Beam Upgradable Dao Security Analysis

by Pessimistic

This report is private

October 25, 2021

Abstract

In this report, we consider the security of the code base of Beam project. Our task is to find and describe security issues in the code base of the platform.

Disclaimer

The audit does not give any warranties on the security of the code. One audit cannot be considered enough. We always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of the code. Besides, security audit is not an investment advice.

Summary

In this report, we considered the security of Beam project code base. We performed our audit according to the procedure described below.

The audit did not discover discrepancies between project's requirements and implementation. The code is still under development, the audited commit is not part of established release strategy. The contracts are not separate from Beam project, which should not be a long-term solution.

The contracts do not build as C++ projects, therefore one cannot debug the code nor run static analysis tools for the whole project. One can compile the contracts to WASM bytecode though.

The code is not properly documented. However, it mostly follows common solutions. The code calls BVM functions, some of which are not documented, e.g., `get_SlotImage()` or `get_BlindSk()`.

The project's architecture is adequate to the solved problem. The codebase is split into several modules that are well separated. However, these modules are not reusable as is. Additionally, code style does not follow best practices, which complicates understanding of the code. This might become an issue if new members join the project or other teams decide to build on top of this codebase.

Project overview

Project description

For the audit, we were provided with Beam project on a public GitHub repository, commit 7fb3a2c2206c5c62d502446f8ca5954ec8ea3e37.

The scope of the audit includes **bvm/Shaders/dao-core** and **bvm/Shaders/upgradable2** folders.

Procedure

We perform the audit according to the following procedure:

- Automated analysis
  - We compile contracts and deploy them locally.
  - We run provided tests in emulated environment.
  - We run Valgrind on the contracts in emulated environment.
  - We manually verify (reject or confirm) all the issues found by PVS-Studio.
- Manual audit
  - We manually review the code and assess its quality.
  - We check the code for known vulnerabilities.
  - We check whether the code logic complies with provided documentation.
  - We suggest possible charge optimizations.
  - We check whether interacting with the contract might degrade user's privacy.
- Report
  - We reflect all the gathered information in the report.

Issues

We are actively looking for:

- Access control issues (incorrect admin or users identification/authorization).
- Lost/stolen assets (assets being stuck on the contract or sent to nowhere or to a wrong person).
- DoS due to logical issues (deadlock, state machine error, etc).
- DoS due to technical issues (charge, other limitations).
- Blockchain-related (cross-transaction) issues (replay, reorder, race condition)
- Contract interaction issues (reentrancy, insecure calls).
- Arithmetic issues (overflow, underflow, rounding issues).
- Incorrect `ENV::X` usage.
- Other issues.

Automated analysis

We analyze the code by PVS-Studio and Clang Tidy. Since the code does not compile as a C++ project, output of other tools is unavailable.

The tools have correctly identified four types of issues:

- Class members are left uninitialized.
- Constructors with one parameter are not defined as `explicit`.
- Iteration over fixed-size arrays does not utilize "range loop".
- Static members are accessed via class instances

All discovered issues are of low severity, i.e., they do not affect the security of current implementation, but might lead to bugs or vulnerabilities in case of code modifications.

We manually verify (reject or confirm) all issues reported by the tools. All confirmed occurrences are listed in the Other issues section.

Manual analysis

Code logic

In **beam/bvm/Shaders/uprgadable2/contract.cpp** file, `void TestNumApprovers() const` method at line 76 does not check whether proposed quorum exceeds the number of multisig members. As a result, it is possible to set the required number of votes (`m_ApproveMask` value) to a greater value than the number of participants and thus make multisig functionality unavailable.

Code quality and style

General recommendations and best coding practices for C/C++:

- Use range-based loops based on span array declaration.
- Make constructors for public classes (structures).
- All class members should be initialized at the moment of a class instantiation.
- Templates and defines are widely used in the source code. They should be well documented and explained.
- "Magic" constants should be explained and declared as a constant with appropriate names.
- Repeated text constants should be definitely declared once.
- Current state of source code makes the entry threshold for newcomers pretty high and provides the wide possibility to make avoidable mistakes.

Contract improvements

We suggest the following security and code quality improvements of the contracts:

- Failing fast is considered the most secure option for smart contracts. We recommend calling `Env::Halt()` rather than handling errors and unexpected situations.
- Return values are not checked on multiple occasions. E.g., `Env::SaveVar_T` and `Env::LoadVar_T` storage operations are not checked for possible incorrect return code.
- `#define`s are widely used in the project, which often leads to confusing code. E.g., the macro `ON_METHOD(manager, schedule_upgrade)` translates to `void On_manager_schedule_upgrade(const ContractID& cid, const ContractID& cidVersion, const Height& dh, int unused = 0)`, which is not evident. As a rule of thumb, we recommend optimizing smart contracts for readability, security, and robustness rather than for efficiency.
- Multiple structures are initialized field by field, which is error prone. Consider using constructor instead.
- `dao-core` app utilizes hardcoded key material ("upgr2-dao-core") to generate admin keys. One must change the key material for each deployment to preserve privacy, so user's interactions with different contracts cannot be tracked by third party.

Other issues

Detailed results of confirmed issues grouped by source files:

- beam/bvm/Shaders/dao_core/app.cpp
- beam/bvm/Shaders/dao_core/contract.cpp
- beam/bvm/Shaders/uprgadable2/contract.cpp
- beam/bvm/Shaders/uprgadable2/app_common_impl.h
- beam/bvm/Shaders/uprgadable2/contract.h

The issues are listed below in one of the following formats:

- Text description
- <line-number-in-the-source-file>: `initial-text` → `variant-proposed`
- <line-number-in-the-source-file> `initial-text` - <note>

beam/bvm/Shaders/dao_core/app.cpp

We recommend adding comments with final substitutions for `ON_METHOD(...)` macro at lines 102, 121, 128, 146, 187, 199, 211, 218, 249, 348, 371, 392, and 429.

Hardcoded constants are used. Consider defining each constant only once:

- Line 270: `Env::DocAddNum("total", fs.s_Emission);` - hardcoded constant `total` is also used in lines 284, 422.
- Line 271: `Env::DocAddNum("avail", valAvail);` - hardcoded constant `avail` is also used in line 424.
- Line 272: `Env::DocAddNum("received", valReceived);` - hardcoded constant `received` is also used in lines 285, 425.
- Line 356: `Env::DocAddNum("duation", fs.m_hTotal);` - hardcoded constant `duation` is also used in line 421. The name of the constant has a typo. Consider replacing `duation` with `duration`.

beam/bvm/Shaders/uprgadable2/contract.cpp

- `Env::LoadVar_T` is called inside `void Load()` without result check at line 66.
- `Env::SaveVar_T` is called inside `void Save()` without result check at line 71.

beam/bvm/Shaders/dao_core/contract.cpp

Consider using range-based loop at line 125:

```
for (uint32_t i = 0; i < _countof(s_pE); i++)
{
    const auto& e = s_pE[i];
    _POD_(puk.m_Pk) = e.m_Pk;

    pu.m_Total = g_Beam2Groth * (Amount) e.m_ValueBeams;

    pu.m_Vesting_h0 = h + nBlockPerMonth * e.m_Month_0;
    pu.m_Vesting_dh = nBlockPerMonth * e.m_Month_Delta;
```

→

```
for (const auto & i : s_pE)
{
    _POD_(puk.m_Pk) = i.m_Pk;

    pu.m_Total = g_Beam2Groth * (Amount) i.m_ValueBeams;

    pu.m_Vesting_h0 = h + nBlockPerMonth * i.m_Month_0;
    pu.m_Vesting_dh = nBlockPerMonth * i.m_Month_Delta;
```

beam/bvm/Shaders/uprgadable2/app_common_impl.h

`!!` is unnecessary in `return !!(1 & (msk >> i));` expression at line 362.

Consider using range-based loops at lines 33, 52, 107, 156, 251, 384, 430, 446, 472, and 487. See an example for beam/bvm/Shaders/dao_core/contract.cpp.

Static methods are accessed through instance at:

- Line 105: `uint32_t iFree = arg.m_Settings.s_AdminsMax;` → `uint32_t iFree = Upgradable2::Settings::s_AdminsMax;`
- Line 124: `if (iFree >= arg.m_Settings.s_AdminsMax)` → `if (iFree >= Upgradable2::Settings::s_AdminsMax)`
- Line 371: `if (iSender >= stg.s_AdminsMax)` → `if (iSender >= ManagerUpgadable2::Settings::s_AdminsMax)`
- Line 427: `Comm::Channel pPeers[stg.s_AdminsMax];` → `Comm::Channel pPeers[ManagerUpgadable2::Settings::s_AdminsMax];`
- Lines 251, 384, 430, 446, 472, and 487: `for (uint32_t iPeer = 0; iPeer < stg.s_AdminsMax; iPeer++)` → `for (uint32_t iPeer = 0; iPeer < ManagerUpgadable2::Settings::s_AdminsMax; iPeer++)`

Fields are not initialized:

- Line 201: `} m_VerInfo` → `} m_VerInfo{}`
- Line 205: `Upgradable2::State m_State;` → `Upgradable2::State m_State{};`
- Line 206: `uint32_t m_VerCurrent;` → `uint32_t m_VerCurrent{};`
- Line 207: `uint32_t m_VerNext;` → `uint32_t m_VerNext{};`
- Line 325: `Msg2 m_Msg2;` → `Msg2 m_Msg2{};`
- Line 335: `uint32_t m_iMethod;` → `uint32_t m_iMethod{};`
- Line 336: `Upgradable2::Control::Signed* m_pArg;` → `Upgradable2::Control::Signed* m_pArg{};`
- Line 337: `uint32_t m_nArg;` → `uint32_t m_nArg{};`
- Line 338: `const char* m_szComment;` → `const char* m_szComment{};`
- Line 339: `PubKey m_pPks[Upgradable2::Settings::s_AdminsMax];` → `PubKey m_pPks[Upgradable2::Settings::s_AdminsMax]{};`
- Line 340: `Secp_scalar_data m_pE[Upgradable2::Settings::s_AdminsMax];` → `Secp_scalar_data m_pE[Upgradable2::Settings::s_AdminsMax]{};`
- Line 341: `uint32_t m_nPks;` → `uint32_t m_nPks{};`
- Line 520: `Msg1 msg1;` → `Msg1 msg1{};`
- Line 536: `Msg3 msg3;` → `Msg3 msg3{};`
- Line 605: `Upgradable2::State s;` → `Upgradable2::State s{};`

Consider declaring function as as `[[nondiscard]]`:

- Line 31: `uint32_t FindAdmin(const PubKey& pk) const` → `[[nodiscard]] uint32_t FindAdmin(const PubKey& pk) const`
- Line 48: `bool TestValid() const` → `[[nodiscard]] bool TestValid() const`
- Line 192: `uint32_t Find(const ContractID& cid) const` → `[[nodiscard]] uint32_t Find(const ContractID& cid) const`

beam/bvm/Shaders/uprgadable2/contract.h

Constructor with a single argument should be declared as `explicit`:

- Line 52: `Base(uint8_t nType) :m_Type(nType) {}` → `explicit Base(uint8_t nType) :m_Type(nType) {}`
- Line 69: `Signed(uint8_t nType) :Base(nType) {}` → `explicit Signed(uint8_t nType) :Base(nType) {}`

Fields are not initialized:

- Line 70: `uint32_t m_ApproveMask;` → `uint32_t m_ApproveMask{};`
- Line 80: `Next m_Next;` → `Next m_Next{};`
- Line 89: `uint32_t m_iAdmin;` → `uint32_t m_iAdmin{};`
- Line 90: `PubKey m_Pk;` → `PubKey m_Pk{};`
- Line 99: `uint32_t m_NewVal;` → `uint32_t m_NewVal{};`

This analysis was performed by Pessimistic:

Sergey Grigoriev, Security Engineer
Evgeny Marchenko, Senior Security Engineer
Boris Nikashin, Analyst
Alexander Seleznev, Founder

October 25, 2021