SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for: Jala Swap
Prepared by: Sherlock
Lead Security Expert: bughuntoor
Dates Audited: February 29 - March 5, 2024
Prepared on: March 29, 2024

Introduction

The first DEX on Chiliz to swap, wrap, provide liquidity, and stake fan tokens.

Scope

Repository: jalaswap/jalaswap-dex-contract

Branch: main

Commit: 1629e1110572d961942b8f7167fe94a7f714a2e3

For the detailed scope, see the contest details.

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

Medium	High
3	0

Issues not fixed or acknowledged

Medium	High
0	0

Issue M-1: The functions about `permit` won't work and always revert

Source: https://github.com/sherlock-audit/2024-02-jala-swap-judging/issues/40

Found by

Aizen, AuditorPraise, PawelK, Spearmint, b0g0, bughuntoor, deth, giraffe, jasonxiale, jokr, recursiveEth, santiellena, zhuying

Summary

The functions about `permit` won't work and always revert

Vulnerability Detail

`JalaRouter02.sol` has functions (`removeLiquidityWithPermit/removeLiquidityETHWithPermit/removeLiquidityETHWithPermitSuppo` about `permit`. These functions will call `permit` function in `JalaPair.sol`. `JalaPair` is inherited from `JalaERC20`. Although `JalaERC20` is out of scope. But both `JalaPair` and `JalaERC20` have no `permit` functions. So when you call `removeLiquidityWithPermit/removeLiquidityETHWithPermit/removeLiquidityETHWithPermitSupportingFeeOnTransferToken` it will always revert.

POC

Add this test function in `JalaRouter02.t.sol`.

```
function test_Permit() public {
        tokenA.approve(address(router), 1 ether);
        tokenB.approve(address(router), 1 ether);

        router.addLiquidity(
            address(tokenA), address(tokenB), 1 ether, 1 ether, 1 ether, 1
↪   ether, address(this), block.timestamp
        );

        address pairAddress = factory.getPair(address(tokenA), address(tokenB));
        JalaPair pair = JalaPair(pairAddress);
        uint256 liquidity = pair.balanceOf(address(this));

        liquidity = (liquidity * 3) / 10;
        pair.approve(address(router), liquidity);
```

```
        vm.expectRevert();
        router.removeLiquidityWithPermit(
            address(tokenA),
            address(tokenB),
            liquidity,
            0.3 ether - 300,
            0.3 ether - 300,
            address(this),
            block.timestamp,
            true,
            1, // this value is for demonstration only
            bytes32(uint256(1)), // this value is for demonstration only
            bytes32(uint256(1)) // this value is for demonstration only
        );
    }
```

Impact

We can't remove liquidity by using `permit`.

Code Snippet

https://github.com/sherlock-audit/2024-02-jala-swap/blob/main/jalaswap-dex-contract/contracts/JalaRouter02.sol#L150-L167 https://github.com/sherlock-audit/2024-02-jala-swap/blob/main/jalaswap-dex-contract/contracts/JalaRouter02.sol#L169-L185 https://github.com/sherlock-audit/2024-02-jala-swap/blob/main/jalaswap-dex-contract/contracts/JalaRouter02.sol#L202-L225

Tool used

manual review and foundry

Recommendation

Implement `permit` function in `JalaERC20`. Reference: https://github.com/Uniswap/v2-core/blob/master/contracts/UniswapV2ERC20.sol.

Discussion

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/jalaswap/jalaswap-dex-contract/commit/c73dda6e81268bb329ec80ef851707f3b95ff0df.

spacegliderrrr

fix looks good, permit function is now added

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-2: User wrapped tokens get stuck in master router because of incorrect calculation

Source: https://github.com/sherlock-audit/2024-02-jala-swap-judging/issues/146

Found by

Arabadzhiev, C1rdan, cawfree, den_sosnovskyi, jah, mahmud, merlinboii, recursiveEth, yotov721

Summary

Swapping exact tokens for ETH swaps underlying token amount, not wrapped token amount and this causes wrapped tokens to get stuck in the contract.

Vulnerability Detail

In the protocol the `JalaMasterRouter` is used to swap tokens with less than 18 decimals. It is achieved by wrapping the underlying tokens and interacting with the `JalaRouter02`. **Wrapping** the token gives it decimals 18 (18 - token.decimals()). There are also functions that swap with native ETH.

In the `swapExactTokensForETH` function the tokens are transferred from the user to the Jala master router, **wrapped**, approved to `JalaRouter2` and then `IJalaRouter02::swapExactTokensForETH()` is called with **the amount of tokens to swap**, to address, deadline and path.

The amount of tokens to swap that is passed, is the amount before the wrap. Hence the wrappedAmount - underlyingAmount is stuck.

Add the following test to `JalaMasterRouter.t.sol` and run with `forge test --mt testswapExactTokensForETHStuckTokens -vvv`

```
function testswapExactTokensForETHStuckTokens() public {
    address wrappedTokenA =
↪   IChilizWrapperFactory(wrapperFactory).wrappedTokenFor(address(tokenA));

    tokenA.approve(address(wrapperFactory), type(uint256).max);
    wrapperFactory.wrap(address(this), address(tokenA), 100);

    IERC20(wrappedTokenA).approve(address(router), 100 ether);
    router.addLiquidityETH{value: 100 ether}(wrappedTokenA, 100 ether, 0, 0,
↪   address(this), type(uint40).max);

    address pairAddress = factory.getPair(address(WETH),
↪   wrapperFactory.wrappedTokenFor(address(tokenA)));
```

```
    uint256 pairBalance = JalaPair(pairAddress).balanceOf(address(this));

    address[] memory path = new address[](2);
    path[0] = wrappedTokenA;
    path[1] = address(WETH);

    vm.startPrank(user0);
    console.log("ETH user balance before:       ", user0.balance);
    console.log("TokenA user balance before:    ", tokenA.balanceOf(user0));
    console.log("WTokenA router balance before: ",
↪   IERC20(wrappedTokenA).balanceOf(address(masterRouter)));

    tokenA.approve(address(masterRouter), 550);
    masterRouter.swapExactTokensForETH(address(tokenA), 550, 0, path, user0,
↪   type(uint40).max);
    vm.stopPrank();

    console.log("ETH user balance after:       ", user0.balance);
    console.log("TokenA user balance after:    ", tokenA.balanceOf(user0));
    console.log("WTokenA router balance after: ",
↪   IERC20(wrappedTokenA).balanceOf(address(masterRouter)));
}
```

Impact

User wrapped tokens get stuck in router contract. The can be stolen by someone performing a `swapExactTokensForTokens()` because it uses the whole balance of the contract when swapping: `IERC20(wrappedTokenIn).balanceOf(address(this))`

```
amounts = IJalaRouter02(router).swapExactTokensForTokens(
    IERC20(wrappedTokenIn).balanceOf(address(this)),
    amountOutMin,
    path,
    address(this),
    deadline
);
```

Code Snippet

https://github.com/sherlock-audit/2024-02-jala-swap/blob/main/jalaswap-dex-contract/contracts/JalaMasterRouter.sol#L284-L301

Tool used

Manual Review, foundry

Recommendation

In `JalaMasterRouter::swapExactTokensForETH()` multiply the `amountIn` by decimal off set of the token:

```
    function swapExactTokensForETH(
        address originTokenAddress,
        uint256 amountIn,
        uint256 amountOutMin,
        address[] calldata path,
        address to,
        uint256 deadline
    ) external virtual override returns (uint256[] memory amounts) {
        address wrappedTokenIn =
↪   IChilizWrapperFactory(wrapperFactory).wrappedTokenFor(originTokenAddress);

        require(path[0] == wrappedTokenIn, "MS: !path");

        TransferHelper.safeTransferFrom(originTokenAddress, msg.sender,
↪   address(this), amountIn);
        _approveAndWrap(originTokenAddress, amountIn);
        IERC20(wrappedTokenIn).approve(router,
↪   IERC20(wrappedTokenIn).balanceOf(address(this)));

+         uint256 decimalOffset =
↪   IChilizWrappedERC20(wrappedTokenIn).getDecimalsOffset();
+         amounts = IJalaRouter02(router).swapExactTokensForETH(amountIn *
↪   decimalOffset, amountOutMin, path, to, deadline);
-         amounts = IJalaRouter02(router).swapExactTokensForETH(amountIn ,
↪   amountOutMin, path, to, deadline);
    }
```

Discussion

nevillehuang

If a sufficient slippage (which is users responsibility) is set, this will at most cause a revert, so medium severity is more appropriate. (The PoC set slippage to zero)

sherlock-admin4

The protocol team fixed this issue in PR/commit https://github.com/jalaswap/jalaswap-dex-contract/commit/9ed6e8f4f6ad762ef7b747ca2d367f6cfd78973e.

spacegliderrrr

fix looks good, right amount is now passed to the router

sherlock-admin4

The Lead Senior Watson signed off on the fix.

Issue M-3: JalaPair potential permanent DoS due to overflow

Source: https://github.com/sherlock-audit/2024-02-jala-swap-judging/issues/186

The protocol has acknowledged this issue.

Found by

0k, 0xMojito, 0xRstStn, 0xloscar01, Stoicov, ZanyBonzy, den_sosnovskyi, deth, fibonacci, giraffe, mahmud, n1punp, santiellena, sunill_eth, tank

Summary

In the `JalaPair::_update` function, overflow is intentionally desired in the calculations for `timeElapsed` and `priceCumulative`. This is forked from the UniswapV2 source code, and it's meant and known to overflow. UniswapV2 was developed using Solidity 0.6.6, where arithmetic operations overflow and underflow by default. However, Jala utilizes Solidity >=0.8.0, where such operations will automatically revert.

Vulnerability Detail

```
uint32 timeElapsed = blockTimestamp - blockTimestampLast; // overflow is desired
if (timeElapsed > 0 && _reserve0 != 0 && _reserve1 != 0) {
    // * never overflows, and + overflow is desired
    price0CumulativeLast +=
↪   uint256(UQ112x112.encode(_reserve1).uqdiv(_reserve0)) * timeElapsed;
    price1CumulativeLast +=
↪   uint256(UQ112x112.encode(_reserve0).uqdiv(_reserve1)) * timeElapsed;
}
```

Impact

This issue could potentially lead to permanent denial of service for a pool. All the core functionalities such as `mint`, `burn`, or `swap` would be broken. Consequently, all funds would be locked within the contract.

I think issue with High impact and a Low probability (merely due to the extended timeframe for the event's occurrence, it's important to note that this event will occur with 100% probability if the protocol exists at that time), should be considered at least as Medium.

References

There are cases where the same issue is considered High.

https://solodit.xyz/issues/h-02-uniswapv2priceoraclesol-currentcumulativeprices-will-revert-when-pricecumulative-addition-overflow-code4rena-phuture-finance-phuture-finance-contest-git https://solodit.xyz/issues/m-02-twavsol_gettwav-will-revert-when-timestamp-4294967296-code4rena-nibbl-nibbl-contest-git https://solodit.xyz/issues/trst-m-3-basev1pair-could-break-because-of-overflow-trust-security-none-satinexchange-markdown_

Code Snippet

https://github.com/sherlock-audit/2024-02-jala-swap/blob/main/jalaswap-dex-contract/contracts/JalaPair.sol#L97-L102

Tool used

Manual Review

Recommendation

Use the `unchecked` block to ensure everything overflows as expected

Discussion

nevillehuang

Request poc

Would like the watson/watsons to present a scenario where a reasonable overflow can be achieved, because based on my discussion with LSW this is likely not reasonable considering frequency of liquidity addition and ratio of reserves required

> #186 - let's put it this way. Ratio is scaled up by 2^112. Having a `(ratio * timeElapsed1) + (ratio * timeElapsed2)` is the same as having `(ratio * (timeElapsed1 + timeElapsed2)`. So if we have a token1 max reserve of uint112 and token0 reserve is 1, `uint112.max * uint112.max * totalTimeElapsed`. In order for this to overflow, we need totalTimeElapsed to be > uint32.max, which is approx 132 years. So for this to overflow, we'd need to have the pool running for 132 years with one of the reserve being the max and the other one being just 1 wei for the entirety of the 132 years.

sherlock-admin3

PoC requested from @0xf1b0

Requests remaining: **10**

0xf1b0

Even if we do not take reserves into account, the timestamp is converted into a 32-bit value.

```
uint32 blockTimestamp = uint32(block.timestamp % 2 ** 32);
```

When the timestamp gets >4294967296 (in 82 years), the `_update` function will always revert.

nevillehuang

@Czar102 what do you think? I don't think its severe enough to warrant medium severity. I remember you rejected one of a similar finding in dodo as seen here

Czar102

The linked issue presented a way to DoS functionality. Is this also the case here? Or are funds locked here as well? It seems to me that the funds are locked, so I'd accept this as a valid Medium (High impact in far future – a limitation).

nevillehuang

@Czar102 Agree, if long enough, all core functionalities `burn()` (Remove liquidity), `mint()` (add liquidity) and `swap()` that depends on this low level functions have the potential to be DoSed due to `_update()` reverting, will leave as medium severity

deadrosesxyz

Escalate

`block.timestamp` stored in a uint32 will overflow in year 2102. This is way too far in the future and likely even beyond our lifetime. Even in the unlikely scenario where the protocol will be used in 80 years from now, users will know years in advance that they'll have to withdraw their funds as the protocol is about to shut down. Issue should be considered low/info

sherlock-admin2

> Escalate
>
> `block.timestamp` stored in a uint32 will overflow in year 2102. This is way too far in the future and likely even beyond our lifetime. Even in the unlikely scenario where the protocol will be used in 80 years from now, users will know years in advance that they'll have to withdraw their funds as the protocol is about to shut down. Issue should be considered low/info

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.

nevillehuang

I will invite any watsons to prove if accumulated actions (swap, burn and mint) can cause overflow in price variables (whether maliciously or accidentally)

giraffe0x

Timestamp overflow is less of a concern here, cumulative value is.

1) As with uniswap, pools can be created with all sorts of exotic pairs. So ratio between reserve0/1 may be arbitrarily large.
2) Code comments clearly state that "overflow is desired" when overflow cannot happen in current format.

It's a clear mistake on the devs, which has an easy fix to wrap with unchecked block. Not sure why they won't fix it.

deadrosesxyz

@giraffe0x hey, please check the initial message by @nevillehuang on why cumulatives cannot actually overflow. Reserves are uint112s, so ratio can be max uint112.max : 1. Also, the comment is not made by the devs, but actually left out from the copying of original univ2 contracts

nevillehuang

@deadrosesxyz, Could a malicious user(s) (despite requiring alot of funds) be able to brick the pool permanently by constantly performing actions (burn, swap, mint) and incrementing price variables?

deadrosesxyz

@nevillehuang No, constantly performing actions will not help for an overflow to occur in any way (as the cumulative increases based on seconds passed). Even in the most extreme scenario where one of the reserves has max value and the other one has simply 1 wei, it would take 132 years for an overflow to occur. Quoting my comments from earlier:

> let's put it this way. Ratio is scaled up by 2^112. Having a (ratio * timeElapsed1) + (ratio * timeElapsed2) is the same as having (ratio * (timeElapsed1 + timeElapsed2). So if we have a token1 max reserve of uint112 and token0 reserve is 1, uint112.max * uint112.max * totalTimeElapsed. In order for this to overflow, we need totalTimeElapsed to be > uint32.max, which is approx 132 years. So for this to overflow, we'd need to have the pool running for 132 years with one of the reserve being the max and the other one being just 1 wei for the entirety of the 132 years.

nevillehuang

This would depend on how @Czar102 interprets the following rule given he agreed initially that this should remain as a valid medium severity as seen here

> Causes a loss of funds but requires certain external conditions or specific states, or a loss is highly constrained. The losses must exceed small, finite amount of funds, and any amount relevant based on the precision or significance of the loss.

mahmudoloyede

> @giraffe0x hey, please check the initial message by @nevillehuang on why cumulatives cannot actually overflow. Reserves are uint112s, so ratio can be max uint112.max : 1. Also, the comment is not made by the devs, but actually left out from the copying of original univ2 contracts

It really doesn't matter if it was left out from the copying of original univ2 contracts. The intent is for it to overflow but it won't in this case. Also if the pairs are tokens with decimals greater than 18, the DOS will occur sooner.

santiellena

> @nevillehuang No, constantly performing actions will not help for an overflow to occur in any way (as the cumulative increases based on seconds passed). Even in the most extreme scenario where one of the reserves has max value and the other one has simply 1 wei, it would take 132 years for an overflow to occur. Quoting my comments from earlier:
>
> > let's put it this way. Ratio is scaled up by 2^112. Having a (ratio * timeElapsed1) + (ratio * timeElapsed2) is the same as having (ratio * (timeElapsed1 + timeElapsed2). So if we have a token1 max reserve of uint112 and token0 reserve is 1, uint112.max * uint112.max * totalTimeElapsed. In order for this to overflow, we need totalTimeElapsed to be > uint32.max, which is approx 132 years. So for this to overflow, we'd need to have the pool running for 132 years with one of the reserve being the max and the other one being just 1 wei for the entirety of the 132 years.

@deadrosesxyz It will actually take less than 132 years because the pair could have a token1 max reserve of uint112 and token0 reserve of 1 wei on the first block. This means that timeElapsed will be the block.timestamp of the first block (something like 1710940084 now). This means that 52.25 years of those 132 years have already passed. It is still a big number, however, it is easy to fix and a clear mistake on the devs.

Czar102

I maintain my judgment from preliminary judging – I think it should be Medium

especially because of the "overflow is desired" comment, which means that sponsors care about the behavior in far future. Without these comments, the judgment on this issue would be disputable.

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

Czar102

Result: Medium Has duplicates

sherlock-admin3

Escalations have been resolved successfully!

Escalation status:

- deadrosesxyz: rejected

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.