Security Research Labs

Acala runtime audit

Security review of the Acala runtime implementation

v1.1 – 12. January 2022

Regina Bíró regina@srlabs.de

Mostafa Sattari mostafa@srlabs.de

Vincent Ulitzsch vincent@srlabs.de

Abstract. This report describes the results of a thorough, independent security assurance review of the Acala runtime.

SRLabs conducted a second security audit on the Acala codebase – 6 issues were identified across several runtime modules, most of which are already fixed by the Acala team.

While the security mindset and the application of best practices significantly improved since the first audit, we further suggest conducting thorough internal reviews on the extrinsic weight function implementation, leveraging continuous fuzz-testing during development, and improving documentation. Additionally, we recommend the usage of Sassafras to replace Aura for block authorship as soon as it is production ready.

Security Research Labs

Content

1 Scope and methodology

To ensure the security of the new features that have been added to the codebase of Acala since our last audit in 2020 [1], SRLabs has conducted a second review on the runtime implementation of Acala, namely the runtime modules (including community maintained ORML modules that Acala depends on [2]) in scope as indicated and summarized in Table 1. The goal of this review was an in-depth analysis of the implementation of the Acala codebase, ensuring that the new modules that were introduced to add new features for Acala for example, supporting EVM) do not compromise the security of the network. The analysis consisted of two parts:

1. Developing a threat and attack model against the Acala network (focusing on the modules in scope that could enable such attacks)
2. Reviewing and testing whether the Acala codebase is secured against all threats/attacks shown in the threat model

Repository	Runtime module	Previously audited	Audit completed
Acala	auction-manager	yes	yes
	cdp-engine	yes	yes
	cdp-treasury	yes	yes
	collator-selection	no	yes
	dex	yes	yes
	homa	no	yes
	honzon	yes	yes
	incentives	no	yes
	loans	yes	yes
	nft	no	yes
	support	no	yes
	transaction-pause	no	yes
	transaction-payment	no	yes
	evm (all modules)	no	yes
ORML	auction	yes	yes
	nft	no	yes
	rewards	no	yes
	traits	yes	yes
	utilities	yes	yes
	xcm	no	yes
	xcm-support	no	yes
	xtokens	no	yes

Stable-asset	stable-asset	no	yes

Table 1 Runtime modules in scope

During the audit, we carried out a hybrid strategy utilizing a combination of code review and dynamic tests (e.g. fuzz-testing) to assess the security of the Acala codebase. The detailed description of our audit approach was presented in the deliverable report of the first audit that we carried out for Acala [1].

2 Threat modeling and attacks

The threats presented in this section are described in detail as part of a holistic threat analysis of the Acala network, that was part of the audit process. The threat model has been shared with Acala in a separate deliverable earlier during the audit. While we identified and analyzed common threats in blockchain networks and in the Substrate ecosystem, we mainly focused on the ones that are specific and applicable to Acala's functionality and use-case. For each threat, we identified at least one attack scenario, which were then tested against during the static and dynamic analysis. A detailed description of our threat model framework can be found in [3].
According to our analysis, the following threats can be realized via the vulnerabilities found in the Acala codebase:

1. **Block timeout by 'cheap' extrinsics.** A malicious actor can assemble a list of transactions that do not exceed the maximum block weight, yet still produce a timeout by including extrinsics with high resource consumption. Validator nodes will try to include this list in a block, applying the transactions in the process, but then timeout. This would cause the validator node to miss its slot and halt block production as other validators would try to put the same transactions into the block, also failing.
   While the extrinsics in the Acala codebase were thoroughly benchmarked at the time of this audit, we found a couple of instances (described in [4] and [5]) where the weight functions of the extrinsics were not reflecting their computational complexity, and thus could be used to carry out such an attack.

2. **Stall block production by performing a DoS attack against collators.** By targeting collators, an attacker could attempt performing DoS attacks against them in order to prevent the collator from authoring a block. If successful, such attacks could result in a denial of service for the whole Acala parachain, affecting business and tampering with the reputation of the service. Using Aura, as described in [6] opens an attack vector for targeted DoS attacks against collators, since the next block author is always predictable by the design of Aura.

3. **Compromise node availability by crashing nodes.** Another way for an attacker to perform a denial of service attack is by crashing nodes – by for example crafting malicious transactions that trigger a crash on nodes that try to execute it. During the audit, we found an integer overflow in the XCM::V1 implementation that was exploitable by an extrinsic call in Acala [7], which would crash any node compiled in debug mode or with overflow checks enabled.

4. **Compromise node availability by exhausting the storage.** Storing infinite amount of data on storage must be desensitized (by for example deposits) to prevent a full blockchain storage which can lead to an infeasible amount of storage required to rune a node. We found some instances of missing storage deposits while reviewing Acala's code base (described in [8] and [9]).

5. **Cause financial harm to other users.** By causing financial harm to users, attackers can jeopardize the reputation of the network and its users' trust. Bugs in the runtime implementation can also result in financial loss to users in case they can be triggered by normal operations. By triggering the integer overflow vulnerability described in [7], on nodes that are compiled without overflow checks, the asset calculation would wrap around, resulting in an underestimated asset value to be transferred to a recipient.

3 Findings summary

We identified 6 issues summarized in Table 2 during our analysis of the runtime modules in scope Table 1 in the Acala codebase that enable the above-mentioned attacks. In summary, 1 high-severity, 4 moderate-severity issues and 1 low-severity issue were found.
Most of the vulnerabilities were already mitigated by the Acala team, as detailed in the table below. Acala decided to accept the risk posed by [6] until a better solution is available in the ecosystem. Additionally, after discussing with developers we consider issue [8] mitigated by charging a sufficiently high Existential Deposit.

Issue description	Severity	Reference	Mitigated
Integer overflow in the xcm::v1 pallet used in orml/xtokens could result in loss of asset tokens	high	[7]	PR 667 [10]
AURA collator selection opens up the possibility for DoS issue	moderate	[6]	Risk accepted
The weight calculation for EVM extrinsics does not take the computational complexity of substrate operations into account	moderate	[4]	PR 1674 [11]
Missing storage deposit in *claim_account* and *claim_default_account* extrinsics	moderate	[8]	By ED value

Missing storage deposit in *set_alternative_fee_swap_path*	moderate	[9]	PR 1730 [12]
Static weight calculation might underestimate extrinsic weights in the stable-asset pallet	low	[5]	Commit 0c1c317 [13]

Table 2. Overview of issues identified, detailed in [1]

The individual findings are summarized in the following sections.

3.1 Integer overflow in the xcm::v1 pallet could result in loss of asset tokens

We identified an integer overflow in Polkadot/xcm/src/v1/multiasset.rs in the *from* function that converts a vector of *MultiAsset* to one *MultiAssets* object. In the Acala runtime, this vulnerability could be triggered by an extrinsic call in the orml/xtokens pallet, for example by calling *transfer_with_fee*:

```
Call::XTokens(Call::transfer_with_fee { currency_id:
CurrencyId::Token(TokenSymbol::KAR), amount:
340282346643790096696078969353342681088, fee:
133444057505305437198628432850924339 1, dest: V1(MultiLocation { parents: 1,
interior: X1(AccountId32 { network: Named([1, 1, 1, 142, 175, 4, 175, 175, 175, 175,
175, 212, 0, 0, 0, 0, 0, 0, 0, 48, 20, 26, 189, 4, 169, 159, 214, 130, 44, 133, 88, 133, 76,
205, 227, 154, 86, 132, 231, 165, 109, 162, 125, 0, 1, 0, 50, 50, 50, 49, 49, 49, 141, 18,
63, 60, 33, 1, 0, 0, 0, 0, 1, 1]), id: [1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 31,
175, 175, 175, 175, 175, 175, 175, 175, 175, 175, 175, 175] }) }), dest_weight:
12659530246663417775 })
```

By exploiting this integer overflow, an attacker could:

- Crash any node compiled in debug mode or with overflow checks enabled.
- On nodes that are compiled without overflow checks, this will lead to the asset value calculation wrapping around and thus an underestimate of the asset value.

Parity Technologies mitigated this issue in 676c4dab0769c82ebf6ff68fbab8870c29cafcb6. Since the latest release, v0.9.13 does not contain the patch, we suggest to implement sanity checks in orml/xtokens to ensure that adding *assets.value* and *fee.value* does not lead to an integer overflow before calling *WithdrawAsset(vec![asset, fee.clone()].into())*. This issue was fixed in PR 667 [10].

3.2 AURA collator selection opens up the possibility for DoS issue

Currently, the Acala parachains (Karura, etc.) use AURA to identify which collator is eligible (and in fact, should) to produce a block at a given time. There are two security issues with AURA:

- **Predictability** The next block author is completely predictable well in advance. Thus, an attacker can easily cause a Denial of Service situation by DoSing the next block authors through a "rolling" Denial of Service, constantly switching targets to the next authors. DoSing a limited number of collators is feasible even with limited financial resources.
- **Missing Redundancy** This issue is exacerbated by the missing redundancy of AURA. At each given point in time, only one author is eligible to produce a block – if that author fails to produce a block, for example because of a DoS attack, no other node can fill in.

As a result of these two security concerns, it is arguably feasible for an attacker, even one with limited financial resources, to massively reduce the throughput of the parachain by always DoSing the collators that would be eligible to produce the next block.

As a long term solution, Parity is planning to switch to Sassafras from Aura in the future which should mitigate the aforementioned issue. We recommend to follow Parity and switch to Sassafras once it is released and stable.

Acala decided to accept the above-mentioned risk of using Aura for the time being, until Sassafras is released.

3.3 Weight calculation of EVM extrinsics underestimates computational complexity

In the EVM pallet (evm/src/lib.rs), all of the extrinsics wrapping around EVM calls (*eth_call*, *call*, *scheduled_call*, *create*, *create2*, *create_network_contract*, *create_predeploy_contract*) have the same weight which is the *gas_limit* argument converted into the substrate-based extrinsic weight.

However, most of these extrinsics do not only call the EVM Runner calls that they wrap around, but also perform some other substrate operations, for example database reads/writes or other method calls. The computational complexity (and as a result, weight) of these operations is not accounted for by only charging the gas limit (*T::GasToWeight::convert(*gas_limit))*) as the weight for these extrinsics.

As the weight of the substrate operations is not charged when calling these extrinsics, an attacker could craft EVM calls that reach their *gas_limit* when executed, making these extrinsics have an underestimated weight as a result. Underweighted extrinsics allow attackers to cause block timeouts by calling these insuffienctly weighted extrinsics that would take longer to execute than what their assigned weight indicates. In the case of parachains, having blocks that execute longer than required and as a result exceeding a block production time of 6 seconds could stall the block inclusion to the relay chain.

Note: not all the affected extrinsics are callable by any user, as some of them require a root origin. Still, underweighted root calls could accidentally cause the block production exceed the 6 seconds limit.

We suggest to benchmark the substrate operations in these extrinsics and add the benchmarked weight to *T::GasToWeight::convert(*gas_limit))* as the total weight for them. The fix was implemented in [11].

3.4 Missing storage deposit in *claim_account* and *claim_default_account* extrinsics

In the evm-accounts pallet the *claim_account* and *claim_default_account* extrinsics do not require a deposit when they are called. Since these extrinsics are callable by any Substrate account and this data is stored on-chain, an attacker could create an arbitrary number of accounts and fill up the storage by calling these extrinsics. A full blockchain storage is problematic because it could lead to an infeasible amount of storage being required to run a blockchain node. We recommended Acala to charge a storage deposit upon claiming accounts. After discussing the issue with the developers, we concluded that Acala's approach by charging a sufficiently high amount of Existential Deposit sufficiently mitigates the issue.

3.5 Missing storage deposit in *set_alternative_fee_swap_path*

In the transaction-payment pallet the *set_alternative_fee_swap_path* extrinsic takes the vector parameter *fee_swap_path* and adds it to the underlying data storage (*AlternativeFeeSwapPath).* While a sufficiently high existential deposit could compensate for the cost of the storage, there is another issue with this implementation in this pallet: the *AlternativeFeeSwapPath* record for dust accounts will not be removed automatically and thus will continue to reside on-chain forever and bloat the blockchain's storage.

Since this extrinsic is callable by any Substrate account and this data is stored on-chain, and attacker could create an arbitrary number of accounts and fill up the storage by calling this extrinsic (since this record is not removed after an account's balance drops below the existential deposit, attackers do not even need to lockup some funds to perform these attacks). A full blockchain storage is problematic because it could lead to an infeasible amount of storage being required to run a blockchain node.

We suggest requiring a deposit for calling this extrinsic or cleaning up the storage after an account is reaped. The fix was implemented in [12].

3.6 Static weight calculation underestimates weights in the stable-asset pallet

In the stable-asset pallet, the *mint*, *redeem_proportion* and *redeem_multi* extrinsics take an array of Balances (*Vec<T::Balance>)* as an input parameter. This vector is then iterated through in the extrinsics, making their computational complexity *O(n)* where *n* is the length of *Vec<T::Balance>.* The assigned weight functions to these extrinsics however, do not adapt to the length of these arrays, they just add the weight of db reads/writes to a constant benchmarked value.

If users call these extrinsics with the maximum allowed array length of the *Vec<T::Balance>* argument, the weight of the extrinsic might underestimate the *O(n)* execution time. Underweighted extrinsics allow attackers to cause block timeouts by

calling these insuffienctly weighted extrinsics that would take longer to execute than what their assigned weight indicates. In the case of parachains, having blocks that execute longer than required and as a result exceeding a block production time of 6 seconds could stall the block inclusion to the relay chain.

The *create_pool* extrinsic logic the number of assets was limited to maximum 3 at the time of the audit, hence users cannot call these extrinsics with an array of arbitrary length as an argument. However, since that limit may change in the future, we recommend to calculate the weights of these extrinsics as a dynamic function based on the length of the *Vec<T::Balance>* argument. The fix was implemented in [13].

4 Evolution suggestion

We saw a significant improvement in the security of the implementation logic of Acala since our previous audit in 2020. Overall, we found that the code was written with a security mindset and a strong emphasis on security best practices in the Substrate ecosystem. Most of the issues, as summarized in Table 2, were already mitigated by the Acala team.

One exception is the usage of Aura, that is currently considered the industry standard for block authorship on Parachains in the Kusama/Polkadot networks. The fact that Aura leaves the opportunity to attackers to predict the next block author and perform a targeted DoS attack against these collators is a known issue in the Substrate ecosystem and to Parity Technologies – a better mechanism, Sassafras [14] is currently being fine-tuned and implemented to close the security gap imposed by the usage of Aura and similar round-robin selection mechanisms. We therefore recommend switching to Sassafras for selecting the next block author from the collator set whenever it is declared production ready by Parity.

Despite that the approach of benchmarking extrinsics was thoroughly applied in the codebase following our suggestions from the previous audit, we identified two issues [4, 5] where the weight function was not corresponding to the (worst-case) computational complexity of the corresponding extrinsics. While the risk in these cases was significantly lower than using default weights, it could still allow attackers to compose transactions with these underweighted extrinsics in order to cause a block timeout. We suggest thoroughly reviewing the weight function implementations of newly added extrinsics or pallets before proceeding the benchmarking process.

While we found that Acala's implementation logic was not prone to integer overflows due to the usage of safe math, we identified an overflow bug [7] in the XCM::V1 implementation in Polkadot that was exploitable by an extrinsic call in Acala. To discover such bugs early on in the development lifecycle, we recommend leveraging continuous fuzz-testing. Acala can draw

some inspiration from the Substrate codebase, which includes multiple fuzzing harnesses based on honggfuzz [15].

One common pitfall in Substrate-based blockchain projects is that developers tend to confuse the transaction length fee with storage deposit. While the transaction fee does cover storing parameters on-chain, it's relatively cheap, and since over time the storage of a blockchain grows, it gets cluttered if it's not cleaned up regularly (or if there are missing measures preventing users from filling up the storage). Therefore, we suggest making sure that storing data on-chain is disincentivized by higher fees or deposits that users can unlock when they delete that item.

Incomplete documentation combined with the high complexity of the Acala design and codebase increases the risk of more vulnerabilities in the future – to mitigate this risk, we recommend extending the existing documentation and code annotation (especially concerning the ORML modules), being as verbose as possible to ease further development and bug fixes.

5 References

[1] "SRL-Laminar Security Review Findings," [Online]. Available: https://github.com/AcalaNetwork/Acala/blob/master/audit/SRL_Acala-review-online-public.pdf.

[2] "ORML modules," [Online]. Available: https://github.com/open-web3-stack/open-runtime-module-library.

[3] [Online]. Available: https://github.com/AcalaNetwork/srlabs-audit#readme.

[4] [Online]. Available: https://github.com/AcalaNetwork/srlabs-audit/issues/2.

[5] [Online]. Available: https://github.com/AcalaNetwork/srlabs-audit/issues/4.

[6] [Online]. Available: https://github.com/AcalaNetwork/srlabs-audit/issues/1.

[7] [Online]. Available: https://github.com/AcalaNetwork/srlabs-audit/issues/5.

[8] [Online]. Available: https://github.com/AcalaNetwork/srlabs-audit/issues/3.

[9] [Online]. Available: https://github.com/AcalaNetwork/srlabs-audit/issues/6.

[10] [Online]. Available: https://github.com/open-web3-stack/open-runtime-module-library/pull/667.

[11] [Online]. Available: https://github.com/AcalaNetwork/Acala/pull/1674.

[12] [Online]. Available: https://github.com/AcalaNetwork/Acala/pull/1730.

[13] [Online]. Available: https://github.com/nutsfinance/stable-asset/compare/13bf7a723aeeb14bf06e9ba4f4af06abf0446b5f...0c1c317a296201ff515d18d7ad60c81b1bbf1b3b.

[14] [Online]. Available: https://corepaper.org/substrate/consensus/sassafras/.

[15] "Honggfuzz," [Online]. Available: https://github.com/google/honggfuzz.