



PALADIN
BLOCKCHAIN SECURITY

Smart Contract Security Assessment

Final Report

For Wallchain (Meta Intents)

13 September 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	5
1.3 Findings Summary	6
1.3.1 Global Issues	7
1.3.2 MetaSwapWrapperV2	7
1.3.3 SearcherBase	7
1.3.4 SearcherExecutionCapsule	8
1.3.5 SearcherRequestCall	8
2 Findings	9
2.1 Global Issues	9
2.1.2 Issues & Recommendations	10
2.2 MetaSwapWrapperV2	13
2.2.1 Privileged Functions	13
2.2.2 Issues & Recommendations	14
2.3 SearcherBase	16
2.3.1 Privileged Functions	16
2.3.2 Issues & Recommendations	17
2.4 SearcherExecutionCapsule	18
2.4.1 Privileged Functions	18
2.4.2 Issues & Recommendations	19
2.5 SearcherRequestCall	25
2.5.1 Privileged Functions	25
2.5.2 Issues & Recommendations	26

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 advocacy 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 the right to re-use any and all knowledge and expertise gained during the audit process, including, but not limited to, vulnerabilities, bugs, or new attack vectors. Paladin is therefore allowed and expected to use this knowledge in subsequent audits and to inform any third party, who may or may not be our past or current clients, whose projects have similar vulnerabilities. Paladin is furthermore allowed to claim bug bounties from third-parties while doing so.

1 Overview

This report has been prepared for Wallchain Meta Intents contracts on the Ethereum 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
URL	https://www.wallchain.xyz/
Platform	Ethereum
Language	Solidity
Preliminary Contracts	https://github.com/Wallchain-xyz/meta-intents/tree/da2314ff9001430b0136e65bf2da569f000c0635
Resolution 1	https://github.com/Wallchain-xyz/meta-intents/tree/5fe024afdf1689fe5eadce19f609c2e209248ab2/src
Resolution 2	https://github.com/Wallchain-xyz/meta-intents/tree/824ed096803132cccfd3aabf4690717a5a94d567/src

1.2 Contracts Assessed

Name	Contract	Live Code Match
MetaSwapWrapperV2	0xC0ffeE00c3F5A11369EeB57693C56Fd939dc6DBb	✓ MATCH
SearcherBase	Dependency	✓ MATCH
SearcherExecutionCapsule	Dependency	✓ MATCH
SearcherRequestCall	Dependency	✓ MATCH

1.3 Findings Summary

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
● High	3	2	1	-
● Medium	3	2	-	1
● Low	0	-	-	-
● Informational	10	10	-	-
Total	16	14	1	1

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 with 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 Global Issues

ID	Severity	Summary	Status
01	MEDIUM	Bid mechanism presently incentivizes excessive risk-taking backrun strategies leading to a sub-optimal auction outcome	ACKNOWLEDGED
02	INFO	The contract is inconsistent with the usage of requirements and if-error reversions	✓ RESOLVED

1.3.2 MetaSwapWrapperV2

ID	Severity	Summary	Status
03	INFO	Gas optimizations	✓ RESOLVED
04	INFO	Typographical issues	✓ RESOLVED

1.3.3 SearcherBase

ID	Severity	Summary	Status
05	INFO	Contract does not implement Ownable	✓ RESOLVED
06	INFO	Typographical issue	✓ RESOLVED

1.3.4 SearcherExecutionCapsule

ID	Severity	Summary	Status
07	HIGH	Nonce logic is broken due to incorrect truncation, causing many nonces to point to the same storage slot	✓ RESOLVED
08	HIGH	Searchers can be abused by executing their backruns on different users	PARTIAL
09	HIGH	Gas refund logic is fundamentally flawed	✓ RESOLVED
10	MEDIUM	Various calls are not gas-limited	✓ RESOLVED
11	INFO	Gas optimizations	✓ RESOLVED
12	INFO	Typographical issues	✓ RESOLVED

1.3.5 SearcherRequestCall

ID	Severity	Summary	Status
13	MEDIUM	Low-level call is vulnerable to return-bomb griefing, which causes the searcher call to use significantly more gas than the desired provided gas	✓ RESOLVED
14	INFO	Ownable and SafeERC20 are unused	✓ RESOLVED
15	INFO	Gas optimization	✓ RESOLVED
16	INFO	Typographical issues	✓ RESOLVED

2 Findings

2.1 Global Issues

The issues listed in this section apply to the protocol as a whole. Please read through them carefully and take care to apply the fixes across the relevant contracts.



2.1.2 Issues & Recommendations

Issue #01

Bid mechanism presently incentivizes excessive risk-taking backrun strategies leading to a sub-optimal auction outcome

Severity



Description

The bid of the highest searcher is allowed to execute the backrun for a user. However, as it turns out within the codebase, specifically within the capsule and call contract, no bid is due if the backrun strategy reverts. This means that if, for any reason, the strategy which is supposed to generate more profit than bid fails, the searcher loses nothing except some gas.

More formally speaking: Given that there are many backrun strategies possible, some more likely to succeed in execution than others, the current incentives will cause that the strategy with the highest potential payoff, however unlikely that strategy is, will always be the one that ends up winning the auction.

Example:

Alice creates a swap, causing an imbalance on pool A. Statistically speaking, when arbitrated against pool B, there is a 1% chance of \$100 profit. When arbitrated against pool C, there is a 100% chance of \$10 profit.

Searcher 1 wants to execute an arbitrage against both pools (whichever works) and bids \$10.5 (keeping \$0.5 in expected profit)

However, Searcher 2 knows that they can simply revert if pool B is not rebalanceable and so bids \$12, more than the expected profit of any backrun strategy.

The optimal backrun strategy is to always rebalance C and rebalance B whenever possible, which would result in an expected profit of \$11 on average. However, the strategy which actually wins is the one by searcher 2 who rebalances B when possible and reverts otherwise. The expected profit of this winning strategy is \$1 (with ~\$0.12 going to the user on average, though with competition ~\$1 would go to the user).

This means that the strategy which will win the auction time and time again is 90% less efficient than the optimal strategy, and the user will collect, on average, significantly less than in the optimal scenario.

It should finally be noted that many states still allow for the original transaction to execute except for the ensure modifier within the capsule, which causes the whole transaction to revert instead. This appears inconsistent.

Recommendation Consider incentivizing searchers to bid based on their expected return and not permit them to bid on their highest possible return.

This could be done by still requiring them to pay their bid even if their strategy fails. It should be noted that in such a scenario, it might be possible for the user to strategically execute the transaction in a way where the strategy always fails, causing them to still receive the bid.

An example of such a counter-exploit would be that the user rebalances the main swap pool in the other direction before executing their transaction with backrun payload. This would cause the actual swap transaction to re-balance it right to the optimal price and any backruns to always fail to generate value.

In the scenario where backruns always need to pay their bid, this user was effectively able to prevent any value from leaking, while still being allowed to collect the bid.

It might therefore be inadequate to simply always collect the bid regardless of execution.

The more optimal resolution might therefore be to simply revert the whole transaction if the backrun fails, and to pick the second best backrun or request a new set of backruns (excluding the previous searcher).

Resolution

 ACKNOWLEDGED

Issue #02

The contract is inconsistent with the usage of requirements and if-error reversions

Severity

 INFORMATIONAL

Description

It has recently become common practice to use abi encoded errors within Solidity. However, many protocols prefer to stick to the old `require` method of reverting. Within Paladin, we are fine with our clients doing either. However, WallChain appears to use both methods inconsistently: using `require` in some locations while using `if-revert` in others.

Recommendation

Consider sticking to a single methodology for creating reverts.

Resolution

 RESOLVED

The client has taken steps to use `require` consistently.



2.2 MetaSwapWrapperV2

MetaSwapWrapperV2 is a generic contract developed by Wallchain which allows users to collect some of the value themselves which would typically be given to MEV.

The way it works is that users can install a browser extension which checks if any backrun opportunity exists (through a searcher auction) whenever they plan to do a transaction. When they plan to proceed with this transaction, the transaction is sent to the MetaSwapWrapper instead of to the swap router for example. 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, through the price paid by the highest bidder in the auction. At the end, most of that bid value can be given back to the user.

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

2.2.1 Privileged Functions

- `setSearcherExecutionCapsule`
- `pause`
- `unpause`
- `addTarget`
- `removeTarget`
- `transferOwnership`
- `renounceOwnership`

2.2.2 Issues & Recommendations

Issue #03	Gas optimizations
Severity	 INFORMATIONAL
Description	<u>Lines 98 and 109</u> <pre>require(!_whitelistedTargets.contains(_target), "Target is already added"); require(_whitelistedTargets.contains(_target), "Target is not present");</pre> The add and remove functions on the EnumerableSet return a success boolean which can be tested in a require statement to avoid fetching the status twice. This also reduces the line-count of the contract. <u>Line 310</u> <pre>returnAmount = srcBalance - srcBalanceBefore;</pre> Given that this scenario is somewhat unlikely, the calculation should probably be done within the if-statement. <u>Lines 230-234</u> <pre>require(_whitelistedTargets.contains(execution.approveTarget) && _whitelistedTargets.contains(execution.callTarget), "Target must be whitelisted");</pre> These statements waste gas in the common case where these two variables are identical.
Recommendation	Consider implementing the gas optimizations mentioned above.
Resolution	 RESOLVED

Issue #04	Typographical issues
Severity	INFORMATIONAL
Description	<u>Line 27</u> <pre data-bbox="448 331 1286 360">address(0xEeeeeEeeeEeEeeEeEeEEEEeeeeEEEEEEEE);</pre> The address cast is unnecessary here. <u>Lines 31, 32 and 36</u> <pre data-bbox="448 573 916 692">event WithdrawAll(); event WithdrawEth(); event NonProfitTransaction();</pre> <u>Line 259</u> <pre data-bbox="448 831 1091 860">selector := mload(add(exchangeData, 32))</pre> This whole selector logic should probably only execute if the calldata is at least 4 bytes long; for shorter calldata, this logic might have side-effects. --- The constructor looks a bit inconsistent: the whitelisted targets add logic does not have an isContract check while the setter does. --- It may make sense to unset all the approvals to fully eliminate them for the next swap if they are not fully consumed. The benefit of this is limited however, as the case where such approval abuse occurs is rather unlikely as the abuser would need to be able to execute code within the user's call.
Recommendation	Consider fixing the typographical issues.
Resolution	RESOLVED
	Most of the errors have been fixed.

2.3 SearcherBase

SearcherBase is a recommended dependency for searcher contracts to extend upon. It defines the `prePayGas` function which allows for the `SearcherExecutionCapsule` to pull in ETH from the searcher to cover the gas cost of the backrun. It should be noted that such a pull is only permitted given that the searcher's owner signs an approval to do so.

The contract finally also exposes a template `onlyTrustedRequestedCall` function which can be used to guard external backrun functions. There is no function in the template using this modifier by default, as it's up to the implementing contracts to define such functions.

2.3.1 Privileged Functions

- `prePayGas [ SearcherExecutionCapsule ]`



2.3.2 Issues & Recommendations

Issue #05	Contract does not implement Ownable
Severity	 INFORMATIONAL
Description	SearcherExecutionCapsule requires that the owner of every searcher signs permission to use the searcher for a backrun. However, for such a check to be possible, the searcher needs to expose an owner() function. This is not done by default within this dependency.
Recommendation	Consider whether it makes sense to extend Ownable for this purpose. We do understand that for gas reasons, certain searchers might want to use an immutable owner(), in which case extending Ownable by default may not make as much sense.
Resolution	 RESOLVED

Issue #06	Typographical issue
Severity	 INFORMATIONAL
Location	<u>Line 38</u> <pre>(bool sent,) = address(msg.sender).call{value: gasAmount}("");</pre>
Description	address(msg.sender) is redundant as msg.sender is already an address.
Recommendation	Consider fixing the typographical issue.
Resolution	 RESOLVED

2.4 SearcherExecutionCapsule

SearcherExecutionCapsule is responsible for organizing the backrun. It ensures that all data provided by the user is actually valid: the backrun searcher must have authorized their involvement and when the backrun occurs, the searcher must have provided sufficient gas to the user for the backrun. Additionally, it organizes things as the bid price distribution to the user and protocol.

The highest price a searcher is willing to pay is distributed to the user's originators for 90%, equally amongst all originators, the remaining 10% goes to the protocol.

2.4.1 Privileged Functions

- `setSearcherRequestCall` [callable once]
- `withdrawEth`
- `withdrawAll`
- `backrun` [MetaSwapWrapperV2]
- `transferOwnership`
- `renounceOwnership`



2.4.2 Issues & Recommendations

Issue #07	Nonce logic is broken due to incorrect truncation, causing many nonces to point to the same storage slot
Severity	 HIGH SEVERITY
Location	<u>Line 91-92</u> <code>wordPos = uint256(nonce >> 8);</code> <code>bitPos = uint256(nonce);</code>
Description	The bitmap logic is supposed to provide a bitPos between 0-255. This can be achieved through <code>uint8(nonce)</code> , similar to how Permit2 achieves this. However, <code>uint256</code> was accidentally used, which will in the majority of cases cause the bitPos to overflow and result in the 0 position being used.
Recommendation	Consider using the trusted OpenZeppelin library called BitMaps to avoid running a custom implementation of this primitive: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/structs/BitMaps.sol By using a shared library, such mistakes can be avoided.
Resolution	 RESOLVED OpenZeppelin is used.

Issue #08**Searchers can be abused by executing their backruns on different users****Severity** **HIGH SEVERITY****Description**

The searcher gives a signature to the user, allowing them to utilize the searcher to backrun their transaction. Such a signature is important as the searcher subsidizes most of the backrun gas even if the transaction fails.

This signature signs many pieces of information to ensure that the backrun transaction matches the one the searcher expects, and is timely. However, a critical piece of information is not verified: the user address. This means that anyone can execute this signature, not just the user wallet the searcher expects.

Recommendation

Consider adding `msg.sender` to `userCallHash`.

Resolution **PARTIALLY RESOLVED**

The client argues that this design is intended and voices the following cases against our recommendation:

1. Adding `msg.sender` to `userCallHash` would expose the user address to the searcher. We prefer to avoid this in the future.
2. If another user executes the same swap transaction, it is still fine for the searcher.
3. If a user wants to break the searcher's backrunning strategy, they can frontrun.
4. There is no financial incentive to break the searcher execution.

We believe that issue (1) can be resolved by using a commit-reveal scheme of the user address within the hash (e.g. `userHash = keccak256(msg.sender, secret)`) and have therefore marked this as partially resolved, as we still believe this would make the system safer, even if searchers are supposed to be safeguarded against this.

Issue #09**Gas refund logic is fundamentally flawed****Severity** HIGH SEVERITY**Description**

Throughout the contract, gas refunds are introduced which refund the gas cost to the searcher if they did not use the whole gas allocation. However, these portions of code incorrectly assume that the `gasleft()` built-in function returns a wei amount. This is not the case as it returns a raw gas amount which must be multiplied by `tx.gasprice()`.

Thus, the whole gas management logic will fundamentally malfunction.

Recommendation

Consider fixing all gas related portions of code to use the correct units of accounting.

Resolution RESOLVED

This has been fixed throughout the contract



Issue #10	Various calls are not gas-limited
Severity	 MEDIUM SEVERITY
Description	Throughout the contract, attempts are made to ensure that the searcher does not waste gas. After all, they could be minting gas tokens or proof of work tokens to capture extra value, wasting the user's gas. However, the gas refunds are not capped in gas cost, allowing these calls to waste gas. Additionally, all these calls appear to be vulnerable to return bombing.
Recommendation	Consider using a gas-capped and return-bomb protected call.
Resolution	 RESOLVED All untrusted sections appear to now be protected against such attacks.

Issue #11	Gas optimizations
Severity	 INFORMATIONAL
Description	<u>Line 36</u> <pre>address public wethAddress;</pre> This variable can be marked as immutable to save gas. —— It appears sufficient to fetch searcherEOA from <code>searcherRequest.to.owner()</code> compared to also providing it within the data struct.
Recommendation	Consider implementing the gas optimizations mentioned above.
Resolution	 RESOLVED

DescriptionLine 13

```
interface IWETH {
```

This interface can extend IERC20 to allow it to be used as the type for the weth storage immutable and avoid casting in various sections of the codebase.

Line 22

```
event GaspriceOverSaercherLimit();
```

This event title contains a typo: "ae" should be "ea" within "Searcher".

Line 36

```
address public wethAddress;
```

Consider using IWETH as the type for this address.

Line 79

```
) EIP712("SearcherExecutionCapsule", "0.1.0") {
```

Though this does not appear to be a problem right now, the EIP 712 specification indicates that the versioning should be done on major versions and gives an example of "1" as the version. We recommend confirming whether this version format causes any issues with wallets etc.

Line 176

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

It is sensible to provide this array as an array of IERC20s to avoid subsequent casting.

Line 313

```
// Distribute bid to originators
```

This comment should state "distribute".

setSearcherRequestCall lacks an event.

Recommendation Consider fixing the typographical issues.

Resolution  RESOLVED



2.5 SearcherRequestCall

`SearcherRequestCall` is a dependency used by `SearcherExecutionCapsule` to execute the actual backrun transactions.

The reason for another layer of separation (compared to just executing within the capsule) is to ensure that no malicious actions can be taken within the backrunning call: for example, if the call was executed within the capsule, a malicious call could be an approval to an ERC20 token like WETH which might be in the capsule at that point. Since no tokens or any value at all is supposed to be stored within the `SearcherRequestCall`, such malicious actions do not appear to matter.

`SearcherRequestCall` also validates that the WETH balance of the capsule has increased by at least the bid which the searcher indicated they were willing to pay for the privilege of being able to execute the backrun.

2.5.1 Privileged Functions

- `transferOwnership`
- `renounceOwnership`



2.5.2 Issues & Recommendations

Issue #13	Low-level call is vulnerable to return-bomb griefing, which causes the searcher call to use significantly more gas than the desired provided gas
Severity	 MEDIUM SEVERITY
Description	Within low-level Solidity calls, the return data is always immediately stored within memory. This consumes non-linear gas as the cost of allocating memory is non-linear as the memory pointer increases. A griefing vector which has been documented and used in the passed is called the "return bomb" — in such a grief, the untrusted underlying contract returns a large amount of data causing the caller to use significant amounts of gas (as much as the underlying contract wants) outside of the allocated gas for the call.
Recommendation	Consider using the ExcessivelySafeCall implementation by Nomad: https://github.com/nomad-xyz/ExcessivelySafeCall Another way to prevent this return bomb is to also enforce the gas limit on the <code>searcherRequestCall.executeSearcherCall</code> call. In this case we do not believe the library above would be necessary.
Resolution	 RESOLVED ExcessivelySafeCall is used.

Issue #14	Ownable and SafeERC20 are unused
Severity	 INFORMATIONAL
Description	The contract extends the Ownable implementation and imports SafeERC20. However, these dependencies are unused.
Recommendation	Consider removing these dependencies.
Resolution	 RESOLVED

Issue #15	Gas optimization
Severity	 INFORMATIONAL
Description	<u>Lines 26 and 38</u> <pre>uint256 balanceBefore = IERC20(trustedCapsule.wethAddress()).balanceOf([...] IERC20(trustedCapsule.wethAddress()).balanceOf(The wethAddress is fetched twice within every iteration — this is exceptionally costly on chains like Ethereum, especially since this address is a constant. Consider instead caching it within the constructor within an immutable variable.</pre>
Recommendation	Consider implementing the gas optimization mentioned above.
Resolution	 RESOLVED



Issue #16	Typographical issues
Severity	 INFORMATIONAL
Description	<u>Line 19</u> <pre>bytes memory searchData,</pre> This variable can be marked as calldata. <u>Lines 30-31</u> <pre>bool success; (success,) = searcher.call{value: 0, gas: gas} (searcherData);</pre> This can be written into a single line by moving the bool within the brackets: (bool success,). <u>Line 41</u> <pre>"Bid was not repayed"</pre> This error should state that the bid was not "paid" as it does not need to be "re"paid (note that the correct spelling is also "repaid").
Recommendation	Consider fixing the typographical issues.
Resolution	 RESOLVED



PALADIN
BLOCKCHAIN SECURITY