PALADIN
BLOCKCHAIN SECURITY

Smart Contract Security Assessment

Final Report

For Wallchain (MetaSwapWrapper)

04 May 2023

paladinsec.co

info@paladinsec.co

Table of Contents

Table of Contents 2

Disclaimer 3

1 Overview 4

1.1 Summary 4

1.2 Contracts Assessed 4

1.3 Findings Summary 5

1.3.1 MetaSwapWrapper 6

2 Findings 7

2.1 MetaSwapWrapper 7

2.1.1 Privileged Functions 8

2.1.2 Issues & Recommendations 9

Disclaimer

Paladin Blockchain Security ("Paladin") has conducted an independent audit to verify the integrity of and highlight any vulnerabilities or errors, intentional or unintentional, that may be present in the codes that were provided for the scope of this audit. This audit report does not constitute agreement, acceptance or advocation for the Project that was audited, and users relying on this audit report should not consider this as having any merit for financial advice in any shape, form or nature. The contracts audited do not account for any economic developments that may be pursued by the Project in question, and that the veracity of the findings thus presented in this report relate solely to the proficiency, competence, aptitude and discretion of our independent auditors, who make no guarantees nor assurance that the contracts are completely free of exploits, bugs, vulnerabilities or deprecation of technologies. Further, this audit report shall not be disclosed nor transmitted to any persons or parties on any objective, goal or justification without due written assent, acquiescence or approval by Paladin.

All information provided in this report does not constitute financial or investment advice, nor should it be used to signal that any persons reading this report should invest their funds without sufficient individual due diligence regardless of the findings presented in this report. Information is provided 'as is', and Paladin is under no covenant to the completeness, accuracy or solidity of the contracts audited. In no event will Paladin or its partners, employees, agents or parties related to the provision of this audit report be liable to any parties for, or lack thereof, decisions and/or actions with regards to the information provided in this audit report.

Cryptocurrencies and any technologies by extension directly or indirectly related to cryptocurrencies are highly volatile and speculative by nature. All reasonable due diligence and safeguards may yet be insufficient, and users should exercise considerable caution when participating in any shape or form in this nascent industry.

The audit report has made all reasonable attempts to provide clear and articulate recommendations to the Project team with respect to the rectification, amendment and/or revision of any highlighted issues, vulnerabilities or exploits within the contracts provided. It is the sole responsibility of the Project team to sufficiently test and perform checks, ensuring that the contracts are functioning as intended, specifically that the functions therein contained within said contracts have the desired intended effects, functionalities and outcomes of the Project team.

Paladin retains full rights over all intellectual property (including expertise and new attack or exploit vectors) discovered during the audit process. Paladin is therefore allowed and expected to re-use this knowledge in subsequent audits and to inform existing projects that may have similar vulnerabilities. Paladin may, at its discretion, claim bug bounties from third-parties while doing so.

1 Overview

This report has been prepared for Wallchain's Paraswap Augustus wrapper contracts on the Polygon network. Paladin provides a user-centred examination of the smart contracts to look for vulnerabilities, logic errors or other issues from both an internal and external perspective.

1.1 Summary

Project Name	Wallchain (Paraswap Augustus wrapper)
URL	https://www.wallchain.xyz/
Platform	Polygon
Language	Solidity
Preliminary Contracts	https://github.com/Wallchain-Inc/metaswapwrapper/blob/528de3ce131175695462c4d828aabd3ea7e26d3b/contracts/MetaSwapWrapper.sol
Resolution 1	https://github.com/Wallchain-Inc/metaswapwrapper/blob/e1750ca6aacd96652fd0439a9419b733f64daeed/contracts/MetaSwapWrapper.sol
Resolution 2	https://github.com/Wallchain-Inc/metaswapwrapper/blob/ab41d1562ebae4155a45011c668c1e83655376cd/contracts/MetaSwapWrapper.sol
Resolution 3	https://github.com/Wallchain-Inc/metaswapwrapper/blob/8606a34d6cb5b8a08a68cbe83e490d2b7bc22908/contracts/MetaSwapWrapper.sol

1.2 Contracts Assessed

Name	Contract	Live Code Match
`MetaSwapWrapper`	`0x4DeF20E7DbfeAB6b2cFEB53bdcA28a6DCEd12317`	MATCH

1.3 Findings Summary

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	1	1	-	-
Medium	2	2	-	-
Low	4	4	-	-
Informational	2	2	-	-
Total	**9**	**9**	**-**	**-**

Classification of Issues

Severity	Description
High	Exploits, vulnerabilities or errors that will certainly or probabilistically lead towards loss of funds, control, or impairment of the contract and its functions. Issues under this classification are recommended to be fixed with utmost urgency.
Medium	Bugs or issues that may be subject to exploit, though their impact is somewhat limited. Issues under this classification are recommended to be fixed as soon as possible.
Low	Effects are minimal in isolation and do not pose a significant danger to the project or its users. Issues under this classification are recommended to be fixed nonetheless.
Informational	Consistency, syntax or style best practices. Generally pose a negligible level of risk, if any.

1.3.1 MetaSwapWrapper

ID	Severity	Summary	Status
01	HIGH	Contract structure allows for an exploiter to drain all open approvals to the `MetaSwapWrapper`	✓ RESOLVED
02	MEDIUM	ETH to ETH wrapped transactions will fully fail	✓ RESOLVED
03	MEDIUM	Users will lose value through many router interactions including `transferTokensForExactTokens` due to input dust being stuck in the wrapper	✓ RESOLVED
04	LOW	`withdrawETH` might fail due to the use of transfer instead of call	✓ RESOLVED
05	LOW	Lack of validation	✓ RESOLVED
06	LOW	Contract does not work for non-compliant ERC20 tokens like USDT on Ethereum	✓ RESOLVED
07	LOW	Contract might not work for non-EOA senders	✓ RESOLVED
08	INFO	Gas optimizations	✓ RESOLVED
09	INFO	Typographical errors	✓ RESOLVED

2 Findings

2.1 MetaSwapWrapper

`MetaSwapWrapper` is a generic contract developed by Wallchain which allows users to collect some of the value themselves which would typically have to be given to MEV.

Users can install a browser extension which checks if any backrun opportunity exists whenever they plan to do a transaction. When they do plan to execute the transaction, instead of sending the transaction to the swap router for example, it is sent to the `MetaSwapWrapper`. Wallchain will calculate an arbitrage payload which is then also sent to the wrapper. The wrapper then first does the swap and then executes the arbitrage (backrun) payload, collecting the traditional MEV value within the transaction itself.

At the end, part of that value can be given back to the user.

The contract also contains various utility functions for the Wallchain governance to take out ERC20 tokens and ETH out of the contract. It also contains a function which allows for Wallchain to upgrade the backrunning algorithm.

The contract is pausable and users should not assume that any value stuck within this contract is safe as it can be taken out by users and by the Wallchain team.

2.1.1 Privileged Functions

- `pause`
- `unpause`
- `addTarget`
- `removeTarget`
- `withdrawEth`
- `withdrawAll`
- `upgradeMaster`
- `transferOwnership`
- `renounceOwnership`

2.1.2 Issues & Recommendations

Issue #01	**Contract structure allows for an exploiter to drain all open approvals to the MetaSwapWrapper**
Severity	HIGH SEVERITY
Location	Lines 221-223 `(bool success, bytes memory data) = execution.callTarget.call{ value: _isETH(execution.srcToken) ? msg.value : 0 }(execution.targetData);`
Description	Users need to approve the `MetaSwapWrapper` with any tokens they want to swap, then they set the `callTarget` to the swap router and the tokens are first transferred into the wrapper and then to the router. However, a malicious exploiter will be able to freely set their `callTarget` to the approved ERC20 token and frontrun the swap by calling `transferFrom(user, exploiter, userBalance)` as the `targetData`. This specifically means that any and all user approvals will likely eventually be drained by an exploiter in a catastrophic exploit with a critical expected impact.
Recommendation	The de-facto best way to mitigate this issue is by addressing it at the source: **NO VALUE SHOULD EVER BE TIED TO THE METASWAPWRAPPER** Since `MetaSwapWrapper` executes generic logic, we STRONGLY believe it is in the best interest of Wallchain to just assume that exploiters can cause that contract to execute bad logic. Wallchain should therefore not assume that token balances within the wrapper are safe and that any approvals to the wrapper are safe. Instead, what can be done is to use an `ApprovalHelper` contract. Users approve the `ApprovalHelper` which has a fully isolated context and the `ApprovalHelper` is called from the wrapper instead. Then, `ApprovalHelper` is explicitly not permitted to be set as the `callTarget`.

Apart from implementing this, we also recommend implementing a few more improvements as extra safeguards:

1. Use separate whitelists for tokens and `callTargets`. This is very important to reduce the attack surface as sharing a single whitelist is what makes these exploits so trivial.
2. Refund any `srcTokens` that have been added to the contract over the duration of the swap.
3. Require that the balances of the `srcToken` and `dstToken` have increased (not necessary when assuming that they can be taken out anyway).
4. Reset allowances of the `srcToken` if it was not fully consumed (not necessary when assuming that they can be taken out anyway)
5. Prevent functions like `transfer`, `transferFrom`, `approve`, `increaseAllowance`… from being called as the data (not necessary when assuming that they can be taken out anyway)

We do believe that recommendations 3-5 are kind of silly to implement since they really do not address the core issue and it is akin to trying to figure out all the different ways an exploiter might take advantage of this setup, while there may be more ways it can be done.

Instead, we strongly advise to simply not attach any value to this contract. When this is done correctly, the whitelist can even be fully removed and the wrapper can be made more generic.

Resolution

RESOLVED

The critical recommendation has been implemented, and an approval proxy is now used (a separate contract which holds the approvals and is blacklisted within the metawrapper). A few other non-critical safeguards have been added: refunds of any `src` and `dest` token increments over the swap and blacklisting of the `transferFrom` signature. Though not every non-critical recommendation was introduced, this is fine as no value should be within this contract.

We would like to remember the reader that any value that ends up in this contract can be trivially stolen. This is now by design and does not require further resolution. Users should not send tokens to this contract nor approve it directly.

Issue #02	ETH to ETH wrapped transactions will fully fail
Severity	MEDIUM SEVERITY
Location	Line 210 `uint256 balanceBefore = _tokenBalance(`
Description	ETH to ETH transactions cannot be correctly wrapped as the refund logic of the ETH balance increase is incorrect in this scenario. This is because the `balanceBefore` in this scenario also erroneously includes `msg.value`. These wraps will therefore completely malfunction. This issue has been rated as medium instead of high since we find it rather unlikely that people would do a swap from ETH to ETH.
Recommendation	Consider subtracting `msg.value` from the `balanceBefore`. This is correct in combination with the `msg.value` validation which was recommended.
Resolution	RESOLVED

Issue #03	**Users will lose value through many router interactions including `transferTokensForExactTokens` due to input dust being stuck in the wrapper**
Severity	MEDIUM SEVERITY
Description	Many router interactions do not pull all tokens from the users and only pull what is necessary at the execution time — most notably, swaps where the user expects an exact output amount. For these swaps, the wrapper will likely pull in the maximum amount (based on slippage) from the user, while the actual router might take a smaller amount from the wrapper. This results in the difference between these two tokens to be fully stuck in the wrapper, and we expect that an exploiter will steal these tokens shortly after (e.g. through an exploit where they set the token as the target). Though we do not think such exploits are trivially preventable, it is trivial to prevent the source and destination tokens from ever being stuck in the contract by just blindly sending the full balances of these tokens back to the user. If the full source and destination balances are always sent to the user, the client essentially acknowledges that tokens are not safe at all within the contract. This is acceptable given that it is hard to prevent exploits that drain the contract balances. The only case where this might be a risk is if some router interaction sends back two tokens: e.g. `removeLiquidity`, in which case, the second token would be stolen by an exploiter instead of simply being stuck. However, preventing the second token from being stolen is rather difficult. It is therefore much more simple to make sure that `removeLiquidity` is called with the user address as the destination instead of the wrapper, and to not use a destination token in such cases.
Recommendation	Consider always sending back the full source and destination balance of the contract at the end of any transaction. Consider adding off-chain validation logic that no tertiary tokens end up becoming stuck in the wrapper by accident (through log inspection or similar). Consider simply setting the recipient of all swaps and router interactions as the user instead of the wrapper to more fully enforce the fact that the wrapper never has value.

Resolution	RESOLVED A refund now occurs of any increase in the `src` balance.

Issue #04	**`withdrawETH` might fail due to the use of `transfer` instead of `call`**
Severity	LOW SEVERITY
Location	Line 88 `to.transfer(address(this).balance)`
Description	Using `transfer` to transfer ETH value is generally discouraged as it uses a limited gas allocation. There is no guarantee that this function will not break due to future hardforks as gas fees might evolve over time. This was recently proven to be the case within zkSync which uses different gas fee accounting and where `transfer` did not work for a lot of contracts, resulting in over a million USD being locked. Additionally, if the recipient has a fallback function, `transfer` might fail as well. This is most notably the case in certain multi-signature wallets, timelocks and proxies.
Recommendation	Consider using `msg.sender.call{address(this).balance}("");` instead.
Resolution	RESOLVED The recommendation has been implemented.

Issue #05	Lack of validation
Severity	LOW SEVERITY

Description	The contract contains functions with parameters which are not properly validated. Having unvalidated parameters could allow the governance or users to provide variable values which are unexpected and incorrect. This could cause side-effects or worse exploits in other parts of the codebase.
	
	Consider validating the following function parameters:
	
	Line 56
	`whitelistedTargets[_whitelistedTargets[i]] = true;`
	
	Line 70
	`function addTarget(address _target) external onlyOwner {`
	
	Consider validating that the target is a contract (using OpenZeppelin's `Address.isContract` library). This prevents the issue where the owner accidentally whitelists an EOA and users accidentally send ETH to that EOA.
	
	Line 105
	`require(nextAddress != address(0), "nextAddress can't be address(0)");`
	
	An `isContract` check would be superior here as well, instead of only testing for the arbitrary zero address.
	
	Line 196
	`function swapWithWallchain(WallchainExecutionParams calldata execution)`
	
	Consider adding the following requirement:
	`if (_isETH(execution.srcToken)) {`
	`  require(msg.value != 0, "...");`
	`} else {`
	`  require(msg.value == 0, "...");`
	`}`

Recommendation	Consider validating the function parameters mentioned above.
Resolution	RESOLVED

Issue #06	Contract does not work for non-compliant ERC20 tokens like USDT on Ethereum
Severity	LOW SEVERITY
Location	Line 134 `token.approve(target, amount);`
Description	Certain older tokens on the Ethereum blockchain are not perfectly compliant with the ERC20 standard. The most notable example is USDT on the Ethereum blockchain. Not only does it not return a success boolean whenever an operation is done, it also reverts if someone tries to approve from a non-zero allowance to another non-zero allowance.

Recommendation	Consider adjusting this section of code to the following:
	`if (token.allowance(address(this), target) < amount) {`
	`  token.forceApprove(target, amount);`
	`}`
	
	Note that `forceApprove` is a function present within OpenZeppelin's latest SafeERC20 library versions.

Resolution	RESOLVED `forceApprove` is now used.

Issue #07	Contract might not work for non-EOA senders
Severity	LOW SEVERITY
Location	Line 145 `(bool result, ) = destination.call{value: amount, gas: 10000}(`
Description	The contract needlessly limits the gas consumption on the above call — this prevents the contract from potentially working with more advanced callers with callback logic. Since the function implicitly does not assume that any reentrancy may not occur at this point, we do not see the point of explicitly limiting gas here.
Recommendation	Consider removing the gas limit if it is not explicitly needed. Consider carefully evaluating why it was present in the first place and testing that any initial concerns are in fact not present. Alternatively, it might make sense to not hardcode this and instead have it as a parameter.
Resolution	RESOLVED The gas parameter has been removed.

Issue #08	Gas optimizations
Severity	INFORMATIONAL

Description	Line 222
	`value: _isETH(execution.srcToken) ? msg.value : 0`
	
	Once the recommendation has been implemented to validate `msg.value`, this can just blindly forward `msg.value` instead to make the code more readable as well.
	
	Line 224-225
	`emit CallResponse(success, data);`
	`require(success, "Call Target failed");`
	
	These lines should be inverted to safe gas in the failing case.

Recommendation	Consider implementing the gas optimizations mentioned above.
Resolution	RESOLVED

Issue #09	Typographical errors
Severity	INFORMATIONAL

Description	Line 13
	`import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";`
	
	The contract uses both `safeTransfer` from OpenZeppelin and TransferHelper from Uniswap. Try to stick with only 1. Our recommendation is OpenZeppelin's.
	
	Lines 21-22
	`address private constant ETH_ADDRESS = address(0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE);`
	
	It is not necessary to cast `0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE` as `address` again. The address cast can be removed.
	
	Lines 24-33
	`event MasterUpgrade(address old, address newMaster);`
	`[...]`
	`event MasterUpgradeFailed(address attemptMaster);`
	
	Ensure that the important addresses from the events that have been declared are indexed. Specifically `MasterUpgrade`, `TargetAdded`, `TargetRemoved`, `MasterUpgradeFailed` should be indexed.
	
	Line 40
	`address[] originator; // Transaction originator.`
	
	`address[] originator` are not actually the originators. It is an address of beneficiaries. Therefore, the comment description should be changed to say "transaction beneficiaries".

Line 48

```
mapping(address => bool) public whitelistedTargets;
```

For better usage and integration with the front-end, we encourage in this case, to use and enumerable set. You can use the one from OpenZeppelin: `import "@openzeppelin/contracts/utils/structs/EnumerableSet.sol";`

Note that `view` functions need to be added to get the length and items at indices.

Lines 62-68

```
function pause() public onlyOwner {
    _pause();
}

function unpause() public onlyOwner {
    _unpause();
}
```

Make sure to always try to make important functions as restricted as they can. In this case, they should be marked as `external` instead of `public` to ensure it is not called inside the contract.

Line 83

```
return (address(token) == ETH_ADDRESS);
```

There is no need to wrap the return in brackets; they can be deleted.

Line 87

```
address payable to = payable(msg.sender);
```

There is no need to cast `msg.sender` as payable.

Line 92

```
function withdrawAll(address[] calldata tokens) external
onlyOwner {
```

Instead of using `address[]`, use an array of type `IERC20[]` to avoid casting later.

Line 106

```
if (address(wchainMaster) != nextAddress) {
```

Use `require` instead of an `if` statement.

Line 120

```
if (token == ETH_ADDRESS) {
```

Line 144

```
if (token == ETH_ADDRESS){
```

Use the getter function `_isETH` on line 82 to perform the input validation.

Line 148

```
require(result, "Failed to transfer Ether");
```

This error message should not include Ether if the contracts will be deployed to other chains. Delete it or change it to say "native token" or "gas token". A nicer error message could be "Transfer failed".

Line 236

```
payable(msg.sender)
```

`msg.sender` is already payable; the wrap is unnecessary.

Finally, we would like to remind the client that this contract might malfunction for targets which try to call some callback function on the wrapper. In these cases, the wrapper would need to explicitly return the right data on this callback, which can be difficult to predict. An example would be if an ERC1155 token was sent to the wrapper. Supporting this might not be the best idea because this token would then be stuck in the wrapper. It might make more sense to revert instead.

Recommendation Consider fixing the typographical errors.

Resolution RESOLVED

Most of the issues have been resolved.


PALADIN
BLOCKCHAIN SECURITY