of nearly 50%. And any annualized fees below 1.578% will not be recorded, representing a 100% error.

The second source of precision loss combines with the aforementioned error, to both increase the severity and frequency of error. It's related to how percentages are handled in `params.vy`, particularly when the long/short utilization is calculated to determine funding & borrow rates. The utilization is shown below:

This function is in turn used to calculate borrowing (and funding rates, following a slightly different approach that similarly combines the use of `utilization` and `scale`), in `[dynamic_fees](https://github.com/sherlock-audit/2024-08-velar-artha/blob/main/gl-sherlock/contracts/params.vy#L33-L55)` of `params.vy`:

Note that `interest` and `reserves` maintain the same precision. Therefore, the output of `utilization` will have just 2 digits of precision, resulting from the division of `reserves` by `100`. However, this approach can similarly lead to fee rates losing a full percentage point in their absolute value. Since the utilization is used by `dynamic_fees` to calculate the funding / borrow rates, when combined with the formerly described source of precision loss the error is greatly amplified.

Consider a scenario when the long open interest is 199_999 * 10^{18} and the reserves are 10_000_000 * 10^{18}. Under the current `utilization` functionality, the result would be a 1.9999% utilization rounded down to 1%. Further assuming the value of `max_fee = 65` (this represents a 100% annual rate and 0.19% 8-hour rate), the long borrow rate would round down to 0%. Had the 1.9999% utilization rate not been rounded down 1%, the result would have been `r = 1.3`. In turn, the precision loss in `DENOM` would have effectively rounded down to `r = 1`, resulting in a 2.051% borrow rate rounded down to 1.578%.

In other words, the precision loss in `DENOM` alone would have resulted in a 23% error in this case. But when combined with the precision loss in percentage points represented in `utilization`, a 100% error resulted. While the utilization and resulting interest rates will typically not be low enough to produce such drastic errors, this hopefully illustrates the synergistic combined impact of both sources of precision loss. Even at higher, more representative values for these rates (such as `r = 10`), errors in fee calculations exceeding 10% will consistently occur.

Impact

All fees in the system will consistently be underpaid by a significant margin, across all pools and positions. Additionally trust/confidence in the system will be eroded as fee application will be unpredictable, with sharp discontinuities in rates even given moderate changes in pool utilization. Finally, positions will be subsidized at the expense of LPs, since the underpayment of fees will make liquidations less likely and take longer to occur. As a result, LPs and funding recipients will have lesser incentive to provide liquidity, as they are consistently being underpaid while taking on a greater counterparty risk.

As an example, consider the scenario where the long open interest is 1_099_999 * 10^{18} and the reserves are 10_000_000 * 10^{18}. Under the current `utilization` functionality, the result would be a 10.9999% utilization rounded down to 10%. Assuming `max_fee = 65` (100% annually, 0.19% 8-hour), the long borrow rate would be `r = 6.5` rounded down to `r = 6`. A 9.5% annual rate results, whereas the accurate result if neither precision loss occurred is `r = 7.15` or 11.3% annually. The resulting error in the borrow rate is 16%.

Assuming a long collateral of 100_000 * 10^{18}, LPs would earn 9_500 * 10^{18}, when they should earn 11_300 * 10^{18}, a shortfall of 1_800 * 10^{18} from longs alone. Additional borrow fee shortfalls would occur for shorts, in addition to shortfalls in funding payments received.

Liquidation from borrow rates along should have taken 106 months based on the expected result of 11.3% per annum. However, under the 9.5% annual rate it would take 127 months to liquidate a position. This represents a 20% delay in liquidation time from borrow rates alone, not including the further delay caused by potential underpaid funding rates.

When PnL is further taken into account, these delays could mark the difference between a period of volatility wiping out a position. As a result, these losing positions could run for far longer than should otherwise occur and could even turn into winners. Not only are LP funds locked for longer as a result, they are at a greater risk of losing capital to their counterparty. On top of this, they are also not paid their rightful share of profits, losing out on funds to take on an unfavorably elevated risk.

Thus, not only do consistent, material losses (significant fee underpayment) occur but a critical, core functionality malfunctions (liquidations are delayed).

Code Snippet

In the included PoC, three distinct tests demonstrate the individual sources of precision loss, as well as their combined effect. Similar scenarios were demonstrated as discussed above, for example interest = 199_999 * $10^{18} with reserves = 10_000_000 * $10^{18} with a max fee of 65.

The smart contracts were stripped to isolate the relevant logic, and foundry was used for testing. To run the test, clone the repo and place `Denom.vy` in vyper_contracts, and place `Denom.t.sol`, `Cheat.sol`, and `IDenom.sol` under src/test.

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.8.13;

import {CheatCodes} from "./Cheat.sol";

import "../../lib/ds-test/test.sol";
import "../../lib/utils/Console.sol";
import "../../lib/utils/VyperDeployer.sol";

import "../IDenom.sol";

contract DenomTest is DSTest {
    ///@notice create a new instance of VyperDeployer
    VyperDeployer vyperDeployer = new VyperDeployer();
    CheatCodes vm = CheatCodes(0x7109709ECfa91a80626fF3989D68f67F5b1DD12D);
    IDenom denom;
    uint256 constant YEAR = (60*60*24*(365) + 60*60*24 / 4); //includes leap year

    function setUp() public {
        vm.roll(1);
        ///@notice deploy a new instance of ISimplestore by passing in the address
↪   of the deployed Vyper contract
```

```
        denom = IDenom(vyperDeployer.deployContract("Denom"));
    }

    function testDenom() public {
        uint256 blocksInYr = (YEAR) / 2;

        vm.roll(blocksInYr);

        denom.update();

        // pools were initialized at block 0
        denom.calc(true, 1e25, 0);
        denom.calc2(true, 1e25, 0);
    }

        function testRounding() public {
        uint max_fee = 100; // set max 8-hour rate to 0.288% (157.8% annually)

        uint256 reserves = 10_000_000 ether;
        uint256 interest1 = 1_000_000 ether;
        uint256 interest2 = 1_099_999 ether;

        uint256 util1 = denom.utilization(reserves, interest1);
        uint256 util2 = denom.utilization(reserves, interest2);

        assertEq(util1, util2); // current calculation yields same utilization for
↪  both interests

        // borrow rate
        uint fee1 = denom.scale(max_fee, util1);
        uint fee2 = denom.scale(max_fee, util2);

        assertEq(fee1, fee2); // results in the same fee also. fee2 should be ~1%
↪  higher
    }

    function testCombined() public {
        // now let's see what would happen if we raised the precision of both fees
↪  and percents
        uint max_fee = 65;
        uint max_fee2 = 65_000; // 3 extra digits of precision lowers error by 3
↪  orders of magnitude

        uint256 reserves = 10_000_000 ether;
        uint256 interest = 199_999 ether; // interest & reserves same in both, only
↪  differ in precision.

        uint256 util1 = denom.utilization(reserves, interest);
        uint256 util2 = denom.utilization2(reserves, interest); // 3 extra digits
↪  of precision here also
```

```
        // borrow rate
        uint fee1 = denom.scale(max_fee, util1);
        uint fee2 = denom.scale2(max_fee2, util2);

        assertEq(fee1 * 1_000, fee2 - 999); // fee 1 is 1.000, fee 2 is 1.999 (~50%
↪  error)
    }
}
```

```
// Possible caller modes for readCallers()
enum CallerMode {
    None,
    Broadcast,
    RecurrentBroadcast,
    Prank,
    RecurrentPrank
}

enum AccountAccessKind {
    Call,
    DelegateCall,
    CallCode,
    StaticCall,
    Create,
    SelfDestruct,
    Resume
}

struct Wallet {
    address addr;
    uint256 publicKeyX;
    uint256 publicKeyY;
    uint256 privateKey;
}

struct ChainInfo {
    uint256 forkId;
    uint256 chainId;
}

struct AccountAccess {
    ChainInfo chainInfo;
    AccountAccessKind kind;
    address account;
    address accessor;
    bool initialized;
    uint256 oldBalance;
```

```
    uint256 newBalance;
    bytes deployedCode;
    uint256 value;
    bytes data;
    bool reverted;
    StorageAccess[] storageAccesses;
}

struct StorageAccess {
    address account;
    bytes32 slot;
    bool isWrite;
    bytes32 previousValue;
    bytes32 newValue;
    bool reverted;
}

// Derives a private key from the name, labels the account with that name, and
↪  returns the wallet
function createWallet(string calldata) external returns (Wallet memory);

// Generates a wallet from the private key and returns the wallet
function createWallet(uint256) external returns (Wallet memory);

// Generates a wallet from the private key, labels the account with that name, and
↪  returns the wallet
function createWallet(uint256, string calldata) external returns (Wallet memory);

// Signs data, (Wallet, digest) => (v, r, s)
function sign(Wallet calldata, bytes32) external returns (uint8, bytes32, bytes32);

// Get nonce for a Wallet
function getNonce(Wallet calldata) external returns (uint64);

// Set block.timestamp
function warp(uint256) external;

// Set block.number
function roll(uint256) external;

// Set block.basefee
function fee(uint256) external;

// Set block.difficulty
// Does not work from the Paris hard fork and onwards, and will revert instead.
function difficulty(uint256) external;

// Set block.prevrandao
// Does not work before the Paris hard fork, and will revert instead.
function prevrandao(bytes32) external;
```

```
// Set block.chainid
function chainId(uint256) external;

// Loads a storage slot from an address
function load(address account, bytes32 slot) external returns (bytes32);

// Stores a value to an address' storage slot
function store(address account, bytes32 slot, bytes32 value) external;

// Signs data
function sign(uint256 privateKey, bytes32 digest)
    external
    returns (uint8 v, bytes32 r, bytes32 s);

// Computes address for a given private key
function addr(uint256 privateKey) external returns (address);

// Derive a private key from a provided mnemonic string,
// or mnemonic file path, at the derivation path m/44'/60'/0'/0/{index}.
function deriveKey(string calldata, uint32) external returns (uint256);
// Derive a private key from a provided mnemonic string, or mnemonic file path,
// at the derivation path {path}{index}
function deriveKey(string calldata, string calldata, uint32) external returns
↪  (uint256);

// Gets the nonce of an account
function getNonce(address account) external returns (uint64);

// Sets the nonce of an account
// The new nonce must be higher than the current nonce of the account
function setNonce(address account, uint64 nonce) external;

// Performs a foreign function call via terminal
function ffi(string[] calldata) external returns (bytes memory);

// Set environment variables, (name, value)
function setEnv(string calldata, string calldata) external;

// Read environment variables, (name) => (value)
function envBool(string calldata) external returns (bool);
function envUint(string calldata) external returns (uint256);
function envInt(string calldata) external returns (int256);
function envAddress(string calldata) external returns (address);
function envBytes32(string calldata) external returns (bytes32);
function envString(string calldata) external returns (string memory);
function envBytes(string calldata) external returns (bytes memory);

// Read environment variables as arrays, (name, delim) => (value[])
function envBool(string calldata, string calldata)
```

```
    external
    returns (bool[] memory);
function envUint(string calldata, string calldata)
    external
    returns (uint256[] memory);
function envInt(string calldata, string calldata)
    external
    returns (int256[] memory);
function envAddress(string calldata, string calldata)
    external
    returns (address[] memory);
function envBytes32(string calldata, string calldata)
    external
    returns (bytes32[] memory);
function envString(string calldata, string calldata)
    external
    returns (string[] memory);
function envBytes(string calldata, string calldata)
    external
    returns (bytes[] memory);

// Read environment variables with default value, (name, value) => (value)
function envOr(string calldata, bool) external returns (bool);
function envOr(string calldata, uint256) external returns (uint256);
function envOr(string calldata, int256) external returns (int256);
function envOr(string calldata, address) external returns (address);
function envOr(string calldata, bytes32) external returns (bytes32);
function envOr(string calldata, string calldata) external returns (string memory);
function envOr(string calldata, bytes calldata) external returns (bytes memory);

// Read environment variables as arrays with default value, (name, value[]) =>
↪  (value[])
function envOr(string calldata, string calldata, bool[] calldata) external returns
↪  (bool[] memory);
function envOr(string calldata, string calldata, uint256[] calldata) external
↪  returns (uint256[] memory);
function envOr(string calldata, string calldata, int256[] calldata) external
↪  returns (int256[] memory);
function envOr(string calldata, string calldata, address[] calldata) external
↪  returns (address[] memory);
function envOr(string calldata, string calldata, bytes32[] calldata) external
↪  returns (bytes32[] memory);
function envOr(string calldata, string calldata, string[] calldata) external
↪  returns (string[] memory);
function envOr(string calldata, string calldata, bytes[] calldata) external returns
↪  (bytes[] memory);

// Convert Solidity types to strings
function toString(address) external returns(string memory);
function toString(bytes calldata) external returns(string memory);
```

```
function toString(bytes32) external returns(string memory);
function toString(bool) external returns(string memory);
function toString(uint256) external returns(string memory);
function toString(int256) external returns(string memory);

// Sets the *next* call's msg.sender to be the input address
function prank(address) external;

// Sets all subsequent calls' msg.sender to be the input address
// until `stopPrank` is called
function startPrank(address) external;

// Sets the *next* call's msg.sender to be the input address,
// and the tx.origin to be the second input
function prank(address, address) external;

// Sets all subsequent calls' msg.sender to be the input address until
// `stopPrank` is called, and the tx.origin to be the second input
function startPrank(address, address) external;

// Resets subsequent calls' msg.sender to be `address(this)`
function stopPrank() external;

// Reads the current `msg.sender` and `tx.origin` from state and reports if there
↪  is any active caller modification
function readCallers() external returns (CallerMode callerMode, address msgSender,
↪  address txOrigin);

// Sets an address' balance
function deal(address who, uint256 newBalance) external;

// Sets an address' code
function etch(address who, bytes calldata code) external;

// Marks a test as skipped. Must be called at the top of the test.
function skip(bool skip) external;

// Expects an error on next call
function expectRevert() external;
function expectRevert(bytes calldata) external;
function expectRevert(bytes4) external;

// Record all storage reads and writes
function record() external;

// Gets all accessed reads and write slot from a recording session,
// for a given address
function accesses(address)
    external
    returns (bytes32[] memory reads, bytes32[] memory writes);
```

```
// Record all account accesses as part of CREATE, CALL or SELFDESTRUCT opcodes in
↪  order,
// along with the context of the calls.
function startStateDiffRecording() external;

// Returns an ordered array of all account accesses from a
↪  `startStateDiffRecording` session.
function stopAndReturnStateDiff() external returns (AccountAccess[] memory
↪  accesses);

// Record all the transaction logs
function recordLogs() external;

// Gets all the recorded logs
function getRecordedLogs() external returns (Log[] memory);

// Prepare an expected log with the signature:
//   (bool checkTopic1, bool checkTopic2, bool checkTopic3, bool checkData).
//
// Call this function, then emit an event, then call a function.
// Internally after the call, we check if logs were emitted in the expected order
// with the expected topics and data (as specified by the booleans)
//
// The second form also checks supplied address against emitting contract.
function expectEmit(bool, bool, bool, bool) external;
function expectEmit(bool, bool, bool, bool, address) external;

// Mocks a call to an address, returning specified data.
//
// Calldata can either be strict or a partial match, e.g. if you only
// pass a Solidity selector to the expected calldata, then the entire Solidity
// function will be mocked.
function mockCall(address, bytes calldata, bytes calldata) external;

// Reverts a call to an address, returning the specified error
//
// Calldata can either be strict or a partial match, e.g. if you only
// pass a Solidity selector to the expected calldata, then the entire Solidity
// function will be mocked.
function mockCallRevert(address where, bytes calldata data, bytes calldata retdata)
↪  external;

// Clears all mocked and reverted mocked calls
function clearMockedCalls() external;

// Expect a call to an address with the specified calldata.
// Calldata can either be strict or a partial match
function expectCall(address callee, bytes calldata data) external;
// Expect a call to an address with the specified
```

```
// calldata and message value.
// Calldata can either be strict or a partial match
function expectCall(address callee, uint256, bytes calldata data) external;

// Gets the _creation_ bytecode from an artifact file. Takes in the relative path
↪  to the json file
function getCode(string calldata) external returns (bytes memory);
// Gets the _deployed_ bytecode from an artifact file. Takes in the relative path
↪  to the json file
function getDeployedCode(string calldata) external returns (bytes memory);

// Label an address in test traces
function label(address addr, string calldata label) external;

// Retrieve the label of an address
function getLabel(address addr) external returns (string memory);

// When fuzzing, generate new inputs if conditional not met
function assume(bool) external;

// Set block.coinbase (who)
function coinbase(address) external;

// Using the address that calls the test contract or the address provided
// as the sender, has the next call (at this call depth only) create a
// transaction that can later be signed and sent onchain
function broadcast() external;
function broadcast(address) external;

// Using the address that calls the test contract or the address provided
// as the sender, has all subsequent calls (at this call depth only) create
// transactions that can later be signed and sent onchain
function startBroadcast() external;
function startBroadcast(address) external;
function startBroadcast(uint256 privateKey) external;

// Stops collecting onchain transactions
function stopBroadcast() external;

// Reads the entire content of file to string, (path) => (data)
function readFile(string calldata) external returns (string memory);
// Get the path of the current project root
function projectRoot() external returns (string memory);
// Reads next line of file to string, (path) => (line)
function readLine(string calldata) external returns (string memory);
// Writes data to file, creating a file if it does not exist, and entirely
↪  replacing its contents if it does.
// (path, data) => ()
function writeFile(string calldata, string calldata) external;
// Writes line to file, creating a file if it does not exist.
```

```
// (path, data) => ()
function writeLine(string calldata, string calldata) external;
// Closes file for reading, resetting the offset and allowing to read it from
↪  beginning with readLine.
// (path) => ()
function closeFile(string calldata) external;
// Removes file. This cheatcode will revert in the following situations, but is not
↪  limited to just these cases:
// - Path points to a directory.
// - The file doesn't exist.
// - The user lacks permissions to remove the file.
// (path) => ()
function removeFile(string calldata) external;
// Returns true if the given path points to an existing entity, else returns false
// (path) => (bool)
function exists(string calldata) external returns (bool);
// Returns true if the path exists on disk and is pointing at a regular file, else
↪  returns false
// (path) => (bool)
function isFile(string calldata) external returns (bool);
// Returns true if the path exists on disk and is pointing at a directory, else
↪  returns false
// (path) => (bool)
function isDir(string calldata) external returns (bool);

// Return the value(s) that correspond to 'key'
function parseJson(string memory json, string memory key) external returns (bytes
↪  memory);
// Return the entire json file
function parseJson(string memory json) external returns (bytes memory);
// Check if a key exists in a json string
function keyExists(string memory json, string memory key) external returns (bytes
↪  memory);
// Get list of keys in a json string
function parseJsonKeys(string memory json, string memory key) external returns
↪  (string[] memory);

// Snapshot the current state of the evm.
// Returns the id of the snapshot that was created.
// To revert a snapshot use `revertTo`
function snapshot() external returns (uint256);
// Revert the state of the evm to a previous snapshot
// Takes the snapshot id to revert to.
// This deletes the snapshot and all snapshots taken after the given snapshot id.
function revertTo(uint256) external returns (bool);

// Creates a new fork with the given endpoint and block,
// and returns the identifier of the fork
function createFork(string calldata, uint256) external returns (uint256);
// Creates a new fork with the given endpoint and the _latest_ block,
```

```
// and returns the identifier of the fork
function createFork(string calldata) external returns (uint256);

// Creates _and_ also selects a new fork with the given endpoint and block,
// and returns the identifier of the fork
function createSelectFork(string calldata, uint256)
    external
    returns (uint256);
// Creates _and_ also selects a new fork with the given endpoint and the
// latest block and returns the identifier of the fork
function createSelectFork(string calldata) external returns (uint256);

// Takes a fork identifier created by `createFork` and
// sets the corresponding forked state as active.
function selectFork(uint256) external;

// Returns the currently active fork
// Reverts if no fork is currently active
function activeFork() external returns (uint256);

// Updates the currently active fork to given block number
// This is similar to `roll` but for the currently active fork
function rollFork(uint256) external;
// Updates the given fork to given block number
function rollFork(uint256 forkId, uint256 blockNumber) external;

// Fetches the given transaction from the active fork and executes it on the
↪  current state
function transact(bytes32) external;
// Fetches the given transaction from the given fork and executes it on the current
↪  state
function transact(uint256, bytes32) external;

// Marks that the account(s) should use persistent storage across
// fork swaps in a multifork setup, meaning, changes made to the state
// of this account will be kept when switching forks
function makePersistent(address) external;
function makePersistent(address, address) external;
function makePersistent(address, address, address) external;
function makePersistent(address[] calldata) external;
// Revokes persistent status from the address, previously added via `makePersistent`
function revokePersistent(address) external;
function revokePersistent(address[] calldata) external;
// Returns true if the account is marked as persistent
function isPersistent(address) external returns (bool);

/// Returns the RPC url for the given alias
function rpcUrl(string calldata) external returns (string memory);
/// Returns all rpc urls and their aliases `[alias, url][]`
```

```
function rpcUrls() external returns (string[2][] memory);
```

}

```


</details>

<details>
<summary>IDenom.sol</summary>

```solidity
// SPDX-License-Identifier: MIT
pragma solidity >=0.8.13;

interface IDenom {
    function update() external;
    function calc(bool, uint256, uint256) external returns(SumFees memory);
    function calc2(bool, uint256, uint256) external returns(SumFees memory);

    function utilization(uint256, uint256) external returns(uint256);
    function utilization2(uint256, uint256) external returns(uint256);
    function scale(uint256, uint256) external returns(uint256);
    function scale2(uint256, uint256) external returns(uint256);

    struct SumFees{
        uint256 funding_paid;
        uint256 funding_received;
    }
}
```

Tool used

Manual Review

Recommendation

Consider increasing the precision of `DENOM` by atleast 3 digits, i.e. `DENOM: constant(uint256) = 1_000_000_000_000` instead of `1_000_000_000`. Consider increasing the precision of percentages by 3 digits, i.e. divide / multiply by 100_000 instead of 100.

Each added digit of precision decreases the precision loss by an order of magnitude. In other words the 1% and 1.5% absolute errors in precision would shrink to 0.01% and 0.015% when using three extra digits of precision.

Consult `Denom.vy` for further guidance on necessary adjustments to make to the various functions to account for these updated values.
```

Discussion

KupiaSecAdmin

Escalate

This is a system design choice. It just charges less fees, there is no loss of funds happening to the protocol. Also this can be modified by updating parameters.

sherlock-admin3

> Escalate
> This is a system design choice. It just charges less fees, there is no loss of funds happening to the protocol. Also this can be modified by updating parameters.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

msheikhattari

It's not a design choice, this is clearly a precision loss issue. There are real loss of funds and delayed liquidation that consistently occur given the real world parameters such as blocktime. these are clearly documented in the PoC with specific errors exceeding 10% consistently, and sometimes even total loss of fee payment.

DENOM is a constant that cannot be updated. It must be corrected by increasing the decimals of precision.

Given the high likelihood and high impact of loss of funds not only from fee miscalculation but delayed/avoided liquidations, this is a high severity issue.

KupiaSecAdmin

@msheikhattari - The logic only modifies the fee ratio by rounding it down, if the ratio doesn't meet what the protocol team is looking for, they can simply increase the numerator by updating parameters. Also even though with lower fee ratio, why does it cause loss? If borrow fee becomes less, there will be more positions openers who will also increase utilization ratio which also will increase the borrowing fee. Still, this is low/info at most.

msheikhattari

No parameters can be updated to fix this issue - DENOM must directly be changed (which is a constant). It currently supports annual fees of about 1.5% increments which is way too much precision loss with errors consistently exceeding 10% in all fee calculations.

The numerator is blocks * rate, the point is that the rate component cannot support fees with a precision below 1.5% because of the DENOM parameter that is too small.

I included a PoC which clearly demonstrates this currently unavoidable precision loss.

rickkk137

> No parameters can be updated to fix this issue - DENOM must directly be changed (which is a constant). It currently supports annual fees of about 1.5% increments which is way too much precision loss with errors consistently exceeding 10% in all fee calculations.
> The numerator is blocks * rate, the point is that the rate component cannot support fees with a precision below 1.5% because of the DENOM parameter that is too small.
> I included a PoC which clearly demonstrates this currently unavoidable precision loss.

I read your PoC,root cause in this report is here which mentioned in this report and borrowing_paid calculation doesn't have effect

WangSecurity

Though, this may be considered a design decision, the calculation of fees still has precision loss which would pay less fees due to rounding down. Hence, this should've been included in the known issues question in the protocol's README, but it wasn't. Also, `DENOM` couldn't be changed after the contracts were deployed, but it wasn't flagged as a hardcoded variable that could be changed.

Hence, this should remain a valid issue. Planning to reject the escalation.

msheikhattari

Given that significant disruptions to liquidations and loss of funds result, this issue should in consideration for high severity.

WangSecurity

Agree with the above, as I understand, there are no extensive limitations, and the loss can be significant. Planning to reject the escalation but make the issue high severity.

rickkk137

root cause in this issue and #72 is same

KupiaSecAdmin

@WangSecurity - Even though `DENOM` can't be modified, `denominator` and `numerator` are updatable which determines the fee ratio. So making this valid doesn't seem appropriate and it can never be high severity.

rickkk137

@KupiaSecAdmin I agree with u ,this issue talks about two part first one:

```
def utilization(reserves: uint256, interest: uint256) -> uint256:
    return 0 if (reserves == 0 or interest == 0) else (interest / (reserves / 100))
```

second one:

```
def apply(x: uint256, numerator: uint256) -> Fee:
  fee      : uint256 = (x * numerator) / DENOM
```

first one has precision loss but second one doesn't have

KupiaSecAdmin

@rickkk137 - Regarding first one, isn't this issue same?
https://github.com/sherlock-audit/2024-08-velar-artha-judging/issues/87

rickkk137

@KupiaSecAdmin no,I don't think so, imo #87 is not possible i wrote my comment there

msheikhattari

Please refer to the PoC, the issue is currently unmitigatable due to the precision of all fees that result from the `DENOM` parameter. It doesn't result directly from any of the calculations but the lowest representable precision of fee parameters. Let me know if I can clarify anything further.

WangSecurity

I believe @rickkk137 is correct that this and #72 have the same root cause and explain the same problem. The escalation will still be rejected, and #72 + #60 (duplicate) will be duplicated with this report because it goes more in-depth in explaining the issue and shows a higher severity.

msheikhattari

#72 is describing a different issue. It doesn't mention the DENOM parameter at all and should not be duped with this one.

That issue is talking about setting min values for `long\_utilization` and `short\_utilization`, the only similarity is that they are both sources of precision loss. Otherwise the underlying issue is different.

WangSecurity

Yes, excuse me, focused too much on the utilisation part. The precision loss from Denom is relatively low based on the discussion under #126. But here the report combines 2 precision loss factors resulting in a major precision loss. Hence, I'm returning to my previous decision to reject the escalation, increase severity to high and leave it solo, because it combines 2 precision loss factors resulting in a more major issue than #72 which talks only about utilisation. Planning to apply this decision in a couple of hours.

aslanbekaibimov

@WangSecurity

> The duplication rules assume we have a "target issue", and the "potential duplicate" of that issue needs to meet the following requirements to be considered a duplicate.

> Identify the root cause Identify at least a Medium impact Identify a valid attack path or vulnerability path Fulfills other submission quality requirements (e.g. provides a PoC for categories that require one)

Don't #72 and #60 satisfy all 4 requirements?

WangSecurity

Good question, but #72 and #60 identified only one source of precision loss, so the following should apply:

> The exception to this would be if underlying code implementations OR impact OR the fixes are different, then they may be treated separately.

That's the reason I think they should be treated separately. The decision remains, reject the escalation, increase severity to high.

rickkk137

> But here the report combines 2 precision loss factors resulting in a major precision loss

```
def utilization(reserves: uint256, interest: uint256) -> uint256:
    """
    Reserve utilization in percent (rounded down). @audit this is actually rounded
     ↪  up...
    """
    @>>> return 0 if (reserves == 0 or interest == 0) else (interest / (reserves /
     ↪  100))

@external
@pure
def utilization2(reserves: uint256, interest: uint256) -> uint256:
    """
    Reserve utilization in percent (rounded down). @audit this is actually rounded
     ↪  up...
    """
   @>>> return 0 if (reserves == 0 or interest == 0) else (interest / (reserves /
    ↪  100\_000))



function testCombined() public {
        // now let's see what would happen if we raised the precision of both fees
         ↪  and percents
        uint max\_fee = 65;
        uint max\_fee2 = 65\_000; // 3 extra digits of precision lowers error by 3
         ↪  orders of magnitude

        uint256 reserves = 10\_000\_000 ether;
```

```
        uint256 interest = 199\_999 ether; // interest & reserves same in both,
        ↪  only differ in precision.

        uint256 util1 = denom.utilization(reserves, interest);
    @>>>    uint256 util2 = denom.utilization2(reserves, interest); // 3 extra
    ↪  digits of precision here also

        // borrow rate
        uint fee1 = denom.scale(max\_fee, util1);
     @>>>  uint fee2 = denom.scale2(max\_fee2, util2);

        assertEq(fee1 * 1\_000, fee2 - 999); // fee 1 is 1.000, fee 2 is 1.999
        ↪  (~50\% error)
    }
```

the watson passed util2 to denom.scale2 function in his PoC and that made huge difference and utilization2 function is custom function has written by watson

msheikhattari

Yes, these combined sources of precision loss were demonstrated to have a far more problematic impact as shown in the PoC.

WangSecurity

But, as I understand from @rickkk137, his main point is without the precision inside the Utilisation, the loss would be small and not sufficient for medium severity. Is it correct @rickkk137 ?

rickkk137

Yes,correct

msheikhattari

The precision loss resulting from DENOM is more significant than that from the utilization. The PoC and described scenario in the issue provides exact details on the loss of each.

When combined, not only is the precision loss more severe but also more likely to occur.

WangSecurity

But as I see in #126, the precision loss from Denom is not that severe, is it wrong?

msheikhattari

That issue is slightly different, but what was pointed out here is that the most granular annual fee representable is about 1.6% - these are the intervals for fee rates as well (ex. 2x 1.6, 3x 1.6...)

Utilization on the other hand experiences precision loss of 1% in the extreme case (ex 14.9% -> 14%)

So in absolute terms the issue arising from DENOM is more significant, when combined these issues become far more significant than implied by their nominal values, not only

due to multiplied loss of precision but increased likelihood of loss (if precision loss from one source bumps it just over the boundary of another, as outlined in the PoC)

WangSecurity

Yeah, I see. #126 tries to show a scenario where DENOM precision loss would round down the fees to 0, and for that to happen, the fees or collateral have to be very small, which results in a very small loss. But this issue just shows the precision loss from DENOM and doesn't try to show rounding down to 0. That's the key difference between the two reports.

Hence, my decision remains that this will remain solo with high severity as expressed above. Planning to reject the escalation. The decision will be applied tomorrow at 10 am UTC:

> *Note: #126 won't be duplicated with this report as it doesn't show Medium impact*

WangSecurity

Result: High Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- KupiaSecAdmin: rejected

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Velar-co/gl-sherlock/pull/3

Issue H-2: User can sandwich their own position close to get back all of their position fees

Source: https://github.com/sherlock-audit/2024-08-velar-artha-judging/issues/94

Found by

0x37, KupiaSec, bughuntoor

Summary

User can sandwich their own position close to get back all of their position fees

Vulnerability Detail

Within the protocol, borrowing fees are only distributed to LP providers when the position is closed. Up until then, they remain within the position. The problem is that in this way, fees are distributed evenly to LP providers, without taking into account the longevity of their LP provision.

This allows a user to avoid paying fees in the following way:

1. Flashloan a large sum and add it as liquidity
2. Close their position and let the fees be distributed (with most going back to them as they've got majority in the pool)
3. WIthdraw their LP tokens
4. Pay back the flashloan

Impact

Users can avoid paying borrowing fees.

Code Snippet

https://github.com/sherlock-audit/2024-08-velar-artha/blob/main/gl-sherlock/contracts/positions.vy#L156

Tool used

Manual Review

Recommendation

Implement a system where fees are gradually distributed to LP providers.

Discussion

KupiaSecAdmin

Escalate

This is invalid, `FEES.update` function is called after every action, so when the liquidity of flashloan is added, the accrued fees are distributed to prev LP providers.

sherlock-admin3

> Escalate
> This is invalid, `FEES.update` function is called after every action, so when the liquidity of flashloan is added, the accrued fees are distributed to prev LP providers.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

spacegliderrrr

Comment above is simply incorrect. `Fees.update` doesn't actually make a change on the LP token value - it simply calculates the new fee terms and makes a snapshot at current block.

The value of LP token is based on the pool reserves. Fees are actually distributed to the pool only upon closing the position, hence why the attack described is possible.

rickkk137

invalid it is intended design

```
def f(mv: uint256, pv: uint256, ts: uint256) -> uint256:
  if ts == 0: return mv
  else      : return (mv * ts) / pv
```

```
def g(lp: uint256, ts: uint256, pv: uint256) -> uint256:
  return (lp * pv) / ts
```

Pool contract uses above formola to compute amount of LP token to mint and also burn value for example: pool VEL-STX: VEL reserve = 1000 STX reserve = 1000 VEL/STX price = $1 LP total_supply = 1000 **pv stand for pool value and ts stand for total_supply and mv stand for mint value** LP_price = ts / pv = 1000 / 1000 = $1 its mean if user deposit $1000 into pool in result get 1000 LP token and for burn burn_value = lp_amount * pool_value / total_supply and based on above example total_supply=2000 pool_value=2000 because user deposit $1000 into pool and mint 1000lp token

the issue want to say mailicious user with increase lp price can retrieve borrowing_fee ,but when mailicious users and other users closes their profitable positions pool_value will be decreased[the protocol pay users' profit from pool reserve],hence burn_value become less than usual state

requirment internal state for attack path:

- mailicious user's position be in loss[pool value will be decreased because user collateral will be added to pool reserve]
- other user don't close their profitable positions in that block
- flashloan's cost be low

WangSecurity

I see how it can be viewed as the intended design, but still, the user can bypass the fees essentially and LPs lose them, when they should've been to receive it. Am I missing anything here?

rickkk137

attack path is possible when user's position is in lose and loss amount has to be enough to increase the LP price , note that if loss amount be large enough the position can be eligible for liquidation and the user has to close his position before liquidation bot

spacegliderrrr

@WangSecurity Your comment is correct. Anytime the user is about to close their position (whether it be unprofitable, neutral or slightly profitable), they can perform the attack to avoid paying the borrowing fees, essentially stealing them from the LPs.

WangSecurity

> attack path is possible when user's position is in lose and loss amount has to be enough to increase the LP price , note that if loss amount be large enough the position can be eligible for liquidation and the user has to close his position before liquidation bot

So these are the constraints:

1. Loss should be relatively large so it increases the LP price.
2. The attacker has to perform the attack before the liquidation bot liquidates their position which is also complicated by the private mempool on Bob, so we cannot see the transaction of the liquidation bot.

Are the above points correct and is there anything missing?

deadrosesxyz

No, loss amount does not need to be large. Attack can be performed on any non-profitable position, so the user avoids paying fees.

WangSecurity

In that case, I agree that it's an issue that the user can indeed bypass the fees and prevent the LPs from receiving it. Also, I don't see the pre-condition of the position being non-profitable as an extensive limitation. Moreover, since it can be any non-profitable position, then there is also no requirement for executing the attack before the liquidation bot (unless the position can be liquidated as soon as it's non-profitable).

Thus, I see that it should be high severity. If something is missing or these limitations are extensive in the context of the protocol, please let me know. Planning to reject the escalation, but increase severity to high.

WangSecurity

Result: High Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- KupiaSecAdmin: rejected

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Velar-co/gl-sherlock/pull/1

Issue M-1: LPs will withdraw more value than deposited during pegged token de-peg events

Source: https://github.com/sherlock-audit/2024-08-velar-artha-judging/issues/52

The protocol has acknowledged this issue.

Found by

4gontuk, KupiaSec

Summary

The CONTEXT function in gl-sherlock/contracts/api.vy uses the `<quote-token>/USD` price for valuation, assuming a 1:1 peg between the quote token and USD. This assumption can fail during de-peg events, leading to incorrect valuations and potential exploitation.

Root Cause

The `CONTEXT` function calls the `price` function from the `oracle` contract to get the price of the quote token. This price is adjusted based on the `quote_decimals`, implying it is using the `<quote-token>/USD` price for valuation.

Detailed Breakdown

1. `CONTEXT` **Function in** `api.vy`: The `CONTEXT` function calls the price function from the oracle contract to get the price of the quote token.

```
def CONTEXT(
    base_token : address,
    quote_token: address,
    desired    : uint256,
    slippage   : uint256,
    payload    : Bytes[224]
) -> Ctx:
  base_decimals : uint256 = convert(ERC20Plus(base_token).decimals(), uint256)
  quote_decimals: uint256 = convert(ERC20Plus(quote_token).decimals(), uint256)
  # this will revert on error
  price         : uint256 = self.ORACLE.price(quote_decimals,
                                              desired,
                                              slippage,
                                              payload)
  return Ctx({
```

```
    price         : price,
    base_decimals : base_decimals,
    quote_decimals: quote_decimals,
  })
```

2. `price` **Function in** `oracle.vy`: The `price` function in oracle.vy uses the extract_price function to get the price from the oracle.

```
################################################################################
TIMESTAMP: public(uint256)

@internal
def extract_price(
    quote_decimals: uint256,
    payload       : Bytes[224]
) -> uint256:
  price: uint256 = 0
  ts   : uint256 = 0
  (price, ts) = self.EXTRACTOR.extractPrice(self.FEED_ID, payload)

  # Redstone allows prices ~10 seconds old, discourage replay attacks
  assert ts >= self.TIMESTAMP, "ERR_ORACLE"
  self.TIMESTAMP = ts

  # price is quote per unit base, convert to same precision as quote
  pd   : uint256 = self.DECIMALS
  qd   : uint256 = quote_decimals
  s    : bool    = pd >= qd
  n    : uint256 = pd - qd if s else qd - pd
  m    : uint256 = 10 ** n
  p    : uint256 = price / m if s else price * m
  return p

################################################################################
PRICES: HashMap[uint256, uint256]

@internal
def get_or_set_block_price(current: uint256) -> uint256:
  """
  The first transaction in each block will set the price for that block.
  """
  block_price: uint256 = self.PRICES[block.number]
  if block_price == 0:
    self.PRICES[block.number] = current
    return current
  else:
    return block_price
```

```
##########################################################################
@internal
@pure
def check_slippage(current: uint256, desired: uint256, slippage: uint256) -> bool:
  if current > desired: return (current - desired) <= slippage
  else                : return (desired - current) <= slippage

@internal
@pure
def check_price(price: uint256) -> bool:
  return price > 0

# eof
```

3. `extract_price` **Function in** `oracle.vy`: The `extract_price` function adjusts the price based on the `quote_decimals`, which implies it is using the `<quote-token>/USD` price for valuation.

Impact

During a de-peg event, LPs can withdraw more value than they deposited, causing significant losses to the protocol.

Attack Path

1. **Deposit**: Attacker deposits 1 BTC and 50,000 USDT when 1 BTC = 50,000 USD.
2. **De-peg Event**: The pegged token (USDT) de-pegs to 0.70 USD.
3. **Withdraw**: Attacker withdraws their funds, exploiting the incorrect assumption that 1 USDT = 1 USD.

Proof of Concept (PoC)

1. **Deposit**:

```
@external
def mint(
  base_token  : address, #ERC20
  quote_token : address, #ERC20
  lp_token    : address, #ERC20Plus
  base_amt    : uint256,
  quote_amt   : uint256,
  desired     : uint256,
  slippage    : uint256,
  payload     : Bytes[224]
```

```
) -> uint256:
  """
  @notice            Provide liquidity to the pool
  @param base_token  Token representing the base coin of the pool (e.g. BTC)
  @param quote_token Token representing the quote coin of the pool (e.g. USDT)
  @param lp_token    Token representing shares of the pool's liquidity
  @param base_amt    Number of base tokens to provide
  @param quote_amt   Number of quote tokens to provide
  @param desired     Price to provide liquidity at (unit price using onchain
                     representation for quote_token, e.g. 1.50$ would be
                     1500000 for USDT with 6 decimals)
  @param slippage    Acceptable deviaton of oracle price from desired price
                     (same units as desired e.g. to allow 5 cents of slippage,
                     send 50000).
  @param payload     Signed Redstone oracle payload
  """
  ctx: Ctx = self.CONTEXT(base_token, quote_token, desired, slippage, payload)
  return self.CORE.mint(1, base_token, quote_token, lp_token, base_amt, quote_amt,
  ↪  ctx)
```

2. **De-peg Event**: The pegged token de-pegs to 0.70 USD (external event).
3. **Withdraw**:

```
def burn(
  base_token  : address,
  quote_token : address,
  lp_token    : address,
  lp_amt      : uint256,
  desired     : uint256,
  slippage    : uint256,
  payload     : Bytes[224]
) -> Tokens:
  """
  @notice            Withdraw liquidity from the pool
  @param base_token  Token representing the base coin of the pool (e.g. BTC)
  @param quote_token Token representing the quote coin of the pool (e.g. USDT)
  @param lp_token    Token representing shares of the pool's liquidity
  @param lp_amt      Number of LP tokens to burn
  @param desired     Price to provide liquidity at (unit price using onchain
                     representation for quote_token, e.g. 1.50$ would be
                     1500000 for USDT with 6 decimals)
  @param slippage    Acceptable deviaton of oracle price from desired price
                     (same units as desired e.g. to allow 5 cents of slippage,
                     send 50000).
  @param payload     Signed Redstone oracle payload
  """
  ctx: Ctx = self.CONTEXT(base_token, quote_token, desired, slippage, payload)
  return self.CORE.burn(1, base_token, quote_token, lp_token, lp_amt, ctx)
```

4. **Incorrect Price Calculation:**

```
def CONTEXT(
    base_token : address,
    quote_token: address,
    desired    : uint256,
    slippage   : uint256,
    payload    : Bytes[224]
) -> Ctx:
  base_decimals : uint256 = convert(ERC20Plus(base_token).decimals(), uint256)
  quote_decimals: uint256 = convert(ERC20Plus(quote_token).decimals(), uint256)
  # this will revert on error
  price         : uint256 = self.ORACLE.price(quote_decimals,
                                              desired,
                                              slippage,
                                              payload)
  return Ctx({
    price         : price,
    base_decimals : base_decimals,
    quote_decimals: quote_decimals,
  })
```

Mitigation

To mitigate this issue, the protocol should use the `<base-token>/<quote-token>` price directly if available, or derive it from the `<base-token>/USD` and `<quote-token>/USD` prices. This ensures accurate valuations even if the quote token de-pegs from USD.

Discussion

mePopye

Escalate

On behalf of the watson

sherlock-admin3

> Escalate
> On behalf of the watson

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

WangSecurity

After additionally considering this issue, here's my understanding. Let's assume a scenario of 30% depeg and USDT = 0.7 USD.

1. The pool has 10 BTC and 500k USDT.
2. User deposits 1 BTC and 50k USDT, assuming 1 BTC = 50k USDT = 50k USD.
3. USDT depegs to 0.7 USD, i.e. 1 USDT = 0.7 USD. Then BTC = 50k USD = ~71.5k USDT.
4. The user withdraws 1 BTC and 50k USDT. They receive it because the code still considers 1 USDT = 1 USD. There are 10 BTC and 500k USDT left in the contracts.
5. The protocol didn't lose any funds, the amount of BTC and USDT remained the same as it was before the depeg.
6. But, in reality, the user has withdrawn 50k worth of BTC and 35k worth of USDT since 1 USDT = 0.7 USD.
7. Hence, if the protocol accounted for the depeg, there had to be 10 BTC and 515k USDT left in the contract after the user had withdrawn during the depeg.

Hence, even though it's not a direct loss of funds but a loss in value, this should be a valid medium (considering depeg as an extensive limitation). Thus, planning to accept the escalation and validate with medium severity. The duplicate is #113, are there any additional duplicates?

WangSecurity

Result: Medium Has duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- mePopye: accepted

Issue M-2: Funding fee will be zero because of precision loss

Source: https://github.com/sherlock-audit/2024-08-velar-artha-judging/issues/72

The protocol has acknowledged this issue.

Found by

aslanbek, pashap9990

Summary

funding fee in some cases will be zero because of precision loss

Root Cause

`long_utilization = base_interest / (base_reserve / 100)` `short_utilization = quote_interest / (quote_reserve / 100)`

`borrow_long_fee = max_fee * long_utilization` [min = 10] `borrow_short_fee = max_fee * short_utilization` [min = 10]

`funding_fee_long = borrow_long_fee * (long_utilization - short_utilization) / 100`

`funding_fee_short = borrow_short_fee * (short_utilization - long_utilization) / 100`

let's assume alice open a long position with min collateral[5e6] and leverage 2x when btc/usdt $50,000

long_utilization = 0.0002e8 / (1000e8 / 100) = 2e4 / 1e9 = 0.00002[round down => 0]

short_utilization = 0 / 1e12 /100 = 0

borrowing_long_fee = 100 * (0) = 0 [min fee = 1] ==> 10 borrowing_short_fee = 100 * (0) = 0 [min fee = 1] ==> 10

funding_fee_long = 10 * (0) = 0 funding_fee_short = 10 * 0 = 0

1000 block passed

funding_paid = 5e6 * 1000 * 0 / 1_000_000_000 = 0 borrowing_paid = (5e6) * (1000 * 10) / 1_000_000_000 = 50

** long_utilization and short_utilization are zero until **base_reserve / 100 >= base_interest** and **quote_reserve / 100 >= quote_interest**

Internal pre-conditions

pool status: ”base_reserve” : 1000e8 BTC ”quote_reserve” : 1,000,000e6 USDT

```
"collector"     : "0xCFb56482D0A6546d17535d09f571F567189e88b3",
    "symbol"         : "WBTCUSDT",
    "base_token"     : "0x03c7054bcb39f7b2e5b2c7acb37583e32d70cfa3",
    "quote_token"    : "0x05d032ac25d322df992303dca074ee7392c117b9",
    "base_decimals" : 8,
    "quote_decimals": 6,
    "blocktime_secs": 3,
    "parameters"     : {
        "MIN_FEE"              : 1,
        "MAX_FEE"              : 100,
        "PROTOCOL_FEE"         : 1000,
        "LIQUIDATION_FEE"      : 2,

        "MIN_LONG_COLLATERAL"  : 5000000,
        "MAX_LONG_COLLATERAL"  : 100000000000,
        "MIN_SHORT_COLLATERAL" : 10000,
        "MAX_SHORT_COLLATERAL" : 200000000,

        "MIN_LONG_LEVERAGE"    : 1,
        "MAX_LONG_LEVERAGE"    : 10,
        "MIN_SHORT_LEVERAGE"   : 1,
        "MAX_SHORT_LEVERAGE"   : 10,

        "LIQUIDATION_THRESHOLD": 5
    },
    "oracle": {
        "extractor": "0x3DaF1A3ABF9dd86ee0f7Dd13a256400d01866E04",
        "feed_id"  : "BTC",
        "decimals" : 8
    }
```

Code Snippet

https://github.com/sherlock-audit/2024-08-velar-artha/blob/main/gl-sherlock/contracts/params.vy#L63

Impact

Funding fee always is lower than what it really should be

PoC

Place below test in tests/test_positions.py and run with `pytest -k test_precision_loss -s`

```
def test_precision_loss(setup, open,VEL, STX, long, positions, pools):
    setup()
    open(VEL, STX, True, d(5), 10, price=d(50000), sender=long)
    chain.mine(1000)
    fee = positions.calc_fees(1)
    assert fee.funding_paid == 0
```

Mitigation

1-scale up long_utilzation and short_utilzation 2-set min value for long_utilzation and short_utilzation

Discussion

sherlock-admin3

Escalate
Let's assume interest = 1_099_999e18 reserve=10_000_000e18 max_fee = 100 long_utilization = interest / reserve / 100 = 1_099_999e18 / 10_000_000e18 / 100 = 10.99 ~ 10 //round down borrowing_fee = max_fee * long_utillization / 100 = 100 * 10.99 / 100 = 10.99 after one year
//result without precision loss block_per_year = 15_778_800 funding_fee_sum = block_per_year * funding_fee = 15778800 * 10.99 = 173,409,012 borrowing_long_sum = block_per_year * borrowing_fee = 15778800 * 10.99 = 173,409,012
borrowing_paid = collateral * borrowing_long_sum / DENOM = 1_099_999e18 * 173,409,012 / 1e9 = 190,749e18
funding_paid = collateral * funding_fee_sum / DENOM = 190,749e18
//result with precision loss block_per_year = 15_778_800 funding_fee_sum = block_per_year * funding_fee = 15778800 * 10 = 157788000 borrowing_long_sum = block_per_year * borrowing_fee = 15778800 * 10 = 157788000
borrowing_paid = collateral * borrowing_long_sum / DENOM = 1_099_999e18 * 157788000 / 1e9 = 173,566e18
funding_paid = collateral * funding_fee_sum / DENOM = 173,566e18
result:1% difference exists in result

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

WangSecurity

To clarify, due to this precision loss, there will be a 1% loss of the funding fee or am I missing something?

rickkk137

open_interest = 1,099,999e6 reserve = 10,000,000e6

long_util = open_interest / reserve / 100 = 10.99 borrowing_fee = max_fee * long_util / 100 = 100 * 10.99 / 100 = 10.99 funding_fee = borrowing_fee * long_util / 100 = 10.99 * 10.99 / 100 = 1.20

lets assume there is a long position with 10000e6 colleral and user want to close his position after a year[15778800 blocks per year]

result without precision loss

borrowing_paid = collateral * borrowing_sum / DENOM

borrowing_paid = 10,000e6 * 15778800 * 10.99 / 1e9 = 1,734,090,120[its mean user has to pay $1734 as borrowing fee] funding_paid = collateral * funding_sum / DENOM = 10,000e6 * 1.20 * 15778800 / 1e9 = 189,345,600[its mean user has to pay $189 as funding fee]

result with precision loss

borrowing_paid = collateral * borrowing_sum / DENOM borrowing_paid = 10,000e6 * 15778800 * 10 / 1e9 = 1,577,880,000[its mean user has to pay $1,577 as borrowing fee] funding_paid = collateral * funding_sum / DENOM = 10,000e6 * 1 * 15778800 / 1e9 = 157,788,000[its mean user has to pay $157 as funding fee]

LPs loss = $157[~1%] user pay $32 less than expected [32 * 100 / 189 ~ 16%]

rickkk137

#60 dup of this issue

WangSecurity

I agree that this issue is correct and indeed identifies the precision loss showcasing the 1% loss. Planning to accept the escalation and validate with medium severity.

WangSecurity

Result: Medium Has duplicates

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- rickkk137: accepted

Issue M-3: LPs cannot specify min amount received in burn function, causing loss of fund for them

Source: https://github.com/sherlock-audit/2024-08-velar-artha-judging/issues/74

The protocol has acknowledged this issue.

Found by

PASCAL, pashap9990

Summary

LPs cannot set minimum base or quote amounts when burning LP tokens, leading to potential losses due to price fluctuations during transactions.

Root Cause

LPs cannot set the base received amount and the quote received amount

Impact

LPs may receive significantly lower amounts than expected when burning LP tokens, resulting in financial losses

Code Snippet

https://github.com/sherlock-audit/2024-08-velar-artha/blob/main/gl-sherlock/contracts/api.vy#L104

Internal pre-conditions

Consider change config in tests/conftest.py I do it for better understanding,Fee doesn't important in this issue

```
PARAMS = {
  'MIN_FEE'               : 0,
  'MAX_FEE'               : 0,

  'PROTOCOL_FEE'          : 1000
```

```
    ...
}
```

PoC

Textual PoC: we assume protocol fee is zero in this example 1-Bob mints 20,000e6 LP token[base_reserve:10,000e6, quote_reserve:10,000e6] 2-Alice opens long position[collateral 1000 STX, LEV:5,Price:1] 3-Price goes up til $2 4-Bob calls calc_burn[lp_amt:10,000e6,total_supply:20,000e6][return value:base 3750 VEL,quote 7500 STX] 5-Bob calls burn with above parameters 6-Alice calls close position 7-Alice's tx executed before Bob's tx 8-Bob's tx will be executed and Bob gets 3875 VEL and 4750 STX 9-Bob losts $2500 **Coded PoC:** place this test in tests/test_positions.py and run this command `pytest -k test_lost_assets -s`

```
def test_lost_assets(setup, VEL, STX, lp_provider, LP, pools, math, open, long,
↪  close, burn):
    setup()
    #Alice opens position
    open(VEL, STX, True, d(1000), 5, price=d(1), sender=long)

    reserve = pools.total_reserves(1)
    assert reserve.base == 10000e6
    assert reserve.quote == 10000e6
    #Bob calls calc_burn, Bob's assets in term of dollar is $15,000
    amts = pools.calc_burn(1, d(10000)  , d(20000), ctx(d(2)))

    assert amts.base == 3752500000
    assert amts.quote == 7495000000

    #Alice closes her position
    bef = VEL.balanceOf(long)
    close(VEL, STX, 1, price=d(2), sender=long)
    after = VEL.balanceOf(long)

    vel_bef = VEL.balanceOf(lp_provider)
    stx_bef = STX.balanceOf(lp_provider)

    amts = pools.calc_burn(1, d(10000)  , d(20000), ctx(d(2)))
    assert amts.base == 3877375030
    assert amts.quote == 4747749964
    #Bob's tx will be executed
    burn(VEL, STX, LP, d(10000), price=d(2), sender=lp_provider)

    vel_after = VEL.balanceOf(lp_provider)
    stx_after = STX.balanceOf(lp_provider)
```

```
    print("vel_diff:", (vel_after - vel_bef) / 1e6)#3877.37503 VEL
    print("stx_diff:", (stx_after - stx_bef) / 1e6)#4747.749964 STX
    #received values in term of dollar is ~ $12,500,Bob lost ~ $2500
```

Mitigation

Consider adding min_base_amount and min_quote_amount to the burn function's params or adding min_assets_value for example when the price is $2 LPs set this param to $14800, its mean received value worse has to be greater than $14800

Discussion

rickkk137

Escalate LP token price directly compute based pool reserve and total supply lp token and the issue clearly states received amount can be less than expected amount and in Coded PoC liquidity provider expected $15000 but in result get $12500

loss = $2500[1.6%]

> Causes a loss of funds but requires certain external conditions or specific states, or a loss is highly constrained. The loss of the affected party must exceed 0.01% and 10 USD

sherlock-admin3

> Escalate
>
> Slippage related issues showing a definite loss of funds with a detailed explanation for the same can be considered valid high

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

WangSecurity

However, the LPs can add input slippage parameters, i.e. `desired` and `slippage` to mitigate these issues. Am I missing something here?

rickkk137

@WangSecurity `desired` and `slippage` just has been used to control price which fetch from oracle and protocol uses of that for converting quote to base or base to quote to compute total pool's reserve in terms of quote token but there is 2 tips here

- burn value = lp_amt * pool_reserve / total_supply_lp

- pool_reserve will be decreased when users closes their profitable positions

let's examine an example with together: u have 1000 lp token and u want convert them to usd and pool_reserve and total_supply_lp is 1000 in our example, burn_value = lp_amt * pool_reserve / total_supply = 1000 * 1000 / 1000 = 1000 usd based on above value u send your transaction to network but a close profitable transaction will be executed before your transaction and get $100 as payout,its mean pool reserve is 900 burn_value = lp_amt * pool_reserve / total_supply = 1000 * 900 / 1000 = 900 usd u get $900 instead of $1000 and u cannot control this situation as a user

WangSecurity

Yeah, I see, thank you. Indeed, the situation which would cause an issue to the user happens after the slippage is checked and this conversion cannot be controlled by the user. Planning to accept the escalation and validate with medium severity. Are there any duplicates (non-escalated; I see there are some escalations about slippage; I will consider them as duplicates if needed)?

WangSecurity

Result: Medium Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- rickkk137: accepted

Issue M-4: Invalid Redstone oracle payload size prevents the protocol from working properly

Source: https://github.com/sherlock-audit/2024-08-velar-artha-judging/issues/75

The protocol has acknowledged this issue.

Found by

KupiaSec

Summary

In api contract, it uses 224 bytes as maximum length for Redstone's oracle payload, but oracle price data and signatures of 3 signers exceeds 225 bytes thus reverting transactions.

Vulnerability Detail

In every external function of api contract, it uses 224 bytes as maximum size for Redstone oracle payload.

However, the `RedstoneExtractor` requires oracle data from at least 3 unique signers, as implemented in `PrimaryProdDataServiceConsumerBase` contract. Each signer needs to send token price information like token identifier, price, timestamp, etc and 65 bytes of signature data. Just with basic calculation, the oracle payload size exceeds 224 bytes.

Here's some proof of how Redstone oracle data is used:

- Check one of transactions from here that uses Redstone oracle.
- One of transaction is this one on Avalanche, which has 9571 bytes of data.
- Check this Blocksec Explorer, and it also shows the oracle data of 3 signers are passed.

As shown from the proof above, the payload size of Redstone data is huge, so setting 224 bytes as upperbound reverts transactions.

Impact

Protocol does not work because the payload array size limit is too small.

Code Snippet

https://github.com/sherlock-audit/2024-08-velar-artha/blob/18ef2d8dc0162aca79bd71710f08a3c18c94a36e/gl-sherlock/contracts/api.vy#L83
https://github.com/sherlock-audit/2024-08-velar-artha/blob/18ef2d8dc0162aca79bd71710f08a3c18c94a36e/gl-sherlock/contracts/api.vy#L58

Tool used

Manual Review

Recommendation

The upperbound size of payload array should be increased to satisfy Redstone oracle payload size.

Discussion

KupiaSecAdmin

Escalate

Information required for this issue to be rejected.

sherlock-admin3

> Escalate
> Information required for this issue to be rejected.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

WangSecurity

@KupiaSecAdmin, could you help me understand how did you get the 9571 data size? can't see it in the links, but I assume I'm missing it somewhere.

KupiaSecAdmin

@WangSecurity - The data size is fetched from this transaction which uses RedStone oracle. The calldata size of the transaction is 9574 bytes, but the function receives 2 variables which sums up 64 bytes, so remaining bytes `9574 - 64 = 9510` bytes of data is for RedStone oracle payload.

Maybe, it includes some additional oracle data, but the main point is that oracle price data of 3 oracles can't fit in 224 bytes. Because each oracle signature is 65 bytes which

means it has 195 bytes of signature, and also the oracle data should include price, timestamp, and token name basically. So it never fits in 224 bytes.

rickkk137

i used minimal foundry package to generate some payload

```
bytes memory redstonePayload0 = getRedstonePayload("BTC:120:8,ETH:69:8");
//result length 440 chars
4254430000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪  0000000000000000000000000000000002cb417800455448000000000000000000000000000000
↪  0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪  00000019b45a50001921f3ed12d000000200000022af29790252e02178ff033567c55b4145e48cb
↪  ec0434729f363a01675d9a88d618583b722a092e3e6d9944b69c79cadf793b2950f1a99fd595933
↪  609984ef1c01c0001000000000002ed57011e0000
 bytes memory redstonePayload = getRedstonePayload("BTC:120:8");
//result length 312 chars
4254430000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪  0000000000000000000000000000000002cb41780001921f3ed1d400000020000001e3cabe40
↪  b42498dccea014557946f3bae4d29783c9dc7deb499c5a2d6d1901412cba9924d1c1fdfc429c073
↪  61ed9fab2789e191791a31ecdbbd77ab95a373f491c0001000000000002ed57011e0000

def mint(
  base\_token  : address, \#ERC20
  quote\_token : address, \#ERC20
  lp\_token    : address, \#ERC20Plus
  base\_amt    : uint256,
  quote\_amt   : uint256,
  desired     : uint256,
  slippage    : uint256,
 @>>> payload     : Bytes[224]
```

payload parameter's length is 224 its mean we can pass a string with max length 448 character ,hence we can pass above payload to api functions,I want to say length of payload depend on number of symbols which we pass to get payload and max size for 2 symbol is 440 character

KupiaSecAdmin

@rickkk137 - Bytes type is different from strings as documented here, Bytes[224] means 224 bytes.

rickkk137

Each pair of hexadecimal digits represents one byte and in above examples first example's length is [440 / 2] 220 bytes and second one's length is [312/2] 156 bytes, further more in redstoneExtractor contract developer just uses index 0 which its mean requestRedstonePayload function just get one symbol as a parameter in Velar

KupiaSecAdmin

@rickkk137 - Seems you misunderstand something here. You generated Redstone oracle

payload using their example repo that results in 220 bytes, this is the oracle data of one signer. For Verla, it requires oracle data of 3 signers so that it can take median price of 3 prices.

And I agree with the statement that it uses one symbol, yes Verla only requires BTC price from Redstone oracle.

rickkk137

Data is packed into a message according to the following structure

> The data signature is verified by checking if the signer is one of the approved providers

its mean just one sign there is in every payload

Symbol	Value	Timestamp	Size(n)	Signature
32b	32b	32b	1b	65b

max size for 1 symbol: 32 + 32 + 32 + 1 + 65 = 172 bytes
https://github.com/redstone-finance/redstone-evm-connector

```
function getAuthorisedSignerIndex(
    address signerAddress
  ) public view virtual override returns (uint8) {
    if (signerAddress == 0x8BB8F32Df04c8b654987DAaeD53D6B6091e3B774) {
      return 0;
    } else if (signerAddress == 0xdEB22f54738d54976C4c0fe5ce6d408E40d88499) {
      return 1;
    } else if (signerAddress == 0x51Ce04Be4b3E32572C4Ec9135221d0691Ba7d202) {
      return 2;
    } else if (signerAddress == 0xDD682daEC5A90dD295d14DA4b0bec9281017b5bE) {
      return 3;
    } else if (signerAddress == 0x71d00abE308806A3bF66cE05CF205186B0059503) {
      return 4;
    } else {
      revert SignerNotAuthorised(signerAddress);
    }
  }
```

WangSecurity

@KupiaSecAdmin just a small clarification, the idea that you need 3 signers in payload is based on which documentation?

KupiaSecAdmin

@WangSecurity - It comes from the RedStone implementation that Verla uses:
https://github.com/redstone-finance/redstone-oracles-monorepo/blob/2bbf16cbbaa36f7046034dbbd968f3673a0657e8/packages/evm-connector/contracts/data-services/

PrimaryProdDataServiceConsumerBase.sol
allowbreak #L12-L14

And you know, usually, using one signer data as oracle causes issue because its data can be malicious, that's how the protocol takes 3 signers and take median price among them.

WangSecurity

Unfortunately, I'm still not convinced enough this is actually a valid finding. Firstly, the transaction linked before, which has 9574 bytes size of calldata and uses the RedStone oracle, doesn't have the Redstone's payload as an input parameter as Velar does it. Secondly, this transaction is on Avalanche, while Velar will be deployed on Bob.

Hence, this is not a sufficient argument that 224 Bytes won't be enough. Thirdly, `payload` is used when calling the extract
allowbreak _price function which doesn't even use that payload. Hence, I don't see a sufficient argument for this being a medium, but before making the decision, I'm giving some time to correct my points.

rickkk137

@WangSecurity u can pass n asset to fetchPayload function and that isn't const its mean payload length is flexible which depend on protocol and when we look at extract_price which just uses index 0 its mean they are suppose to pass just one symbol to redstone function to get payload and base on redstone document payload's length just for one symbol is 172 bytes which is less than 224 bytes

payload_size = n*(32 + 32) + 32 + 1 + 65 payload_size_for_one_asset = 1 * (32 + 32) + 32 + 1 + 65 = 172 bytes payload_size_for_two_asset = 2 * (32 + 32) + 32 + 1 + 65 = 226 bytes payload_size_for_three_asset = 3 * (32 + 32) + 32 + 1 + 65 = 290 bytes ...

KupiaSecAdmin

@WangSecurity - The 9574 bytes of payload is one example of Redstone payload.

The point is that it's obvious the price data can't fit in 224 bytes. As @rickkk137 mentioned, payload size for one asset of one signer is 172 bytes, but Verla requires oracle data of 3 signers, which will result in >500 bytes.

rickkk137

Velar protocol uses version 0.6.1 redstone-evm-connector and in this version `RedstoneConsumerBase::getUniqueSignersThreshold` returns 1 in this path `/ @redstone-finance/evm-connector / contracts / core / RedstoneConsumerBase.sol`,hence just one signer is required https://www.npmjs.com/package/@redstone-finance/evm-connector/v/0.6.1?activeTab=code also when we look at make file we realized Velar's developers directly copy RedstoneConsumerBase contract without any changes,furthermore usingDataService function get unique signer as a parameter and when they restricted payload to 224 bytes its mean they want to pass 1 as a uniqueSignersCount

```
const wrappedContract = WrapperBuilder.wrap(contract).usingDataService({
```

```
    dataServiceId: "redstone-rapid-demo",
@>>>    uniqueSignersCount: 1,
    dataFeeds: ["BTC", "ETH", "BNB", "AR", "AVAX", "CELO"],
  });
```

https://github.com/redstone-finance/redstone-showroom/blob/0db580be39bdccb9632ee4d8d8c80e4182d8e266/example/getTokensPrices.ts allowbreak #L22

KupiaSecAdmin

@rickkk137 - Check `RedstoneExtractor.sol`, the contract inherits from `PrimaryProdDataServiceConsumerBase` contract which is located in "./vendor/data-services/PrimaryProdDataServiceConsumerBase.sol" that is copied after make, which returns 3 as number of signers.

WangSecurity

As I understand, @KupiaSecAdmin is indeed correct here, and the current payload size won't work when the contracts are deployed on the Live chain. After clarifying with the sponsor, they've said they used 224 only for testnet and will increase it for the live chain, but there's no info about it in README or code comments. Hence, this should be indeed a valid issue. Planning to accept the escalation and validate with Medium severity. Medium severity because there is no loss of funds, and the second definition of High severity excludes contracts not working:

> Inflicts serious non-material losses (doesn't include contract simply not working).

Hence, medium severity is appropriate:

> Breaks core contract functionality, rendering the contract useless or leading to loss of funds.

Are there any duplicates?

KupiaSecAdmin

@WangSecurity - Agree with having this as Medium severity. Thanks for your confirmation.

WangSecurity

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- KupiaSecAdmin: accepted

Issue M-5: Not decreasing oracle timestamp validation leads to DoS for protocol users

Source: https://github.com/sherlock-audit/2024-08-velar-artha-judging/issues/79

The protocol has acknowledged this issue.

Found by

KupiaSec

Summary

The protocol only allows equal or increased timestamp of oracle prices whenever an action happens in the protocol. This validation is wrong since it will lead to DoS for users.

Vulnerability Detail

The protocol uses RedStone oracle, where token prices are added as a part of calldata of transactions. In RedStone oracle, it allows prices from 3 minutes old upto 1 minute in the future, as implemented in `RedstoneDefaultsLib.sol`.

In `oracle.vy`, it extracts the token price from the RedStone payload, which also includes the timestamp of which the prices were generated. As shown in the code snippet, the protocol reverts when the timestamp extracted from the calldata is smaller than the stored timestamp, thus forcing timestamps only increase or equal to previous one. This means that the users who execute transaction with price 1 minute old gets reverted when there is another transaction executed with price 30 seconds old.

NOTE: The network speed of all around the world is not same, so there can be considerable delays based on the location, api availability, etc.

By abusing this vulnerability, an attacker can regularly make transactions with newest prices which will revert all other transactions with slightly older price data like 10-20 seconds older, can be all reverted.

Impact

The vulnerability causes DoS for users who execute transactions with slightly older RedStone oracle data.

Code Snippet

https://github.com/sherlock-audit/2024-08-velar-artha/blob/18ef2d8dc0162aca79bd71710f08a3c18c94a36e/gl-sherlock/contracts/oracle.vy#L91-L93

Tool used

Manual Review

Recommendation

It's recommended to remove that non-decreasing timestamp validation. If the protocol wants more strict oracle price validation than the RedStone does, it can just use the difference between oracle timestamp and current timestamp.

Discussion

KupiaSecAdmin

Escalate

Information required for this issue to be rejected.

sherlock-admin3

> Escalate
> Information required for this issue to be rejected.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

rickkk137

the protocol for every action fetch a redstone's payload and attach that to the transaction let's assume: User A:fetch payload a in T1 user B:fetch payload b in T2 user C:fetch payload c in T3 and T3 > T2 > T1 and block.timestamp-[T1,T2,T3] < 3 minutes(I mean all of them are valid) if all of them send their Txs to the network sequence is very important its mean just with this sequence [t1,t2,t3] none of TXs wouldn't be reverted, if t2 will be executed first then t1 will be reverted and if t3 will be executed first t1 and t2 will be reverted

```
 contract RedstoneExtractor is PrimaryProdDataServiceConsumerBase {
+  uint lastTimeStamp;
+  uint price;
   function extractPrice(bytes32 feedId, bytes calldata)
```

```
       public view returns(uint256, uint256)
   {
+    if(block.timestamp - lastTimeStamp > 3 minutes){
     bytes32[] memory dataFeedIds = new bytes32[](1);
     dataFeedIds[0] = feedId;
     (uint256[] memory values, uint256 timestamp) =
         getOracleNumericValuesAndTimestampFromTxMsg(dataFeedIds);
     validateTimestamp(timestamp); //!!!
-    return (values[0], timestamp);
+    lastTimeStamp = block.timestamp;
+    price = values[0];
+    }
+    return (price, lastTimeStamp);
   }
 }
```

But there isn't loss of funds ,users can repeat their TXs

KupiaSecAdmin

@rickkk137 - Thanks for providing the PoC. Of course there's no loss of funds, and users can repeat their transactions but those can be reverted again. Overall, there's pretty high chance that users' transactions will bereverted.

rickkk137

I agree with u and this can be problematic and Here's an example of this approach but the issue's final result depend on sherlock rules

WangSecurity

Not sure I understand how this can happen.

For example, we fetched the price from RedStone at timestamp = 10. Then someone fetches the price again, and for the revert to happen, the timestamp of that price has to be 9, correct?

How could that happen? Is it the user who chooses the price?

KupiaSecAdmin

@WangSecurity - As a real-world example, there will be multiple users who get price from RedStone and call Verla protocol with that price data. NOTE: price data(w/ timestamp) is added as calldata for every call. So among those users, there will be users calling protocol with price timestamp 8, some with 9, some with 10.

If one call with price timestamp 10 is called, other remaining calls with price timestamp 8 and 9 will revert.

WangSecurity

Thank you for this clarification. Indeed, this is possible, and it can happen even intentionally. On the other hand, this is only one-block DOS and the users could proceed with the transactions in the next block (assuming they use the newer price). However, this

vulnerability affects the opening and closing of positions, which are time-sensitive functions. Hence, this should be a valid medium based on this rule:

> Griefing for gas (frontrunning a transaction to fail, even if can be done perpetually) is considered a DoS of a single block, hence only if the function is clearly time-sensitive, it can be a Medium severity issue.

Planning to accept the escalation and validate with medium severity.

WangSecurity

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- KupiaSecAdmin: accepted

Issue M-6: Usage of `tx.origin` to determine the user is prone to attacks

Source: https://github.com/sherlock-audit/2024-08-velar-artha-judging/issues/82

The protocol has acknowledged this issue.

Found by

Bauer, Greed, Japy69, KupiaSec, Waydou, bughuntoor, ctf_sec, y4y

Summary

Usage of `tx.origin` to determine the user is prone to attacks

Vulnerability Detail

Within `core.vy` to user on whose behalf it is called is fetched by using `tx.origin`.

This is dangerous, as any time a user calls/ interacts with an unverified contract, or a contract which can change implementation, they're put under risk, as the contract can make a call to `api.vy` and act on user's behalf.

Usage of `tx.origin` would also break compatibility with Account Abstract wallets.

Impact

Any time a user calls any contract on the BOB chain, they risk getting their funds lost. Incompatible with AA wallets.

Code Snippet

https://github.com/sherlock-audit/2024-08-velar-artha/blob/main/gl-sherlock/contracts/core.vy#L166

Tool used

Manual Review

Recommendation

Instead of using `tx.origin` in `core.vy`, simply pass `msg.sender` as a parameter from `api.vy`

Discussion

T1MOH593

Escalate

Noticed there were 19 escalations on preliminary valid issues. This is final escalation to make it 20/20 □

sherlock-admin3

> Escalate
> Noticed there were 19 escalations on preliminary valid issues. This is final escalation to make it 20/20 □

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

WangSecurity

bruh

WangSecurity

Planning to reject the escalation and leave the issue as it is.

WangSecurity

Result: Medium Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- T1MOH593: rejected

Issue M-7: Funding Paid != Funding Received

Source: https://github.com/sherlock-audit/2024-08-velar-artha-judging/issues/83

The protocol has acknowledged this issue.

Found by

0x37, 0xbranded, bughuntoor

Summary

Due to special requirements around receiving funding fees for a position, the funding fees received can be less than that paid. These funding fee payments are still payed, but a portion of them will not be withdrawn, and become stuck funds. This also violates the contract specification that `sum(funding_received) = sum(funding_paid)`.

Vulnerability Detail

In calc_fees there are two special conditions that impact a position's receipt of funding payments:

If the position has run out of collateral by the time it is being closed, he will receive none of his share of funding payments. Additionally, if the available collateral is not high enough to service the funding fee receipt, he will receive only the greatest amount that is available.

These funding fee payments are still always made (deducted from remaining collateral), whether they are received or not:

When a position is closed under most circumstances, the pool will have enough collateral to service the corresponding fee payment:

When positions are closed, the original collateral (which was placed into the pool upon opening) is removed. However, the amount of funding payments a position made is added to the pool for later receipt. Thus, when positions are still open there is enough position collateral to fulfill the funding fee payment and when they close the funding payment made by that position still remains in the pool.

Only when the amount of funding a position paid exceeded its original collateral, will there will not be enough collateral to service the receipt of funding fees, as alluded to in the comments. However, it's possible for users to pay the full funding fee, but if the borrow fee exceeds the collateral balance remaining thereafter, they will not receive any funding fees. As a result, it's possible for funding fees to be paid which are never received.

Further, even in the case of funding fee underpayment, setting the funding fee received to 0 does not remedy this issue. The funding fees which he underpaid were in a differing token from those which he would receive, so this only furthers the imbalance of fees received to paid.

Impact

`core.vy` includes a specification for one of the invariants of the protocol:

This invariant is clearly broken as some portion of paid funding fees will not be received under all circumstances, so code is not to spec. This will also lead to some stuck funds, as a portion of the paid funding fees will never be deducted from the collateral. This can in turn lead to dilution of fees for future funding fee recipients, as the payments will be distributed evenly to the entire collateral including these stuck funds which will never be removed.

Code Snippet

Tool used

Manual Review

Recommendation

Consider an alternative method of accounting for funding fees, as there are many cases under the current accounting where fees received/paid can fall out of sync.

For example, include a new state variable that explicitly tracks unpaid funding fee payments and perform some pro rata or market adjustment to future funding fee recipients, specifically for *that token*.

Discussion

spacegliderrrr

Escalate

Issue should be invalidated. It does not showcase any Medium-severity impact, but rather just a slightly incorrect code comment. Contest readme did not include said invariant as one that must always be true, so simply breaking it slightly does not warrant Medium severity.

sherlock-admin3

> Escalate
> Issue should be invalidated. It does not showcase any Medium-severity impact, but rather just a slightly incorrect code comment. Contest readme did not include said invariant as one that must always be true, so simply breaking it slightly does not warrant Medium severity.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

msheikhattari

> If the protocol team provides specific information in the README or CODE COMMENTS, that information stands above all judging rules.

The included code comment was an explicit invariant outlined by the team:

```
\#  - the protocol handles the accounting needed to maintain the sytem invariants:
\#    * funding payments match
\#        sum(funding\_received) = sum(funding\_paid)
```

The problematic code was actually intended to enforce this invariant, however it does so incorrectly. In the case of a negative position due to fee underpayment, `sum(funding\_received) > sum(funding\_paid)`. To correct for this, or serve as a deterrent, these positions will not receive their funding payment. However the token in which they underpaid is not the same token that they will not receive funding payment for. As a result the imbalance between funding paid and received is not corrected - it is actually worsened.

Not only that, but users may have paid their full funding payment and the `sum(funding\_received) = sum(funding\_paid)` invariant holds. But if the remaining balance was then not enough to cover their borrow fee, they will not receive their funding payment which would actually cause this invariant to break.

This specification is the source of truth, and the code clearly does not abide by it. The issue proposes alternative methods for accounting funding fee payments to ensure this invariant consistently holds.

msheikhattari

Also I do think this issue is similar to #18 and #93

Will leave it to HoJ if they are combined since the source of the error is the same, but different impacts are described.

This issue discusses the code being not to spec and breaking an invariant. The other two issues mention a known issue from the sponsor of stuck/undistributed funds

WangSecurity

@spacegliderrrr is correct that indeed breaking the invariant from the code comment doesn't make the issue medium-severity necessarily. But, as I understand, it also works as intended. The position pays its fees, but if there is not enough collateral, then position cannot pay the fee and this fee will be unpaid/underpaid. And, IIUC, in this protocol the borrowing fees comes first and if the remaining collateral cannot pay the funding fee in full, then it shouldn't be. Hence, this is also enough of a contextual evidence that the comment is outdated. Do I miss anything here?

spacegliderrrr

@WangSecurity Most of your points are correct. Yes the code works as expected. Funding fee comes before borrowing fee. But for both of them, since due to fail to liquidate position on time, fees can exceed the total position collateral. Because of this, funding fees are paid with priority and if there's anything left, borrowing fees are paid for as much as there's left.

In the case where funding fees are overpaid (for more than the total position collateral), the other side receives these fees in a FIFO order, which is also clearly stated in the comments.

msheikhattari

@spacegliderrrr is correct, funding fees are paid before borrow fees. As such there is no evidence that the comment spec is outdated, nor does any other documentation such as the readme appear to indicate so.

> In the case where funding fees are overpaid (for more than the total position collateral), the other side receives these fees in a FIFO order, which is also clearly stated in the comments.

To elaborate on this point, the vulnerability describes the case that `funding\_received` from the other side is greater than the `funding\_paid`. While it is indeed acknowledged fees are received in FIFO order, this is a problematic mitigation for this issue. This current approach is a questionable means of correcting the imbalance of funding payments to receipts. There are many cases where it not only doesn't correct for the funding payment imbalance, but actually worsens it, as explained above.

Regardless, this seems to be a clearly defined invariant. Not only from this comment but further implied by the logic of this fee handling, which penalizes negative positions to build up some ”insurance” tokens to hedge against the case that funding fees are underpaid. It also intuitively makes sense for this invariant to generally hold as otherwise malfunctions can occur such as failure to close positions; several linked issues reported various related problems stemming from this invariant breaking as well.

WangSecurity

Another question I've got after re-reading the report. The impact section says there will be stuck tokens in the contract which will never be paid. But, as I understand the problem is different. The issue is that these tokens are underpaid, e.g. if the funding fee is 10 tokens, but the remaining collateral is only 8, then there are 2 tokens underpaid. So,

how the are stuck tokens in the contract, if the funding fee is not paid in full. Or it refers to the funding_received being larger than the funding_received actually?

WangSecurity

Since no answer is provided, I assume it's a typo in the report and the contract doesn't have fewer tokens, since the fee is underpaid, not overpaid. But, the issue here is not medium severity, because the position with collateral less than the funding fees and it cannot pay the full fee. Just breaking the invariant is not sufficient for medium severity. Planning to accept the escalation and invalidate the issue.

msheikhattari

Apologies for the delayed response @WangSecurity

Yes, the original statement was as intended. That portion of the report was pointing out that there are two problematic impacts of the current approach that cause `sum(funding\_paid) != sum(funding\_received)`

1. It's possible for `funding\_received` by one side of the pool to exceed the `funding\_paid` by the other side, in the case that a position went negative.
2. It's possible for `funding\_received` to be less than `funding\_paid`, even for positions which did not go insolvent due to funding rates.

The outlined invariant is broken which results in loss of funds / broken functionality. It will prevent closing of some positions in the former case, and will result in some funds being stuck in the latter case.

The current approach of penalizing negative positions is intended to 'build up an insurance' as stated by the team. But as mentioned here, not only does it not remediate the issue it actually furthers the imbalance. The funding tokens have already been underpaid in the case of a negative position - preventing those positions from receiving their fees results in underpayment of their funding as well.

WangSecurity

> It's possible for funding_received by one side of the pool to exceed the funding_paid by the other side, in the case that a position went negative.

Could you elaborate on this? Similar to my analysis here, If the position is negative, but the collateral is not 0, the funding paid is larger than the position's collateral, the funding paid will be decreased to pos. collateral. The funding received will be 0 (if collateral is < funding paid, c1: remaining =0 and deducted =pos.collateral. Then c2: remaining =0 and deducted =0. funding received =0, because remaining =0. So, not sure how funding_received can be > funding paid and I need a more concrete example with numbers.

> It's possible for funding_received to be less than funding_paid, even for positions which did not go insolvent due to funding rates.

Agree that it's possible.

> The outlined invariant is broken which results in loss of funds / broken functionality. It will prevent closing of some positions in the former case, and will result in some funds being stuck in the latter case.

However, the report doesn't showcase that this broken invariant (when funding received < funding paid) will result in any of this. The report only says that it will lead to a dilution of fees, but there is still no explanation of how.

Hence, without a concrete example of how this leads to positions not being closed, thus leading to losses of these positions, and owners, I cannot verify.

I see that the invariant doesn't hold, but this is not sufficient for Medium severity (only broken invariants from README are assigned Medium, not invariants from code comments). Hence, the decision for now is to invalidate this issue and accept the escalation. But, if there's a POC, how does this lead to loss of funds in case funding received < funding paid, or how does funding received > funding paid and how does this lead to a loss of funds, I will consider changing my decision.

WangSecurity

If no answer is provided, planning to accept the escalation and invalidate the issue.

msheikhattari

> I see that the invariant doesn't hold, but this is not sufficient for Medium severity (only broken invariants from README are assigned Medium, not invariants from code comments)

Doesn't the below rule from the severity categorization apply here? That's what I had in mind when creating the issue, since this comment was the only source of documentation on this specification. There is no indication that this is out of date based on any other specs or code elsewhere.

```
Hierarchy of truth: If the protocol team provides no specific information, the
 ↪  default rules apply (judging guidelines).

If the protocol team provides specific information in the README or CODE COMMENTS,
 ↪  that information stands above all judging rules. In case of contradictions
 ↪  between the README and CODE COMMENTS, the README is the chosen source of truth.
```

Nevertheless there are loss of funds from each case, let me first prove that `funding\_received > funding\_paid` is possible:

> Then c2: remaining =0 and deducted =0. funding received =0, because remaining =0.

This is the crux of the issue I am reporting here. That's true in the case of a single position, the issue is that `funding\_received` of that position being set to 0 does not mitigate the underpayment of funding which it made by going negative - they are opposite tokens in the pair.

While that positions `funding\_payment` is capped at the collateral of the specific position, the other side of the pool will continue to accrue funding receipts from this portion of the collateral until it's liquidated:

This is because this collateral is not excluded from `fs.long\_collateral`, it must be explicitly subtracted upon liquidation.

Now the issue causing loss of funds on this side is that positions will fail to close. On the other side, where `funding\_received < funding\_paid`, this is especially problematic in cases where the full funding payment was made, but the collateral fell short of the borrow fee.

In this case, the balance `sum(funding\_received) = sum(funding\_paid)` was not broken by this position, as it made its full funding payment, but it will be excluded from receiving its portion of funding receipts. These tokens will not be directly claimable by any positions in the pool, causing loss of funds in that sense.

Upholding this balance of funding payments to receipts is an important invariant which causes loss of funds and protocol disruptions as outlined above. This is even acknowledged by the team, since this current approach is meant to build up an ”insurance” by penalizing negative positions to pay out future negative positions.

What it fails to acknowledge is that the penalized position has already underpaid its funding fee (in token A), and now the balance is further distorted by eliminating its funding payment (in token B). To move closer to equilibrium between the two sides of funding, a direct approach such as socializing fee underpayments is recommended instead.

There are a few different moving parts so let me know if any component of this line of reasoning is unclear and I could provide specific examples if needed.

WangSecurity

> Doesn't the below rule from the severity categorization apply here? That's what I had in mind when creating the issue, since this comment was the only source of documentation on this specification. There is no indication that this is out of date based on any other specs or code elsewhere

I didn't say the rule applies here, but if there's something said in the code comments and the code doesn't follow it, the issue still has to have at least Medium severity to be valid:

> The protocol team can use the README (and only the README) to define language that indicates the codebase's restrictions and/or expected functionality. Issues that break these statements, irrespective of whether the impact is low/unknown, will be assigned Medium severity. High severity will be applied only if the issue falls into the High severity category in the judging guidelines.

So if the issue breaks an invariant from code comments, but it doesn't have Medium severity, then it's invalid.

> This is the crux of the issue I am reporting here. That's true in the case of a single position, the issue is that funding_received of that position being set to 0 does not mitigate the underpayment of funding which it made by going negative - they are opposite tokens in the pair. While that positions funding_payment is capped at the collateral of the specific position, the other side of the pool will continue to accrue funding receipts from this portion of the collateral until it's liquidated:

That still doesn't prove how `funding\_recevied` can be > `funding\_paid`. The function calc allowbreak _fees which is where this calculation of `funding\_received` and `funding\_paid` happens is called inside value function which is called only inside close and is allowbreak _liquidatable , which is called only inside liquidate.

Hence, this calculation of funding paid and received will be made only when closing or liquidating the position. So, the following is wrong:

> the other side of the pool will continue to accrue funding receipts from this portion of the collateral until it's liquidated

It won't accrue because the position is either already liquidated or closed. Hence, the scenario of `funding\_received > funding\_paid` is still not proven.

About the `funding\_received < funding\_paid`. As I understand, it's intended that the position doesn't receive any funding fees in this scenario which evident by code comment.

So if the position didn't manage to pay the full `funding\_paid` (underpaid the fees), they're intentionally excluded from receiving any funding fees and it's not a loss of funds and these will be received by other users.

Hence, my decision remains: accept the escalation and invalidate the issue. If you still see that I'm wrong somewhere, you're welcome to correct me. But, to make it easier, provide links to the appropriate LOC you refer to.

msheikhattari

> Hence, this calculation of funding paid and received will be made only when closing or liquidating the position. So, the following is wrong:

That's not quite correct. Yes, `calc\_fees` is only called upon closing/liquidating the position. But, this only only calculates the user's pro-rate share of the accrued interest:

The terms `period.received\_long`, `period.received\_short` are relevant here and these are continuously, globally updated upon any interaction by any user with the system. As a result, those positions will count towards the total collateral until explicitly closed, inflating `funding\_received` beyond its true value.

> and it's not a loss of funds and these will be received by other users.

The point of this issue is that user's will not receive those lost funds. Since the global `funding\_received` included the collateral of positions which were later excluded from

receiving their fee, the eventual pro-rata distribution of fees upon closing the position will not be adjusted for this. Thus some portion of fees will remain unclaimed.

The current approach is problematic with loss of funds and disrupted liquidation functionality. There are more direct ways to achieve the correct balance of funding payments to receipts.

WangSecurity

To finally confirm, the problem here is that these funds are just locked in the contract, and no one can get them, correct?

rickkk137

as I got the main point in this report is funding_received can exceed funding_paid but this isn't correct

> It's possible for funding_received by one side of the pool to exceed the funding_paid by the other side, in the case that a position went negative.

WangSecurity

Yeah, there isn't a proof it can happen, the problem we are discussing now is that in cases where funding received is larger than funding paid (can happen when the position is negative), the funding received would be stuck in the contract with no one being able to get them. @rickkk137 would the funding fees be distributed to other users in this case?

rickkk137

In my poc the position finally become negative but funding received is equal to funding paid But funding receiving wouldn't stuck in contracts and will be distributed to later positions

WangSecurity

Thank you for this correction. With this, the escalation will be accepted and the issue will be invalidated. The decision will be applied in a couple of hours.

msheikhattari

Hi @rickkk137 can you specifically clarify the scenario which the PoC is describing? Because from my understanding, it is not showing the case which I described. The nuance is subtle, allow me to explain

Let's say there is one position on the long side, two on the short side, and that longs are paying shorts. Now if one of the short positions goes negative, the other will not receive the total funding fees over the period, it will only get its pro-rata share from the time that the (now liquidated) position was still active.

For comparison, it seems your PoC is showing a single long and short, and you are showing that when the long is liquidated the short should still receive their funding payment. This is a completely different scenario from what is described in this issue.

To answer your earlier question @WangSecurity

To finally confirm, the problem here is that these funds are just locked in the contract, and no one can get them, correct?

Yes, thats correct. That portion of the funding payments is not accessible to any users.

rickkk137

when a position be penalized and funding_received for that position become zero and base or quote collateral's pool will be decrease

```
@external
def close(id: uint256, d: Deltas) -> PoolState:
...
    base\_collateral  : self.MATH.eval(ps.base\_collateral,  d.base\_collateral),
    quote\_collateral : self.MATH.eval(ps.quote\_collateral, d.quote\_collateral),
```

its mean other position get more funding_received compared to past because funding_recieved has reserve relation with base or qoute collateral

```
 paid\_long\_term      : uint256 = self.apply(fs.long\_collateral, fs.funding\_long
  ↪  * new\_terms)
@>> received\_short\_term : uint256 = self.divide(paid\_long\_term,
 ↪  fs.short\_collateral)
```

msheikhattari

The pro rata share of funding received will be correctly adjusted moving forward from liquidation. But the point is that the period.funding_received terms already included the now liquidated collateral, so the other positions do not receive adjusted distributions to account for that.

rickkk137

```
def query(id: uint256, opened\_at: uint256) -> Period:
  """
  Return the total fees due from block `opened\_at` to the current block.
  """
  fees\_i : FeeState = Fees(self).fees\_at\_block(opened\_at, id)
  fees\_j : FeeState = Fees(self).current\_fees(id)
  return Period({
    borrowing\_long  : self.slice(fees\_i.borrowing\_long\_sum,
     ↪  fees\_j.borrowing\_long\_sum),
    borrowing\_short : self.slice(fees\_i.borrowing\_short\_sum,
     ↪  fees\_j.borrowing\_short\_sum),
    funding\_long    : self.slice(fees\_i.funding\_long\_sum,
     ↪  fees\_j.funding\_long\_sum),
    funding\_short   : self.slice(fees\_i.funding\_short\_sum,
     ↪  fees\_j.funding\_short\_sum),
   @>>> received\_long   : self.slice(fees\_i.received\_long\_sum,
    ↪  fees\_j.received\_long\_sum),
```

```
  @>>>  received\_short  : self.slice(fees\_i.received\_short\_sum,
   ↪  fees\_j.received\_short\_sum),
  })
both will be updated when liquidable position will be closed


**msheikhattari**

That's not quite right. From
 ↪  [`current\_fees`](https://github.com/sherlock-audit/2024-08-velar-artha/blob/18
 ↪  ef2d8dc0162aca79bd71710f08a3c18c94a36e/gl-sherlock/contracts/fees.vy\#L137):

```vyper
  paid_short_term     : uint256 = self.apply(fs.short_collateral,
   ↪  fs.funding_short * new_terms)
  received_long_term  : uint256 = self.divide(paid_short_term,
   ↪  fs.long_collateral)
  received_long_sum   : uint256 = self.extend(fs.received_long_sum,
   ↪  received_long_term,  1)
```

So as mentioned in my point above, the collateral at the time of each global fee update is used. When a user later claims his pro-rate share, he will receive his fraction of the total collateral *at the time of the fee update*. Fee updates are performed continuously after each operation, and the total collateral may no longer be representative due to liquidated positions being removed from this sum. However, they were still included in the `received\_{short/long}\_term`, which is added onto the globally stored `received\_{short/long}\_sum`

Thus some share of these global fees are not accessible.

WangSecurity

As I understand it: The fee state is checked twice in the liquidate/close. Let's take close for example:

1. The state is checked during the close in the positions. It doesn't update the state and calculates the funding_received and funding_paid (well, in fact, it calls `close`, then it calls `value`, which calls `calc\_fees`) which gives us 0 funding_received and thus 0 added to the quote_collateral.
2. Then `core::close` updates the Pool and then updates the fees.
3. When we update the fees, we use FeeState.base
   allowbreak _collateral (assuming the scenario with two shorts) when calculating the fees received by shorts for this term. But, the `FeeState.base collateral` is changed only after calculating received
   allowbreak _short
   allowbreak _term/sum

So even though the closed/liquidated position didn't receive any fees and they should go to another short, the `received\_short\_term` accounted as there were two shorts
```

opened, and each received their funding fees.

Hence, when the other short position gets the fees, they will add only a portion from that period, not the full fee.

Thus, I agree it should remain valid. However, medium severity should be kept because, in reality, if this situation occurs, the collateral of that closing/liquidatable position would be quite low (lower than `funding\_paid`), there would be many positions, so the individual loss would be smaller (in terms of the amount, not %), the period with incorrectly applied fees would be small (given the fact that fees are updated at each operation). Hence, the loss is very limited. Planning to reject the escalation and leave the issue as it is, it will be applied at 10 am UTC.

WangSecurity

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- spacegliderrrr: rejected

WangSecurity

#93 and #18 are duplicates of this issue; they both identify that if the negative position is closed, the funding_received for it reset to 0, but these funding fees are not received by any other user and remain stuck in the contract forever.

Issue M-8: First depositor could DoS the pool

Source: https://github.com/sherlock-audit/2024-08-velar-artha-judging/issues/85

The protocol has acknowledged this issue.

Found by

bughuntoor

Summary

First depositor could DoS the pool

Vulnerability Detail

Currently, when adding liquidity to a pool, the way LP tokens are calculated is the following:

1. If LP total supply is 0, mint LP tokens equivalent the mint value
2. If LP total supply is not 0, mint LP tokens equivalent to `mintValue * lpSupply / poolValue`

However, this opens up a problem where the first user can deposit a dust amount in the pool which has a value of just 1 wei and if the price before the next user deposits, the pool value will round down to 0. Then any subsequent attempts to add liquidity will fail, due to division by 0.

Unless the price goes back up, the pool will be DoS'd.

Impact

DoS

Code Snippet

https://github.com/sherlock-audit/2024-08-velar-artha/blob/main/gl-sherlock/contracts/pools.vy#L178

Tool used

Manual Review

Recommendation

Add a minimum liquidity requirement.

Discussion

msheikhattari

Escalate

> DoS has two separate scores on which it can become an issue:
> 1. The issue causes locking of funds for users for more than a week (overrided to 4 hr)
> 2. The issue impacts the availability of time-sensitive functions (cutoff functions are not considered time-sensitive). If at least one of these are describing the case, the issue can be a Medium. If both apply, the issue can be considered of High severity. Additional constraints related to the issue may decrease its severity accordingly. Griefing for gas (frontrunning a transaction to fail, even if can be done perpetually) is considered a DoS of a single block, hence only if the function is clearly time-sensitive, it can be a Medium severity issue.

Low at best. Per the severity guidelines, this is not DoS since no user funds are locked and no sensitive functionality is impacted (ongoing positions/LPs are not impacted). Additionally, this both assumes that no other LPs make any deposits within the same block as the attacker (as the price would be equivalent), and that the price is monotonically decreasing after the attack was initiated. Not only is it low impact, but also low likelihood.

sherlock-admin3

Escalate

DoS has two separate scores on which it can become an issue:

1. The issue causes locking of funds for users for more than a week (overrided to 4 hr)
2. The issue impacts the availability of time-sensitive functions (cutoff functions are not considered time-sensitive). If at least one of these are describing the case, the issue can be a Medium. If both apply, the issue can be considered of High severity. Additional constraints related to the issue may decrease its severity accordingly. Griefing for gas (frontrunning a transaction to fail, even if can be done perpetually) is considered a DoS of a single block, hence only if the function is clearly time-sensitive, it can be a Medium severity issue.

Low at best. Per the severity guidelines, this is not DoS since no user funds are locked and no sensitive functionality is impacted (ongoing positions/LPs are not impacted). Additionally, this both assumes that no other LPs make any deposits within the same block as the attacker (as the price would be equivalent), and that the price is monotonically decreasing after the attack was initiated. Not only is it low impact, but also low likelihood.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

WangSecurity

While indeed this doesn't qualify the DOS requirements, this issue can still result in the functionality of the protocol being blocked and the users wouldn't be able to use this pool. I agree it's low likelihood, but it's still possible.

@spacegliderrrr since this issue relies on the precision loss, it requires a POC, can you make one? And also how low should be the price so it leads to rounding down?

spacegliderrrr

Price doesn't really matter - it's just needed to deposit little enough tokens that they're worth 1 wei of quote token. So if for example the pair is WETH/ USDC, a user would need to deposit ~4e8 wei WETH (considering price of $2,500).

As for the PoC, because the code is written in Vyper and Foundry does not support it, I cannot provide a PoC.

WangSecurity

In that case, could you make a very detailed attack path, I see you provided an example of the price that would cause an issue, but still would like a very detailed attack path.

WangSecurity

One important thing to consider is the following question:

> We will report issues where the core protocol functionality is inaccessible for at least 7 days. Would you like to override this value? Yes, 4 hours

So, I see that DOS rules say that the funds should be locked for a week. But, the question is about core protocol functionality being inaccessible. The protocol specified they want to have issues about core protocol functionality being inaccessible for 4 hours.

This finding, indeed doesn't lead to locking funds or blocking time-sensitive functions, but it can lead to the core protocol functionality being inaccessible for 4 hours. I see that it's low likelihood, but the likelihood is not considered when defining the severity. Hence, even though for this finding there have to be no other depositors in the next block or the price being lower for the next 4 hours, this can happen. Thus, medium severity for this issue is appropriate. Planning to reject the escalation and leave the issue as it is.

msheikhattari

This issue doesn't impact ongoing operations, so its similar to frontrunning of initializers. No irreversible damage or loss of funds occur.

Core protocol functionality being inaccessible should have some ramifications like lock of funds or broken time-sensitive functionality (like withdrawals). No funds are in the pool when this issue is taking place.

msheikhattari

In any case this issue does need a PoC - per the severity criteria all issues related to precision loss require one.

WangSecurity

@spacegliderrrr could you make a detailed numerical POC, showcasing the precision loss and DOS.

> Core protocol functionality being inaccessible should have some ramifications like lock of funds or broken time-sensitive functionality (like withdrawals). No funds are in the pool when this issue is taking place

As I've said previously, this still impacts the core protocol functionality, as the users cannot deposit into the pool, and this very well could last for more than 4 hours. Hence, this is sufficient for medium severity as the core protocol functionality is inaccessible for more than 4 hours.

spacegliderrrr

1. Market pair used is WETH/USDT. Current WETH price $2,500.
2. User is the first depositor in the pool, depositing 4e8 WETH. Based on the lines of code below, the user is minted 1 lp token.: `amt0 = 4e8 * 2500e18 / 1e18 = 10000e8 = 1e12 lowered = 1e12 / 1e12 = 1`
3. Next block comes and WETH price drops to $2,499.

4. A user now attempts to deposit. Since LP tokens minted are respective to the current pool value, contract calculates pool value, using the same formula above `amt0 = 4e8 * 2499e18 / 1e18 = 0.9996e12` `lowered = 0.9996e12 / 1e12 = 0`
5. User's LP tokens are calculated using the following formula. As the pool's value is 0, a division by 0 is attempted, which forces the tx to revert.
6. Pool is DoS'd until ETH price goes above $2,500 again.

WangSecurity

As far as I can tell, the POC is correct and this issue will indeed happen. As was said previously, planning to reject the escalation and leave the issue as it is.

WangSecurity

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- msheikhattari: rejected

Issue M-9: Whale LP providers can open positions on both sides to force users into high fees.

Source: https://github.com/sherlock-audit/2024-08-velar-artha-judging/issues/89

The protocol has acknowledged this issue.

Found by

bughuntoor

Summary

Whale LP providers can open positions on both sides to force users into high fees.

Vulnerability Detail

LP fees within the protocol are based on utilization percentage of the total funds in the pool. The problem is that this could easily be abused by LP providers in the following way.

Consider a pool where the majority of the liquidity is provided by a single user.

1. Users have opened positions at a relatively low utilization ratio
2. The whale LP provider opens same size positions in both directions at 1 leverage.
3. This increases everyone's fees. Given that the LP provider holds majority of the liquidity, most of the new fees will go towards them, making them a profit.

As long as the whale can maintain majority of the liquidity provided, attack remains profitable. If at any point they can no longer afford maintaining majority, they can simply close their positions without taking a loss, so this is basically risk-free.

Impact

Loss of funds

Code Snippet

https://github.com/sherlock-audit/2024-08-velar-artha/blob/main/gl-sherlock/contracts/params.vy#L33

Tool used

Manual Review

Recommendation

Consider a different way to calculate fees

Discussion

msheikhattari

Escalate Invalid. Quoting a valid point from your own comment:

> Issue should be low/info. Ultimately, all LPs would want is fees and this would give them the highest fees possible. Furthermore, the attack is extremely costly, as it would require user to lock up hundreds of thousands/ millions, losing a significant % of them. Any user would have an incentive to add liquidity at extremely high APY, which would allow for both new positions opens and LP withdraws.

This attack inflates borrow fees, but the high APY will attract other LP depositors which would drive the utilization back down to normal levels, reducing the fee. Unlike the issue that you were escalating, this one has no such time sensitivity - the market would naturally tend towards rebalance within the next several days / weeks. It's not reasonable to assume that the existing positions would remain open despite high fees and other LPs would not enter the market over the coming days/weeks.

Not only that, the other assumptions of this issue are incorrect:

> If at any point they can no longer afford maintaining majority, they can simply close their positions without taking a loss, so this is basically risk-free.

Wrong. Each opened long AND short position must pay a fixed fee, so the whale is taking a risk. He is betting that the current positions will not close, and his stake will not get diluted, just long enough to eke out a net profit. And this is assuming he had a majority stake to begin with, which for the more liquid pools where the attack is most profitable due to a large amount of open interest, is a highly questionable assumption.

The game theory makes it unlikely that the whale would be able to extract enough extra fees to even make a profit net of the operating fees of such an attack.

sherlock-admin3

Escalate Invalid. Quoting a valid point from your own comment:

> Issue should be low/info. Ultimately, all LPs would want is fees and this would give them the highest fees possible. Furthermore, the attack is extremely costly, as it would require user to lock up hundreds of thousands/ millions, losing a significant % of them. Any user would have an incentive to add liquidity at extremely high APY, which would allow for both new positions opens and LP withdraws.

This attack inflates borrow fees, but the high APY will attract other LP depositors which would drive the utilization back down to normal levels, reducing the fee. Unlike the issue that you were escalating, this one has no such time sensitivity - the market would naturally tend towards rebalance within the next several days / weeks. It's not reasonable to assume that the existing positions would remain open despite high fees and other LPs would not enter the market over the coming days/weeks.
Not only that, the other assumptions of this issue are incorrect:

> If at any point they can no longer afford maintaining majority, they can simply close their positions without taking a loss, so this is basically risk-free.

Wrong. Each opened long AND short position must pay a fixed fee, so the whale is taking a risk. He is betting that the current positions will not close, and his stake will not get diluted, just long enough to eke out a net profit. And this is assuming he had a majority stake to begin with, which for the more liquid pools where the attack is most profitable due to a large amount of open interest, is a highly questionable assumption. The game theory makes it unlikely that the whale would be able to extract enough extra fees to even make a profit net of the operating fees of such an attack.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

WangSecurity

@spacegliderrrr do you have any counterarguments?

spacegliderrrr

@WangSecurity Issue above showcases a real issue which could occur if a whale decides to _attack_ a pool.

> Each opened long AND short position must pay a fixed fee, so the whale is taking a risk. He is betting that the current positions will not close, and his stake will not get diluted, just long enough to eke out a net profit.

True, there's some risk, though most of it can be mitigated. For example if the attack is performed when most opened positions are at negative PnL (which would mean closing them is profitable to the LP providers), most of the risk is mitigated as users have 2 choices - close early at a loss or keep the position open at high fees (either way, profitable for the LP provider).

> the market would naturally tend towards rebalance within the next several days / weeks.

True, though as mentioned, it would take days/ weeks in which the whale could profit.

The issue does involve some game theory, but nonetheless shows an actual risk to honest users.

WangSecurity

I also agree there are lots of risks, with this scenario. But, it's still possible to pose losses on other users in a way of arbitrary increasing fees. The market would rebalance, but it can even take less than a day to cause losses to users. Hence, I agree that this issue should remain medium severity, because even though the issue has high constraints, still can cause losses. Planning to reject the escalation.

WangSecurity

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- msheikhattari: rejected

Issue M-10: User could have impossible to close position if funding fees grow too big.

Source: https://github.com/sherlock-audit/2024-08-velar-artha-judging/issues/96

Found by

bughuntoor

Summary

User could have impossible to close position if funding fees grow too big.

Vulnerability Detail

In order to prevent positions from becoming impossible to be closed due to funding fees surpassing collateral amount, there's the following code which pays out funding fees on a first-come first-serve basis.

However, this wrongly assumes that the order of action would always be for the side which pays funding fees to close their position before the side which claims the funding fee.

Consider the following scenario:

1. There's an open long (for total collateral of `X`) and an open short position. Long position pays funding fee to the short position.
2. Eventually the funding fee grows larger than the whole long position (`X + Y`). it is due liquidation, but due to bot failure is not yet liquidated (which based on comments is expected and possible behaviour)
3. A new user opens a new long position, once with `X` collateral. (total quote collateral is currently 2X)
4. The original long is closed. This does not have an impact on the total quote collateral, as it is increased by the `funding_paid` which in our case will be counted as exactly as much as the collateral (as in these calculations it cannot surpass it). And it then subtracts that same quote collateral.
5. The original short is closed. `funding_received` is calculated as `X + Y` and therefore that's the amount the total quote collateral is reduced by. The new total quote collateral is `2X - (X + Y) = X - Y`.
6. Later when the user attempts to close their position it will fail as it will attempt subtracting `(X - Y) - X` which will underflow.

Marking this as High, as a user could abuse it to create a max leverage position which cannot be closed. Once it is done, because the position cannot be closed it will keep on accruing funding fees which are not actually backed up by collateral, allowing them to double down on the attack.

Impact

Loss of funds

Code Snippet

https://github.com/sherlock-audit/2024-08-velar-artha/blob/main/gl-sherlock/contracts/positions.vy#L250 https://github.com/sherlock-audit/2024-08-velar-artha/blob/main/gl-sherlock/contracts/positions.vy#L263 https://github.com/sherlock-audit/2024-08-velar-artha/blob/main/gl-sherlock/contracts/positions.vy#L211

Tool used

Manual Review

Recommendation

Fix is non-trivial.

Discussion

KupiaSecAdmin

Escalate

The underflow does not happen by nature of `deduct` function. Thus this is invalid.

sherlock-admin3

> Escalate
> The underflow does not happen by nature of `deduct` function. Thus this is invalid.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

spacegliderrrr

Escalate

Severity should be High. Issue above describes how a user could open risk-free max leverage positions, basically making a guaranteed profit from the LPs.

Regarding @KupiaSecAdmin escalation above - please do double check the issue above. The underflow does not happen in `deduct` but rather in the `MATH.eval` operations. The problem lies within the fact that if order of withdraws is reversed, funding receiver can receive more fees than the total collateral (as long as it is available by other users who have said collateral not yet eaten up by funding fees). Then, some of the funding paying positions will be impossible to be closed.

sherlock-admin3

> Escalate
> Severity should be High. Issue above describes how a user could open risk-free max leverage positions, basically making a guaranteed profit from the LPs.
> Regarding @KupiaSecAdmin escalation above - please do double check the issue above. The underflow does not happen in `deduct` but rather in the `MATH.eval` operations. The problem lies within the fact that if order of withdraws is reversed, funding receiver can receive more fees than the total collateral (as long as it is available by other users who have said collateral not yet eaten up by funding fees). Then, some of the funding paying positions will be impossible to be closed.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

ami0x226

This is invalid.

> 4. The original long is closed. This does not have an impact on the total quote collateral, as it is increased by the funding_paid which in our case will be counted as exactly as much as the collateral (as in these calculations it cannot surpass it). And it then subtracts that same quote collateral.

When original long is closed, total quote collateral is changed.

```
File: gl-sherlock\contracts\positions.vy
209:     quote\_reserves  : [self.MATH.PLUS(pos.collateral), \#does not need min()
210:                         self.MATH.MINUS(fees.funding\_paid)],
211:     quote\_collateral: [self.MATH.PLUS(fees.funding\_paid),
212:                         self.MATH.MINUS(pos.collateral)],
```

Heres, `pos.collateral = X, fees.funding\_paid = X + Y`. Then, `quote\_collateral <- quote\_collateral + X + Y - X = quote\_collateral + Y = 2X + Y`, and `quote\_reserves <- quote\_reserves  + X - X - Y = quote\_reserves - Y`.

When original short is closed in step5, new total quote collateral is `2X + Y - (X + Y) = X`

and there is no underflow in step6. As a result, the scenario of the report is wrong. The loss causes in `quote\_reserves`, but, in practice, Y is enough small by the frequent liquidation and it should be assumed that the liquidation is done correctly. Especially, because the report does not mention about this vulnerability, I think this is invalid

ami0x226

Also, Funding paid cannot exceed collateral of a position from the `apply` function.

spacegliderrrr

> Heres, pos.collateral = X, fees.funding_paid = X + Y.

Here's where you're wrong. When the user closes their position, `funding\_paid` cannot exceed `pos.collateral`. So `fees.funding\_paid == pos.collateral` when closing the original long. Please re-read the issue and code again.

ami0x226

> Heres, pos.collateral = X, fees.funding_paid = X + Y.
>
> Here's where you're wrong. When the user closes their position, `funding\_paid` cannot exceed `pos.collateral`. So `fees.funding\_paid == pos.collateral` when closing the original long. Please re-read the issue and code again.

That's true. I mentioned about it in the above comment

> Also, Funding paid cannot exceed collateral of a position from the apply function.

I just use `fees.funding\_paid = X + Y` to follow the step2 of `bughuntoor`'s scenario:

> 2. Eventually the funding fee grows larger than the whole long position (X + Y). it is due liquidation, but due to bot failure is not yet liquidated (which based on comments is expected and possible behaviour)

rickkk137

invalid funding_paid cannot exceed than collateral also funding_received cannot be greater funding_paid

WangSecurity

@spacegliderrrr can you make a coded POC showcasing the attack path from the report?

WangSecurity

We've got the POC from the sponsor:

```
from ape import chain
import pytest
from conftest import d, ctx
```

```
\# https://github.com/sherlock-audit/2024-08-velar-artha-judging/issues/96

\# 1) There's an open long (for total collateral of X) and an open short position.
↪  Long position pays funding fee to the short position.
\# 2) Eventually the funding fee grows larger than the whole long position (X + Y).
↪  it is due liquidation, but due to bot failure is not yet liquidated (which
↪  based on comments is expected and possible behaviour)
\# 3) A new user opens a new long position, once with X collateral. (total quote
↪  collateral is currently 2X)
\# 4) The original long is closed. This does not have an impact on the total quote
↪  collateral, as it is increased by the funding\_paid which in our case will be
↪  counted as exactly as much as the collateral (as in these calculations it
↪  cannot surpass it). And it then subtracts that same quote collateral.
\# 5) The original short is closed. funding\_received is calculated as X + Y and
↪  therefore that's the amount the total quote collateral is reduced by. The new
↪  total quote collateral is 2X - (X + Y) = X - Y.
\# 6) Later when the user attempts to close their position it will fail as it will
↪  attempt subtracting (X - Y) - X which will underflow.

\# OR
\# 1. Consider the original long and short positions, long pays funding fee to
↪  short.
\# 2. Time goes by, liquidator bots fails and funding fee makes 100\% of the long
↪  collateral (consider collateral is X)
\# 3. Another long position is opened again with collateral X.
\# 4. Time goes by, equivalent to funding fee of 10\%. Total collateral at this
↪  moment is still 2X, so the new total funding paid is 1.2X
\# 5. Short position closes and receives funding paid of 1.2X. Quote collateral is
↪  now reduced from 2X to 0.8X.
\# 6. (Optional) Original long closes position. For them funding paid is capped at
↪  their collateral, so funding paid == collateral, so closing does not make a
↪  difference on the quote collateral.
\# 7. The next long position holder tries to close. They're unable because their
↪  collateral is 1x, funding paid is 0.1x. Collateral calculation is 0.8X + 0.1X -
↪  1X and underflow reverts

PARAMS = {
  'MIN\_FEE'               : 1\_00000000,
  'MAX\_FEE'               : 1\_00000000, \# 10\%/block
  'PROTOCOL\_FEE'          : 1000,
  'LIQUIDATION\_FEE'       : 2,
  'MIN\_LONG\_COLLATERAL'   : 1,
  'MAX\_LONG\_COLLATERAL'   : 1\_000\_000\_000,
  'MIN\_SHORT\_COLLATERAL'  : 1,
  'MAX\_SHORT\_COLLATERAL'  : 1\_000\_000\_000,
  'MIN\_LONG\_LEVERAGE'     : 1,
  'MAX\_LONG\_LEVERAGE'     : 10,
  'MIN\_SHORT\_LEVERAGE'    : 1,
  'MAX\_SHORT\_LEVERAGE'    : 10,
  'LIQUIDATION\_THRESHOLD' : 1,
```

```
}

PRICE = 400\_000

def test\_issue(core, api, pools, positions, fees, math, oracle, params,
            VEL, STX, LP,
            mint, burn, open, close,
            long, short, lp\_provider, long2, owner,
            mint\_token):

    \# setup
    core.fresh("VEL-STX", VEL, STX, LP, sender=owner)
    mint\_token(VEL, d(100\_000), lp\_provider)
    mint\_token(STX, d(100\_000), lp\_provider)
    mint\_token(VEL, d(10\_000) , long)
    mint\_token(STX, d(10\_000) , long)
    mint\_token(VEL, d(10\_000) , long2)
    mint\_token(STX, d(10\_000) , long2)
    mint\_token(VEL, d(10\_000) , short)
    mint\_token(STX, d(10\_000) , short)
    VEL.approve(core.address, d(100\_000), sender=lp\_provider)
    STX.approve(core.address, d(100\_000), sender=lp\_provider)
    VEL.approve(core.address, d(10\_000) , sender=long)
    STX.approve(core.address, d(10\_000) , sender=long)
    VEL.approve(core.address, d(10\_000) , sender=long2)
    STX.approve(core.address, d(10\_000) , sender=long2)
    VEL.approve(core.address, d(10\_000) , sender=short)
    STX.approve(core.address, d(10\_000) , sender=short)
    mint(VEL, STX, LP, d(10\_000), d(4\_000), price=PRICE, sender=lp\_provider)
    params.set\_params(PARAMS, sender=owner)         \# set 10 \% fee / block

    START\_BLOCK = chain.blocks[-1].number
    print(f"Start block: {START\_BLOCK}")

    \# 1) There's an open long (for total collateral of X) and an open short
    ↪  position. Long position pays funding fee to the short position.
    \# open pays funding when long utilization > short utilization
    ↪  (interest/reserves)
    X  = d(100)
    p1 = open(VEL, STX, True  , X   , 10, price=PRICE, sender=long)
    p2 = open(VEL, STX, False , d(5),  2, price=PRICE, sender=short)
    assert not p1.failed, "open long"
    assert not p2.failed, "open short"

    fees = params.dynamic\_fees(pools.lookup(1))
    print(f"Pool fees: {fees}")

    \# 2. Time goes by, liquidator bots fails and funding fee makes 100\% of
    \# the long collateral (consider collateral is X)
    chain.mine(10)
```

```
\# fees/value after
value = positions.value(1, ctx(PRICE))
print(value['fees'])
print(value['pnl'])
assert value['fees']['funding\_paid']         == 99900000
assert value['fees']['funding\_paid\_want']    == 99900000
assert value['fees']['borrowing\_paid']       == 0
\# assert value['fees']['borrowing\_paid\_want']  == 99900000
assert value['pnl']['remaining']             == 0

value = positions.value(2, ctx(PRICE))
print(value['fees'])
print(value['pnl'])
assert value['fees']['funding\_paid']            == 0
assert value['fees']['funding\_received']        == 99900000
assert value['fees']['funding\_received\_want']   == 99900000
\# assert value['fees']['borrowing\_paid']          == 4995000
assert value['pnl']['remaining']                == 4995000

\# 3. Another long position is opened again with collateral X.
p3 = open(VEL, STX, True, X, 10, price=PRICE, sender=long2)
assert not p3.failed

\# 4. Time goes by, equivalent to funding fee of 10\%.
\# Total collateral at this moment is still 2X, so the new total funding paid
↪  is 1.2X
chain.mine(1)

print(f"Pool: {pools.lookup(1)}")

value = positions.value(1, ctx(PRICE))
print(f"Long 1: {value['fees']}")
print(f"Long 1: {value['pnl']}")

value = positions.value(3, ctx(PRICE))
print(f"Long 2: {value['fees']}")
print(f"Long 2: {value['pnl']}")

assert value['fees']['funding\_paid'] == 9990000     \#TODO: value with mine(2)
↪  is high?
assert value['pnl']['remaining']     == 89910000

print(f"Blocks: {chain.blocks[-1].number - START\_BLOCK}")

\# 5. Short position closes and receives funding paid of 1.2X. Quote collateral
↪  is now reduced from 2X to 0.8X.
tx = close(VEL, STX, 2, price=PRICE, sender=short)
print(core.Close.from\_receipt(tx)[0]['value'])
\# fees: [0, 0, 139860000, 139860000, 0, 0, 4995000]
```

```
    assert not tx.failed

    print(f"Blocks: {chain.blocks[-1].number - START\_BLOCK}")

    pool = pools.lookup(1)
    print(f"Pool: {pool}")
    \# assert pool['quote\_collateral'] == 9990000 * 0.8  \#59940000
    value = positions.value(3, ctx(PRICE))
    print(f"Long 2: {value['fees']}")
    print(f"Long 2: {value['pnl']}")
    print(f"Long 2: {value['deltas']}")

    \# 7. The next long position holder tries to close. They're unable because
    ↪  their collateral is 1x, funding paid is 0.1x.
    \# Collateral calculation is 0.8X + 0.1X - 1X and underflow reverts
    tx = close(VEL, STX, 3, price=PRICE, sender=long2)
    print(core.Close.from\_receipt(tx)[0]['value'])
    assert not tx.failed, "close 2nd long"

    \# 6. (Optional) Original long closes position. For them funding paid is capped
    ↪  at their collateral,
    \# so funding paid == collateral, so closing does not make a difference on the
    ↪  quote collateral.
    tx = close(VEL, STX, 1, price=PRICE, sender=long)
    print(core.Close.from\_receipt(tx)[0]['value'])
    assert not tx.failed, "close original"

    print(f"Blocks: {chain.blocks[-1].number - START\_BLOCK}")

    pool = pools.lookup(1)
    print(f"Pool: {pool}")


    assert False
```

Hence, the issue is indeed valid. About the severity, as I understand, it's indeed high, since there are no extensive limitations, IIUC. Anyone is free to correct me and the POC, but from my perspective it's indeed correct.

But for now, planning to reject @KupiaSecAdmin escalation, accept @spacegliderrrr escalation and upgrade severity to high.

KupiaSecAdmin

@WangSecurity - No problem with having this issue as valid but the severity should be Medium at most I think. Because, 1) Usually in real-world, funding fees don't exceed the collateral. 2) When positions get into risk, liquidation bots will work most of times.

WangSecurity

As far as I understand, this issue can be triggered intentionally, i.e. the first constraint

can be reached intentionally, as explained at the end of the Vulnerability Detail section:

> as a user could abuse it to create a max leverage position which cannot be closed

But you're correct that it depends on the liquidation bot malfunctioning, which is also mentioned in the report:

> Eventually the funding fee grows larger than the whole long position (X + Y). it is due liquidation, but due to bot failure is not yet liquidated (which based on comments is expected and possible behaviour)

I agree that this is indeed an external limitation. Planning to reject both escalations and leave the issue as it is.

WangSecurity

Result: Medium Has duplicates

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- KupiaSecAdmin: rejected
- spacegliderrrr: rejected

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Velar-co/gl-sherlock/pull/2

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.