Audit Report
Marble NFT contract

Summary and Scope

55	3	16	1
Notes	Warnings	Bugs	Vulnerabilities

This audit covers stripped down version of Marble NFT contract (initially targeted for Telos T-Bonds sale), consisting of:

- Marble Core (the core NFT standard)
- Bonds Layer (Marble plug-in, part of the official distribution)
- Events Layer (Marble plug-in, part of the official distribution)
- Tags Layer (Marble plug-in, part of the official distribution)
- Wallets Layer (Marble plug-in, part of the official distribution)

The audited version of Marble was AreaX's marble-digital-items repository (HEAD `ebedff7cd61dcb4e08c466ccba9a69527944f234`) which is forked from Dappetizer, LLC's upstream marble-digital-items (HEAD `96d3fa07d95699044b3485335709395069f42694`) with three additional commits (1, 2, 3), last commit removing unnecessary layers (Marble plug-ins). When built with CDT 1.7, the WASM checksum is `b81b483964486d6d058d0335875d9d06b291d6fdbf8f8c1f3c88f5c74b9433a7`.

Marble non-fungible token implementation is generally for EOSIO networks, although the contract is currently hardcoded to work only on Telos network.

Although multiple issues of varying degree were found, for use cases where the tokens are created all-at-once (such as Telos T-Bonds), mitigations not involving code changes can be taken. See "Mitigations" chapter for more information. For other use cases this particular version of Marble is not suitable for production.

The goal of an audit is to ensure that all the parties are safe and secure, in accordance with the auditor's best skills and knowledge. This does not mean that the contract is bug-free: smart

contracts in general are still an experimental piece of technology, and should be used with caution.

Design

Marble is well designed, and consists of two parts:

- **Core**: the core standard is implemented here, and every Marble implementation must implement the Core in order to be a valid Marble implementation.
- **Layers**: these are Marble's plug-ins, all of the layers are optional, and not required for a valid implementation. Unfortunately enabling/disabling layers is not in the build system, and manual removal may be difficult and prone to errors. For safety reasons (in order to minimize the code footprint) all layers should be disabled, and only explicitly enabled. Marbe upstream bundles some layers with the distribution, which are:
    - Attributes (not present in this audited version)
    - Bonds
    - Events
    - Frames (not present in this audited version)
    - Tags
    - Wallets

Despite being well designed, there is some room for improvement:

- Modular design is a good idea, however, in order to disable layers, one must manually remove files, and make modifications to some of the files. A build system should be implemented, by which layers can be enabled and disabled at will. All the layers should also be disabled by default, and enabled explicitly (to make code footprint smaller and safer). Currently some of the layers can't be completely removed, like "Bonds", since Core code makes an assumption that the Bonds header is always present.
- The contract implements its own admin permission system, which is a bad practice: EOSIO has a comprehensive permission management logic which should be used for contract wide administrative actions instead of creating your own logic. EOSIO accounts can contain roles invoking certain actions only, this way making possible both: resigning and having permissions for administrative actions. This would greatly simplify the logic for system wide administrative tasks.
- All critical features of a non-fungible token can be *locked* for immunity. This is vital for the "trust by immutability" principle of smart contracts. However, these safeguards can be easily circumvented: removal of these features does not contain any checks for immutability. So the locked feature can be removed, and re-added with a different value,

nullifying the whole concept. If all the tokens are created all-at-once, this can be remedied by resigning all administrative roles to the “eosio” account. See “Mitigations” chapter for more information.

- RAM is paid by the contract (initiated by the group manager, or the contract admin), this is a bad practice: a rogue manager could constantly fill the contract's RAM, impairing normal operation of the contract.
- Deposit/withdrawal pattern for fungible tokens is good: this avoids the *rollback attack* where an attacker can revert a transaction at their will.
- Supported fungible token should be any eosio.token compatible token and symbol, not only the hardcoded core token and core symbol.
- Behavior concept is good one, however, enabling and disabling a behavior should be explicit (without toggling), as should be locking: otherwise the group manager could accidentally be off-sync with the chain state, and change or lock a behavior to an undesired state.
- Although documentation is of high quality, easy to read and mostly up to date, Ricardian contracts should be amended, preferably with the content of the documentation.

Implementation

Generally, the code is of high quality, easy to read and understand, and consistent coding style is used throughout the codebase.

The Positive

- Behavior is a good design choice: this gives a lot of flexibility.
- Deposit/withdrawal pattern for tokens is good, see chapter “Design” for more details.

The Negative

- Const should be used with function arguments, like in system contracts: this would signal how the variable should be used.
- Math is relying too much on success assumption: integer overflows/undeflows should be handled. This seems to cause no acute problems at the moment, though. Always use checked datatypes.
- Assets should be properly verified, and for example “`.is_valid()`“ should be used.
- Don’t comment out code: delete. Git has the history if needed.
- There could be more informative actions propagated to accounts (not to jeopardize the contract operation creating *rollback attacks*, though).
- Using “`memo`” is inconsistent.

- Instead of “shared” features, features should be assigned to either items or groups, and two set of functions should be provided. This would simplify the code alot, and would make the “shared” concept easier to understand.
- The configuration table should be loaded as a singleton, and should be populated with hard coded defaults if not present (EOSIO system contract style). Although their pattern cost a few extra CPU cycles, block.one seems to prefer clean code over CPU performance.
- Tags do not need checksumming: after all, checksum is anyway changed when updating the data.
- For some reason Bonds use item specific events, instead of group (“shared”) events. Current one event per item convention increases needless redundancy.

The Neutral

- EOSLIB_SERIALIZE used in headers is not mandatory, but can be used: EOSIO system contracts use it to shorten compilation time.
- Additional features (such as tags) can exist in RAM after the item has been removed. Otherwise removal would get too complicated in on-chain code. This is something to take into account when designing Marble based software.

Vulnerabilities (1)

Vulnerabilities are bugs which have serious consequences, such as loss of capital. Only one vulnerability was found.

items.cpp:120	Despite all the safeguards, the group manager can remove the `reclaim` behavior for the group of the token, and then set it again to be “`true`”, and confiscate the tokens, and then call `::destroyitem()` to claim the bonded TLOS immediately (if “`destroy`” is also locked to be “`false`”, the group manager can remove it and set it again to be “`true`”).

Bugs (16)

Bugs are programming errors which are not serious enough to cause serious consequences, such as loss of capital, but can be problematic in other ways. Multiple bugs were found.

behaviors.cpp:21	RAM should be paid by the group manager, not the contract: this gives a rogue manager a way to fill up contract RAM indefinitely (using the u64 space), and impair the whole contract, instead of just their own group.
behaviors.cpp:85	Locked behaviors shouldn't be able to be removed. This nullifies the whole concept of locking, since the group manager can just remove and re-add the behavior again. And by removing a locked behavior, a rogue manager could impair their own group (by removing the "`transfer`" behavior for instance, which would prevent any tokens from being transferred).
groups.cpp:73	Since the contract is paying for the RAM, a rogue group manager could use these two strings to fill up the RAM.
items.cpp:45	Contract is paying for RAM instead of the group manager, giving rogue group managers a way to fill up the RAM, and obstruct operation of the contract.
bonds.cpp:54	The group manager should pay for the RAM, not the contract. However at the moment this is not a security concern, since it's more likely that the group admin would just fill the ram with items or tags.
bonds.cpp:124	Release events can be removed (despite being "locked") by the group manager. If this is done, releasing the NFT will fail, locking the funds until the manager re-adds the release event. In case of rogue group managers, they would most probably use the confistication vulnerability instead, so we classify this as a "bug" instead of "vulnerability".
events.cpp:31	Group manager should pay for the RAM, not the contract. This provides a rogue manager a way to fill up the contract's RAM.
events.cpp:46	Group manager should pay for the RAM, not the contract. This provides a rogue manager a way to fill up the contract's RAM.
events.cpp:150	Locked shared events should not be removed. Removing an pending event could lock bonds (until the event is added again).
events.cpp:157	Locked item events should not be removed. Removing an pending event could lock bonds (until the event is added again).
tags.cpp:34	Group manager should pay for the RAM, not the contract. This provides a rogue manager a way to fill up the contract's RAM.
tags.cpp:51	Group manager should pay for the RAM, not the contract. This provides a rogue manager a way to fill up the contract's RAM.
tags.cpp:95	Group manager should pay for the RAM, not the contract. This provides a rogue manager a way to fill up the contract's RAM.

tags.cpp:109	Group manager should pay for the RAM, not the contract. This provides a rogue manager a way to fill up the contract's RAM.
tags.cpp:174	Locked group tags should not be removed.
tags.cpp:181	Locked item tags should not be removed.

Warnings (3)

Warnings are features which work in a way that could differ from its original purpose, but are not bugs. These might have unintended consequences.

bonds.cpp:142	If a rogue group manager (this group, or any other group) fills the RAM, direct claiming of the bond is not possible if a wallet needs to be created. Instead the claimer needs to craft a transaction where first RAM is bought for the contract, and then claim is called.
bonds.cpp:174	If a rogue group manager (this group, or any other group) fills the RAM, direct claiming of the bond is not possible if a wallet needs to be created. Instead the claimer needs to craft a transaction where first RAM is bought for the contract, and then claim is called.
wallets.cpp:48	More checks needed: in the future when more tokens will be supported, our contract can't trust anymore that the token contract stays trusted. Verify positive amount, and preferably use is_valid().

Notes (55)

Notes are not problems per se, but instead points to pay attention to.

marble.cpp:1	`#including` cpp source files is not preferred, should be handled only by build logic (and utilizing `#defines` for conditional building).
behaviors.cpp:28	Setting a behavior should be explicit (without toggling), as should be locking: otherwise the group manager could accidentally be off-sync with the chain state, and change or lock a behavior to an undesired state.

behaviors.cpp:50	Setting a behavior should be explicit (without toggling), as should be locking: otherwise the group manager could accidentally be off-sync with the chain state, and change or lock a behavior to an undesired state.
groups.cpp:3	For consistency, the argument list should have the group name first.
groups.cpp:17	Validation not needed, because group name is the primary key, failing emplacing if trying to duplicate.
groups.cpp:25	In documentation “`description`” is named “`subtitle`”.
groups.cpp:34	Maybe this should be moved somewhere else, such as the header?
groups.cpp:44	Coding style inconsistent regarding the “`for`” block.
groups.cpp:46	This line should be before “`for`”
groups.cpp:79	Is “`memo`” really needed?
items.cpp:10	Only group manager should mint: ideally group manager could be separated from contract operator in case of contract having hundreds of groups. At the moment the contract operator can mint too. This is an undocumented feature, and seems to differ from the original design.
items.cpp:28	Why is current time needed? The action/transaction has the current time
items.cpp:58	Why do we have a separate logging call? Actions can be logged as-is.
items.cpp:30	This is not needed: redundant information.
items.cpp:33	Should be “`=new_serial`”, instead of “`+1`”.
items.cpp:36	Checking mechanism not needed, since “`serial`” is the primary key: emplace with a duplicate key would fail anyway.

items.cpp:146	Should share the item deletion code with `::destroyitem()`
items.cpp:161	Is not clear enough that consuming means item deletion: if "`destroy`" behavior is disabled, but "`consume`" behavior is enabled, the item will be still destroyed. The previous note would fix this too.
items.cpp:172	Bond layer is not fully independent from the code, since this function from Core is depending on it.
items.cpp:216	Bond layer is not fully independent from the code, since this function from Core is depending on it.
items.cpp:190	Why `::destroyitem()` has memo, but `::activateitem()` does not?
bonds.cpp:3	Creating a bond and adding to the bond should be separate actions. This would simplify the code. At the moment `newbond()` and `addtobond()` contain a lot of duplicate code.
bonds.cpp:36	Validation not needed, since the asset symbol is already the primary key, emplacing a duplicate would fail.
bonds.cpp:43	Technically events can be created with `name(0)` name. This might lead to misunderstandings, since event `==` `name(0)` is not allowed here.
bonds.cpp:103	Probably should be renamed "`releasebond`" for consistency.
bonds.cpp:151	Note: this function should also be renamed: `release()` is already releasing the whole bond. This could be named to "`forcerelease`".
events.cpp:3	There is no need for `custom_event_time` to be optional: 0 could be used to refer any time.
events.cpp:18	Why do we default to the current time? 0 would do the same.

events.cpp:27	Validation not needed: `event_name` is already the primary key, failing emplacing if trying to duplicate.
events.cpp:31	Validation not needed: `event_name` is already the primary key, failing emplacing if trying to duplicate.
events.cpp:115	Add “already” to the error message to make it clearer, and consistent with the other message.
events.cpp:136	Why is “group” taken as a separate argument here, but not in other functions?
events.cpp:163	A separate logging apparatus is not needed: individual actions can be logged as-is.
events.cpp:163	This seems to be in the wrong place: despite its name, this function has nothing to do with these kinds of events.
tags.cpp:17	Why the tag name can’t be empty, but all other names can?
tags.cpp:29	Validation not needed: tag name is already the primary key, failing emplacing if trying to duplicate.
tags.cpp:46	Validation not needed: tag name is already the primary key, failing emplacing if trying to duplicate.
tags.cpp:77	This commented-out code is not needed anymore, duplicates above two-liner. Delete.
tags.cpp:158	Why is the group an argument here, but not in other functions? Why is a memo needed?
wallets.cpp:18	Be extra sure of the user supplied amount, and use “`is_valid()`”.
wallets.cpp:32	If the contract's RAM is depleted, and the transaction target is a non-existent wallet, the transfer will fail. Can be fixed by the user opening a token wallet for themselves.
wallets.cpp:48	EOSIO header wrapper already does this check here.

wallets.cpp:64	Contract should not pay for the RAM, the user should "`open`" a wallet with this contract instead.
marble.hpp:32	Only `CORE_SYMBOL` is used, other consts are not.
behaviors.hpp:26	Contract pays for the RAM, not the manager (although the manager should pay!)
groups.hpp:22	Documentation not up to date: labels this as "`subtitle`".
items.hpp:52	Pagination issue seems to be a client's problem, and should not be solved here: the previous implementation was idiomatic EOSIO, and should be preserved.
bonds.hpp:34	Contract pays for the RAM, not the manager (although the manager should pay!)
events.hpp:26	One struct is enough for both, per item, and shared data. No need for two identical structs.
events.hpp:30	Contract pays for the RAM, not the manager (although the manager should pay!)
events.hpp:44	Contract pays for the RAM, not the manager (although the manager should pay!)
tags.hpp:22	One struct is enough for both, per item, and shared data. No need for two identical structs.
tags.hpp:26	Contract pays for the RAM, not the manager (although the manager should pay!)
tags.hpp:42	Contract pays for the RAM, not the manager (although the manager should pay!)
wallets.hpp:30	Contract pays for the RAM, not the owner.

Mitigations

For use cases where tokens are created all at once, the following mitigations can be taken after all the tokens have been created, without any changes to the smart contract code:

- The token contract must be resigned before launch.
- Admin should be the team multisig before launch (so new groups can be created when needed). If new groups are not needed, the admin should be “eosio”.
- Group manager should be "eosio", and tokens should be created into groups semi-atomically, and the NFTs should not be sold before manager of the group in question is "eosio".

For mitigating the RAM issues, one should:

- Set contract admin (and admins for all the groups) to “eosio”.

If a rogue group admin fills up the RAM, the user can craft a special transaction to buy RAM before each operation. These kinds of special transactions need to be manually crafted, and require EOSIO expertise.

About the Author

Ville Sundell has been involved with smart contracts since Bitcoin’s “Script”, stumbled upon Ethereum in late 2015 and published his first Solidity smart contract in summer 2016. Since his first audit in 2017, he has worked full time with smart contracts: developing and auditing for Ethereum and EOSIO based platforms. He is also a co-founder of CRYPTOSUVI, a block producer on Telos.