Tari

Base Layer

Security Assessment

coinspect

Tari Base Layer

Security Assessment

Version: v240115 | Prepared for: Tari | January 2024

Base Layer Audit

Executive Summary

Summary of Findings

Solved issues & recommendations

Assessment and Scope

Audit Process

Review of Fixes

Overview

Detailed Findings

Disclaimer

© Coinspect 2024

Executive Summary

In **July 2023**, **Tari** engaged **Coinspect** to perform an audit of their base layer, the **Tari Base Layer**. This engagement included an audit of their L1 node as well as the Tari Wallet. The engagement lasted for a total of 15 weeks.

The following issues were identified:

	Solved	Caution Advised	Resolution Pending
High	22	0	0
Medium	9	0	0
Low	10	0	0
No Risk	9	0	0
Total	**50**	**0**	**0**

Auditors found critical issues in most areas of the codebase, which led to 44% of the identified vulnerabilities being of HIGH severity. The two most important categories of findings were those related to proof of work checks (**TARI-005**, **TARI-006**, **TARI-008**, **TARI-012**) and arbitrarily triggered denial of services (**TARI-009**, **TARI-010**, **TARI-014**, **TARI-021** and others). These issues would have resulted in double spends if leveraged by attackers.

Many other types of issues and bugs were identified: miners could be delayed by attackers to win an unfair advantage (**TARI-001**, **TARI-002**), the wallet software is

© Coinspect 2024

vulnerable to denial of services (**TARI-024**), nodes can waste resources validating attacker-crafted data (**TARI-003**, **TARI-004**, **TARI-007**, **TARI-015**).

Coinspect also found exploitable weaknesses in Tari's dependencies (**TARI-028**, **TARI-030**). These were shared with the Tari team and disclosed to the maintainers.

All issues were correctly addressed by Tari's team and are now mitigated.

© Coinspect 2024

Summary of Findings

Solved issues & recommendations

These issues have been fully fixed or represent recommendations that could improve the long-term security posture of the project.

Id	Title	Risk
TARI-001	Block producer can delay competing miners	High
TARI-002	Attackers can prevent block processing	High
TARI-005	Attackers can mine several blocks at once by abusing ExtraField data	High
TARI-006	Attackers can mine several blocks at once by using multiple Monero coinbase transactions	High
TARI-008	Attackers can create the same block with different hashes	High
TARI-009	Attackers can halt long-syncing by inflating block headers	High
TARI-010	Attackers can halt long-sync by appending trailing data to Proof of Work buffer	High
TARI-011	Attackers can crash nodes by manipulating proof of work data	High
TARI-012	Proof of work can be copy-pasted	High
TARI-013	Attackers can force sync on a peer eternally	High
TARI-014	Miner can ban competition	High
TARI-016	Attackers can halt synchronization	High

© Coinspect 2024

TARI-019	Attackers can crash a node by forcing over-allocation	High
TARI-021	Attackers can crash nodes by broadcasting JOIN messages	High
TARI-022	Attackers can crash miners by sending a block with a high coinbase fee	High
TARI-028	Attackers can crash the node with a malicious Monero coinbase	High
TARI-030	Peer can crash horizon-syncing node with a crafted bitmap	High
TARI-031	Attackers can crash the node with a script	High
TARI-032	Attackers can crash node with a mempool message	High
TARI-035	Attackers can crash the node by sending transaction with big fee	High
TARI-038	Miner can force peers to sync	High
TARI-049	Attackers can crash whole node by panicking a single thread	High
TARI-003	Nodes waste resources validating fake orphan blocks	Medium
TARI-004	Attackers can spam mempool at no cost	Medium
TARI-017	Attackers can avoid ban on horizon sync	Medium
TARI-023	Attackers can observe and manipulate gRPC requests to wallets	Medium
TARI-024	Attackers can crash the wallet via the gRPC service	Medium
TARI-026	Attackers can hold the peer-stream open indefinitely	Medium
TARI-029	Attackers can crash the node with a single gRPC request	Medium
TARI-042	Attackers can hold the DB write lock with low difficulty blocks	Medium
TARI-044	Attackers can flood victim with peer addresses	Medium
TARI-007	Malicious script makes validators waste computations	Low

TARI-015	Known bad blocks are validated	Low
TARI-020	Default output selection criteria can be abused to increase the sender's fee	Low
TARI-025	Wallet username vulnerable to timing attacks	Low
TARI-027	Overflow when computing accumulated difficulty will eventually halt mining	Low
TARI-034	Wrong test for double spend	Low
TARI-039	Wallet reports stale balance results with no warnings	Low
TARI-041	Attackers can continuously spam low difficulty blocks	Low
TARI-045	Some errors are not obscured by gRPC server	Low
TARI-050	Privacy compromise fetching code template repository URL	Low
TARI-018	Consensus is dependent on platform	None
TARI-033	Dependencies that depend on wrap-on-overflow will crash Tari	None
TARI-036	Overflows can potentially crash the node	None
TARI-037	Sync is always done with a single peer	None
TARI-040	DirectOnly wallets report the transaction being sent when it is not	None
TARI-043	Wallet can minimize trust with base node by requesting proof of work	None
TARI-046	Not possible to disable gRPC methods	None
TARI-047	Inaccuracy on pruning on RFC-0140	None
TARI-048	No warning when using weak or empty password on account recovery	None

© Coinspect 2024

Assessment and Scope

The audit started on **July 3, 2023** and was conducted on the commit `87c070305951152c62a0179e13fadc55065cc318` tag `v0.51.0-pre.4` of the https://github.com/tari-project/tari repository.

On **September 4, 2023**, Tari provided a new commit for Coinspect to review. The commit intended to fix some of the vulnerabilities reported by Coinspect and also contained fixes for other bugs found by the Tari team. This new commit is `63f61ebc2639730060d45ed63d0b839b0b321565` tag `0.52.0-pre.1` of the https://github.com/tari-project/tari repository.

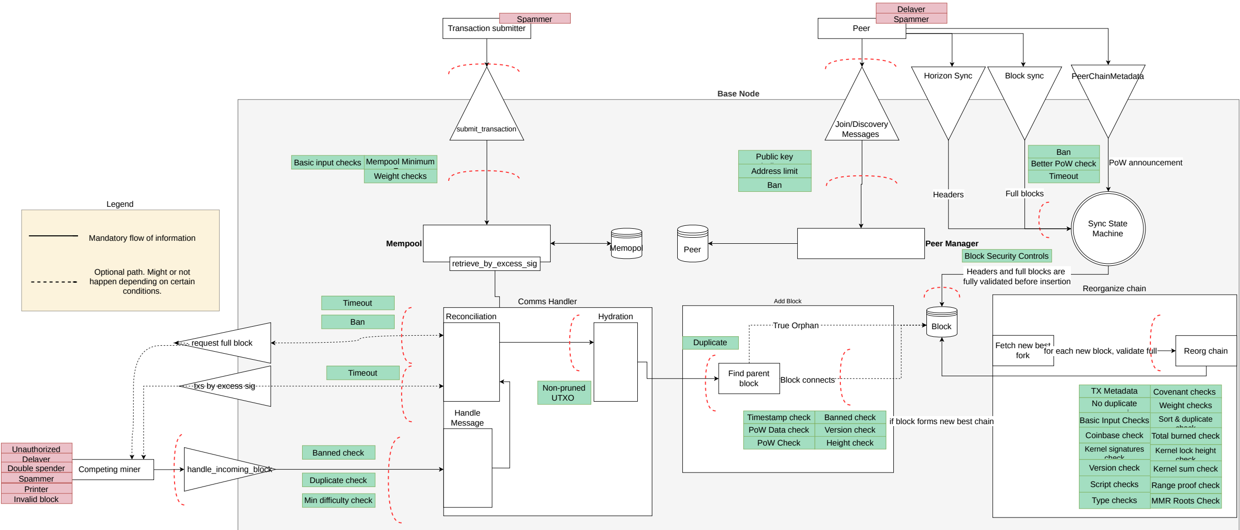
The audit was focused on the following folders:

```
/base-layer/core/consensus
/base-layer/core/proof_of_work
/base-layer/core/transactions
/base-layer/core/validation
/base-layer/core/mempool
/base-layer/wallet/
/base-layer/wallet_ffi
/base-layer/tari_script/
/base-layer/key_manager/
/comms/dht/src/crypt.rs
/comms/dht/src/outbound/
/comms/dht/src/inbound/
/comms/core/
/base-layer/core/base_node
/base-layer/core/blocks
/base-layer/core/chain_storage
/base-layer/core/covenants
/base-layer/core/proto
```

Audit Process

Coinspect followed its standard audit process for L1 layers, spending the first weeks understanding the threats and risks in the Tari threat model and its differences with other blockchains to better identify vulnerabilities. This was an ongoing process during the whole audit which culminated in a diagram of Tari's threat model which was shared with the Tari team at the end of the audit.

© Coinspect 2024



Coinspect's threat model for Tari. Full resolution.

By **Week 6**, auditors had discovered 11 high severity issues on the platform. Due to the amount, severity and nature of the issues Coinspect communicated to Tari that the best path forward in order to minimize risk was to let Tari work on the issues found, provide a new commit and allocate time of the audit to re-review these components.

Tari decided to accept the proposal and extend the audit by two weeks.

Week 9 was set as the last one where auditors would review the original commit. By **Week 10**, Tari had a new commit-to-review and auditors used it moving forward.

The issues reported here from **TARI-001** to **TARI-031** were found in the original commit.

Starting from **Week 11**, Tari and Coinspect agreed that details about unconfirmed overflows or underflows would be collected in an **INFO** issue.

Review of Fixes

Coinspect started reviewing **Tari**'s fixes on **November 10, 2023**. This review was conducted on the commit `14e334aff346aae8a081599488135c905c2c1f84` of the https://github.com/tari-project/tari repository on the `audit-fixes` branch.

The changes in the codebase up to `14e334aff346aae8a081599488135c905c2c1f84` intend to fix vulnerabilities which were not addressed during the audit itself, except for the fix **TARI-023** which was delayed to a later commit. Coinspect also found that **TARI-032** was incorrectly mitigated.

© Coinspect 2024

On November 30, Tari provided a new commit `4be85e0f0d778ee813fa41df927cfcac79a3f1db` on the same `audit-fixes` branch. This commit has solved the incomplete fix to **TARI-032** and partially addressed the problems outlined in **TARI-023**. On December 5, a new commit `17676ce8f90b0f2aa413218cc72c21c587293d32` was shared with Coinspect, which aimed to totally remedy **TARI-023**. To fully asses the fix of this issue, Coinspect also considered some commits on the `development` branch, particularly commit `b80f7e366b14e10b3fb0e9835fb76dd5596d0cf8`. The `development` branch was **not** audited, it was only reviewed in relation to **TARI-023**.

Overview

Tari aims to be the most useful, decentralized platform that empowers anyone to create digitally scarce things people love. It is a new cryptocurrency implementation inspired in Monero and the Mimblewimble protocol. The implementation is programmed in Rust from scratch.

The blockchain operates on a UTXO-based structure with private transactions, ensuring confidentiality by hiding addresses and amounts. Transaction aggregation facilitates a compact blockchain, and its history is prunable, ensuring efficient storage while maintaining the currency's total supply verification.

Two synchronization methods are available:

- Block sync, which downloads and verifies the full blocks
- Horizon sync, which only downloads information necessary to assert no unintended inflation happened

Node operators are able to choose how many full blocks should be kept before they are pruned.

The system adopts the Nakamoto Consensus via a hybrid proof of work mechanism, integrating Monero's RandomX merge mining with Sha3x proof of work. These two algorithms are configured with different difficulty targets and the aim is for a 60/40 block distribution on each mining strategy and a two-minute block interval.

UTXOs are accessible either through showing ownership of the excess blinding factor (like in Mimblewimble) or via TariScript, a system akin to Bitcoin's Script. Tari also introduces covenants to define spending conditions.

A significant part of the threat model of Tari is its wallet. Unlike many traditional blockchains, it plays a crucial role due to the potential interactivity of transactions,

necessitating negotiation between online peers. The wallet interfaces with the node through an RPC system.

Tari offers in-depth documentation and rationale of their design decisions organized mostly in the form of RFC documents.

During Coinspect's assessment, two components - merged mining and synchronization processes - emerged as areas of concern due to the severity and number of issues identified.

The merged mining implementation was initially vulnerable to several attacks: it allowed miners and intermediate nodes to skip the PoW, modify headers and reuse PoW, inflate headers to perform denial of service or trigger a panic crashing the node. These issues led to a changes in the design and implementation of the RandomX PoW.

The synchronization processes were notably vulnerable to several DoS scenarios. These vulnerabilities were accentuated by the fact that nodes synchronize with only one peer and the ease with which synchronization status can be manipulated.

Coinspect also noted that two project-wide policies were problematic:

1. Panic on overflow configuration
2. Unhandled panics leading to program halt

The panic-on-overflow configuration is not traditional for Rust-based projects, and while it has some benefits, the codebase performed unsafe math operations ignoring this policy, which led to several denial-of-service issues reported via over or underflows.

The handling of panics was considered risky enough so merit its own issue described in **TARI-049**. This was combined with the already mentioned panic-on-overflow policy with severe results, as it meant that a crash on a single thread resulted in a crash of the whole process.

Another point leveraged in many issues is the decision to use global locks for multiple operations. This allowed several types of attacks where miners could take advantages that are similar to selfish mining, but with greater impact.

During the audit process, Tari team showed to be open to communication, promptly addressing emerging issues as they arose. The team also showed proactivity identifying issues similar to those reported in highlighted areas of the codebase.

© Coinspect 2024

Detailed Findings

TARI-001

Block producer can delay competing miners

Location

`base_layer/core/src/base_node/comms_interface/inbound_handlers.rs`

Description

A miner that produces a new *best block* can deliberately slow down their network communication, forcefully acquiring the *new block lock* of peers by up to two minutes.

The full impact of this attack comes from a miner leveraging this to attack other miners. An evil miner using this attack would gain unfair advantage over other miners and is encouraged to exploit it.

© Coinspect 2024

The attack works as follows: when a miner finds a new block, they propagate it to the network. Receiving nodes process the new block message in the `handle_block_message` method of the `InboundNodeCommsHandler`.

It is important to note that only a single *new block message* can be processed at a time:

```
        // Only a single block request can complete at a time.
        // As multiple NewBlock requests arrive from propagation, this
semaphore prevents multiple requests to nodes for
        // the same full block. The first request that succeeds will
stop the node from requesting the block from any
        // other node (block_exists is true).
        // Arc clone to satisfy the borrow checker
        let semaphore = self.new_block_request_semaphore.clone();
        let _permit = semaphore.acquire().await.unwrap();
```

After the lock is acquired, receivers call `reconcile_block`. This method makes up to two requests to the sender:

1. `FetchMempoolTransactionsByExcessSigs`
2. `GetBlockFromAllChains`

It is important to note that a miner can force both requests to be made: the miner must add a transaction to the block that they have not broadcasted to the network to force the `FetchMempoolTransactionsByExcessSigs` to be made and then they must answer that they have `not_found` that transaction, forcing the receiver to request the whole block.

Joining all of these facts together, we can see that a miner can force other nodes to stop listening to new block requests for about two minutes by following these steps:

1. Add a transaction to the miner's mempool that will not be broadcasted
2. Find a new valid block for the chain
3. Broadcast this block to peers
4. Wait 59 seconds
5. Respond to `FetchMempoolTransactionsByExcessSigs` with some transaction in the `not_found` array
6. Wait 59 seconds
7. Respond to `GetBlockFromAllChains` with the full, valid block.

The value `59 seconds` is derived from the fact that we want to avoid timing out, specially for the `GetBlockFromAllChains` method as it leads to a ban at no particular advantage for the attacker. The default timeout is 60 seconds.

© Coinspect 2024

Note that Tari average block time is 120 seconds. So we were able to lock block propagation on all our competing miners for almost the full average block time.

Recommendation

There should be no lock when receiving a block from the network.

Status

Fixed. The code can now process several concurrent blocks from the network as long as they satisfy a `MIN_DIFFICULTY` (see `Status` on **TARI-003**). This prevents an attacker from executing this attack with any serious consequence.

TARI-002

Attackers can prevent block processing

Location

```
base_layer/core/src/base_node/comms_interface/inbound_handlers.rs
```

Description

Anyone can deliberately slow down their peers, forcefully acquiring the *new block lock* of peers by up to a minute. The attack is similar to TARI-001 in nature, but leverages another path which allows an attacker **with no mining power** to execute it, causing victim nodes to be unable to receive new blocks from the network. The attacker node does not get banned or otherwise penalized by the victim node.

As with TARI-001, the attacker's objective is to acquire the receive block lock of the victim's node. In this variant, the attacker simply sends an orphan block with invalid proof of work.

The goal is to trigger the following `if` condition in `InboundNodeCommsHandlers::reconcile_block`:

© Coinspect 2024

```
if header.prev_hash != *current_meta.best_block() {
```

Once we reached this condition, the attacker uses the same strategy as in **TARI-001**: to force the victim to request the full block, they insert an unknown transaction in the invalid block. This triggers a request to the attacker which can arbitrarily halt for up to 60 seconds.

```
let block = self.request_full_block_from_peer(source_peer,
block_hash).await?;
```

At this point, the attacker has succeeded. They need only to set a delay in their response. The attacker can avoid being banned because they are inserting a *true orphan*: proof of work validations are not done on orphans as they are context dependent (see **TARI-003**).

The attacker has an alternative to sending a true orphan. They can simply respond `Trash`, an unexpected API response, after the delay. This is the strategy Coinspect auditors used for their Proof of Concept:

```
            NodeCommsRequest::GetBlockFromAllChains(hash) => {
                warn!("[COIN] [ATTACKER] Victim has requested block.
Wait 55s before giving it.");
                tokio::time::sleep(Duration::from_secs(55)).await;
                let block_hex = hash.to_hex();
                debug!(
                    target: LOG_TARGET,
                    "A peer has requested a block with hash {}",
block_hex
                );
                warn!("[COIN] [ATTACKER] Time has passed. Now send
trash as response");
                return Ok(NodeCommsResponse::Trash("!! TRASH
!!".to_owned()));
            },
```

Recommendation

As with **TARI-001**, the main fix for this issue would be to remove the lock for new blocks.

Furthermore: store only the orphan header and request the full block only when the parent is known.

To prevent trash responses, treat `UnexpectedAPIResponse` errors as malicious and ban peers sending them.

Status

Fixed. See `Status` of **TARI-001**.

TARI-003

Nodes waste resources validating fake orphan blocks

Location

`base_layer/core/src/base_node/comms_interface/inbound_handlers.rs`

Description

An attacker can send a fake orphan block with invalid proof of work. The victim node will perform validations on it, including range proofs, therefore wasting computing resources. Attackers are not penalized for this action.

When receiving a new block, a node will first check if they know the block's parent. If they do not, they will consider it an *orphan*. Only internal consistency validations can be performed on an orphan block, as its relationship with the chain is not known when it is received.

These consistency validations are located in `AggregateBodyInternalConsistencyValidator::validate` and they include

© Coinspect 2024

validations of the range proofs, script execution, covenants and kernel signatures; all potentially expensive computations.

This issue can be combined with **TARI-002**, resulting in a combination of their impacts: the victim's lock is acquired and computation is wasted.

Recommendation

To prevent wasted computations on true orphans, delay **all** orphan validation until their parents are known.

Status

Fixed. The minimum difficulty is now updated according to the last known target.

Previous feedback

This issue was only **Partially Fixed** on commit `63f61ebc2639730060d45ed63d0b839b0b321565`. For that commit, Coinspect gave the following feedback:

Tari implemented a minimum difficulty threshold that all blocks must achieve in order to be processed by the receiver. Nevertheless, this difficulty is constant:

```
    async fn check_min_block_difficulty(&self, new_block: &NewBlock) ->
Result<(), CommsInterfaceError> {
        let constants =
self.consensus_manager.consensus_constants(new_block.header.height);
        let min_difficulty =
constants.min_pow_difficulty(new_block.header.pow.pow_algo);
```

As the network hashrate increases, this minimum difficulty will become increasingly easy to achieve relative to the amount of work present in the network.

To fully mitigate risk, there are several approaches that can be implemented. The minimum difficulty strategy can be kept, but updated the minimum difficulty value according to the last known target and how `LWMA-1` changes. This, plus adding a maximum distance from the tip to the block to keep difficulty values reasonable, would make too expensive for an attacker to exploit the bug only to make a victim waste resources.

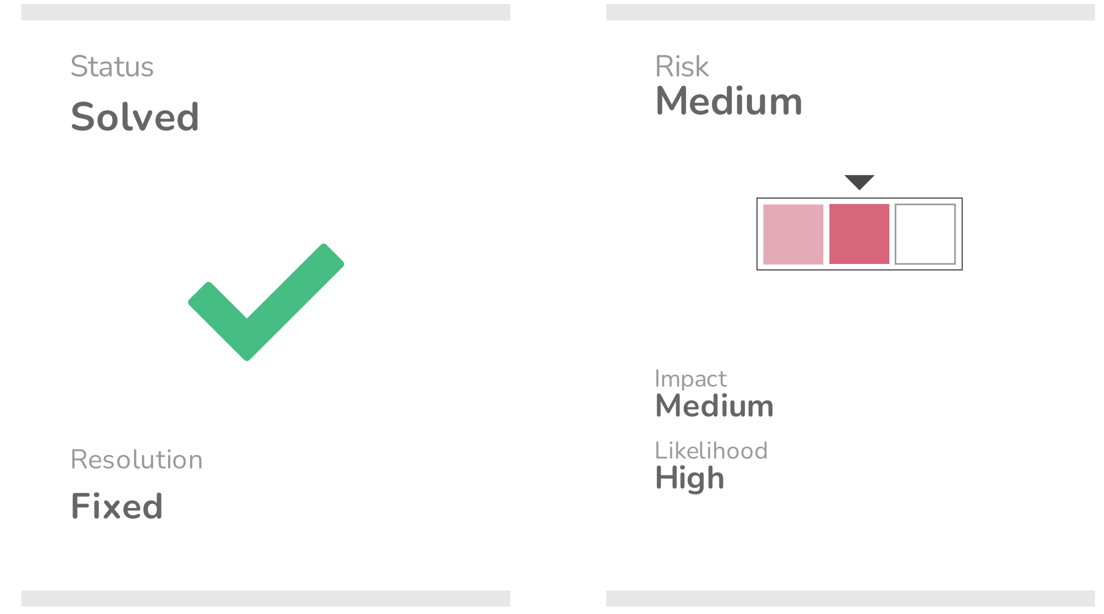
Other strategies are possible:

1. Tari is also considering not receiving orphan blocks at all. This is the strongest protection possible, but the network looses the ability to optimistically receive blocks out of order.
2. Orphans could be validated only when their parents are known and not before. An attacker could send orphan headers that are never connected, but in this case the node would waste no resources validating them until it is absolutely required. This, together with a light `MIN_DIFFICULTY` to prevent spam, also mitigates the risk.

© Coinspect 2024

TARI-004

Attackers can spam mempool at no cost



Location

`base_layer/core/src/mempool/service/inbound_handlers.rs`

Description

Attackers can waste victim's resources and bandwidth by sending a zero-fee transactions to the victim's mempool. The transaction will get validated, added to the mempool and broadcasted.

Even though the transaction will likely not get included in a block as long as there are fee-paying transactions due to priority-policies, the attacker successfully made the victim node process and broadcast a non-paying transaction. It is worth noting that the code has a `MINIMUN_TRANSACTION_FEE`, but it's present as part of the wallet protocol to create a transaction.

```
if fee < Fee::MINIMUM_TRANSACTION_FEE {
    return Err(TPE::ValidationError("Fee is less than the
```

```
minimum".into()));
            }
```

An attacker can forge a transaction that does not respect these limits.

Recommendation

Enforce a minimum fee at the mempool level. This fee does not need to be part of consensus: each operator could decide what minimum fee makes sense for their node. This would be similar to Bitcoin's minimum relay fee.

Status

Fixed. The mempool now checks for a fee first before validating the transaction. It is worth noting that an attacker can still present an invalid transaction that *seems* to pay a fee but is found to be invalid after validation.

The impact of said attack is minimal and lacks a clear incentive, so the risk is negligible.

Previous feedback

This issue was only **Partially Fixed** on commit `63f61ebc2639730060d45ed63d0b839b0b321565`. For that commit, Coinspect gave the following feedback:

The mempool now has logic to prevent storing a transaction if the fee is too low. Nevertheless, this is only checked after validations on the transaction have been made. This leaves the door open for attackers to freely use node resources by spamming it with low-fee but heavy-on-validate transactions. Coinspect recommends making sure node operators are aware of this possibility and mitigate this threat on another layer: most operators should require API keys to access their RPC and invalidate keys that behave badly. Operators that need to keep their RPC fully public can implement strong monitoring for these methods and attempt to block misbehaving users.

© Coinspect 2024

TARI-005

Attackers can mine several blocks at once by abusing ExtraField data

Location

`base_layer/core/src/proof_of_work/monero_rx/helpers.rs`

Description

A miner can mine different blocks at the same height concurrently at no cost by committing to several Tari blocks in a single Monero block. This means that miners must not *choose* a fork and instead can mine on *all* forks available, including private forks. This in turn makes double spending much more feasible. This is somewhat similar to the nothing at stake design issue found on some early proof of stake implementations.

The exploiter of the issue is a merge-miner. Tari allows blocks to be merged-mined along Monero blocks, as described in RFC-0132. For this issue, it is important to keep in mind that a commitment to a Tari block must be put into the `ExtraField` of the coinbase transaction.

The problem lies in the `verify_header` method:

```
fn verify_header(header: &BlockHeader) -> Result<MoneroPowData,
MergeMineError> {
    let monero_data = MoneroPowData::from_header(header)?;
    let expected_merge_mining_hash = header.mining_hash();
    let extra_field =
ExtraField::try_parse(&monero_data.coinbase_tx.prefix.extra)
        .map_err(|_| MergeMineError::DeserializeError("Invalid extra
field".to_string()))?;
    // Check that the Tari MM hash is found in the monero coinbase
transaction
    let is_found = extra_field.0.iter().any(|item| match item {
        SubField::MergeMining(Some(depth), merge_mining_hash) => {
            depth == &VarInt(0) && merge_mining_hash.as_bytes() ==
expected_merge_mining_hash.as_slice()
        },
        _ => false,
    });

    if !is_found {
        return Err(MergeMineError::ValidationError(
            "Expected merge mining tag was not found in Monero coinbase
transaction".to_string(),
        ));
    }

    if !monero_data.is_valid_merkle_root() {
        return Err(MergeMineError::InvalidMerkleRoot);
    }

    Ok(monero_data)
}
```

Note that the `is_found` variable will be true if **any** of the `MergedMining` subfields have depth zero (set by the miner) and the `merge_mining_hash` is equal to the `expected_merged_mining_hash`. This allows a miner puts *several* commitments to Tari blocks in the same Monero block. This means they can claim to have done work on several Tari blocks when they only did work on a single Minero block.

Recommendation

There should be at most one valid `merge_mining_hash` in the Monero block. Alternatively, if several `merge_mining_hash` fields are needed to support Monero blocks that merge mine with other blockchains as well, consider adding an identifier prefix to valid Tari merged mining commitments (eg: `hex("TARI") + hash`) and enforce its uniqueness.

Status

Fixed.

If more than a single merge mining hash is added into the coinbase transaction the `verify_header` function returns an error.

TARI-006

Attackers can mine several blocks at once by using multiple Monero coinbase transactions

Location

base_layer/core/src/proof_of_work/monero_rx/helpers.rs

Description

A miner can mine different blocks at the same height concurrently at no cost by creating invalid Monero blocks that have several coinbase transactions in them. This issue has the same exact impact as **TARI-005**.

Consider the verify_header() method. The last call is made to is_valid_merkle_root().

```
fn verify_header(header: &BlockHeader) -> Result<MoneroPowData,
MergeMineError> {
    let monero_data = MoneroPowData::from_header(header)?;
    let expected_merge_mining_hash = header.mining_hash();
    let extra_field =
ExtraField::try_parse(&monero_data.coinbase_tx.prefix.extra)
```

© Coinspect 2024

```
        .map_err(|_| MergeMineError::DeserializeError("Invalid extra
field".to_string()))?;
    // Check that the Tari MM hash is found in the monero coinbase
transaction
    let is_found = extra_field.0.iter().any(|item| match item {
        SubField::MergeMining(Some(depth), merge_mining_hash) => {
            depth == &VarInt(0) && merge_mining_hash.as_bytes() ==
expected_merge_mining_hash.as_slice()
        },
        _ => false,
    });

    if !is_found {
        return Err(MergeMineError::ValidationError(
            "Expected merge mining tag was not found in Monero coinbase
transaction".to_string(),
        ));
    }

    if !monero_data.is_valid_merkle_root() {
        return Err(MergeMineError::InvalidMerkleRoot);
    }

    Ok(monero_data)
}
```

The `is_valid_merkle_root()` method will check that the coinbase is included in the block by calling `calculate_root()`, which in turns uses information found on the Monero block to reconstruct the root.

```
    pub fn is_valid_merkle_root(&self) -> bool {
        let coinbase_hash = self.coinbase_tx.hash();
        let merkle_root =
self.coinbase_merkle_proof.calculate_root(&coinbase_hash);
        self.merkle_root == merkle_root
    }
```

```
    pub fn calculate_root(&self, hash: &Hash) -> Hash {
        if self.depth == 0 {
            return self.branch[0];
        }

        let mut root = *hash;
        for d in 0..self.depth {
            if (self.path_bitmap >> (self.depth - d - 1)) & 1 > 0 {
                root = cn_fast_hash2(&self.branch[d as usize], &root);
            } else {
                root = cn_fast_hash2(&root, &self.branch[d as usize]);
            }
        }

        root
    }
```

© Coinspect 2024

The issue is that an attacker is able to create two versions of the same Monero block which point to different transactions as the `coinbase` via the `path_bitmap`. By doing this, an attacker can commit to several Tari blocks while working on a single Monero block by putting each Tari block in a different Monero transaction and then pointing the coinbase to the one they wish to publish.

These blocks are invalid in the Monero network as the consensus rules prevent a block from having more than a single coinbase. But Tari does not verify Monero consensus rules, so the block is considered valid for the purposes of verifying proof of work.

Recommendation

Make the path where a coinbase transaction should be constant. Monero creates the the transaction root using the coinbase transaction as the first item. Using that as a constant would incur in no incompatibility with Monero while mitigating this issue.

Status

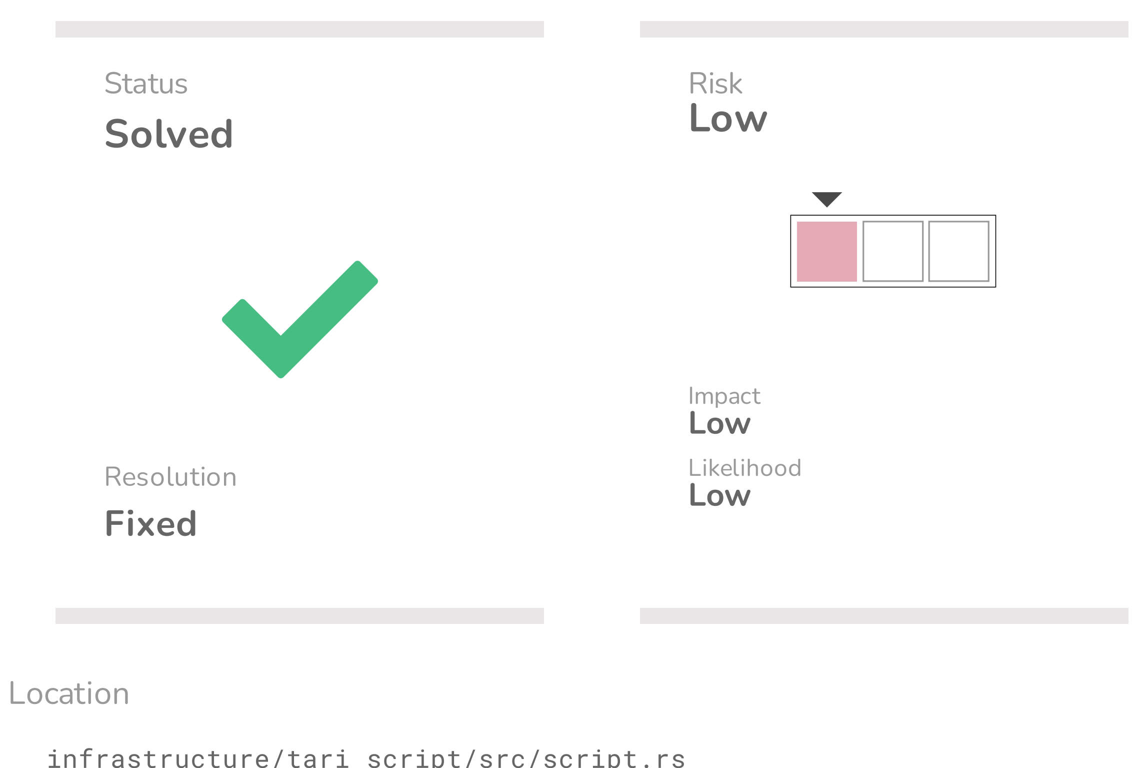
Fixed.

Miners can no longer provide the `path_bitmap` value, and only the leftmost Monero transaction is accepted.

© Coinspect 2024

TARI-007

Malicious script makes validators waste computations



Location

`infrastructure/tari_script/src/script.rs`

Description

An attacker can force the `CHECK_MULTISIG` family of opcodes to be always worst case by ordering the public keys and their signatures in a way that maximizes the amount of iterations the nested `for-loop` must do in order to verify the script.

By putting the public key of the signers and their signatures in reverse order (ie: the signer for the first signature is the last public key, the second is the second to last, and so on), an attacker makes all validators waste resources. This is not reflected in the protocol fees. It is worth keeping in mind that verifying signatures is an expensive process for validators.

```
        for s in &signatures {
            for (i, pk) in public_keys.iter().enumerate() {
```

```
                if !sig_set.contains(s) && !key_signed[i] &&
s.verify_challenge(pk, &message) {
                    // This prevents Alice creating 2 different sigs
against her public key
                    key_signed[i] = true;
                    sig_set.insert(s);
                    agg_pub_key = agg_pub_key + pk;
                    break;
                }
            }
            // Make sure the signature matched a public key
            if !sig_set.contains(s) {
                return Ok(None);
            }
        }
```

The impact of this issue is currently low, as the upper bound is around ~500 iterations of the loop.

Recommendation

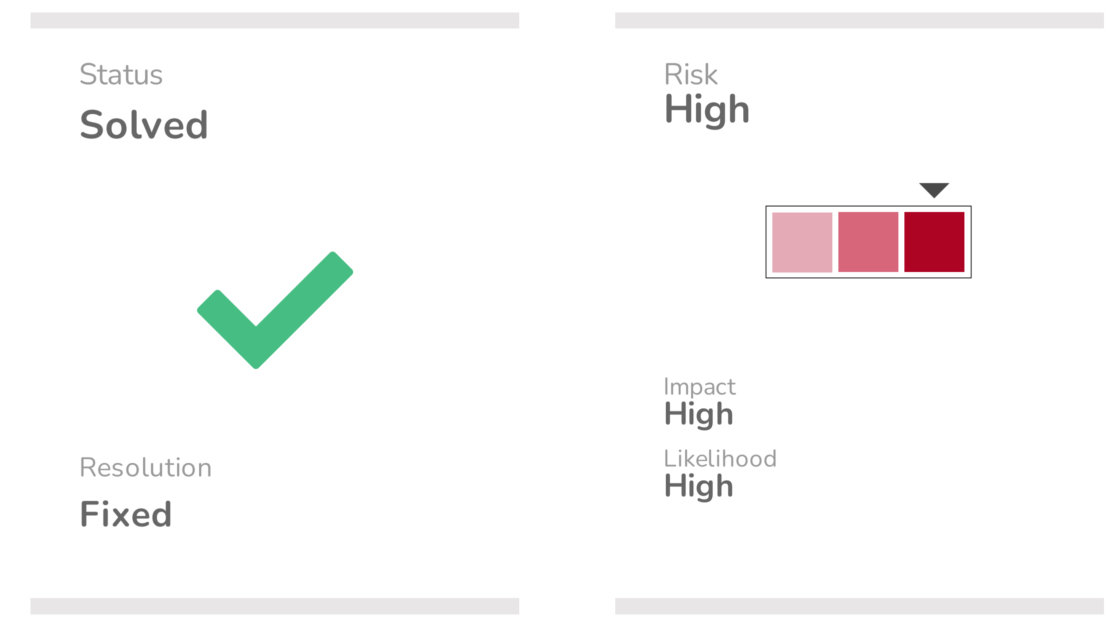
Make `signatures` and `public_keys` be ordered, just like Bitcoin does.

Status

Fixed. The code now makes sure the `public_keys` iterator does not backtrack, enforcing that signatures and public keys are presented in order.

TARI-008

Attackers can create the same block with different hashes



Location

`base_layer/core/src/proof_of_work/monero_rx/helpers.rs`

Description

Multiple Tari blocks can be created from a single valid proof of work. The blocks are identical for most purposes, but have a different hash. Because of this, an attacker can create several different forks, all with the same information but being built on different hashes. Once these malicious blocks are spread on the network, the attacker can use its hashing power to double spend much more easily, as the hashing power of the network is divided.

The issue exploit a malleability bug in the merkle proof used for merge mined blocks. The difference with TARI-005 and TARI-006 is that in this case multiple Tari blocks that share the same mining hash are created, instead of using different mining hash values. Additionally, any user in the network can modify a valid Tari block mined with RandomX to create new valid blocks. Produced blocks are, in

essence, the same blocks but with different hash in Tari due to a difference in the proof of work data.

The `MerkleProof` data structure has a `path_bitmap` which is used for deciding the path used for the merkle inclusion.

```
pub struct MerkleProof {
    branch: Vec<Hash>,
    depth: u16,
    path_bitmap: u32,
}
```

This path is used in the `calculate_root` method for generating the root hash:

```
    pub fn calculate_root(&self, hash: &Hash) -> Hash {
        if self.depth == 0 {
            return self.branch[0];
        }

        let mut root = *hash;
        for d in 0..self.depth {
            if (self.path_bitmap >> (self.depth - d - 1)) & 1 > 0 {
                root = cn_fast_hash2(&self.branch[d as usize], &root);
            } else {
                root = cn_fast_hash2(&root, &self.branch[d as usize]);
            }
        }

        root
    }
```

The issue lies in that only the bits used for walking the path are validated by the Tari node. Once `d` reaches `self.depth`, the remaining bits are completely ignored.

An attacker can modify the bits to generate new Tari blocks, as these bits are included in the encoding of the block when calculating the block's hash, as seen in `base_layer/core/src/blocks/block_header.rs`:

```
    pub fn hash(&self) -> FixedHash {
        DomainSeparatedConsensusHasher::
<BlocksHashDomain>::new("block_header")
            .chain(&self.mining_hash())
            .chain(&self.pow)
            .chain(&self.nonce)
            .finalize()
            .into()
    }
```

Recommendation

© Coinspect 2024

Change the calculation of the root to use a list of hashes provided by the miner instead of a `depth` and a `bitmap`.

By asking for the hashes of the siblings the data necessary to construct the block is provided. Any extra data that is not used would lead to the root being different.

Alternatively, make sure that the entirety of the `branch` and `bitmap` fields are used before exiting the method.

Status

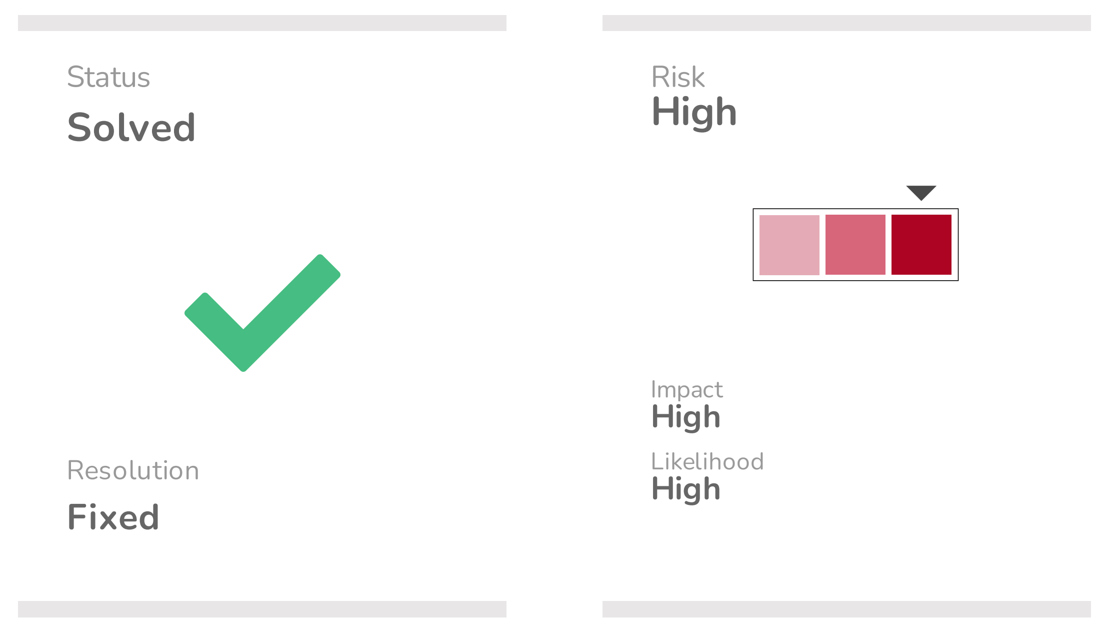
Fixed

The proof of inclusion in the Monero block changed, and now only the path is provided with no `bitmap`, removing malleability issues.

© Coinspect 2024

TARI-009

Attackers can halt long-syncing by inflating block headers



Location

```
base_layer/core/src/proof_of_work/monero_rx/merkle_tree.rs
```

Description

An attacker can flood the network and stop long-syncing by inflating block headers. The strategy used is exactly the same one as described in **TARI-008**: an attacker simply appends data to the `branch` or `bitmap` of the `ProofOfWork` data, which makes the header grow in size.

An attacker would not only be able to flood the network with these inflated blocks, but also would break long syncing: they can create several blocks which weight under the network size limit of 4Mb, but when requested via chunks, exceed this value. Because the blocks are valid and under the limit, already-synced clients which are requesting these blocks as soon as they are available would not encounter this problem.

Nevertheless, new nodes (or nodes that are behind the tip and need to start long syncing) would never be able to receive the blocks, as each chunk would exceed the 4Mb limit.

During the long sync there are multiple points where several block headers are requested in a single message. The first of which is in the `find_chain_split()` method

```
            let request = FindChainSplitRequest {
                block_hashes: block_hashes.clone().iter().map(|v|
v.to_vec()).collect(),
                header_count,
            };

            let resp = match client.find_chain_split(request).await {
                Ok(r) => r,
                Err(RpcError::RequestFailed(err)) if
err.as_status_code().is_not_found() => {
                    // This round we sent less hashes than the max, so
the next round will not have any more hashes to
                    // send. Exit early in this case.
                    if block_hashes.len() < NUM_CHAIN_SPLIT_HEADERS {
                        return
Err(BlockHeaderSyncError::ChainSplitNotFound(peer.clone()));
                    }
                    // Chain split not found, let's go further back
                    offset = NUM_CHAIN_SPLIT_HEADERS * iter_count;
                    continue;
                },
                Err(err) => {
                    return Err(err.into());
                },
            };
```

The `find_chain_split` gRPC request receives up to 1000 block headers. If headers are big enough, this message will surpass the message size limit and will never be received by the syncing node.

Recommendation

Fixing issue **TARI-008** would also mitigate this one, as an attacker would not be able to arbitrarily append data to block headers.

Consider also making implicit network limits (such as the one stating the 1000 headers weight less than 4Mb) an explicit consensus rule.

Status

Fixed

It is not possible to append arbitrary data to block headers anymore.

TARI-010

Attackers can halt long-sync by appending trailing data to Proof of Work buffer

Location

```
base_layer/core/src/proof_of_work/monero_rx/pow_data.rs
base_layer/core/src/proof_of_work/monero_rx/merkle_tree.rs
```

Description

An attacker can flood the network and stop long-syncing by inflating block headers. The impact and requirements of this issue are exactly the same as those of **TARI-009**. The difference is that even if **TARI-009** is mitigated, the attacker can still use the `MoneroPowData` field, which is sent on the network as a raw `Vec<u8>`, to achieve the same result.

The `MoneroPowData` is encoded and decoded after being received by `Borsh`, which reads certain fields. The attacker can simply put trailing data on the buffer, which will be ignored by `Borsh`, but still be appended to the block as a raw buffer. This buffer is part of the block hash calculations:

```
pub struct ProofOfWork {
    pub pow_algo: PowAlgorithm,
    pub pow_data: Vec<u8>,
}
```

```
    pub fn hash(&self) -> FixedHash {
        DomainSeparatedConsensusHasher::
<BlocksHashDomain>::new("block_header")
            .chain(&self.mining_hash())
            .chain(&self.pow)
            .chain(&self.nonce)
            .finalize()
            .into()
    }
```

The issue appears in two different steps of the deserialization logic of the `RandomX` proof of work. Note that in both instances some data from the `buf` is being deserialized, but any trailing data in the `buf` is ignored.

When the `MerkleProof` struct is being deserialized:

```
impl BorshDeserialize for MerkleProof {
    fn deserialize(buf: &mut &[u8]) -> io::Result<Self> {
        let len = buf.read_varint()?;
        let mut branch = Vec::with_capacity(len);
        for _ in 0..len {
            branch.push(
                Hash::consensus_decode(buf)
                    .map_err(|err|
io::Error::new(io::ErrorKind::InvalidData, err.to_string()))?,
            );
        }
        let depth = BorshDeserialize::deserialize(buf)?;
        let path_bitmap = BorshDeserialize::deserialize(buf)?;
        Ok(Self {
            branch,
            depth,
            path_bitmap,
        })
    }
}
```

And when the MoneroPowData is being deserialized:

```
impl BorshDeserialize for MoneroPowData {
    fn deserialize(buf: &mut &[u8]) -> io::Result<Self> {
        let header = monero::BlockHeader::consensus_decode(buf)
            .map_err(|e| io::Error::new(io::ErrorKind::InvalidData,
e.to_string()))?;
        let randomx_key = BorshDeserialize::deserialize(buf)?;
        let transaction_count = BorshDeserialize::deserialize(buf)?;
        let merkle_root = monero::Hash::consensus_decode(buf)
            .map_err(|e| io::Error::new(io::ErrorKind::InvalidData,
```

© Coinspect 2024

```
e.to_string()))?;
        let coinbase_merkle_proof =
BorshDeserialize::deserialize(buf)?;
        let coinbase_tx = monero::Transaction::consensus_decode(buf)
            .map_err(|e| io::Error::new(io::ErrorKind::InvalidData,
e.to_string()))?;
        Ok(Self {
            header,
            randomx_key,
            transaction_count,
            merkle_root,
            coinbase_merkle_proof,
            coinbase_tx,
        })
    }
}
```

Recommendation

Make sure there is no trailing data after deserializing the buffers.

Status

Fixed

It is not possible to append data on the encoded structs anymore.

TARI-011

Attackers can crash nodes by manipulating proof of work data

Location

`base_layer/core/src/proof_of_work/monero_rx/merkle_tree.rs`

Description

Attackers can remotely trigger a panic with malicious messages.

The vulnerability lies in the RandomX proof of work validation process. Attackers can trigger an out of index access in the `branch` vector of the `MerkleProof` struct.

During the `calculate_root` procedure the index used for accessing the vector is limited by the `depth` variable which can be larger than the `branch` length.

```
    /// Calculates the merkle root hash from the provide Monero hash
    pub fn calculate_root(&self, hash: &Hash) -> Hash {
        if self.depth == 0 {
            return self.branch[0];
        }
```

```
        let mut root = *hash;
        for d in 0..self.depth {
            if (self.path_bitmap >> (self.depth - d - 1)) & 1 > 0 {
                root = cn_fast_hash2(&self.branch[d as usize], &root);
            } else {
                root = cn_fast_hash2(&root, &self.branch[d as usize]);
            }
        }

        root
    }
```

The accesses at `&self.branch[d as usize]` may lay outside the array bounds.

Attackers can forge headers with a proof of work whose depth excess the size of the branch array triggering the exception. The attack does not require hashing power nor any special privilege other than being connected to the victim's node.

Recommendation

Check array bounds before accessing it.

Status

Fixed

The algorithm has been changed and now it is not possible to access out of bound items.

© Coinspect 2024

TARI-012

Proof of work can be copy-pasted

Location

`base_layer/core/src/proof_of_work/monero_rx/merkle_tree.rs`

Description

Attackers can grab any Monero block that satisfies Tari difficulty and use it to create infinite Tari blocks. This is because attackers are able to skip the merkle root calculation used to make sure that the Monero commitment to a Tari block is protected by proof of work.

Attackers are able to do this by providing a `depth` of zero in the `MerkleProof` data along with a `branch` which has the merkle root on its first position. It is important to note that the `MerkleProof` is not protected by Monero proof-of-work and attackers are free to modify it.

By doing this, attackers can reuse existing blocks to commit to new ones by changing the first element of the `branch` to be equal to the merkle root and forging a `coinbase` that commits to a new Tari block.

Attackers need not have any mining power. They are, effectively, reusing the proof of work of a block to produce several.

The vulnerability can be found in the `calculate_root` method, which trusts the `MerkleProof` data when `depth == 0`, instead of using the `hash` which is derived from the `coinbase`.

```
    /// Calculates the merkle root hash from the provide Monero hash
    pub fn calculate_root(&self, hash: &Hash) -> Hash {
        if self.depth == 0 {
            return self.branch[0];
        }

        let mut root = *hash;
        for d in 0..self.depth {
            if (self.path_bitmap >> (self.depth - d - 1)) & 1 > 0 {
                root = cn_fast_hash2(&self.branch[d as usize], &root);
            } else {
                root = cn_fast_hash2(&root, &self.branch[d as usize]);
            }
        }

        root
    }
```

Because the `coinbase` is not used, attackers are able to forge its `ExtraField`, so the new blocks would have a `coinbase` with the appropriate commitment, albeit one not validated by proof of work.

```
    let is_found = extra_field.0.iter().any(|item| match item {
        SubField::MergeMining(Some(depth), merge_mining_hash) => {
            depth == &VarInt(0) && merge_mining_hash.as_bytes() ==
expected_merge_mining_hash.as_slice()
        },
        _ => false,
    });
```

Recommendation

Return the `hash` value in the `depth == 0` case.

Status

Fixed.

When the `depth` is 0 the coinbase hash is returned.

© Coinspect 2024

TARI-013

Attackers can force sync on a peer eternally

Location

`base_layer/core/src/base_node/sync/header_sync/synchronizer.rs`

Description

Attackers can lie about the difficulty and height they have achieved, and send already known headers to force a peer to sync. Therefore, the attacker can waste victim's resources by making the peer re-validate headers and blocks. The victim would also deprioritize syncing from benign peers. The attacker is not penalized.

The sync process starts in the `Listening` phase, where a peer will request `ChainMetadata`. If a peer announces that they have a higher total difficulty and are at least a couple of blocks ahead, the node will try to sync from them:

```
            (
                Listening(_),
                FallenBehind(Lagging {
                    local: local_metadata,
                    sync_peers,
                    ..
```

```
                  }),
              ) => {
                  db.set_disable_add_block_flag();
                  HeaderSync(HeaderSyncState::new(sync_peers,
local_metadata))
              },
```

It then proceeds with the HeaderSync, where the node will try to determine_sync_status to find the point from where it should sync. To determine this, the node instructs the peer to *find a chain split*:

```
        let (resp, block_hashes, steps_back) = self
            .find_chain_split(sync_peer.node_id(), client,
NUM_INITIAL_HEADERS_TO_REQUEST as u64)
            .await?;
```

The node expects the peer to send the block_hashes that are missing. If all goes OK, the node will set its sync status to Lagging, which should proceed to sync the rest of the headers in the synchronize_headers method.

While the synchronize_headers method has some checks to validate the peer's declared difficulty, these are skipped if the peer sent less than the expected amount of headers:

```
        if pending_len < NUM_INITIAL_HEADERS_TO_REQUEST {
            // Peer returned less than the number of requested headers.
This indicates that we have all the available
            // headers.
            debug!(target: LOG_TARGET, "No further headers to
download");

            if !has_better_pow {
                return
Err(BlockHeaderSyncError::PeerSentInaccurateChainMetadata {
                    claimed:
sync_peer.claimed_chain_metadata().accumulated_difficulty(),
                    actual: Some(total_accumulated_difficulty),
                    local: split_info
                        .local_tip_header
                        .accumulated_data()
                        .total_accumulated_difficulty,
                });
            }

            return Ok(());
        }
```

Note the code will consider has_better_pow as true because the alternative chain proposed by the peer is *exactly* the same as the one stored locally, so the accumulated difficulty is exactly equal and the check is done with a *less or equal* condition:

© Coinspect 2024

```
    fn pending_chain_has_higher_pow(&self, current_tip: &ChainHeader) -
> bool {
        let chain_headers = self.header_validator.valid_headers();
        if chain_headers.is_empty() {
            return false;
        }

        // Check that the remote tip is stronger than the local tip
        let proposed_tip = chain_headers.last().unwrap();
        self.header_validator.compare_chains(current_tip,
proposed_tip).is_le()
    }
```

As synchronize_headers returns OK(()), the synchronization process continues. So, all in all, an attacker has to:

1. Lie about their difficulty to a peer
2. Send an amount of headers small enough so that NUM_INITIAL_HEADERS_TO_REQUEST is bigger than pending, but big enough to reach the tip of the local chain

This will place the victim in a state in which it constantly tries to sync from the attacker, making it waste resources and time. This vulnerability also allows other sync-related vulnerabilities to be triggered arbitrarily, as attackers do not have to find a node that is lagging behind.

Recommendation

Ban peers who lie about their difficulty. Do not continue synchronization process if the peer has not provided a better chain.

Status

This particular vector has been corrected by making sure that nodes present a better proof of work. Nevertheless, a similar issue which uses another strategy to trigger the problem can be found in **TARI-039**.

TARI-014

Miner can ban competition

Location

/base_layer/core/src/base_node/sync/block_sync/synchronizer.rs

Description

A miner can ban blocks from the competition to get a higher percentage of hashing power on the network. To do this, the miner leverages the **bad block** system, which bans known bad block hashes from the network and the fact that block headers have a field `validator_node_mr`, that is not protected by proof of work nor considered when hashing the header.

Although bad blocks are added only on **block sync**, a miner can harness **TARI-013** to force synchronization on a victim.

```
            let block = match res {
                Ok(block) => block,
                Err(err @ ValidationError::BadBlockFound { .. }) |
                Err(err @ ValidationError::FatalStorageError(_)) |
                Err(err @ ValidationError::AsyncTaskFailed(_)) |
                Err(err @ ValidationError::CustomError(_)) => return
```

© Coinspect 2024

```
Err(err.into()),
                Err(err) => {
                    // Add to bad blocks
                    if let Err(err) = self
                        .db
                        .write_transaction()
                        .delete_orphan(header_hash)
                        .insert_bad_block(header_hash, current_height)
                        .commit()
                        .await
                    {
                        error!(target: LOG_TARGET, "Failed to insert
bad block: {}", err);
                    }
                    return Err(err.into());
                },
            };
```

The attacker has to be well connected in the network to have a relatively high chance of seeing the competing blocks before victims. The attacker also has to be able to spin disposable nodes on the network, as peers will ban them once they execute the attack.

Once they are, they:

1. Modify the `validator_node_mr` field of the block. Note that this does not alt the block hash nor invalidates the proof of work.
2. Force a node to sync using **TARI-013**
3. Provide the modified block when syncing.
4. Block validation will fail due to the invalid `validator_node_mr`.
5. The block hash will be marked as **bad** and will not be accepted any longer.

Recommendation

Do not have fields on blocks that are susceptible to malleability and not protected by proof of work. In particular, the `validator_node_mr` should be part of the `mining hash` of the header.

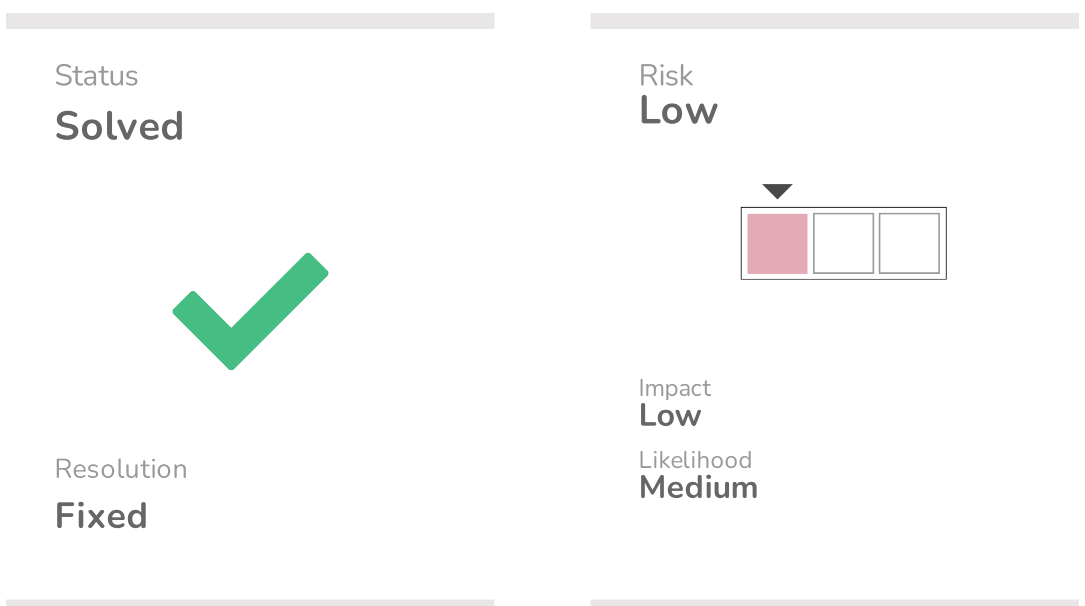
Status

The `validator_node_mr` is now part of the mining hash, making it protected by proof of work.

© Coinspect 2024

TARI-015

Known bad blocks are validated



Description

The **bad block** system only adds bad blocks when received via syncing. This makes it possible for attackers to force nodes to waste resources by repeatedly sending known bad blocks.

Although attacker's nodes will be banned, spinning up a node with a different `Node ID` is possible for dedicated attackers.

Recommendation

Make sure bad blocks are marked as such no matter how they are detected. It is important to note that this should be implemented only after the block header malleability issue described in **TARI-014** is addressed, as if it is not, it would only make **TARI-014** easier for attackers.

© Coinspect 2024

Status

The bad block system is more robust now and blocks are banned when appropriate.

TARI-016

Attackers can halt synchronization

Location

```
base_layer/core/src/base_node/sync/header_sync/synchronizer.rs
```

Description

An attacker can send the same header indefinitely to cause a client to become stuck, thereby preventing it from syncing from other peers. In combination with **TARI-013**, this can be triggered arbitrarily.

The vulnerable path is found in the header sync process, which ignores a header if it is already found on the database and waits for a new one on the gRPC stream:

```
            if let Some(h) = existing_header {
                warn!(
                    target: LOG_TARGET,
                    "Received header #{} `{}` that we already have.
Ignoring",
                    h.height,
                    h.hash().to_hex()
                );
```

```
                continue;
            }
```

An attacker can send the same known header forever, keeping the victim on that loop. As long as the header keeps sending messages at certain rates, timeouts are not triggered.

Recommendation

A well-behaved peer should not send known headers on sync. Cut communications when a peer sends a known-header.

Status

Fixed by returning error when a repeated block is provided.

© Coinspect 2024

TARI-017

Attackers can avoid ban on horizon sync

Location

```
base_layer/core/src/base_node/state_machine_service/states/horizon_sta
te_sync.rs
```

Description

An attacker can send invalid data and avoid getting banned because of it, consequently wasting the victim's resources. Victims must be running horizon sync.

Horizon sync logic is contained in the `horizon_state_sync`. The entry point for the process is the `next_step` function, which calls `synchronize`:

```
        match horizon_sync.synchronize().await {
            Ok(()) => {
                info!(target: LOG_TARGET, "Horizon state has
synchronized.");
                StateEvent::HorizonStateSynchronized
            },
            Err(err) => {
```

© Coinspect 2024

```
                let _ignore =
shared.status_event_sender.send(StatusInfo {
                    bootstrapped,
                    state_info:
StateInfo::SyncFailed("HorizonSyncFailed".to_string()),
                    randomx_vm_cnt,
                    randomx_vm_flags,
                });
                warn!(target: LOG_TARGET, "Synchronizing horizon state
has failed. {}", err);
                StateEvent::HorizonStateSyncFailure
            },
```

The issue is that no inner methods, including `next_step` or `synchronize`, ban a peer when they send invalid data. This, in conjunction with **TARI-013**, means that attackers can continuously force a sync on horizon-syncing peers and send them trash responses with no consequences for them.

Recommendation

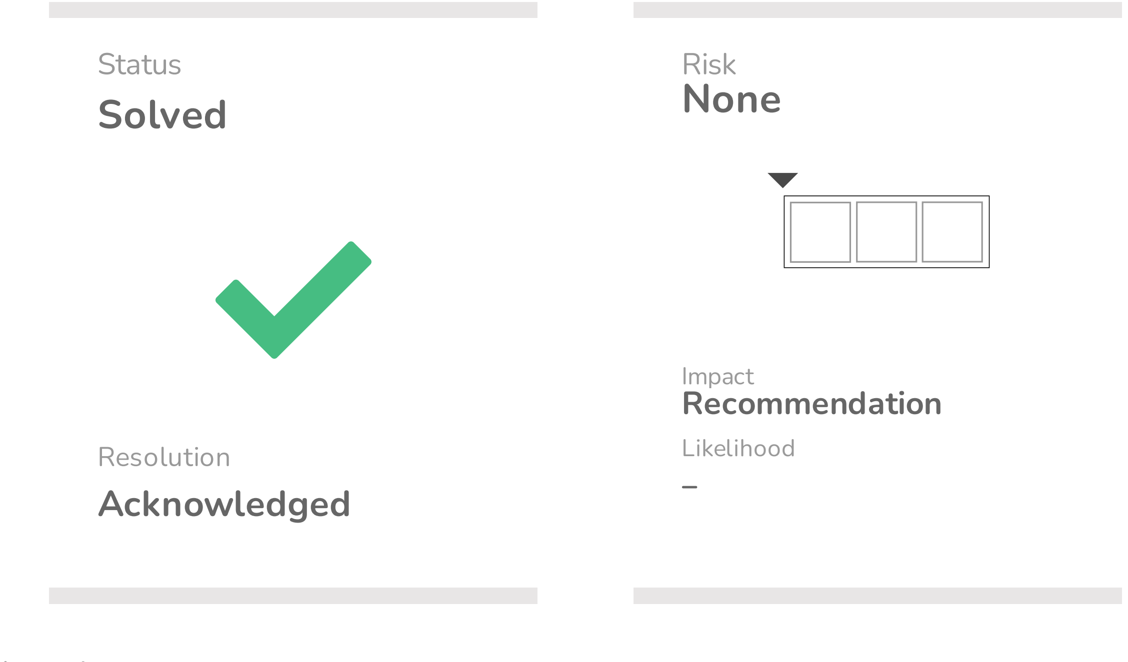
Ban nodes that send invalid data on Horizon Sync, just like nodes are banned in Block Sync.

Status

The misbehaving nodes are now banned.

TARI-018

Consensus is dependent on platform



Location

`base_layer/mmr/src/common.rs`

Description

Merkle mountain range maximum size is different on different platforms. In particular, the creation of `new` MMRs depends on the result of the `checked_n_leaves` method, which returns different values depending on the platform:

```
pub fn checked_n_leaves(size: usize) -> Option<usize> {
    if size == 0 {
        return Some(0);
    }
    if size == usize::MAX {
        return None;
    }
```

As `usize` is platform dependent, the comparison with `usize::MAX` will be different for equal values of `size` on different platforms. A value of `2**32-1` would return `None` for users on a 32-bit platform while otherwise work and return the leave size for users of 64-bit platforms.

The likelihood of this issue is considered extremely low, as it would need a tree with at least `2**32-1` leaves. Theoretically, it is possible to have a tree of *exactly* this size because insertion checks assert only that the position *fits* into an `u32`, which `2**32-1` does. For an example, see the insertion in `HorizonStateSync`:

```
txn.insert_pruned_output_via_horizon_sync(
    utxo.hash.try_into()?,
    *current_header.hash(),
    current_header.height(),
    u32::try_from(mmr_position)?,
    current_header.timestamp(),
);
```

Similarly, the `find_peaks`, `family_branch` and `peak_map_height` return different results depending on the amount of leading zeros in the `pos` argument:

```
pub fn peak_map_height(mut pos: usize) -> (usize, usize) {
    if pos == 0 {
        return (0, 0);
    }
    let mut peak_size = ALL_ONES >> pos.leading_zeros();
```

As the `pos` argument is of type `usize`, its amount of leading zeros is platform dependent. Nevertheless, the analysis is not as straightforward, as the differences would only be observable for values for which `usize` would overflow anyway.

Recommendation

Even though the attacks described here are theoretical, it is recommended that consensus critical calculations are always performed on a fixed-sized type to avoid any kind of hard fork.

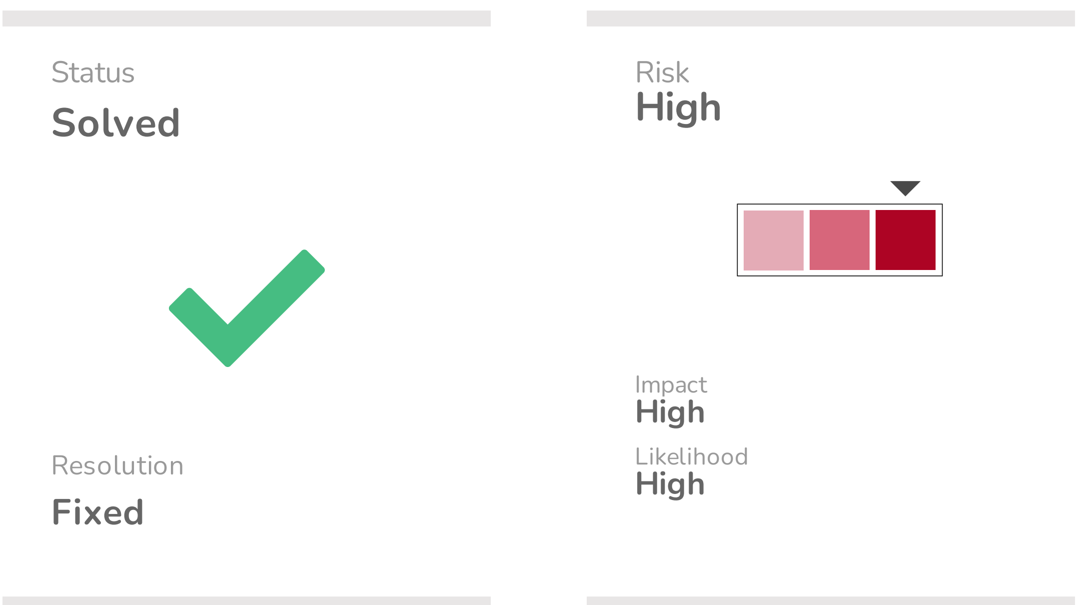
Status

While Tari has removed the parts of the code mentioned in this issue in favor of another merkle mountain range implementation, the team has also stated that:

```
Default Tari applications only support 64bit only and will block when
running on 32 bit applications
```

TARI-019

Attackers can crash a node by forcing over-allocation



Description

Attackers can crash the node by sending crafted serialized covenants, merkle trees, scripts or execution stacks. This is because the `BorshDeserialize::deserialize` trusts the data to announce its `len` honestly and allocates that amount of bytes in a vector.

```
    fn deserialize(buf: &mut &[u8]) -> io::Result<Self> {
        let mut len = buf.read_varint()?;
        let mut data = Vec::with_capacity(len);
```

If an attacker crafts data such that the `varint` in its first position is big enough, the node will crash is it cannot allocate enough bytes.

Attackers can take advantage of several paths where data is deserialized. For example:

- They can create a covenant with an evil `ARG_TARI_SCRIPT`, triggering the panic from `CovenantArg::read_from`.
- They can craft a malicious merkle tree and put it in a monero merkle proof to make `MoneroPowData::deserialize` method panic.
- They can create `TransactionOutput` or `TransactionInput` protobuf messages with crafted covenants to make their `try_from` methods panic.

Note that this list is not meant to be exhaustive.

Recommendation

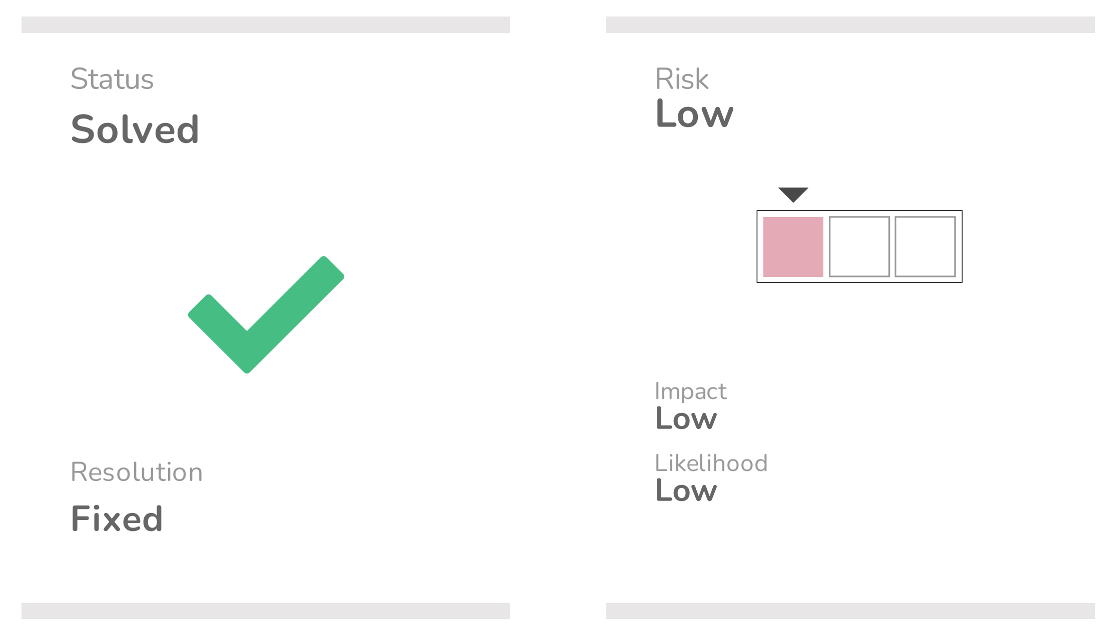
Do not trust data to report its correct length. Set maximums to the data length. After parsing, consider checking that the reported size matches and actual size and take punitive action if not.

Status

The `varint` is now read and checked against a maximum before allocating memory.

TARI-020

Default output selection criteria can be abused to increase the sender's fee



Location

`.../wallet/src/output_manager_service/storage/sqlite_db/output_sql.rs`

Description

The output selection mechanism can be abused to increase the sender's transaction fee in scenarios where the adversary can control the amount to be transferred.

The default output selection mechanism can use two modes depending on the amount to be transferred. If the amount is larger than the maximum output value, the order will be descendant. Otherwise, it will be ascendant, meaning it will select smaller outputs first.

```
UtxoSelectionOrdering::SmallestFirst =>
query.then_order_by(outputs::value.asc()),
UtxoSelectionOrdering::LargestFirst =>
```

© Coinspect 2024

```
query.then_order_by(outputs::value.desc()),
UtxoSelectionOrdering::Default => {
    // NOTE: keeping filtering by `script_lock_height` and `maturity`
for all modes
    // lets get the max value for all utxos
    let max: Option<i64> = outputs::table
        .filter(outputs::status.eq(OutputStatus::Unspent as i32))
        .filter(outputs::script_lock_height.le(i64_tip_height))
        .filter(outputs::maturity.le(i64_tip_height))
        .order(outputs::value.desc())
        .select(outputs::value)
        .first(conn)
        .optional()?;

    match max {
        // Want to reduce the number of inputs to reduce fees
        Some(max) if amount > max as u64 =>
query.then_order_by(outputs::value.desc()),

        // Use the smaller utxos to make up this transaction.
        _ => query.then_order_by(outputs::value.asc()),
    }
},
```

Therefore, an adversary able to control the amount to be transferred can force the sender to use dust values to increase the transaction fee. The dust input values to be used by the sender could be previously sent by the adversary.

Even though the wallet admits configuring a flag to drop the transaction in case the fees exceed the amount to be transferred, it can still be abused to spend an amount of fees slightly lower than the value to be transferred.

Recommendation

Ignore dust amounts in the output selection mechanism.

Also, consider implementing algorithms such as Branch and Bound which minimize the creation of change outputs which can be of dust amounts, by trying to find the exact match for transaction amounts in the UTXO set.

Status

The code now ignores amounts below a user-configurable `dust_amount`.

TARI-021

Attackers can crash nodes by broadcasting JOIN messages

Location

`comms/dht/src/inbound/dht_handler/task.rs`

Description

An attacker can send crafted `Join` messages which crashes the node with an Out Of Memory error. The crash can be generated in less than three minutes in a system with 32GB of RAM available. The attacker is not banned or otherwise impacted.

The attacker has to send `JoinMessages` which contain many addresses, although taking into account the `MAX_FRAME_SIZE` limit for DHT messages. In tests, auditors successfully crashed a victim node with messages with `28801` valid multiaddresses.

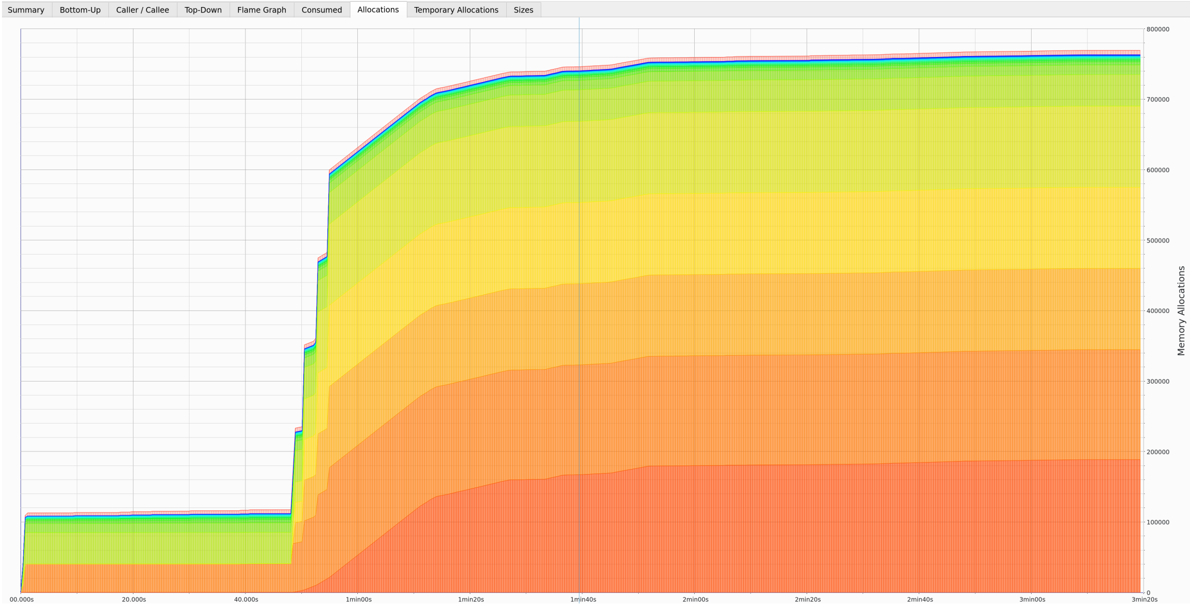
After processing a few messages (six in auditor's tests), the victim requires more memory than available in the system and crashes.

© Coinspect 2024

Heapstack was used as a profiler and reported this memory consumption after the process finished:

```
total runtime: 199.44s.
calls to allocation functions: 769441 (3858/s)
temporary memory allocations: 43460 (217/s)
peak heap memory consumption: 43.54G
peak RSS (including heaptrack overhead): 30.67G
total memory leaked: 43.54G
```

The amount of allocations being requested can also be seen in the graph produced by the same profiler:



It is likely the main cause is the quadratic memory requirement for `addresses` in the `handle_join` method. Valid `MultiAddresses` are `cloned` into the `peer_identity_claim` struct and then both the original `addresses` vector and the one in `peer_identity_claim` are passed to `Peer::new`.

```
        let peer_identity_claim =
PeerIdentityClaim::new(addresses.clone(), features, identity_signature,
None);
        let new_peer = Peer::new(
            authenticated_pk,
            node_id.clone(),

MultiaddressesWithStats::from_addresses_with_source(addresses,
&PeerAddressSource::FromJoinMessage {
                peer_identity_claim,
            }),
            PeerFlags::empty(),
            features,
            vec![],
            String::new(),
        );
```

The `MultiAddressesWithStats::from_addresses_with_source` then `clones` again the `peer_identity_claim` for *each* address:

```
        let mut addresses_with_stats =
Vec::with_capacity(addresses.len());
        for address in addresses {
            addresses_with_stats.push(MultiaddrWithStats::new(address,
source.clone()));
        }
```

This results in quadratic memory requirements. When predicting the memory usage with these rationale it matches what is observed in the profiler.

Recommendation

There are several strategies that can be taken to mitigate this issue, including limiting the amount of addresses a peer can send or further decreasing the maximum message size.

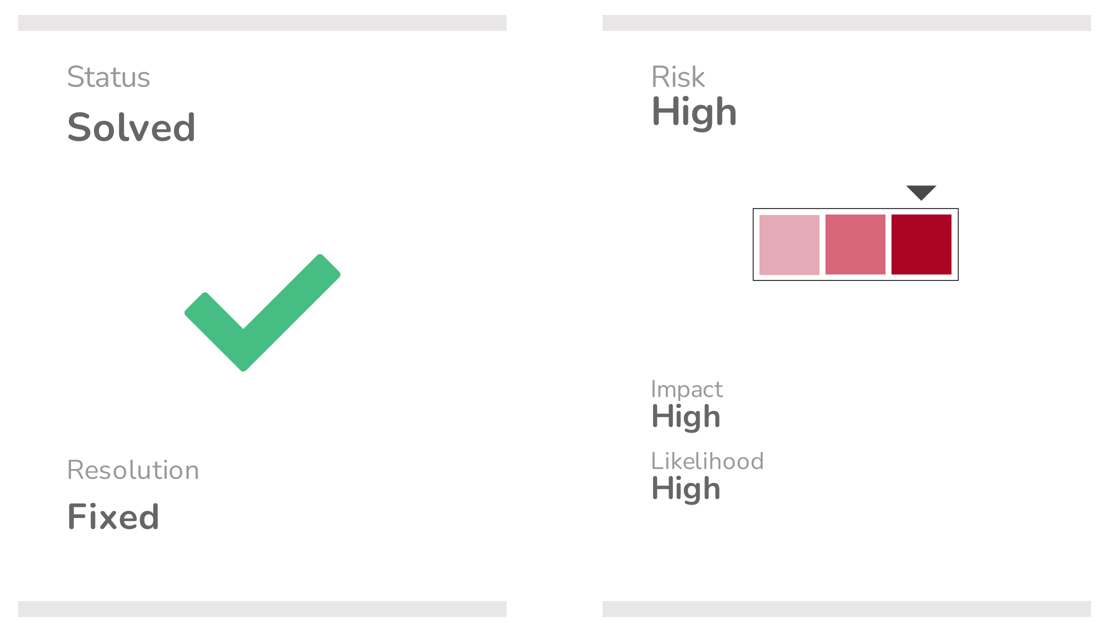
Status

The amount of addresses a peer can send is now limited.

© Coinspect 2024

TARI-022

Attackers can crash miners by sending a block with a high coinbase fee



Location

`base_layer/core/src/consensus/consensus_manager.rs`

Description

Attackers can construct a block that has a coinbase fee of `u64::MAX`. Nodes that receive this block crash due to an overflow when trying to check the coinbase output.

The attack is possible for anyone because the overflow happens in internal validations, before proof of work checks are done.

The overflow occurs in the `calculate_coinbase_and_fees` method:

```
    pub fn calculate_coinbase_and_fees(&self, height: u64, kernels: &
[TransactionKernel]) -> MicroTari {
        let coinbase = self.emission_schedule().block_reward(height);
```

© Coinspect 2024

```
        kernels.iter().fold(coinbase, |total, k| total + k.fee)
    }
```

The attacker is not penalized in any way and can reuse the same block to trigger the same overflow over and over again.

There are several other places where overflow might potentially occur. For example, the `TransactionWeight::calculate` also naively multiplies non-trusted values; although in that case the amount of data needed to effectively trigger the error is too big.

Recommendation

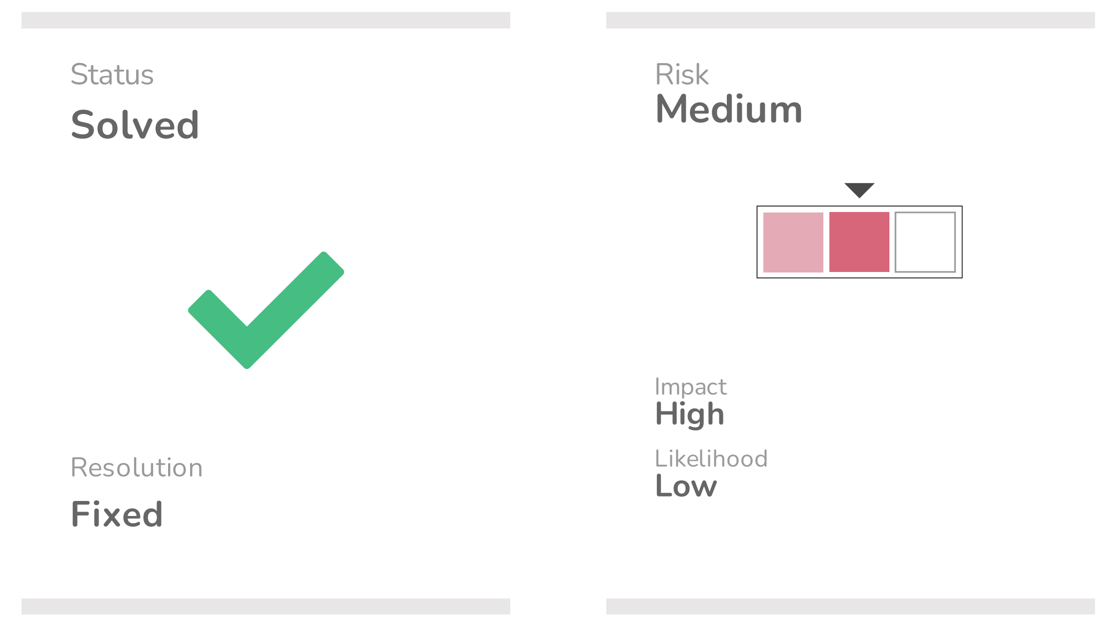
All operations in the codebase should use `checked_` operations unless they are proven not to trigger an overflow.

Status

Fixed. Calculations now use `checked_add`.

TARI-023

Attackers can observe and manipulate gRPC requests to wallets



Location

`applications/tari_console_wallet/src/grpc/wallet_grpc_server.rs`

Description

The use of HTTP allows an unauthorized attacker to intercept credentials or even seize control of the connection between a wallet and a node, thereby compromising the integrity and confidentiality of the data.

The following code snippet shows how the merge miner connects to a wallet using HTTP:

```
info!(target: LOG_TARGET, "Connecting to wallet at {}", wallet_addr);
let wallet_addr = format!("http://{}", wallet_addr);
let wallet_client =
        WalletGrpcClient::connect_with_auth(&wallet_addr,
&config.console_wallet_grpc_authentication).await?;
```

The likelihood of this issue is deemed low because it is expected most configurations will keep wallets and node running on the same host. It also requires the adversary to be placed on the same network.

Recommendation

Follow the recommended guidelines at Authentication | gRPC to implement `SSL/TLS` based authentication and transport security.

Status

Tari addressed this issue in two commits:

`4be85e0f0d778ee813fa41df927cfcac79a3f1db`
`17676ce8f90b0f2aa413218cc72c21c587293d32`.

The first commit had all the infrastructure necessary for the project to use SSL, including a warning outlying the risks of generating self-signed certificates. Nevertheless, there were two instances where connections via `http` were made: in `run_merge_miner.rs` and in `run_miner.rs`, in both cases in the method `connect_wallet()`.

The Tari team shared a fix for the `audit-fixes` branches in `17676ce8f90b0f2aa413218cc72c21c587293d32` for the `run_miner.rs` file, but the `run_merge_miner.rs` still made the connection through HTTP.

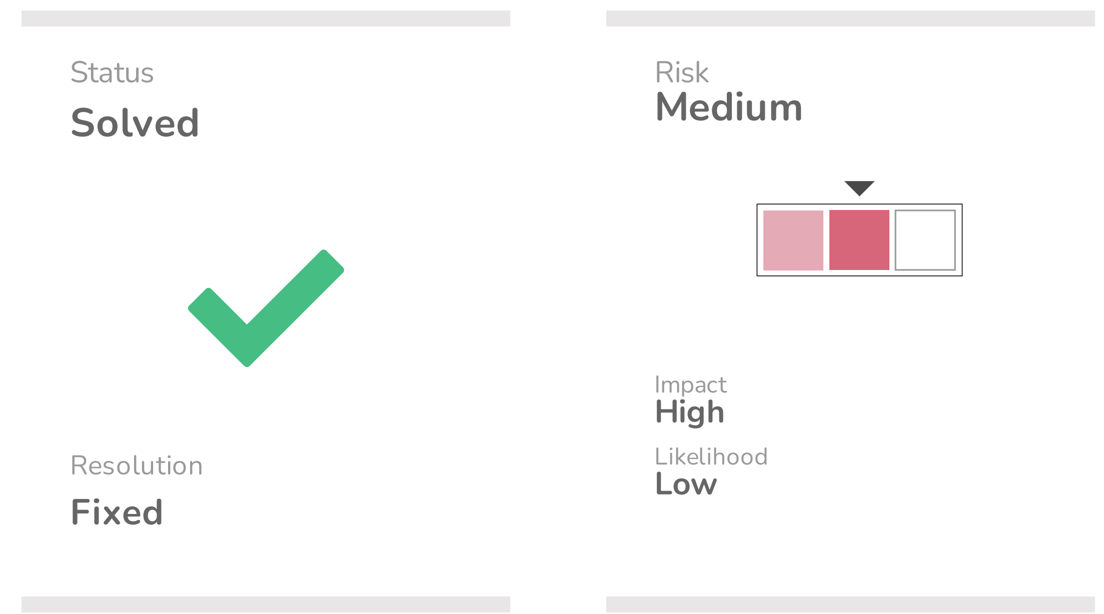
The Tari dev team explained that the was the case because in their `development` branch, which includes changes not in scope for this audit, the `connect_wallet()` method does not exist anymore: miners do not need to speak to wallets anymore.

Coinspect checked that commit `89b19f6de8f2acf28557ca37feda03af2657cf30` of the `development` branch indeed removed the vulnerable lines of code and that in commit `b80f7e366b14e10b3fb0e9835fb76dd5596d0cf8` the issue is entirely fixed.

The risk is fully mitigated in the `development` branch as of commit `b80f7e366b14e10b3fb0e9835fb76dd5596d0cf8`.

TARI-024

Attackers can crash the wallet via the gRPC service



Location

`applications/tari_app_grpc/src/authentication/server_interceptor.rs`

Description

A design flaw in the implementation of the `ServerAuthenticationInterceptor` allows an attacker to crash the wallet and reduces the complexity of a brute-force attack.

The `ServerAuthenticationInterceptor` as it is implemented allows an unauthenticated attacker to crash the wallet service by supplying a crafted `Argon2` hash value with certain parameters that will allocate a high amount of memory or employ too much CPU time.

The attacker needs to be able to send authentication requests to the wallet. This string consists of a string in the form `Basic: Base64Data`. The standard version of the encoded data is a pair (username, password) delimited by a colon character.

© Coinspect 2024

The parsing of the header is performed by the method `BasicAuthCredentials::from_header`:

```
impl ServerAuthenticationInterceptor {
    pub fn new(auth: GrpcAuthentication) -> Self {
        Self { auth }
    }

    fn handle_basic_auth(
        &self,
        req: Request<()>,
        valid_username: &str,
        valid_password: &[u8],
    ) -> Result<Request<()>, Status> {
        match req.metadata().get(AUTHORIZATION.as_str()) {
            Some(t) => {
                let val = t.to_str().map_err(unauthenticated)?;
                let credentials =
BasicAuthCredentials::from_header(val).map_err(unauthenticated)?;
                credentials
                    .validate(valid_username, valid_password)
                    .map_err(unauthenticated)?;
                Ok(req)
            },
            _ => Err(unauthenticated("Missing authorization header")),
        }
    }
}
```

Once the authorization header has been parsed, it is then used to compare it to the username and password set by the owner of the service (in this case the wallet service):

```
impl BasicAuthCredentials {
    pub fn validate(&self, username: &str, password: &[u8]) ->
Result<(), BasicAuthError> {
        if self.user_name.as_bytes() != username.as_bytes() {
            return Err(BasicAuthError::InvalidUsername);
        }
        // These bytes can leak if the password is not utf-8, but since
argon encoding is utf-8 the given
        // password must be incorrect if conversion to utf-8 fails.
        let bytes = self.password.reveal().to_vec();
        let str_password = Zeroizing::new(String::from_utf8(bytes)?);
        let header_password = PasswordHash::parse(&str_password,
Encoding::B64)?;
        Argon2::default().verify_password(password, &header_password)?;
        Ok(())
    }
}
```

The `BasicAuthCredentials::validate` method compares the username and password for the service (specified in the service's configuration file) with the `BasicAuthCredentials` parsed from the network. To do so it uses

`Argon2::default().verify_password` which takes a plain text password string and a hash of the expected password. Unfortunately, the wallet implementation inverts these parameters and instead of passing the client supplied password to the function, it uses the expected password and performs `PasswordHash::parse` on the attacker controlled input.

The way argon2 hashes are designed, allows users to store the hashing parameters (salt, memory usage, CPU usage, etc) in the hash itself for future updates and flexibility. This means that `PasswordHash::parse` does more than parsing a hashed password; it sets the parameters (including algorithm, memory used, and time spent) that will be used to verify the password, therefore the parsed data should come from a trusted source, for instance a database or a local configuration file, as is the case with the wallet.

The following test case illustrates how an attacker can make the wallet service allocate a large amount of memory (m=10000000) and run for large amount of iterations (t=10000):

```
#[test]
fn it_allocates_a_lot_of_memory_and_runs_for_a_while() {
        let hashed =
"$argon2id$v=19$m=10000000,t=10000,p=1$PhPhJQKM2tLzYr34/myoOw$fCBdhFCIE
E65MIjeeR6NdeNTAfjwhCEQ84jGj9n8uig";
        let credentials =
BasicAuthCredentials::new("admin".to_string(), hashed.into());
        credentials.validate("admin", b"secret").unwrap();
}
```

Because the parameters are controlled by the attacker, it is possible for them to crash the program or use an arbitrary amount of resources on the victim system.

This issue could also lead to complete authentication bypass due to the current design: an attacker can supply a hash with parameters that are less secure than the default ones defined by the `argon2` crate. For example the `argon2` specification defines the minimum hash byte length to be 4 bytes, so an attacker could send a hash with these parameters and they would have to only brute-force `2^32` passwords to find a collision with the actual password.

This situation is rendered several orders of magnitude harder due to the better defaults that the argon2 crate defines for the minimum hash size, which is 10 bytes, making the attack unfeasible.

The likelihood of this issue is deemed low as it is expected that most configurations will keep wallets and node running on the same host.

© Coinspect 2024

Recommendation

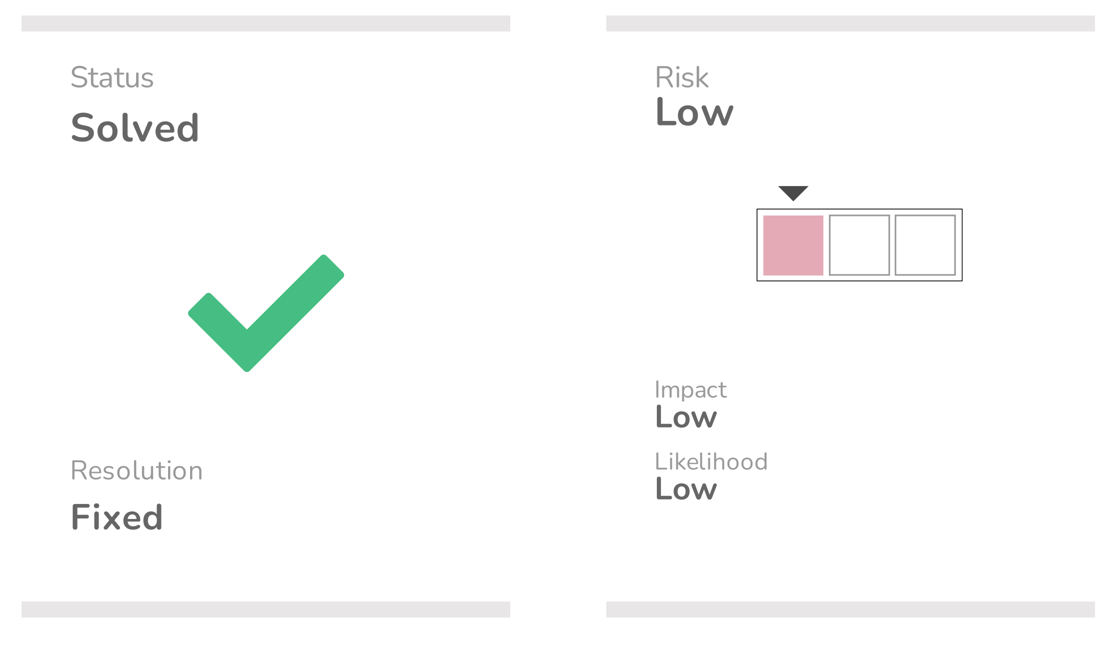
If the existing authentication method is retained, the client must provide a password for the server to hash and compare with the stored hash of the expected password.

Status

Fixed. The implementation now correctly creates a `BasicAuthCredentials` from the stored values instead of the user-supplied ones.

TARI-025

Wallet username vulnerable to timing attacks



Location

`applications/tari_app_grpc/src/authentication/basic_auth.rs`

Description

The `validate` method does not use a constant-time compare function to check usernames. This makes it possible for an attacker to analyze the time it takes for the method to return in order to optimize their bruteforce attack.

```
if self.user_name.as_bytes() != username.as_bytes() {
    return Err(BasicAuthError::InvalidUsername);
}
```

The absence of constant-time comparison permits an attacker to infer details about the correct username by observing how long the system takes to respond to various inputs. Consequently, an attacker can exploit this vulnerability to perform a side-channel timing attack, systematically guessing the username and analyzing

the time taken for the system to respond. Over repeated attempts, this could allow an attacker to reconstruct the correct username.

Recommendation

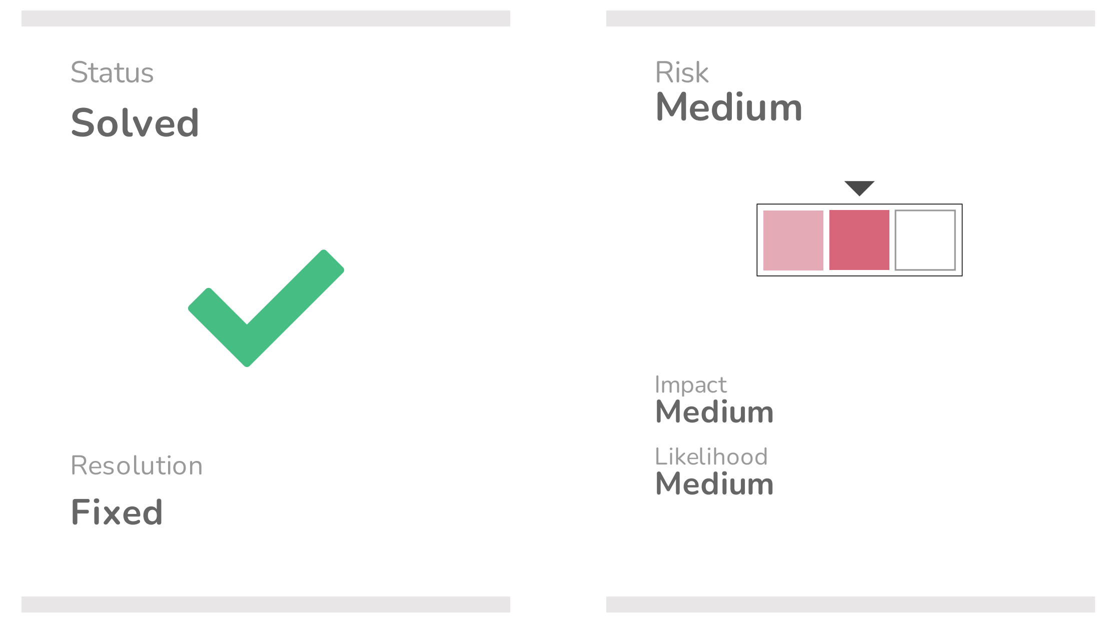
Implement a constant-time comparison method for usernames to prevent side-channel attacks.

Status

Fixed. The code now implement constant-time comparisons.

TARI-026

Attackers can hold the peer-stream open indefinitely



Location

`comms/dht/src/network_discovery/discovering.rs`

Description

Malicious peers can hold a `GetPeersRequest` stream open indefinitely when an innocent peers joins the network and requests peers from them. The malicious peer can send the same peer over and over again.

The victim validates the shared peers before marking it as a duplicates and never takes any action against the attacker as long as peers sent are valid.

The cause can be seen in the `request_peers` file:

```
    async fn request_peers(
        &mut self,
        sync_peer: &NodeId,
        mut client: rpc::DhtClient,
```

```
    ) -> Result<(), NetworkDiscoveryError> {
        debug!(
            target: LOG_TARGET,
            "Requesting {} peers from `{}`",
            self.params
                .num_peers_to_request
                .as_ref()
                .map(ToString::to_string)
                .unwrap_or_else(|| "∞".into()),
            sync_peer
        );
        match client
            .get_peers(GetPeersRequest {
                n: self
                    .params
                    .num_peers_to_request
                    .map(|v| u32::try_from(v).unwrap())
                    .unwrap_or_default(),
                include_clients: true,
            })
            .await
        {
            Ok(mut stream) => {
                while let Some(resp) = stream.next().await {
                    match resp {
                        Ok(resp) => match resp.peer.and_then(|peer|
peer.try_into().ok()) {
                            Some(peer) => {
                                self.validate_and_add_peer(sync_peer,
peer).await?;
                            },
                            None => {
                                debug!(target: LOG_TARGET, "Invalid
response from peer `{}`", sync_peer);
                            },
                        },
                        Err(err) => {
                            debug!(target: LOG_TARGET, "Error response
from peer `{}`: {}", sync_peer, err);
                        },
                    }
                }
            },
            Err(err) => {
                debug!(
                    target: LOG_TARGET,
                    "Failed to request for peers from peer `{}`: {}",
sync_peer, err
                );
            },
        }

        Ok(())
    }
```

As long as the attacker keeps sending peers, `validate_and_add_peer` will be called.

An attacker could take advantage of this by setting up several malicious nodes on the network which kept the discovery process busy in order to make it more difficult for victims to find honest peers.

Recommendation

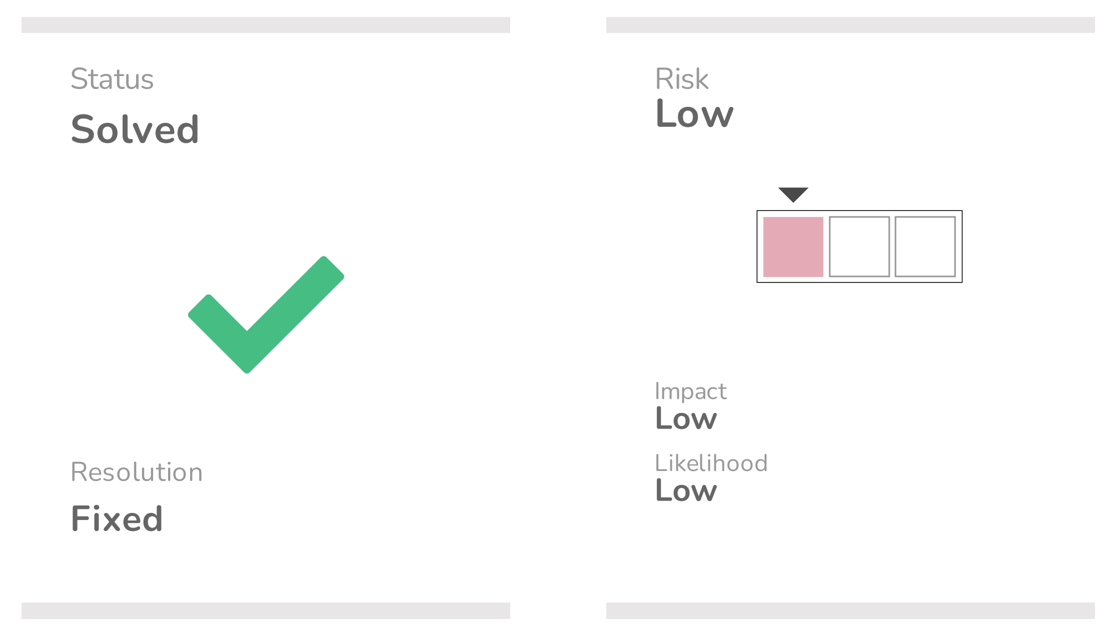
Set a maximum amount of peers the client will listen for as well as timeouts. Have low tolerance for sending duplicated peers.

Status

Fixed. The node receiving peers now have sanity checks that prevent an attacker from keeping the stream open indefinitely.

TARI-027

Overflow when computing accumulated difficulty will eventually halt mining



Location

base_layer/core/src/blocks/accumulated_data.rs:291

Description

An overflow of the u64 variable that keeps track of the accumulated chain difficulty will cause nodes to stop ingesting new blocks.

Tari uses a u64 variable to record the accumulated difficulty from its genesis block. Similarly, the current difficulty of each block is stored in a u64 variable, which is added to the accumulated difficulty each time a block is received. Over time, this will lead to an overflow of accumulated difficulty variable. As a result, nodes will be unable to process new blocks due to the inability to track the rising accumulated difficulty.

The code below adds the achieved difficulty of the current block (achieved_target) to the accumulated difficulty since the genesis block. In case of overflow, it will

© Coinspect 2024

trigger the `BlockError::DifficultyOverflow` error.

```
let (randomx_diff, sha3x_diff) = match achieved_target.pow_algo() {
    PowAlgorithm::RandomX => (
        previous_accum
            .accumulated_randomx_difficulty
            .checked_add(achieved_target.achieved())
            .ok_or(BlockError::DifficultyOverflow)?,
        previous_accum.accumulated_sha3x_difficulty,
    ),
    PowAlgorithm::Sha3x => (
        previous_accum.accumulated_randomx_difficulty,
        previous_accum
            .accumulated_sha3x_difficulty
            .checked_add(achieved_target.achieved())
            .ok_or(BlockError::DifficultyOverflow)?,
    ),
};
```

This error will not cause the node to crash, but it will prevent the new block from being added to the current chain.

The impact and likelihood of these issue are low due to several factors that mitigate its risk, such as the high amount of work needed to reach the `u64`.

Recommendation

Store the accumulated difficulty in a `u256` variable instead, similar to Bitcoin's approach for storing accumulated work.

Status

Fixed. Tari has increased the accumulated difficultly to be `u128`, which is big enough to be of no concern while remaining practical.

TARI-028

Attackers can crash the node with a malicious Monero coinbase

Location

`base_layer/core/src/proof_of_work/monero_rx/pow_data.rs`

Description

Attackers can crash the node by sending a crafted Monero proof-of-work. The crash happens due to an attempt to allocate more memory than possible into a vector. The root cause can be found in the `monero` dependency used to parse the monero data.

For Tari, the issue is triggered in the `BorshDeserialize` method of the `MoneroPowData` struct:

```
impl BorshDeserialize for MoneroPowData {
    fn deserialize(buf: &mut &[u8]) -> io::Result<Self> {
        let header = monero::BlockHeader::consensus_decode(buf)
            .map_err(|e| io::Error::new(io::ErrorKind::InvalidData,
e.to_string()))?;
```

© Coinspect 2024

```
        let randomx_key = BorshDeserialize::deserialize(buf)?;
        let transaction_count = BorshDeserialize::deserialize(buf)?;
        let merkle_root = monero::Hash::consensus_decode(buf)
            .map_err(|e| io::Error::new(io::ErrorKind::InvalidData,
e.to_string()))?;
        let coinbase_merkle_proof =
BorshDeserialize::deserialize(buf)?;
        let coinbase_tx = monero::Transaction::consensus_decode(buf)
            .map_err(|e| io::Error::new(io::ErrorKind::InvalidData,
e.to_string()))?;
        Ok(Self {
            header,
            randomx_key,
            transaction_count,
            merkle_root,
            coinbase_merkle_proof,
            coinbase_tx,
        })
    }
}
```

An attacker must craft a malicious message so as to reach the `Transaction::consensus_decode` method. The attempted allocation happens inside the `monero` crate in the `decode_size_vec!` macro, called when attempting to parse a `Bulletproof` in the coinbase transaction:

```
macro_rules! decode_sized_vec {
    ( $size:expr, $d:expr ) => {{
        let mut ret = Vec::with_capacity($size as usize);
        for _ in 0..$size {
            ret.push(Decodable::consensus_decode($d)?);
        }
        ret
    }};
}
```

The program calls expands the macro with a `$size` of 4,177,002,576. The amount of bytes this actually holds depends on the size of the type of the value `$d`, which is 8, resulting in a minimum allocation of 33,416,020,608 bytes or about 31 Gb.

Rust nevertheless tries to allocate even more space (around 1307 Gb in auditors' tests) and the program crashes immediately:

```
Conseunsus decoding RctSigPrunnable
Non-Bulletproof2 , non-Clsag
decoding sized vec with capacity: 4177002576
vector holds type of size: 8
so total size is: 33416020608
memory allocation of 1403472865536 bytes failed
```

© Coinspect 2024

Recommendation

The issue is not easy to fix from Tari's codebase. Consider collaborating with the monero library to find similar bugs and fix them upstream.

Status

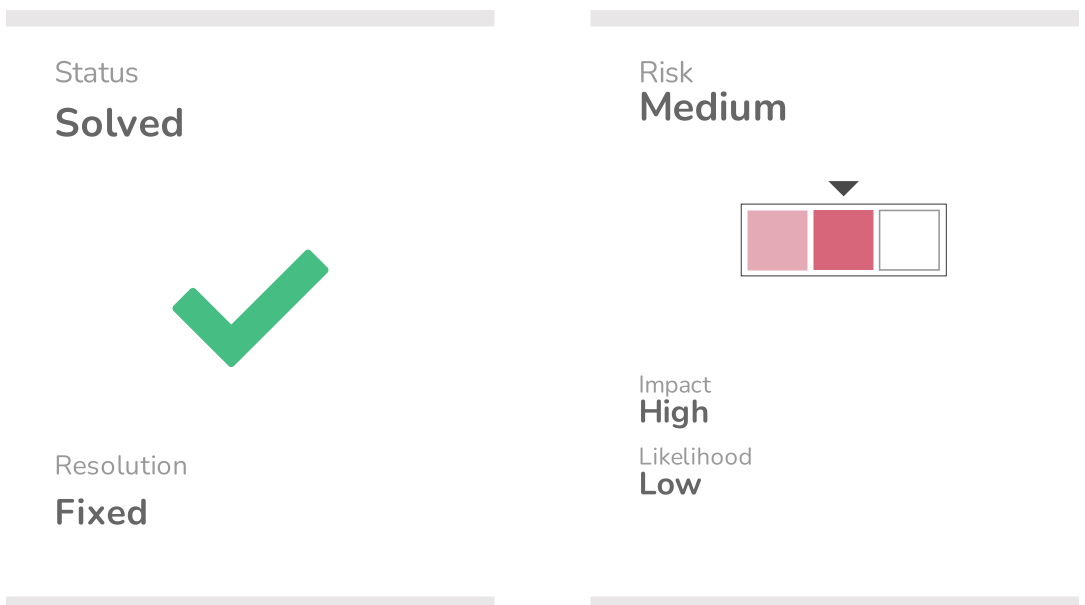
This particular issue has been mitigated by downgrading the `monero-rs` library to version `0.18`, where this behavior is not reproduced.

It is worth stating that continued assessment of dependencies is needed to make sure other vulnerabilities are not introduced when changing, upgrading or downgrading third-party components.

© Coinspect 2024

TARI-029

Attackers can crash the node with a single gRPC request



Description

An attacker with access to the gRPC can crash the node by sending a malicious `get_network_difficulty` request.

The crash is triggered by an overflow when calculating the page offsets:

```
let page_iter = NonOverlappingIntegerPairIter::new(start_height,
end_height + 1, GET_DIFFICULTY_PAGE_SIZE);
```

An attacker crafts a message such that `end_height` is `u64::max` and adjust `start_height` and `from_tip` so that the request is valid and reaches that line and causes a panic in the node.

This is the exact program that an attacker can use:

```
async fn main() {
    let mut client =
BaseNodeGrpcClient::connect("http://localhost:18162").await.unwrap();
    client.get_network_difficulty(HeightRequest {
        from_tip: 0,
        start_height: u64::MAX,
        end_height: u64::MAX
    }).await.unwrap();
}
```

Recommendation

As mentioned in `TARI-022`, all mathematical operations should be protected from overflowing.

Status

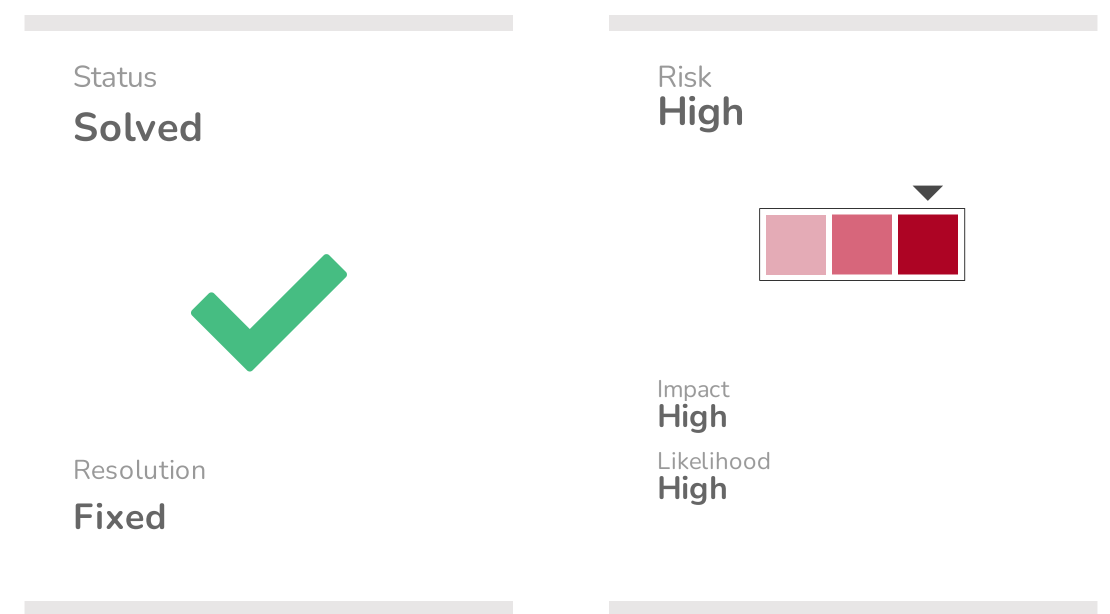
Fixed. This particular instance of overflow has been addressed by using a safe adding method:

```
    NonOverlappingIntegerPairIter::new(start_height,
end_height.saturating_add(1), GET_DIFFICULTY_PAGE_SIZE)
```

© Coinspect 2024

TARI-030

Peer can crash horizon-syncing node with a crafted bitmap



Location

`base_layer/core/src/base_node/sync/horizon_state_sync/synchronizer.rs`

Description

Attackers can crash a peer which is horizon-syncing from them with a crafted bitmap. The bitmap makes the `CRoaring` dependency try to allocate invalid memory.

The crash is triggered when receiving the `DeletedDiff` of the horizon sync process. In particular, it is caused by an attempt to deserialize the malicious bitmap:

```
let diff_bitmap = Bitmap::try_deserialize(&diff_bitmap).ok_or_else(|| {
    HorizonSyncError::IncorrectResponse(format!(
        "Peer {} sent an invalid difference bitmap",
        sync_peer.node_id()
```

© Coinspect 2024

```
    ))
})?;
```

Recommendation

As with **TARI-028**, the issue is hard to mitigate from the Tari codebase, and should be addressed upstream.

Status

Fixed. `Croaring` is no longer a dependency of Tari.

TARI-031

Attackers can crash the node with a script

Location

infrastructure/tari_script/src/script.rs

Description

An attacker can craft a transaction with a malicious script that crashes any node which attempts to execute it.

The crash is triggered by an underflow in the `OP_COMPARE_HEIGHT` opcode:

```
    fn handle_compare_height(stack: &mut ExecutionStack, block_height:
u64) -> Result<(), ScriptError> {
        let target_height = stack.pop_into_number::<i64>()?;
        let block_height = i64::try_from(block_height)?;

        let item = StackItem::Number(block_height - target_height);

        stack.push(item)
    }
```

An attacker creates a script which puts into the stack the `i64::MIN`. Any subtraction will then cause the underflow.

```
let script = TariScript::new(vec![PushInt(i64::MIN), CompareHeight]);
```

Recommendation

Like **TARI-022** and **TARI-029**: review all mathematical operations to make sure they cannot overflow or do so in a safe manner.

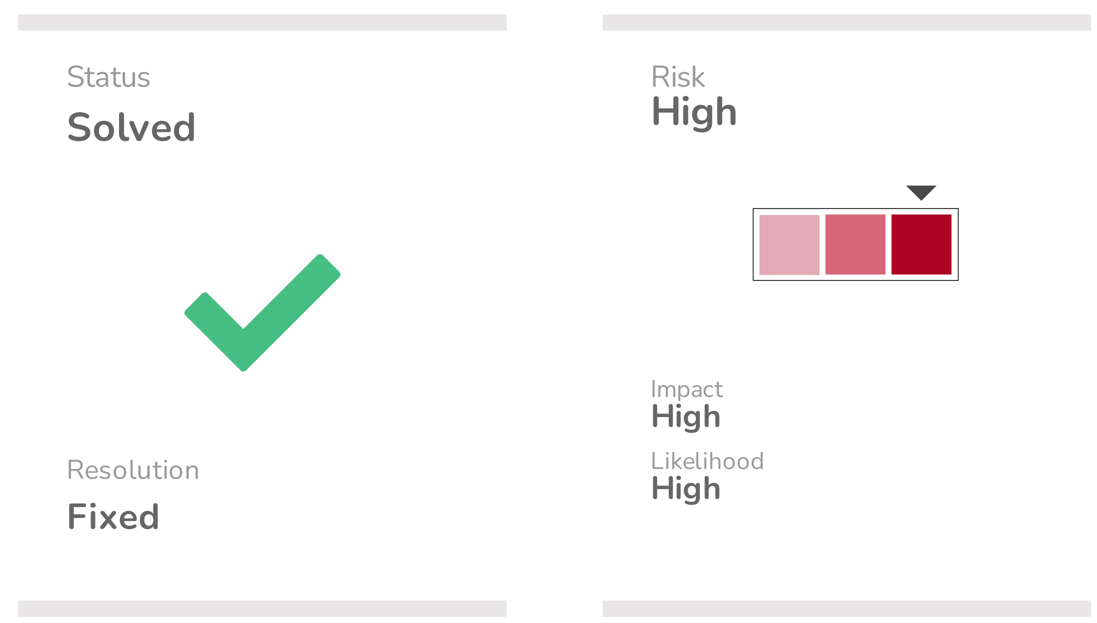
Status

Fixed. This instance has been checked by using safe math operators:

```
        let item = match block_height.checked_sub(target_height) {
            Some(num) => StackItem::Number(num),
            None => {
                return Err(ScriptError::CompareFailed(
                    "Couldn't subtract the target height from the
current block height".to_string(),
                ))
            },
        };
```

TARI-032

Attackers can crash node with a mempool message



Location

`base_layer/core/src/mempool/service/inbound_handlers.rs`

Description

Attackers can crash the node with a `SubmitTransaction` request to a node's mempool.

The cause is the `debug!` log line that is called whenever a new request is received:

```
    pub async fn handle_request(&mut self, request: MempoolRequest) ->
Result<MempoolResponse, MempoolServiceError> {
        debug!(target: LOG_TARGET, "Handling remote request: {}",
request);
```

This will indirectly call the `fmt` of the `Display` trait for the request. In the case of a `SubmitTransaction`, the method performs an unsafe access to the first element of

© Coinspect 2024

the `kernels()` array of the request.

```
            MempoolRequest::SubmitTransaction(tx) => write!(
                f,
                "SubmitTransaction ({})",
                tx.body.kernels()
[0].excess_sig.get_signature().to_hex()
            ),
```

The attacker only has to send an invalid transaction with no kernels to trigger a panic and halt the node.

It is worth noting that the unsafe access is repeated later in the `handle_request` method, although it is impossible to trigger now because the panic will happen in previous `debug!` macro:

```
            SubmitTransaction(tx) => {
                debug!(
                    target: LOG_TARGET,
                    "Transaction ({}) submitted using request.",
                    tx.body.kernels()
[0].excess_sig.get_signature().to_hex(),
                );

Ok(MempoolResponse::TxStorage(self.submit_transaction(tx,
None).await?))
            },
```

Recommendation

Do not directly access collection values that might not exist. Prefer `.get()` instead of `[]` so an `Option` is returned.

Consider using Clippy's `indexing_slicing` linter to warn or deny slicing on vectors and arrays.

Status

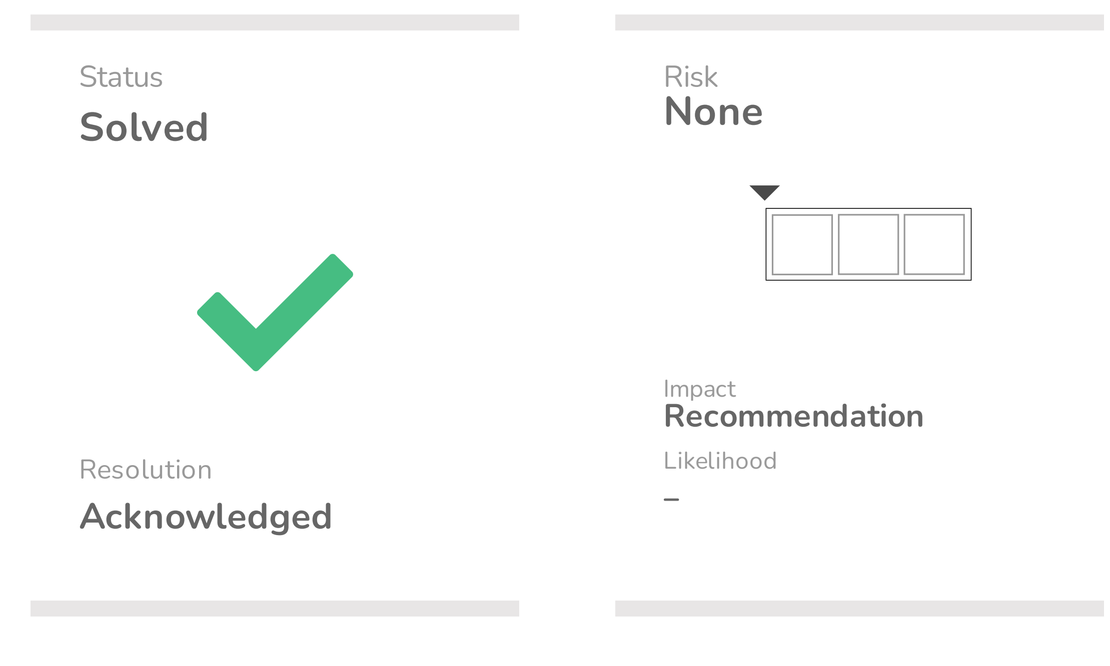
Fixed on `a172b389e89514d7191f6aa48291e625fd8a9a1c`.

This issue was originally intended to be fixed as of commit `14e334aff346aae8a081599488135c905c2c1f84`. Nevertheless, on that commit only the impossible-to-trigger vulnerability was addressed. Coinspect warned Tari

about the oversight and the development team fully patched the issue on the new commit.

TARI-033

Dependencies that depend on wrap-on-overflow will crash Tari



Location

Description

Tari configures its release profile to throw a `panic!` on overflow. While the intention behind this is correct, as a `panic!` and even a network halt is preferable over incorrect computations, it is not the default behavior in Rust programs.

This raises the concern that some external dependencies might depend on the default wrap-on-overflow behavior of the Rust compiler. It is worth noting that the base project has full control over the `profile` section of the `Cargo.toml`:

```
Cargo supports custom configuration of how rustc is invoked through
profiles at the top level. Any manifest may declare a profile, but only
the top level project's profiles are actually read. All dependencies'
profiles will be overridden. This is done so the top-level project has
control over how its dependencies are compiled.
```

This is specially important because Tari depends on several cryptographic third-party crates. It is common for cryptography operations to use wrap-on-overflow semantics as their operations work on a prime field where wrapping leads to the correct result.

Recommendation

This is an informational issue as no direct impact has been identified. Nevertheless, Coinspect considered this issue worth of Tari's attention due to the amount of overflow related panics found on the audit.

Tari's team needs to take into account that their build options affect their dependencies as well and test each of their dependencies with this in mind. It is likely that at least some will need the development of a wrapper to provide safe mathematical operations that do not rely on Rust's defaults.

Status

Tari has acknowledged this risk and stated:

```
Currently no dependancy has this set, will continue to check this as we
move along
```

TARI-034

Wrong test for double spend

Location

`base_layer/core/src/validation/block_body/test.rs`

Description

The Tari test case that checks for double spends is wrong, as it actually tests for duplicated excess factors and does not consider that an input can be reused while changing the excess factor of the transaction.

The test is named `it_checks_double_spends`:

```
async fn it_checks_double_spends() {
    let (mut blockchain, validator) = setup(true);

    let (_, coinbase_a) = blockchain.add_next_tip(block_spec!
("A")).await.unwrap();
    let (txs, _) =
        schema_to_transaction(&[txn_schema!(from: vec![coinbase_a], to:
vec![50 * T])], &blockchain.km).await;

    blockchain
```

```
        .add_next_tip(block_spec!("1", transactions: txs.iter().map(|t|
(**t).clone()).collect()))
        .await
        .unwrap();
    let (block, _) = blockchain
        .create_next_tip(
            BlockSpec::new()
                .with_transactions(txs.iter().map(|t|
(**t).clone()).collect())
                .finish(),
        )
        .await;
    let txn = blockchain.db().db_read_access().unwrap();
    let err = validator.validate_body(&*txn,
block.block()).unwrap_err();
    assert!(matches!(err, ValidationError::DuplicateKernelError(_)));
}
```

The issue is that the test uses the `txs` transactions twice, adding them to the blockchain and then trying to validate a new block with the same transactions again. This results in a `DuplicateKernelError`. But to correctly test for a double spend, the inputs used are more important than the excess factor: an input can be reused and added to other inputs to change the excess factor while still resulting in a double spend.

No tests that check for correct error `ContainsSTxO` are in the project.

Recommendation

Rename the test to `it_checks_for_duplicated_kernels`. Create a new test that reuses an input to check for double spends.

Coinspect auditors drafted this test which checks for the correct expected error:

```
async fn it_checks_double_spends() {
    let (mut blockchain, validator) = setup(true);

    // Create a new tip with a block, return its coinbase
    let (_, coinbase_a) = blockchain.add_next_tip(block_spec!
("A")).await.unwrap();

    // Create a TX using the coinbase as an input
    let (txs, _) =
        schema_to_transaction(&[txn_schema!(from: vec!
[coinbase_a.clone()], to: vec![50 * T])], &blockchain.km).await;

    // Add to the blockchain a block with the tx
    blockchain
        .add_next_tip(block_spec!("1", transactions: txs.iter().map(|t|
(**t).clone()).collect()))
        .await
```

```
        .unwrap();

    let (txs_reusing_coinbase, _) = schema_to_transaction(&[txn_schema!
(from: vec![coinbase_a], to: vec![100 * T])],
        &blockchain.km).await;

    // Add to the blockchain a new block
    let (block, _) = blockchain
        .create_next_tip(
            BlockSpec::new()
                .with_transactions(txs_reusing_coinbase.iter().map(|t|
(**t).clone()).collect())
                .finish(),
        )
        .await;
    let txn = blockchain.db().db_read_access().unwrap();
    let err = validator.validate_body(&*txn,
block.block()).unwrap_err();
    assert!(matches!(err, ValidationError::ContainsSTxO));
}
```

Status

Fixed. A new test has been added which checks for the correct error.

TARI-035

Attackers can crash the node by sending transaction with big fee

Location

```
.../core/src/validation/aggregate_body/aggregate_body_internal_validat
or.rs
```

Description

An attacker can send a transaction with a big fee and crash the node. This is similar to TARI-022 but leverages another path in the codebase.

The vulnerable method is `sum_kernels`, which will panic when trying to sum the fees from a transaction that has over `u64::MAX` in fees.

```
fn sum_kernels(body: &AggregateBody, offset_with_fee:
PedersenCommitment) -> KernelSum {
    // Sum all kernel excesses and fees
    body.kernels().iter().fold(
        KernelSum {
            fees: MicroMinotari(0),
```

```
            sum: offset_with_fee,
        },
        |acc, val| KernelSum {
            fees: acc.fees + val.fee,
            sum: &acc.sum + &val.excess,
        },
    )
}
```

The `sum_kernels` method is called by the `validate_kernel_sum` method, part of the `TransactionInternalConsistencyValidator`, used to validate transactions from the mempool.

Recommendation

Always use safe methods to perform mathematical operations.

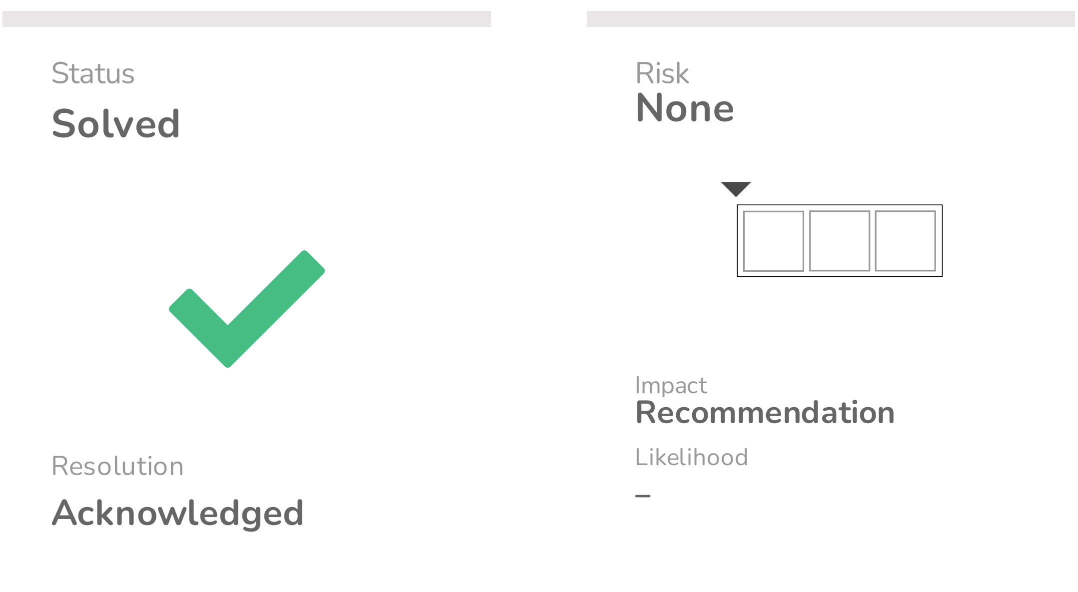
Status

Fixed. `sum_kernels` now uses a safe method to add values and errs on overflow.

© Coinspect 2024

TARI-036

Overflows can potentially crash the node



Description

As demonstrated by **TARI-022**, **TARI-027**, **TARI-031** and **TARI-036**; overflows are a serious concern for the Tari base node. In order to help development team assess and address each potential overflow in the node, this issue lists all potential overflows Coinspect auditors have found.

Not all of the overflows in this list are exploitable. Nevertheless, the recommendation is to always use safe math operations such as `checked_add` when computing to prevent the node from crashing.

1. `sum_commitments` method at ``base_layer/core/src/validation/aggregate_body/aggregate_body_internal_validator.rs`
2. `get_side_chain_utxos` method on `request.count - 1` at `applications/minotari_node/src/grpc/base_node_grpc_server.rs`
3. `generate_coinbase_transaction` method on `let amount = reward + fees` at `base_layer/wallet/src/transaction_service/service.rs`

Recommendation

Address all potential overflows by **always** performing math with safe method such as `checked_add`, `wrapped_add` or `saturating_add`.

Status

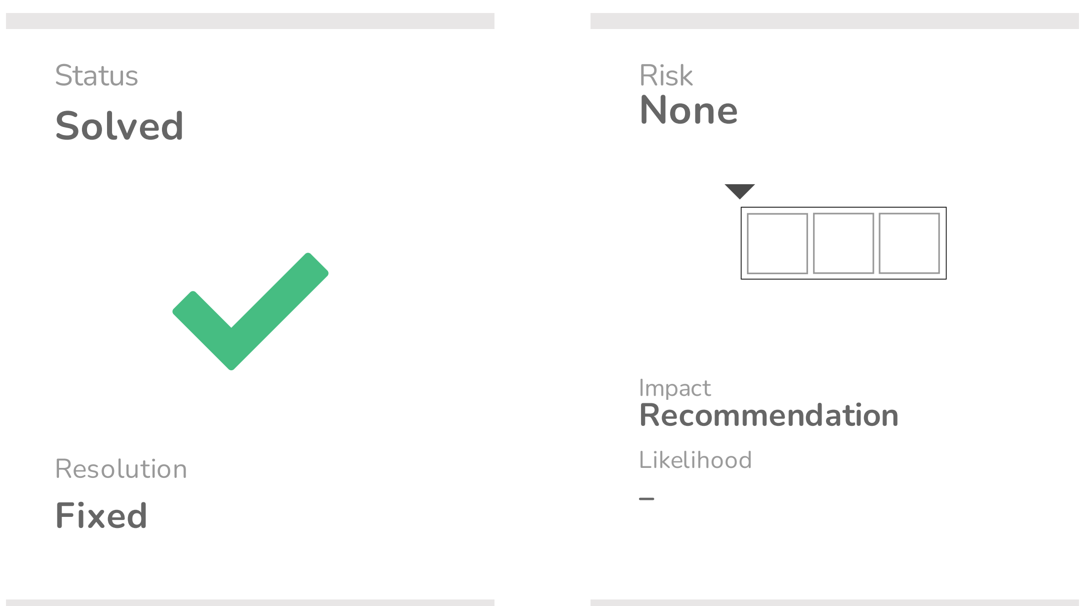
Tari has acknowledged the risk here and stated:

```
(we) have been auditing the code for evey single math operation to
either use checked/saturating or we have verified an overflow not
possible at all.
```

© Coinspect 2024

TARI-037

Sync is always done with a single peer



Description

The sync process always attempts to sync the blockchain with a single peer at a time.

This is considered a fragile design because it gives an untrusted peer too much control over the syncing process, allowing malicious actors to potentially execute exploits more effectively.

A syncing process that gets information from multiple peers to sync would be less susceptible to this kind of manipulation.

Leveraging issues like `TARI-013` into Denial of Service attacks are much more likely when the syncing process is restricted into a single peer at a time.

Recommendation

Download sync data from multiple peers.

Status

Fixed. Tari has implemented a way to wait for multiple peers to present sync data before starting the initial sync.

It is worth noting that this protection is only implemented for initial sync and not for other scenarios where the peer has lagged behind.

© Coinspect 2024

TARI-038

Miner can force peers to sync

Location

`base_layer/core/src/base_node/sync/header_sync/synchronizer.rs`

Description

A miner or attacker with a block with higher proof-of-work can lie about their difficulty and height, forcing a peer to perform sync. This makes the victim not accept new blocks and perform useless validations. The attacker is not penalized.

This issue is similar to **TARI-013**, but attackers need to provide a block with higher proof of work. This makes the motivation similar to **TARI-001**: a miner wants to delay other miners and gain advantage. In this case they can get up to 60 seconds advantage due to acquiring the `add_block` lock.

The attack leverages the fact that the initial header sync performed in `find_chain_split` does not check for duplicated headers and makes requests to an untrusted node.

All in all, an attacker has to:

1. Have a block with at least a slightly higher difficulty than the victim
2. Announce a much greater higher difficulty to the victim in order for them to believe they have `FallenBehind`.
3. Wait `29s` before responding to their `find_chain_split` request
4. Answer `not_found`
5. Victim will repeat request, wait `29s` again
6. Answer with `998` known blocks and the one new block

This causes the victim to validate again the 998 known blocks plus the new one here:

```
self.header_validator.initialize_state(chain_split_hash).await?;
        for header in headers {
            debug!(
                target: LOG_TARGET,
                "Validating header #{} (Pow: {}) with hash: ({})",
                header.height,
                header.pow_algo(),
                header.hash().to_hex(),
            );
            self.header_validator.validate(header).await?;
        }
```

When the victim goes on to continue the `header_sync`, they will correctly set the `has_better_pow` flag to true, as the new chain has indeed a better proof of work.

Because the attacker sent exactly 999 headers, the `pending_len` will be less than `NUM_INITIAL_HEADERS_TO_REQUEST` which is set to `1000`. The `has_better_pow` flag will have been correctly set to true, as the chain is indeed better. This causes the method to return with no more validations.

It is important to note that while in `HeaderSync` mode, the `add_block` lock is taken. This makes adding new blocks to the database impossible while sync is ongoing. The impact analysis is then similar to **TARI-001**: a miner gains advantage over other ones by being able to mine on-top of a new block before the competition. In this scenario, they also force the victim to revalidate known blocks.

It is also worth mentioning that the attack can also be carried out by an attacker that has not found the best block, but will mint or receive it eventually.

Recommendation

Make sure that the syncing peer does not provide already-known blocks during the initial header sync in `determine_sync_status` to mitigate the revalidation of known blocks.

The attacker can still make the node wait for `29s` on each request nevertheless. To address this, consider:

- Banning peers how have lied about their best height and difficulty to force the peer to sync - although consideration must be taken that a peer might have announced a difficulty and then changed their mind about the best chain.
- Reducing the default timeout values.
- Making the `add_block` lock more granular, making sure it is held only when necessary and not during the whole header sync process.

Status

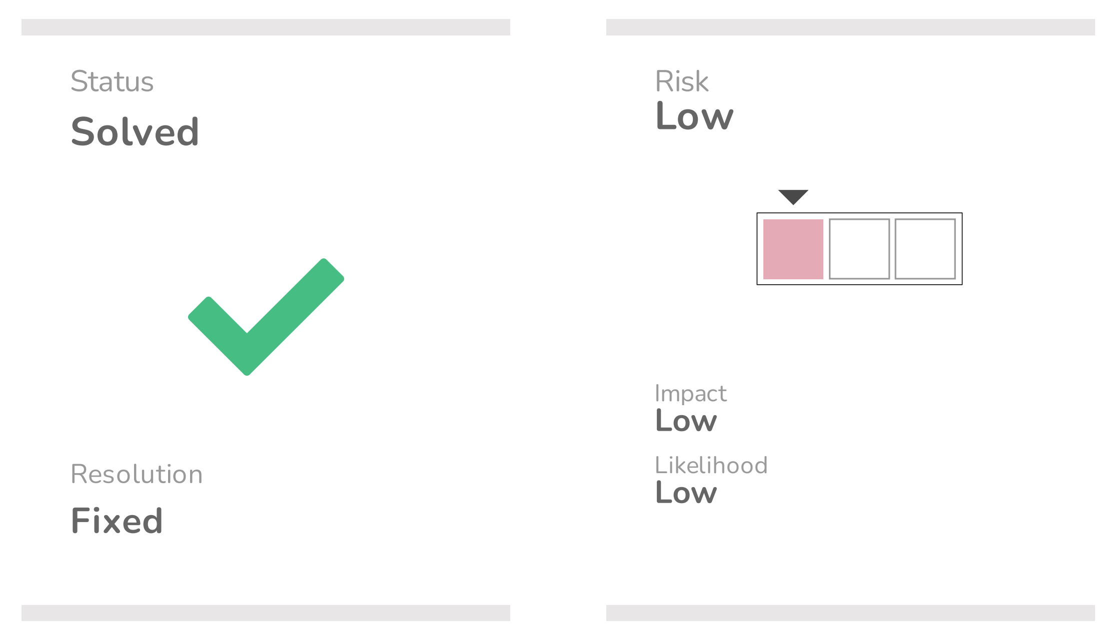
Fixed. New checks have been added that mitigate the chance of a peer sending repeated blocks. In case they do, they are now banned.

The tolerance has been lowered from `30s` to `15s`, further mitigating the impact of the attack.

© Coinspect 2024

TARI-039

Wallet reports stale balance results with no warnings



Location

base_layer/wallet/src/base_node_service/service.rs

Description

The wallet returns stale balance results with no warning to the user if it cannot get a live state from the base node. This will lead clients to use stale results. There is no way to distinguish between stale and updated results.

This is because the `GetBalance` operation uses the `get_chain_metadata` function to calculate the current tip. `GetChainMetadata` in turn will return an `Ok` result even when it fetches possibly outdated data from its database.

```
            OutputManagerRequest::GetBalance => {
                let current_tip_for_time_lock_calculation = match
self.base_node_service.get_chain_metadata().await {
                    Ok(metadata) => metadata.map(|m|
m.height_of_longest_chain()),
```

© Coinspect 2024

```
                    Err(_) => None,
                };
                self.get_balance(current_tip_for_time_lock_calculation)
                    .map(OutputManagerResponse::Balance)
            },
```

```
            BaseNodeServiceRequest::GetChainMetadata => match
self.get_state().await.chain_metadata {
                Some(metadata) =>
Ok(BaseNodeServiceResponse::ChainMetadata(Some(metadata))),
                None => {
                    // if we don't have live state, check if we've
previously stored state in the wallet db
                    let metadata = self.db.get_chain_metadata()?;

Ok(BaseNodeServiceResponse::ChainMetadata(metadata))
                },
            },
```

Recommendation

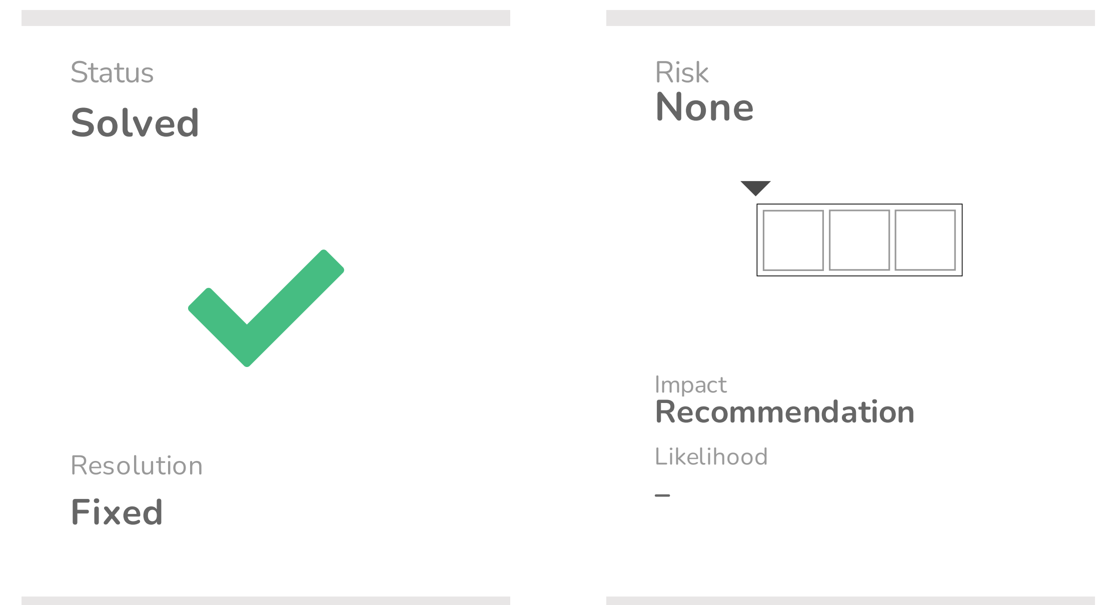
Provide a clear warning to users that they are getting outdated data.

Status

Fixed. The wallet not prints a warning when the latency with the base node passes a threshold.

TARI-040

DirectOnly wallets report the transaction being sent when it is not



Location

`base_layer/wallet/src/transaction_service/protocols/transaction_send_protocol.rs`

Description

Due to a known bug with `DirectSend` which occasionally reports that a transaction has been sent successfully when it has not, even the `send_transaction_direct` method will attempt to use a SAF strategy as a backup.

The problem is that `DirectOnly` nodes are prevented from using SAF:

```
        if self.resources.config.transaction_routing_mechanism ==
TransactionRoutingMechanism::DirectOnly {
            return Ok(false);
        }
```

© Coinspect 2024

This results in `DirectOnly` reporting success on transactions that have not been received.

Recommendation

Deprecate `DirectOnly` or provide a clear warning that this bug is known and its use is not recommended.

Status

Fixed. The wallet now prints a warning message if using `DirectOnly`.

© Coinspect 2024

TARI-041

Attackers can continuously spam low difficulty blocks

Location

base_layer/core/src/base_node/comms_interface/inbound_handlers.rs

Description

An attacker can continuously spam a victim with low difficulty blocks, and this will not result in the banning of the sender. This makes the victim node waste resources and facilitate attacks such as TARI-042.

The lack of ban can be seen in the check_min_block_difficulty method:

```
    async fn check_min_block_difficulty(&self, new_block: &NewBlock) ->
Result<(), CommsInterfaceError> {
        let constants =
self.consensus_manager.consensus_constants(new_block.header.height);
        let min_difficulty =
constants.min_pow_difficulty(new_block.header.pow.pow_algo);
        let achieved = match new_block.header.pow_algo() {
```

© Coinspect 2024

```
            PowAlgorithm::RandomX =>
randomx_difficulty(&new_block.header, &self.randomx_factory)?,
            PowAlgorithm::Sha3x =>
sha3x_difficulty(&new_block.header)?,
        };
        if achieved < min_difficulty {
            self.blockchain_db
                .add_bad_block(
                    new_block.header.hash(),

self.blockchain_db.get_chain_metadata().await?.height_of_longest_chain(
),
                )
                .await?;
            return Err(CommsInterfaceError::InvalidBlockHeader(

BlockHeaderValidationError::ProofOfWorkError(PowError::AchievedDifficul
tyBelowMin),
            ));
        }
        Ok(())
    }
```

Recommendation

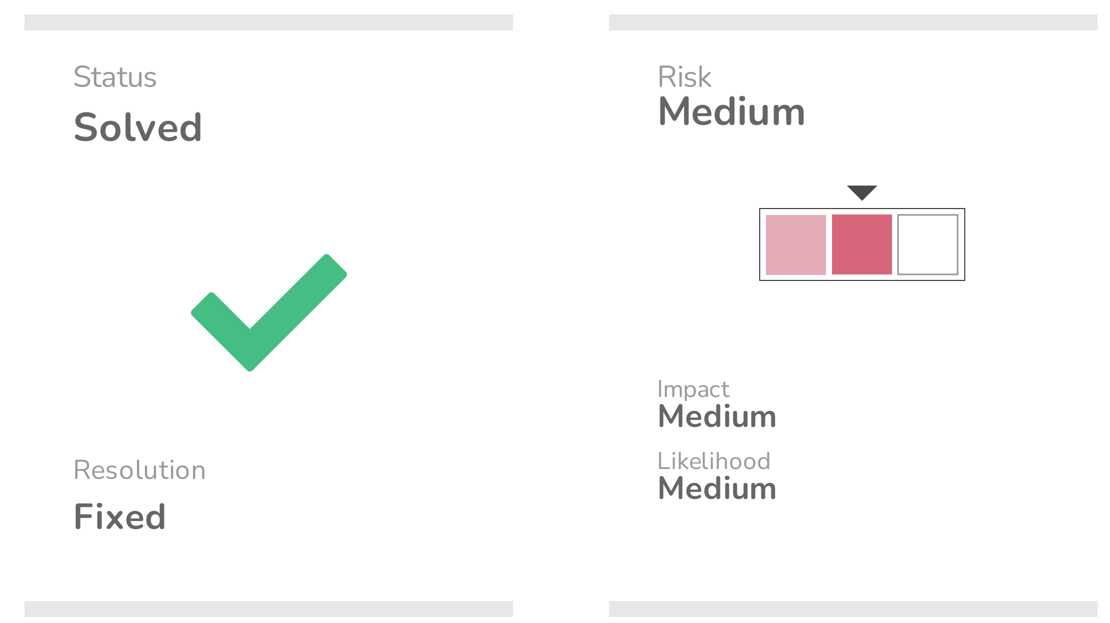
Ban nodes that send blocks with a lower difficulty that the minimum.

Status

Fixed. The `check_min_block_difficulty` has improved so it uses a dynamically adjusted minimum difficulty and bans nodes that send garbage blocks.

TARI-042

Attackers can hold the DB write lock with low difficulty blocks



Location

`base_layer/core/src/base_node/comms_interface/inbound_handlers.rs`

Description

An attacker can get a hold of the DB write lock by sending blocks that do not meet the minimum difficulty.

The cause is that adding the blocks to the bad block list requires the node to acquire a lock for writing in the DB. While the time that lock is held is very short, an attacker can continuously send messages and maximize the lock time to prevent other writes to the database.

Recommendation

© Coinspect 2024

Do not add blocks that do not meet the minimum difficulty to the bad block list. This poses no risk, as producing blocks with low PoW is easy for attackers so banning them does not hamper attackers.

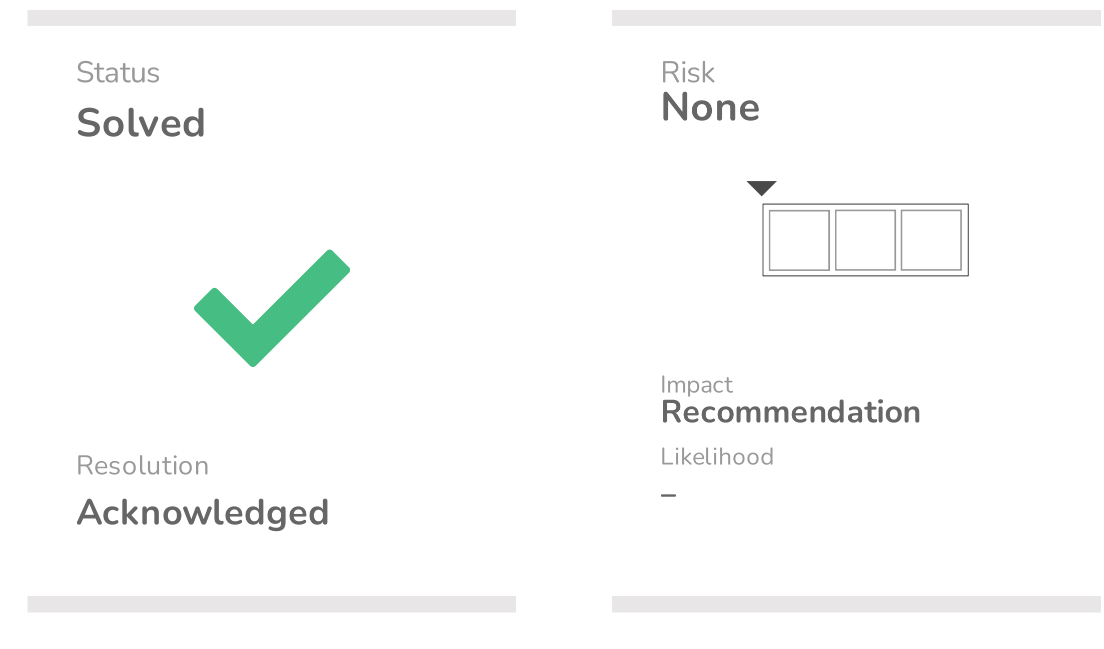
Status

Fixed. Low difficulty blocks are not added to the bad block list.

© Coinspect 2024

TARI-043

Wallet can minimize trust with base node by requesting proof of work



Location

Description

The wallet consumes new blocks from a base node set by the user. However, it fully trusts the node to provide valid main-chain blocks, as it performs validations only on the transactions received:

```
            BaseNodeEvent::NewBlockDetected(_hash, height) => {
                let _operation_id = self
.start_transaction_validation_protocol(transaction_validation_join_hand
les)
                    .await
                    .map_err(|e| {
                        warn!(target: LOG_TARGET, "Error validating
txos: {:?}", e);
                        e
                    });
```

```
                self.last_seen_tip_height = Some(height);
            },
```

Recommendation

The relationship between the wallet and the base node could be trust-minimized by making the wallet check the proof of work of blocks provided by the base node.

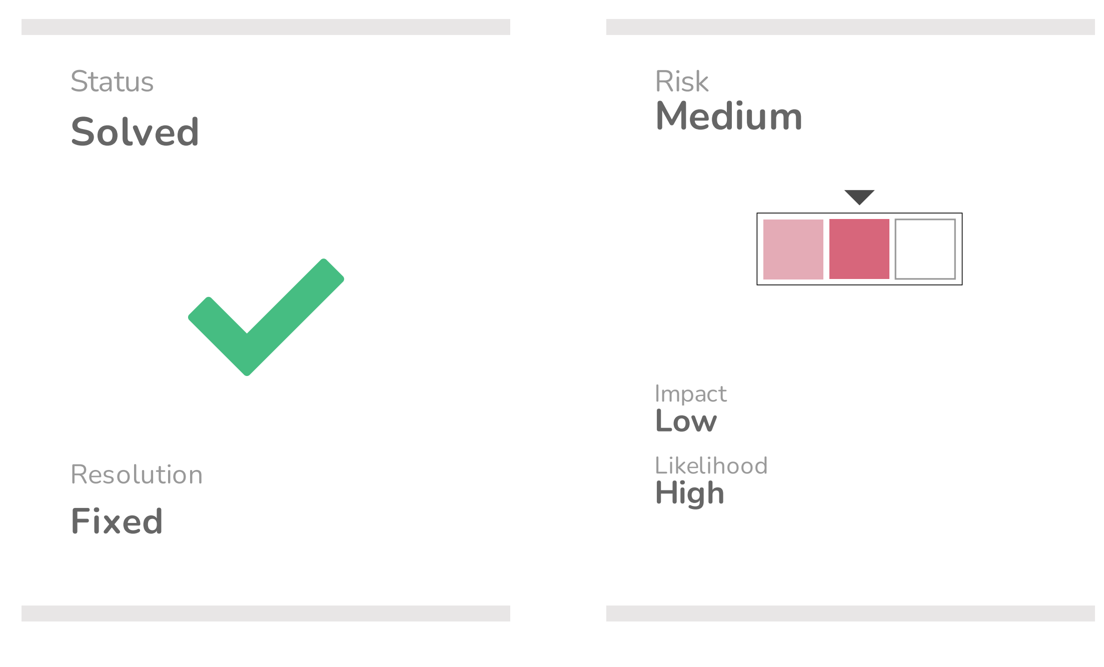
Status

Tari has stated that it is assumed that the base node is trusted. In particular, Tari said:

```
The current wallet design is created with the assumption that base
nodes are trusted by the wallet. There is no alternative model at the
moment, but is something that could be for future development.
```

TARI-044

Attackers can flood victim with peer addresses



Location

`comms/core/src/peer_manager/peer_storage.rs`

Description

An attacker can continuously and quickly announce new addresses to a victim, flooding a victim with `Join` messages and making the `addresses` vector of the victim grow indefinitely. The attacker can thus make the victim waste CPU time and storage.

The storage use is negligible due to the limit on the amount of addresses sent on each `Join` message imposed by the fixes on issue `TARI-021`. Nevertheless, the computing time is asymmetrical, as the attacker uses less computing time to send messages than the victim to process them.

Processing the addresses grows quadratically in the number of values sent, because the `merge` procedure checks if each address is already known

© Coinspect 2024

```
    pub fn merge(&mut self, other: &MultiaddressesWithStats) {
        for addr in &other.addresses {
            if let Some(existing) =
self.find_address_mut(addr.address()) {
                existing.merge(addr);
            } else {
                self.addresses.push(addr.clone());
            }
        }
    }

    /// Finds the specified address in the set and allow updating of
its variables such as its usage stats
    fn find_address_mut(&mut self, address: &Multiaddr) -> Option<&mut
MultiaddrWithStats> {
        self.addresses.iter_mut().find(|a| a.address() == address)
    }
```

This process is protected from being multi-threaded by a lock, which further slows down the processing of the flood of messages.

```
    pub async fn add_peer(&self, peer: Peer) -> Result<PeerId,
PeerManagerError> {
        let t = Instant::now();
        let mut lock = self.peer_storage.write().await;
        let peer_id = lock.add_peer(peer)?;
        #[cfg(feature = "metrics")]
        {
            let count = lock.count();
            metrics::peer_list_size().set(count as i64);
        }
        Ok(peer_id)
    }
```

Any node that connects to the victim can be automatically attacked by the attacker, as the victim will share the attacker as a normal, benign peer.

Recommendation

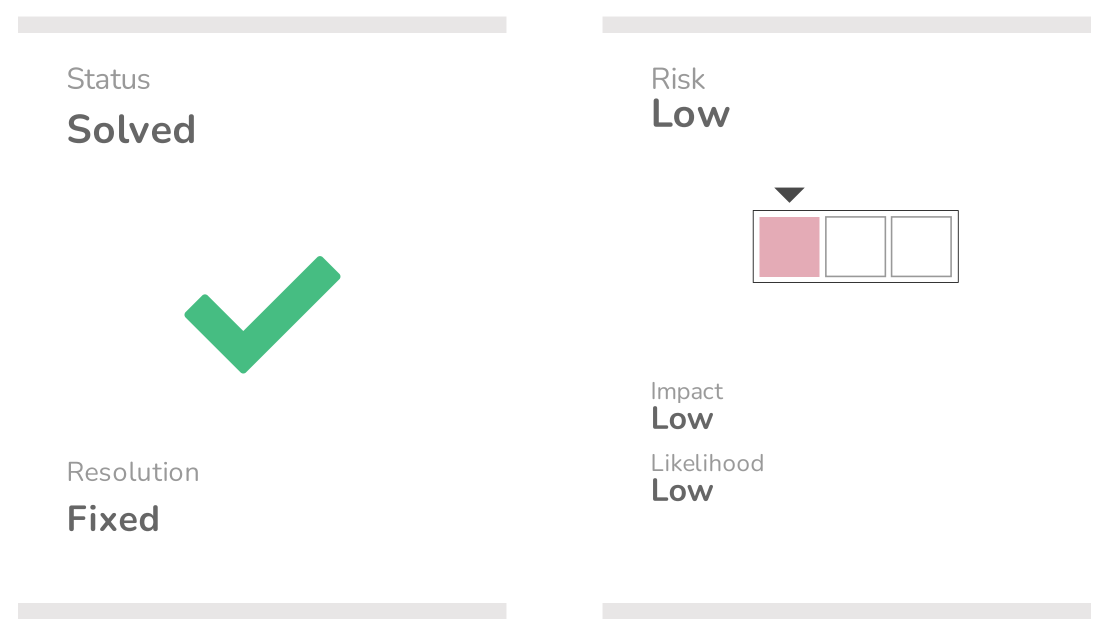
Limit the amount of addresses a peer can have. Implement a rate limit mechanism for all messages.

Status

The `merge` method now limits the amount of addresses a peer has which is enough to render the impact of the issue negligible.

TARI-045

Some errors are not obscured by gRPC server



Location

`applications/minotari_node/src/grpc/base_node_grpc_server.rs`

Description

Attackers can get information about the base node via gRPC errors, even when the user has set the configuration to obscure the errors.

The particular method that fails to obscure the errors is `get_new_block_blob`, where the `obscure_error_if_true` call is missing:

```
        let new_block = match
handler.get_new_block(block_template).await {
            Ok(b) => b,

Err(CommsInterfaceError::ChainStorageError(ChainStorageError::InvalidAr
guments { message, .. })) => {
                return Err(Status::invalid_argument(message));
```

© Coinspect 2024

```
            },

Err(CommsInterfaceError::ChainStorageError(ChainStorageError::CannotCal
culateNonTipMmr(msg))) => {
                let status = Status::with_details(
                    tonic::Code::FailedPrecondition,
                    msg,
                    Bytes::from_static(b"CannotCalculateNonTipMmr"),
                );
                return Err(status);
            },
            Err(e) => return Err(Status::internal(e.to_string())),
        };
```

Errors related to the serialization of the blob also will not be obscured:

```
        let mut header_bytes = Vec::new();
        BorshSerialize::serialize(&header, &mut
header_bytes).map_err(|err| Status::internal(err.to_string()))?;
```

There are also other inconsistencies in the usage of the `obscure_error_if_true` method. For example, the majority of invalid arguments are not obscured, possible to let the requester know what they must correct for their request to succeed. But *some*, like errors related to the public key serialization in `get_shard_key`, are obscured:

```
        let public_key = PublicKey::from_bytes(&request.public_key)
            .map_err(|e| obscure_error_if_true(report_error_flag,
Status::invalid_argument(e.to_string())))?;
```

Recommendation

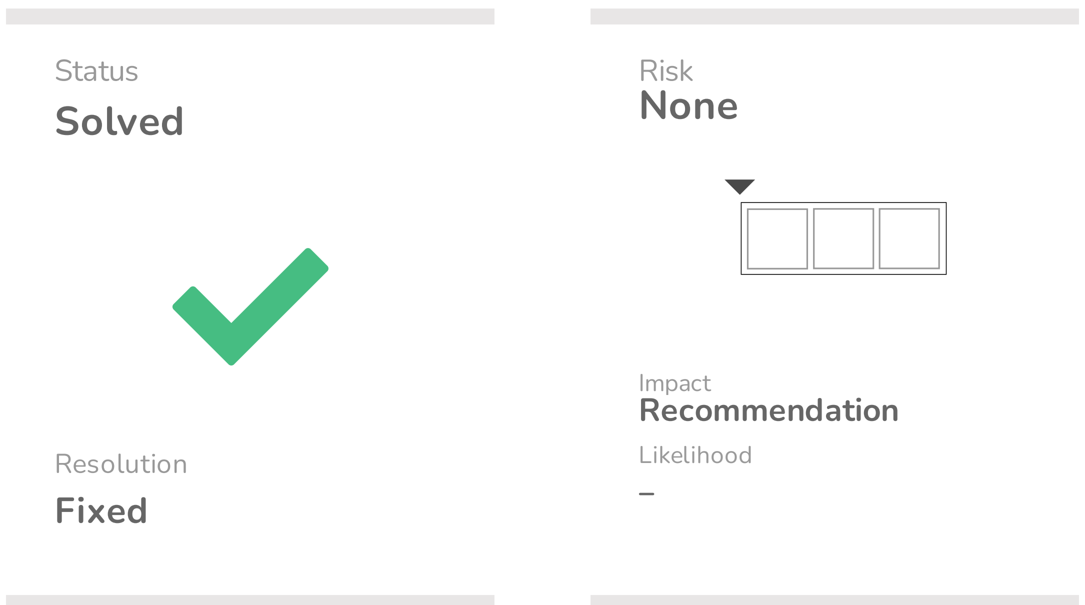
Document which errors are obscured and which are not. Make sure all errors follow the documented policy.

Status

Fixed. Errors are obscured now all the time if the user configured the node to do so.

TARI-046

Not possible to disable gRPC methods



Description

gRPC server is either enabled or disabled, with no way to filter which methods the node or wallet operator wants to expose.

While this can be resolved by the use of a reverse-proxy, it is recommended that the configuration exists to protect unaware operators. In particular, certain methods should be disabled by default unless the operator explicitly requests them, because they provide information about a node that in most cases should be private, such as:

- `check_for_updates`
- `get_version`
- `list_connected_peers`
- `get_sync_progress`
- `get_network_status`
- `get_peers`

Recommendation

Add the capability to disable or enable gRPC methods. Disable methods that provide internal information of the node by default.

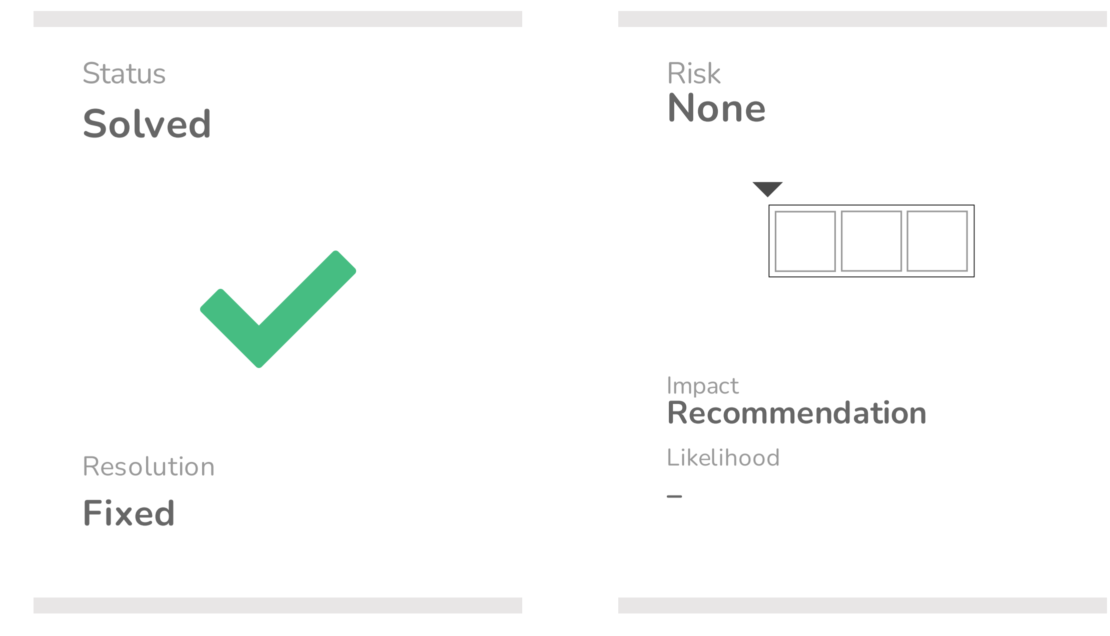
Status

Tari implemented this recommendation and now sensitive methods are disabled by defaults.

© Coinspect 2024

TARI-047

Inaccuracy on pruning on RFC-0140



Location

Description

RFC-0140 states that:

```
When running in pruning mode, Base Nodes MUST remove all spent outputs
that are older than the pruning horizon in their current stored UTXO
set when a new block is received from another Base Node.
```

Nevertheless, pruning code does not follow exactly this. Base nodes remove spent outputs (and inputs) when receiving a new block only after a certain `pruning_interval` has passed. This makes it possible for a pruning node to actually have full blocks available that are older than their `pruning horizon`.

```
    if metadata.pruned_height() <
abs_pruning_horizon.saturating_sub(pruning_interval) {
        prune_to_height(db, abs_pruning_horizon)?;
    }
```

© Coinspect 2024

Recommendation

Amend to RPC to reflect actual base node behavior.

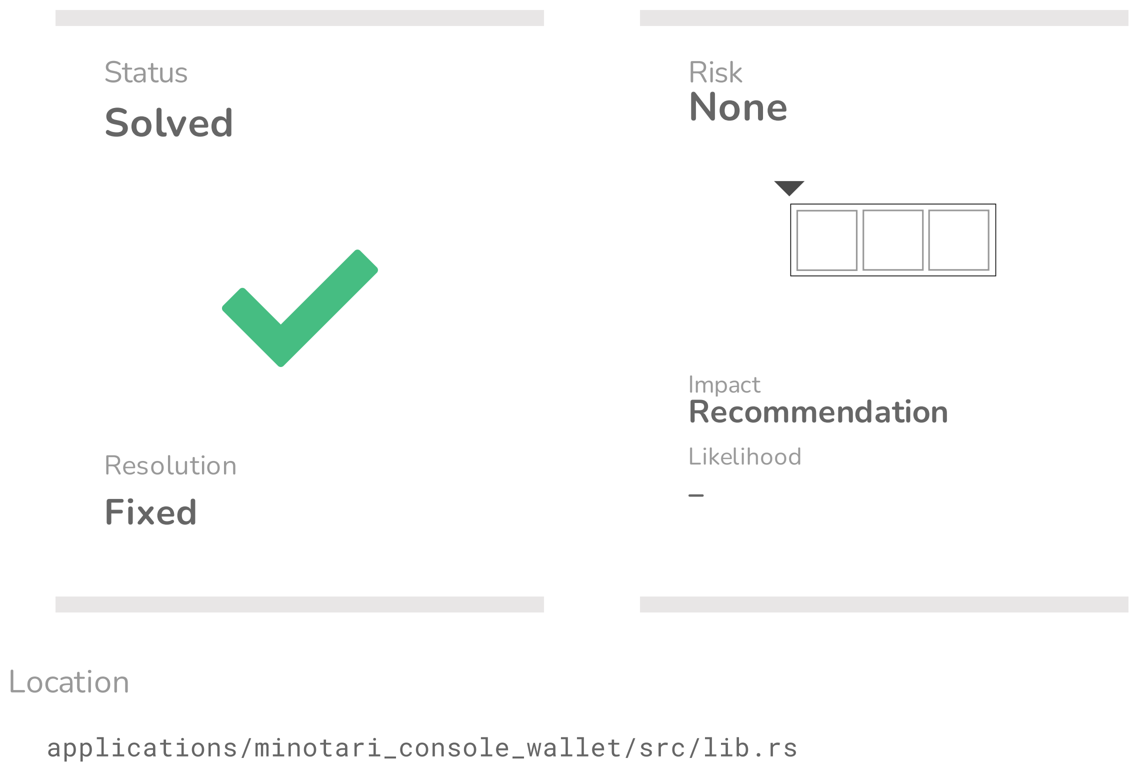
Status

Fixed. Tari amended their RPC to read `SHOULD` instead of `MUST`.

© Coinspect 2024

TARI-048

No warning when using weak or empty password on account recovery



Location

`applications/minotari_console_wallet/src/lib.rs`

Description

When recovering an account from a seed phrase, the console wallet does not analyze the strength of the password set to protect the seed phrase. Thus, the wallet does not warn the user if providing an empty or weak password.

```
    let password = match boot_mode {
        WalletBoot::New => {
            // Get a new passphrase
            debug!(target: LOG_TARGET, "Prompting for passphrase.");
            get_new_passphrase("Create wallet passphrase: ", "Confirm
wallet passphrase: ")?
        },
        WalletBoot::Existing | WalletBoot::Recovery => {
            debug!(target: LOG_TARGET, "Prompting for passphrase.");
            prompt_password("Enter wallet passphrase: ")?
```

© Coinspect 2024

```
        },
    };
```

Recommendation

Execute the `display_password_feedback` function on the password used to protect the seed phrase to be recovered.

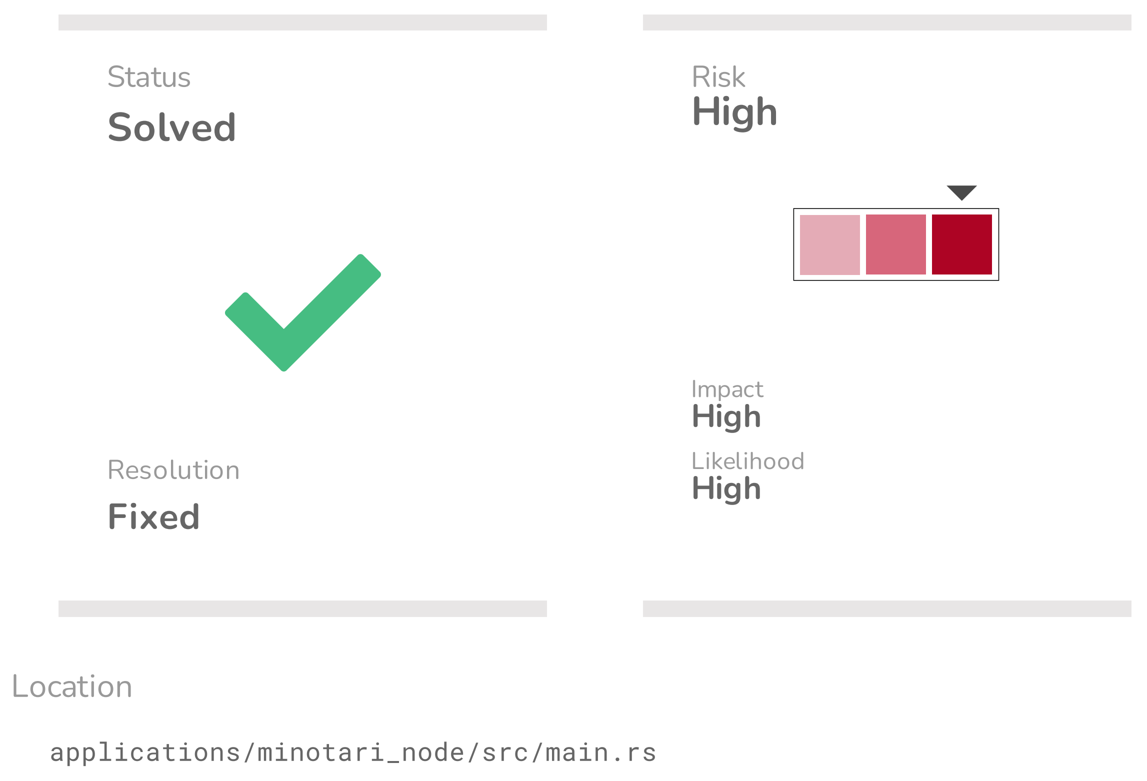
Status

Fixed. The `Recovery` flow now executes `get_new_passphrase()`, which will in turn execute `display_password_feedback`. /--

© Coinspect 2024

TARI-049

Attackers can crash whole node by panicking a single thread



Location

`applications/minotari_node/src/main.rs`

Description

A panic on a thread will cause the whole process to be aborted. This happens both in the wallet and in the base node.

```
fn main() {
    // Setup a panic hook which prints the default rust panic message
but also exits the process. This makes a panic in
    // any thread "crash" the system instead of silently continuing.
    let default_hook = panic::take_hook();
    panic::set_hook(Box::new(move |info| {
        default_hook(info);
        process::exit(1);
    }));
```

© Coinspect 2024

This design decision allowed several vulnerabilities in this report to be of critical severity. For example, panics on RPC requests would have practically no impact when their thread panics instead of being a tool for triggering a denial of service on the whole network.

Panics on threads should be treated as bugs and fixed, but an attacker would have a much harder time converting them to successful exploits if the program did not crash immediately.

Recommendation

Panics must be handled more gracefully. For example, a panic during a block processing should end with the ban of the peer that sent and the invalidation of the block, instead of closing the program.

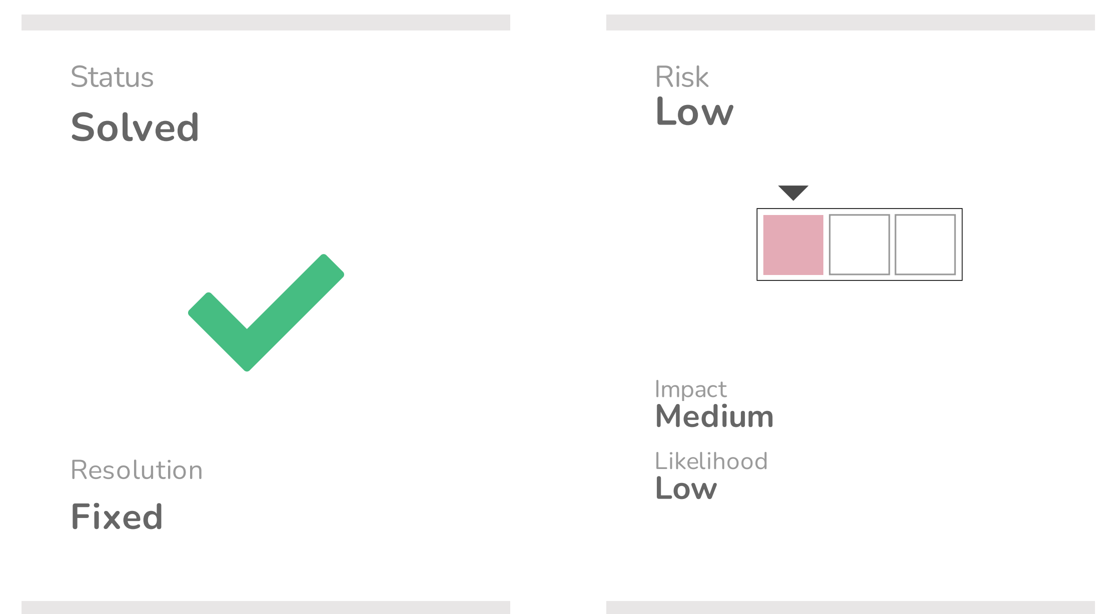
Status

Fixed. A panicking thread will now not crash the whole application.

© Coinspect 2024

TARI-050

Privacy compromise fetching code template repository URL



Location

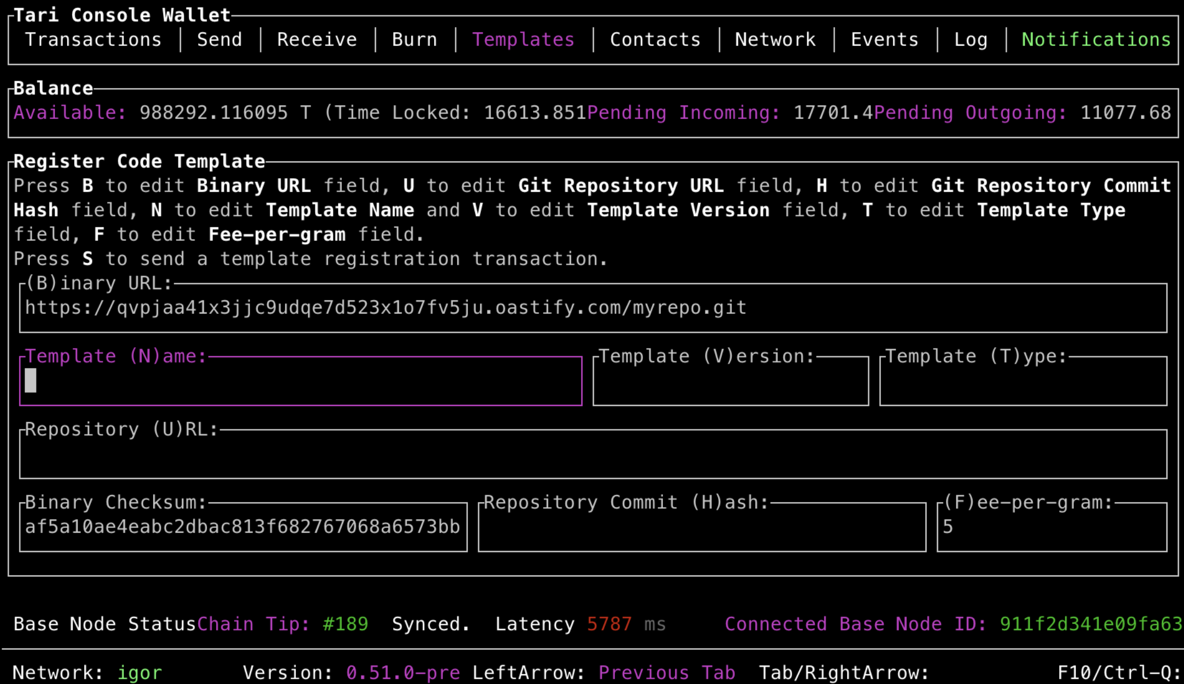
```
applications/tari_console_wallet/src/ui/components/register_template_t
ab.rs:538
```

Description

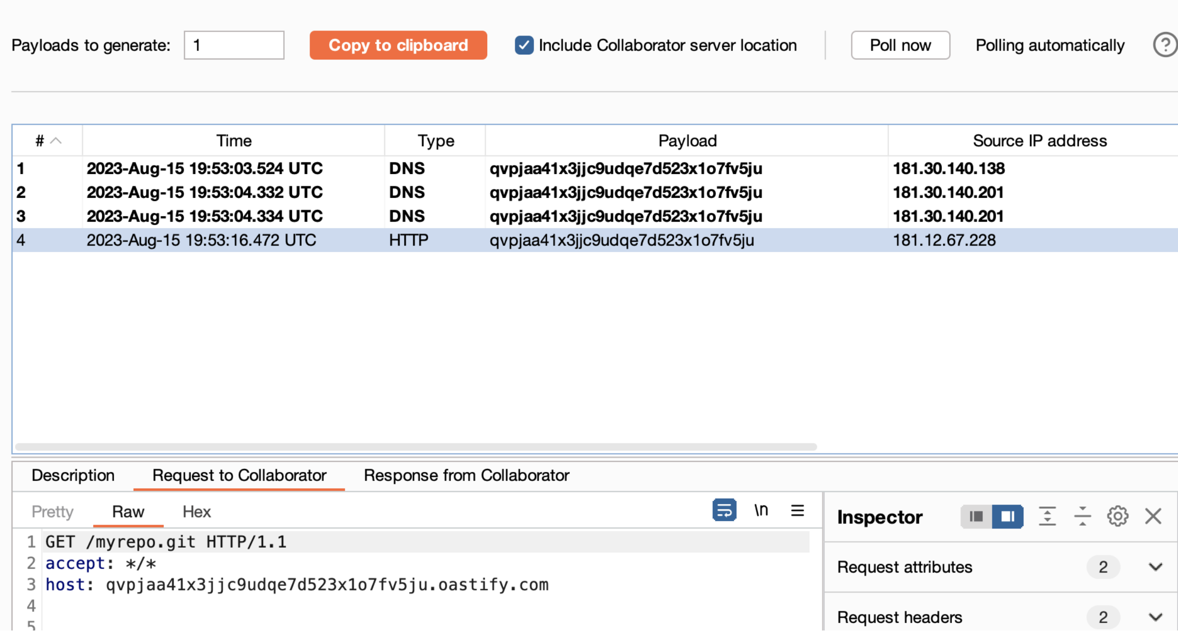
In the console wallet's code template registration functionality, the URL data will be automatically fetched to compute its hash. In an scenario where an adversary controls the site hosting the repository fetched by the wallet, they can reveal the IP associated with the Tari address that registered the code template, compromising the user's privacy.

Once the user completes the `(B)inary URL:` field in the screen below:

© Coinspect 2024



The wallet will issue a GET request to this URL:



Due to the `reqwest::get(url)` line from the following code block:

```
let data = reqwest::get(url).await;
match data {
    Ok(data) => match data.status() {
        StatusCode::OK => match data.bytes().await {
            Ok(bytes) => {
                let mut hasher = Blake256::new();
                hash_domain!(TariEngineHashDomain,
"com.tari.dan.engine", 0);
                TariEngineHashDomain::add_domain_separation_tag(&mut
hasher, "Template");
                let hash: [u8; 32] =
```

```
hasher.chain(bytes).finalize().into();
                    hex_string = hash.to_hex();
               },
               Err(e) => {
                    error = Some(format!("Error {:?}\nPress Enter to
continue.", e));
               },
          },
```

Recommendation

Route the GET request via a Tor SOCKS proxy.

Otherwise, clearly communicate users the privacy risks involved, and urge them to use only trusted URLs, avoiding those provided by third parties.

Status

Fixed. A warning stating that the feature should only be used with trusted URLs has been added.

© Coinspect 2024

Disclaimer

The information presented in this document is provided "as is" and without warranty. The present security audit does not cover any on-chain systems or frontends that communicate with the network, nor the general operational security of the organization that developed the code.

© Coinspect 2024