Tinyman Governance Protocol

Security Audit

April 11, 2024

Prepared for:
Tinyman

Prepared by:
Kevin Wellenzohn
Blockshake

BLOCK
SHAKE

Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only, and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of the contract.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive, and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds

Introduction

The Tinyman team has asked Blockshake as well as a number of other well-known companies and individuals in the Algorand ecosystem to audit their upcoming governance system in a peer reviewing process. We accepted this assignment and worked along with the following partners to audit Tinyman's contracts:

- Folks Finance
- Vestige Labs
- Steve Ferrigno
- d13

Each auditor reviewed the code on its own and chose autonomously how to audit the code, we discuss further below our own methodology. Findings were discussed openly and shared immediately with the whole group. Weekly meetings were held to monitor progress and discuss open issues. The audit was conducted between the end of February and beginning of April.

Scope

In-Scope

The audit was conducted on commit 9d116ff75f3bfcd54d59fecaa287b5f63c2f4b77 with a focus on the following source code files:

- `contracts/proposal_voting/proposal_voting_approval.tl`
- `contracts/proposal_voting/proposal_voting_clear_state.tl`
- `contracts/rewards/rewards_approval.tl`
- `contracts/rewards/rewards_clear_state.tl`
- `contracts/staking_voting/staking_voting_approval.tl`
- `contracts/staking_voting/staking_voting_clear_state.tl`
- `contracts/vault/vault_approval.tl`
- `contracts/vault/vault_clear_state.tl`

It was stated by the Tinyman team that this code will be open sourced upon launch of the contracts.

We were provided with supplementary documentation:

- A system description, presented as Google Docs document, that describes the overall architecture and each individual contract.
- A whiteboard on excalidraw.com that explains the math that underpins the contracts

Out-of-Scope

The above source code files are written in Tealish, a programming language developed by the Tinyman team. We did not inspect the TEAL files that Tealish generates, nor did we attempt to verify that the Tealish compiler outputs valid TEAL code. This aspect was covered by a different auditor.

Tinyman intends to extend the proposal voting contract such that proposals that are approved by governors can automatically be executed by so called *executors*. The current code lays the groundwork for executors, but the executors are not yet implemented. It is intended that future upgrades to the proposal voting contract implement the missing pieces. The executors are thus not within the scope of this audit.

We did not audit the web application / user interface that governors will use to call the audited contracts.

Methodology

Exploration

We started the audit with a thorough review of the provided documentation and discussions with the Tinyman team to understand the system's overall expected functionality and design. Since the system consists of several smart contracts, we looked at how the contracts interact with each other and what dependencies exist between them.

Manual code review

We analyzed each smart contract and focused on the following areas that are a common source of bugs and vulnerabilities:

- **Data storage**: all contracts extensively use global and box storage. The contracts implement arrays on top of box storage that store lists of other data structures (proposals, votes, etc.). We checked that this array abstraction is implemented correctly, which means that no buffer overflows occur that could inadvertently overwrite other items, that items are accessed & manipulated only by the intended users, etc.

- **Arithmetic overflow / underflow**: Algorand smart contracts fail upon integer overflows or underflows in arithmetic operations. This can block contracts and lock up funds if exploited by attackers. We checked all arithmetic operations and the quantities they operate on to detect any potential overflows or underflows. We computed the ranges of all important quantities (e.g., TINY power) to see if they can be represented within their allotted integer type (e.g., `uint64`).
- **Interactions between contracts**: The contracts interact with each other through inner transactions. We checked that the calling contract and the called contract (caller & callee) are in the correct state, accept each other's calls, and return the correct result.
- **Permissioned methods**: some of the contracts have permissioned methods that can only be called by specific users / Algorand accounts. We checked that these methods can indeed only be called by the appropriate accounts.
- **Upgradeable contracts**: some of the contracts can be changed. It was checked that (a) changing the contract can only be performed by a permissioned account, and (b) if upgrading a contract is necessary for the system. For example, some of Tinyman's contracts are by design upgradable to support future governance votes.

Besides the above general security checks, we studied each individual contract in depth, looking for the presence or absence of code that could lead to a vulnerability. Depending on a contract's functionality, we focused on different areas. For example, in the **proposal voting** contract, we focused on the lifecycle of a proposal and how it transitions between its various states (created, voted, executed, etc.). In the **rewards** contract we checked that governors receive their appropriate rewards, and in the **vault** contract we checked that governors are able to reclaim their locked tokens after the correct time.

Weekly meetings

Throughout the audit, we kept a tight feedback loop with the Tinyman team and the other independent auditors to discuss any open questions, findings, etc.

Findings

We first start with general findings that apply to multiple or all contracts and later we discuss findings that are specific to an individual contract.

Note that we do not include findings in this report that other parties in this collaborative peer reviewing process have previously uncovered. We refer the reader to the respective audits to get a comprehensive picture of the auditing process.

Summary

The audited code is of high quality as is evident by the fact that most findings in this report are purely informational and do not have any security implications. The code is well structured and extensively tested. The provided documentation was detailed and helpful, albeit not always one hundred percent accurate.

We use the severity classification as defined by Trail of Bits in this report, which is given in the Appendix.

General

F1. Centralization concerns

Severity: Informational

Three of the four audited contracts have permissioned methods and can be upgraded by a manager account, namely the staking voting, proposal voting, and rewards contracts. Only the vault contract is immutable.

Impact

Upgradable contracts allow certain accounts (in this case a manager account operated by the Tinyman team) to change the code of the contracts after they have been deployed. The changed code can slightly or completely change how a contract works and hence this is a big attack surface.

We note that upgradable smart contracts and permissioned methods are only used in places where this is a stated design goal. It is the intention of Tinyman governance that governors can vote on future updates to the contracts that allow new behavior.

The vault contract that holds the users' TINY tokens is immutable, which means it is impossible that a future contract upgrade could lock users out from their tokens.

Response

It is important that the manager accounts are properly secured. The Tinyman team has stated that they intend to use a ⅗ multisig account where all the individual accounts are backed by hardware ledgers. This is considered a highly secure setup that protects from malicious actors from the outside and inside.

F2. Inconsistent use of data types

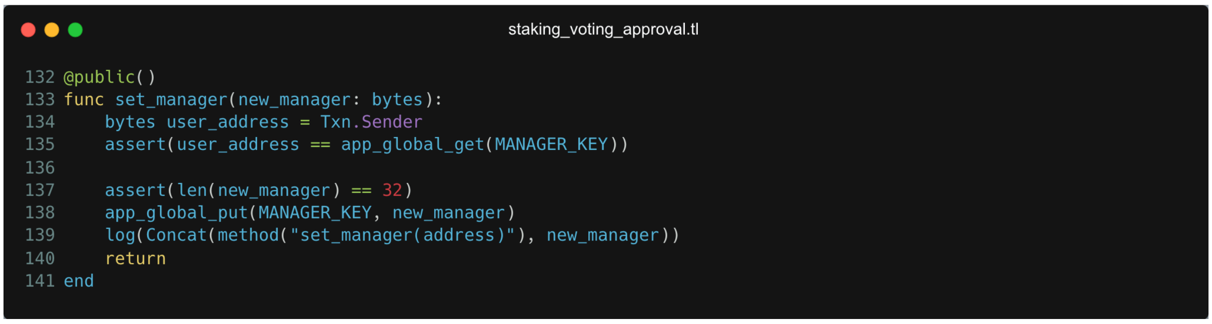
Severity: Informational

The code is not consistent in its use of data types for various variables and arguments. Tealish supports multiple data types in its type systems like fixed- or variable-length byte arrays, integers, addresses etc. We observed that the code often uses weaker or more generic data types than the code actually expects. In the following we give two examples of this.

Show case #1.

In several cases there are functions that take variable-length byte arrays (type `bytes`) as input where fixed-length arrays are expected. The code then performs an explicit length-check on the provided argument.

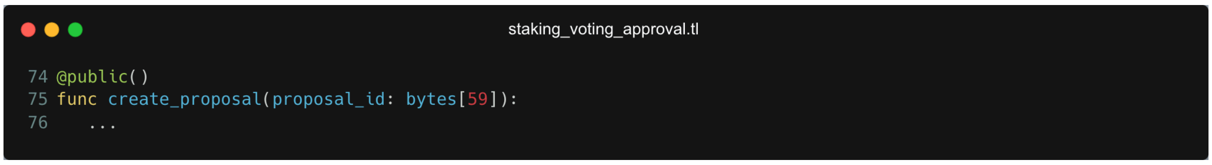
Consider the function `set_manager` in the staking voting contract that takes an Algorand account address as input, using `bytes` as input type. Later, the code checks that the provided address is 32 bytes long. Instead of this explicit assertion, the function signature could specify the `bytes[32]` data type, which forces the compiler to include an implicit runtime check that the provided argument is indeed 32 bytes long.


staking_voting_approval.tl

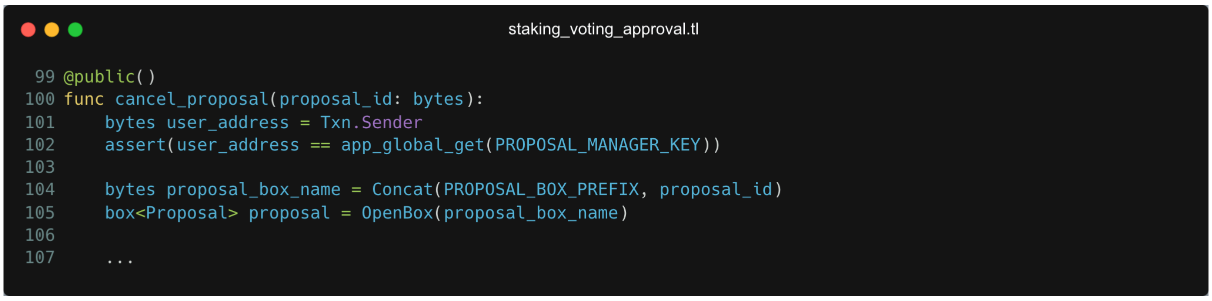
```
@public()
func set_manager(new_manager: bytes):
    bytes user_address = Txn.Sender
    assert(user_address == app_global_get(MANAGER_KEY))

    assert(len(new_manager) == 32)
    app_global_put(MANAGER_KEY, new_manager)
    log(Concat(method("set_manager(address)"), new_manager))
    return
end
```

Show case #2.

The following screenshots show two functions from the staking voting contract that create and cancel a proposal, respectively. The function to create the proposal enforces in the type signature that the proposal ID must be 59 bytes long, while the second function does not check the length of the proposal ID. However, this is not a problem since a proposal can only be canceled if a valid (i.e., previously existing) proposal ID is provided.


staking_voting_approval.tl

```
@public()
func create_proposal(proposal_id: bytes[59]):
    ...
```


staking_voting_approval.tl

```
@public()
func cancel_proposal(proposal_id: bytes):
    bytes user_address = Txn.Sender
    assert(user_address == app_global_get(PROPOSAL_MANAGER_KEY))

    bytes proposal_box_name = Concat(PROPOSAL_BOX_PREFIX, proposal_id)
    box<Proposal> proposal = OpenBox(proposal_box_name)

    ...
```

Impact

The code diligently and correctly checks the size of input arguments, but it does use different approaches to do so. Sometimes length checks are enforced as part of the type signature, sometimes they are performed as explicit length-checks. None of the studied cases poses a security problem, but it is nevertheless recommended that length-checks are done consistently in the same manner across the code base.

Response

This comment was addressed by the Tinyman team in commit: [cdef5159371a6c116e26982379a76d60182124d1]()

F3. Code duplication

Severity: Informational

The audited contracts contain several instances of duplicated code. In some cases code is copied one to one across contracts and in some cases slightly modified.

A non-exhaustive list of examples of duplicated code are:

- The `transfer` function in the rewards and vault contracts
- The `check_and_set_user_as_voted` function in the staking & proposal voting contracts
- The `increase_budget` function in all but the vault contract
- The `get_box` function in all contracts
- The way indexes for arrays (modeled on top of box storage) are computed is basically always the same but is re-implemented for every box type.
- ...

Impact

The duplicated code in the audited contracts is not a problem and was always modified correctly if necessary. We understand that Tealish lacks the ability to package code in libraries and that this is the reason why code had to be copied. However, in our experience, copying code is a frequent source of bugs. In addition, avoiding code duplication means that code needs to be checked only once as opposed to multiple times. For these reasons we recommend removing code duplication as much as possible.

Response

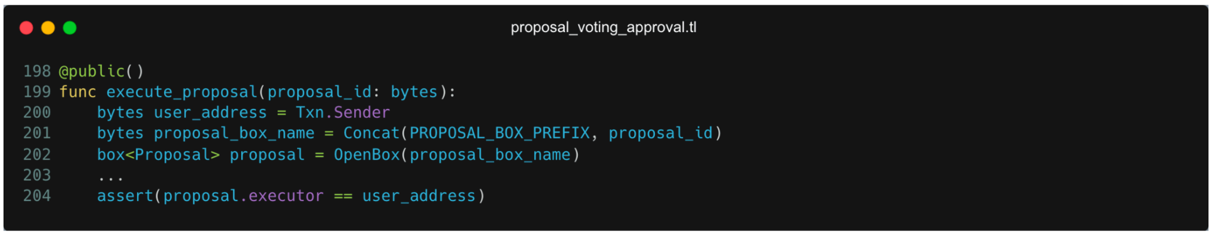
Tinyman acknowledged the existence of duplicate code but chose not to make significant changes to the code given that no security problem was identified.

F4. Discrepancies between documentation & code

Severity: Informational

There are several instances where the code and the provided documentation do not match one another. Examples are missing arguments or return values, wrong function names, etc. While most discrepancies are completely harmless, some imply differences in business logic.

For example, the `execute_proposal` method in the proposal voting contract can only be called by the executor account that was specified when the proposal was created (see the following screenshot). The documentation instead says “This is a permissioned method, accessible exclusively to the *proposal manager*”.



```
@public()
func execute_proposal(proposal_id: bytes):
    bytes user_address = Txn.Sender
    bytes proposal_box_name = Concat(PROPOSAL_BOX_PREFIX, proposal_id)
    box<Proposal> proposal = OpenBox(proposal_box_name)
    ...
    assert(proposal.executor == user_address)
```

Impact

There is no security impact, but it is recommended that the documentation is updated and aligned with the code if the documentation is published.

Response

Tinyman acknowledged this finding and stated that the documentation will be updated.

Vault Contract

F5. Deleting boxes & account state leads to loss of rewards, voting power

Severity: Informational

The vault contract records each action that a governor performs with respect to her locked amount of TINY tokens (e.g., creating a lock, extending the lock, adding more TINY tokens, etc). These actions are stored in box storage and the contract computes from this data the governor's TINY power at a given point in time. All other contracts depend on a governor's TINY power to determine rewards, voting power etc.

The vault contract provides two destructive functions, `delete_account_power_boxes` and `delete_account_state`, that delete a governor's boxes & state, respectively. Function `delete_account_state` can only be called when the user has not locked any TINY tokens at the moment, but `delete_account_power_boxes` can be called at any time.

Impact

Deleting account power boxes means that a governor loses out on rewards and voting power, but the documentation does not mention that. It should be absolutely clear to a governor what consequences these actions have. Therefore, it is recommended that the documentation is updated to reflect this.

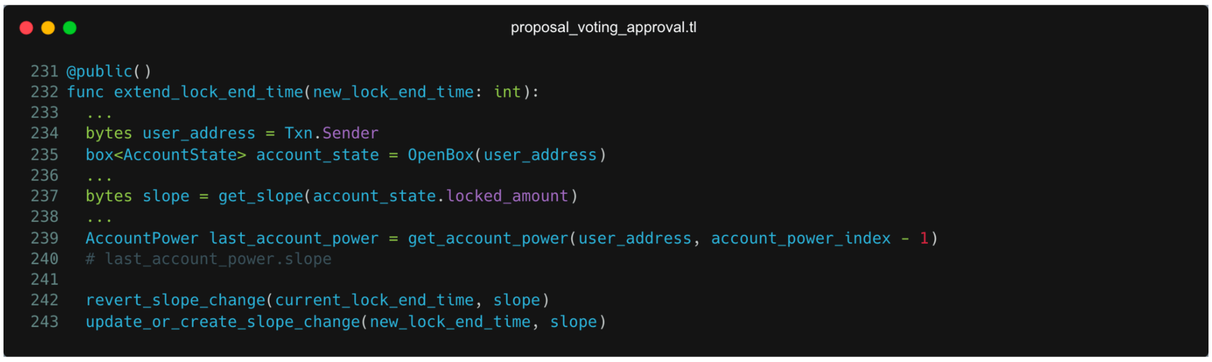
Response

The Tinyman team stated that their user interface will make it difficult to call these destructive functions and that the consequences will be made clear to the governor.

F6. Redundant slope computation

Severity: Informational

Function `extend_lock_end_time` can be used to extend an existing lock and thus gain more TINY power. The code computes the current `slope` from the currently locked amount of TINY tokens as stored in the account state. The function also retrieves the last account power box, which already stores the current slope. Thus, the first slope computation is redundant.


proposal_voting_approval.tl

```
@public()
func extend_lock_end_time(new_lock_end_time: int):
  ...
  bytes user_address = Txn.Sender
  box<AccountState> account_state = OpenBox(user_address)
  ...
  bytes slope = get_slope(account_state.locked_amount)
  ...
  AccountPower last_account_power = get_account_power(user_address, account_power_index - 1)
  # last_account_power.slope

  revert_slope_change(current_lock_end_time, slope)
  update_or_create_slope_change(new_lock_end_time, slope)
```

Impact

This has no security relevance, but reusing the previous slope value has two advantages: (a) it avoids recomputing the slope which is more computationally expensive as it uses wide integer math and (b) using the same value that was computed earlier makes the intention of the code clearer to revert the current slope.

Response

Tinyman acknowledged this finding but chose not to change the math in a critical section of the code.

Proposal Voting Contract

No findings to report.

Rewards Contract

F7. TINY Funding

Severity: Informational

The rewards contract distributes TINY tokens as rewards for locking TINY tokens in the vault contract. The amount of TINY token to be distributed depends on the allocated amount for the reward period and the TINY power that the governor holds during that time. The rewards contract can only hand out TINY tokens as rewards if it has a sufficient number of tokens in reserve.

Impact

If the rewards contract is not (sufficiently) funded, governors cannot claim their earned rewards.

Response

The Tinyman team stated that they intend to transfer all TINY tokens that are dedicated to staking rewards according to their tokenomics to the rewards contract when it is deployed.

Staking Voting Contract

Preface

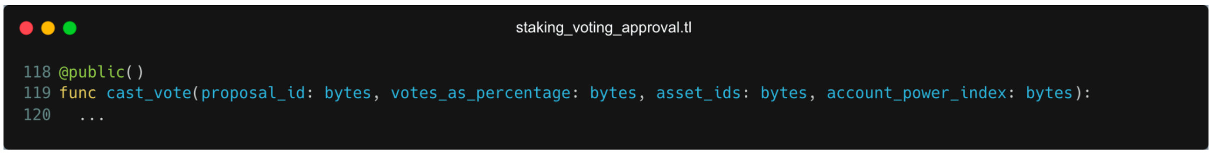
The staking voting contract is designed to work along Tinyman's current staking farms that are managed off-chain. As a result, this contract is mainly used for bookkeeping purposes: creating proposals, tracking their votes, etc., but it cannot implement / execute proposals.

F8. Voting on invalid asset IDs

Severity: Informational

Users can allocate their voting power to liquidity pool (LP) tokens of Tinyman farms. In the corresponding application call they need to specify the asset IDs of the LP tokens as well as the percentage of their vote that they want to allocate to these specific tokens. As the following screenshot shows, the asset IDs as well as the percentages are represented as a `bytes` array that encodes `N` 64-bit integers each (hence the arrays must be of length `N * 8`).

The code does not check if the asset IDs are valid Tinyman LP tokens, or indeed any asset. It is possible that a governor votes on a non-existent token.



```
@public()
func cast_vote(proposal_id: bytes, votes_as_percentage: bytes, asset_ids: bytes, account_power_index: bytes):
    ...
```

Impact

A governor can burn or waste her voting power on a non-existing asset ID. This only harms the governor herself but not the overall system. The Tinyman team assured us that their user interface will not allow casting a vote on a non-existent token.

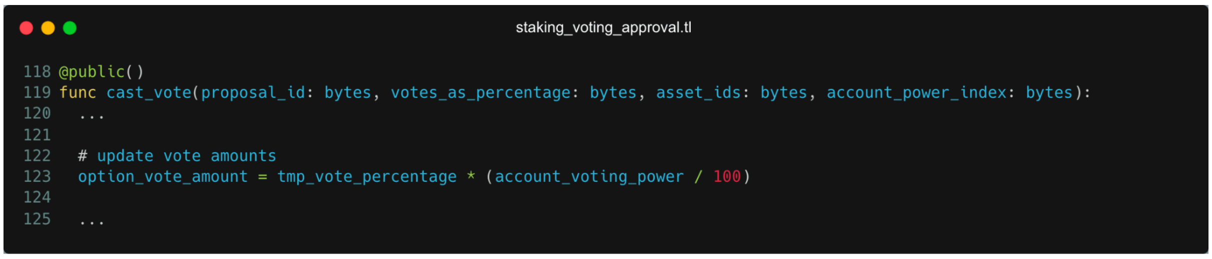
Response

Tinyman acknowledged this finding but chose not to change the system because governors can only hurt themselves.

F9. Rounding when casting a vote

Severity: Low	Difficulty: Low

As explained in the previous finding, with `cast_vote` a governor can allocate a percentage of her voting power at the time the proposal was created to a particular LP token. The following screenshot shows the code that computes the voting power for a given LP token. The `tmp_vote_percentage` is a number between 1 and 100 and represents the percentage of the vote that is attributed to a given LP token. The computation is such that the account voting power is first divided by 100 and then multiplied by the percentage.



```
@public()
func cast_vote(proposal_id: bytes, votes_as_percentage: bytes, asset_ids: bytes, account_power_index: bytes):
  ...

  # update vote amounts
  option_vote_amount = tmp_vote_percentage * (account_voting_power / 100)

  ...
```

Because the voting power is first divided by 100 and then multiplied by a number that is at most 100, it is safe from integer overflows. However, dividing first and then multiplying incurs precision loss.

Impact

To quantify this precision loss we look at the range of `account_voting_power`. The *maximum* possible value of the account voting power occurs right when a single governor holds all TINY tokens and locks them for the maximum possible duration (four years). In this case, the maximum account voting power is 10^{15}. The smallest possible account voting power occurs when a user locks the smallest possible number of TINY tokens and the lock is about to expire. Locks expire always at midnight (UTC time zone) and a voting period always starts and ends at midnight, too. Hence the smallest possible non-zero voting power occurs when the lock expires the midnight after the proposal's voting period opens (thus, the smallest time difference is one day). The smallest possible number of TINY tokens that can be locked is 10^7 base units (i.e., 10 tokens). Using the formula *power = amount * time_remaining / max_lock_time* it can be computed that the smallest possible non-zero account voting power in this function is 6849. With this as the smallest possible voting power, the precision loss by dividing first and then multiplying is less than 1% and hence negligible.

Since the maximum value of `tmp_vote_percentage` is 100 and the maximum value of `account_voting_power` is 10^{15}, it is safe to multiply these two quantities as the result of the multiplication is less than 2^{64}. Therefore, it is recommended that the multiplication is executed before the division.

Response

This comment was addressed by the Tinyman team in commit: ac45c588388512acac8b1fa6ed53ffdd542f0948

Appendix

The findings reference a severity and difficulty classification which has been defined by Trail of Bits.

Severity Classifications	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Classifications	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.