Security Research Labs

Astar dApp staking version 3:
Baseline Security Assurance

Threat model and hacking assessment report

V1.0, 31 January 2024

Audited by: Regina Bíró regina@srlabs.de
Rachna Shriwas rachna@srlabs.de
Haroon Basheer haroon@srlabs.de

Abstract. This work describes the result of the thorough and independent security assurance audit of the Astar dApp staking logic performed by Security Research Labs. Security Research Labs is a consulting firm that has been providing specialized audit services in the Polkadot ecosystem since 2019, including for the Substrate and Polkadot projects.

During this study, the Astar team provided access to relevant documentation, GitHub repository and supported the research team effectively for auditing their new components, the decentralized applications (dApps) staking mechanism version 3 and its corresponding inflation logic. The implementation of both components was verified to assure that the business logic of Astar network is resilient to hacking and abuse.

The research team identified several issues ranging from critical severity to low, many of which concerned the dApps staking and inflation logic.

Security Research Labs recommends considering opportunistic staking in the tokenomics model and conducting another assessment when additional features are implemented in the future.

Content

1 Disclaimer 3
2 Motivation and scope 4
3 Methodology 4
4 Threat modeling and attacks 6
5 Findings summary 9
6 Detailed findings 10
6.1 Issue 1: Double spending of tokens via infinite staking on unlock 10
6.2 Issue 2: Payouts round down to zero for collator and treasury rewards 11
6.3 Issue 3: Unregistered dApps not removed from storage 12
6.4 Issue 4: Loyal staker status can be abused 14
7 Evolution suggestions 15
7.1 Business logic improvement suggestions 15
7.2 Further recommended best practices 15
8 Bibliography 18

1 Disclaimer

This report describes the findings and core conclusions derived from the audit carried out by Security Research Labs within the agreed-on timeframe and scope as detailed in Table 1. Please note that this report does not guarantee that all existing security vulnerabilities were discovered in the codebase exhaustively and that following all evolution suggestions described in Chapter 7 may not ensure all future code to be bug free.

2 Motivation and scope

Blockchains evolve in a trustless and decentralized environment, which by its own nature could lead to security issues. Ensuring availability and integrity is a priority for Astar network to deploy the new dApp staking mechanism v3 for staking and reward distribution to its participants. As such, a security review of the project should not only highlight the security issues uncovered during the audit process, but also bring additional insights from an attacker's perspective, which the Astar network team can then integrate into their own threat modeling and development process to enhance the security of the product.

Astar is a blockchain network built on top of Substrate. Like other Substrate-based blockchain networks, the Astar code is written in Rust, a memory safe programming language. Mainly, Substrate-based chains utilize three technologies: a WebAssembly (WASM) based runtime, decentralized communication via libp2p, and a block production engine.

The Astar runtime consists of multiple modules compiled into a WASM Binary Large Object (blob) that is stored on-chain. Nodes execute the runtime code either natively or will execute the on-chain WASM blob.

The core business logic of Astar is a hub for dApps within the Polkadot Ecosystem. With Astar Network and Shiden Network, users can stake their tokens to a smart contract to reward projects that provide value to the network.

Security Research Labs collaborated with the Astar team to create an overview of the audit scope containing the local runtime configuration, dApp staking v3 pallet and precompile; and inflation model (referred together as dApp staking v3 from here on). The in-scope components and their assigned priorities are reflected in Table 1. During the audit, Security Research Labs used a threat model [1] to guide efforts on exploring potential security flaws and realistic attack scenarios. Additionally, Astar network's online documentation [2] [3] provided the testers with a good runtime module design and implementation overview.

Repository	Priority	Component(s)
https://github.com/AstarNetwork/Astar	High	pallets/dapp-staking-v3 pallets/inflation precompiles/dapp-staking-v3
	Medium	runtime/local

Table 1. In-scope Astar network components with audit priority

3 Methodology

This report details the baseline security assurance results for Astar with the aim of creating transparency in four steps, treat modeling, security design coverage checks, implementation baseline check and finally remediation support:

Threat Modeling. The threat model is considered in terms of *hacking incentives*, i.e., the motivations to achieve the goals of breaching the integrity, confidentiality, or availability of dApp staking v3. For each hacking incentive, hacking *scenarios* were postulated, by which these goals could be achieved. The threat model provides guidance for the design,

implementation, and security testing of Astar network. Our threat modeling process is outlined in Chapter 4.

Security design coverage check. Next, the Astar network design was reviewed for coverage against relevant hacking scenarios. For each scenario, the following two aspects were investigated:

a. **Coverage**. Is each potential security vulnerability sufficiently covered?

b. **Underlying assumptions**. Which assumptions must hold true for the design to effectively reach the desired security goal?

Implementation baseline check. As a third step, the current Astar network implementation was tested for openings whereby any of the defined hacking scenarios could be executed.

To effectively review the Astar network codebase, we derived our code review strategy based on the threat model that we established as the first step. For each identified threat, hypothetical attacks were developed and mapped to their corresponding threat category.

Prioritizing by risk, the codebase was assessed for present protections against the respective threats and attacks as well as the vulnerabilities that make these attacks possible. For each threat, the auditors:

1. Identified the relevant parts of the codebase, for example the relevant pallets and the runtime configuration.

2. Identified viable strategies for the code review. Manual code audits, fuzz testing, and manual tests were performed where appropriate.

3. Ensured the code did not contain any vulnerabilities that could be used to execute the respective attacks, otherwise, ensured that sufficient protection measures against specific attacks were present.

4. Immediately reported any vulnerability that was discovered to the development team along with suggestions around mitigations.

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 Astar network codebase.

While fuzz testing and dynamic tests establish a baseline assurance, the focus of this audit was a manual code review of the Astar network codebase to identify logic bugs, design flaws, and best practice deviations. We reviewed the Astar network repository up to the commit f9391d34926f1dbbc48e5fb537367a970f730b0f [4] in the master branch. The approach of the review was to trace the intended functionality of the runtime modules in scope and to assess whether an attacker can bypass/misuse/abuse these components or trigger unexpected behavior on the blockchain due to logic bugs or missing checks. Since the Astar network codebase is entirely open source, it is realistic that a malicious actor would analyze the source code while preparing an attack.

Fuzz testing is a technique to identify issues in code that handles untrusted input, which in Astar network's case is extrinsic in the runtime. (Note that the network part is handled by Substrate, which was not in scope for this review, but is built with a strong emphasis on security and where fuzz testing is also used). Fuzz testing works by taking some valid input for a method under test, applying a semi-random mutation to it, and then invoking the method under test again with this semi-valid input. Through repeating this process,

fuzz testing can unearth inputs that would cause a crash or other undefined behavior (e.g., integer overflows) in the method under test. The fuzz testing methods written for this assessment utilized the test runtime Genesis configuration as well as mocked externalities to execute the fuzz test effectively against the extrinsics in scope.

Remediation support. The final step was supporting Astar network with the remediation process of the identified issues. Each finding was documented and published with mitigation recommendations. Once the mitigation solution is implemented, the fix is verified by the auditors to ensure that it mitigates the issue and does not introduce other bugs.

During the audit, findings were shared via a private GitHub repository [5]. We also used a private Slack channel for asynchronous communication and weekly status updates – in addition, weekly meetings were held to provide detailed updates and to address open questions.

4 Threat modeling and attacks

The goal of the threat model framework is to be able to determine specific areas of risk in Astar network's blockchain system. Familiarity with these risk areas can provide guidance for the design of the implementation stack, the actual implementation of the stack, as well as the security testing. This section introduces how risk is defined and provides an overview of the identified threat scenarios. The *Hacking Value*, categorized into *low*, *medium*, and *high*, considers the incentive of an attacker, as well as the effort required by an attacker to successfully execute the attack. The hacking value is calculated as:

$$Hacking\ Value = \frac{Incentive}{Effort}$$

While *incentive* describes what an attacker might gain from performing an attack successfully, *effort* estimates the complexity of this same attack. The degrees of incentive and effort are defined as follows:

Incentive:

- Low: Attacks offer the hacker little to no gain from executing the threat.
- Medium: Attacks offer the hacker considerable gains from executing the threat.
- High: Attacks offer the hacker high gains by executing this threat.

Effort:

- Low: Attacks are easy to execute. They require neither elaborate technical knowledge nor considerable amounts of resources.
- Medium: Attacks are difficult to execute. They might require bypassing countermeasures, the use of expensive resources or a considerable amount of technical knowledge.
- High: Attacks are difficult to execute. The attacks might require in-depth technical knowledge, vast amounts of expensive resources, bypassing countermeasures, or any combination of these factors.

Incentive and Effort are divided according to Table 2.

Hacking Value	Low incentive	Medium Incentive	High Incentive
High effort	Low	Medium	Medium
Medium effort	Medium	Medium	High
Low effort	Medium	High	High

Table 2. Hacking value measurement scale.

Hacking scenarios are classified by the risk they pose to the system. The risk level, also categorized into low, medium, and high, considers the hacking value, as well as the damage that could result from successful exploitation. The risk of a threat scenario is calculated by the following formula:

$$Risk = Damage \times Hacking\ Value = \frac{Damage \times Incentive}{Effort}$$

Damage describes the negative impact that a given attack, performed successfully, would have on the victim. The degrees of damage are defined as follows:

Damage:

- Low: Risk scenarios would cause negligible damage to the Astar network
- Medium: Risk scenarios pose a considerable threat to Astar network's functionality as a network.
- High: Risk scenarios pose an existential threat to Astar network's network functionality.

Damage and Hacking Value are divided according to Table 3.

Risk	Low hacking value	Medium hacking	High hacking
Low damage	Low	Medium	Medium
Medium damage	Medium	Medium	High
High damage	Medium	High	High

Table 3. Risk measurement scale.

After applying the framework to the Astar network system, different threat scenarios according to the CIA triad were identified.

The CIA triad describes three security promises that can be violated by a hacking attack, namely confidentiality, integrity, availability.

Confidentiality:

Confidentiality threat scenarios concern sensitive information regarding the blockchain network and its users. Native tokens are units of value that exist on the blockchain - confidentiality threat scenarios include for example attackers abusing information leaks to steal native tokens from nodes participating in the Astar network ecosystem and claiming the assets (represented in the token) for themselves.

Integrity:

Integrity threat scenarios threaten to disrupt the functionality of the entire network by undermining or bypassing the rules that ensure that Astar network transactions/operations are fair and equal for each participant. Undermining Astar network's integrity often comes with a high monetary incentive, like for example, if an attacker can double spend or mint tokens for themselves. Other threat scenarios do not yield an immediate monetary reward, but rather, could threaten to damage Astar network's functionality and, in turn, its reputation. For example, tampering the new inflation model either by exploiting rounding or arithmetic bugs to earn extra tokens.

Availability:

Availability threat scenarios refer to compromising the availability of data stored by the Astar network as well as the availability of the network itself to process normal transactions. Important threat scenarios regarding availability for blockchain systems include Denial of Service (DoS) attacks on the staking mechanism by stalling or halting block production, stalling the transaction queue, and spamming.

Table 4 provides a high-level overview of the hacking risks concerning dApp staking with identified example threat scenarios and attacks, as well as their respective hacking value and effort. The complete list of threat scenarios identified along with attacks that enable them are described in the threat model deliverable [1]. This list can serve as a starting point to the Astar network developers to guide their security outlook for future feature implementations. By thinking in terms of threat scenarios and attacks during code review or feature ideation, many issues can be caught or even avoided altogether.

For dApp staking v3, the auditors attributed the most hacking value to the integrity class of threats. Since the efforts required to exploit this kind of issue is considered lower, we identified threat scenarios to the integrity of the staking mechanism of the highest risk category. Undermining the integrity of the dynamic inflation models of the Astar network means exploiting the rewards implementation to remove or claim free tokens. Some of the scenarios can have a direct effect on the financial model of the system. This can include market manipulation, gaining tokens for free or artificially manipulating the inflation to gain bonus rewards without repercussions.

Security promise	Hacking value	Example threat scenarios	Hacking effort	Example attack ideas
Confidentiality	High	N/A	High	N/A
Integrity	High	- Attackers could stake or transfer the same token to multiple dApps during the staking period causing reputation damage and unfair reward distribution - Attacker could abuse the loyalty status of an honest staker	Low	-Exploit weakness in dApp staking implementation to claim or spend the same token twice -Exploit a bug in the code to gain loyal staker status -Exploit bugs in the rewards implementation to remove or claim free tokens

Availability	Medium	- Attackers could try to sabotage the staking mechanism by stalling or halting block production - Attacker could DoS the dApp staking functionality to affect reward processing	Low	-Exploit arithmetic bugs in the reward or inflation calculation -Transaction spamming via underpriced extrinsics -Crash the chain by abusing reachable panic conditions in the configs during block initialization and finalization

Table 4. Risk overview. The threats for Astar network's blockchain were classified using the CIA security triad model, mapping threats to the areas: (1) Confidentiality, (2) Integrity, and (3) Availability.

5 Findings summary

We identified 4 issues - summarized in Table 5 - during our analysis of the local runtime modules in scope in the Astar codebase that enable some of the attacks outlined above. In summary, 2 critical severity, 1 high severity and 1 low severity issues were found.

Please note that in our methodology, critical severity issues refer to high severity issues that could be exploited immediately by an attacker on already deployed infrastructure, including a parachain or a non-incentivized testnet.

Issue	Severity	References	Status
The unlocking logic in dapp-staking-v3 allows attackers to double spend tokens via infinitely staking them on different accounts	Critical	[6]	Mitigated [7]
Collator and Treasury reward payout can round down to zero in the new inflation, thereby tampering with Astar's economic model	High	[8]	Mitigated [9] and [10]
Unregistered dApps prevent new dApps from registration when the *MaxNumberOfContracts* limit is reached	Medium	[11]	Mitigated [12]
A loyal staker will maintain their loyalty status after they unstake all their balance within the Voting sub-period, thereby remaining loyal	Low	[13]	Mitigated [14]

staker in the Build and Earn (B&E) sub period	

Table 5 Issue summary

6 Detailed findings

6.1 Issue 1: Double spending of tokens via infinite staking on unlock

Attack scenario	The unlocking logic in dapp-staking-v3 allows attackers to double spend tokens via infinitely staking them on different accounts
Location	pallets/dapp-staking-v3
Tracking	[6]
Attack impact	The attacker can stake the same tokens multiple times and gain rewards for the staked amount.
Severity	Critical
Status	Mitigated [7]

To stake an amount for a dApp, a user needs to first *lock* the amount and then *stake*. When a user wants to unstake from a dApp, the amount first needs to be *unstake*d and then *unlock*ed. The unlocking logic in dapp-staking-v3 marks the unlocked amount as free balance which can then be transferred to another account. The logic for *relock_unlocking* however, still considers the unlocked amount as *unlocked_chunks* and doesn't verify if the amount exists in the ledger or not. An attacker can exploit this by unlocking tokens, transferring them to another account, relock again and stake the non-existent tokens in the original account.

Please consider the following test as a PoC.

```
#[test]
fn lock_relock_unlocking_iter() {
    ExtBuilder::build().execute_with(|| {
        let account = 2;

        // We have 1000
        let free_balance = Balances::free_balance(&account);
        assert_eq!(free_balance, 1000);

        // Lock some amount
        let lock_amount = 500;
        assert_lock(account, lock_amount);

        // Start unlocking
        assert_unlock(account, lock_amount);

        // We transfer all of the free balance to another account
        let other_account = 42;
        assert_ok!(Balances::transfer_all(RuntimeOrigin::signed(account),
other_account, true));

        // Relock_unlocking will "relock" the 500, even though they have already
been transfered
        assert_relock_unlocking(account);

        // Account only has the existential deposit
        assert_eq!(Balances::free_balance(&account), EXISTENTIAL_DEPOSIT);
        assert_eq!(Balances::reserved_balance(&account), 0);
        assert_eq!(frame_system::Account::<Test>::get(&account).data.frozen,
500);
        // The other account has the remaining 998
        assert_eq!(Balances::free_balance(&other_account), 998);
```

```
30.         // Dapp_staking_v3::Ledger still thinks there is 500 locked!
31.         assert_eq!(Ledger::<Test>::get(&account).locked, 500);
32.
33.         // We can do that over and over again, re-locking the same balance on
different accounts
34.         for iter_account in 42..142 {
35.             assert_lock(iter_account, lock_amount);
36.             assert_unlock(iter_account, lock_amount);
37.
assert_ok!(Balances::transfer_all(RuntimeOrigin::signed(iter_account), iter_account
+ 1, true));
38.             assert_relock_unlocking(iter_account);
39.             assert_eq!(Balances::free_balance(&iter_account), 2);
40.             assert!(Balances::free_balance(&iter_account + 1) > 500);
41.             assert_eq!(Ledger::<Test>::get(&iter_account).locked, 500);
42.         }
43.     })
44. }
```

As a result, the attacker can stake the same tokens multiple times on a dApp, resulting in a higher staking score for the dApp and the attacker gaining more rewards. This can also potentially tamper with the inflation scheme of Astar.

The attack doesn't require deep technical knowledge about the Astar, regular users might also exploit it unknowingly, thinking that they can use the unlocked tokens immediately after calling unlock.

We suggest two alternatives to mitigate this issue:

- Implement a custom lock for the *UnlockingChunk* that doesn't allow them to be transferred/used.
- Do not lift the *DAppStaking* lock, until the unlocking has finished. Track the difference between locked tokens and *UnlockingChunk* in the ledger info.

The issue was mitigated [7] by Astar team by setting the freeze lock for *total_locked_amount* instead of *active_locked_amount* in *update_ledger* which would treat the unlocked chunks as frozen and not as free balance.

6.2 Issue 2: Payouts round down to zero for collator and treasury rewards

Attack scenario	Collator and Treasury reward payout can round down to zero in the new inflation, thereby tampering with Astar's economic model
Location	pallets/inflation
Tracking	[8]
Attack impact	The reward per block for collators can round down to zero, deterring non-Astar collators from participating in the block production.
Severity	High
Status	Mitigated [9] and [10]

The inflation recalculation can result in *collator_reward_per_block* and *treasury_reward_per_block* to be zero when the condition *blocks per cycle > collator_emission & treasury_emission* is true.

```
let collator_reward_per_block = collators_emission / blocks_per_cycle;
let treasury_reward_per_block = treasury_emission / blocks_per_cycle;
```

Consider for instance:

```
let collators_emission = params.collators_part * max_emission;
```

where *params.collators_part* is configured to be 3%, as in the mock similar to Tokenomics 2.0 specs. This results in *collators_emission* being smaller than the *blocks_per_cycle* causing *collator_reward_per_block* as 0.

The following UT confirms the assertion with similar configuration to UT *inflation_recalculation_works*:

```
#[test]
fn srl_test_collator_treasury_payouts() {
    ExternalityBuilder::build().execute_with(|| {
        //1. Initial parameters used as same in the UT
inflation_recalculation_works
        let total_issuance = Balances::total_issuance();
        let params = InflationParams::<Test>::get();
        let now = System::block_number();

        //2. Calculate new config
        let new_config = Inflation::recalculate_inflation(now);
        let max_emission = params.max_inflation_rate * total_issuance;

        //3. Assert the payout config
        //BUG: both reward reaches zero
        assert_eq!(new_config.collator_reward_per_block,0);
        assert_eq!(new_config.treasury_reward_per_block,0);

        //4.Do the payout using the new_config
        let collator_amount = <mock::Test as
pallet::Config>::Currency::issue(new_config.collator_reward_per_block);
        let treasury_amount = <mock::Test as
pallet::Config>::Currency::issue(new_config.treasury_reward_per_block);
        //Payouts to Zero
        println!("{:?},{:?}",collator_amount,treasury_amount);
        <mock::Test as
pallet::Config>::PayoutPerBlock::collators(collator_amount);
        <mock::Test as
pallet::Config>::PayoutPerBlock::treasury(treasury_amount);

    })
}
```

As collators receive zero rewards for block production, it could deter non-Astar collators from participating in block production and disrupt the staking mechanism. In addition, this could also lead to free balances and extra rewards floating in the system, breaking the economic model.

We recommend mitigating the issue by ensuring that collator and treasury rewards don't result in incorrect payouts, and *InflationConfiguration* payouts are distributed to all actors in the network in a way that ensures *max_emission* is fully burned within the cycle.

The issue was mitigated by Astar team by changing the *total issuance* and init inflation configuration in the pallet-inflation mock to be more realistic [9] and adding log messages in case collator rewards become zero [10].

6.3 Issue 3: Unregistered dApps not removed from storage

Attack scenario	Unregistered dApps prevent new dApps from registration when the *MaxNumberOfContracts* limit is reached
Location	pallets/dapp-staking-v3

Tracking	[11]
Attack impact	dApp developers are prevented from registering and marketing their dApp during the voting sub-period, subsequently affecting stakers from bidding on these dApps for bonus reward.
Severity	Medium
Status	Mitigated [12]

New dApps will not be allowed to register over time, when the *MaxNumberOfContracts* [15] count is reached. This is due to the *register* extrinsic logic counting all the *IntegratedDApps* including the unregistered dApps towards *MaxNumberOfContracts*, not because of malicious behaviour of the Manager origin.

During dApps' registration, the *register* extrinsic checks for:

- *IntegratedDApps* [16] count is within the range of *MaxNumberOfContracts* as follows: (say [A1])

```
ensure!(IntegratedDApps::<T>::count() < T::MaxNumberOfContracts::get().into(),
        Error::<T>::ExceededMaxNumberOfContracts);
```

Once the [A1] conditions holds, the dApp is inserted into the storage [17] using the same extrinsic, firing successful deposit event for dApp registration.

However, when the *unregister* extrinsic is called, the de-registered dApp is not removed from the *IntegratedDApps* storage. Rather it is updated with *DAppState* set to *Unregistered* [18]. This implies that the *IntegratedDApps::<T>::count()* will always be incremented, as more and more dApps are registered.

When [A1] doesn't hold during registration, no additional dApp will be allowed to register, throwing an *ExceededMaxNumberOfContracts* Error.

As the register is called using root call/Manager origin, this forces to perform runtime upgrade to increase the *MaxNumberOfContracts* or manually removing the unregistered dApps from the storage. Both these operations are expensive during dApp staking and may result in moving the chain to maintenance mode.

This may cause knock on effects on the new dApp developers from failing to register or participate in marketing during the voting sub-period, subsequently affecting stakers bidding on these dApps for bonus rewards.

Increasing the *MaxNumberOfContracts* in the runtime to a higher value is not an ideal solution (currently set to 100 in local runtime and 500 in shibuya runtime), as the ceiling (u32 MAX) will be reached at some point. The ideal mitigation will be:

1. Remove the de-registered dApps from the *IntegratedDapps* storage as proposed here when the unregister extrinsic is called (similar to removing the *ContractStake* for de-registered dApps)

2. If mitigation 1 is not feasible, in favor of keeping the historical records of the dApps registered. Consider:

Ensure by counting the *DAppState::Registered* state in the DAppInfo stays within the range of *MaxNumberOfContracts* during register instead of [A1]. This helper function may be useful to keep count of the currently registered dApps.

The issue was mitigated by Astar team by removing the unregistered dApps from *IntegratedDApps* storage and replacing *NotOperatedDApp* error with *ContractNotFound* error. The *dApp Id* cleanup will be addressed later in the PR 1152 [19].

6.4 Issue 4: Loyal staker status can be abused

Attack scenario	A loyal staker will maintain their loyalty status even after they unstake all their balance in the Voting sub-period
Location	pallets/dapp-staking-v3
Tracking	[13]
Attack impact	The loyalty status may be exploited by any staker for dApps if the developer relies on this status for additional payouts or specific privileges within their implementation.
Severity	Low
Status	Mitigated [14]

By staking at least once during a voting sub period, a staker will be loyal. If they are able to *unstake* all their balance before the end of same sub-period, they remain a loyal staker for the dApp in the B&E sub period.

The following UT confirms the assertion:

```
 1. //Execute inside test_types.rs
 2. fn srl_check_loyal_staker_during_BE(){
 3.     //1. Stake some amount during voting sub period
 4.     let period_number = 1;
 5.     let subperiod = Subperiod::Voting;
 6.     let mut staking_info = SingularStakingInfo::new(period_number, subperiod);
 7.     let mut era_1 = 2;
 8.     let vote_stake_amount_1 = 100;
 9.     staking_info.stake(vote_stake_amount_1, era_1, Subperiod::Voting);
10.     assert!(staking_info.is_loyal());
11.
12.     //2. lets unstake everything during Voting period, voting balance is zero
13.     let unstake_amount_1 = 100;
14.     assert_eq!(
15.         staking_info.unstake(unstake_amount_1, era_1, Subperiod::Voting),
16.         (unstake_amount_1, Balance::zero())
17.     );
18.     assert!(staking_info.is_loyal()); //its a feature, because its voting sub
period
19.
20.     // 3. B&E sub-period, stake and unstake
21.     let bep_stake_amount_1 = 23;
22.     staking_info.stake(bep_stake_amount_1, era_1+1, Subperiod::BuildAndEarn);
23.     staking_info.unstake(23, era_1+2, Subperiod::BuildAndEarn);
24.     let remaining_stake = staking_info.total_staked_amount();
25.     assert_eq!(remaining_stake,Balance::zero());
26.     // BUG: Cant be loyal staker??
27.     assert!(staking_info.is_loyal());
28. }
```

The potential risks associated are:

- The loyalty status may be exploited by any staker for dApps if the developer relies on this status for additional payouts or specific privileges within their implementation.

- This disrupts the concept of maintaining loyalty status, as a staker with zero balance during the Voting sub-period still retains their loyal status.

- While the incorrect loyalty status presently does not lead to an additional bonus reward payout, it has the potential to cause user inconvenience when claiming bonus rewards in the extrinsic *claim_bonus_reward,* resulting in *InternalClaimBonusError.*

We suggest to reset the loyalty flag when the staking balance reaches zero during the voting sub-period to avoid incorrect state transitions into the B&E sub-period. Fix the unstake as:

```
self.loyal_staker = self.loyal_staker
          && (self.staked.voting != zero()
          || subperiod == Subperiod::BuildAndEarn
          &&  self.staked.voting == snapshot.voting);
```

The issue was fixed by Astar team by resetting the *loyal_staker* flag to false if the staked amount in Voting period becomes zero [14].

7 Evolution suggestions

The overall impression of the auditors was that Astar's dApp Staking as a product is designed and written with security in mind. To ensure that Astar network is secure against unknown or yet undiscovered threats, we recommend considering the evolution suggestions and best practices described in this section.

7.1 Business logic improvement suggestions

To enhance the security and robustness of the dApp staking and inflation design, we recommend considering the following suggestions:

Ensure fair bonus reward for early stakers at the start of an era during voting sub-period. To promote equitable bonus payouts for stakers at the onset of a new era during the voting against staking towards the end of the last era, the calculation for bonus rewards may incorporate the era number in the formula. This also prevents stakers from influencing the tier level of a dApp from promotion or demotion through opportunistic last-minute staking. This scenario should be considered when a new economic audit is performed to update the Tokenomics report v2.0 [2] and its implementation in the inflation pallet.

Periodically audit currently registered dApps' performance and business logic to prevent them from becoming rogue/malicious. Implement a regular security audit program for the deployed dApps within the Astar ecosystem. This program should encompass a thorough examination of the dApps' business and implementation to guarantee their validity and security of its intended functionality. Additionally, it is recommended to periodically involve external security experts for impartial evaluations. To maintain transparency and keep all stakeholders informed, ensure the publication of the audit reports and assessments detailing the progress and outcomes of the security audits for the dApps.

7.2 Further recommended best practices

Regular code review and continuous fuzz testing. Regular code reviews are recommended to avoid introducing new logic or arithmetic bugs, while continuous fuzz testing can identify potential vulnerabilities early in the development process. Ideally,

Astar should continuously fuzz their code on each commit made to the codebase. The substrate-runtime-fuzzer [20] (which uses Ziggy [21], a fuzzer management tool) can be a good starting point.

Regular updates. New releases of Substrate may contain fixes for critical security issues. Since Astar network is a product that heavily relies on Substrate, updating to the latest version as soon as possible whenever a new release is available is recommended.

Appropriate benchmarking. Inappropriate benchmarking can lead to overestimation/underestimation of weights which can be exploited by an attacker for their advantage. One such case of overestimation was found during the audit:

- In the function *get_dapp_tier_assignment*, the counter is incremented at line 1604 [22], even if the dApp has zero stake. This counter is used for benchmarking and thus could result in a slight overestimation of the weight.

  The issue was acknowledged by the Astar team, and the code would be optimized in the future.

Miscellaneous best practices and recommendations. During the audit, a few code discrepancies and missing best practices were reported to the Astar team which are detailed below:

1. The dApp sorting into tiers is done using *sort_unstable_by* which will give lower position to the dApp with lower dApp ID in case two dApps have same staking score. So, the dApps that were registered first, will get the priority in the list. The comments in the source code [23] and the README [24] give the impression to the users that the sorting behavior is undefined. It was recommended to update the comments to reflect the current behavior.

   The suggestion was acknowledged, and the source code comments [25] and README [26] were modified to remove the confusion in sorting behavior.

2. The *pub maintenance: bool* [27] visibility for the *ProtocolState* struct should be set to a private for maintenance mode to prevent overriding of the default implementation [28]. Although setting the maintenance mode requires root/manager origin, making the fields private enforces an additional security guarantee to the *ProtocolState*. It was recommended to remove the pub identifier from the struct fields.

   The recommendation was acknowledged, and it was discussed that the pallet-dapp-staking-migration manipulates the maintenance mode when doing the migration, so it needs access to it. The suggested change can be applied once dApp staking v3 has been deployed on all the networks and the migration pallet can be removed.

3. Set safe maximum for *InflationParameters* (probably in *fn is_valid()*) and *InflationConfiguration*

   The function *is_valid()* [29] checks if the sum of all inflation parameters is one whole, but there is no maximum limit for individual parameters. For example, one of the parameters could be 80% and others summed together 20%. This would be mathematically valid but could harm the inflation model.

   Additionally, there are no bounds checks implemented for *InflationConfiguration*. It is recommended to add sanity checks for expected ranges of the parameters.

It was discussed with the Astar team that in case a parameter value in *InflationParameters* becomes excessively low or excessively large, it will not stop or break the chain and is something that can be remedied quickly. For *InflationConfiguration,* warnings were added in the code [10] in case any of the reward values becomes zero.

4. We observed that safe math operations (such as *saturating_mul, saturating_div*) are not used everywhere for inflation calculation [30]. It was recommended to always use safe math operations to avoid overflow, underflow, and division by zero errors.

   The integer division issue was resolved in PR 1146 [10].

5. Remove force extrinsics from the following pallets: dApp staking and inflation (such as *force_set_inflation_config, force_set_tier_config, force_set_tier_params and Error InvalidTierParameters*). This was already noted in the ToDo code comments and the Astar team was notified during the regular sync calls. As these functions have security implications on the dApp staking during production, its recommended to remove them prior to the mainnet launch. This will also improve the codebase's readability and ease of maintenance.

8 Bibliography

[1] [Online]. Available: https://securityresearchlabs.sharepoint.com/:x:/s/Astar/EdyWlhEtLP5GhaaG6x55oqEB8expHnKWp72EYLjhKVP8BQ?e=o0fwq8.

[2] [Online]. Available: https://forum.astar.network/t/astar-tokenomics-2-0-a-dynamically-adjusted-inflation/4924.

[3] [Online]. Available: https://forum.astar.network/t/dapps-staking-v3-proposal/4206.

[4] [Online]. Available: https://github.com/AstarNetwork/Astar/tree/f9391d34926f1dbbc48e5fb537367a970f730b0f.

[5] [Online]. Available: https://github.com/AstarNetwork/dappstaking-v3-audit/issues?q=is%3Aissue+is%3Aclosed.

[6] [Online]. Available: https://github.com/AstarNetwork/dappstaking-v3-audit/issues/1.

[7] [Online]. Available: https://github.com/AstarNetwork/Astar/pull/1111.

[8] [Online]. Available: https://github.com/AstarNetwork/dappstaking-v3-audit/issues/5.

[9] [Online]. Available: https://github.com/AstarNetwork/Astar/pull/1144.

[10] [Online]. Available: https://github.com/AstarNetwork/Astar/pull/1146.

[11] [Online]. Available: https://github.com/AstarNetwork/dappstaking-v3-audit/issues/6.

[12] [Online]. Available: https://github.com/AstarNetwork/Astar/pull/1153.

[13] [Online]. Available: https://github.com/AstarNetwork/dappstaking-v3-audit/issues/4.

[14] [Online]. Available: https://github.com/AstarNetwork/Astar/pull/1136.

[15] [Online]. Available: https://github.com/AstarNetwork/Astar/blob/0b0d082274cc735b82b1fc3996dee9dc3eafe7b6/pallets/dapp-staking-v3/src/lib.rs#L158.

[16] [Online]. Available: https://github.com/AstarNetwork/Astar/blob/0b0d082274cc735b82b1fc3996dee9dc3eafe7b6/pallets/dapp-staking-v3/src/lib.rs#L626-L629.

[17] [Online]. Available: https://github.com/AstarNetwork/Astar/blob/0b0d082274cc735b82b1fc3996dee9dc3eafe7b6/pallets/dapp-staking-v3/src/lib.rs#L635-L643.

[18] [Online]. Available: https://github.com/AstarNetwork/Astar/blob/0b0d082274cc735b82b1fc3996dee9dc3eafe7b6/pallets/dapp-staking-v3/src/lib.rs#L758.

[19] [Online]. Available: https://github.com/AstarNetwork/Astar/issues/1152.

[20] [Online]. Available: https://github.com/srlabs/substrate-runtime-fuzzer.

[21] [Online]. Available: https://github.com/srlabs/ziggy.

[22] [Online]. Available: https://github.com/AstarNetwork/Astar/blob/d9a76ff3f49b7dee423a558d4cdf72bfb5f99e63/pallets/dapp-staking-v3/src/lib.rs#L1604C17-L1604C17.

[23] [Online]. Available: https://github.com/AstarNetwork/Astar/blob/d9a76ff3f49b7dee423a558d4cdf72bfb5f99e63/pallets/dapp-staking-v3/src/lib.rs#L1658C12-L1658C12.

[24] [Online]. Available: https://github.com/AstarNetwork/Astar/blob/d9a76ff3f49b7dee423a558d4cdf72bfb5f99e63/pallets/dapp-staking-v3/README.md.

[25] [Online]. Available: https://github.com/AstarNetwork/Astar/blob/f9391d34926f1dbbc48e5fb537367a970f730b0f/pallets/dapp-staking-v3/src/lib.rs#L1733.

[26] [Online]. Available: https://github.com/AstarNetwork/Astar/blob/master/pallets/dapp-staking-v3/README.md.

[27] [Online]. Available: https://github.com/AstarNetwork/Astar/blob/0cba858fe201c0dfd31f9db12eb44b1b307d7f51/pallets/dapp-staking-v3/src/types.rs#L198.

[28] [Online]. Available: https://github.com/AstarNetwork/Astar/blob/0cba858fe201c0dfd31f9db12eb44b1b307d7f51/pallets/dapp-staking-v3/src/types.rs#L201.

[29] [Online]. Available: https://github.com/AstarNetwork/Astar/blob/f9391d34926f1dbbc48e5fb537367a970f730b0f/pallets/inflation/src/lib.rs#L588.

[30] [Online]. Available: https://github.com/AstarNetwork/Astar/blob/master/pallets/inflation/src/lib.rs#L354-L386.