The University of Melbourne

SWEN90010: High Integrity Systems Engineering

Assignment 2

Due Date: 11:59pm Melbourne Time, Wednesday April 27th, 2022

1 Introduction

This handout is the Assignment 2 sheet. The assignment is worth 20% of your total mark. You will carry out the assignment in the same pairs as for Assignment 1.

In this assignment you will use Alloy to diagnose and fix a vulnerability in a (fragment of) a smart contract. A smart contract is a (typically small) computer program that facilitates electronic financial transactions. While their history dates back to the 1990s [2], today smart contracts are widely deployed on various decentralised ledger platforms, of which the Ethereum blockchain is perhaps the most prominent. The vulnerability that you will analyse is inspired by the widely reported *recursive reentrancy* vulnerability of The DAO. The DAO was one of the first high profile vulnerabilities discovered and exploited in an Ethereum smart contract. Attackers used this vulnerability to steal around $50 million USD in Ether; the theft was only thwarted when the Ethereum community agreed to revert the state of the Ethereum blockchain to undo the transfers in which the funds were stolen. This vulnerability stands as a stark reminder of the risks associated with smart contracts, and the need to ensure they are free of vulnerabilities. It also spurred much interest in methods for detecting and proving the absence of vulnerabilities due to recursive invocation; however, formal methods for doing so had been studied at least 8 years prior [1].

1.1 The Scenario

We will consider a simplified version of The DAO, executing on top of a simplified smart contract platform. The real DAO and Ethereum platform are far more complicated. But the system you will analyse still contains all of the essential details to understand, diagnose and fix the vulnerability in question.

1.1.1 Smart Contract Platform

For the purposes of this assignment, the *smart contract platform* is a computational system in which small programs called *smart contracts* can execute. The smart contract platform is an object-based system. Specifically, smart contract programs are implemented as *objects* and are executed by *invoking* them (similarly to calling a method on an object in an object-oriented programming langugae like Java). At all times there is (at most) one *active object*, which is the object that has currently been invoked and is therefore currently executing.

1.1.2 Invocations and Call Stack

Objects can be invoked in one of two ways: firstly one object can *call* another to invoke it, passing some data as an argument if desired. This is similar to calling a method in a language

like Java. However, in the simplified platform we consider in this assignment, objects do not have multiple methods (i.e. each object effectively only has one method and calling the object invokes that method). The second way an object can be invoked is when the active object *returns*, which causes it to become inactive and its caller to become the active object. Returns operate similarly to languages like Java. There is a *call stack* which records the current sequence of calls. When active object A calls object B, a new *stack frame* is pushed on to the call stack. This frame records the fact that object A called object B and that B is now the active object. We refer to A as B's *caller*. We refer to B as the *callee*. When B is executing the invocation, we refer to A as the *sender*. When B returns from this invocation, we refer to A as the *returnee* (since it is to A that B is returning control). When B returns, that stack frame is popped from the stack and A becomes the active object again.

As in languages like Java, objects can be recursively invoked. For instance, object A can call itself.

1.1.3 Object Balances

As with other object-based programming languages, objects (smart contracts) may have internal state which can be modified by invoking them. Each object has a *primitive* (in-built) piece of state called its *balance*. The balance of an object is a (private) integer field that stores a non-negative value. This value represents how much currency is held by the smart contract that the object implements. An object cannot modify its balance directly; nor can it modify the balance of any other object directly. Instead, when one object A calls another B, A can include some of its currency with the invocation, which is atomically transferred from A's balance to B's balance (i.e. the amount is deducted from A's balance and is added to B's balance). The amount that A includes cannot exceed A's own balance.

1.1.4 The DAO

The DAO is an object (smart contract) that stores value (currency) on behalf of other objects. In particular, just like every other object, it has a balance, and it can transfer some of its balance to other objects by calling them. The DAO's job is to store currency on behalf of other objects. For instance, suppose it is storing 5 units of currency on behalf of object A, and 3 units of currency on behalf of object B. Then its balance would be at least 8.

We refer to the value stored by The DAO on behalf of some other object as that object's *credit*. The DAO maintains a mapping that records for each other object A the amount of credit that The DAO holds on behalf of A.

The only operation that the simplified DAO in this assignment supports is for objects to call it. When an object A calls the DAO, A is requesting to withdraw all of its credit, i.e. for all of the currency that The DAO is holding on behalf of A to be released and transferred to an object of A's choosing. Specifically, when A calls The DAO, the argument passed to the call indicates the object that should receive all of A's credit. When called by some object A passing some object name B as the argument, the DAO simply calls B, including with the call the amount of credit that The DAO is holding on behalf of A, thereby transferring that credit from The DAO to the object B that A chose to receive its credit. Once this call to B returns, The DAO then updates the mapping that records how much credit it is holding on behalf of object A to say that it is now storing 0 credit on behalf of A and returns. The update of The DAO's internal state (to

record that A has has 0 credit) and the return to its caller A are performed atomically, i.e. as part of the same action (transition).

1.2 The Formal Model

You are provided with a partial Alloy model of this system. Part of your tasks for this assignment is to complete this model using the description above and the following information.

1.2.1 Signatures

The Alloy model contains a signature to represent *Data*. Object names are a special kind of data. Therefore the signature *Object* extends *Data*. Each *Object* has a variable field that stores its *balance* as a single `Int`.[1]

The signature *Op* contains just two items: *Call* and *Return* that represent the two kind of invocations, described above.

The singleton *Invocation* signature is used to store information about the currently-executing invocation. It stores what kind of invocation it it (i.e. the operation, *Call* or *Return*). In addition, if the invocation is a *Call* it also records what argument (if any) was passed with the call.

The call stack is modelled as a singleton signature *Stack* that contains a variable field *callstack* that stores a sequence of *StackFrame*s. Recall that when a call is performed, a new stack frame is pushed on to the stack. Each *StackFrame* records the caller and callee for its call.

Alloy has an in-built type for (bounded) sequences. We use this type to model the stack. Sequences in Alloy are similar to lists in other programming languages. You can (and should) read more about them here: `http://alloytools.org/quickguide/seq.html`. The top of the stack is the *last* stack frame in the sequence. (The bottom is the first frame in the sequence.)

The DAO is a special, single object, modelled by the singleton signature *DAO*. Its internal state (besides its balance) records for every other object the amount of credit that The DAO is holding on behalf of that object. This is modelled by a variable field *credit* of type `(Object - DAO) -> one Int`.

1.2.2 Functions

The model is designed to make use of a number of utility functions. These are documented below. Some of these you will implement.

active_obj: When the stack is non-empty, this function gives the currently active object, i.e. the unique object that is executing the current invocation. When the stack is empty, its value is not defined, i.e. it returns the empty set.

sender: When the stack is non-empty, this function gives the object that called the currently active object. In particular, Suppose object A calls object B, and then, while executing

[1] We discussed `Int`s in Alloy in Live Lecture 10. You can also read more about them here: `https://alloy.readthedocs.io/en/latest/modules/integer.html`

A's call, B calls object C. Once the call to C returns, making B the active object again, A becomes the sender. When the stack is empty, this function is not defined, i.e. it returns the empty set.

returnee: This function is a synonym for **sender** above.

sum_DAO_credit: This function returns the sum of `DAO.credit[o]` across every non-DAO object `o`.

sum_Object_balances: This function returns the sum of `o.balance` across every object `o`.

1.2.3 `objs_unchanged` Predicate

The model makes use of a predicate `objects_unchanged[objs: set Object]` that takes a set `objs` of Objects as its argument. This predicate holds when the balances of all objects in the set `objs` are not changed between the current state and the next. Additionally, if `objs` contains the object `DAO`, then the amount of credit that the DAO holds on behalf of any other object cannot change between the current state and the next.

1.2.4 Operations of the Model

The model supports two primary operations: `call` and `return`. These are each implemented by separate predicates.

`call` The `call` predicate has type

```
call[dest: Object, arg: lone Data, amt: one Int]
```

and models the currently active object calling the object `dest`, passing the argument `arg` and transferring the amount `amt` from its balance to that of `dest`.

This operation can occur only when `amt` ≥ 0 and `amt` does not exceed the balance of the currently active object, and the sequence that models the stack has not exceeded its maximum length bound (i.e. adding a new stack frame will extend the stack's length).

In this case a new stack frame is pushed on to the stack, which records that the caller of the invocation is the active object in the state in which the call occurred, and the callee is the object being called, `dest`. The `Invocation` is updated to record the argument `arg` as the parameter being passed to the call, and to record that the current invocation is now a `Call`.

Additionally, the amount `amt` is deducted from the balance of the active object who made the call, and added to the balance of the object being called. The balances of all other objects are unchanged. If the active object who made the call is not The DAO, then no object's credit can change.

`return` The `return` predicate takes no arguments. It models when the currently active object returns to its caller.

It can occur only when the stack is non-empty. In this case the top stack frame is popped from the stack and the `Invocation` is updated to record that the invocation is now a `Return` with no paramter.

The balances of all objects remain unchanged. If the active object who did the return is not The DAO, then no object's credit can change.

`dao_withdraw_call` This predicate takes no arguments and models what happens in response to when some object `Call`s The DAO, i.e. it models what happens in the step *immediately after* The DAO is called.

`dao_withdraw_return` This predicate takes no arguments and models what happens in response to when some object *Return*s to The DAO. Suppose at some point in time The DAO calls some object A. Then this predicate models what happens in the step *immediately after* A returns to The DAO.

`untrusted_action` This predicate takes no arguments. It models the behaviour of objects besides The DAO. We refer to those objects as *untrusted* because they might be under the control of attackers who are trying to exploit The DAO (e.g. to steal currency).

It occurs only when the currently active object is not The DAO. Untrusted objects can perform one of two actions: either they can call an object, passing any argument and any amount that does not exceed their balance; or they can return to their caller.

`unchanged` This predicate models an execution step of the model in which no state changes occur.

1.2.5 Facts

`init` The model includs the fact `init` that specifies the system's initial state. The initial state has a call stack with a single stack frame whose caller and callee are not The DAO (i.e. the initial active object is not The DAO). Initially the balance of every object is between 0 and 2 inclusive, as is the amount of credit that The DAO holds on behalf of each other object.

Initially the invariant `DAO_inv` also holds. (You will implement this invariant as part of your tasks below.)

`trans` The model includes the fact `trans` that says that at all times in all states one of the following must occur: an untrusted action, `dao_withdraw_call`, `dao_withdraw_return`, or `unchanged`.

1.3 Your Tasks

Your first task is to complete the parts of the model that are missing. Doing so will require you to carefully understand the model you are given so you can work out how to fill in the missing parts. You will then use Alloy to diagnose a vulnerability in The DAO, as well as to implement and analyse a fix for the vulnerability.

*You should not add any additional **sig** declarations nor modify any of the existing **sig** declarations during this assignment.*

1. [**10 marks**] Complete the `call`, `return`, and `objects_unchanged` predicates to complete the initial model.

2. [**1 mark**] Implement the invariant (predicate) `DAO_inv` that holds when The DAO's balance is sufficient to cover all of the credit it is holding on behalf of other objects, i.e. when The DAO's balance is greater than or equal to the sum of the credit it holds on behalf of all other objects, and when the credit the DAO holds on behalf of each other object is non-negative.

3. [**5 marks**] You will find that the system can easily reach states in which this invariant does not hold *temporarily*. A good example is when object A calls The DAO to withdraw their credit. The DAO then calls object A to transfer its credit back to object A. At this point, before the call to A has returned, A's credit has been deducted from The DAO's balance but not (yet) from the amount of A's credit that The DAO records it is holding on behalf of A. Once the call to A returns, The DAO then deducts A's credit, to restore the invariant.

   However, The DAO does have a vulnerability in which an untrusted object can cause The DAO to transfer to it more currency than The DAO holds on behalf of that object. Doing so causes the `DAO_inv` invariant to fail in a way which represents a real vulnerability.

   Using the predicate `DAO_inv` or otherwise, write an Alloy assertion that characterises what it means for The DAO to remain secure, so that a violation of your assertion represents a real attack on The DAO. Use Alloy to discover the attack on The DAO using an appropriate **check** command.

   Document the attack that you find in comments in your Alloy file.

   Hint: By default, the maximum size of sequences (like the one used to model the stack) is quite small. You should specify the scope of your ***check*** *command to include a sufficiently long stack size. For example, writing "*`check blah for 7 seq`*" tells Alloy to search for counter-examples to the assertion "blah" using a maximum sequence length of 7.*

4. [**1 mark**] A simple fix to this vulnerability is to change The DAO so that when it is called by some object A, requesting A's credit be transferred to the balance of object B, it deducts A's credit at the same time that it calls B; rather than waiting to do so until B returns.

   Implement this fix by making a minimal change to the Alloy model. You should not need to add new predicates, signatures, etc. Your change should not prevent legitimate behaviour of The DAO.

   You should document the change you made by using appropriate comments in the Alloy file.

5. [**3 mark**] Use Alloy to confirm that your change indeed prevents the attack without ruling out legitimate behaviour. In particular, write **run** commands to confirm that The DAO still exhibits legitimate behaviour (i.e. allows objects to withdraw their credit, and have it transferred to other objects, etc.). In particular, your model should allow an object A to invoke The DAO asking its credit to be transferred to object A itself, thereby causing The DAO to (recursively) call object A. Write a **run** command that shows a complete execution in which an untrusted object withdraws its creit in this way (including the subsequent return steps, so that at the end of the execution the stack size is again 1).

Use Alloy to confirm that your assertion you wrote above now seems to hold.

Think carefully about the bounds needed for your **check** command. Document your choice of bounds and your reasons for choosing them as comments in your Alloy file. Document what you believe these checks and bounds imply about the fixed implementation in reality.

To obtain full marks here we want to see you choose a bound that is large enough to provide some guarantees but not larger than necessary to obtain those guarantees. You need to think carefully about what your bound means, i.e. what behaviours are covered by that bound and the degree to which showing the absence of the attack for all of those behaviours provides assurance about the fixed system in reality.

If you think that your chosen bound does not provide good guarantees you should say so and explain why.

2 Academic Misconduct

The University misconduct policy applies to this assignment. Students are encouraged to discuss the assignment topic, but all submitted work must represent the pair's understanding of the topic.

The subject staff take plagiarism very seriously. In the past, we have successfully prosecuted several students that have breached the university policy. Often this results in receiving 0 marks for the assessment, and in some cases, has resulted in failure of the subject.

3 Submission

Submit your Alloy file using the link on the subject LMS.

Only *one* student from the pair should submit the solutions, and each submission should clearly identify **both** authors.

Late submissions Late submissions will attract a penalty of 10% (2 marks) for every day that they are late. If you have a reason that you require an extension, email Toby *well before the due date* to discuss this.

Please note that having assignments due around the same date for other subjects is not sufficient grounds to grant an extension. It is the responsibility of individual students to ensure that, if they have a cluster of assignments due at the same time, they start some of them early to avoid a bottleneck around the due date.

References

[1] Toby Murray. *Analysing the security properties of object-capability patterns.* PhD thesis, University of Oxford, 2010.

[2] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 1997.