Vesting security report

Prepared by Pruvendo at 07/30/22

Executive summary

The current report validates the system of smart contracts called “Vesting” (located at) and confirms no bugs that can lead to losing or freezing of any funds, as well as to its improper distribution were discovered. The claim above is based on mathematical proving of things rather than on manual inspection that minimizes the chance of missing the potential threats.

Thus, the verifier recommends the Vesting system of smart contracts for moving into production.

Any questions or comments regarding the present report are to be requested by email info@pruvendo.com .

Brief project description

The project is intended to set up a tool that distributes the predefined amount of EVERs[1] to the recipients splitting the distribution into a set of monthly uniform tranches. Each tranche can be initiated by any of so called claimants (supervisors of the deal) thus allowing to stop or suspend the payments in case all the claimants are in consensus that the recipient is violating the rules or expectations defined by the deal.

Methodology

Unlike many audit approaches where the smart contracts are validated by manual inspection Pruvendo uses the formal methods where verification is made by the mathematical methods (roughly speaking, the code correctness is proved as a theorem). Some basic details are described in Appendix I while further information can be provided by request.

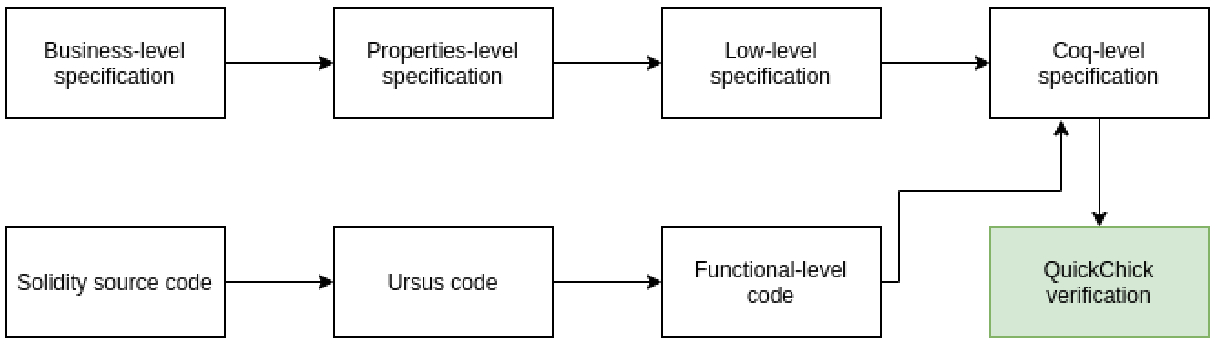
Briefly, the whole verification process can be splitted into the following phases:

- Specification - before any verification it's needed to state what it's planned to verify[2]. This part can be splitted into the following sections:
    - Business-level specification (high-level description intended for the end-users of the smart contract system)
    - Properties-level specification (description of all the properties of the system in a natural[3] language)
    - Intermediate-level specification (description of all the properties in a language understandable by software developers[4])
    - Low-level specification (description of all the properties in the form Coq[5]-predicates, intended for verifiers only). This activity is typically done when the next part - *Translation* - is completed
- Translation - conversion of the Solidity code into a set of functions whose properties can be proven. The process being described contains two stages:
    - Translation from Solidity into Coq-based DSL called *Ursus*[6]
    - Translation from the intermediate representation mentioned above into a set of functions
- Verification - the correctness of low-level specification towards the set of functions received above is conducted by QuickChick toolkit. This approach does not provide

[1] EVER is a native currency of Everscale blockchain
[2] It's important to mention that the statement "The program works correctly" is meaningless because "correctness" fully depends on context. The right statement is "The program works strictly according to the specification"
[3] For example, English
[4] Specification of this intermediate language called FeLiz currently is available by request only. However, it follows Java-like notations understandable by most software developers.
[5] The framework used for verification. Some basic information about it is available in Appendix I
[6] The specification is available by request

so full confidence as deductive software verification[7] but is considered as sufficient for the simple smart contract system being discussed.

Graphically the process can be described by the following diagram.



The verification process is described in the following sections.

Project structure

The verification project is located at github address and has the following structure.

File/Directory	Description
`README.md`	A copy of the executive summary of the present document
`dune-project`	Description file for the build system for OCaml[8] - dune
`/ref_2022_07_05_simple`	The source code of the contracts were verified
`/ref*`	The historical versions of the contracts
`/src`	The verification source code
`src/VestingPool/QuickChicks/*/Props.v`	Coq-level specification
`src/VestingPool/*.v`	Ursus code
`where`	Functional-level code

[7] If needed, the full deductive verification can be conducted upon a separate request
[8] OCaml - generic purpose functional programming language on which Coq proof assistant is based

Specification

Business-level description

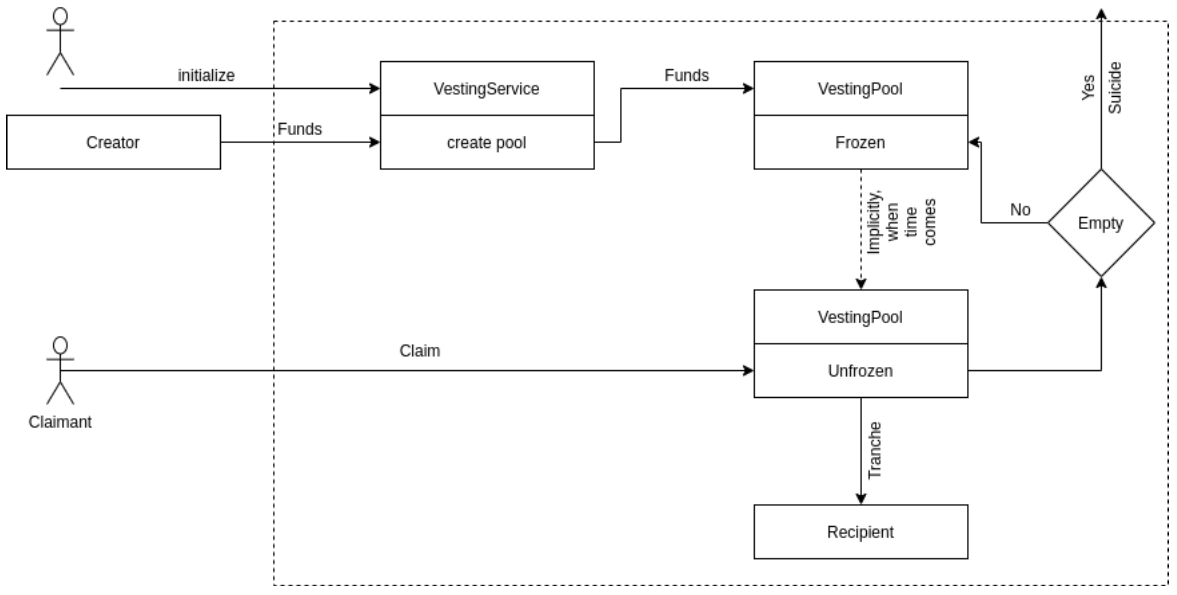
The set of Vesting contracts make a decentralized system for accepting a fund from one source and its distribution later to another one. Redistribution happens with a delay and is validated by the predefined set of so called claimants. The contracts are implemented as a distributed smart contract model.

Informally organization of the system can be explained as follows. There is a creator of the pool. The creator assigns a time moment in future, when the fund opens. It acts only once at the beginning. The crateror also acts as a donor to the pool, who allocats tokens into the pool, along with a list of claimants, who can address the fund. The claimants are identified by their addresses only. This is also a recipient of the fund. The donor does not move later. Further, the fund responds to requests from claimants and each period transfers equal amounts to the reciepient.

The pool is supported by the contract *VestingService*, which deploys the *VestingPool* contract inside. The *VestingPool* contract features: it has a delay in opening, and, when the initial delay is completed, is able to redistribute funds on a monthly basis to the recipient by request made by one of the claimants. When all the funds are distributed the VestingPool is suicided returned all the technical funds back to the fund creator.

The portions of unlocked funds are equal to each other.

Operations of the fund are presented at the illustration below.



System properties

Property code	Property description	Low-level property representation
GVS.1	New pool can be created by contract only	∀ params : params.sender.address = 0 → eval(createPool(params)) = err(100)
GVS.2	Anybody non-empty can be included into the client public key list	∀ params1, params2, claimers1, claimers2 : eval(createPool(params1)) = ok(void) → params1.claimers = claimers → params1 = params2[claimers = claimers1] → params2.claimers = claimers2 → (∃ claimer : claimer In claimers2) → (∀ claimer In claimers2 : claimer ≠ 0)→ claimer2.size ≤ MAX_CLAIMERS → eval(createPool(params2)) = ok(void)
GVS.3	At least one client must exists	∀ params: params.claimers.size = 0 → eval(createPool(params)) = err(ERR_NO_CLAIMERS)
GVS.4	No more than MAX_CLAIMERS can exist	∀ params: params.claimers.size > MAX_CLAIMERS → eval(createPool(params)) = err(ERR_TOO_MANY_CLAIMERS)
GVS.5	Anybody can be a recipient but one with	∀ params1, params2, r1, r2 :

	null address or from non-null shard	eval(createPool(params1)) = ok(void) → r1 = params1.recepient → params1 = params2 [recepient = r2] → params2.recepient = r2 → r2.address ≠ 0 → r2.wid = 0 → eval(createPool(params2)) = ok(void)
		∀ params : (params.recipient.address = 0 ⋁ params.recipient.wid ≠ 0)→ eval(createPool(params)) = err(ERR_INVALID_RECIPIENT)
GVS.6	The value of pool creation must cover the vesting amount as well as following fee : for pool creation, for each claim, for storage	∀ params : params.value < params.amount + CONSTRUCTOR_FEE + STORAGE_FEE + CREATE_FEE + params.vestingMonths * CLAIM_FEE → eval(createPool(params)) = err(ERR_LOW_FEE)
GVS.7	If all the input is correct a new VestingPool is created	∀ params, exit : params.recipient.address ≠ 0 → params.recipient.wid = 0 → params.claimers.size > 0 → params.claimers.size ≤ MAX_CLAIMERS → (∀ claimer : claimer In params.claimers → claimer ≠ 0)→ params.sender. = params.this.creator.address → params.value ≥ params.amount + CONSTRUCTOR_FEE + FEE_CREATE + STORAGE_FEE + params.vestingMonths *

		CLAIM_FEE → exit = exec(createPool(params)) → (eval(createPool(params)) = ok(Void) ⋀ exit.out.messages.size = 1 ⋀ (let pool = exit.out.messages.head.receiver in pool.message = VestingPool.constructor ⋀ pool.params.value ≥ params.amount + STORAGE_FEE + CONSTRUCTOR_FEE + params.vestingMonths * CLAIM_FEE)⋀ pool.params.amount = params.amount ⋀ pool.params.cliffMonths = params.cliffMonths ⋀ pool.params.vestingMonths = params.vestingMonths ⋀ pool.params.recipient = params.recipient ⋀ pool.params.claimers = params.claimers)
GVS.8	EveryAny pools created by the same the same Vesting service has have uniquedifferent addresses	∀ params1, params2, vestingService : params1 ≠ params2 → params1.this = params2.this → params1.messages.trunk = params2.messages.trunk → createPool(params1) In params1.messages.trunk → createPool(params2) In params2.messages.trunk → eval(createPool(params1)) = ok(Void)→

		`eval(createPool(params2)) = ok(Void)→ exec(createPool(params1)).out.message s.head.receiver ≠ exec(createPool(params2)).out.message s.head.receiver`
GVS.9	A list of clients can not be updated after the pool is created	`∀ f, params : f ∈ VestingPoolFunctions → params.this ∈ VestingPool → params.this.m_claimers = exec(f(params)).this.m_claimers`
GVS.10	A receiver of funds can not be updated after the pool is created.	`∀ f, params : f ∈ VestingPoolFunctions → params.this ∈ VestingPool → params.this.m_recepient = exec(f(params)).this.recipient`
GVS.11	Vesting pool parameters must be initialized by the constructor	`∀ params : params.value ≥ params.amount + STORAGE_FEE + CONSTRUCTOR_FEE + params.vestingMonths * CLAIM_FEE → params.sender = senderFromAddress(params) → (let exit = exec(constructor).params in exit.m_createdAt = params.now ⋀ exit.m_vestingMonths = params.vestingMonths ⋀ exit.m_recipient = params.recipient ⋀ exit.m_claimers = claimers ⋀ exit.m_cliffEnd = params.now + params.cliffMonths * 30 * 86400 ⋀ exit.m_vestingEnd = params.now +`

		(params.cliffMonths + params.vestingMonths) * 30 * 86400 ⋀ exit.m_totalAmount = params.amount ⋀ exit.m_remainingAmount = params.amount)
GVS.12	Funds are locked in the pool to return until the deadline of the open window.	∀ f, params, params2 : f ∈ VestingPoolFunctions → params.this ∈ VestingPool → params.now < params.this.m_cliffEnd → params2 = exec(f(params)).params → (params.this.balance >= params.this.amount ⋀ params.this.amount = params.this.remainingAmount)
GVS.13	To prevent incorrect usage of the pool it can transfer only once per time-slot after the first call. The rest of calls to transfer within a time are rejected.	∀ params1, params2 : params1.this.address = params2.this.address → params1.messages.trunk = params2.messages.trunk → claim(params1) In params1.messages.trunk → claim(params2) In params1.messages.trunk → params1.now < params2.now → (params1.now - params1.this.m_vestingFrom) / VESTING_PERIOD = (params1.now - params1.this.m_vestingFrom) / VESTING_PERIOD → eval(claim(params2))

		`= err(100)`
GVS.14	Only claimers can claim	`∀ params : params.sender NotIn params.this.m_claimers → ∃ code : eval(claim(params)) = err(code)`
GVS.15	Any attempts to claim during the cliff period must lead to exception	`∀ params : params.now < params.this.m_cliffEnd → ∃ code : eval(claim(params)) = err(code)`
GVS.16	Any attempt to claim after vestingEnd must lead to sending the rest of the amount to the recipient and the change back to the creator. Also the pool must suicide itself.	`∀ params : params.now ≥ params.this.m_vestingEnd → params.sender In params.this.m_claimers → (eval(claim(params)) = ok(Void)) ⋀ (let exit = exec(claim(params)) in exit.params. m_remainingAmount = 0 ⋀ exit.out.messages.size = 2 ⋀ exit.out.messages[0].receiver = params.this.m_recipient ⋀ exit.out.messages[1].receiver = params.this.m_creator ⋀ exit.out.messages[0].method = transfer ⋀ exit.out.messages[1].method = transfer ⋀ exit.out.messages[0].value = params.m_remainingAmount ⋀ exit.out.messages[1].value = exit.this.balance - params.m_remainingAmount ⋀ exit.this.suicide)`

GVS.17	The remaining amount for each successful claim is decreased by the transfer amount to the recipient	∀ params : eval(claim(params)) = ok(Void) → (let exit = exec(claim(params)).out.messages in exit.size > 0 ⋀ exit[0].method = transfer ⋀ exit[0].receiver = params.m_recipient ⋀ exit[0].value = params.m_remainingValue - exec(claim(params)).this.m_remainingValue)
GVS.18	If the current time is after cliff period and before the vesting end (that effectively means that vesting period is more than zero) the amount to vest is calculated by the following formula: $min(remainingValue,\ max(0,\ \frac{\left\lfloor \frac{now - cliffEnd}{2592000} \right\rfloor}{vestingMonths} amount)$	∀ params : eval(claim(params)) = ok(Void)→ params.this.m_remainingValue - exec(claim(params)).params.this.m_remainingValue = min (params.m_remainingValue, max(0, div((params.now - params.this.m_cliffEnd) / 2592000) / params.this.m_vestingMonths * params.this.m_amount))

Coq-level specification

With low-level properties being defined in the previous section the next level was to translate them into native *QuickChick* statements. This activity has been after completion of translation and the results are available at `src/VestingPool/QuickChicks/*/Props.v`.

Translation and Verification

Translation

At the first stage the Solidity source code was transformed into Ursus representation. The resulting Ursus files are located at `src/VestingPool/*.v`. This activity was made by the proprietary fully-automated translator from *Solidity* to *Ursus* developed by Pruvendo.

Then the conversion from *Ursus* into functional-level code was performed by the semi-automated[9] *Generator* tool also developed by Pruvendo. The results are available here.

Verification

As a result of the previous stages the following stuff was available:

- The list of the required properties in a Coq-friendly representation
- The code to be verified against these properties (in a Coq-friendly manner as well)

Then, the two options were considered:

- Perform the manual full-scale mathematically strict (deductive) verification
- Perform the lighter version of the verification using QuickChick[10] tool

It was decided that for this simple contract system that does not have a non-trivial logic the usage of the former approach is a definite overkill (deductive verification is very expensive and time-consuming), so the QuickChick approach was chosen.

The corresponding environment has been created and the tool was successfully executed.

Outcome

As a result of verification it was proven the implementation, indeed, has all the declared properties, that means the overall outcome of the formal verification **IS POSITIVE**.

[9] It's planned to make it fully automated in the nearest future
[10] Some basic information about QuickChick can be found in Appendix I

Appendix I. Behind the scene

The present appendix provides some more information about the verification process. It may be rather difficult to read and require some advanced skills and knowledge to understand it. The authors, however, tried to be as simple as possible, avoiding hard stuff.

If you, by another hand, would like to know more, feel free to contact us using the means of communications provided on the top of the present document.

In mathematics, there are such software products as Proof Assistants, one of them is Coq, that are able to automatically check if a theorem was proved correctly or not (and also provide some assistance with proving). Surprisingly, according to Curry-Howard correspondence, the software programs are isomorphic to the mathematical proofs, which gives an opportunity to use Proof Assistants for the proving of software.

However, the Curry-Howard correspondence considers the computer program as one written in a typed declarative programming language. Even more, all the programs must halt at some point. At the same time the Everscale smart contract developers use typed imperative programming languages - such as *Solidity* or *C++*, that are Turing-complete that means that, generally speaking, it's not possible to check if they halt at some moment or not[11].

So the first goal of the verifiers was to translate the imperative Turing-complete code into the functional code that always terminates.

To achieve this goal the *Ursus* language has been invented. It's an imperative language with syntax very close to *Everscale Solidity*, but at the same time it's a DSL on top of declarative Coq-environment. Also the correct Ursus program always terminates, which means that some Solidity programs can not be translated into Ursus (or require some refactoring). Fortunately, it's an extremely rare case in the real world and usually means a poor design of original smart contracts.

While the *Ursus* program can be already handled by Coq Proof Assistant, its imperative structure makes this activity extremely difficult so one more big step is required: convert imperative *Ursus* DSL code into a set of functions.

Each original function is converted into two:

- *eval* - that represents the return value of the function
- *exec* - that represents the state of the machine after the execution of the function

Both functions must use the current state of the machine as one of the input parameters.

[11] Due to the halting problem

When this task is completed the verification becomes much easier. The program becomes a set of functions that can be used to define the properties to be verified (this kind of definition was referenced as Coq-level specification throughout the present document). And nothing prevents us any more from proving these properties using the powerful Coq Proof Assistant capabilities.

Just one important notice. While the approach described above is fully valid and should be used in many cases, its serious drawback is high-cost of the proving as well as very high requirements for the qualification of executors.

While the systems of smart contracts with complicated business logic and/or implementation give no other option, for simple systems with straightforward logic the lighter approach can be used. The idea is to use *QuickChick* randomized property-based Coq plugin that verifies the properties not by strict mathematical proving but by providing random input data checking all the properties are in place. It’s worth mentioning that this approach, while inferior to deductive verification, is still much more powerful than traditional random testing, as predicates are challenged here rather than just numbers, making the whole process much more reliable.